Design of a Vehicle Rental System

Team-Based Project

Autumn Semester 2024 - 2025

J.J. Collins

Group:

Shane Barden (24033944)

Ivor D Souza (24177431)

Manjeshwar Aniruddh Mallya (24124133)

David Parreño (24196223)

Rohan Sikder (24165816)

Master of Science in Software Engineering

CS5721 - Software Design

Contents

1. Project Description

1.1. Overview

VRS (Vehicle Rental System) is a robust vehicle rental management system, designed to streamline vehicle rental operations allowing up-and-coming vehicle rental startups and companies to provide their rental services quickly and reliably. It allows not only staff members to manage their rental services efficiently, but also customers to carry out the rental process in an easy and straightforward manner.

VRS will allow easy management of all vehicles, allowing staff to add vehicles to be available to be booked and giving customers various rental options based on their preferences such as make, model and rental period. Customers can customize their booking to their liking e.g. adding extras such as GPS and Premium insurance.

VRS will be able to integrate the payment service with the company's needs seamlessly. Customers will also be able to cancel reservations which the company can specify. Customers can return vehicles and integrate with systems the company may already have such as an AI Vision Damage checker, notifying the customer of any detected damage and letting them dispute any claims or further act. The system gives continuous feedback throughout the process, sending confirmation emails when the booking is confirmed when the vehicle is picked up and returned.

The system automatically sets up tickets for managers if there is any damage and allows them to assign a mechanic to fix the vehicle or general servicing. Managers can generate sales reports and statistics for the company and view feedback from customers. VRS is built to scale with businesses allowing for expanding vehicle fleets and a larger user base as VRS is a modular and scalable system.

Companies can request the addition of new features, which can be quickly and efficiently implemented into the system due to its modular and scalable design. This flexibility allows the system to adapt and expand according to specific business needs easily.

1.2. Case Tools

1. **Microsoft Teams:** collaborative online platform where users can communicate with each other through chat, video conferencing, and file sharing. MS Teams has been used primarily as a tool not only to organize online meetings but also to share the progress of each team member.
2. **Spring Boot:** open-source Java-based framework that simplifies the progress of creating and building stand-alone production-ready applications.
3. **Eclipse:** integrated development environment (IDE) used for developing, compiling and executing Java software on Windows, Linux, and macOS operating systems.
4. **GitHub:** cloud-based platform used for version control and collaboration built around the distributed

version control system Git. It allows developers to manage and store code, track changes and work on projects collaboratively.

5. **Postman:** Application Programming Interface (API) development platform used to design, test, and document APIS. It supports HTTP requests like GET, POST, PUT, and DELETE, and allows developers to create and manage collections of requests for API testing and collaboration across teams.

6. **Lucid Chart:** web-based diagramming tool used to create different types of diagrams such as flowcharts, organizational charts, network diagrams and other visual representations. For this project, it has been used as the main tool for creating all diagrams as it allows collaborative work.

2. Software Life-cycle

The Software Development Life Cycle (SDLC) is a systematic and structured approach that divides the software development process into stages, from initial conception to deployment and maintenance. Its main objective is to develop quality software in the most cost-efficient manner using an organized methodology.

Nowadays, there is a wide range of different methodologies that define the SDLC in various ways, such as Waterfall, Iterative, Agile, etc. This report explains some of these methodologies, and it also illustrates which of these methodologies were used for the VRS.

2.1. V-Model

The V-shaped model, also known as the Verification and Validation model, is an extension of the waterfall model with its defining feature being that each development phase has a corresponding testing phase, which on paper, forms a "V" shape. This model improves quality control which makes it suitable for projects where each phase's output can be verified against the requirements. The only concern with using this type of model is its inflexibility to change on completion of a phase. For this project, a more flexible approach would be ideal.

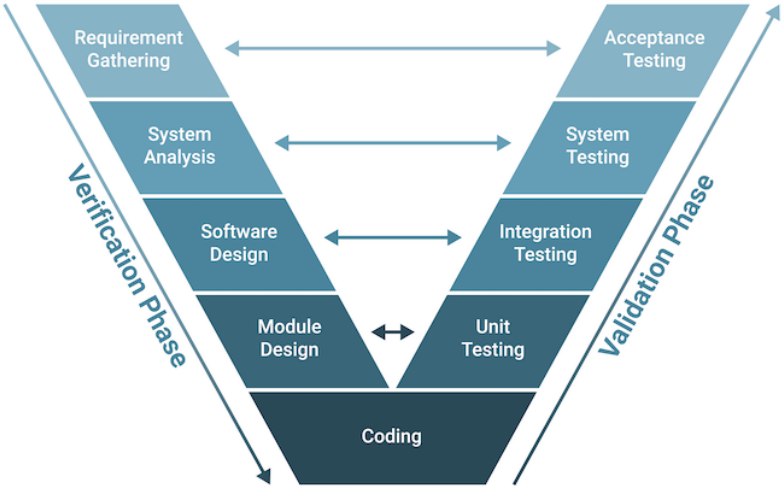


Figure 1: V-Model Software Development (Oppermann 2023)

2.2. Waterfall Model

The Waterfall model is a more traditional methodology which employs a linear sequential approach where each phase must be completed before proceeding to the next phase. This type of model is essential for projects with well-defined, clear requirements as each phase is fully documented and completed sequentially. However, compared to the V-shaped model, its lack of flexibility is a significant drawback as it means that once a phase is completed, it is difficult to revisit it to incorporate changes in requirements.

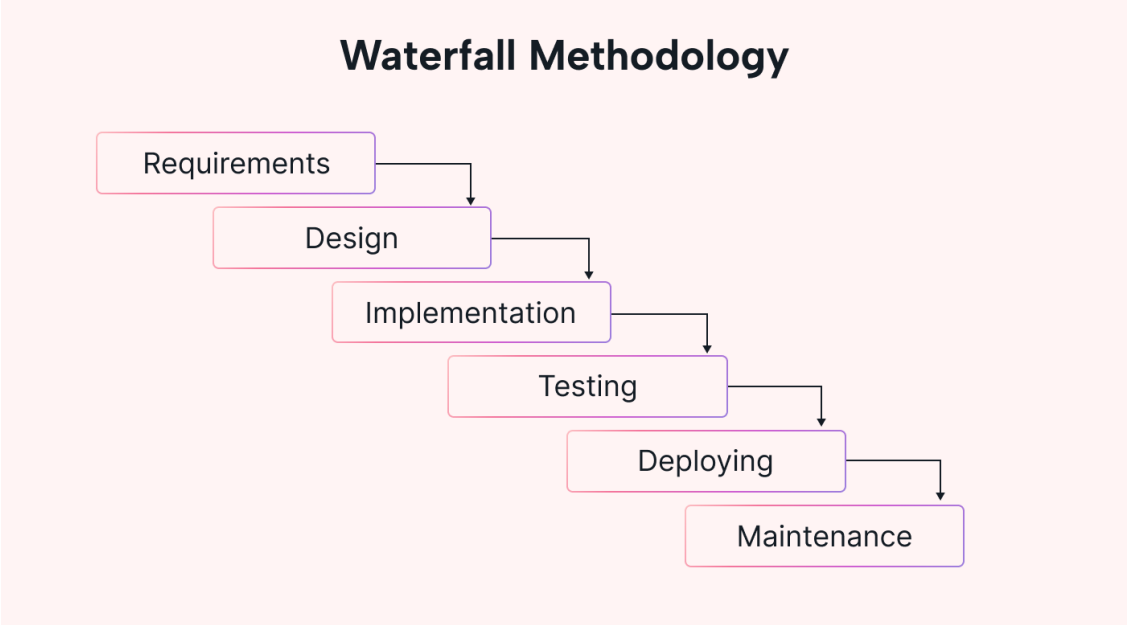


Figure 2: Waterfall Model Software Development (Motion 2023)

2.3. Agile Methodology

Agile is a modern, iterative approach to software development designed to be more lightweight and flexible compared to other approaches and allows for changes to design or requirements. It also ensures rapid delivery of functional software. It differs from the V-shape model and the waterfall methods as it centred around the idea of continuous development and frequent reassessment of design by using cycles or "sprints". Each sprint is used to deliver each piece of software which allows for quick feedback and adaptation to changes.

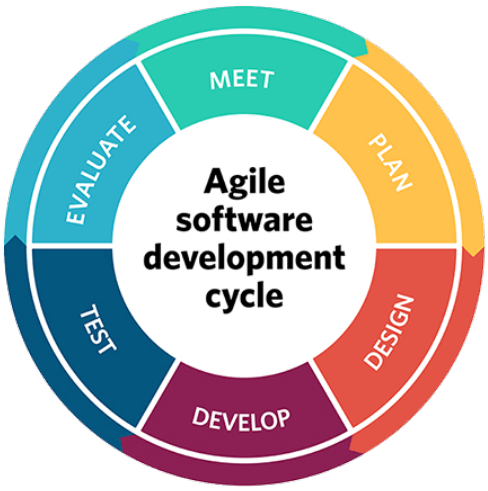


Figure 3: Agile Software Development (Michaud 2024)

2.4. Methodology employed

For this project, Agile's lightweight documentation, flexibility and adaptability make it the clear choice as it allows for breaking the project down into smaller tasks, helps to prioritize essential features, and addresses changes in requirements as soon as possible. The project's complexity can be managed more efficiently by focusing on iterative improvements and early testing. This also ensures that critical features are developed and tested early on, improving the ability to detect errors in the code or design.

3. Project Plan

3.1. Role Assignment

	Role	**Description**	**Designated Team Member**
1	Project Manager	Sets up group meetings gets agreement on the project plan, and tracks progress	Rohan
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	David
3	Business Analyst / Requirements Engineer	Responsible for section 4 - Requirements	David and Shane
4	Architect	Defines system architecture	Rohan and Ivor
5	Systems Analysts	Creates conceptual class model	ALL
6	Designer	Responsible for recovering design time blueprints from implementation	ALL
7	Technical Lead	Leads the implementation effort	Ivor
8	Programmers	Each team member to develop at least 1 package in the architecture	ALL
9	Tester	Coding of automated test cases	ALL
10	Dev Ops	Must ensure that each team member is competent with development infrastructure, e.g. GitHub, etc.	Manjeshwar

3.2. Project Plan

	Deliverable	Team Member	Due Week
	Role assignment	Team	3
	Business Scenario	Team	3
	Functional Requirement Description	David	4
	Use Case Diagrams	Team	5
	Use Case Descriptions	Team	6
	Non-functional Requirements	Shane	7
	Quality attributes	Shane	7
	GUI Prototypes	Rohan and Manjeshwar	7
	Architectural Patterns	Rohan and Ivor	7
	Class Diagrams	David, Shane and Manjeshwar	8
	Sequence Diagrams	Rohan and Ivor	8
	State Chart	David and Shane	8
	Entity Relationship Diagram	Manjeshwar	8

3.3. Industry Experience

Name	ID	Experience	Domain/Framework/Programming Languages
Shane Barden	24033944	Fresher	
Ivor D Souza	24177431	4 years	Haskell, R, Rust, Data Engineering, Dev Ops
Manjeshwar Aniruddh Mallya	24124133	2 years	Python, C Sharp, .net, SQL
David Parreño	24196223	1.5 years	Java, Python, Bash, ASP .NET MVC, Scrum
Rohan Sikder	24165816	Fresher	

4. Requirements

4.1. Functional Requirements

- **Customer**
  - Customer can sign up to the rental system with a unique username, and password.
  - Customer can update their account details (first and last name, email, address, phone number).
  - Customer can log in using their username and password.
  - Customer can view a list of the vehicles registered in the system.
  - Customer can view specific details of each registered vehicle.
  - Customer can check which vehicles are available for rent.
  - When account details have been set up, the customer can book a vehicle.
  - During the reservation of a vehicle, the customer has to make the payment.
  - Customer can cancel reservations of vehicles booked by them.
  - Customer can report technical issues to mechanics.
  - Customer can give feedback regarding the service.
- **Manager**
  - Manager can log in using a username and password.
  - Manager can add, modify, and remove vehicles listed in the inventory.
  - Manager can add, modify, and remove customer accounts.
  - Manager can add, modify, and remove staff accounts.
  - Manager can view the feedback from users.
  - Manager can log incidents and crashes.
  - Manager can update vehicle availability.
  - Manager can assess and document the condition of each vehicle.
- **Mechanic**
  - Mechanic can sign up to the rental system with a unique username, and password.
  - Mechanic can update their account details (first and last name, email, address, phone number).

- Mechanic can log in using their username and password.
- Mechanic can view a list of technical issues reported by customers.
- Mechanic can view specific details of each of these issues.
- Mechanic can mark a vehicle as repaired.
- Mechanic can notify customers about the repair status of a vehicle.

4.1.1 Use Case Diagrams

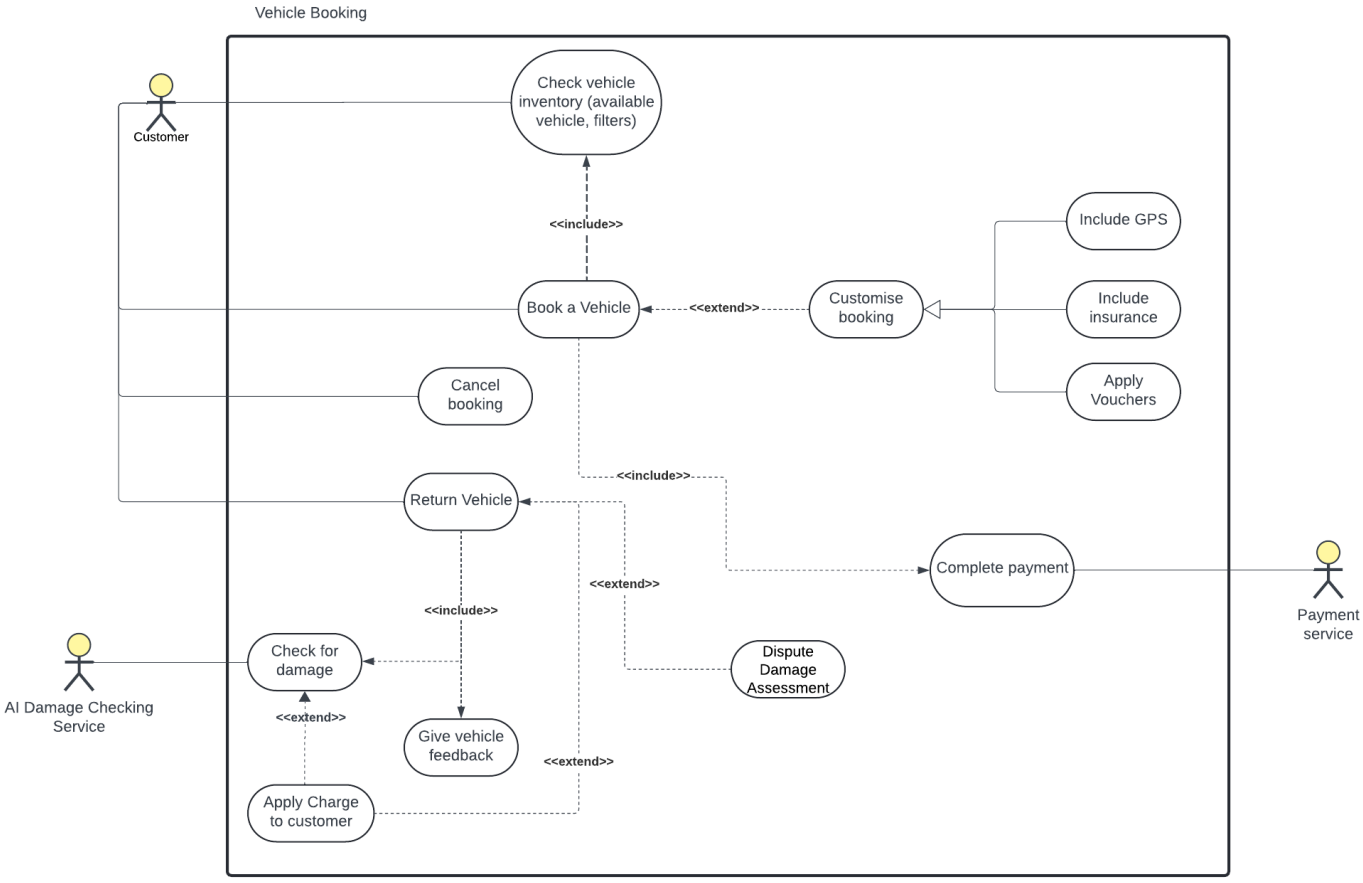


Figure 4: Vehicle Booking Use Case Diagram

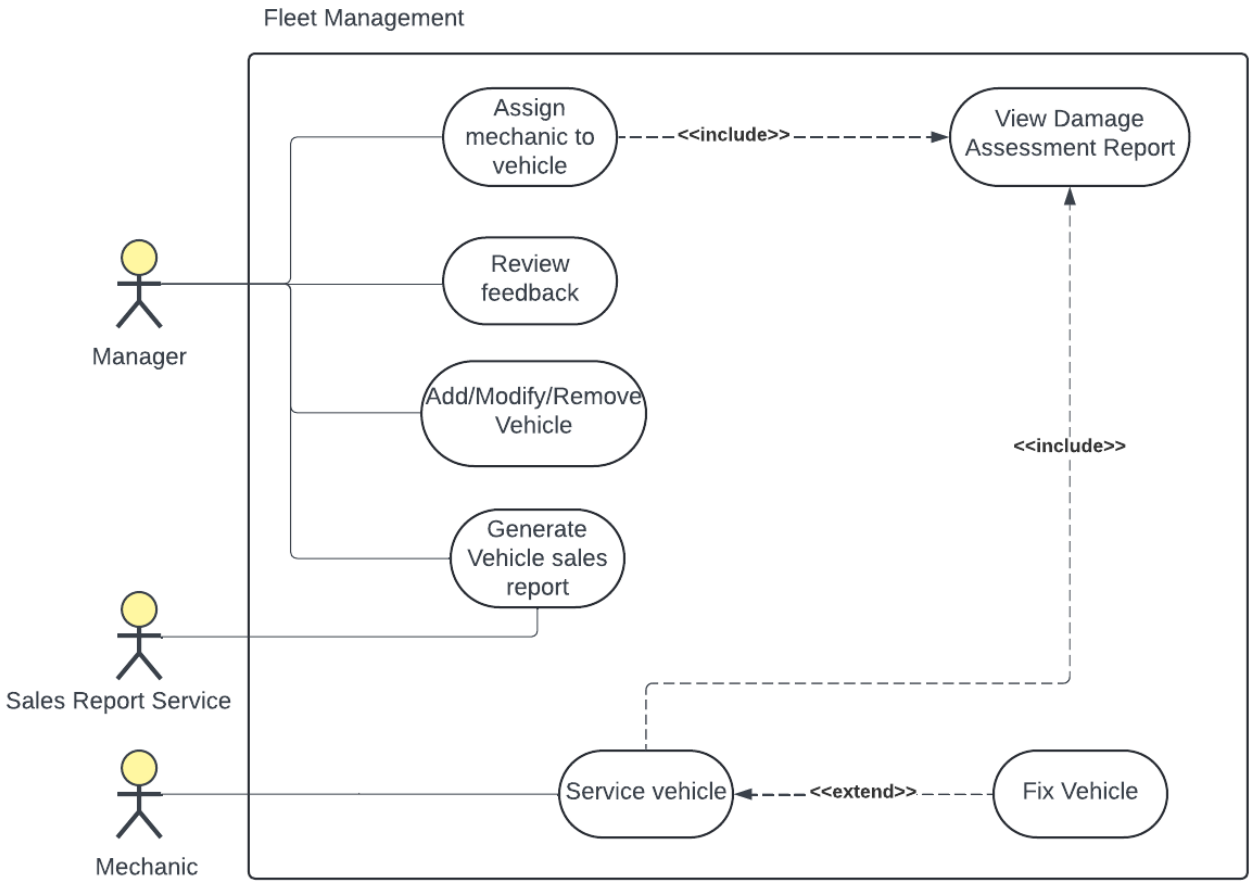


Figure 5: Fleet Management Use Case Diagram

4.1.2 Use Case Descriptions

Use Case:	Book a Vehicle
Goal In Context:	Customer reserves a vehicle of their choice through a system that manages vehicle availability, confirms booking, and processes payment. The user should be able to browse available vehicles, choose preferred options, and finalize bookings.
Scope and Level:	Vehicle Package and Vehicle Booking System
Preconditions:	Customer's contact and license details are pre-registered.
Success End Conditions:	Customer has rented a vehicle; payment has been processed.

Failed End Condition:	Customer does not rent a vehicle.
Primary, Secondary Actors:	Customer, Payment Service
Trigger:	Customer initiates a vehicle booking.
Description:	**Step** **Action** 1 Customer views available vehicles (with details like price, availability, location, etc.). 2 Customer selects a vehicle from the list to book. 3 VehiclePackage captures the customer's details and booking information (name, license, requested vehicle, time and date of booking, customization options, etc.). 4 VehiclePackage shows detailed information about the selected vehicle to the customer (price, pickup location). 5 Customer verifies booking and makes payment via payment service. 6 VehiclePackage reserves the vehicle for the customer.
Extension:	**Step** **Branching Action** 2a Customer customizes different options for the chosen vehicle (including insurance, GPS, etc.).

Variations:	**Step** **Branching Action** 1 Buyer may use: • Website • Mobile app • Telephone service • On-site booking 2 Buyer may pay by: • Cash • Credit card • Debit card
Related Information:	1. Book vehicle
Priority:	Top
Security:	Customer picks up vehicle with confirmation number
Frequency:	100 vehicles/day
Channel to Actors:	Online and In-Person Staff
Open Issues:	• What if we don't have customization options? • What if the license doesn't belong to the customer?
Due Date:	Release 1.0

Use Case:	Return Vehicle
Goal In Context:	Customer return vehicle at the end of rthe ental period. The system records the return, checks the vehicle for damage, and confirms the completion of the rental. The customer also should be able to give feedback on their experience using a rental vehicle.
Scope and Level:	Rental car company and Vehicle booking system
Preconditions:	Customer has active rental and knows designated return location

Success End Conditions:	Vehicle is successfully returned, and the system is notified and updated on the vehicle status.
Failed End Condition:	The return is unsuccessful if the customer has not returned the vehicle.
Primary, Secondary Actors:	Customer, Damage Checking Service
Trigger:	Customer attempts to return vehicle
Description:	**Step** **Action** 1 Customer returns vehicle 2 Customer gives feedback 3 Damage checking service checks for damages on vehicle 4 Customer gets notified that return has been successful
Extension:	**Step** **Branching Action** 1b Vehicle is returned late 1bi The system calculates late fees, which are added to the final charge 3a The vehicle is returned with damages 3ai Apply charge to the customer 4a Customer disputes damage assessment
Variations:	**Step** **Branching Action** 1 Customer may return the vehicle to alternate locations 2 Payment options for additional charges may include debit and credit card
Related Information:	2. Return Vehicle
Priority:	High
Security:	Return Vehicle and receive return confirmation email
Frequency:	100 returns/day on average

Channel to Actors:	In-person desk
Open Issues:	• What if the vehicle is not returned to the correct location? • How should the System handle disputes over damage returns? • What if the vehicle is not returned?
Due Date:	Release 1.0

4.2. Non-functional requirements

- The user interface should be intuitive and easy to navigate for customers and staff, minimizing the need for extensive training or support.
- Error messages should be user-friendly, offering clear explanations to guide users in resolving issues independently.
- The system should provide real-time updates on vehicle availability, booking status, and vehicle condition to prevent delays in the booking processes.
- Access control should clearly distinguish between roles (Customer, Manager, Staff, Mechanic) to restrict permissions appropriately and ensure data privacy.
- VRM should be scalable to support an increasing number of vehicles and users as businesses expand.
- The modular design should allow developers to add, remove, or modify system features without extensive rework, supporting rapid deployment of requested custom features.
- Following Agile practices, the system should be built to allow iterative improvements, with regular updates that can incorporate changing requirements and user feedback efficiently.

4.2.1 Quality tactics

4.2.2 Extensibility

For a system that requires continuous modifications or the addition of new elements, such as a vehicle rental system, achieving extensibility is crucial to accommodate changes efficiently. Extensibility ensures that new features, classes, or objects can be added without altering the core structure or existing functionalities. One effective way to achieve this is by utilizing the Factory Pattern for object creation. This ensures a future modification without direct modification to the code base, reinforcing the open/close principle.

4.2.3 Security

In an application that facilitates customer transactions, ensuring security is ideal. By employing the Decorator Pattern, security can be enhanced by dynamically adding features like encryption or authentication checks to service methods without altering the existing codebase. This allows for flexible, modular security implementations that can adapt to changing requirements while preserving the integrity and functionality of the system.

4.3. GUI Prototypes

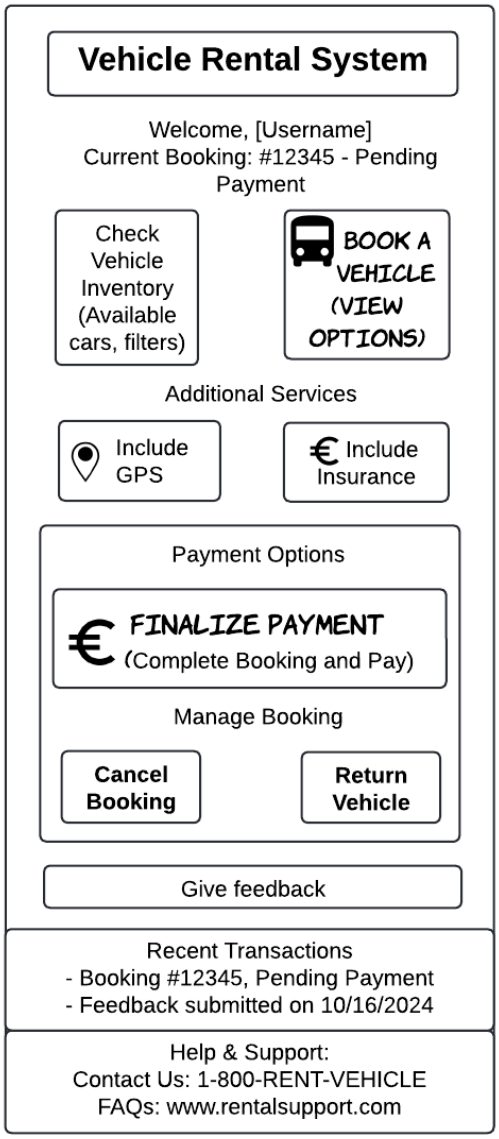


Figure 6: VRS GUI Prototype

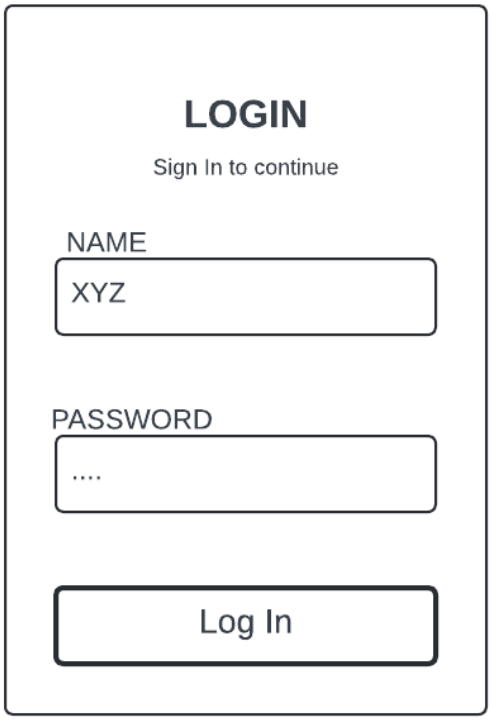


Figure 7: Login GUI Prototype

5. Architectural Patterns

5.1. Model View Controller (MVC)

The Model View Controller (MVC) is a software design architectural pattern that divides the program logic into three different components: model, view, controller. Each of these interconnected components will have associated a particular aspect of the program. Having well-defined independent components improves the overall modularization and scalability of a system, as in the MVC the business logic is separated from the front end, and vice versa.

As it was explained previously, using the MVC architectural design pattern in a software development allows the separation of business logic, user interface, and control logic, where each component can be developed and maintained without affecting each other. This allows components to be reused, reducing code duplication. This format makes adding new features easy, as updating the existing code base has less impact.

5.1.1 Model

The Model represents the internal data and business logic of an application. It defines not only the data structures of the program but also its principal operators. It is responsible for handling the data and communication with the system's database.

5.1.2 View

The view is the representation of an application's user interface, and it allows users to view information about a system and interact with the application in a simple and straightforward manner. Normally, the view is defined in HTML/CSS files, but this varies depending on the technology used.

5.1.3 Controller

The Controller represents the interconnection between the View and the Model, and it is responsible for handling user input, updating the Model, and refreshing the View when changes have been made in the Model.

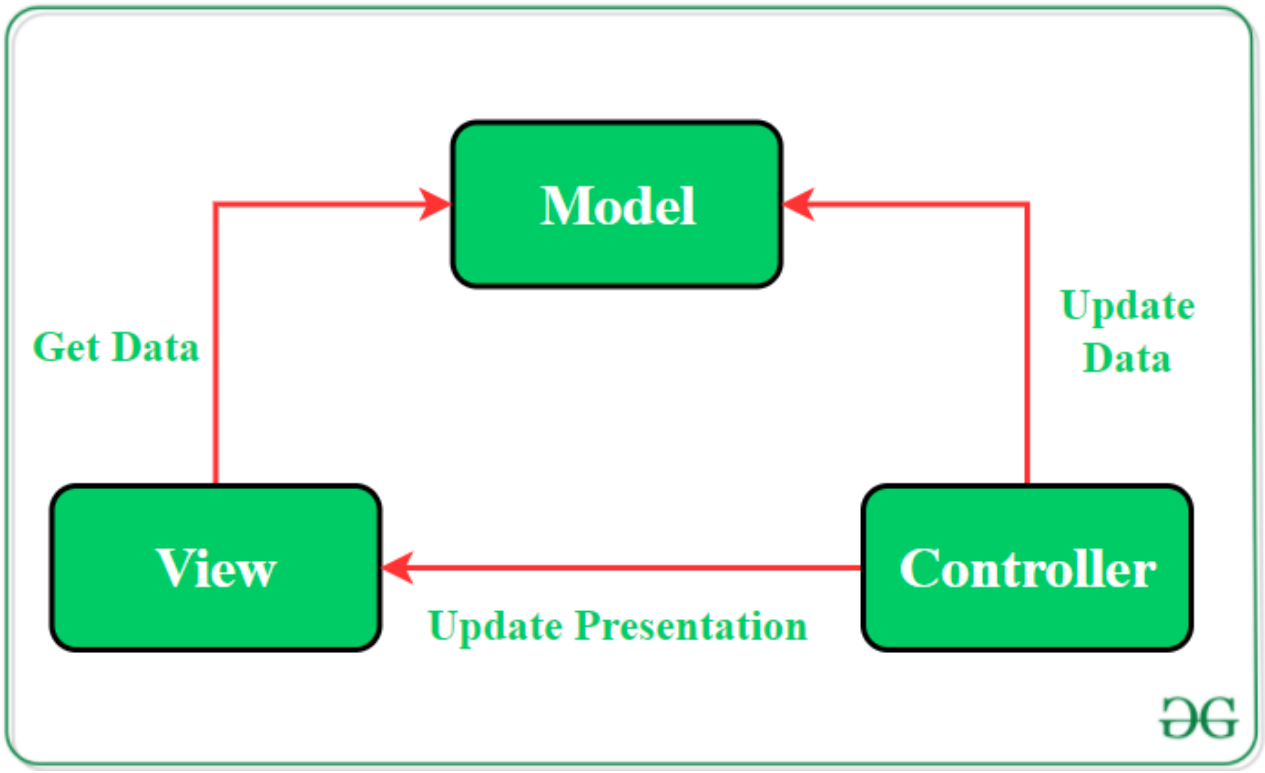


Figure 8: MVC Architecture Diagram (GeeksforGeeks 2023)

5.2. Layered (N-tier) Architecture

The Layered Architecture is an architectural design pattern that structures an application into multiple layers or tiers. Each layer is created for a specific functionality of an application, ensuring that related functionality is grouped and separated from other layers. Dividing the application into multiple layers not only enhances the organization of the code but also the usability and maintainability of the application. Typically, the layers that are commonly used in a Layered Architecture system are the following:

5.2.1 Presentation Layer

The Presentation Layer represents the user interface of a system and is normally used to display the application's data and handle user interactions. This layer is frequently coded with HTML/CSS files, but it can also be coded with frontend frameworks such as React.

5.2.2 Business Layer

The Business Layer is responsible for handling all the business logic of a system, as well as processing input from the Presentation Layer. For this project, the Business Layer can be used to define the business logic of the booking process.

5.2.3 Data Access Layer

The data access layer manages the exchange of data between the application and the database. It handles all data queries and abstracts database operations, ensuring that the business logic interacts only with this layer and not directly with the database. For this project, the Data Access Layer can be used to define the transmission procedure of the vehicles.

5.2.4 Database Layer

The Database Layer represents the back-end data store for the application, i.e. the place where all the data of a system is actually stored. This layer can be implemented as a relational database, such as MySQL, or as a NoSQL database, such as MongoDB.

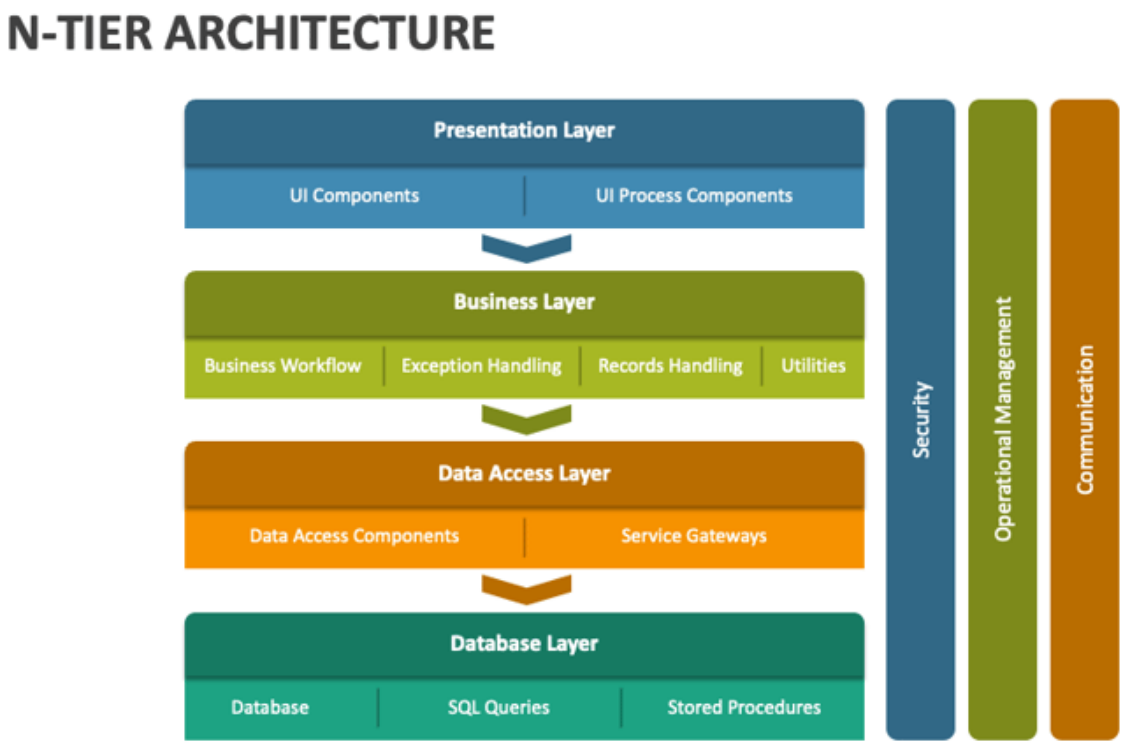


Figure 9: N-Tier Architecture Diagram (3bdelrahman 2023)

5.3. System Architecture

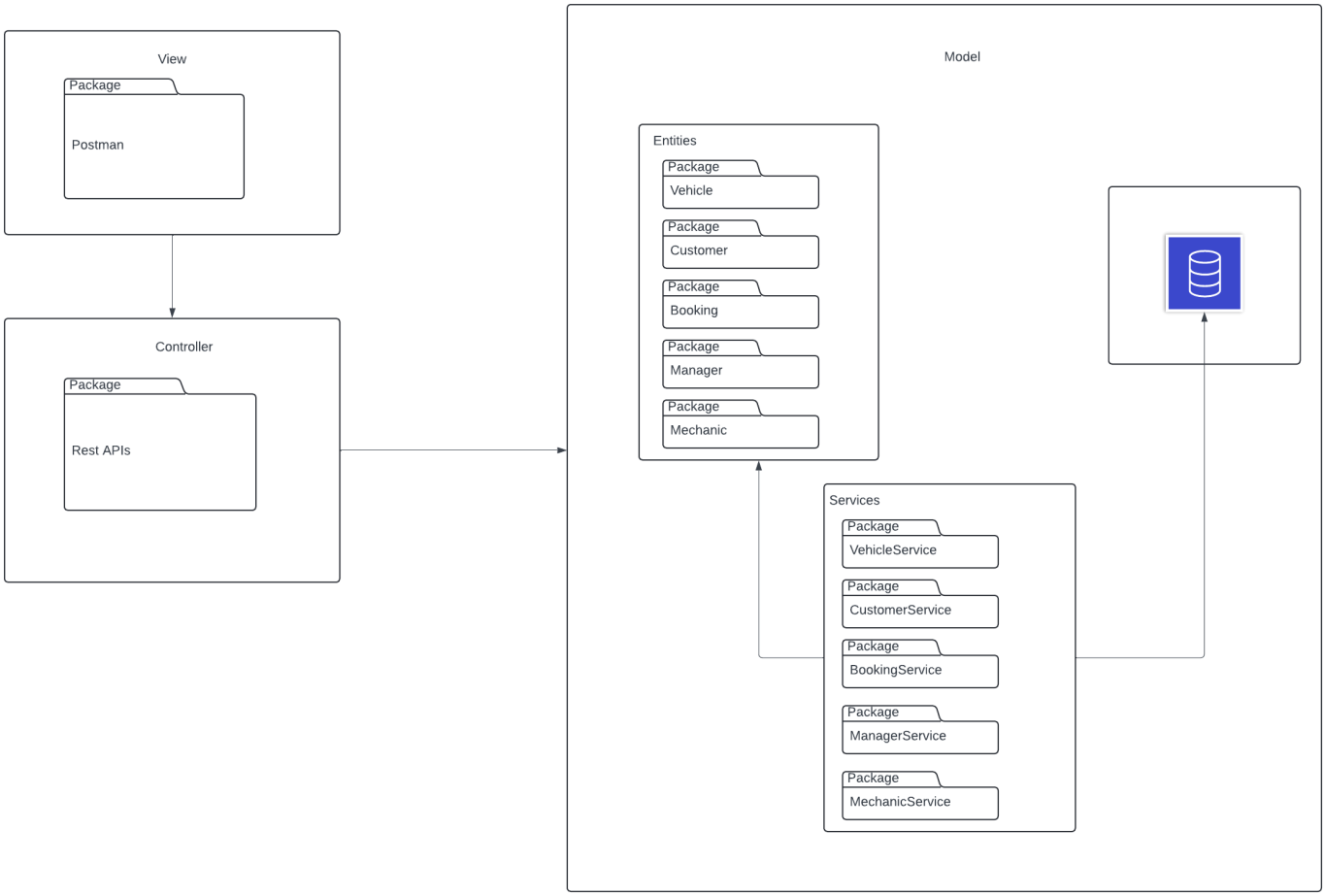


Figure 10: High-Level Package Diagram

5.4. Technology Pipeline

Spring Boot

Spring Boot is a framework that allows the creation of standalone applications in Java, Has Auto Configuration built in reducing boilerplate code and simplifies application setup, and embedded services such as Apache Tomcat which serves as the server that handles HTTP requests and responses. Dependency management is done via Maven allowing simple integration of dependencies. In this pipeline, business logic is structured with a layered architecture (Controllers, Services and Repositories).

Postman

Postman is a tool for testing RESTful APIs to verify endpoints work correctly, handle errors and to see if they meet business requirements. Manual API testing can be done for HTTP requests for Spring Boot API endpoints, and these can be turned into Automated tests using JavaScript to create assertions based on response

data. These requests can be grouped into collections to be reused by all team members when created. Postman also has documentation capabilities to generate API docs for all endpoints.

JUnit

JUnit is a testing framework Junit will ensure code quality by adding tests to components. Using Unit tests for each component like services and controllers tests each method works as expected. Integration test using Springs @SpringBootTest to load the entire application and test interactions between components. Tools such as GitHub Actions can automatically run JUnit test on every code push.

6. Analysis phase

6.1. Candidate Objects

Customers can book vehicles, and each booking is associated with a specific vehicle type. Vehicles can have additional features applied, such as GPS and insurance, which adjust the overall cost of the rental. Managers oversee vehicle availability, manage financial reports, and control staff operations within the booking system. Customers can opt for various vehicle types (cars, vans, trucks, etc.) and select additional features like GPS and insurance through a decorator pattern. They can also apply vouchers to reduce the rental cost. The booking system processes reservations, calculates rental costs, and handles payments. The system generates an invoice for each booking, detailing the transaction, and sends it to the customer.

6.2. Identified Nouns

- Customer
- Vehicle
- Vehicle Type
- Booking
- Booking System
- Manager
- Vehicle Decorator
- GPS
- Insurance
- Voucher
- Pricing
- Payment System
- Invoice

6.3. Class Diagram

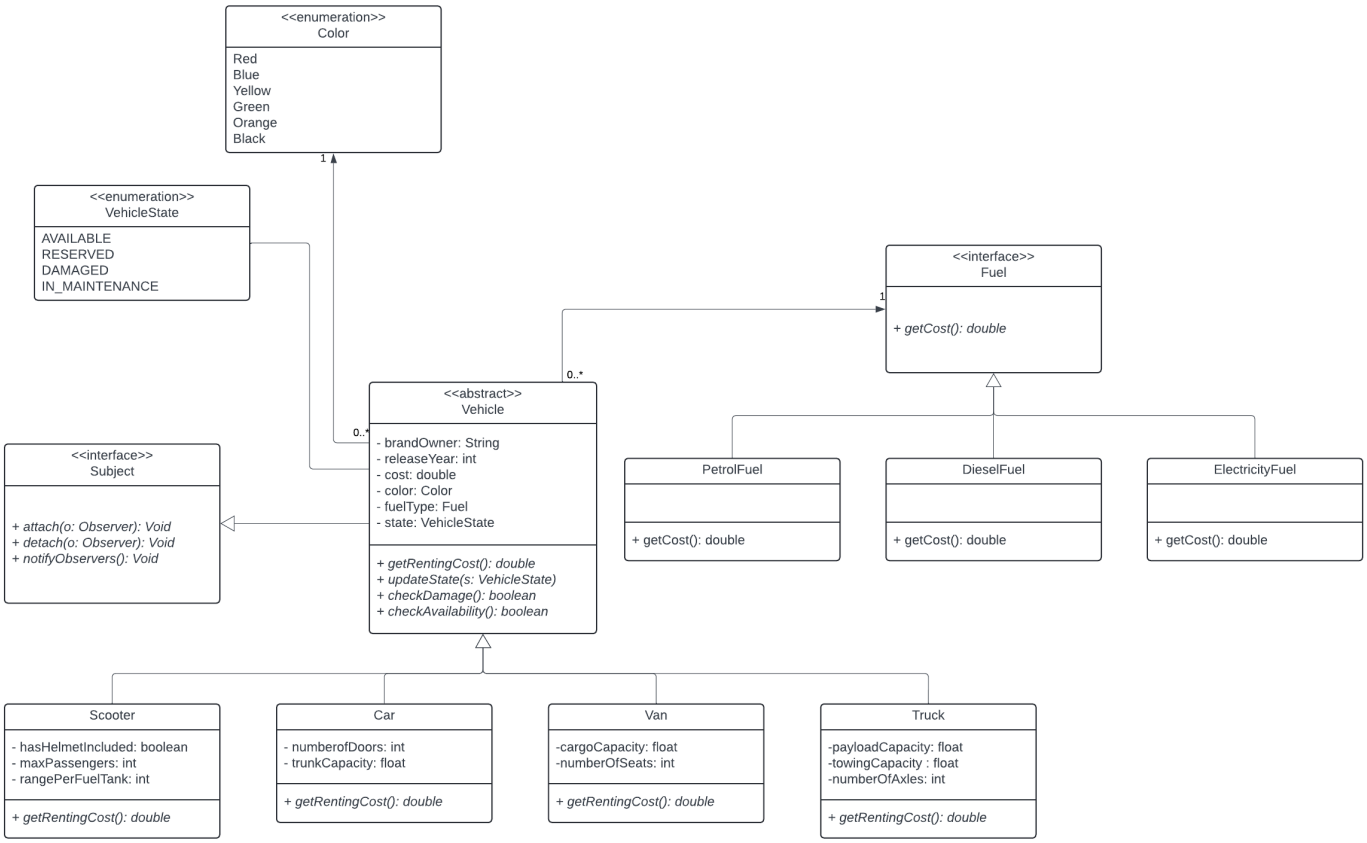


Figure 11: Vehicle class diagram

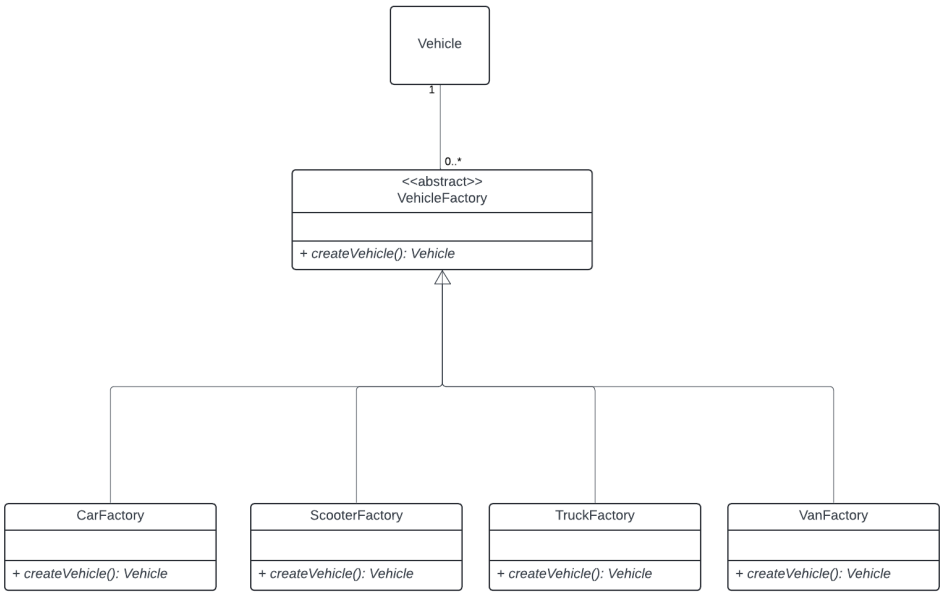


Figure 12: Vehicle factory class diagram

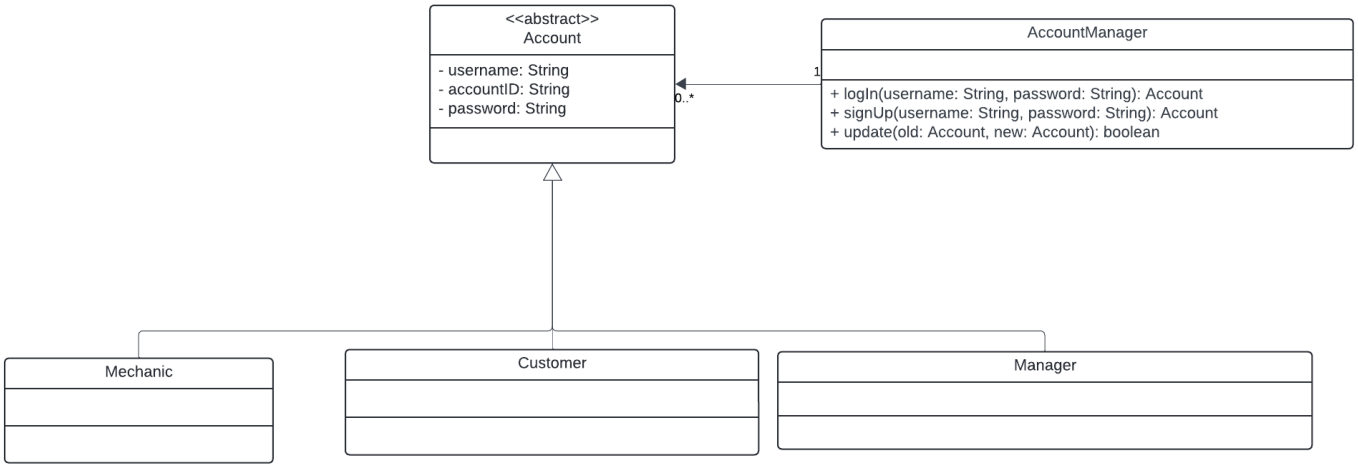


Figure 13: Account class diagram

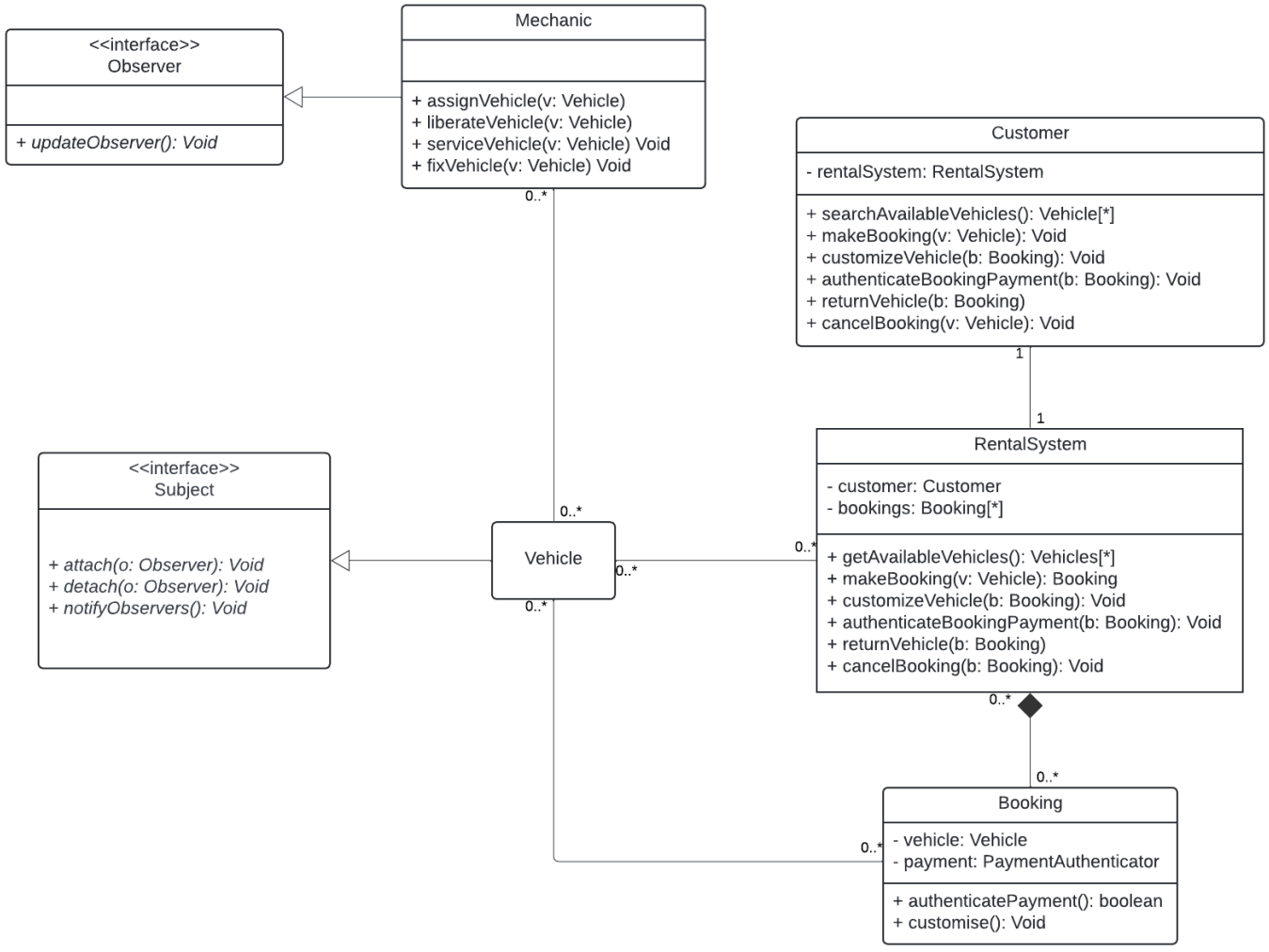


Figure 14: Customer and mechanic class diagram

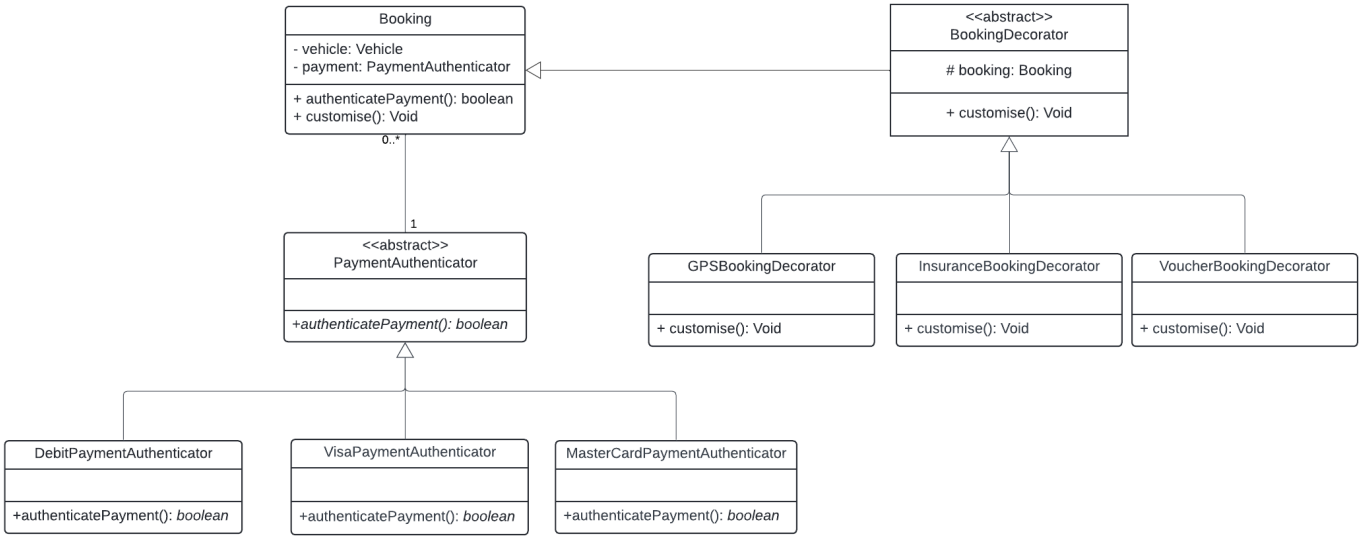


Figure 15: Booking class diagram

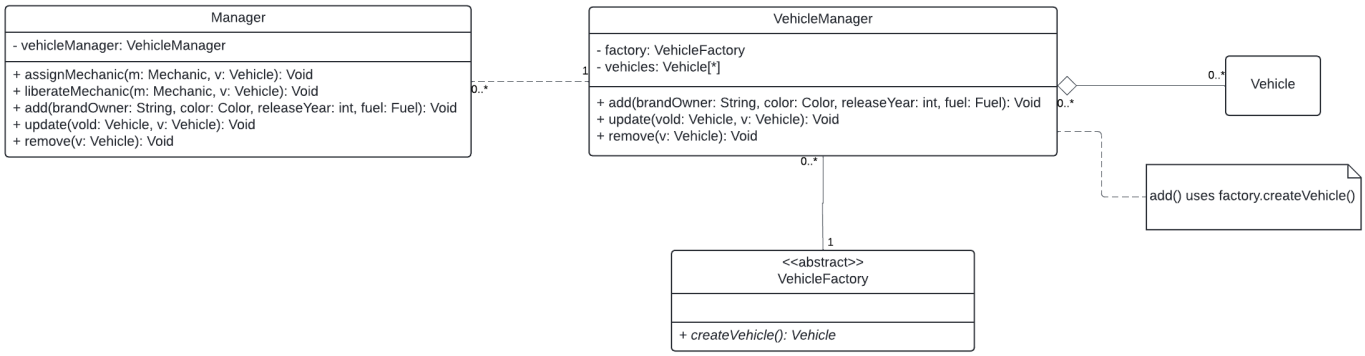


Figure 16: Manager class diagram

6.4. Sequence Diagram

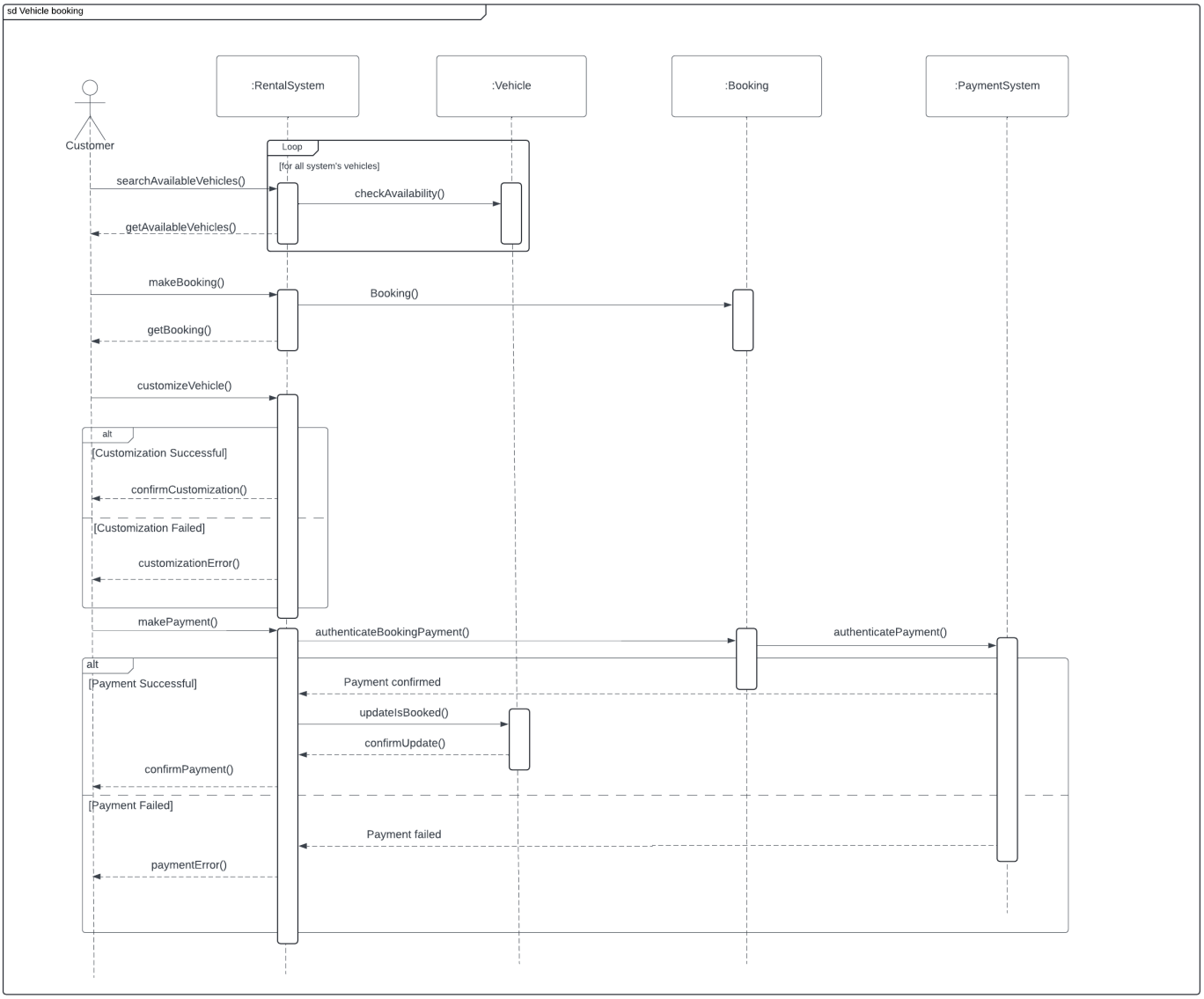


Figure 17: Sequence diagram

6.5. State Chart

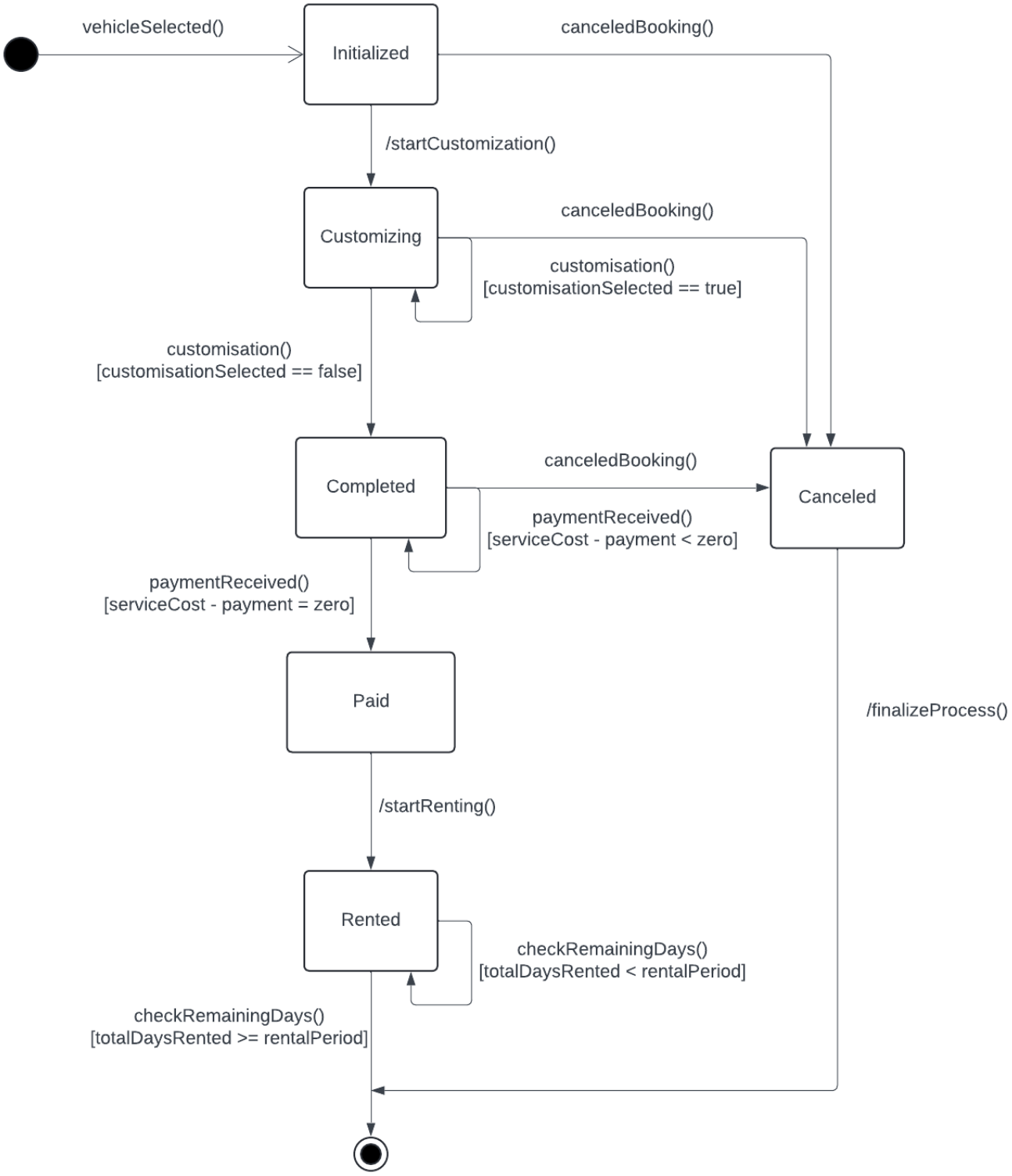


Figure 18: State chart diagram

6.6. Entity Relationship Diagram

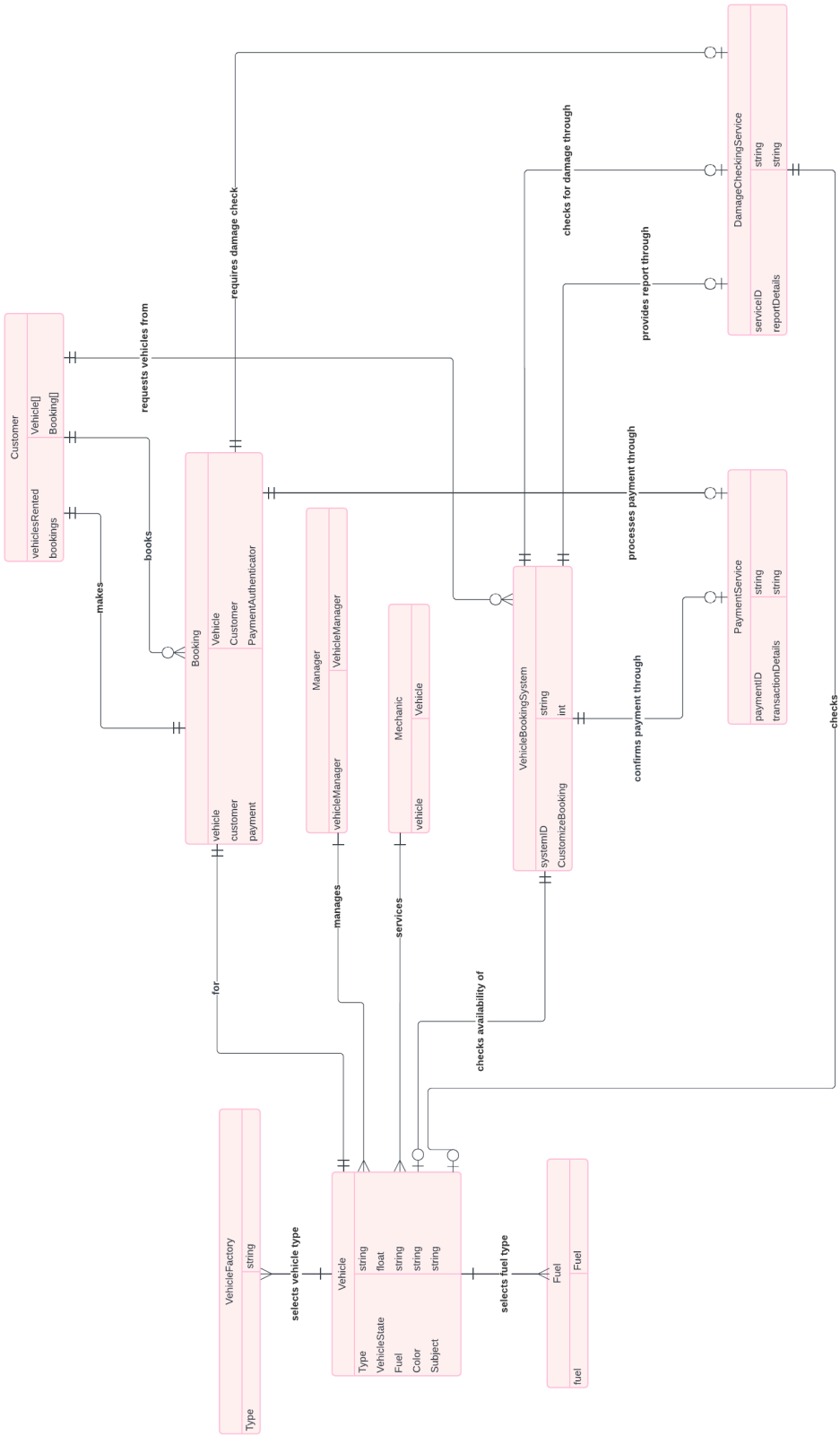


Figure 19: Entity relationship diagram

7. Transparency and Traceability

7.1. Package Summary

Package	Number of Classes
src.main.java.com.ul.vrs	1
src.main.java.com.ul.vrs.controller	4
src.main.java.com.ul.vrs.controller.command	4
src.main.java.com.ul.vrs.entity	3
src.main.java.com.ul.vrs.entity.account	4
src.main.java.com.ul.vrs.entity.booking	1
src.main.java.com.ul.vrs.entity.booking.decorator	5
src.main.java.com.ul.vrs.entity.booking.decorator.factory	1
src.main.java.com.ul.vrs.entity.booking.payment	5
src.main.java.com.ul.vrs.entity.booking.payment.strategy	3
src.main.java.com.ul.vrs.entity.vehicle	5
src.main.java.com.ul.vrs.entity.vehicle.factory	6
src.main.java.com.ul.vrs.entity.vehicle.fuel	5
src.main.java.com.ul.vrs.entity.vehicle.state	6
src.main.java.com.ul.vrs.interceptor	5
src.main.java.com.ul.vrs.jacoco	2
src.main.java.com.ul.vrs.repository	4
src.main.java.com.ul.vrs.security	3
src.main.java.com.ul.vrs.service	8
src.test.java.com.ul.vrs	3
src.test.java.com.ul.vrs.entity.account	3
src.test.java.com.ul.vrs.entity.booking	1
src.test.java.com.ul.vrs.entity.vehicle	4
src.test.java.com.ul.vrs.entity.vehicle.factory	1

src.test.java.com.ul.vrs.interceptor	1
src.test.java.com.ul.vrs.service	1
TOTAL	**89**

Table 4: Package Summary

7.2. Class Details

Package	**Class Name**	**Author**	**LOC**
com.ul.vrs.entity	Observer	David	5
	Subject	Rohan	9
	Color	Rohan	5
com.ul.vrs.entity.account	Manager	Shane	95
	Account	Manjeshwar	35
	Customer	Manjeshwar	58
	Mechanic	Shane	77
com.ul.vrs.entity.booking	Booking	Ivor	93
com.ul.vrs.entity.booking.payment	PaymentMethod	Ivor	5
	ApplePayPayment	Ivor	19
	CreditCardPayment	Ivor	25
	PaymentRequest	Ivor	27
	Payment	Ivor	13
com.ul.vrs.entity.booking.payment.strategy	ApplePayPaymentStrategy	Ivor	20
	CreditCardPaymentStrategy	Ivor	20
	PaymentStrategy	Ivor	17
com.ul.vrs.entity.booking.decorator	VoucherBookingDecorator	Ivor	24
	CFSBookingDecorator	Ivor	24
	InsuranceBookingDecorator	Ivor	24
	BookingDecorator	Ivor	23

	Customization	Ivor	5
com.ul.vrs.entity.booking.decorator.factory	BookingDecoratorFactoryMethod	David	25
com.ul.vrs.entity.vehicle	Vehicle	Rohan	180
	Car	Rohan	97
	Scooter	Rohan	113
	Truck	Rohan	108
	Van	Rohan	93
com.ul.vrs.entity.vehicle.state	AvailableVehicleState	David	28
	ReservedVehicleState	David	28
	InMaintenanceVehicleState	David	29
	DamageVehicleState	David	29
	VehicleState	David	21
	StateConverter	Rohan	37
com.ul.vrs.entity.vehicle.fuel	FuelConverter	Rohan	29
	PetrolFuel	David	8
	DieselFuel	David	8
	ElectricityFuel	David	8
	Fuel	David	18
com.ul.vrs.entity.vehicle.factory	VehicleFactory	Shane	7
	CarFactory	Shane	22
	ScooterFactory	Shane	22
	TruckFactory	Shane	22
	VanFactory	Shane	22
	VehicleFactoryMethod	David	92
com.ul.vrs.repository	AccountRepository	Ivor	10
	BookingRepository	Rohan/Ivor	13
	CustomerRepository	Rohan	10
	VehicleRepository	Rohan/Ivor	10
com.ul.vrs.controller	VehicleController	Rohan	68

	HomeController	Manjeshwar	55
	RentalSystemController	David	136
	AccountController	Manjeshwar	52
com.ul.vrs.controller.command	Command	Rohan	6
	CommandInvoker	Rohan	50
	ReturnCarCommand	Rohan	29
	OpenGateCommand	Rohan	20
com.ul.vrs.service	DamageCheckingService	Manjeshwar	62
	VehicleManagerService	Rohan	83
	RentalSystemService	David/ Manjeshwar	145
	AccountManagerService	Manjeshwar	64
	SalesReportService	Manjeshwar	30
	GateService	Rohan	11
	CustomUserDetailsService	Ivor	31
	VehicleManagerServiceHelper	Rohan,Ivor	16
com.ul.vrs.jacoco	ExcludeConstructorFromGeneratedJacoco	David	11
	ExcludeMethodFromGeneratedJacoco	David	11
com.ul.vrs.security	JwtTokenUtil	Ivor	76
	JwtRequestFilter	Ivor	52
	SecurityConfig	Ivor	47
com.ul.vrs.interceptor	GPSInterceptor	Manjeshwar	24
	InsuranceInterceptor	Manjeshwar	24
	Interceptor	Manjeshwar	19
	LoggingInterceptor	Manjeshwar	22
	VoucherInterceptor	Manjeshwar	24
com.ul.vrs	VehicleRentalSystemApplication	Ivor	11

Table 5: Class details

Package	Test Class Name	Author	LOC
com.ul.vrs.entity.account	CustomerTest	Manjeshwar	297
	ManagerTest	Manjeshwar	116
	MechanicTest	Rohan	224
com.ul.vrs.entity.booking	BookingTest	Shane	90
com.ul.vrs.entity.vehicle	CarTests	David	261
	ScooterTests	David	327
	TruckTests	David	276
	VanTests	David	255
com.ul.vrs.entity.vehicle.factory	VehicleFactoryMethodTest	David	93
com.ul.vrs.interceptor	InterceptorTest	Manjeshwar	73
com.ul.vrs.service	RentalSystemServiceTest	Rohan	269
com.ul.vrs	PaymentSystemTest	Ivor	56
	VehicleRentalSystemApplicationTests	David	56

Table 6: Test Class Details

7.3. Workload Overview in LOC

Member	Lines of Code
Shane Barden	464
Ivor D Souza	1546
Manjeshwar Aniruddh Mallya	1477
David Parreño	2748
Rohan Sikder	1729

Table 7: Workload in Lines of Code (LOC)

Figure 20: Contributions of each member

7.4. Code Responsibilities

The following table gives a comprehensive overview of the features that have been implemented for the car rental system, as well as their authors and contributions. Note that for most of the features, the other members also contribute to the fixing and completeness of the feature. However, this section highlights the contributors who have done more work on each feature.

Feature	Authors	Contribution
Command design pattern	Rohan	Created Commands for Opening Gate for returning vehicle
Singleton design pattern	David	Implementation of the pattern for the vehicle manager service class.

Factory design pattern	Shane and David	**Shane:** Created vehicle factories for Car, Scooter, Truck and Van **David:** Implementation of the vehicle and booking factory method classes.
Interceptor design pattern	Manjeshwar	Implemented the interceptor pattern for customer class in account, with custom interceptors for logging, GPS, Insurance and Voucher. Wrote tests for the interceptor.
Decorator design pattern	Ivor and David	**Ivor:** Implemented the decorator design pattern for booking customization. **David:** Decorator construction bug fixes related to how the booking decorator copies the attributes of the passed booking.
Strategy design pattern	Ivor	Implemented the PaymentStrategy interface and concrete payment strategies. Applied the strategy to 2 payment methods and created PaymentRequest for them.
Observer design pattern	Shane and Rohan	**Shane:** Created Observer methods for mechanic **Rohan:** Created Observer subject for Vehicle.
Database support through ORM	Rohan and Ivor	**Rohan:** Initialized H2 Database, Partial Integration of data layer and created SQL commands for adding existing vehicles. **Ivor:** Created SQL commands for defining database schema and added JPA annotations for table structure and indexes.

JUnit testing	Everyone	Implementation of tests for various classes, especially the entity classes and some services. The workload was divided so that each member had to implement tests for classes other than the ones they developed.
Postman testing	Rohan and Ivor	**Rohan:** Created Requests to test API's and to test request. **Ivor:** Added booking and login related API requests. Created postman environment for VRS.
Software metrics	Ivor	Setup SonarQube and JDepends for collecting metrics.
CI/CD	Rohan	Created Github workflow to automate Springboot build, Junit test and Postman query's.
Security	Ivor	Added JWT based authentication and password hashing for storage.
Changelog generator	David	Integration, implementation, and customisation of the Maven plugin for the changelog generation.
Code analyser	David	Integration of the Maven plugin for static code analyser PMD.
Design-time package diagram	Rohan	Created Package Diagram using PlantUML.
Design-time class diagram	Shane	Created the design time class diagram based off current code using intellij built in functionality.
Design-time sequence diagram	David	Generation of the sequence diagram based on the current code and through IntelliJ plugins.
Component diagram	Manjeshwar	Created Component Diagram using PlantUML.

Deployment diagram	Ivor	Created a containerized deployment diagram of the app.

Table 8: Code responsibilities

7.5. Code Diary

Week 9	**Contributions**	**Issues Arising**	**Plan for next week**
Shane	- Implemented the Factory module with vehicle factories. - Created a new Factory package and updated its parameters. - Updated and relocated package names for better organization. - Merged ‘dev‘ branch into ‘vehicle‘ and ensured smooth integration of vehicle factories. - Created project configuration file.	- Refactoring package structure caused minor delays in development. - Had issues installing Maven, which delayed initial setup. - Initial implementation of the factory design required additional testing to ensure compatibility.	- Focus on creating the ‘Booking‘ module and its associated tests. - Begin implementing tests for new modules to ensure stability.
Ivor	- Initialized the springboot app using Spring Initialzr. - Implemented Booking service and updated CRUD APIs endpoints and requests. - Implemented decorator design pattern for booking customization. - Added postman requests for booking service APIs	- Needed to resolve conflicts with Rental System Service for new booking endpoints	- Implement strategy design pattern for booking payment. - Integrate software metrics collection tools.

Manjeshwar	- Implemented a user login feature providing basic authentication and session management. - Introduced initial user-related domain classes aligned with UML diagrams. - Prepared foundational service and controller endpoints to handle login requests.	- Session handling was not fully clear; needed a strategy for persistent sessions or stateless JWT tokens. - Limited test coverage meant early bugs were missed.	- Add integration tests covering login endpoints and authentication workflows. - Decide on a stateless (JWT) vs. stateful (session) approach for authentication.
David	- README and .gitignore. - Integration of PMD and the CHANGELOG generator. - Organise the project considering an MVC style. - Implementation of the rental system service and controller. - Adapt the classes from the vehicle package to follow the class diagram.	- The CHANGELOG generator needs further customisation, and PMD violations have not been resolved.	- JaCoCo integration. - Customise CHANGELOG.
Rohan	- CRUD operations for vehicle management. - Based on UML diagram: added interfaces, abstract Vehicle class, concrete vehicle classes (Car, Truck, etc.), Fuel implementations, and enums for Color and VehicleState.	- Merge conflicts and old imports and package references. - fix(vehicle): fixed add/update vehicles by using JsonTypeInfo annotations.	- Unit Test for RentalSystemService.

Table 9: Code Diary of Week 9

Week 10	**Contributions**	**Issues Arising**	**Plan for next week**

Shane	- Created ‘BookingTests.java‘ to define the structure for Booking-related tests. - Implemented the ‘Mechanic.java‘ class, including core logic and functionality. - Configured the Observer pattern for interaction between ‘Mechanic‘ and ‘Vehicle‘ classes to enable dynamic updates. - Updated ‘pom.xml‘ to configure project dependencies. - Began implementing tests for the Booking module.	- Encountered challenges with test case coverage for certain edge scenarios. - Dependencies in ‘pom.xml‘ required updates to support newly introduced modules.	- Complete the implementation of tests for the ‘Booking‘ module. - Work on merging test branches and integrating Booking features into the develop branch.
Ivor	- Implemented strategy design pattern for booking payment. - Created PaymentRequest object and update postman API request for booking payment. - Integrated SonarQube and Jdepends for collecting metrics.	- Created github issues of high risk issues from analysis of SonarQube report	- Add booking payment system tests. - Work on creating repository for the app.

Manjeshwar	- Implemented the Vehicle Rental System (VRS) core logic, introducing Customer, Account, and AccountManager classes as per UML specs. - Established clearer layering in controllers (RentalSystemController, AccountController) to separate concerns and simplify code maintenance. - Improved domain-model alignment: ensured Customer and Account entities follow UML-defined attributes and behaviors.	- Integrating new classes with existing authentication flow caused overlapping responsibilities in controllers. - Some business rules (e.g., account creation prerequisites) not fully documented, leading to guesswork.	- Reassess controller responsibilities, possibly introducing a dedicated AuthenticationController to streamline logic. - Document business rules for account setup to prevent ambiguities. - Add unit tests focusing on the interplay between AccountManager and Customer classes.
David	- Bug fixes for decorators. - Fix authentication endpoints. - JaCoCo integration. - CHANGELOG based on the conventional commit naming. - Fix PMD violations.	- Unexisting Sonarqube properties in the Springboot application settings file. - Update Postman data collection for current endpoints. - The Booking decorator creation process is not encapsulated. - Tags and commit URLs still don't work in the CHANGELOG file.	- Delete unexisting Sonarqube properties. - Encapsulation of the booking decorator using the Factory Method. - Implementation of the Vehicle state design pattern. - Implementation of the Singleton design pattern. - Implementation of the Factory Method for vehicles. - Testing of all vehicles and their factories. - Further customisation of the CHANGELOG file.

Rohan	- Add unit tests for RentalSystemService. - Fix(mechanic): Implement required Observer methods and update state checks in Mechanic class. - Fix(mechanic): resolved vehicle update issues and added MechanicTests. - Update Postman Collection.		- Started to look into h2 database implementation. - Create CI/CD to automate Build, JUnit and Postman.

Table 10: Code Diary of Week 10

Week 11	**Contributions**	**Issues Arising**	**Plan for next week**
Shane	- Updated ‘BookingTests.java‘ with additional test cases for broader coverage. - Configured ‘application.properties‘ for booking features. - Merged the ‘test/booking‘ branch into ‘develop‘ for group testing. - Verified and ensured all tests passed. - Implemented and refactored the ‘Manager.java‘ class. - Integrated Manager functionality into the system workflow.	- Resolving merge conflicts while merging the ‘test/booking‘ branch. - Minor configuration issues in ‘application.properties‘ delayed initial tests.	- Begin work on finalizing Booking module tests. - Focus on improving the modularity of the Booking module.

Ivor	- Created SQL statements for database schema. - Added JPA annotations for table structure and indexes. - Updated entities fields and added default constructors for JPA to instantiate entity objects. - Added JUnit tests for booking payment system - Tested and updated postman collection after the repository addition	- Deciding on correct DB schema to utilize for each entity based on the design pattern implemented. - JUnit tests fail - they need to be updated to include mock repository	- Add JWT based auth for secure authentication and identification of user creating a booking. - Fix JUnit tests with mock repositories. - Add hashing for password stored in DB.
Manjeshwar	- Extended domain logic by refining Account and Customer classes to better handle state transitions and validations. - Created Mechanic and Manager classes, providing the foundation for role-based functionalities (vehicle maintenance, user account oversight). - Developed unit tests for the new classes, improving overall code reliability. - Fixed bugs related to null references in AccountManager and misaligned field mappings in Customer entities.	- Mechanic and Manager classes revealed architectural gaps: certain functionalities overlapped with existing AccountManager roles. - Some tests were overly reliant on hard-coded data, causing fragile test scenarios.	- Revisit class responsibilities to ensure clear role definitions for Mechanic, Manager, and AccountManager. - Introduce test utilities or a builder pattern to create test data dynamically, reducing brittle tests. - Increase integration testing to confirm proper end-to-end behavior.

David	- Delete unexisting Sonarqube properties. - Update Postman data collection for current endpoints. - Encapsulation of the booking decorator using the Factory Method. - Implementation of the Vehicle state design pattern. - Implementation of Singleton in the vehicle manager service. - Include exclude annotations for JaCoCo. - Implementation of the Factory Method for vehicles. - Change the methods' signature of vehicle factories. - Testing of all vehicles and their factories. - Further customisation of the CHANGELOG file.	- Vehicle state is not extensible as it is an enum of states. - Booking price is not based on the vehicle's properties. - It is still necessary to implement a user permissions checker.	- Adjust the booking price based on the vehicle. - Implementation of the account logging service. - Refactoring of the Vehicle State to use inheritance.
Rohan	- WIP Database using h2 repository - Fixed PMD violation by removing unused imports in VehicleManagerService. Updated build to run tests during the Maven build process. - CI/CD to automate Springboot Build, JUnit tests and Postman. - Add API endpoint to return the vehicle and open gate using Command Pattern.	- Having ORM issues with the database, got Ivor to look through and help.	- Continue integrating the database.

Table 11: Code Diary of Week 11

Week 12	**Contributions**
Shane	- Renamed ‘MechanicTests.java‘ to ‘MechanicTest.java‘ to follow conventions. - Final updates to ‘BookingTests.java‘ for comprehensive test coverage. - Verified the Booking module’s functionality and integrated feedback from testing. - Refactored the ‘Manager.java‘ class for improved modularity. - Integrated the ‘Manager‘ functionality with other system modules. - Updated ‘VehicleFactory.java‘ to support new Manager workflows.
Ivor	- Added JWT based authentication for secure access to the Vehicle Rental System. - Used BycryptEncoder for hashing passwords stored in DB. - Refactored Account, Customer entities and repositories to allow the new authentication mechanism. - Added postman requests for login and signup. Updated postman script to include token authorization header to each API request. - Fixed minor price calculation issue in booking decorator class. - Add postman environment file for github CI checks

Manjeshwar	- Introduced an interceptor to the Customer class to handle cross-cutting concerns such as request validation or logging. - Created test cases specifically for the interceptor, ensuring it correctly applies logic to incoming requests. - Fixed minor bugs in Customer logic, addressing previously unhandled exceptions and edge cases. - Added missing functionality to Customer, such as dynamic profile updates and improved exception handling in line with requirements.
David	- Refactoring of the Command Invoker initialisation. - Refactoring of the payment strategy implementation. - Adjust the booking price based on the vehicle. - Implementation of the account logging service. - Refactoring of the Vehicle State to use inheritance.
Rohan	- Refactor Command and Invoker implementation for extensibility and dynamic command execution. - Fix Mechanic and Test Classes to Ensure Correct Behavior and Repository Integration. - Fix ManagerTest to work with repository and update assignMechanicToVehicle for proper repository interaction. - Fix(postman): Fixed partial Postman query's. - Fix(tests): Update in to mock repository updates. - Fix: Add and improve state handling logic for VehicleState.

Table 12: Code Diary of Week 12

8. Code snippets

8.1. SOLID principles

SOLID principles are a comprehensive list of statements that include the main techniques that might help to design minimal interfaces for a software system. Considering these principles can enhance the maintainability and extensibility of the software.

Single-responsibility principle:

The Single-responsibility principle states that "there should never be more than one reason for a class to change" (Martin 2003). This principle was applied in the project as the Model-View-Controller, which is an architectural design pattern that separates the concerns of a program into the model, view, and controller.

Open-closed principle

According to Meyer (1997), "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". In other words, the Open-closed principle states that any software solution must guarantee extensibility without modifying the current source code. Concerning the project, this principle has been considered while applying the Command and State design patterns, as they are behaviour design patterns that allow including new commands and states without altering the current implementation of existing commands or states, respectively.

Liskov substitution principle

Based on Liskov (1987), "functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it". In other words, objects of a class should be replaceable with objects of a subclass without affecting the correctness. The project considers this principle as classes have been abstracted when generic attributes are found.

Interface segregation principle

The Interface segregation principle states that "clients should not be forced to depend upon an interface that they do not use." (Martin 2003). This means that clients will only have to know about the methods that are of interest to them. This principle has been considered while using contracts with the minimal defined operations necessary for the correct execution of the program.

Dependency inversion principle

The Dependency inversion principle declares that software should "depend upon abstractions, not concretes" (Martin 2000), that is high-level modules should not depend on low-level modules, and both should only depend on abstractions. The project applies this principle as interfaces and abstract classes have been defined when several classes share common attributes or operations where polymorphism can be beneficial for the maintainability, code understanding, and extensibility of the software.

8.2. Model-View-Controller (MVC)

Explanation of MVC Components

1. **Model:**
    - The Model layer represents the business data and logic. In this project, it is implemented through classes in the entity package, such as Vehicle or Account.
    - The model encapsulates the application's state and provides an interface for data manipulation.
2. **View:**
    - Since this is a backend application, the view is simulated using Postman, which acts as the front-end. API endpoints are used to interact with the system and retrieve or manipulate data.
3. **Controller:**
    - The Controller layer handles incoming HTTP requests, maps them to specific endpoints and delegates the execution to the Service layer. It is implemented in the controller package, containing classes like VehicleController and AccountController.
4. **Service (Business Logic Layer):**
    - The Service layer implements core business logic, ensuring the separation of business rules from both the Controller and Data layers.
5. **Repository (Data Access Layer):**
    - Interfaces in the repository package include CustomerRepository, VehicleRepository and BookingRepository. These extend the JpaRepository interface with CRUD operations and the ability to define custom query methods for database interaction.
    - The database layer is managed in the project by JPA and Hibernate enabling object-relational mapping (ORM) of entities defined in the entity package.
    - JPA annotations are added to Entities such as Customer, Vehicle and Booking to define table mappings and relationships.

MVC Code Snippets

A. Controller Layer: VehicleController.java

This class demonstrates how the application handles HTTP requests and delegates the logic to the Service layer.

```java
@RestController
@RequestMapping("/api/vehicles")
public class VehicleController {
    @Autowired
    private VehicleManagerService vehicleService;

    @Autowired
    private AccountManagerService accountManagerService;

    // Get all vehicles in the system - http://localhost:8080/api/vehicles
    @GetMapping
    public List<Vehicle> getAllVehicles() {
        return vehicleService.getAllVehicles();
    }

    // Get a specific vehicle by its ID - http://localhost:8080/api/vehicles/{id}
    @GetMapping("/{id}")
    public ResponseEntity<Vehicle> getVehicleById(@PathVariable Long id) {
        Optional<Vehicle> vehicle = vehicleService.getVehicleById(id);

        if (vehicle.isPresent()) {
            return ResponseEntity.ok(vehicle.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Add a new vehicle to the system - http://localhost:8080/api/vehicles
    @PostMapping
    public ResponseEntity<Vehicle> addVehicle(@RequestBody Vehicle vehicle) {
        Vehicle newVehicle = vehicleService.addVehicle(vehicle);
        return ResponseEntity.ok(newVehicle);
    }
```

Figure 21: Vehicle Controller

B. Service Layer: VehicleManagerService.java

The Service layer encapsulates the business logic.

```java
@Service
public class VehicleManagerService {
    @Autowired
    VehicleRepository vehicleRepository;

    private static VehicleManagerService instance;

    public static final synchronized VehicleManagerService getInstance() {
        if (instance == null) {
            instance = new VehicleManagerService();
        }

        return instance;
    }

    private VehicleManagerService() {

    }

    public List<Vehicle> getAllVehicles() {
        return vehicleRepository.findAll();
    }

    public Optional<Vehicle> getVehicleById(long id) {
        return vehicleRepository.findById(id);
    }

    public Vehicle addVehicle(Vehicle vehicle) {
        long vehicleID = vehicle.getID();

        // Update ID so it is unique
        if (getVehicleById(vehicleID).isPresent()) {
            vehicleID = generateID();
            vehicle.setID(vehicleID);
        }

        vehicleRepository.save(vehicle);
        System.out.println("Vehicle added with ID: " + vehicle.getID());
        return vehicle;
    }
```

Figure 22: Vehicle Service

C. Model Layer: Vehicle.java

The Model layer represents the data structure used by the application. For example:

```java
public abstract class Vehicle implements Subject {
    @Id
    private long ID;
    private final String name;
    private final String brandOwner;
    private final int releaseYear;
    protected final double cost;
    @Enumerated(EnumType.STRING)
    private final Color color;
    @Convert(converter = FuelConverter.class)
    private final Fuel fuelType;
    @Convert(converter = StateConverter.class)
    @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "type")
    @JsonSubTypes({
        @JsonSubTypes.Type(value = AvailableVehicleState.class, name = "Available"),
        @JsonSubTypes.Type(value = DamagedVehicleState.class, name = "Damaged"),
        @JsonSubTypes.Type(value = InMaintenanceVehicleState.class, name = "InMaintenance"),
        @JsonSubTypes.Type(value = ReservedVehicleState.class, name = "Reserved")
    })
    private VehicleState vehicleState;
    @Transient
    private List<Observer> observers;

    public Vehicle(long ID, String name, String brandOwner, int releaseYear, double cost, Color color,
        this.ID = ID;
        this.name = name;
        this.brandOwner = brandOwner;
        this.releaseYear = releaseYear;
        this.cost = cost;
        this.color = color;
        this.fuelType = fuelType;
        this.vehicleState = vehicleState;
        this.observers = new ArrayList<>();
    }
```

Figure 23: Vehicle Model

D. Data Layer: VehicleRepository.java

The Data layer (Repository) manages data persistence and retrieval using JPA repositories. The `VehicleRepository` interface for example, abstracts CRUD operations and allows defining custom queries for the `Vehicle` entity.

```java
@Repository
public interface VehicleRepository extends JpaRepository<Vehicle, Long> {

}
```

Figure 24: Vehicle Repository

E. Simulation of the View Using Postman

Postman is used to simulate the view. API endpoints are tested and demonstrated for system interactions including creating, reading, updating and deleting entities.

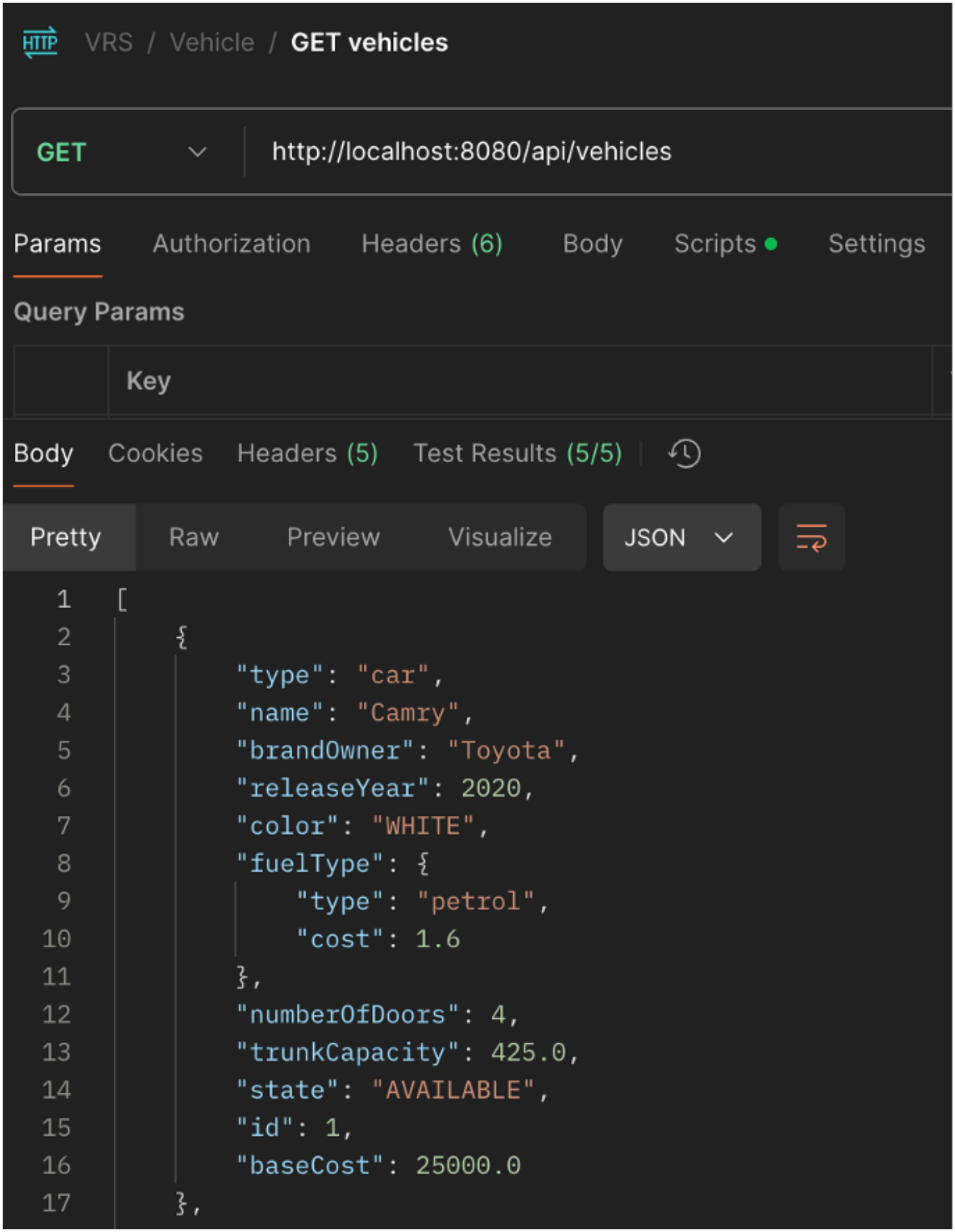


Figure 25: GET Vehicles

Directory Structure Screenshot

- **Controller Layer:** (e.g., `VehicleController` in the `controller` package).
- **Service Layer:** (e.g., `VehicleManagerService` in the `service` package).
- **Model Layer:** (e.g., `Vehicle.java` in the `entity` package).
- **Data Layer:** (e.g., `VehicleRepository.java` in the `repository` package).

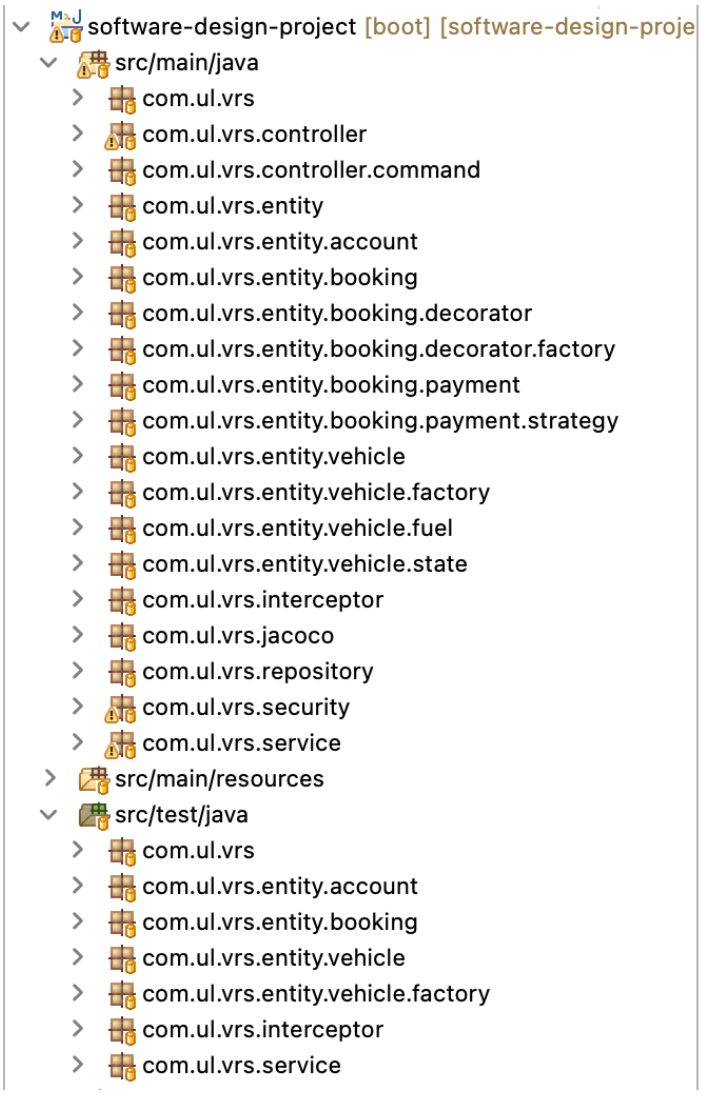


Figure 26: Project Directory Structure

This Spring Boot project implements The Model View Controller architectural pattern with separation of concerns in each layer:

- The Model layer encapsulates business data and logic using plain old java objects.
- The View layer is simulated using Postman which serves as the interface for interacting with the application's API endpoints. This allows testing and visualization of the system's functionality without a

dedicated front-end.

- The Controller layer handles incoming HTTP requests and delegates them to the Service layer.
- The Service layer implements the core business logic which provides an abstraction from both the Controller and data handling concerns.
- The Repository layer provides an abstraction for interacting with the database ensuring separation of data handling concerns from other layers.

By modeling the view in Postman and implementing a persistence layer using Spring Data JPA, the project shows extensibility and extensibility to be coupled with future front-end. This implementation shows that modularity and maintainability are important parts in application development.

8.3. Design patterns

Singleton design pattern

The Singleton pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. This design pattern was used in the *VehicleManagerService* class, which is responsible for managing the CRUD operations of vehicles (add, modify, or delete vehicles from the system). As illustrated in the following figure, the implementation requires having the private static attribute *instance* which will be accessed through a public static method called *getInstance*. Furthermore, the constructor of the *VehicleManagerService* has been encapsulated, so any external component is able to create new instances with the constructor.

```java
@Service
public class VehicleManagerService {
    private final List<Vehicle> vehicles;
    private final List<Long> IDs;

    private static VehicleManagerService instance;

    public static final synchronized VehicleManagerService getInstance() {
        if (instance == null) {
            instance = new VehicleManagerService();
        }

        return instance;
    }

    private VehicleManagerService() {
        this.vehicles = new ArrayList<>();
        this.IDs = new ArrayList<>();
    }
```

Figure 27: Singleton implementation

Advantages

- **Centralized Control:** Allowing the use of one global instance centralizes vehicle management.
- **Memory Efficiency:** Reduces memory usage by preventing the creation of multiple instances.
- **Safe service access**: Access to *VehicleServiceManager* through a singleton method ensure safe access, since all external components would use the same instance.

Decorator design pattern

The Decorator Design Pattern is a structural pattern that dynamically adds new behaviours or responsibilities to an object without altering its structure or modifying the object itself. In the Vehicle Rental System, the Decorator Design Pattern enables dynamic customization of bookings. This approach allows additional features, such as GPS, insurance, and vouchers, to be added to bookings without altering the core logic of the booking.

The system components that implement the Decorator Pattern are as follows:

1. **Component:** The base interface or abstract class that defines the core functionality.
2. **Concrete Component:** Implements the base functionality defined in the component. In the Vehicle Rental System, the booking class services are both the Component and the Concrete Component of a booking.

```java
public class Booking {
    @ManyToOne
    @JoinColumn(name = "account_id")
    private final Account account;

    @OneToOne
    @JoinColumn(name = "vehicle_id")
    private final Vehicle vehicle;

    @Id
    private final UUID bookingId;
    private final double price;
    private final int numberOfRentingDays;
    private boolean isAuthenticated;

    @ElementCollection
    @CollectionTable(name = "booking_decorators", joinColumns = @JoinColumn(name = "booking_id"))
    protected List<Customization> decorators;

    protected Booking(UUID bookingId, Account account, Vehicle vehicle, int numberOfRentingDays) {
        this.account = account;
        this.vehicle = vehicle;
        this.bookingId = bookingId;
        this.isAuthenticated = false;
        this.numberOfRentingDays = numberOfRentingDays;
        this.price = this.vehicle.getRentingCost(numberOfRentingDays);
        this.decorators = new ArrayList<>();
    }
```

Figure 28: Booking Class

3. **Decorator:** An abstract class that extends the component and adds additional behaviours. The BookingDecorator class is the abstract Decorator class for all booking customizations.

```java
public abstract class BookingDecorator extends Booking {
    @ManyToOne
    private final Booking booking;

    public BookingDecorator(Booking booking) {
        super(
            booking.getBookingId(),
            booking.getAccount(),
            booking.getVehicle(),
            booking.getNumberOfRentingDays(),
            booking.getIsAuthenticated(),
            booking.getDecorators(),
            booking.getPrice()
        );
        this.booking = booking;
    }

    public BookingDecorator() {
        Booking booking = new Booking();
        this.booking = booking;
    }
}
```

Figure 29: Booking Decorator class

4. **Concrete Decorators:** Specific decorator implementations that modify or extend the component's behaviour. GPSBookingDecorator, InsuranceBookingDecorator, VoucherBookingDecorator are some of the concrete decorators that can be used to customize a booking.

```
public class GPSBookingDecorator extends BookingDecorator {
    public GPSBookingDecorator(Booking booking) {
        super(booking);
        this.decorators.add(Customization.GPS);
    }

    public GPSBookingDecorator() {
        super();
        this.decorators.add(Customization.GPS);
    }

    public double getPrice() {
        return super.getPrice() + 10;
    }
}
```

Figure 30: GPS Booking Decorator class

```
public class InsuranceBookingDecorator extends BookingDecorator{
    public InsuranceBookingDecorator(Booking booking) {
        super(booking);
        this.decorators.add(Customization.INSURANCE);
    }

    public InsuranceBookingDecorator() {
        super();
        this.decorators.add(Customization.INSURANCE);
    }

    public double getPrice() {
        return super.getPrice() + super.getNumberOfRentingDays() * 100 ;
    }
}
```

Figure 31: Insurance Booking Decorator class

```java
public class VoucherBookingDecorator  extends BookingDecorator{
    public VoucherBookingDecorator(Booking booking) {
        super(booking);
        this.decorators.add(Customization.VOUCHER);
    }

    public VoucherBookingDecorator() {
        super();
        this.decorators.add(Customization.VOUCHER);
    }

    public double getPrice() {
        return super.getPrice() - 10;
    }
}
```

Figure 32: Voucher Booking Decorator class

Advantages

- **Dynamic:** It is possible to add multiple combinations of customizations to the booking.
- **Single Responsibility Principle:** Divides the behaviour of booking customizations into its classes and they can be combined using composition

Observer design pattern

The Observer Pattern is a behavioural design pattern that establishes a one-to-many dependency between objects, ensuring that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. In the Vehicle Rental System, this pattern was employed to associate a mechanic with a vehicle and inform the mechanic when the vehicle's state changed (e.g., requiring maintenance or being repaired). This approach enhances scalability, maintainability, and flexibility in handling state-dependent operations.

The system components that implement the Observer Pattern are as follows:

1. **Interface for Observer:** The `Observer` interface specifies the format for all observers. Each observer implements the `updateObserver` method, which defines how the observer responds to state changes in the subject.

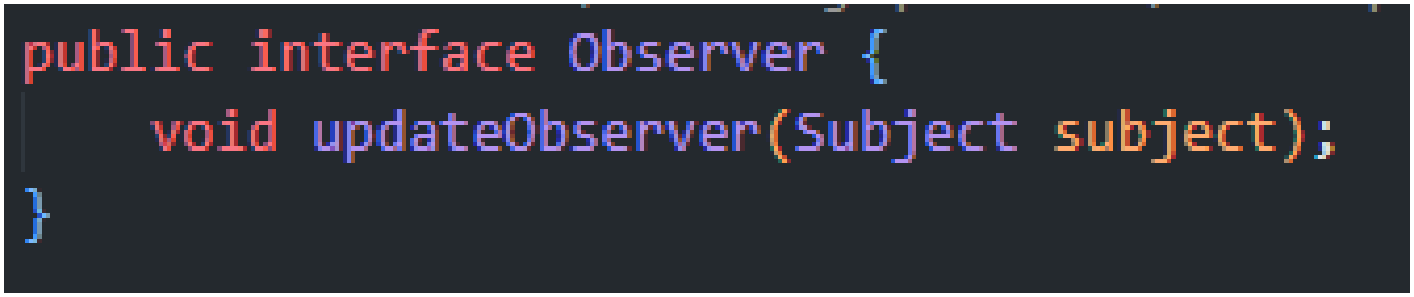


Figure 33: Observer Interface - Method Declaration

2. **Interface for Subject:** The `Subject` interface specifies methods for attaching, detaching, and notifying observers. This interface is implemented by the `Vehicle` class to manage its observers.

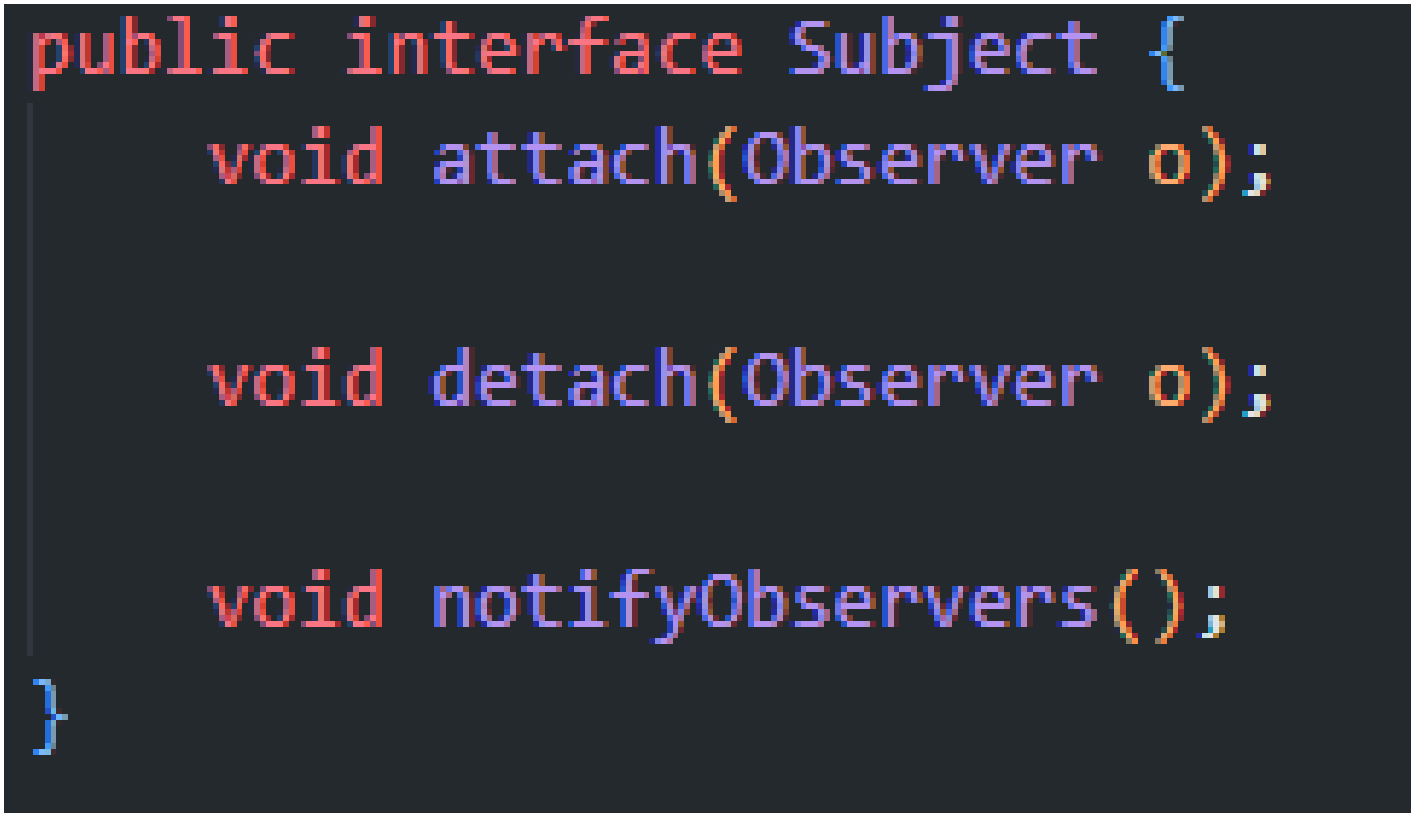


Figure 34: Subject Interface - Method Declaration

3. **Concrete Observer: Mechanic** The `Mechanic` class implements the `Observer` interface. It responds to state changes in vehicles, performing actions such as servicing or repairing them.

```java
public class Mechanic implements Observer {
    private final String name;
    private final VehicleRepository vehicleRepository;

    // Constructor
    public Mechanic(String name, VehicleRepository vehicleRepository) {
        this.name = name;
        this.vehicleRepository = vehicleRepository;
    }

    // Getter for the mechanic's name
    public String getName() {
        return name;
    }

    // Observer method implementation
    @Override
    public void updateObserver(Subject subject) {
        System.out.println("Mechanic " + name + " has been notified of a vehicle state change.");
    }       You, 42 minutes ago • Uncommitted changes
```

Figure 35: Mechanic Class - Observer Implementation

```java
// Assign a mechanic as an observer to a vehicle
public void assignToVehicle(Vehicle v) {
    if (v != null && v.getState().check(className:AvailableVehicleState.class)) {
        v.updateState(new InMaintenanceVehicleState()); // Mark vehicle as in maintenance
        v.attach(this); // Attach this mechanic as an observer
        vehicleRepository.save(v); // Persist the updated vehicle
        System.out.println("Vehicle ID: " + v.getID() + " assigned to Mechanic: " + name);
    } else {
        System.out.println(x:"Vehicle is either null or not available for maintenance.");
    }
}

// Fix a damaged vehicle
public void fixVehicle(Vehicle v) {
    if (v != null && v.getState().check(className:DamagedVehicleState.class)) {
        System.out.println("Mechanic " + name + " is fixing Vehicle ID: " + v.getID());
        v.updateState(new AvailableVehicleState()); // Mark vehicle as available after fixing
        vehicleRepository.save(v); // Persist the updated vehicle
    } else {
        System.out.println(x:"Vehicle is either not damaged or invalid.");
    }
}
```

Figure 36: Mechanic Class - Assign and Fix Methods

4. **Concrete Subject: Vehicle** The `Vehicle` class implements the `Subject` interface. It maintains a list of observers (mechanics) and notifies them when its state changes. For instance, when a vehicle is marked as `In Maintenance`, all subscribed mechanics are informed.

```java
@Override
public void detach(Observer o) {
    if (this.observers.contains(o)) {
        this.observers.remove(o);
    }
}

@Override
public void notifyObservers() {
    for (Observer observer : this.observers) {
        observer.updateObserver(this);
    }
}
```

Figure 37: Vehicle Class - Observer Management

```java
@Override
public void notifyObservers() {
    for (Observer observer : this.observers) {
        observer.updateObserver(this);
    }
}

public void updateState(VehicleState state) {
    this.vehicleState = state;
}
```

Figure 38: Vehicle Class - Notify and Update State Methods

Advantages

- **Decoupling:** The subject and observers are loosely coupled. The vehicle class is unaware of the specific actions performed by the mechanics.
- **Dynamic Updates:** Observers can dynamically subscribe to or unsubscribe from subjects, enabling flexible monitoring.
- **Extensibility:** New types of observers can be added without modifying the subject's implementation.

State design pattern

The State pattern is a behavioural design pattern that allows an object to change its behaviour when its internal state changes. In other words, it is used to model state-dependent variations in behaviour. In this case, the State pattern was employed for the definition and implementation of the different states of a vehicle: available, rented, damaged, and in maintenance. Each of these states has been associated with a specific class, and each of these classes follows the interface *VehicleState*. The defines the essential operations any state must implement.

Therefore, the system components that implement the State Pattern are as follows:

1. **Interface for VehicleState**

```java
package com.ul.vrs.entity.vehicle.state;

import com.fasterxml.jackson.annotation.JsonSubTypes;
import com.fasterxml.jackson.annotation.JsonTypeInfo;
import com.ul.vrs.entity.vehicle.Vehicle;

@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "type")
@JsonSubTypes({
    @JsonSubTypes.Type(value = AvailableVehicleState.class, name = "Available"),
    @JsonSubTypes.Type(value = DamagedVehicleState.class, name = "Damaged"),
    @JsonSubTypes.Type(value = InMaintenanceVehicleState.class, name = "InMaintenance"),
    @JsonSubTypes.Type(value = ReservedVehicleState.class, name = "Reserved")
})
public interface VehicleState {
    boolean check(Class<? extends VehicleState> className);
    void handleRequest(Vehicle vehicle);
}
```

Figure 39: VehicleState Interface

2. **Concrete VehicleState classes**

```java
package com.ul.vrs.entity.vehicle.state;

import com.ul.vrs.entity.vehicle.Vehicle;
import com.ul.vrs.service.VehicleManagerService;

public class AvailableVehicleState implements VehicleState {
    @Override
    public boolean check(Class<? extends VehicleState> className) {
        return this.getClass().getName().equals(className.getName());
    }

    @Override
    public void handleRequest(Vehicle vehicle) {
        if (vehicle != null) {
            VehicleManagerService.getInstance().updateVehicle(vehicle.getID(), vehicle);
        }
    }

    @Override
    public int hashCode() {
        return 1000;
    }

    @Override
    public boolean equals(Object obj) {
        return obj != null && getClass() == obj.getClass();
    }
}
```

Figure 40: AvailableVehicleState Implementation

```java
package com.ul.vrs.entity.vehicle.state;

import com.ul.vrs.entity.vehicle.Vehicle;
import com.ul.vrs.service.VehicleManagerService;

public class DamagedVehicleState implements VehicleState {
    @Override
    public boolean check(Class<? extends VehicleState> className) {
        return this.getClass().getName().equals(className.getName());
    }

    @Override
    public void handleRequest(Vehicle vehicle) {
        if (vehicle != null) {
            vehicle.notifyObservers();
            VehicleManagerService.getInstance().updateVehicle(vehicle.getID(), vehicle);
        }
    }

    @Override
    public int hashCode() {
        return 1001;
    }

    @Override
    public boolean equals(Object obj) {
        return obj != null && getClass() == obj.getClass();
    }
}
```

Figure 41: DamagedVehicleState Implementation

```java
package com.ul.vrs.entity.vehicle.state;

import com.ul.vrs.entity.vehicle.Vehicle;
import com.ul.vrs.service.VehicleManagerService;

public class InMaintenanceVehicleState implements VehicleState {
    @Override
    public boolean check(Class<? extends VehicleState> className) {
        return this.getClass().getName().equals(className.getName());
    }

    @Override
    public void handleRequest(Vehicle vehicle) {
        if (vehicle != null) {
            vehicle.notifyObservers();
            VehicleManagerService.getInstance().updateVehicle(vehicle.getID(), vehicle);
        }
    }

    @Override
    public int hashCode() {
        return 1002;
    }

    @Override
    public boolean equals(Object obj) {
        return obj != null && getClass() == obj.getClass();
    }
}
```

Figure 42: InMaintenanceVehicleState Implementation

```java
package com.ul.vrs.entity.vehicle.state;

import com.ul.vrs.entity.vehicle.Vehicle;
import com.ul.vrs.service.VehicleManagerService;

public class ReservedVehicleState implements VehicleState {
    @Override
    public boolean check(Class<? extends VehicleState> className) {
        return this.getClass().getName().equals(className.getName());
    }

    @Override
    public void handleRequest(Vehicle vehicle) {
        if (vehicle != null) {
            VehicleManagerService.getInstance().updateVehicle(vehicle.getID(), vehicle);
        }
    }

    @Override
    public int hashCode() {
        return 1003;
    }

    @Override
    public boolean equals(Object obj) {
        return obj != null && getClass() == obj.getClass();
    }
}
```

Figure 43: ReservedVehicleState Implementation

Advantages

- **Extensibility:** New states of the vehicle can be included in the system without affecting the current code. In other words, the Open/Closed principle is followed.
- **Reusability:** States can be reused across multiple vehicle types, as they implement a common interface.
- **Encapsulation:** The implementation of the dynamic state behaviour is encapsulated in separate classes.

Factory design pattern

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. In the Vehicle Rental System, the Abstract Factory Pattern was used to streamline the creation of different types of vehicles (e.g., Cars, Scooters, Trucks, Vans) by encapsulating their instantiation logic within specialized factories. This approach enhanced the main-

tainability, scalability, and flexibility of the application by decoupling vehicle creation from the business logic.

The system components that implement the Abstract Factory Pattern are as follows:

1. **Interface for Vehicle Factory:** The `VehicleFactory` interface defines a contract for creating vehicles. This interface declares the `createVehicle` method, which is implemented by concrete factories to produce specific types of vehicles.

```
public interface VehicleFactory {
    Vehicle createVehicle(Object ... params);
}
```

Figure 44: VehicleFactory Interface Implementation

2. **Concrete Factories:** Each concrete factory implements the `VehicleFactory` interface to create a specific type of vehicle. These factories encapsulate the instantiation details, ensuring that the `createVehicle` method initializes all necessary parameters for the vehicle type.

   (a) **CarFactory:** Responsible for creating `Car` objects.

```
@Component
public class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle(Object... params) {
        return new Car(
            (long) params[0], (String) params[1], (String) params[2],
            (int) params[3], (double) params[4], (Color) params[5],
            (Fuel) params[6], (VehicleState) params[7],
            (int) params[8], (float) params[9]
        );
    }
}
```

Figure 45: CarFactory Implementation

   (b) **ScooterFactory:** Responsible for creating `Scooter` objects.

```java
@Component
public class ScooterFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle(Object... params) {
        return new Scooter(
            (long) params[0], (String) params[1], (String) params[2],
            (int) params[3], (double) params[4], (Color) params[5],
            (Fuel) params[6], (VehicleState) params[7], (boolean) params[8],
            (int) params[9], (int) params[10]
        );
    }
}
```

Figure 46: ScooterFactory Implementation

(c) **TruckFactory:** Responsible for creating `Truck` objects.

```java
@Component
public class TruckFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle(Object... params) {
        return new Truck(
            (long) params[0], (String) params[1], (String) params[2],
            (int) params[3], (double) params[4], (Color) params[5],
            (Fuel) params[6], (VehicleState) params[7],
            (float) params[8], (float) params[9], (int) params[10]
        );
    }
}
```

Figure 47: TruckFactory Implementation

(d) **VanFactory:** Responsible for creating `Van` objects.

```java
@Component
public class VanFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle(Object ... params) {
        return new Van(
            (long) params[0], (String) params[1], (String) params[2],
            (int) params[3], (double) params[4], (Color) params[5],
            (Fuel) params[6], (VehicleState) params[7],
            (float) params[8], (int) params[9]
        );
    }
}
```

Figure 48: VanFactory Implementation

3. **VehicleFactoryMethod:** A utility class that simplifies vehicle creation by acting as a central point for selecting the appropriate factory at runtime based on the vehicle type. This class encapsulates the instantiation logic for all vehicle types, ensuring consistency and reusability.

```java
public class VehicleFactoryMethod {
    // Encapsulate constructor ...
    private VehicleFactoryMethod () { }

    public static final Vehicle createVehicle(String type, Object... params) {
        return switch (type) {
            case "Car" -> new CarFactory().createVehicle(
                    (long) params[0], (String) params[1], (String) params[2],
                    (int) params[3], (double) params[4], (Color) params[5],
                    (Fuel) params[6], (VehicleState)params[7], (int) params[8], (float) params[9]);

            case "Scooter" -> new ScooterFactory().createVehicle(
                    (long) params[0], (String) params[1], (String) params[2],
                    (int) params[3], (double) params[4], (Color) params[5],
                    (Fuel) params[6], (VehicleState) params[7], (boolean) params[8], (int) params[9], (int) params[10]);

            case "Truck" -> new TruckFactory().createVehicle(
                    (long) params[0], (String) params[1], (String) params[2],
                    (int) params[3], (double) params[4], (Color) params[5],
                    (Fuel) params[6], (VehicleState) params[7], (float) params[8], (float) params[9], (int) params[10]);

            case "Van" -> new VanFactory().createVehicle(
                    (long) params[0], (String) params[1], (String) params[2],
                    (int) params[3], (double) params[4], (Color) params[5],
                    (Fuel) params[6], (VehicleState) params[7], (float) params[8], (int) params[9]);

            default -> {
                System.err.println(x:"VehicleFactoryMethod: unknown type");
                yield null;
            }
        };
    }
```

Figure 49: Overview of the VehicleFactoryMethod Class

```java
public static final Vehicle createVehicle(Vehicle vehicle) {
    if (vehicle instanceof Car) {
        Car car = (Car) vehicle;

        return new CarFactory().createVehicle(
            car.getID(), car.getName(), car.getBrandOwner(),
            car.getReleaseYear(), car.getBaseCost(), car.getColor(),
            car.getFuelType(), car.getState(), car.getNumberOfDoors(),
            car.getTrunkCapacity()
        );
    }

    else if (vehicle instanceof Scooter) {
        Scooter scooter = (Scooter) vehicle;

        return new ScooterFactory().createVehicle(
            scooter.getID(), scooter.getName(), scooter.getBrandOwner(),
            scooter.getReleaseYear(), scooter.getBaseCost(), scooter.getColor(),
            scooter.getFuelType(), scooter.getState(), scooter.isHasHelmetIncluded(),
            scooter.getMaxPassengers(), scooter.getRangePerFuelTank()
        );
    }

    else if (vehicle instanceof Truck) {
        Truck truck = (Truck) vehicle;

        return new TruckFactory().createVehicle(
            truck.getID(), truck.getName(), truck.getBrandOwner(),
            truck.getReleaseYear(), truck.getBaseCost(), truck.getColor(),
            truck.getFuelType(), truck.getState(), truck.getPayloadCapacity(),
            truck.getTowingCapacity(), truck.getNumberOfAxles()
        );
    }

    else if (vehicle instanceof Van) {
        Van van = (Van) vehicle;

        return new VanFactory().createVehicle(
            van.getID(), van.getName(), van.getBrandOwner(),
            van.getReleaseYear(), van.getBaseCost(), van.getColor(),
            van.getFuelType(), van.getState(), van.getCargoCapacity(),
            van.getNumberOfSeats()
        );
    }

    return null;
}
```

Figure 50: Vehicle Creation Using the Factory Method

Advantages

- **Decoupling:** Vehicle creation logic is encapsulated in factories, separating it from the main application logic.
- **Scalability:** Adding a new vehicle type requires creating a new factory, without modifying existing factories or the core application logic.
- **Maintainability:** Instantiation logic is centralized, making it easier to update or debug.
- **Flexibility:** The system can easily switch between different types of vehicles by injecting the appropriate factory at runtime.

Command design pattern

The Command pattern is a behavioural design pattern that represents a request as an object, allowing a parameterization of clients with queues, requests and operations or undoable functionality. The Vehicle Rental System was used to abstract request handling (e.g., gate opening/car returning) from service business logic. This approach allows scalability, maintainability and flexibility in performing different commands. The *RentalSystemController* integrates seamlessly with the Command Pattern through the *CommandInvoker* component.

The system components that implement the Command Pattern are as follows:

1. **Interface for Command:** This interface specifies the format of all command objects. Every command implements the execute technique, which sets the action to be executed.

```
// Command Interface
public interface Command {
    void execute();
}
```

Figure 51: Command Interface

2. **Concrete Command Classes:** These classes implement the Command interface and perform specific operations, such as gate opening or vehicle return processing.

```java
// Command to handle returning a car in the rental system.
public class ReturnCarCommand implements Command {

    private RentalSystemService rentalSystemService; // Service to manage rental operations.
    private UUID bookingId; // ID of the booking to return.

    // Initializes the command with the rental service and booking ID.
    public ReturnCarCommand(RentalSystemService rentalSystemService) {
        this.rentalSystemService = rentalSystemService;
    }

    public void setBookingID(UUID bookingId) {
        this.bookingId = bookingId;
    }

    // Executes the return vehicle operation.
    @Override
    public void execute() {
        if (bookingId != null) {
            rentalSystemService.returnVehicle(bookingId);
        }
    }
}
```

Figure 52: Return Command implementation

```java
// Command to handle opening a gate.
public class OpenGateCommand implements Command {

    private GateService gateService; // Service to manage gate operations.

    // Initializes the command with the gate service.
    public OpenGateCommand(GateService gateService) {
        this.gateService = gateService;
    }

    // Executes the open gate operation.
    @Override
    public void execute() {
        gateService.openGate();
    }
}
```

Figure 53: Open Gate Command implementation

3. **Command Invoker:** This component provides a centralized user interface for executing commands. It stores a list of commands and enables them to run dynamically using the keys supplied.

```java
@Component // Mark Invoker as a Spring-managed component
public class CommandInvoker {
    @Autowired
    private RentalSystemService rentalSystemService;

    @Autowired
    private GateService gateService;

    private final Map<String, Command> commands = new HashMap<>();

    // Initialize commands after dependencies are injected
    @PostConstruct
    public void initCommands() {
        commands.put("openGate", new OpenGateCommand(gateService));
        commands.put("returnCar", new ReturnCarCommand(rentalSystemService));
    }

    // Set a dynamic UUID for returnCar command before execution
    public void setBookingID(UUID bookingID) {
        if (bookingID != null) {
            ReturnCarCommand command = (ReturnCarCommand) commands.get("returnCar");
            command.setBookingID(bookingID);
        }
    }

    // Executes a specific command based on the key
    public void executeCommand(String key) {
        Command command = commands.get(key);

        if (command == null) {
            System.err.println("Unknown command for key=" + key);
            return;
        }

        command.execute();
    }
}
```

Figure 54: Command Invoker implementation

4. **Service Layer:** Spring-managed services contain the actual business logic for operations. The services are injected into concrete command classes to delegate task execution.

Advantages

- **Decoupling:** The invoker does not need to know the logic contained within the commands. It handles commands through its interface alone.
- **Extensibility:** New commands can be added to the system without affecting the current code.
- **Reusability:** Commands encapsulate reusable operations so they can be called in different contexts without duplicating logic.

Integration with Spring: The implementation used Spring's dependency injection (`@Autowired`).

```
// Return vehicle and open gate - http://localhost:8080/api/renting/return_vehicle_and_open_gate/{id}
@PutMapping("/return_vehicle_and_open_gate/{id}")
public ResponseEntity<String> returnVehicleAndOpenGate(@PathVariable UUID id) {
    // Dynamically set the returnCar command with the current bookingId
    invoker.setBookingID(id);

    // Execute commands
    invoker.executeCommand("openGate");
    invoker.executeCommand("returnCar");

    return ResponseEntity.ok("Vehicle of Booking (ID=" + id + ") has been returned and gate opened.");
}
```

Figure 55: SpringBoot Command implementation

Strategy design pattern

The Strategy Design Pattern is a behavioural design pattern that enables a class's behaviour to be selected at runtime by defining a family of algorithms, encapsulating each one, and making them interchangeable. In the Vehicle Rental System, the payment system is designed using the Strategy Design Pattern to allow the user to pay using different payment methods.

The system components that implement the Strategy Pattern are as follows:

1. **Strategy Interface:** Defines a common interface for all supported strategies. Each strategy encapsulates the logic for processing a specific payment method.

```java
package com.ul.vrs.entity.booking.payment.strategy;

import com.ul.vrs.entity.booking.payment.Payment;

public abstract class PaymentStrategy {
    private Payment payment;

    public PaymentStrategy(Payment payment) {
        this.payment = payment;
    }

    public Payment getPayment() {
        return payment;
    }

    public abstract boolean pay(double amount);
}
```

Figure 56: Payment Strategy Interface

2. **Concrete Strategy Classes:** These classes implement the Strategy interface and perform specific payment operations.

```java
package com.ul.vrs.entity.booking.payment.strategy;

import com.ul.vrs.entity.booking.payment.Payment;

public class CreditCardPaymentStrategy extends PaymentStrategy {
    public CreditCardPaymentStrategy(Payment payment) {
        super(payment);
    }

    public boolean pay(double amount) {
        Payment payment = getPayment();

        if (payment.getAmount() >= amount) {
            payment.setAmount(payment.getAmount() - amount);
            return true;
        }

        return false;
    }
}
```

Figure 57: Credit Card Payment

3. **Context Class:** Maintains a reference to a strategy object and allows selecting payment strategy dynamically at runtime.

```java
import com.ul.vrs.entity.booking.payment.strategy.CreditCardPaymentStrategy;
import com.ul.vrs.entity.booking.payment.strategy.PaymentStrategy;

public class PaymentRequest {
    private PaymentMethod method;
    private Payment payment;

    public PaymentRequest(PaymentMethod method, Payment payment) {
        this.method = method;
        this.payment = payment;
    }

    public PaymentStrategy getPaymentStrategy() {
        PaymentStrategy strategy = null;

        switch (method) {
            case CREDITCARD -> strategy = new CreditCardPaymentStrategy(payment);
            case APPLEPAY -> strategy = new ApplePayPaymentStrategy(payment);
            default -> System.err.println("Unkown payment strategy: " + method.name());
        }

        return strategy;
    }
}
```

Figure 58: Payment Request context class

Advantages

- **Extensibility:** New payment methods can be added by creating new strategy classes without modifying the existing code.
- **Scalability:** Encapsulation of payment logic in separate classes makes the system more modular and easier to maintain.
- **Reusability:** Common functionality is centralized in the abstract PaymentStrategy class, reducing code duplication.

Interceptor design pattern

The Interceptor Design Pattern is a behavioral design pattern that enables the modular addition of pre- and post-request or action modifiers. In this project, the pattern was used to deal with cross-cutting issues including state, logging, and customization of the vehicle rental system.

The system uses several types of interceptors that are intended for precondition validations and logging. Below

is a representation of how interceptors are integrated into the workflow:

```java
public interface Interceptor {
    //Method to execute before the target action.

    void beforeAction(String action, Object target);

    // Method to execute after the target action.

    void afterAction(String action, Object target);
}
```

Figure 59: Main Interface

Each Vehicle entity operates inside the Customer class scenario. Interceptors constantly modify actions both before and after what are considered critical activities such as the booking and customization steps.

GPSInterceptor Implementation

The `GPSInterceptor` validates and logs the usage of GPS functionality in a vehicle. This implementation ensures that GPS operations are not executed when the vehicle is in an incorrect state, thereby preventing performance issues or errors.

```java
public class GPSInterceptor implements Interceptor {

    @Override
    public void beforeAction(String action, Object target) {
        if ("applyGPS".equals(action) && target instanceof Vehicle vehicle) {
            if (!(vehicle.getState() instanceof ReservedVehicleState)) {
                throw new IllegalStateException("GPS can only be applied to RESERVED vehicles.");
            }
            System.out.println("Preparing to add GPS to vehicle ID: " + vehicle.getID());
        }
    }

    @Override
    public void afterAction(String action, Object target) {
        if ("applyGPS".equals(action) && target instanceof Vehicle vehicle) {
            System.out.println("GPS successfully added to vehicle ID: " + vehicle.getID());
        }
    }
}
```

Figure 60: GPSInterceptor Implementation in customer class

```java
public class VoucherInterceptor implements Interceptor {
    @Override
    public void beforeAction(String action, Object target) {
        if ("applyVoucher".equals(action) && target instanceof Vehicle vehicle) {
            if (!(vehicle.getState() instanceof ReservedVehicleState)) {
                throw new IllegalStateException("Voucher can only be applied to RESERVED vehicles.");
            }
            System.out.println("Preparing to apply voucher to vehicle ID: " + vehicle.getID());
        }
    }

    @Override
    public void afterAction(String action, Object target) {
        if ("applyVoucher".equals(action) && target instanceof Vehicle vehicle) {
            System.out.println("Voucher successfully applied to vehicle ID: " + vehicle.getID());
        }
    }
}
```

Figure 61: VoucherInterceptor Implementation in customer class

```java
public class InsuranceInterceptor implements Interceptor {
    @Override
    public void beforeAction(String action, Object target) {
        if ("applyInsurance".equals(action) && target instanceof Vehicle vehicle) {
            if (!(vehicle.getState() instanceof ReservedVehicleState)) {
                throw new IllegalStateException("Insurance can only be applied to RESERVED vehicles.");
            }
            System.out.println("Preparing to apply insurance to vehicle ID: " + vehicle.getID());
        }
    }

    @Override
    public void afterAction(String action, Object target) {
        if ("applyInsurance".equals(action) && target instanceof Vehicle vehicle) {
            System.out.println("Insurance successfully applied to vehicle ID: " + vehicle.getID());
        }
    }
}
```

Figure 62: InsuranceInterceptor Implementation in customer class

```java
public class LoggingInterceptor implements Interceptor {
    @Override
    public void beforeAction(String action, Object target) {
        if (action == null || target == null) {
            System.out.println("Logging BEFORE action: Invalid input. Action or Target is null.");
            return;
        }
        System.out.println("Logging BEFORE action: " + action + " on target: " + target.getClass().getSimpleName());
    }

    @Override
    public void afterAction(String action, Object target) {
        if (action == null || target == null) {
            System.out.println("Logging AFTER action: Invalid input. Action or Target is null.");
            return;
        }
        System.out.println("Logging AFTER action: " + action + " on target: " + target.getClass().getSimpleName());
    }
}
```

Figure 63: LoggingInterceptor Implementation in customer class

This interceptor enforces that all preconditions (`beforeAction`) are satisfied before the execution of the operation. Additionally, it confirms the successful completion of the operation through `afterAction`, ensuring robust and error-free functionality.

```java
private void runBeforeInterceptors(String action, Object target) {
    for (Interceptor interceptor : interceptors) {
        interceptor.beforeAction(action, target);
    }
}

private void runAfterInterceptors(String action, Object target) {
    for (Interceptor interceptor : interceptors) {
        interceptor.afterAction(action, target);
    }
}
```

Figure 64: GPSInterceptor Implementation base code

Dynamic Interceptor Integration

The `Customer` class dynamically binds interceptors into the class, triggering actions before and after important operations such as booking and vehicle customization. This dynamic processing promotes modularity and reusability in the design of procedures.

```java
public Booking bookVehicle(Vehicle vehicle, int numberOfRentingDays) {
    runBeforeInterceptors("bookVehicle", vehicle);

    if (vehicle.getState() instanceof AvailableVehicleState) {
        System.out.println("Booking vehicle: " + vehicle.getName() + ", ID: " + vehicle.getID());
        vehicle.updateState(new ReservedVehicleState());
        Booking booking = new Booking(this, vehicle, 0);
        runAfterInterceptors("bookVehicle", booking);
        return booking;
    }

    System.out.println("Cannot book vehicle: " + vehicle.getName() + " is not available.");
    runAfterInterceptors("bookVehicle", vehicle);
    return null;
}
```

Figure 65: Dynamic Interceptor Binding in the Customer Class

Due to this dynamic integration, rule enforcement or operation logging can be achieved through interceptors, without the need to hard-code these functionalities directly into the `Customer` class.

Advantages

- **Separation of Concerns:** Dependencies are separated to maintain core functionalities aligned with business priorities.
- **Dynamic Extensibility:** Interceptors can be attached and removed without modifying existing code, enhancing scalability.
- **Precondition Enforcement:** Validates operations at the state level, preventing errors and ensuring system integrity.
- **Logging and Transparency:** Provides detailed logs for debugging and monitoring through interceptors like `LoggingInterceptor`.

8.4. Automation Testing

Regarding testing, *JUnit* and *Postman* were the main tools used to assess the software's correctness. The former was employed to test the individual methods defined in the classes, while the latter was used to validate the functionality of the endpoints. In this project, the testing process involves creating the unit test cases with *JUnit* to then define automatic *Postman* tests using the *Postman API*. Furthermore, software metrics were used with tools such as *JaCoCo*, which is a code coverage tool for Java environments, and *Sonarqube*, a tool that allows the monitoring of various software metrics. The *"Added Value"* section provides a more detailed explanation of the testing process, where software metrics and CI/CD tools are discussed.

The following figure shows a sample of a *JUnit* test case for the *RentalSystemService* class.

```java
@Test
public void testMakeBooking_Success() {
    // Test for successfully making a booking
    Vehicle vehicle = mockVehicles.get(index:0); // Select the first vehicle
    UUID bookingId = rentalSystemService.makeBooking(mockCustomer, vehicle, numberOfRentingDays:1); // Create a booking

    assertNotNull(bookingId, message:"Booking ID should not be null"); // Booking ID must not be null
    Optional<Booking> booking = rentalSystemService.getBookingById(bookingId);
    assertTrue(booking.isPresent(), message:"Booking should exist"); // Booking should exist
    assertEquals(vehicle, booking.get().getVehicle(), message:"The vehicle in the booking should match the one provided");
    assertEquals(new ReservedVehicleState(), vehicle.getState(), message:"Vehicle should be RESERVED after booking");
}

@Test
public void testMakeBooking_NullVehicle() {
    // Test for attempting to book a null vehicle
    UUID bookingId = rentalSystemService.makeBooking(mockCustomer, vehicle:null, numberOfRentingDays:0);
    assertNull(bookingId, message:"Booking ID should be null when vehicle is null"); // Booking ID must be null
}
```

Figure 66: Sample *JUnit* test

8.5. Version Control

The development of the vehicle rental system was carried out using a version control tool called Git. The code was published in a public GitHub repository where each team member could contribute to the project (see https://github.com/mallyaaniruddh/software-design-project).

The following figures show the graphical representations of the GitHub actions, the commit activity of the repository, and its branching and merging structure.

Commit Trends

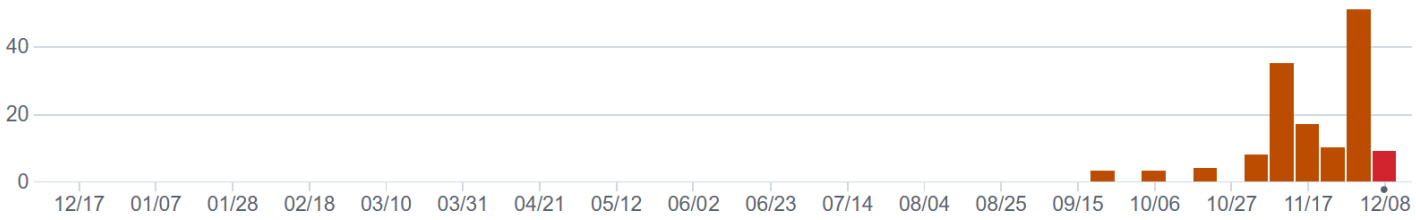


Figure 67: Commit Activity Over Time

Repository Network

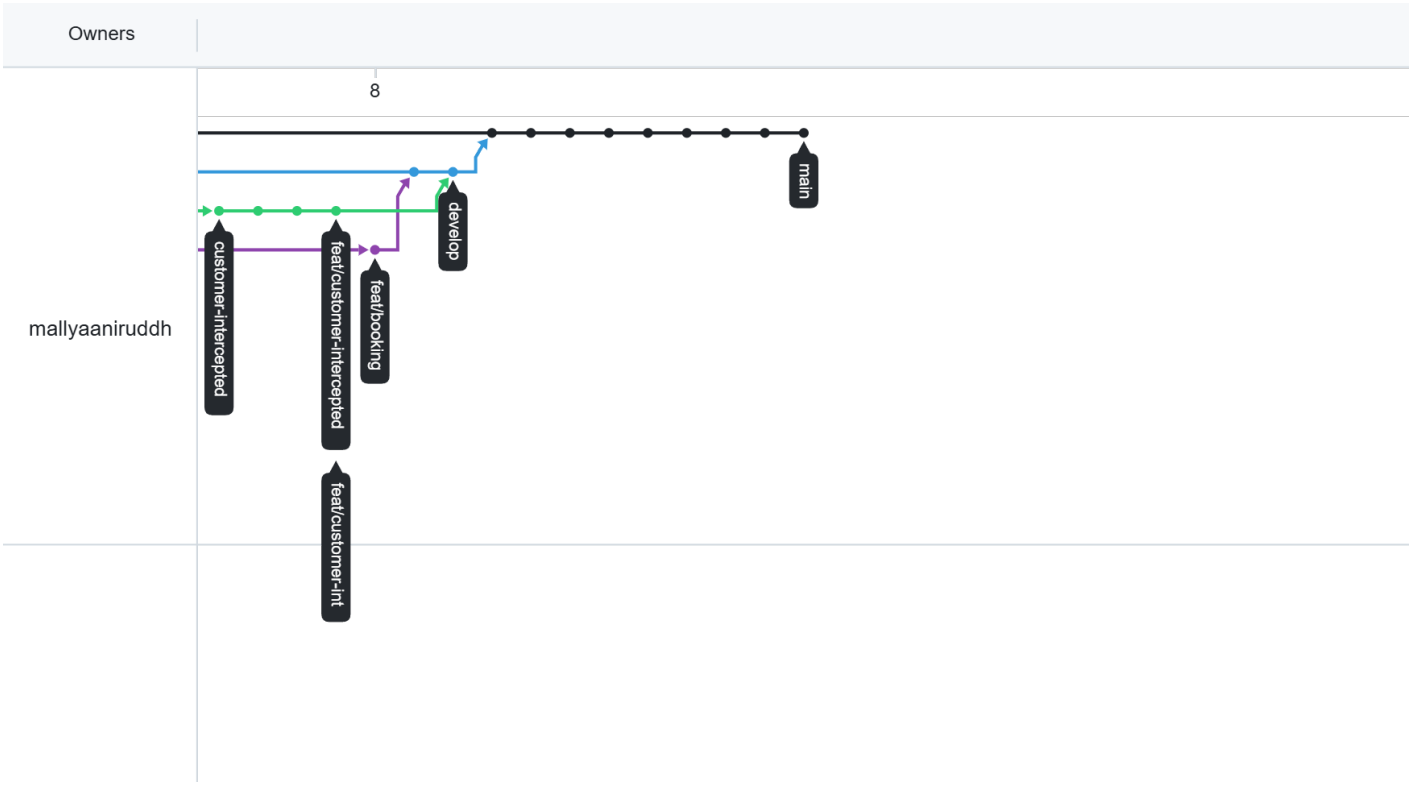


Figure 68: GitHub Repository Network

Action Usage Metrics

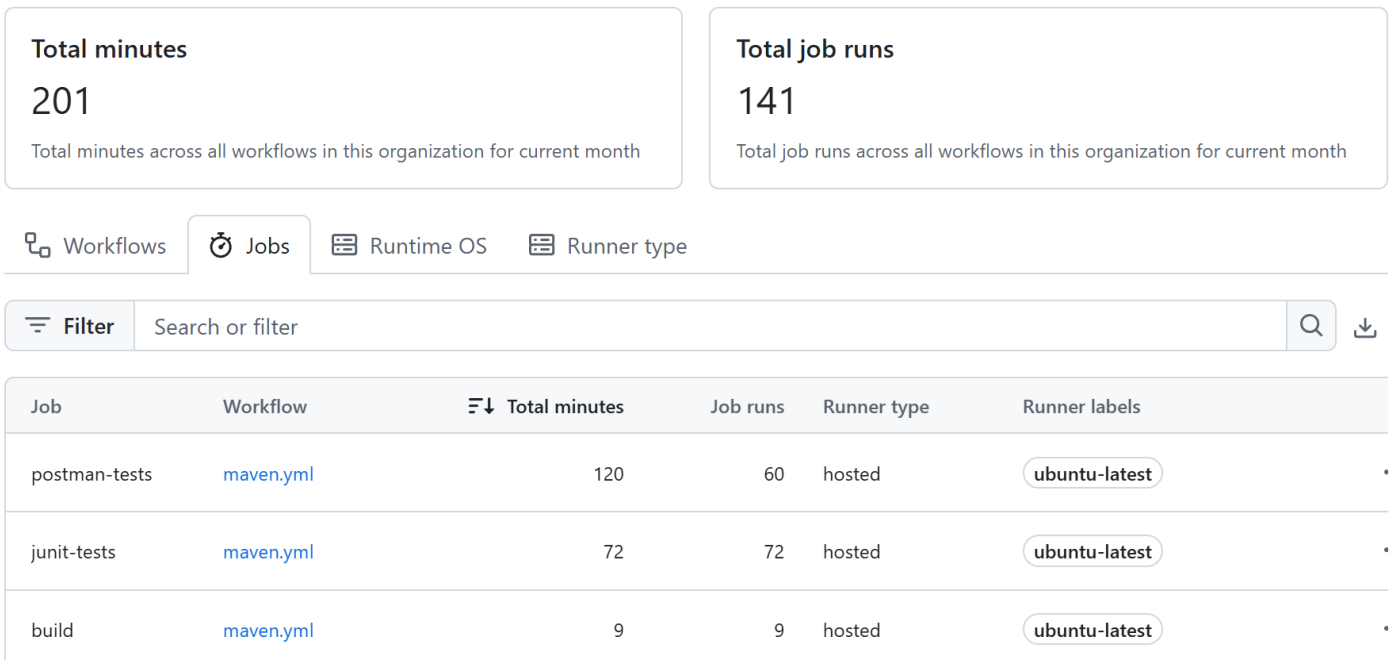


Figure 69: Distribution of GitHub Actions by Type

Action Performance Metrics

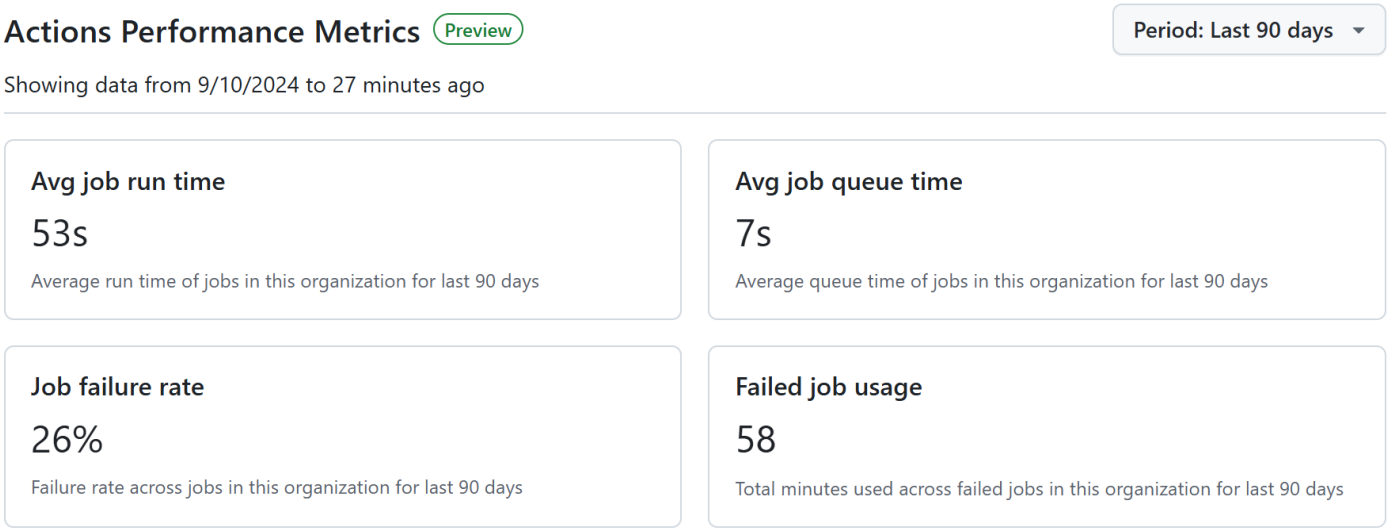


Figure 70: Distribution of GitHub Actions by Type

8.6. Added Value

Database support through ORM

The application uses Spring Boot with JPA (Java Persistence API) and Hibernate as the ORM framework to manage database interactions. ORM is implemented in the project with examples from the codebase.

Entity Mapping

Entities in the system represent database tables. Each entity is annotated with `@Entity` to map it to a corresponding table in the database. For example the Account class:

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Account {
    @Id
    private String username;
    private String password;
```

Figure 71: Account Class Entity

Features implemented:

- The `@Id` annotation marks username as the primary key.
- The `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)` ensures all subclasses of Account share the same table.

- Fields are automatically mapped to table columns.

Relationships

Relationships between entities are mapped using JPA annotations such as `@OneToOne`, `@ManyToOne` and `@ElementColle`
For example the Booking class:

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "booking_type")
@DiscriminatorValue("BASE")
public class Booking {
    @ManyToOne
    @JoinColumn(name = "account_id")
    private final Account account;

    @OneToOne
    @JoinColumn(name = "vehicle_id")
    private final Vehicle vehicle;

    @Id
    private final UUID bookingId;
    private final double price;
    private final int numberOfRentingDays;
    private boolean isAuthenticated;

    @ElementCollection
    @CollectionTable(name = "booking_decorators", joinColumns = @JoinColumn(name = "booking_id"))
    protected List<Customization> decorators;
```

Figure 72: Booking Class Relationships

Features implemented:

- `@ManyToOne` and `@OneToOne` create relationships between Booking and Account/Vehicle.
- `@ElementCollection` maps a list of Customization objects to a separate table (`booking_decorators`).
- `@DiscriminatorColumn` and inheritance allow flexibility in extending booking types.

Repositories

Repositories simplify database access. Interfaces extend `JpaRepository` to provide CRUD operations.

```java
@Repository
public interface BookingRepository extends JpaRepository<Booking, UUID> {

}
```

Figure 73: Booking Repository Interface

Features implemented:

- `JpaRepository` provides built-in methods like `save`, `findById` and `delete`.

CI/CD Pipeline in the Vehicle Rental System

This project incorporates CI/CD pipeline to automate testing and deployment processes, which enhances productivity, and reliability and ensures consistent application quality.

1. Continuous Integration and Testing

- **GitHub Workflows** automate the testing process whenever changes are pushed to the develop or automation branches or when a Pull Request is made to the develop branch.
- **Automated Springboot builds** checks if added code is error-free and can build/compile correctly.
- **Automated JUnit tests** ensure the correctness of business logic at the unit level.
- **Automated Postman tests** validate API endpoints using the Newman CLI.

2. Key Advantages

- **Consistency:** Automated tests run reliably across different environments.
- **Time-saving:** Manual checking is minimized during testing and deployment.
- **Early Detection:** Bugs are caught early through testing in the pipeline.

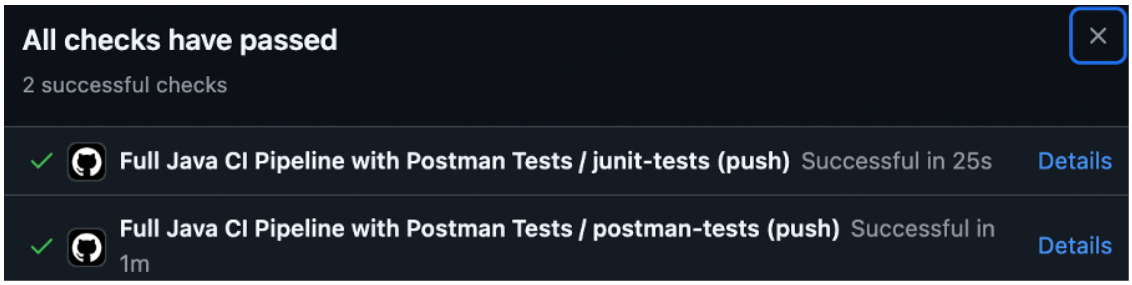
Code Snippets: CI/CD Pipeline Implementation

GitHub Workflow

GitHub workflow has been set up in the project using the following YAML configuration file.

```yaml
name: Full Java CI Pipeline with Postman Tests

on:
  push:
    branches:
      - "develop"
      - "automation"
  pull_request:
    branches:
      - "develop"
      - "automation"

jobs:
  # Job to run Maven tests
  junit-tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      # Set up JDK 17 for Maven and JUnit tests
      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'
          distribution: 'temurin'
          cache: maven

      # Run JUnit tests with Maven
      - name: Run JUnit tests with Maven
        run: mvn -B test --file pom.xml
```

```yaml
# Job to build the Spring Boot app and run Postman tests
postman-tests:
  runs-on: ubuntu-latest
  needs: junit-tests  # Ensure that JUnit tests run first

  steps:
    # Checkout the repository
    - name: Checkout repository
      uses: actions/checkout@v4

    # Set up JDK 17 for Spring Boot
    - name: Set up JDK 17 for Spring Boot
      uses: actions/setup-java@v4
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven

    # Build the Spring Boot application with Maven
    - name: Build Spring Boot application with Maven
      run: mvn -B package --file pom.xml

    # Start the Spring Boot application
    - name: Start Spring Boot application
      run: |
        nohup java -jar target/*.jar &
        sleep 30  # Wait for the Spring Boot app to start

    # Set up Node.js for Newman (Postman test runner)
    - name: Set up Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '16'

    # Install Newman
    - name: Install Newman
      run: npm install -g newman

    # Run Postman collection tests
    - name: Run Postman Collection Tests
      run: |
        newman run postman/VRS.postman_collection.json
```



This project integrates CI/CD workflows, JUnit tests and Postman API tests to show quality assurance and modern software engineering practices. Automated testing enables early bug detection and maintains codebase reliability while the CI/CD pipeline automates repetitive tasks increasing productivity. This approach lays the foundation for scalability and continuous delivery.

JaCoCo

JaCoCo is a code coverage tool for Java projects that helps developers see how much code is tested by automated unit tests. Using this tool in the project ensures that key parts of the application are properly tested, leading to a more robust and maintainable application. JaCoCo was integrated as a Maven plugin by modifying the *pom.xml* file, ensuring that it is executed automatically every time the application is compiled.

As shown in Figure 74, the application's code coverage before implementing unit tests was 20%. This indicates that much of the code remained untested, which also means that there was a higher risk of undetected bugs and

potential issues in those untested areas.

vrs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ul.vrs.entity.vehicle		28%		0%	38	42	59	81	35	39	3	6
com.ul.vrs.entity.vehicle.factory		0%		n/a	9	9	51	51	9	9	5	5
com.ul.vrs.service		26%		0%	28	31	47	57	15	18	0	3
com.ul.vrs.controller		8%		0%	17	19	34	37	11	13	0	2
com.ul.vrs.entity.booking		0%		n/a	16	16	29	29	16	16	6	6
com.ul.vrs		18%		0%	6	8	9	11	5	7	0	2
com.ul.vrs.entity.vehicle.fuel		20%		n/a	5	6	5	6	5	6	2	3
com.ul.vrs.entity.account		66%		n/a	1	2	1	4	1	2	0	1
com.ul.vrs.entity		100%		n/a	0	1	0	2	0	1	0	1
Total	931 of 1,169	20%	44 of 44	0%	120	134	235	278	97	111	16	29

Figure 74: JaCoCo Results before JUnit tests

Once JUnit tests were implemented, the application's overall code coverage increased to 62%. Figure 75 illustrates the improved coverage following the addition of these tests.

vrs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ul.vrs.entity.vehicle.factory		67%		30%	6	18	31	69	1	10	0	5
com.ul.vrs.controller		5%		0%	33	37	70	75	19	23	0	4
com.ul.vrs.service		58%		36%	32	64	67	160	13	39	0	8
com.ul.vrs.security		27%		0%	19	25	38	55	10	16	0	3
com.ul.vrs.entity.account		79%		57%	24	63	38	164	10	42	0	4
com.ul.vrs.entity.vehicle.state		40%		19%	30	46	33	49	8	24	1	6
com.ul.vrs.entity.vehicle		85%		63%	28	78	38	172	7	48	0	5
com.ul.vrs.interceptor		34%		39%	19	31	23	38	3	12	1	4
com.ul.vrs.entity.booking.decorator		56%		n/a	7	12	16	30	7	12	0	5
com.ul.vrs.controller.command		47%		0%	8	12	17	28	5	9	0	3
com.ul.vrs.entity.booking		71%		n/a	1	12	9	38	1	12	0	1
com.ul.vrs.entity.booking.payment		79%		66%	6	15	6	31	5	13	0	5
com.ul.vrs.entity.booking.decorator.factory		46%		25%	3	4	3	7	0	1	0	1
com.ul.vrs.entity.booking.payment.strategy		100%		100%	0	8	0	18	0	6	0	3
com.ul.vrs.entity		100%		n/a	0	1	0	2	0	1	0	1
Total	1,661 of 4,456	62%	191 of 304	37%	216	426	389	936	89	268	2	58

Figure 75: JaCoCo after JUnit tests

SonarQube

SonarQube is a widely-used open-source platform for continuous inspection of code quality. It analyzes source code to identify issues related to bugs, vulnerabilities, code smells, code coverage, and duplications.

An initial SonarQube analysis was run on the application to identify the existing code smells and issues related to maintainability, reliability and security.

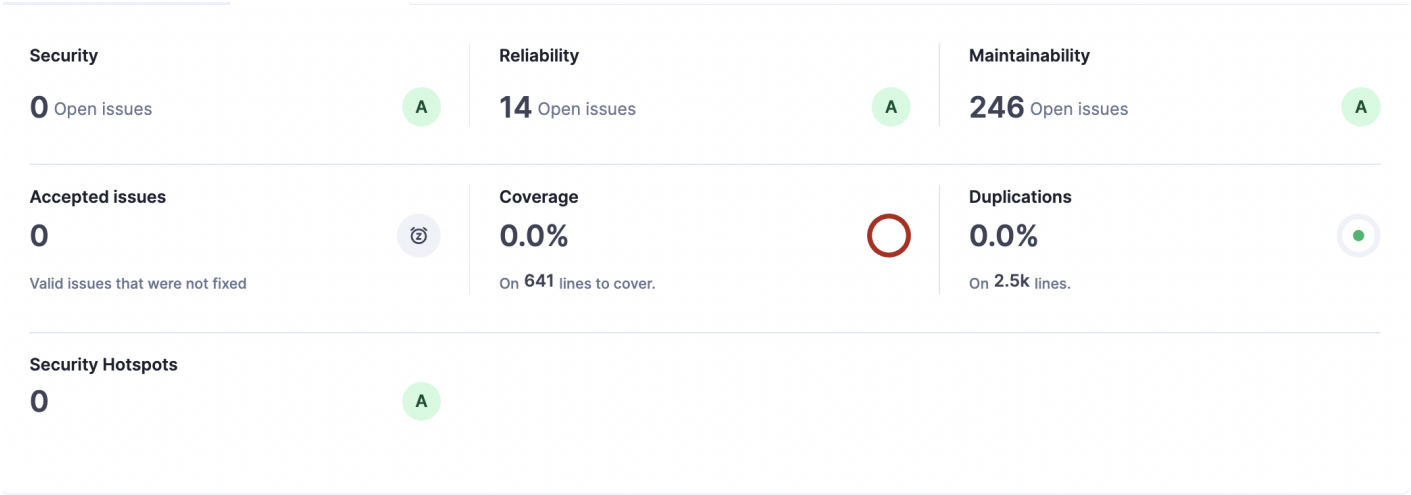


Figure 76: Initial SonarQube Analysis

We noticed there were many issues related to maintainability due to bad code smells such as unused fields, imports and commented out code.

```
14  rohans…
15  ivor20…   import com.ul.vrs.repository.VehicleRepository;
16  rohans…   @Service
17  loseda…   public class VehicleManagerService {
18  loseda…       private final List<Vehicle> vehicles;
19            private long currentId;
```

Remove this unused "currentId" private field.

Figure 77: Bad code smell: Unused field

```
38  ivor20…   // private Customer customer = new Customer(
39            //     "test_username", "test_id", "test_password"
40            // );
```

This block of commented-out lines of code should be removed.

Figure 78: Bad code smell: Commented-out code

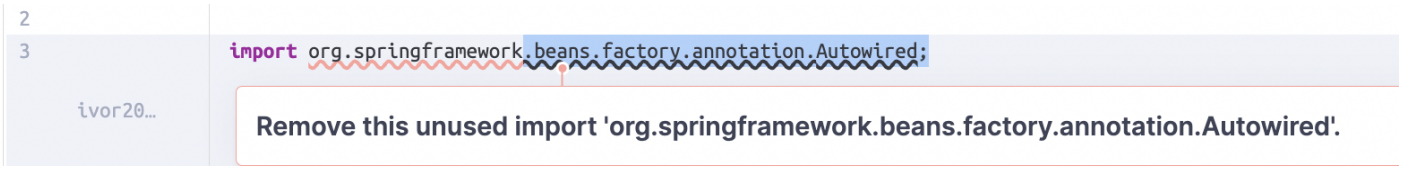


Figure 79: Bad code smell: Unused import

We accepted the suggestions made by SonarQube and fixed some of these bad code smells resulting in a better score for code maintainability for our application.

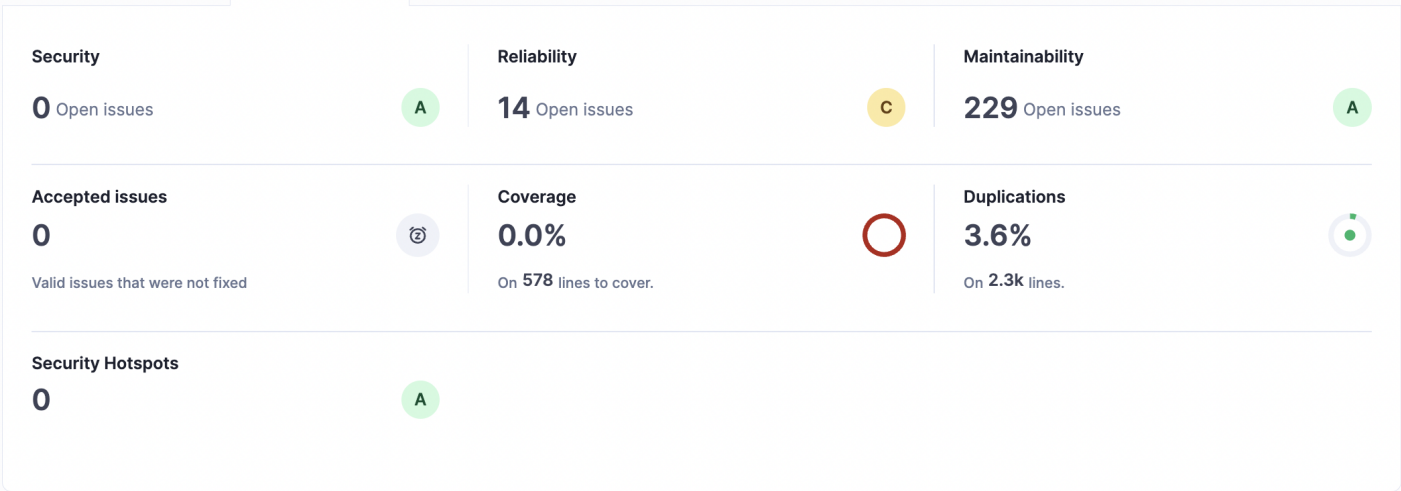


Figure 80: SonarQube Analysis after fixing issues

Martins Metrics

Martin's Metrics are a set of metrics designed to assess the quality of software design, particularly in terms of modularity and dependency management. These metrics help evaluate how well software components are structured and how their dependencies affect maintainability, flexibility, and scalability. To integrate Martins Metrics - we utilized the JDepends java library and JDependsUI to generate a visualisation.

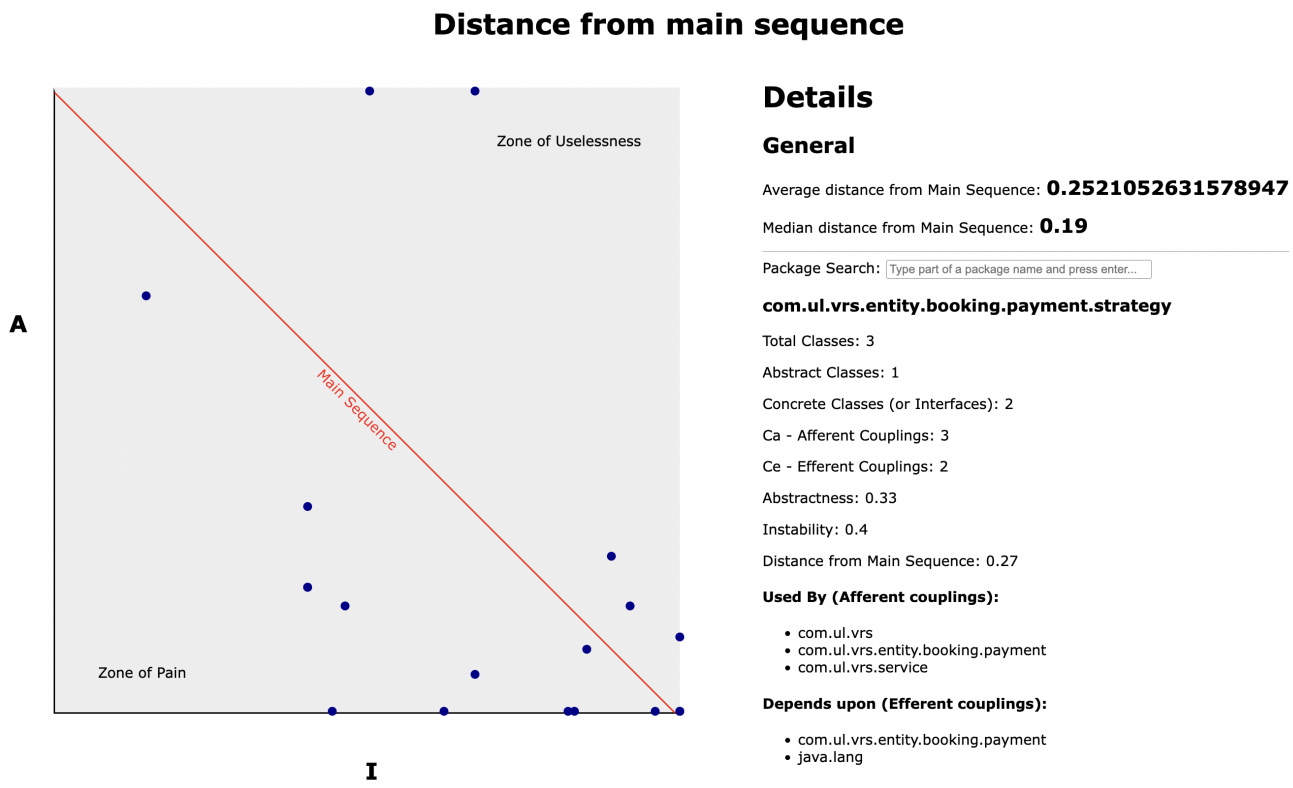


Figure 81: Martins metrics

The visualization plots modules on a graph with Instability (I) on the x-axis and Abstractness (A) on the y-axis, showing their proximity to the ”Main Sequence” (D=0). Modules near the sequence are well-balanced between stability and abstraction, while outliers may require refactoring. This helps us identify overly stable or overly abstract modules for architectural improvement.

The 2 modules on the top close to the ”Zone of Uselessness” are the Repository and Jacoco modules, which is expected as they are external dependencies. The core modules are in good proximity to the main sequence, indicating well-balanced stability and abstraction in the application.

Code analysis through PMD

PMD is a versatile static code analysis tool that supports multiple programming languages. It detects common coding issues such as unused variables, empty catch blocks, and redundant object creation (PMD Development Team 2024). This project integrates PMD with a Maven plugin which enables the detection of common programming flaws during the build process.

As discussed earlier, this tool helps us catch issues early, improving code quality and maintainability. PMD automates analysis, reduces manual work, and keeps our codebase clean and reliable. By integrating PMD, we can continuously monitor code quality as part of the build process, ensuring that new issues are identified promptly before they affect the project. PMD also allows us to enforce project-specific coding standards

through custom rulesets, ensuring consistency and adherence to best practices.

Regarding the plugin configuration, it was essential to modify the *pom.xml* Maven configuration file to include the required dependencies. After this setup, PMD was automatically executed each time the application was compiled.

Finally, to provide evidence supporting the use of PMD in this project, Figure 82 illustrates the violations identified after integrating PMD.

Violations By Priority

Priority 3

com/ul/vrs/entity/Observer.java

Rule	Violation	Line
UnnecessaryModifier	Unnecessary modifier 'public' on method 'updateObserver': the method is declared in an interface type	14

com/ul/vrs/entity/Subject.java

Rule	Violation	Line
UnnecessaryModifier	Unnecessary modifier 'public' on method 'attach': the method is declared in an interface type	16
UnnecessaryModifier	Unnecessary modifier 'public' on method 'detach': the method is declared in an interface type	23
UnnecessaryModifier	Unnecessary modifier 'public' on method 'notifyObservers': the method is declared in an interface type	28

com/ul/vrs/entity/vehicle/fuel/Fuel.java

Rule	Violation	Line
UnnecessaryModifier	Unnecessary modifier 'public' on method 'getCost': the method is declared in an interface type	16

com/ul/vrs/service/RentalSystemService.java

Rule	Violation	Line
CollapsibleIfStatements	This if statement could be combined with its parent	53–59

Figure 82: PMD violations after integration into the project

The following figure shows the updated PMD report once all violations were resolved.

PMD Results

The following document contains the results of PMD 7.7.0.

PMD found no problems in your source code.

Figure 83: PMD Report

Changelog Generator

A changelog is a special file where changes made to a project over time are documented (Lacan 2024). In this project, a Maven plugin (Bjerre 2024) which allows the automatic generation of the changelog has been integrated. This plugin uses the commits of the GitHub repository to customize the changelog file, and it is executed each time the project is compiled.

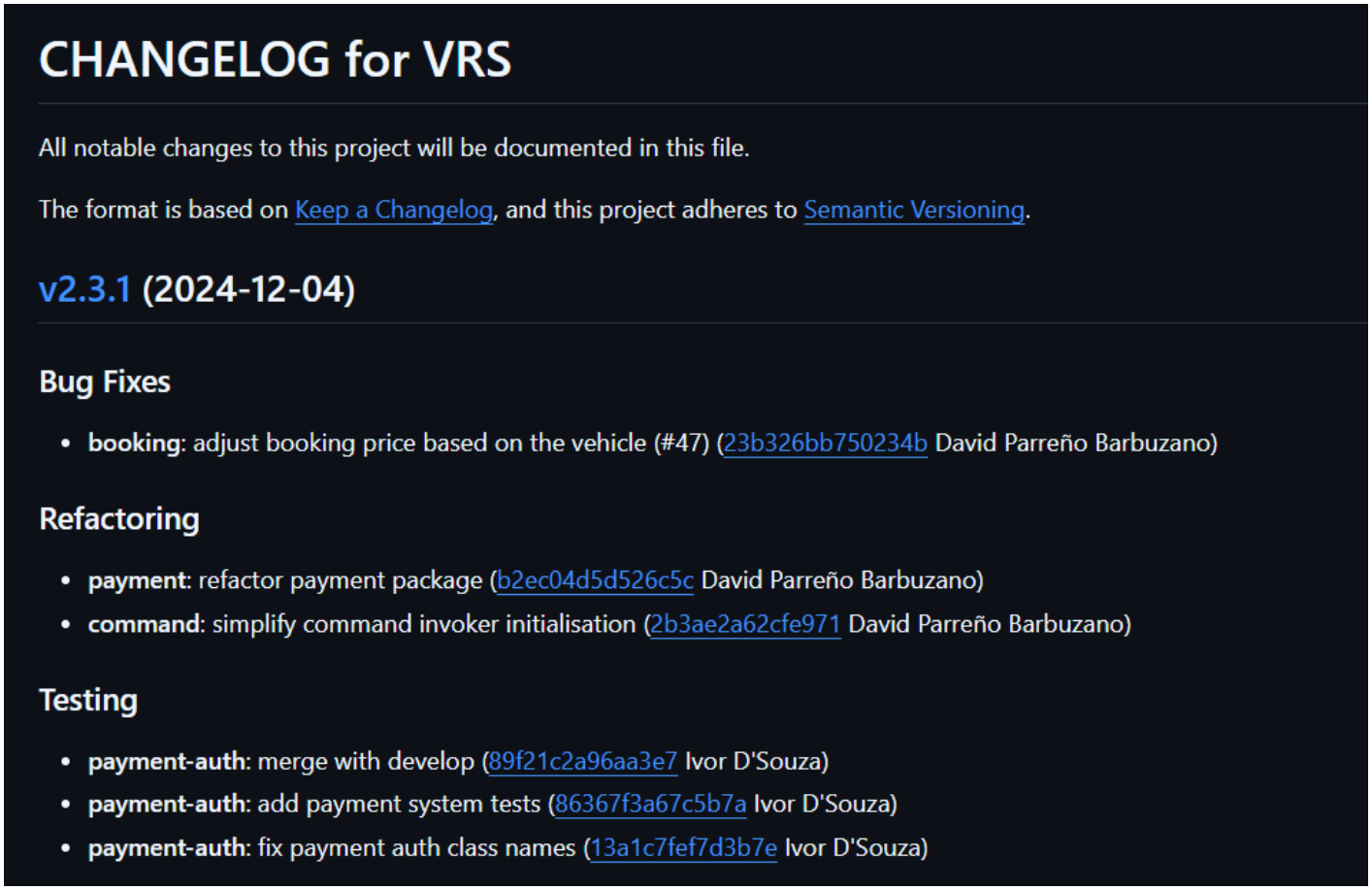


Figure 84: Changelog file

Concerning the configuration of this plugin, three different files were included: *pom.xml*, *changelog.conf.json*, and *changelog-template.mustache*. The *pom.xml* file is the Maven configuration file, where the plugin is both declared and configured, whereas *changelog.conf.json* is used to specify additional customization properties for the plugin and *changelog-template.mustache* defines the structure and format of the generated CHANGELOG.

Rest Architecture Pattern

Method	Path	Description	Body

POST	/users	Create a new user	JSON object with user details
POST	/api/account/signup	Sign up new account	username, password, accountType (form data)
POST	/api/account/login	Log in existing account	username, password (form data)
GET	/api/vehicles	Retrieve all vehicles	None
GET	/api/vehicles/1	Get vehicle by ID	None
POST	/api/vehicles	Add new vehicle	JSON object with vehicle details
PUT	/api/vehicles/1	Update vehicle by ID	JSON object with updated details
DELETE	/api/vehicles/1	Delete vehicle by ID	None
POST	/api/renting/make_booking/2	Make a booking	JSON object with booking details (e.g., numberOfRentingDays)
PUT	/api/renting/customize_booking/{{bookingId}}	Customize booking	JSON object with options (e.g., GPS)
PUT	/api/renting/make_payment/{{bookingId}}	Make payment	JSON object with payment details
PUT	/api/renting/return_vehicle_and_open_gate/{{bookingId}}	Return vehicle and open gate	None
PUT	/api/renting/return_vehicle/{{bookingId}}	Return vehicle	None
DELETE	/api/renting/cancel_booking/{{bookingId}}	Cancel booking	None
GET	/api/renting/list_bookings	List all bookings	None

Table 13: REST API Sheet

Security

We utilized Spring Security to create a robust authentication framework. JSON Web Token (JWT) is a standard for encoding information that may be securely transmitted as a JSON object. A JWT was created on successful login and the same has to be passed as authentication for calling any Vehicle Rental System APIs. In addition, BCryptPasswordEncoder was utilized to hash passwords stored in the DB.

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {
    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Lazy
    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {
        final String authorizationHeader = request.getHeader(name:"Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith(prefix:"Bearer ")) {
            jwt = authorizationHeader.substring(beginIndex:7);
            username = jwtTokenUtil.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

            if (jwtTokenUtil.validateToken(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken authToken =
                    new UsernamePasswordAuthenticationToken(userDetails, credentials:null, userDetails.getAuthorities());
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        chain.doFilter(request, response);
    }
}
```

Figure 85: Request filter for authenticating user

```java
@Component
public class JwtTokenUtil {
    @Value("${jwt.secret}")
    private String SECRET_KEY;

    @Value("${jwt.expiration}")
    private Long EXPIRATION_TIME;

    private Key getSigningKey() {
        return Keys.hmacShaKeyFor(SECRET_KEY.getBytes());
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userDetails.getUsername());
    }

    private String createToken(Map<String, Object> claims, String subject) {
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(subject)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    public String extractUsername(String token) {
        if (token != null && token.startsWith(prefix:"Bearer ")) {
            token = token.substring(beginIndex:7);
        }
        return extractClaim(token, Claims::getSubject);
    }
```

Figure 86: Token util class containing JWT related functions

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .requestMatchers(...patterns:"/api/account/login", "/api/account/signup", "/h2-console/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .headers().frameOptions().disable()
            .and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .addFilterBefore(jwtRequestFilter, beforeFilter:UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Figure 87: Security Config to filter and authorize requests

9. Recovered architecture and design blueprints

9.1. Design-time package diagram

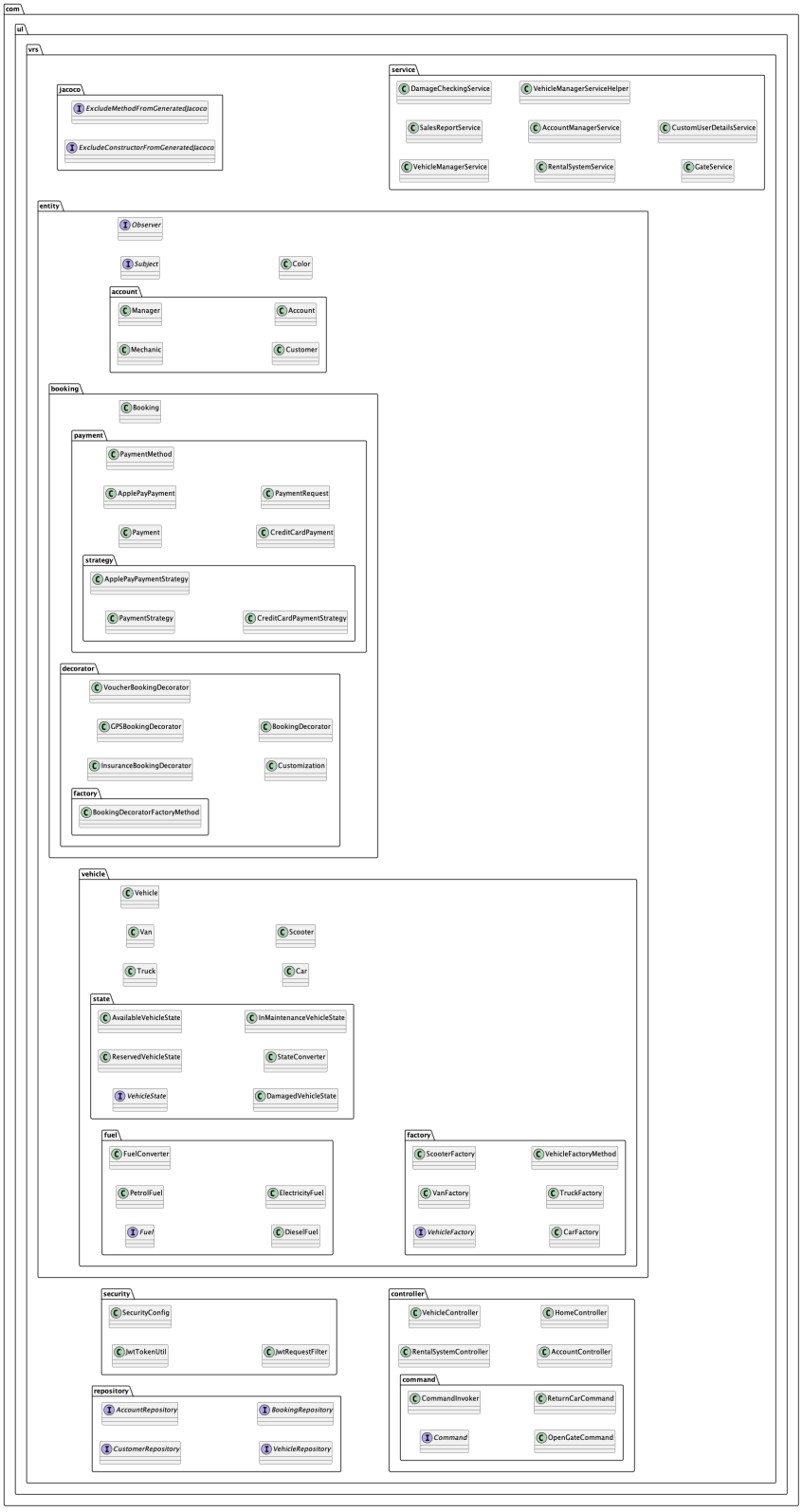


Figure 88: Design-time package diagram

9.2. Design-time class diagram

Recovered Class Diagram

Below are snippets from the class diagram of the Vehicle Rental Application. The full class diagram will be submitted separately due to its size.

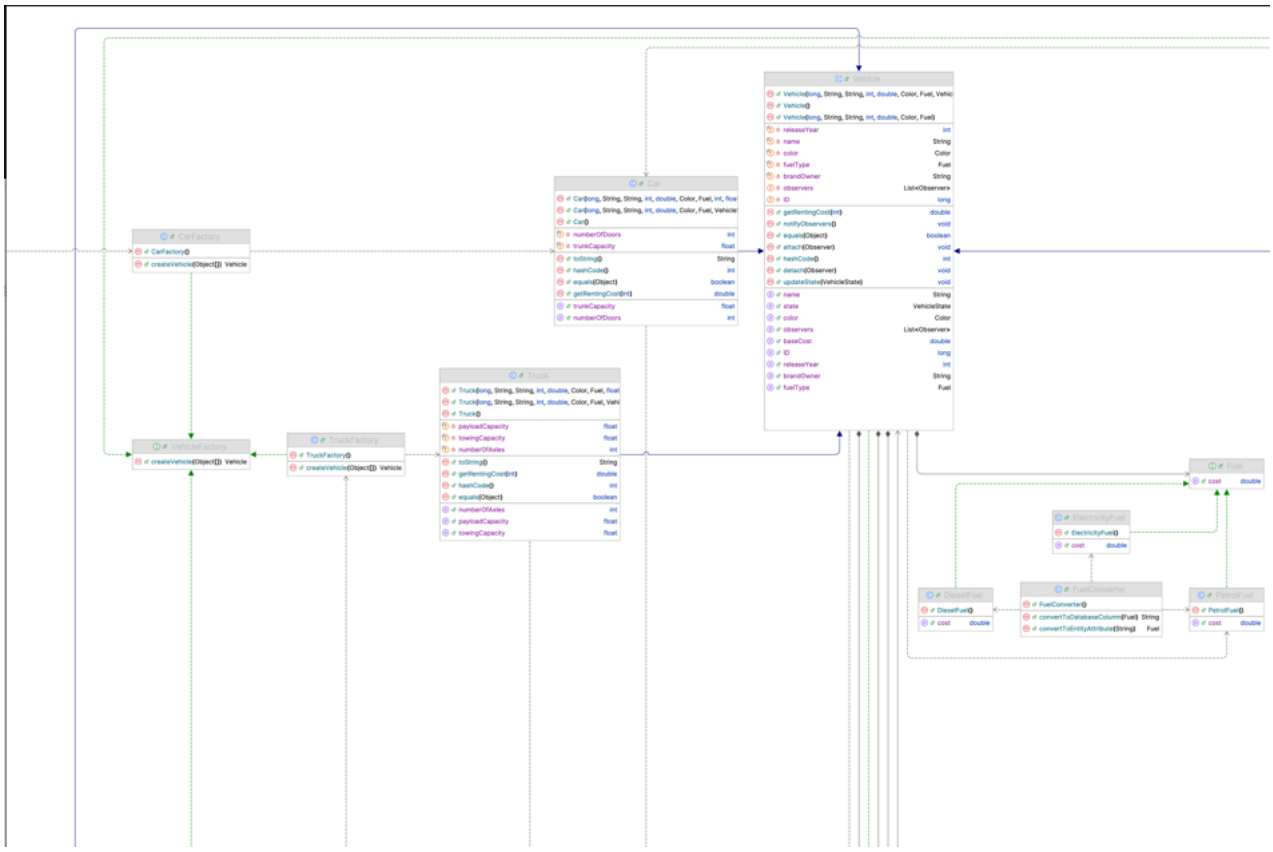


Figure 89: Shows CarFactory, TruckFactory, and VehicleFactory using the Factory design pattern with Vehicle and Fuel hierarchies.

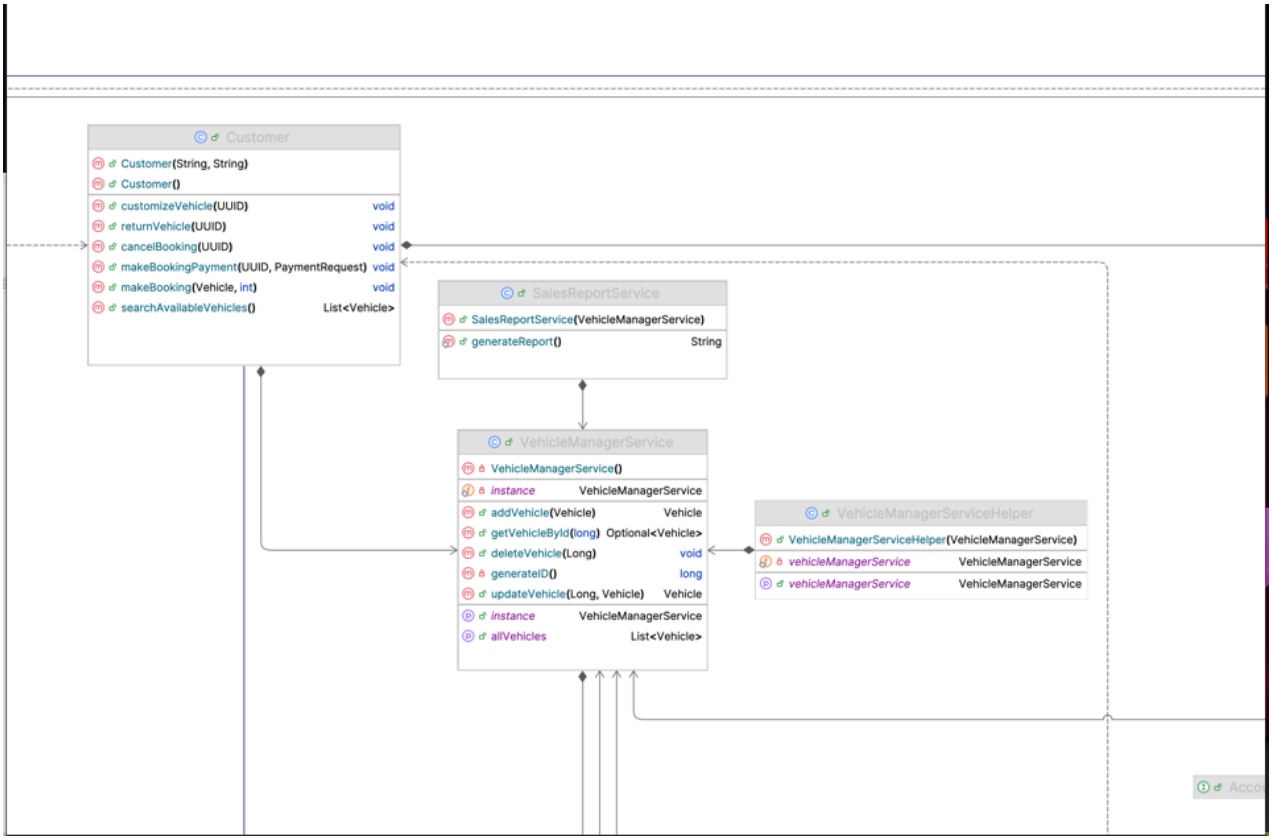


Figure 90: Shows Customer, SalesReportService, and VehicleManagerService collaborating to manage vehicle bookings and reports, with VehicleManagerServiceHelper supporting operations.

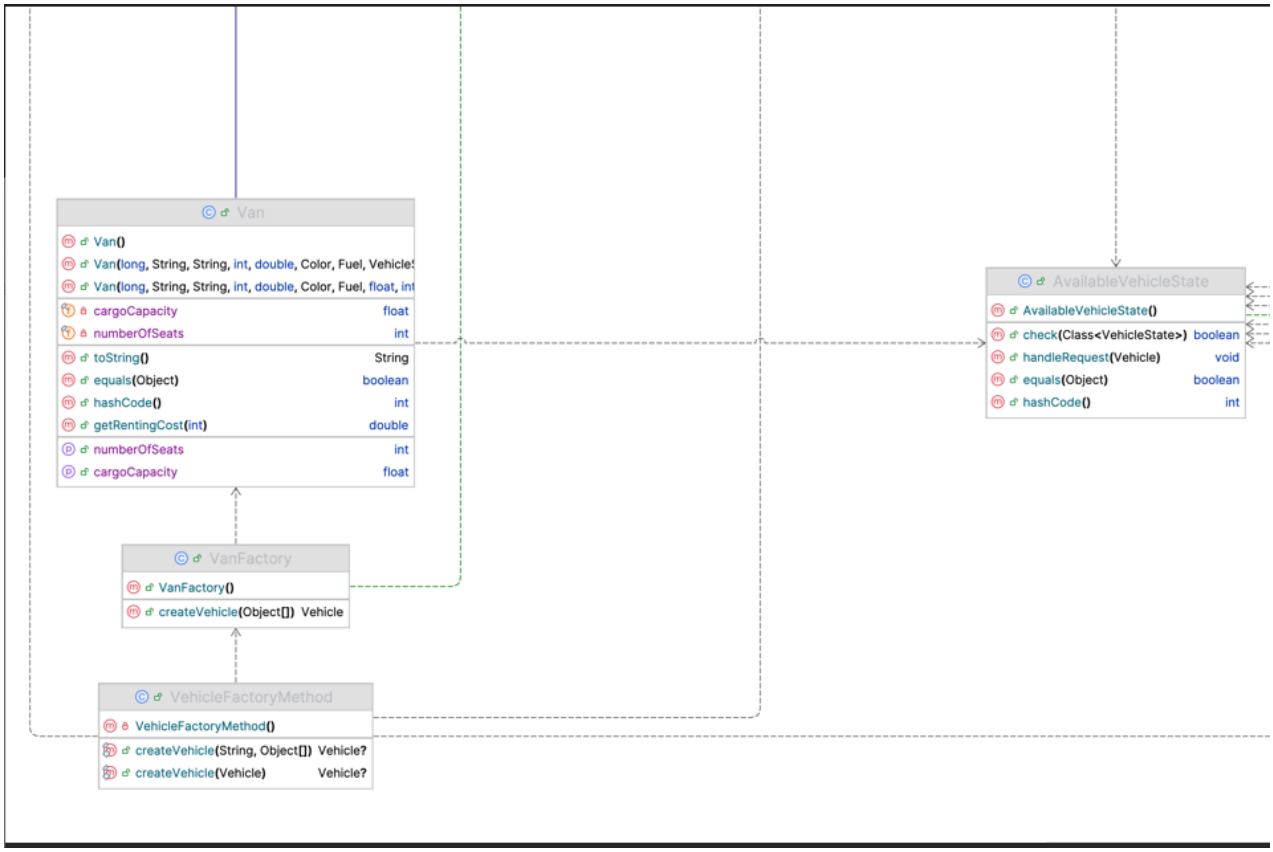


Figure 91: Shows VanFactory implementing the Factory Method pattern to create Van objects, with AvailableVehicleState managing vehicle availability.

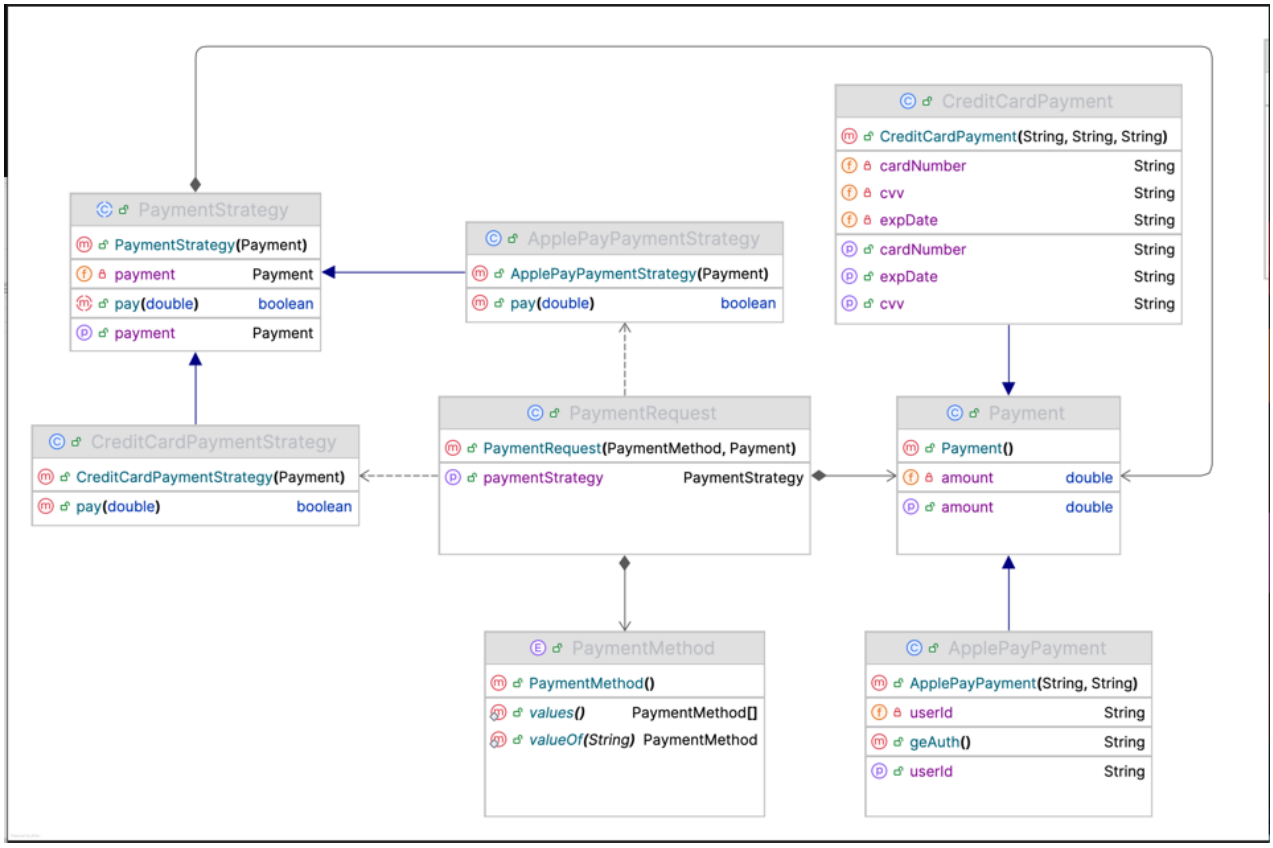


Figure 92: Shows the implementation of the Strategy design pattern with PaymentStrategy defining payment behavior, and concrete strategies like CreditCardPaymentStrategy and ApplePayPaymentStrategy handling specific payment methods.

Figure 93: Shows DamageCheckingService handling vehicle damage management, integrated within the VehicleRentalSystemApplication main flow.

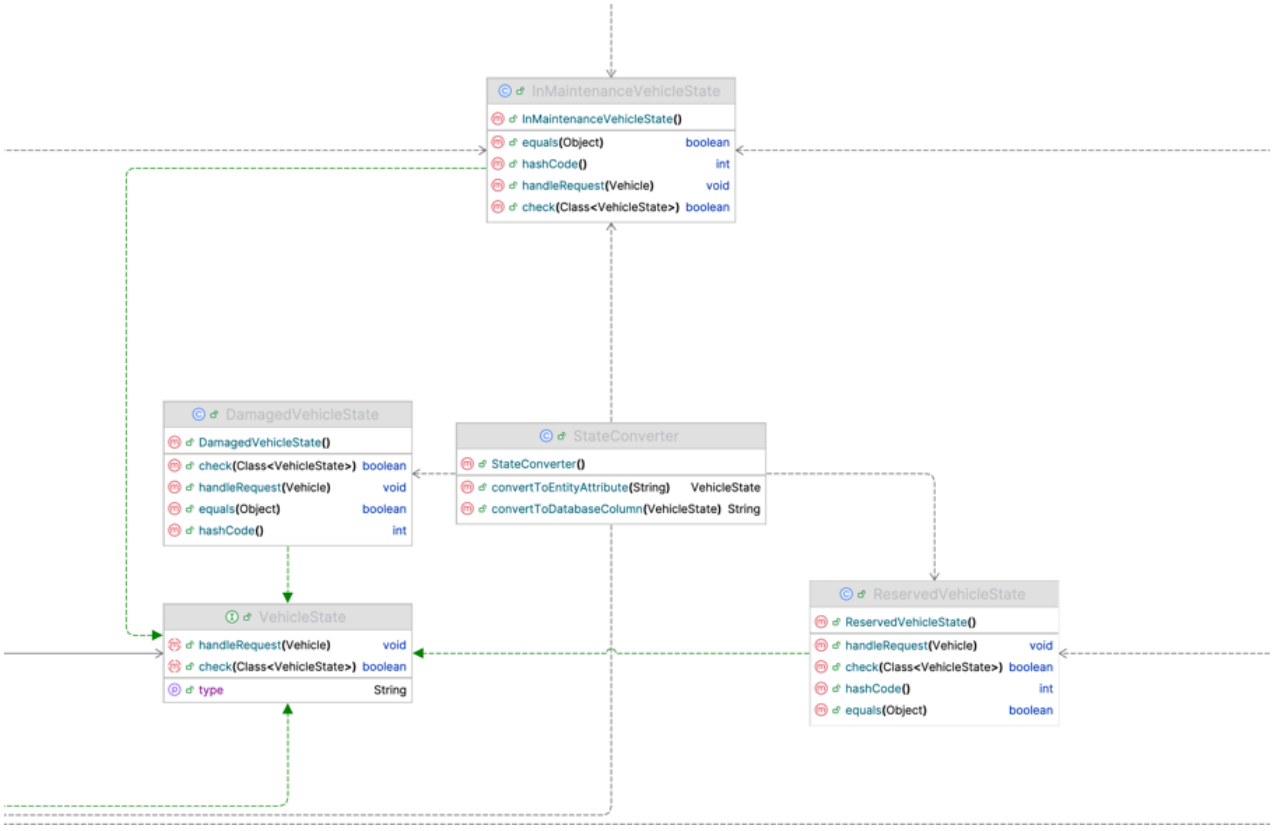


Figure 94: Shows VehicleState implementing the State design pattern with states like DamagedVehicleState, InMaintenanceVehicleState, and ReservedVehicleState, managed through StateConverter.

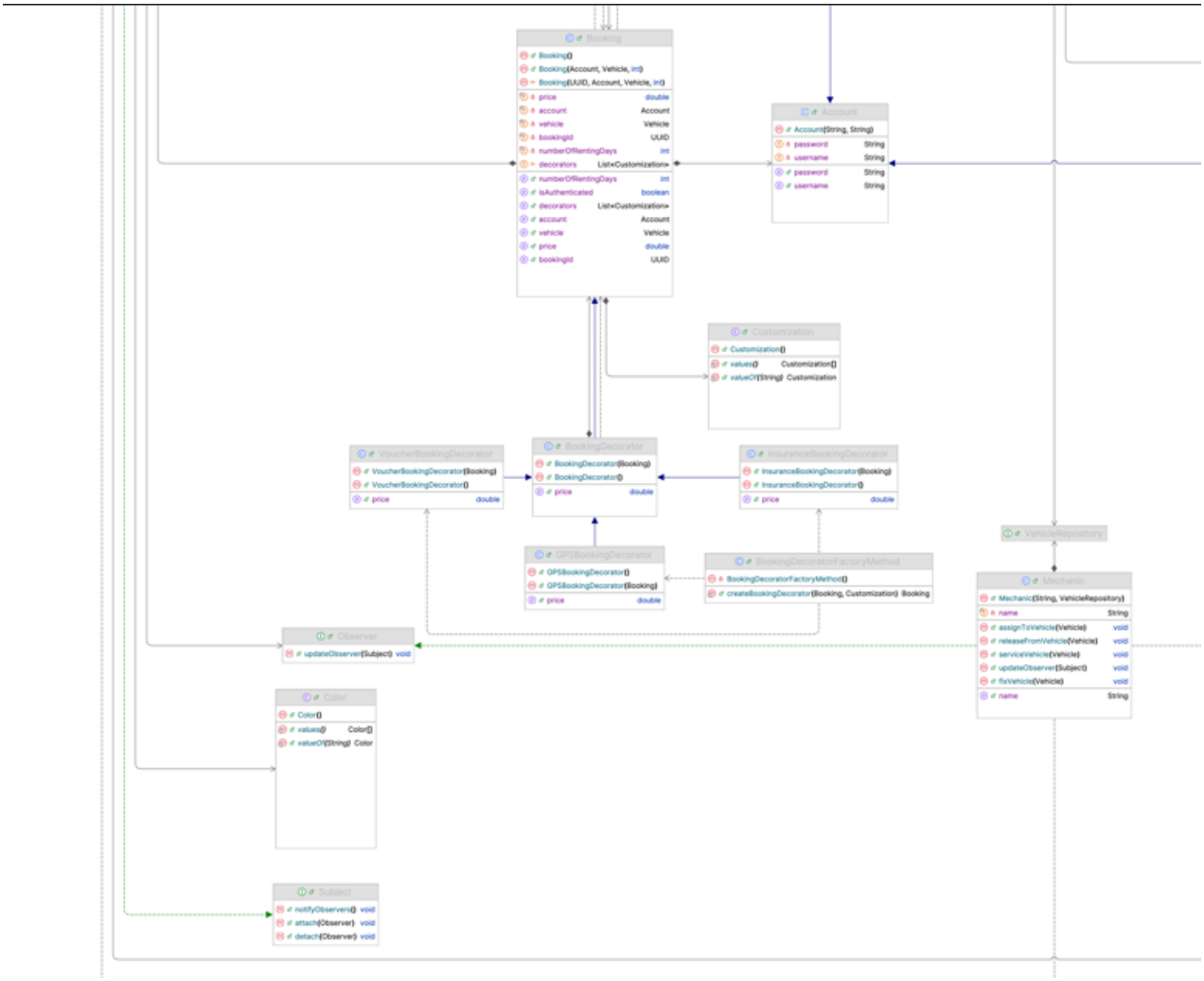


Figure 95: Shows Booking managed through the Decorator design pattern with extensions like VoucherBookingDecorator and InsuranceBookingDecorator, supported by BookingDecoratorFactoryMethod, alongside Observer, Subject, and Color classes.

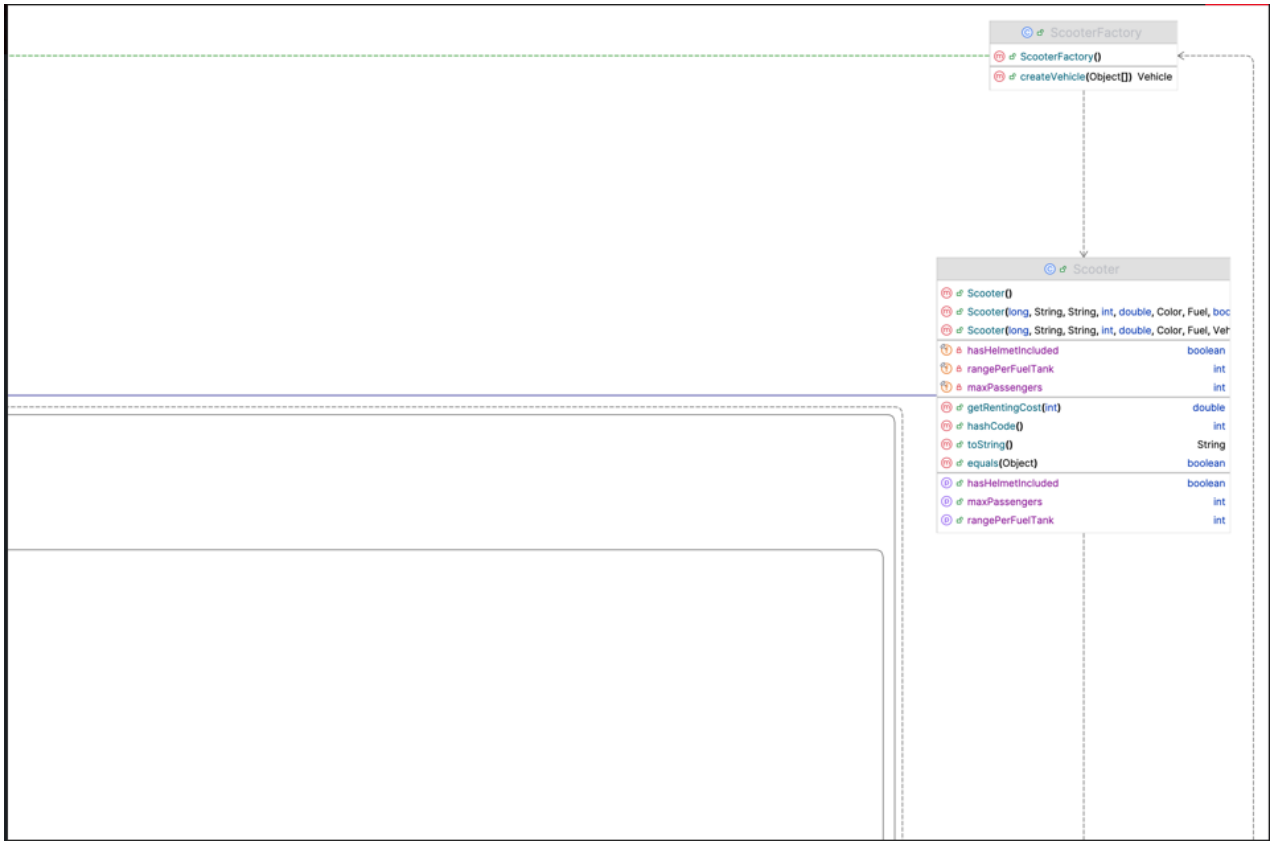


Figure 96: Shows ScooterFactory using the Factory Method pattern to create Scooter objects, which include attributes like rangePerFuelTank and hasHelmetIncluded.

Figure 97: Shows VehicleController, AccountManagerService, and HomeController managing vehicle and account operations, with SecurityConfig and JwtRequestFilter handling authentication, and Manager overseeing vehicle assignments and reports.

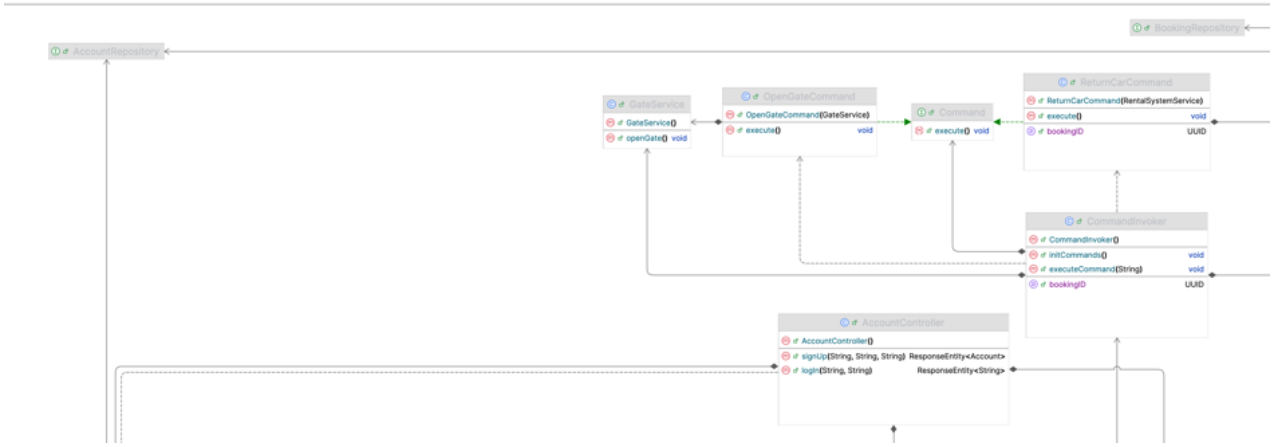


Figure 98: Shows CommandInvoker, OpenGateCommand, and ReturnCarCommand implementing the Command design pattern, with GateService handling gate operations.

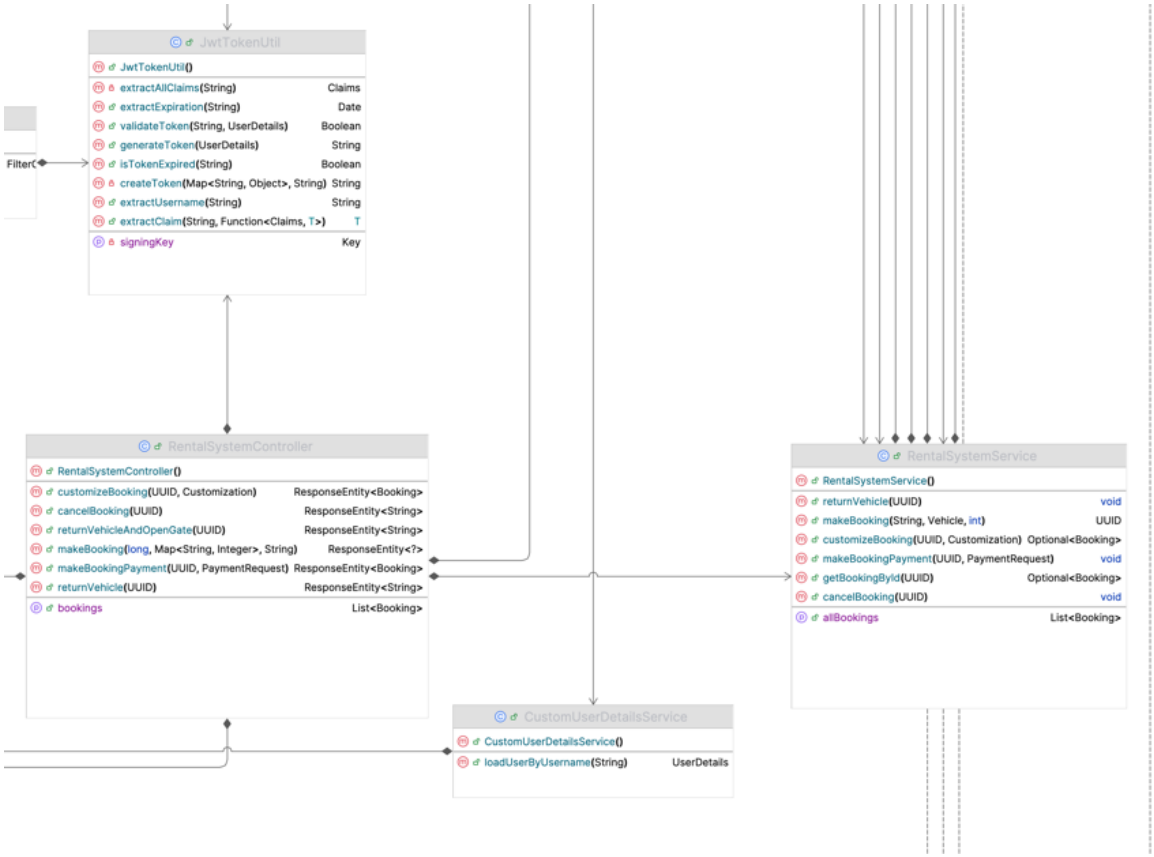


Figure 99: Shows RentalSystemController managing booking operations via RentalSystemService, with JwtTokenUtil handling token validation and CustomUserDetailsService supporting user authentication.

Diagrams generated using IntelliJ IDEA. More information can be found at https://www.jetbrains.com/idea/.

9.3. Design-time sequence diagram

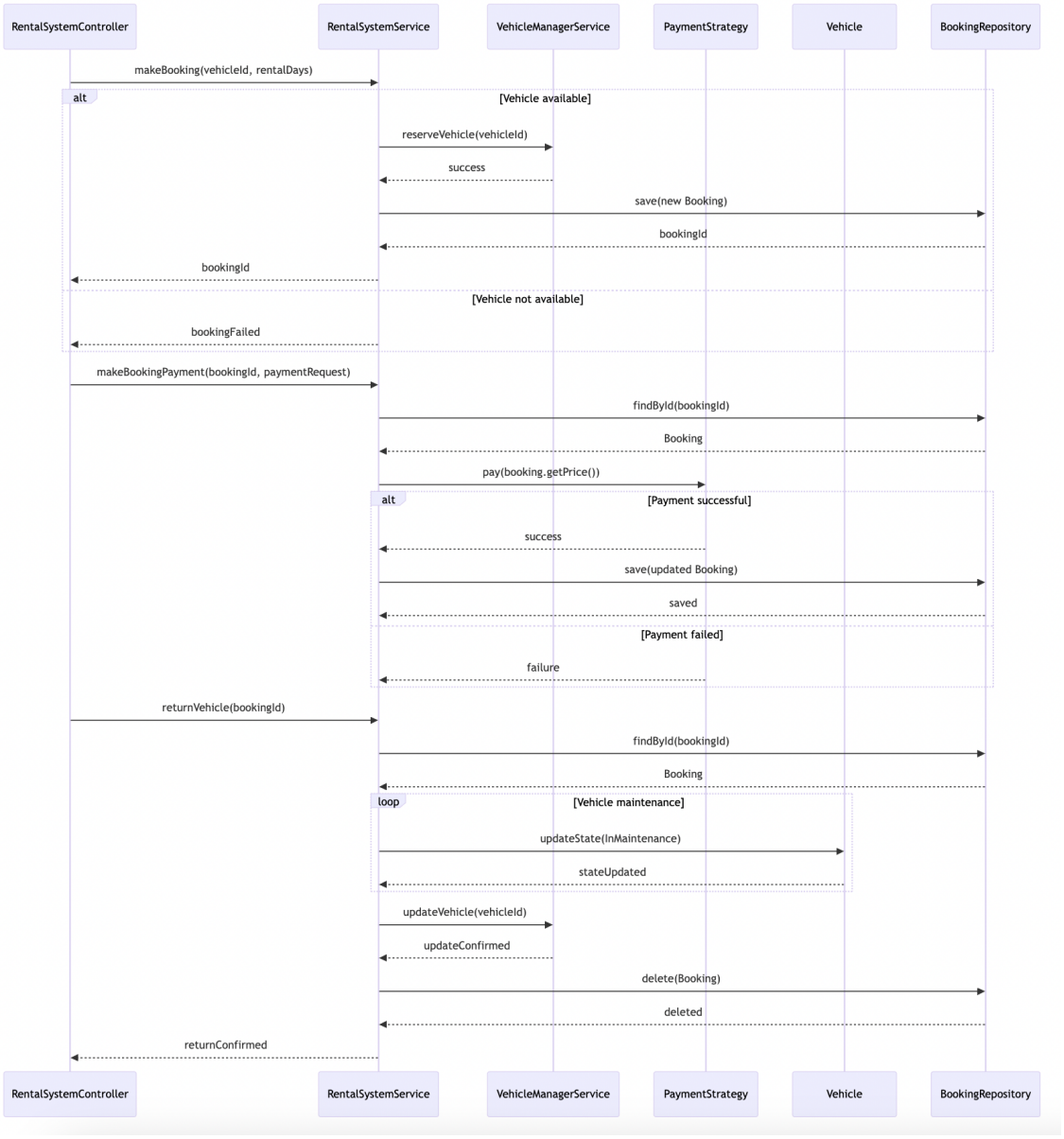


Figure 100: Design-time sequence diagram

9.4. Component and deployment diagrams

Component diagram

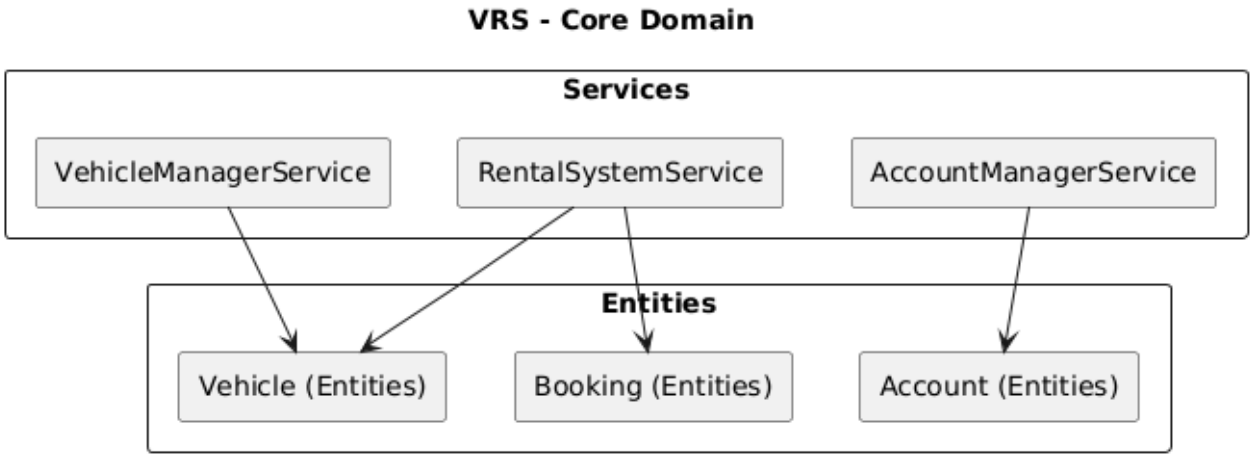


Figure 101: Core Domain: Entities and Core Services

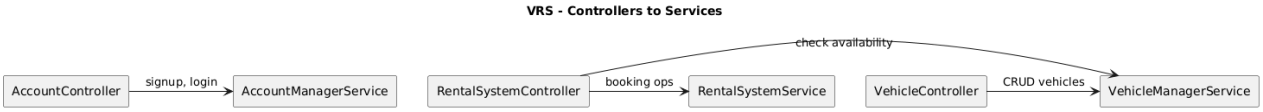


Figure 102: Controllers Interacting with Services

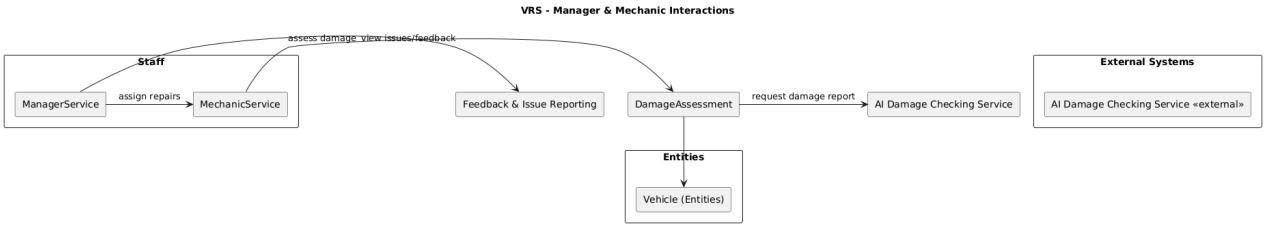


Figure 103: Manager and Mechanic Interactions

Figure 104: Payment and Reporting Integrations

Deployment diagram

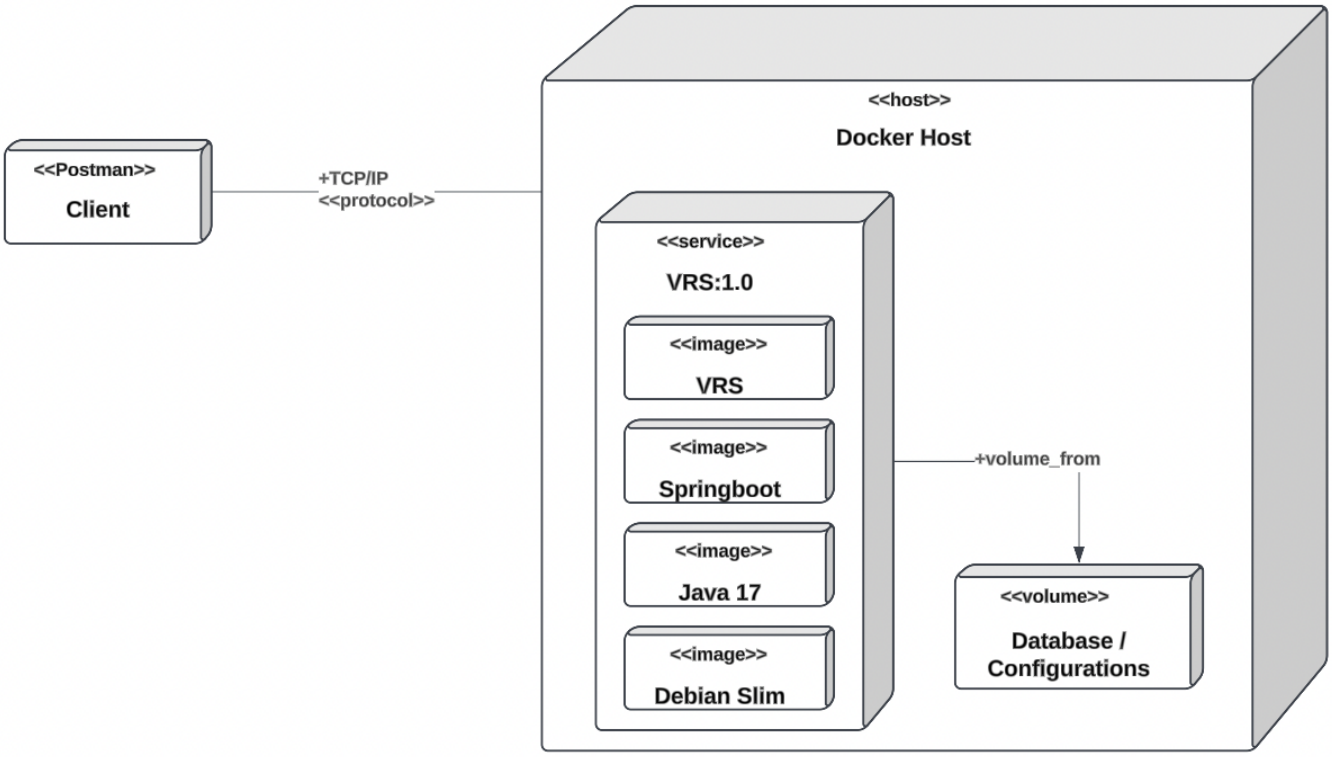


Figure 105: Deployment diagram

10. Critique

This project explores and implements several industry-standard design patterns, including Command, Observer, and Decorator. These patterns are incorporated into an MVC-based application, adhering to SOLID principles to ensure maintainability and scalability. Although the project lacks a front-end interface, the business logic layer has been thoroughly tested using tools like Postman for manual validation of user-accessible endpoints. JUnit and software metrics were employed to ensure code correctness and design quality. Tools such as JDepend and JaCoCo analyzed dependency management and code coverage, while SonarQube provided an interactive assessment of overall software quality. Additionally, PMD was used for static code analysis to detect common programming issues. The project integrates JWT (JSON Web Token) security for user authentication and secure password management. This ensures sensitive data, such as user credentials, is transmitted in a secure, token-based format, reducing unauthorized access and enhancing overall system security.

UML diagrams, including class, sequence, and use-case diagrams, effectively describe the system's components, behaviors, and interactions. Class diagrams highlight the static structure and showcase the integration of design patterns within the system. Sequence diagrams illustrate workflows connecting business needs with technical design, bridging the gap between requirements and implementation. Use-case diagrams present the system's capabilities from the user's perspective, complementing textual and graphical representations to create a holistic view.

The architecture was evaluated for maintainability, scalability, and security. SOLID principles guide the design, ensuring modularity and ease of extension. Metrics tools such as JDepend and SonarQube provided actionable insights into dependency management and quality assurance. Code coverage tools like JaCoCo validated testing thoroughness, while UML diagrams ensured architectural clarity and cohesion. Despite these strengths, challenges such as the time-intensive nature of diagram creation and limitations in representing dynamic behaviors were noted.

The use of UML offers several advantages. Clear visuals simplify understanding of system workflows and component interactions. Its standardized language facilitates effective communication among stakeholders, improving teamwork by bridging gaps between technical and non-technical team members. Early problem solving is enabled by identifying design flaws early in the development cycle, and UML diagrams serve as useful documentation for future maintenance and updates. However, challenges include the time consuming nature of creating and maintaining diagrams, a steep learning curve for new team members, potential for misunderstandings from poorly designed diagrams, a possible focus shift that delays development progress and limitations in representing dynamic system behaviors.

This project demonstrates the effective application of design patterns within an MVC framework, supported by comprehensive testing and architectural analysis. While UML diagrams and static analysis tools help design clarity and quality, future enhancements could address the challenges of runtime representation and streamline diagram updates to further optimize development efficiency.

11. Reflection

Shane's Experience

I have 4 years experience in Java obtained through undergraduate college practical projects, which formed the foundation of my understanding of backend development. While I had some initial knowledge of software design, this assignment allowed me to explore and implement various design patterns in more depth, particularly those commonly applied in Java-based applications. Additionally, collaborating with peers on these projects enhanced my familiarity with tools like SpringBoot, Sonarqube, and GitHub CI/CD actions. These experiences provided me with hands-on exposure to team-based development workflows, including managing pull requests, tracking issues, and resolving merges, which closely mimic real-world scenarios.

Ivor's Experience

I have 4 years of experience working mainly on functional programming and data engineering. Learning and utilizing software design patterns and SOLID framework has been very insightful and has transformed how I approach software development. Integrating the software metrics in our project has helped me understand the importance of utilizing these metrics for assessing the maintainability and reliability of software. I have learned a lot from my peers by collaborating in a team, reviewing code and working on the various UML diagrams. Lastly it has an enriching experience using springboot for REST API development, H2 and Hibernate for database and repository and JUnit for testing our application.

Manjeshwar's Experience

I gained deeper knowledge in this subject area through lectures on critical design patterns, such as the Interceptor Pattern and Factory Methods, which are widely applied in industry environments.

In this project, I developed valuable teamwork skills by successfully implementing solutions using Java and tools like GitHub, Maven, Spring Boot, etc. I also became proficient in using Git and Git Bash to track development progress, pipelines, and unit tests within a collaborative environment.

One of the activities I found the most valuable was reviewing the code of my teammates and discussing the application of these design patterns. The initial idea generation and prototype sessions were particularly enriching, as they encouraged diverse perspectives and helped strengthen my collaboration skills.

This experience has significantly improved my technical knowledge and my ability to effectively contribute to team-driven development processes.

David's Experience

I have one year of industry experience as a Backend Developer. Despite having certain initial knowledge about software design, this subject has allowed me to explore in more detail different design patterns that are commonly used in the industry. Furthermore, working in a team has given me further experience and insight into several tools such as SpringBoot, Sonarqube, and GitHub CI/CD actions. Finally, the project has also allowed me to leverage the features GitHub and Git offer for team-based development, providing hands-on experience with pull requests, issues, and merges in real-world workflows.

Rohan's Experience

Working on this project has been an great learning experience for me. One of the biggest takeaways was understanding how design patterns make adding new features so much easier. It helped keep the code organized and manageable, which was a helpful as the project grew more complex. Using GitHub for version control was another great part. It made collaborating so much smoother especially when we were merging changes or fixing conflicts. Plus having a clear history of edits really helped us stay on the same page. The CI/CD pipeline was a game changer too. Automating our tests meant we could catch issues early and ensure everything stayed on track. Seeing those tests pass (or fail!) after a push gave us instant feedback and a lot of faith in our work. Since I didn't have prior work experience, this project was a great run of coding in a team setting. It wasn't just about writing code, it was about communicating, designing together and learning from each other as we all had varying experience levels. It also allowed me to learn deeper into Java, Spring boot, Junit, Database and Postman which is added benefit!

12. Appendix

Initial Design Sketch

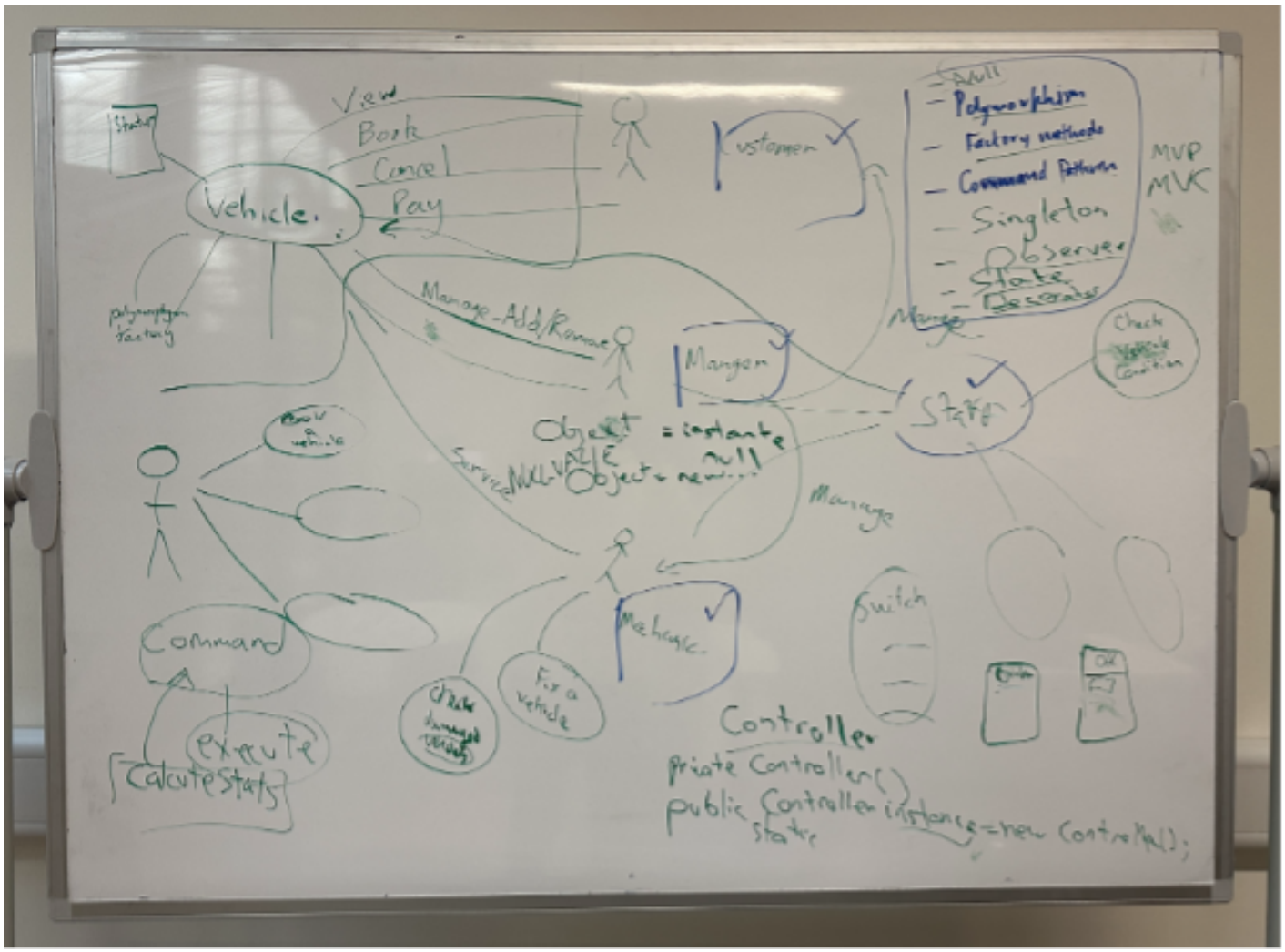


Figure 106: Initial design sketch of the system architecture.

Initial Flow Chart for System Process

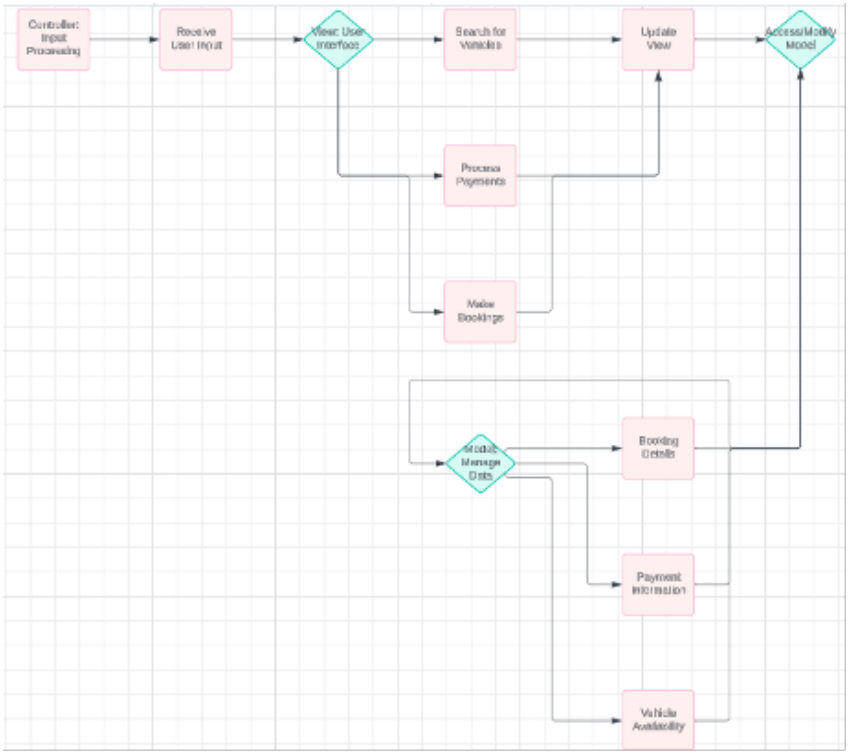


Figure 107: Flowchart of the initial process flow of the system

Process Flowchart

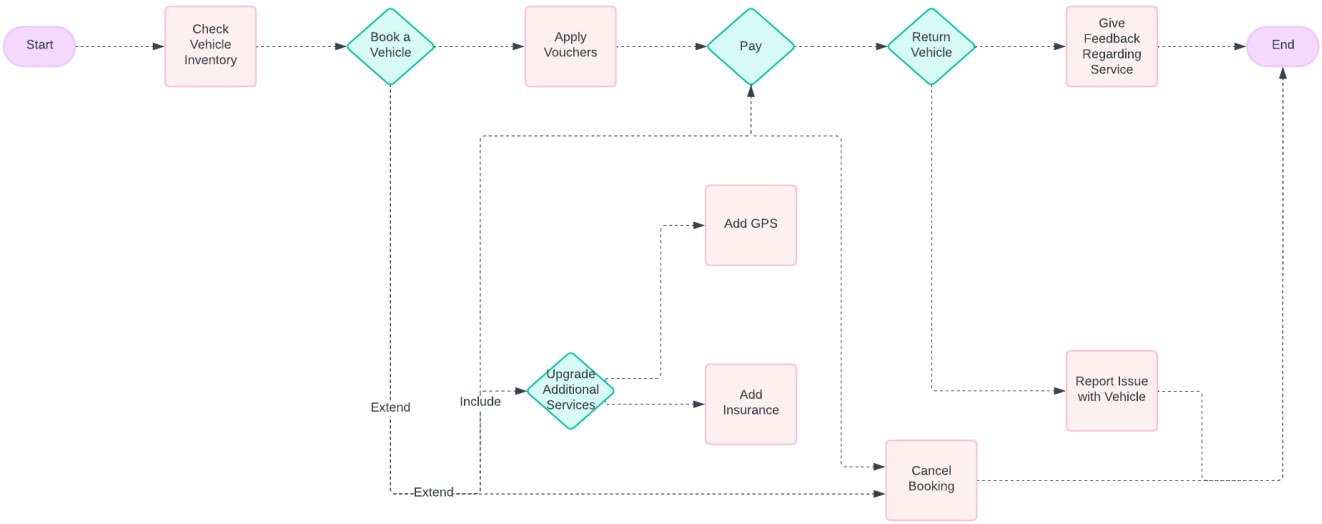


Figure 108: Flowchart of the initial process flow of the system

Initial GUI Prototype Sketch

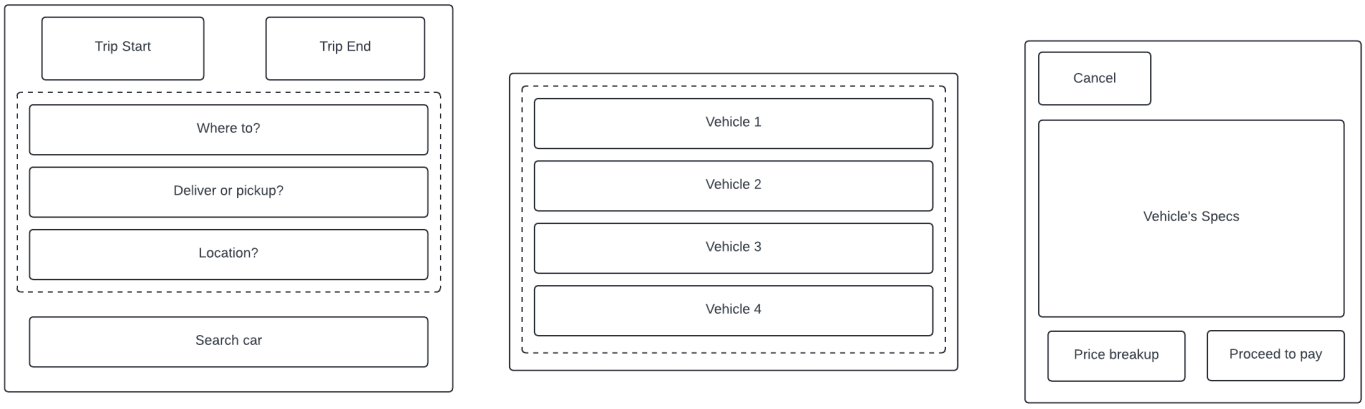


Figure 109: First sketch of the GUI prototype

Early version of the front-end for the logging

login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <h2>Login to Vehicle Rental System</h2>
    <form action="/login" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <br><br>

        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <br><br>

        <button type="submit">Login</button>
        <p th:if="${error}" style="color: red;">${error}</p>
    </form>
</body>
</html>

```

home.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home</title>
</head>
<body>
    <h2>Welcome to the Vehicle Rental System!</h2>
    <p>You're logged in successfully.</p>
    <a href="/logout">Logout</a>
</body>
</html>
```

Figure 110: Early version of the front-end for the logging

Return Vehicle Sequence Diagram

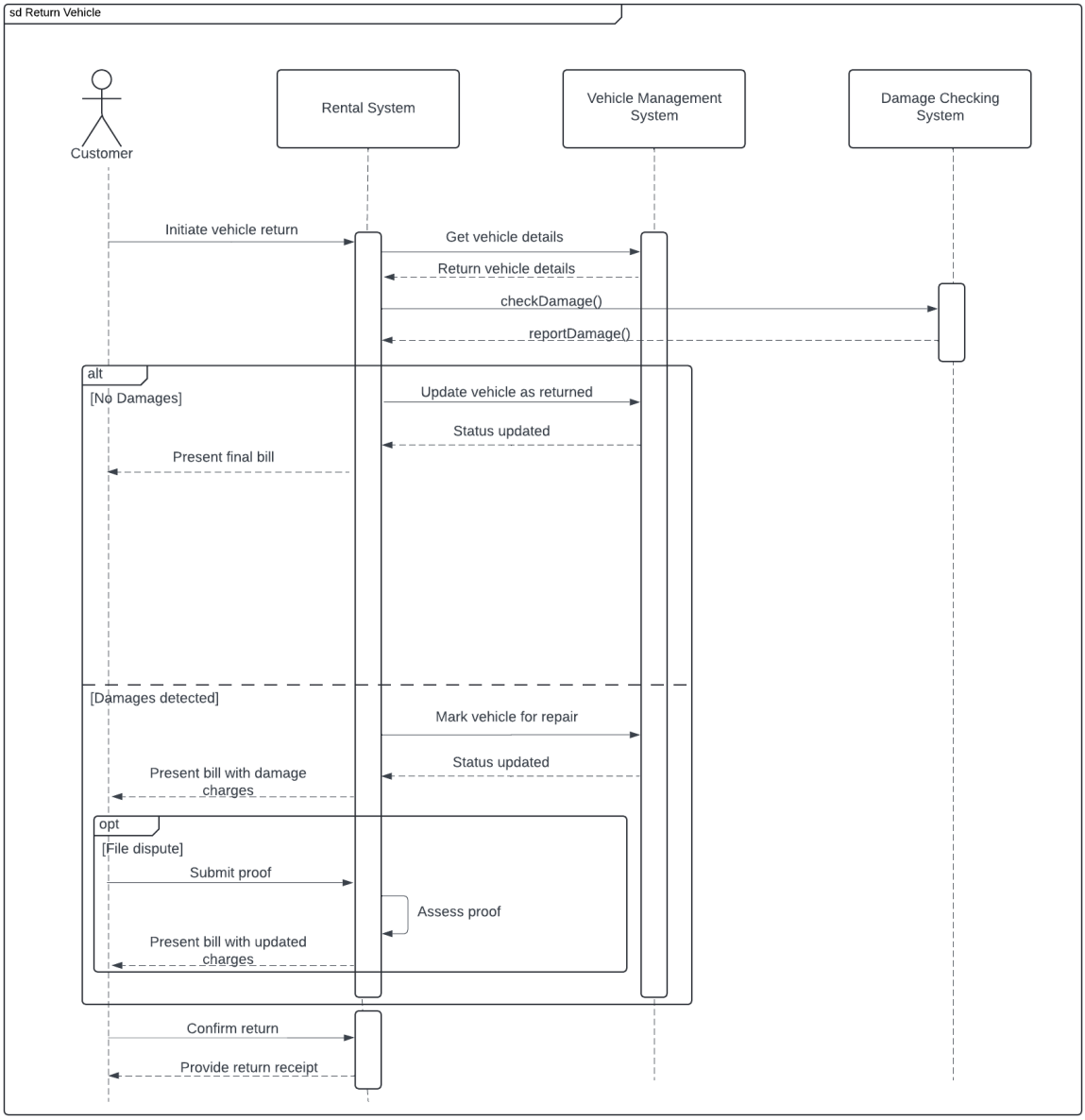


Figure 111: Sequence diagram of the return vehicle use case

Early version of class diagram

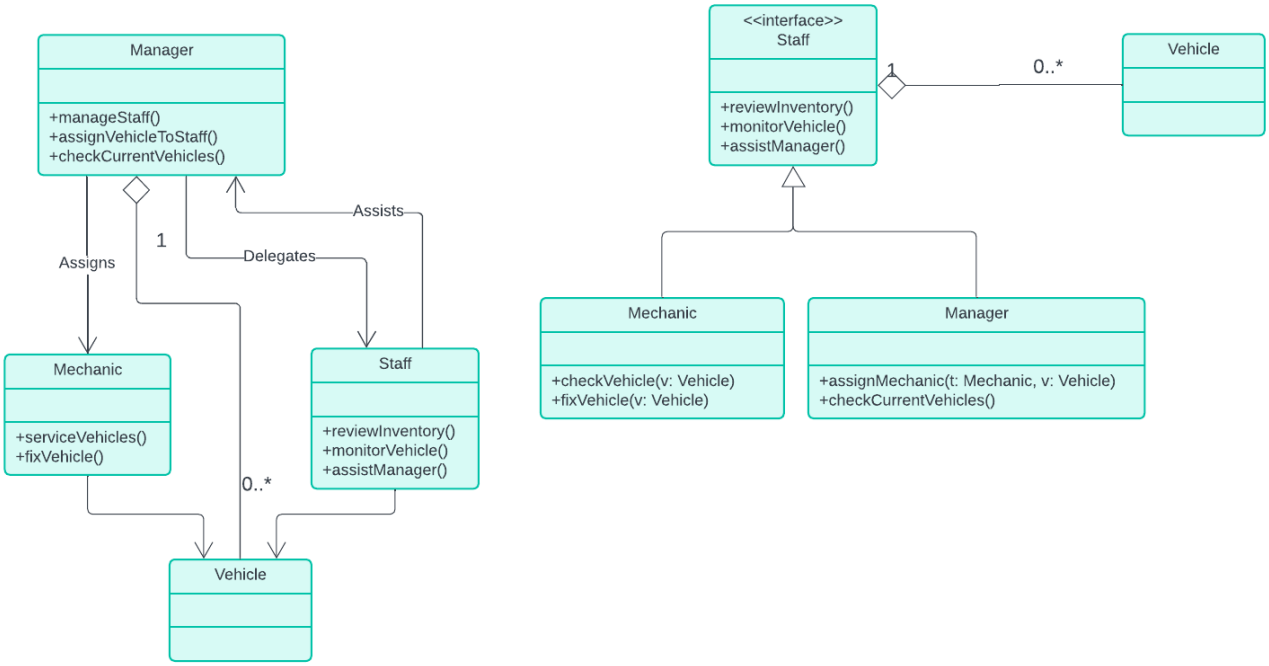


Figure 112: Class Diagram of the initial structure of the system

13. References

3bdelrahman (2023). *Understanding Layers, Tiers, and N-Tier Architecture in Application Development*. Diagram illustrating the N-Tier architecture pattern. URL: https://dev.to/3bdelrahman/understanding-layers-tiers-and-n-tier-architecture-in-application-development-1hlb (visited on 10/25/2024).

Bjerre, Tomas (2024). *Git Changelog Maven Plugin*. URL: https://github.com/tomasbjerre/git-changelog-maven-plugin (visited on 12/05/2025).

GeeksforGeeks (2023). *MVC Model-View-Controller Architecture Pattern*. Diagram illustrating the MVC architecture pattern. URL: https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/ (visited on 10/25/2024).

Lacan, Olivier (2024). *Keep a Changelog*. URL: https://keepachangelog.com/en/1.1.0/ (visited on 12/05/2025).

Liskov, B. (1987). "Keynote address - data abstraction and hierarchy". In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*. Vol. 23. 5. New York, NY, USA: Association for Computing Machinery, 17–34.

Martin, R.C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

– (2000). "Design principles and design patterns". *Object Mentor* 1(34), 597.

Meyer, B. (1997). *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs.

Michaud, Pierre-Jean Lajoie (2024). *Agile Software Development: Everything You Need to Know*. Image illustrating the Agile methodology. URL: https://www.nexapp.ca/en/blog/agile-software-development (visited on 10/27/2024).

Motion (2023). *Waterfall Methodology Diagram*. Image illustrating the Waterfall methodology. URL: https://www.usemotion.com/blog/waterfall-methodology (visited on 10/25/2024).

Oppermann, Artem (2023). *What Is the V-Model in Software Development?* Image. URL: https://builtin.com/software-engineering-perspectives/v-model (visited on 10/25/2024).

PMD Development Team (2024). *PMD - A Source Code Analyzer*. Accessed: 2024-12-08. URL: https://pmd.github.io/.