SWINBURNE UNIVERSITY OF TECHNOLOGY

School of Science, Computing and Engineering Technologies

ASSIGNMENT AND PROJECT COVER SHEET

Subject Code: SWE30003 Unit Title: Software Architectures and Design

Assignment number and title: 2. Object Design Due date: 10th Mar 2025

Tutorial Day and Time: 13:00 - 16:00 Thu Project Group: 5

Tutor: Dr. Do Thi Bich Ngoc

To be completed as this is a group assignment
We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
104182779	Do Duy Hung	Hung
104221559	Doan Trung Nghia	Nghia
104181857	Luong Minh Duc	Duc
104175326	Do Quang Minh	Minh

Marker's comments:

Total Mark:__________________

Extension certification:

This assignment has been given an extension and is now due on ______________________

Signature of Convener:_____________________

Executive Summary

The SmartRide Online Ride-Sharing Platform (ORSP) is designed to address the inefficiencies in traditional ride-hailing services by automating the booking process, optimizing driver-passenger matching, and ensuring secure digital payments. The system aims to provide a seamless experience for both passengers and drivers by integrating real-time tracking, dynamic pricing, user authentication, and an intuitive feedback system.

This document presents the initial object-oriented design of the SmartRide system. It includes an analysis of the problem domain, identification of core system classes, and the relationships between them using UML diagrams. Each class is documented with CRC (Class-Responsibility-Collaborator) Cards, ensuring clarity in responsibilities and interactions. The document also illustrates the bootstrapping process which explains how key components are initialized during system startup.

To make sure SmartRide runs smoothly, grows easily, and stays secure, the design follows some proven software patterns:

- Factory Pattern - Helps create ride requests and payments automatically when needed
- Singleton Pattern - Ensures important services like Database and AuthenticationManager run as a single instance to avoid conflicts.
- Observer Pattern - Keeps passengers and drivers updated with real-time ride status and notifications
- Strategy Pattern - Allows the system to change pricing based on demand and traffic conditions without modifying the whole system

These patterns make SmartRide flexible, easy to manage, and scalable for future improvements.

1, Introduction

1.1, Purpose

The purpose of this document is to present the object-oriented design of the SmartRide Online Ride-Sharing Platform (ORSP). It describes the candidate classes, system interactions, and design choices necessary to build a structured and maintainable software system.This design ensures scalability, reusability, and efficiency, allowing the platform to handle real-time ride booking, driver allocation, and payment processing.

1.2, Document Scope

This document focuses on the high-level object design of SmartRide. It outlines the core system components, including class structure, responsibilities, and relationships. The document also explains the candidate classes and how they interact, design patterns used to improve flexibility and maintainability, the bootstrapping process which shows how system components initialize, verification scenarios which illustrates how objects collaborate in real-world use cases. This document does not cover low-level implementation details such as database schema, API, or UI design.

1.3 Overview of Proposed

SmartRide provides an efficient and automated ride-sharing service where passengers can request rides, track their drivers in real-time, and complete payments securely. The system ensures:

- Seamless ride booking with driver matching.
- Secure payments with multiple payment methods.
- Transparent ratings and reviews to maintain service quality.
- A scalable architecture, supporting future expansion.

The object-oriented design follows software engineering best practices, using modular, reusable, and maintainable code.

1.4 Definitions, Acronyms, and Abbreviations

Online Ride-Sharing Platform	ORSP
Unified Modeling Language	UML
Create, Read, Update, Delete	CRUD
Application Programming Interface	API

Object-Oriented Programming	OOP

2, Problem Analysis

2.1 Functionalities Identified from Assignment 1

Based on assignment 1, there are some key functionalities of SmartRide include:

User (Passenger) Functionalities:

- Create an Account - Passengers can sign up and create a profile.
- Login - Users can access their accounts securely.
- Book a Ride - Passengers can request a ride by specifying a pickup and drop-off location.
- Track the Ride - Users can monitor their driver's real-time location.
- Process payment - Passengers can pay for rides using multiple payment methods.
- Rate & Review - Users can provide feedback on their ride experience.
- Book Additional Services (future implementation) - This will include shared rides and premium options.

Driver Functionalities:

- Create an Account - Drivers can register and submit verification documents.
- Login - Drivers can securely access their accounts.
- Accept a Ride Request - Drivers receive ride requests and choose to accept them.
- Track the Ride - Drivers navigate the trip using route guidance.
- Receive Payment - Drivers are credited for completed rides.
- Rate & Review - Drivers can provide feedback on passengers.

Admin Functionalities:

- Manage Platform - Admins oversee drivers, resolve disputes, and monitor ride activities.
- Administer & Report System - They verify user credentials, handle complaints, and generate financial reports.

2.2 Assumptions and Constraints

Assumptions:

To simplify the design and focus on essential features, we assume:

- All drivers are pre-verified before joining the platform.
- All payments are digital.
- Users have stable internet access to interact with the system.

- Location tracking is always enabled for accurate navigation.
- Each ride is a direct trip from pickup to destination.

Constraints:

- Real-time performance - The system must process ride requests within a few seconds to ensure a smooth experience.
- Data security - Passenger and driver data must be encrypted and protected from unauthorized access.
- Scalability - The platform must support thousands of simultaneous ride requests as it grows.
- Regulatory compliance - The system must follow local ride-hailing laws and payment regulations.

2.3 Simplifications

To keep the system manageable within the scope of this assignment, certain features have been simplified or excluded:

- No multi-passenger ride-sharing - Eachride is booked for one passenger at a time (pooling may be added in the future).
- No cash payments - The system supports only digital transactions for easier tracking and security.
- Limited driver verification process - We assume all drivers are verified beforehand, instead of integrating an external validation system.
- Basic admin functionalities - Admins can monitor rides and user accounts, but advanced analytics and fraud detection are not included in this version.

2.4 Justification of Design Decisions

The SmartRide platform is designed to be simple, flexible, and easy to manage while allowing future improvements. An object-oriented approach (OOD) was chosen to keep different parts of the system separate, making it easier to update and maintain. For example, passengers, drivers, and admins each have their own class with clear roles, preventing the system from becoming too complicated. The design also ensures that important features like ride booking and payments are handled in separate parts of the system, so changes can be made without affecting everything else.

To make the system more efficient and reusable, several design patterns have been used. The Factory Pattern helps create rides and payments in a structured way, while the Singleton Pattern ensures that there is only one database connection to avoid errors. The Observer Pattern is used for real-time updates, so passengers and drivers always know the ride status. The Strategy Pattern makes it easy to adjust pricing based on demand and traffic. These choices make SmartRide a well-structured and scalable platform that can handle more users as it grows.

3, Candidate Classes

3.1 Candidate Class List

- User
    - Passenger
    - Driver
    - Admin
- RideRequest
- Ride
- Vehicle
- Payment
- Payment method
    - CreditCard
    - EWallet
    - BankTransfer
- RatingReview
- Location
- LocationService
- PricingStrategy
- NotificationService
- Report
- AnalyticsEngine
- Database
- AuthenticationManager

3.2 UML Class Diagrams

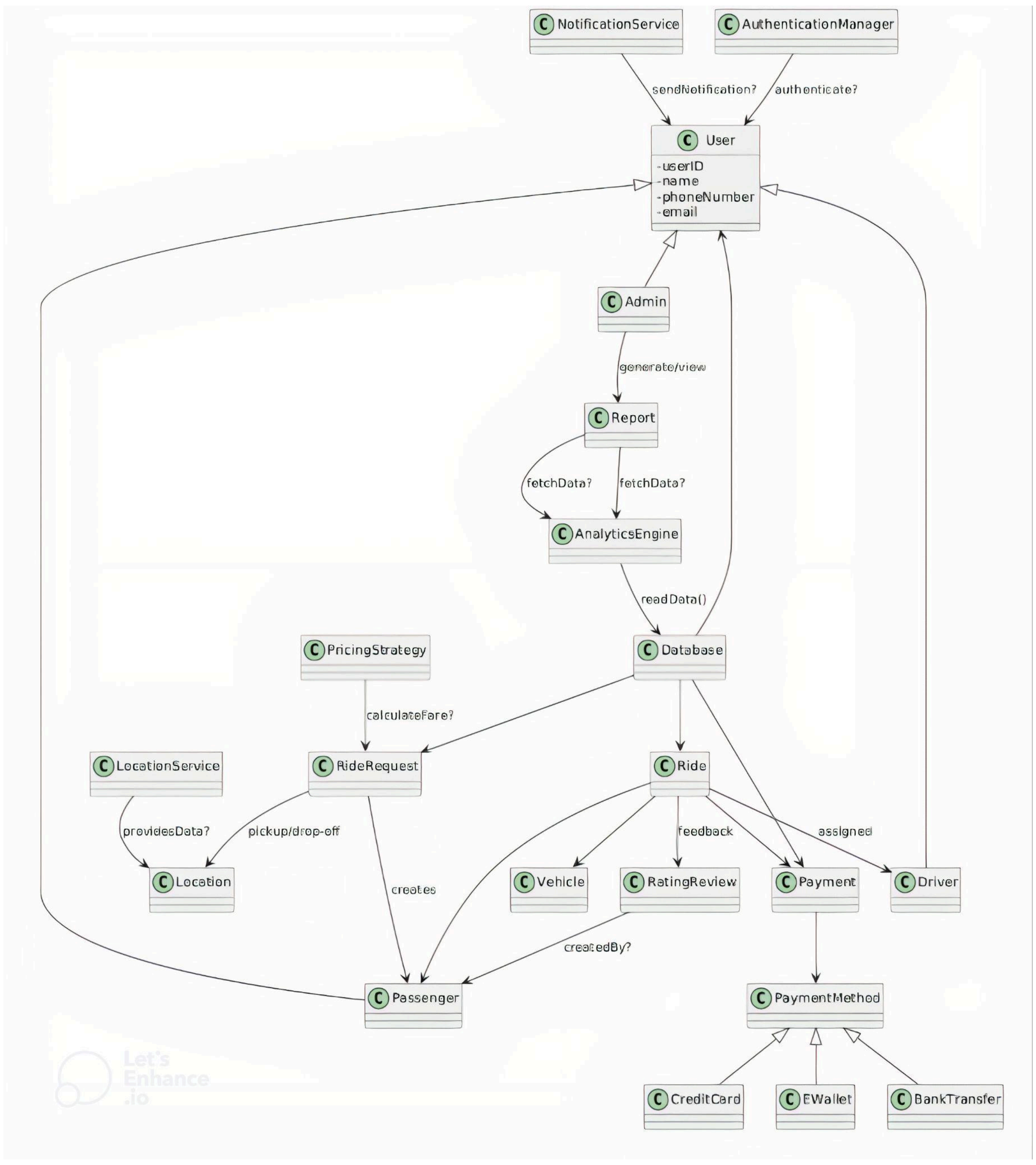


Figure 1 - UML diagram

After a thorough problem analysis, a set of classes was identified by applying the principle of separation of concerns in developing the SmartRide system. Among these classes, a Menu class (or equivalent entry point) serves a key role during the system's bootstrap process. Once instantiated at runtime, it interacts with the user to present a series of tasks (as specified in Section 4.1 of the *Assignment 1 SRS*). The user selects an option, triggering the creation of one or more corresponding objects.

This design is built on the assumption that a Ride entity in a transaction cannot exist without referencing a passenger, a driver, and a payment method. These are the minimal requirements for a ride transaction to take place. The payment structure follows the Factory Design Pattern to handle the creation of different payment types (e.g., credit card, e-wallet, bank transfer).

3.3 CRC Cards

3.3.1 User

Class Name: User **Super Class:** -	
Represents a generic user in the SmartRide system.	
Responsibilities	**Collaborators**
• Knows basic user info: userID, name, phoneNumber, email	N/A
• Can be specialized into Passenger, Driver, or Admin	Passenger, Driver, Admin
• Authenticates credentials (in conjunction with AuthenticationManager)	AuthenticationManager

3.3.2 Passenger

Class Name: Passenger **Super Class:** User	
A user who books rides via the SmartRide platform.	
Responsibilities	**Collaborators**
• Knows default payment method	PaymentMethod
• Creates and manages ride requests	RideRequest
• Views ride history, can leave feedback	Ride, RatingReview
• Authenticates through parent (User) attributes	AuthenticationManager

3.3.3 Driver

Class Name: Driver **Super Class:** User	
A user who drives passengers using a registered vehicle.	
Responsibilities	**Collaborators**
● Manages driver availability	Ride (assigned driver)
● Knows and updates vehicle information	Vehicle
● Receives and accepts ride requests	RideRequest (indirectly)
● Maintains ratings from passengers	RatingReview

3.3.4 Admin

Class Name: Admin **Super Class:** User	
A user with administrative privileges to manage and oversee the system	
Responsibilities	**Collaborators**
● Manages user accounts (activation, suspension)	User
● Generates and reviews system reports	Report
● Oversees the overall platform, resolves escalated issues	AnalyticsEngine
● May authenticate with higher-level security checks	AuthenticationManager

3.3.5 RideRequest

Class Name: RideRequest **Super Class:** -	
A request made by a passenger for a ride before it's confirmed by a driver.	
Responsibilities	**Collaborators**
● Stores pickup and drop-off	Location

locations, request time, and status	
● Links to the passenger who created it	Passenger
● Interacts with pricing to calculate fare estimates	PricingStrategy (indirectly)
● Waits to be assigned to an available driver (via the system's logic)	Driver (indirectly)

3.3.6 Ride

Class Name: Ride **Super Class:** -	
Represents a confirmed trip between a passenger and a driver.	
Responsibilities	**Collaborators**
● Knows assigned driver, passenger, vehicle, pickup/drop-off locations	Driver, Passenger, Vehicle, Location
● Tracks status of the ride (ongoing, completed, canceled)	NotificationService (for updates)
● Calculates or updates final fare (in conjunction with PricingStrategy, Payment)	PricingStrategy, Payment
● Allows feedback from passenger to driver (rating/review)	RatingReview

3.3.7 Vehicle

Class Name: Vehicle **Super Class:** -	
Holds information about the driver's vehicle.	
Responsibilities	**Collaborators**
● Knows make/model/type of the vehicle	N/A
● Associates with a single driver	Driver

• Updated if vehicle details change	N/A

3.3.8 Payment

Class Name: Payment **Super Class:** -	
Represents a financial transaction for a completed ride.	
Responsibilities	**Collaborators**
• Knows paymentID, amount, and status (paid, pending, refunded)	N/A
• Processes payment for a ride	Ride
• Associates with a payment method	PaymentMethod

3.3.9 PaymentMethod

Class Name: PaymentMethod **Super Class:** - *(abstract base class)*	
Parent class for specific payment methods.	
Responsibilities	**Collaborators**
• Common interface for payment-related operations	PaymentMethod
• Subclasses handle actual charging logic (credit card, e-wallet, etc.)	N/A

3.3.9a CreditCard (extends PaymentMethod)

Responsibilities	**Collaborators**
• Stores card details (number, CVV, expiry date, holder name)	Payment
• Performs actual credit card charge	Payment

3.3.9b EWallet (extends PaymentMethod)

Responsibilities	**Collaborators**

● Stores walletID, provider info	Payment
● Processes charges via wallet API	Payment

3.3.9c BankTransfer (extends PaymentMethod)

Responsibilities	**Collaborators**
● Holds bank name, account number	Payment
● Requests transfer confirmations	Payment

3.3.10 RatingReview

Class Name: RatingReview **Super Class:** -	
Records feedback (rating, review text) from passenger to driver.	
Responsibilities	**Collaborators**
● Stores ratingID, rating score (1–5), comment	N/A
● Links to the passenger who created it	Passenger
● Optionally links to the ride (if needed)	Ride

3.3.11 Location

Class Name: Location **Super Class:** -	
Stores coordinates and address information for pickups, drop-offs, or reference points.	
Responsibilities	**Collaborators**
● Holds latitude, longitude, address	N/A
● Used by RideRequest, Ride for navigation	N/A

3.3.12 LocationService

Class Name: LocationService **Super Class:** -	
Provides location-based services such as geocoding, route calculation.	
Responsibilities	**Collaborators**
• Offers current location lookups, distance/ETA calculations	Location
• Potentially interacts with external map APIs	N/A

3.3.13 PricingStrategy

Class Name: Pricing Strategy **Super Class:** -	
Handles fare calculation logic, possibly including dynamic pricing.	
Responsibilities	**Collaborators**
• Calculates fare estimates based on distance/time	RideRequest, Ride
• Considers surge pricing, demand, or special rates	N/A

3.3.14 NotificationService

Class Name: NotificationService **Super Class:** -	
Sends notifications (SMS, push notifications, email) to users.	
Responsibilities	**Collaborators**
• Notifies driver/passenger about ride updates (accepted, arriving, etc.)	User, Ride
• Might use external messaging services	N/A

3.3.15 Report

Class Name: Report **Super Class:** -	
Represents generated information for analytics or admin usage.	
Responsibilities	**Collaborators**
● Retrieves system data for summary	AnalyticsEngine
● Displays or exports ride statistics, revenue, etc.	Admin

3.3.16 AnalyticsEngine

Class Name: AnalyticsEngine **Super Class:** -	
Performs advanced data analysis and forecasting.	
Responsibilities	**Collaborators**
● Analyzes historical ride data, usage patterns, revenue	Database
● Predicts future demand, helps with load balancing	Report

3.3.17 Database

Class Name: Database **Super Class:** -	
Central persistence layer for system data.	
Responsibilities	**Collaborators**
● Stores, retrieves, updates all data (users, rides…)	User, RideRequest, Ride, Payment, etc.
● Ensures data integrity and concurrency control	N/A

3.3.18 AuthenticationManager

Class Name: AuthenticationManager **Super Class:** -	
Handles login, logout, and user validation.	
Responsibilities	**Collaborators**
• Authenticates user credentials (password, token, etc.)	User
• Maintains session or token-based authentication	Database
• Manages roles/permissions for Admin vs. Passenger vs. Driver	Admin, Driver, Passenger

4, Design Quality

The SmartRide Object Design follows high quality software design principles to ensure scalability, maintainability, and efficiency. This section explains how design heuristics and best practices were applied.

4.1 Adherence to Object-Oriented Principle

- Encapsulation: Each class only exposes necessary information while keeping its internal state private.
- Modularity: The system is divided into separate classes to reduce dependencies
- Low Coupling & High Cohesion:
  - Low coupling ensures that changes in one class do not heavily impact others.
  - High cohesion ensures that each class has a single responsibility.

4.2 Avoiding Overcomplicated Classes

- No single class handles everything. Instead, tasks are split into clear roles, like Passenger, Driver, Ride, and Payment.

4.3 Use of Design Heuristics

- Keep it simple: The design is clear and avoids unnecessary complexity.

- Use clear names: Every class and function has a meaningful name (e.g., RideRequest, PricingStrategy).
- Follow proven patterns: The system follows well-known design methods.

5, Design Patterns

5.1 Creational Patterns

In the SmartRide system, Creational Patterns are applied to optimize object creation and lifecycle management. Below are the main Creational Patterns used:

5.1.1 Signleton Pattern

Purpose

The Singleton ensures that only one instance of a class is created throughout the system's runtime. (E., 2004, #)This helps prevent resource conflicts and uncontrolled object creation, which can waste memory or cause inconsistencies when data is shared improperly.

Application in SmartRide

- **Database Connection**: SmartRide can use the Singleton pattern to manage the database connection. Only one connection is created and shared with other components (such as Payment, Ride, User, etc.) instead of creating multiple new connections, which would be wasteful.
- **Configuration Manager**: General configuration parameters (e.g., API keys, payment gateway info, environment variables) are loaded only once and used throughout the system to ensure consistency.

Implementation

Private Constructor: The Singleton class hides its constructor so that other classes cannot instantiate it arbitrarily.
Static Instance: It stores the single instance of the class in a static variable (for example, *private static DatabaseConnection instance*)
Public Access Method: It provides a public method (often *getInstance()*) to retrieve this object. If it does not yet exist, the method initializes it; if it already exists, it returns the existing instance.

5.1.2 Factory Method Pattern

Purpose

The Factory Method allows objects to be created via a unified method rather than calling constructors directly. This gives the creation process flexibility and makes the system easier to extend whenever new “products” (classes) are needed without affecting existing classes.

Application in SmartRide

- **PaymentMethod Creation**: The system supports multiple payment methods (e.g., credit cards, e-wallets, bank transfers, etc.). Instead of calling new *CreditCard()* or new *EWallet()* directly, SmartRide provides a **PaymentFactory** with a factory method that generates objects depending on the user’s chosen payment type.
- **RideRequest / Ride**: The application may use the Factory Method to create different “Ride” types (e.g., regular Ride, premium Ride, motorbike Ride) based on the passenger’s request, separating the logic of object creation from the main code.

Implementation

- **Define an Interface/Superclass** (for example, PaymentMethod) describing shared behaviors.
- **Implement Subclasses** (for example, CreditCard, EWallet) that inherit from the interface/superclass.
- **Provide a Factory** (for example, PaymentFactory) with a createPaymentMethod(paymentType) method that returns the appropriate subclass.
- **Client Code** (classes such as PaymentService or Ride) does not directly call constructors of subclasses but instead uses the Factory’s creation method, making the code cleaner and easier to extend when new payment types are added.

In summary, the Singleton and Factory Method Creational Patterns allow SmartRide to tightly control object lifecycles, increase flexibility, and reduce dependencies among components. This ensures the system is easy to maintain and expand when adding new features in the future.

5.2 Behavioural Patterns

Behavioral Patterns focus on how objects interact and distribute responsibilities among themselves. They help define and manage complex communication flows in a flexible and extensible way. In SmartRide, the following Behavioral Patterns are particularly relevant:

5.2.1 Strategy Pattern (e.g Pricing Strategies..)

Purpose

The Strategy Pattern allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from clients that use it.

Application in SmartRide

Pricing Strategy

- **Context**: Ride pricing can fluctuate based on distance, time, demand surges, or promotional campaigns.
- **Strategy Interface**: PricingStrategy defines a method (e.g., *calculateFare(Ride ride)*) that each concrete pricing strategy must implement.
- **Concrete Strategies**:
    - **StandardPricing:** Calculates fare based on base fare plus additional cost per kilometer/minute.
    - **SurgePricing:** Applies a multiplier to the standard fare during peak hours or high-demand areas.
    - **PromotionPricing:** Applies discounts or coupons for promotional events.
- **Result**: The SmartRide system can dynamically switch pricing algorithms depending on external factors (traffic, time of day, promotions) without modifying the main Ride or Payment classes.

Driver Assignment Strategy

- **Context**: Assigning drivers to passengers involves multiple approaches, such as nearest-driver-first or load-balancing across different regions.
- **Strategy Interface**: AssignmentStrategy might define a method (e.g., *assignDriver(RideRequest request)*) that returns the best-suited driver for that request.
- **Concrete Strategies**:
    - **NearestDriverStrategy:** Selects the driver closest to the pickup location.
    - **LeastBusyDriverStrategy:** Selects the driver with the fewest current or recent rides.
    - **PredictiveAssignmentStrategy:** Uses AI or historical data to forecast future demand and proactively dispatch drivers to high-demand areas.

- **Result**: SmartRide can adapt driver assignments to changing business rules, traffic conditions, or real-time demand seamlessly.

5.2.2 Observer Pattern (Notification System)

Purpose

The Observer Pattern defines a one-to-many relationship between objects so that when one object changes state, all of its dependents are automatically notified and updated. (Erich et al., 1994, #) This promotes loose coupling and makes it easy to broadcast changes to multiple listeners.

Application in SmartRide

Real-Time Notifications

- **Context**: When the status of a ride changes—e.g., driver accepted, driver is arriving, or ride completed—various components (passenger UI, driver UI, or admin dashboard) need to be updated immediately.
- **Subject**: A Ride or RideRequest object can act as the Subject, holding its status and relevant data (e.g., estimated arrival time, driver location).
- **Observers**:
  - **PassengerApp:** Displays updated driver location or arrival time.
  - **DriverApp:** Shows the passenger's pickup details or changes in destination.
  - **AdminPanel:** Monitors overall system health and ride progress for management.
- **Result**: Once the Ride state is updated (e.g., driver assigned, driver en route), the subject notifies all observers, ensuring each UI or subsystem stays consistent with the latest ride status.

Payment Status Updates

- **Context**: Upon successful payment, multiple modules (driver earnings, passenger receipt, and admin reporting) must react to the new transaction state.
- **Subject**: A Payment object that tracks its state (e.g., pending, completed, failed).
- **Observers**:
  - **DriverApp:** Reflects updated earnings.
  - **PassengerApp:** Shows payment confirmation or any errors.
  - **ReportingService:** Records the transaction for analytics and reporting.

- **Result**: Any changes in payment status trigger updates across the system, ensuring that each component stays informed without tight coupling.

SmartRide successfully controls system behaviors as user activities and system states grow by using the Strategy and Observer patterns (as well as other Behavioral Patterns as needed). This approach simplifies maintenance, allows for feature additions, and assures that critical data (such as ride status or payment progress) is reliably updated across several components without introducing burdensome dependencies.

5.3 Structural Patterns

5.3.1 Model-View-Controller (MVC) Pattern

Purpose

The Model-View-Controller (MVC) pattern separates the system's data (Model), how it's displayed (View), and the logic that controls that data flow (Controller). (Erich et al., 1994, #) This separation of concerns makes the codebase more modular, more maintainable, and easier to scale.

Application in SmartRide

SmartRide relies on multiple user interfaces (PassengerApp, DriverApp, AdminPanel) that display data in different ways. Implementing MVC ensures each interface can be modified or enhanced with minimal impact on other parts of the system. Below is how each component fits into SmartRide's architecture:

Model

- **Data Entities**: Classes like User, Passenger, Driver, Ride, Payment, and ReviewRating.
- **Business Logic**: Manages data persistence, validation, and domain operations, e.g., calculating fares, checking ride availability, and handling payment status.
- **Example**:
  - Ride model updates its status from *requested → assigned → in-progress → completed* based on logic in RideService.
  - Payment model stores transaction details and final amounts, possibly linking to external payment gateways via a PaymentService.

View

- **User Interfaces**: The passenger mobile app, driver mobile app, and administrative web dashboard.
- **Tasks**:
    - Display relevant data to the user (e.g., ride availability, driver location, payment summary).
    - Present forms or screens to gather user inputs (e.g., booking details, ratings, or driver documents).
- **Examples:**
    - Passenger sees a map with the driver's location updated in real time.
    - Driver sees current ride requests and an "Accept" button in their mobile UI.
    - Admin sees a dashboard with live metrics and analytics on rides and revenue.

Controller

- **Logic & Coordination**: Sits between the View and Model, handling requests and orchestrating domain operations.
- **Tasks**:
    - Receives user inputs (e.g., "Book a ride" click, "Accept ride" button) and invokes domain logic on corresponding models (*RideService, PaymentService*, etc.).
    - Updates the View by returning model data or status updates.
- **Examples:**
    - When a passenger clicks "Book Ride," the controller calls the AssignmentStrategy to find a suitable driver. Once assigned, the controller updates the Model (Ride) and notifies the View to show ride details.
    - When a driver confirms the ride, the controller triggers an update to the Ride model and sends new data to the Passenger's View (ETA, driver info).

Benefits

- **Separation of Concerns**: Each component focuses on its own task (data, UI, or logic).
- **Easier Maintenance**: Changes in one component (e.g., refining how rides are displayed) do not require major adjustments in other layers.
- **Scalability**: Additional features like loyalty programs, advanced analytics, or notifications can be plugged in with minimal code conflicts.
- **Parallel Development**: Different teams (front-end developers, back-end developers) can work concurrently on Views and Models with well-defined Controller interfaces.

Implementation Tips

- **Models** should contain only domain/state information and relevant business methods (e.g., *calculateFare()*).
- **Views** (Passenger/Driver/Administrator UIs) should focus on rendering data, capturing user inputs, and sending them to the controller.
- **Controllers** act as a mediator: they do not store data themselves but call services and orchestrate changes in the Model, then pass relevant information back to the View

By using MVC, SmartRide keeps its code organized and maintainable, particularly given multiple user-facing interfaces (passenger, driver, admin). This pattern ensures each layer can be developed or changed without causing ripple effects throughout the entire system, aligning with the project's scalability and reliability requirements.

6, Bootstrap Process

This section describes the initialization (bootstrap) sequence for the SmartRide Online Ride-Sharing System. It outlines the order in which classes and their instances are created during the startup of the system.

Step 1: System Initialization

- The AuthenticationManager class was first established to oversee secure access to the platform.
- The Database class is instantiated to retain information pertaining to users, automobiles, rides, payments, and reviews.

Step 2: Administrative and Fundamental Services Commencement

- The primary administrator account has been established, granting complete administrative control.
- LocationService: Configured to manage location tracking and GPS navigation.

Step 3: Service Initialization

- NotificationService: Configured to dispatch alerts to users and drivers via SMS, email, and push notifications.
- AnalyticsEngine: Initialized to analyze data, create reports, and do predictive analytics.

Step 4: User and Payment Initialization

- Pricing Strategy: Designed to compute tariffs dynamically depending on many variables including distance, demand, and time.
- Payment: Payment modules connected with secure third-party gateways (Credit Cards, E-wallets, Bank Transfers).

Step 5: User and Vehicle Setup

- Users (Passengers and Drivers) register through the AuthenticationManager, and user details are securely stored in the Database.
- Drivers register their vehicles, and vehicle details are stored and verified in the Database.

Step 6: Ride Booking and Matching

- RideRequest instances are produced when passengers request transportation, saving pickup and drop-off locations in the Database.
- Once a ride request is granted, the system constructs a Ride instance, assigning an appropriate driver based on the matching algorithm.

Step 7: Payment Processing

- Upon ride completion, the system initializes the Payment process, securely managing transactions via integrated payment gateways, with transaction records updated in the Database.

Step 8: Feedback and Rating

- Users and drivers submit feedback via the RatingReview class after each completed ride, and this data is stored for ongoing quality evaluation.

Step 9: Admin and Operational Management

- Admin accounts are established to administer the overall system operations, oversee user interactions, monitor ongoing activities, create reports, and assure continuous system functioning.

Step 10: Final System Check and Launch

- Conduct final operational checks and confirm the readiness of all system components.
- Officially launch the SmartRide platform for user interaction.

7, Verification

The suggested approach was validated by simulating many use cases described below.

7.1 Create an Account

This situation shows the interactions needed to create an account in SmartRide.

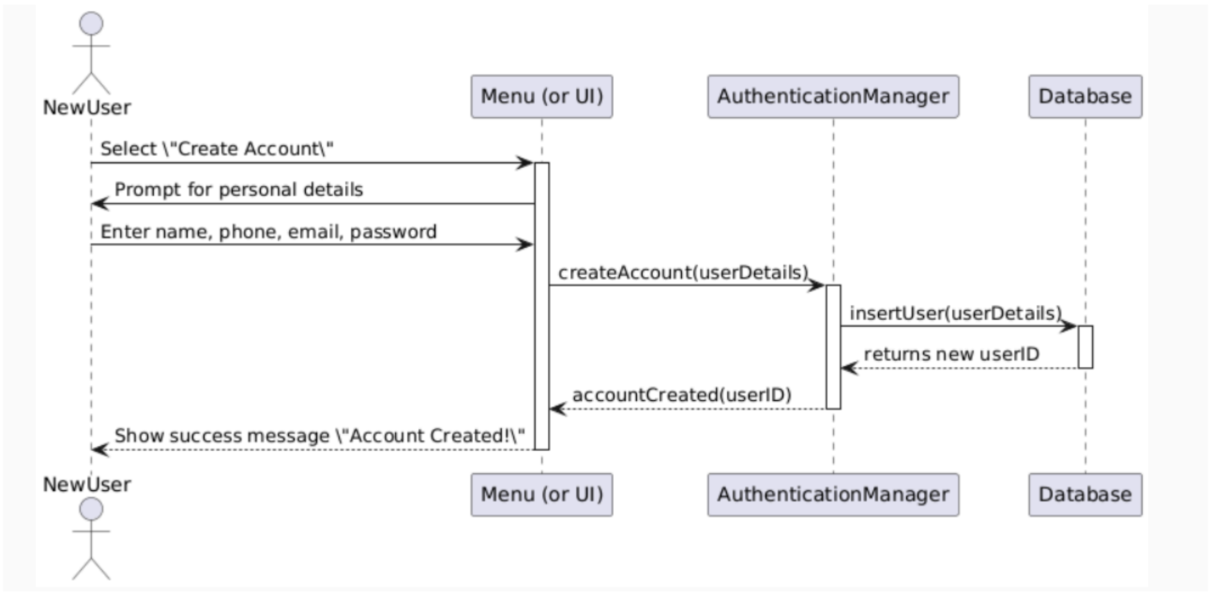

NewUser
Menu (or UI)
AuthenticationManager
Database
Select \"Create Account\"
Prompt for personal details
Enter name, phone, email, password
createAccount(userDetails)
insertUser(userDetails)
returns new userID
accountCreated(userID)
Show success message \"Account Created!\"

Figure 2 - Sequences diagram of Creating an Account

The caller (new user) provides the necessary personal details and indicates they want to sign up. A new User object is then created. The User object notifies the Database that a new account is being added, using the relevant data (e.g., name, email, phone number) to build the query string. The User object returns itself, allowing further actions (for example, linking a payment method or creating a ride request) to be performed afterward.

7.2 Book a Ride

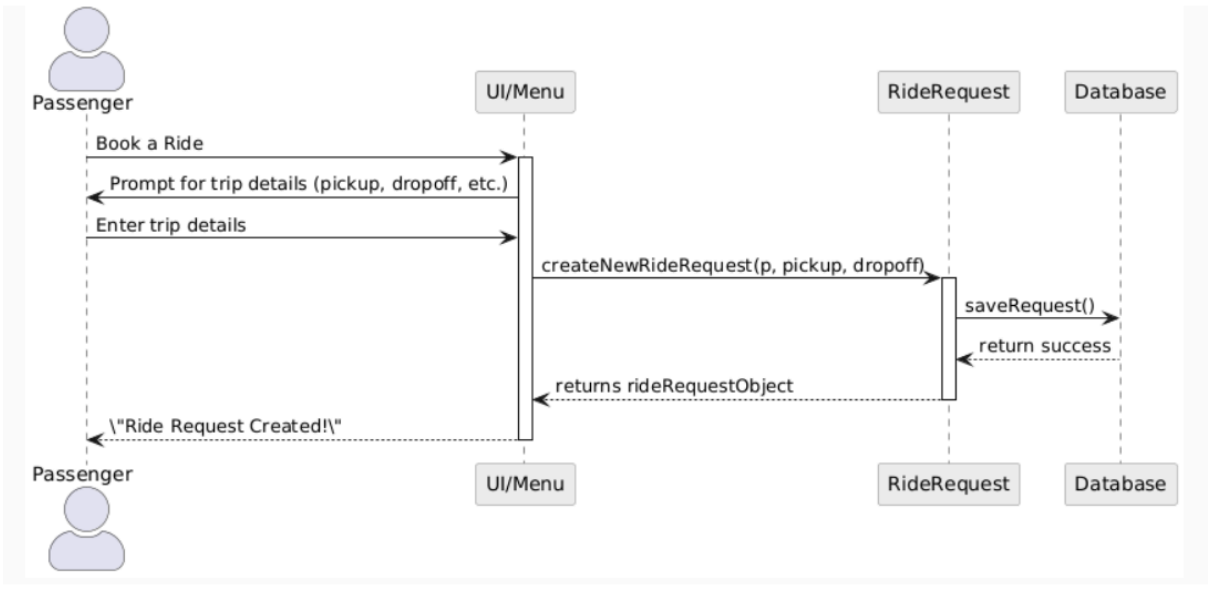

Passenger
UI/Menu
RideRequest
Database
Book a Ride
Prompt for trip details (pickup, dropoff, etc.)
Enter trip details
createNewRideRequest(p, pickup, dropoff)
saveRequest()
return success
returns rideRequestObject
\"Ride Request Created!\"

Figure 3 - Sequences diagram of Booking a Ride

This situation shows the interactions needed to Book a Ride. The passenger provides pickup and drop-off details, indicating they want to request a new ride. The system creates a RideRequest object. The RideRequest informs the Database of its creation and uses the relevant data—such as passenger information and location—to build the query for storage. The RideRequest then returns itself, which can be used for further actions, for example matching with a driver or calculating the fare.

7.3 Process Payment

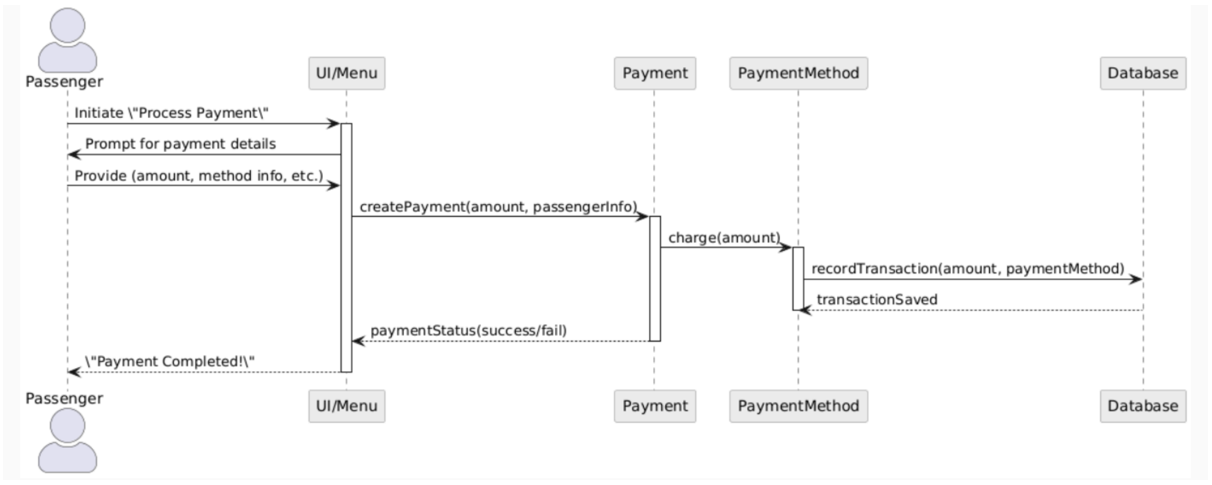
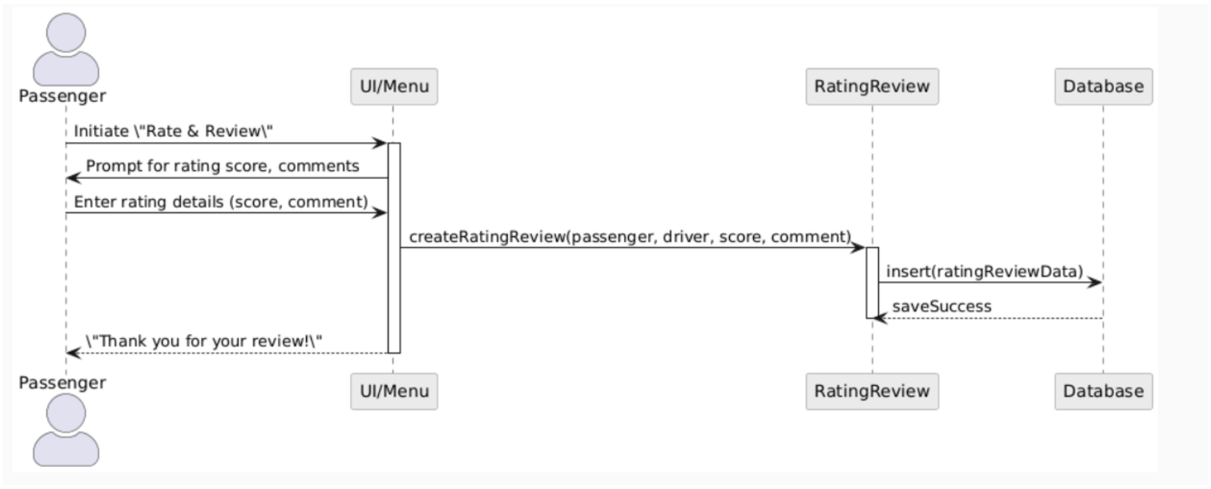

Passenger
UI/Menu
Payment
PaymentMethod
Database
Initiate \"Process Payment\"
Prompt for payment details
Provide (amount, method info, etc.)
createPayment(amount, passengerInfo)
charge(amount)
recordTransaction(amount, paymentMethod)
transactionSaved
paymentStatus(success/fail)
\"Payment Completed!\"

Figure 4 - Sequences diagram of Processing Payment

This situation shows the interactions needed to process a payment. The passenger enters their final fare details and chooses how to pay. A Payment object is created, which calls the PaymentMethod to handle the transaction. The PaymentMethod interacts with the Database to record the transaction and then returns the status. The Payment object provides the result back to the user interface, and the passenger is informed whether the payment succeeded or failed.

7.4 Rate & Review (Passenger -> Driver)



Passenger
UI/Menu
RatingReview
Database
Initiate \"Rate & Review\"
Prompt for rating score, comments
Enter rating details (score, comment)
createRatingReview(passenger, driver, score, comment)
insert(ratingReviewData)
saveSuccess
\"Thank you for your review!\"
Passenger
UI/Menu
RatingReview
Database

Figure 5 - Sequences diagram of Rating & Reviewing

This situation shows the interactions needed to rate and review a driver. The passenger provides a rating score and optional comments through the user interface. A RatingReview object is created, which informs the Database of its creation by inserting the relevant data (passenger ID, driver ID, rating score, and comment). Once saved, the system notifies the passenger that their feedback has been recorded.

8, Appendix

References

E., F. (2004). *Head First Design Patterns*. O'Reilly Media.

Erich, G., Richard, H., Ralph, J., & John, V. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Gang of Four.