VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

A red and blue logo

AI-generated content may be incorrect.

**Instructor: Phạm Kỳ Thái Trung**

**Class: 23K50201**

**Students:**  
**523K0010 – Phạm Lê Anh Khôi**

**523K0013 – Dương Thành Long**

**Signature**



Phạm Lê Anh Khôi

A signature of a person

AI-generated content may be incorrect.

Dương Thành Long

Contents

[PART 1: REQUIREMENT SPECIFICATION 6](#_Toc199361372)

[I. Introduction 6](#_Toc199361373)

[II. Project Overview 7](#_Toc199361374)

[1. Project Objectives 7](#_Toc199361375)

[2. Domain Vocabulary 8](#_Toc199361376)

[3. Project Type 10](#_Toc199361377)

[4. Goals 11](#_Toc199361378)

[5. Project Incentives 12](#_Toc199361379)

[6. Assumptions 13](#_Toc199361380)

[7. Scope 14](#_Toc199361381)

[8. Functional Requirements 16](#_Toc199361382)

[9. Non-functional Requirements 17](#_Toc199361383)

[III. Problem Domain 18](#_Toc199361384)

[1. Pain Points 18](#_Toc199361385)

[2. Domain Entities 19](#_Toc199361386)

[3. Actors 20](#_Toc199361387)

[4. List of Tasks 21](#_Toc199361388)

[IV. Data Model 23](#_Toc199361389)

[V. Tasks and Support 24](#_Toc199361390)

[1. Task 1: User Registration & Authentication 24](#_Toc199361391)

[2. Task 2: Ride Booking 25](#_Toc199361392)

[3. Task 3: Driver Assignment 26](#_Toc199361393)

[4. Task 4: Ride Status Tracking 27](#_Toc199361394)

[5. Task 5: Payment Processing & Receipt Generation 28](#_Toc199361395)

[6. Task 6: Operational Reporting 29](#_Toc199361396)

[VI. Workflow 30](#_Toc199361397)

[1. User Registration & Authentication 30](#_Toc199361398)

[2. Ride Booking 31](#_Toc199361399)

[3. Driver Assignment 32](#_Toc199361400)

[4. Ride Status Tracking 33](#_Toc199361401)

[5. Payment Processing & Receipt Generation 34](#_Toc199361402)

[6. Operational Reporting 35](#_Toc199361403)

[VII. Validation of Requirements 36](#_Toc199361404)

[1. Desk Reviews 36](#_Toc199361405)

[2. Traceability Matrix 37](#_Toc199361406)

[3. CRUD Check 38](#_Toc199361407)

[VIII. Possible Solutions 39](#_Toc199361408)

[1. Problem 1: Ride Matching Delays During Peak Hours 39](#_Toc199361409)

[2. Problem 2: Payment Failures or Discrepancies 40](#_Toc199361410)

[3. Problem 3: Fake Registrations or Fraudulent Use 41](#_Toc199361411)

[PART 2: OBJECT DESIGN 42](#_Toc199361412)

[I. Introduction 42](#_Toc199361413)

[1. Outlook of the Solution 42](#_Toc199361414)

[2. Documentation and Guidelines for Interfaces 42](#_Toc199361415)

[3. Definitions, Acronyms and Abbreviations 43](#_Toc199361416)

[II. Problem Analysis 44](#_Toc199361417)

[1. Assumptions 44](#_Toc199361418)

[2. Simplifications 45](#_Toc199361419)

[III. Candidate Classes 46](#_Toc199361420)

[1. Candidate Class List 46](#_Toc199361421)

[2. Class Justification 47](#_Toc199361422)

[3. UML Class Diagram 48](#_Toc199361423)

[4. CRC Cards 49](#_Toc199361424)

[a. Customer 49](#_Toc199361425)

[b. Driver 50](#_Toc199361426)

[c. Ride 51](#_Toc199361427)

[d. Payment 51](#_Toc199361428)

[e. Manager 52](#_Toc199361429)

[f. Account 52](#_Toc199361430)

[g. Vehicle 53](#_Toc199361431)

[h. Location 53](#_Toc199361432)

[i. Report 54](#_Toc199361433)

[IV. Design Quality 55](#_Toc199361434)

[1. Design Heuristics 55](#_Toc199361435)

[2. Design Patterns 56](#_Toc199361436)

[a. Creation Patterns 56](#_Toc199361437)

[b. Structural Patterns 58](#_Toc199361438)

[c. Behavioral Patterns 59](#_Toc199361439)

[V. Bootstrap Process 61](#_Toc199361440)

[1. Key Components 61](#_Toc199361441)

[2. System Initialization (Main) 62](#_Toc199361442)

[3. Customer Initialization 62](#_Toc199361443)

[4. Driver Initialization 62](#_Toc199361444)

[5. Account Initialization 63](#_Toc199361445)

[6. Ride Initialization 63](#_Toc199361446)

[7. Location Initialization 64](#_Toc199361447)

[8. Vehicle Initialization 64](#_Toc199361448)

[9. Payment Initialization 64](#_Toc199361449)

[10. Report Initialization 65](#_Toc199361450)

[VI. Verification 66](#_Toc199361451)

[1. Customer Registers and Logs In 66](#_Toc199361452)

[2. Customer Books a Ride 67](#_Toc199361453)

[3. Customer Tracks Ride 67](#_Toc199361454)

[4. Customer Processes Payment 68](#_Toc199361455)

[5. Manager Requests Report 68](#_Toc199361456)

[PART 3: Object Design Implementation & Reflection 69](#_Toc199361457)

[I. Detailed Design 69](#_Toc199361458)

[1. Customer Class 69](#_Toc199361459)

[2. Diver Class 69](#_Toc199361460)

[3. Ride Class 70](#_Toc199361461)

[4. Payment Class 70](#_Toc199361462)

[5. Account Class 71](#_Toc199361463)

[6. Data Holder Classes 71](#_Toc199361464)

[7. Controller Classes 71](#_Toc199361465)

[II. Discussion of Quality of Part 2 Design 72](#_Toc199361466)

[III. Lessons Learnt 73](#_Toc199361467)

[IV. Implementation - Source Code 74](#_Toc199361468)

[V. Implementation- Compilation and execution 75](#_Toc199361469)

[Meeting the requirements of the assignment specification 76](#_Toc199361470)

[Conclusion 78](#_Toc199361471)

# PART 1: REQUIREMENT SPECIFICATION

# Introduction

- SmartRide is a transportation service company operating in a busy metropolitan city. The company connects customers with available drivers for a fee, using cars and motorbikes. Currently, SmartRide’s operations are managed manually, leading to inefficiencies such as long wait times, lost ride opportunities, and slow payment handling. To address these challenges and to prepare for future growth, SmartRide has commissioned the development of a new Online Ride-Sharing Platform (ORSP).

- SESoft Consulting has been tasked with designing and developing the ORSP to automate and enhance SmartRide’s operations, focusing on improving ride booking, driver assignment, real-time tracking, and online payments. This system is expected to deliver a faster, more reliable, and user-friendly experience for both customers and drivers, while also providing managerial insights to support business decision-making.

# Project Overview

## Project Objectives

* **Automate ride-booking** by allowing customers to easily request rides through a digital platform.
* **Reduce customer wait times** by efficiently matching available drivers to ride requests in real time.
* **Introduce secure online payment processing** to replace manual cash handling.
* **Provide real-time tracking** for customers to monitor driver location and estimated arrival times.
* **Enable account management** for customers and drivers through user profiles.
* **Generate operational reports** to help SmartRide’s management monitor service performance and demand trends.
* Initially, the system will focus on the core functionality, with the flexibility to add new features such as ride-sharing options, loyalty programs, and promotional offers in the future.

## Domain Vocabulary

| **Term** | | **Definition** |
| --- | --- | --- |
| **ORSP** | | Online Ride-Sharing Platform — the digital system enabling ride booking, tracking, payment, and reporting. |
| **Customer** | | A user of the ORSP who requests rides, tracks driver location, and makes payments. |
| **Driver** | | A service provider registered with the ORSP who accepts ride requests and transports customers. |
| **Ride** | | A single trip from pickup to drop-off, representing the core business transaction. |
| **Vehicle** | | A data holder for a driver’s mode of transport (car or motorbike), including its details. |
| **Location** | | Geographical coordinates and address information for pickup and drop-off points. |
| **ETA** | | Estimated Time of Arrival — the calculated time for a driver to reach the pickup location. |
| **Payment** | | The process and record of transferring fare from customer to driver, managed via a third-party gateway. |
| **Account** | | Authentication and profile information for all users (customers, drivers, managers). |
| **Manager** | | Administrative user who oversees operations, accesses reports, and manages system settings. |
| **Report** | | Aggregated data summaries (e.g., ride volumes, revenues) provided for managerial insight. |
| **Payment Gateway** | | External service integrated into the ORSP for secure payment processing (e.g., Stripe, PayPal). |
| **GPS Navigation Service** | | Third-party system providing real-time location tracking and routing for drivers. |
| **Notification** | | Real-time messages sent to customers and drivers (e.g., ride status updates, payment confirmations). |
| **Availability** | | A driver’s current status (online/offline) indicating readiness to accept new ride requests. |
| **Scalability** | The system’s ability to handle increasing numbers of users, rides, and transactions without performance degradation. | |

## Project Type

The SmartRide ORSP engagement is a **custom software development** project with the following characteristics:

* **Domain:** Transportation / Ride-Sharing Platform
* **Application Type:**
  + **Web Application**: Responsive portal for managers and administrative users
  + **Mobile Applications**: Native or cross-platform apps for iOS and Android, used by customers and drivers
* **Architecture Style:** Client-server, cloud-hosted microservices
* **Technology Stack (Indicative):**
  + Backend: ASP.NET Core with Entity Framework Core
  + Frontend: React (web) and React Native (mobile)
  + Database: Microsoft SQL Server
  + Hosting: Azure or AWS

## Goals

|  |
| --- |
| * Enable customers to create accounts, book rides, and pay online. |
| * Enable drivers to create profiles, accept ride requests, and navigate to customers. |
| * Provide real-time GPS tracking and ETA updates to customers. |
| * Automate payment handling, receipts, and confirmations. |
| * Allow managers to access reports on rides, revenue, and demand trends. |
| * Ensure system security, reliability, and scalability for future growth. |

## Project Incentives

* **Reduce Customer Churn:** By cutting down wait times and offering real-time tracking, SmartRide aims to prevent lost bookings and improve overall customer loyalty.
* **Increase Ride Completion Rate:** Automated driver matching and dynamic dispatch will help fulfill more ride requests, directly boosting revenue.
* **Streamline Operations:** Eliminating manual processes for dispatch and payments will lower administrative overhead and reduce error rates.
* **Enhance Driver Utilization:** Faster, fairer assignment of rides will keep drivers active and reduce idle time, improving their earnings and retention.
* **Enable Data-Driven Growth:** Built-in reporting and analytics will empower management to spot demand trends, optimize pricing, and plan for peak periods.
* **Future-Proof the Platform:** A modular, scalable system foundation will allow SmartRide to roll out new features (shared rides, loyalty programs, new vehicle types) with minimal rework.

## Assumptions

* **City Coverage:** SmartRide initially operates within a single metropolitan area; no immediate multi-city expansion is planned.
* **User Technology Proficiency:** All customers and drivers possess basic smartphone literacy and can install and use mobile apps for booking rides, navigation, and payments.
* **Hardware Availability:** Drivers use their own GPS-enabled smartphones. Server infrastructure (cloud or on-premises) will be provisioned separately and is outside the scope of this project.
* **Ride Type Stability:** Only cars and motorbikes are supported at launch. Any new vehicle types (e.g., vans, e-bikes) will be added via future system updates.
* **Customer Interactions:** The system handles ride requests, real-time tracking, and payments. Customer support inquiries (e.g., refunds, complaints) are managed through external channels.
* **Third-Party Payment Gateway:** SmartRide integrates with an existing PCI-compliant payment provider; the platform does not process or store raw payment card data.
* **External Navigation Service:** A reliable GPS/navigation API is available for real-time location tracking and ETA calculations.
* **User Roles:** The system supports three primary user roles—Customer, Driver, and Manager, and each must register an Account before using the platform.

## Scope

**Included**

* **User Account Management:**
  + Registration, login, and profile management for Customers, Drivers, and Managers.
* **Ride Lifecycle:**
  + Ride request submission with pickup and drop-off locations.
  + Automatic driver matching and assignment.
  + Real-time GPS-based tracking and ETA updates.
  + Ride status updates (requested, accepted, in-progress, completed).
* **Vehicle Management:**
  + Registration and maintenance of Driver’s Vehicle details (type, license plate, status).
* **Payment Processing:**
  + Integration with a third-party, PCI-compliant payment gateway.
  + Fare calculation, transaction execution, and digital receipt generation.
* **Reporting & Analytics:**
  + Generation of operational reports (ride volumes, revenue, driver activity) for Managers.
* **Notifications:**
  + Real-time alerts to Customers and Drivers for ride status changes and payment confirmations.

**Excluded**

* **Hardware Procurement:** Server infrastructure, driver devices, or in-vehicle hardware.
* **Customer Support Workflows:** Ticketing system, dispute resolution, or refund management.
* **Dynamic Pricing & Promotions:** Surge pricing algorithms, loyalty rewards, or discount campaigns.
* **Shared Rides & Pooling:** Carpooling or multi-passenger matching.

## Functional Requirements

* **Enable Customer Account Management:** Customers must be able to create, update, and manage their user accounts, including personal details, payment information, and ride history.
* **Enable Driver Account Management:** Drivers must be able to register, update, and manage their profiles, vehicle information, and availability status.
* **Support Ride Booking:** Customers must be able to input pickup and drop-off locations and submit ride requests through the platform.
* **Automate Driver Assignment:** The system must automatically locate available drivers based on proximity, availability, and estimated arrival times, and assign them to incoming ride requests.
* **Provide Real-time Tracking:** Both customers and drivers must have access to live GPS tracking to monitor driver location, route progress, and estimated arrival times.
* **Enable Secure Online Payments:** Customers must be able to securely complete ride payments through an integrated online payment gateway, with digital receipts generated upon successful transactions.
* **Generate Operational Reports:** Managers must be able to access system-generated reports summarizing ride volumes, driver activity, peak usage periods, and other relevant operational metrics.
* **Support Notifications and Updates:** The system must send real-time updates to customers and drivers about ride status changes, estimated arrival times, and payment confirmations.
* **Ensure Ride History Management:** Customers and drivers must be able to view their respective ride histories, including trip details, payment records, and ratings.
* **Facilitate System Scalability and Maintenance:** The system must be designed to easily accommodate future expansions, such as adding new cities, new vehicle types, or new customer loyalty programs.

## Non-functional Requirements

* **Performance and Scalability**  
  The system must handle high volumes of concurrent users, especially during peak hours, without significant performance degradation. It must be scalable to support future expansion to additional cities and increased numbers of customers and drivers.
* **Reliability and Availability**  
  The platform must maintain a minimum uptime of 99.9% to ensure customers and drivers can access the service at all times. System recovery mechanisms must be in place to handle unexpected failures with minimal service interruption.
* **Security**  
  The system must protect user data through encryption and secure authentication methods. Integration with third-party payment gateways must comply with industry standards for financial data protection (e.g., PCI DSS compliance).
* **Usability**  
  The platform must provide a clean, intuitive user interface that is easy to navigate for customers, drivers, and managers. Key tasks (such as booking a ride, accepting a ride, or making a payment) must be easily completed with minimal steps and training.
* **Maintainability**  
  The system must be designed with a modular architecture to allow easy updates, bug fixes, and the addition of new features without significant system downtime.
* **Interoperability**  
  The ORSP must integrate seamlessly with external systems such as GPS navigation services and third-party payment gateways without requiring manual intervention.
* **Responsiveness**  
  Real-time features such as ride tracking, driver assignment, and notification delivery must occur with minimal latency to enhance the user experience.

# Problem Domain

## Pain Points

SmartRide’s current manual operations present several critical challenges that negatively impact customer satisfaction, driver efficiency, and overall business growth:

* **Long Wait Times**: Customers often experience significant delays in securing a ride, especially during peak hours, due to the lack of an automated driver assignment process.
* **Missed Ride Opportunities**: Without an efficient system to match customers with available drivers, many potential rides are lost, leading to lost revenue and frustrated customers.
* **Manual Payment Processing**: Payments are currently handled manually between drivers and customers, which is time-consuming, error-prone, and inconvenient for both parties.
* **Driver Assignment Inefficiencies**: During periods of high demand, there is no organized method for quickly identifying and assigning nearby available drivers to customers.
* **Limited Visibility for Customers**: Customers have no way to track where their driver is or receive accurate arrival time estimates, leading to uncertainty and dissatisfaction.
* **High Administrative Overhead**: Manual ride tracking, payment recording, and business reporting require significant administrative effort, slowing down operations and decision-making.
* **Scalability Limitations**: The current manual approach is not scalable. As customer demand grows, the company struggles to handle increased ride requests without compromising service quality.

## Domain Entities

| **Entity** | **Description** |
| --- | --- |
| **Customer** | A user who requests and pays for rides via the platform. |
| **Driver** | A service provider who accepts ride requests and transports customers. |
| **Ride** | A record of a single trip, including pickup/drop-off locations, status, and associated users. |
| **Vehicle** | Details about a driver’s transport (car or motorbike), including type and registration data. |
| **Location** | Geographic information (latitude, longitude, address) for ride pickup and drop-off points. |
| **Payment** | Transaction details for a ride, including fare, method, status, and receipt information. |
| **Account** | Authentication and profile information for all users (customers, drivers, managers). |
| **Manager** | An administrative user who oversees operations and accesses reports and analytics. |
| **Report** | Aggregated summaries of rides, payments, and performance metrics for managerial review. |

## Actors

* **Customer:** An individual who uses the platform to book rides, track driver locations, make payments, and manage their ride history and personal profile.
* **Driver:** A registered individual who uses the platform to receive ride requests, accept bookings, navigate to pickup and drop-off locations, and manage their availability and profile.
* **Manager:** A SmartRide administrative user who oversees system operations, reviews reporting and analytics, manages customer or driver accounts when necessary, and monitors service performance trends.
* **Payment Gateway** *(External Actor):* A third-party service responsible for securely processing customer payments. SmartRide will integrate with an external gateway to manage financial transactions without directly handling sensitive payment data.
* **GPS Navigation Service** *(External Actor):* A third-party service used to provide real-time location tracking, route optimization, and estimated arrival times for both customers and drivers.

## List of Tasks

**User Account Management**

* Register new Customer, Driver, and Manager accounts
* Log in and log out of the system
* Update and manage user profiles and credentials

**Ride Booking**

* Enter pickup and drop-off locations
* Submit ride requests
* Display available vehicle types and ETAs

**Driver Assignment**

* Locate nearest available Driver
* Notify Driver of incoming ride request
* Allow Driver to accept or decline a ride

**Real-Time Tracking**

* Continuously update Driver’s GPS location
* Recalculate and display ETA to Customer
* Notify Customer and Driver of status changes

**Ride Status Management**

* Transition a Ride through statuses (Requested → Accepted → In-Progress → Completed → Rated)
* Record timestamps for lifecycle events

**Navigation Support**

* Provide turn-by-turn directions to Driver
* Re-route dynamically based on traffic or route changes

**Payment Processing**

* Calculate fare based on distance, time, and vehicle type
* Integrate with third-party payment gateway for secure transactions
* Generate and deliver digital receipts

**Notifications & Alerts**

* Send real-time notifications for ride confirmations, arrivals, and completions
* Alert users to payment status and account changes

**Reporting & Analytics**

* Aggregate ride and payment data
* Generate on-demand operational reports for Managers (e.g., daily summaries, peak-hour trends)

**Error Handling & Validation**

* Validate user inputs (locations, payment details, profile data)
* Gracefully handle and report system or integration failures

# Data Model

A diagram of a company

AI-generated content may be incorrect.

# Tasks and Support

## Task 1: User Registration & Authentication

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Allow new users (Customers, Drivers, Managers) to create an account and log in securely. |
| **Precondition** | User has downloaded the SmartRide app or accessed the web portal. |
| **Frequency** | Once per user (registration), then at each subsequent app launch or session (login). |
| **Critical** | Insecure or failed authentication exposes privacy/data or prevents use of the system. |
| **Work area** | Registration & Login screens (mobile/web). |
| **Subtask** | Example Solution |
| **1. Collect user details** | Present form for name, email/phone, password, (driver: license) |
| **2. Validate input** | Check mandatory fields, email format, password strength |
| **3. Create account record** | Invoke Account.create() → store credentials in user database |
| **4. Send confirmation (optional)** | Email/SMS with verification link or code |
| **5. Authenticate on login** | Account.login() → check credentials → issue session token |
| **Variants** |  |
| 1a. **Social Login** | User chooses Google/Facebook SSO → OAuth flow → create or link local account. |
| 1b. **Password Reset** | User forgot password → requests reset link → completes password change. |

## Task 2: Ride Booking

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Enable a Customer to request a new ride by specifying pickup and drop-off locations. |
| **Precondition** | User is authenticated and has a valid payment method on file. |
| **Frequency** | Every time a Customer needs a trip. |
| **Critical** | Invalid locations or missing payment info block ride creation and frustrate users. |
| **Work area** | “Book a Ride” screen with map and form. |
| **Subtask** | Example Solution |
| **1. Enter pickup location** | Map-based search / GPS “use my current location” |
| **2. Enter drop-off location** | Address autocomplete + map pin drop |
| **3. Validate service area** | Check coordinates against supported city polygon |
| **4. Show vehicle types & ETAs** | Query ETA service for cars/motorbikes |
| **5. Confirm ride request** | Ride.create() + persist pickup/drop-off and customer ID |
| **Variants** |  |
| **2a. Schedule for later** | Customer selects future date/time → system enqueues ride request. |
| **2b. Fare estimate only** | Customer previews fare and ETA without booking. |

## Task 3: Driver Assignment

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Automatically match a new Ride request to the most suitable available Driver. |
| **Precondition** | A Ride has been created and at least one Driver is marked available nearby. |
| **Frequency** | Once per new Ride request. |
| **Critical** | Poor matching leads to long waits or ride cancellations. |
| **Work area** | Backend dispatch service (RideManager). |
| **Subtask** | Example Solution |
| **1. Locate nearby drivers** | Spatial query: Driver.location within radius of pickup point |
| **2. Filter by availability** | Exclude drivers marked “busy” or “offline” |
| **3. Rank by ETA & fairness** | Sort by estimated arrival time and recent assignment count |
| **4. Dispatch assignment** | Ride.assignDriver(driverId) → notify driver via push/SMS |
| **5. Await driver response** | Timeout after N seconds → retry or escalate to next candidate |
| **Variants** |  |
| **3a. Manual override** | Manager forces assignment to a specific driver in special cases. |
| **3b. Shared ride matching** | (Future) match multiple customers to one driver. |

## Task 4: Ride Status Tracking

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Provide both Customer and Driver with real-time updates on ride progress and ETA. |
| **Precondition** | Driver has accepted the Ride and GPS updates are being sent. |
| **Frequency** | Continuous, throughout the lifecycle of each Ride. |
| **Critical** | Delays or inaccuracies in tracking decrease user trust and can cause missed pickups. |
| **Work area** | Mobile apps (Customer & Driver), backend location service. |
| **Subtask** | Example Solution |
| **1. Driver sends GPS heartbeat** | App → Location.update(latitude, longitude) |
| **2. Update Ride record** | Ride.updatePosition() & recalculate ETA |
| **3. Push notification to Customer** | Real-time socket message or push notification |
| **4. Display map & ETA** | Mobile UI map redraw with driver marker and time label |
| **Variants** |  |
| **4a. Intermittent connectivity** | Buffer location updates locally → sync when online. |
| **4b. Alternative routing** | Driver manually updates route → recalculate ETA. |

## Task 5: Payment Processing & Receipt Generation

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Charge the Customer for a completed Ride and generate a digital receipt. |
| **Precondition** | Ride status = “Completed” and Customer has a valid payment method. |
| **Frequency** | Once per Ride completion. |
| **Critical** | Failed transactions lead to revenue loss and require manual follow-up. |
| **Work area** | Backend payment service (PaymentProcessor) and Customer app. |
| **Subtask** | Example Solution |
| **1. Calculate fare** | Formula: base fare + distance × rate + time × rate |
| **2. Initiate transaction** | Payment.process(paymentInfo) → call third-party gateway API |
| **3. Handle gateway response** | On success → Payment.markPaid(); on failure → retry/alert |
| **4. Generate digital receipt** | Payment.generateReceipt() → PDF or in-app view |
| **5. Notify Customer** | Push notification + email with receipt link |
| **Variants** |  |
| **5a. Wallet top-up** | Deduct from in-app wallet balance before fallback to card. |
| **5b. Split fare** | Customer splits payment among multiple riders. |

## Task 6: Operational Reporting

| **Aspect** | **Description** |
| --- | --- |
| **Purpose** | Provide Managers with insights on ride volumes, revenue, and driver performance over time. |
| **Precondition** | System has processed multiple Rides and Payments to aggregate data. |
| **Frequency** | On-demand or scheduled (e.g., daily at midnight). |
| **Critical** | Inaccurate or delayed reports hinder strategic decision-making. |
| **Work area** | Manager portal / reporting dashboard. |
| **Subtask** | Example Solution |
| **1. Query ride/payment data** | ReportGenerator.fetchRides(startDate, endDate) |
| **2. Aggregate metrics** | Calculate totals, averages, peak-hour counts |
| **3. Format report** | Render as table/chart in dashboard |
| **4. Export or schedule delivery** | CSV/PDF export or automated email dispatch |
| **Variants** |  |
| **6a. Custom date range** | Manager selects arbitrary start/end dates. |
| **6b. Performance alerts** | System automatically flags underperforming metrics via email. |

# Workflow

## User Registration & Authentication

A diagram of a company

AI-generated content may be incorrect.

## Ride Booking

A diagram of a flowchart

AI-generated content may be incorrect.

## Driver Assignment

A screenshot of a diagram

AI-generated content may be incorrect.

## Ride Status Tracking

A diagram of a service

AI-generated content may be incorrect.

## Payment Processing & Receipt Generation

A screenshot of a diagram

AI-generated content may be incorrect.

## Operational Reporting

A diagram of a report

AI-generated content may be incorrect.

# Validation of Requirements

## Desk Reviews

Conduct systematic, internal reviews of the requirements specification to ensure completeness, consistency, clarity, and alignment with stakeholder needs before broader stakeholder involvement.

**Participants:**

* **Lead Business Analyst** (chairs the review)
* **Domain Expert** (ride-sharing operations specialist)
* **Technical Architect** (assesses technical feasibility)
* **QA Lead** (identifies testability and ambiguity issues)
* **UX Designer** (reviews usability and workflow clarity)

**Scope of Review:**

* **Completeness:** Verify that all major business processes (registration, booking, assignment, tracking, payment, reporting) are covered by requirements.
* **Consistency:** Ensure terminology is used uniformly (e.g., “Ride,” “Customer,” “Driver,” “ETA”).
* **Clarity:** Check that each requirement is unambiguous, phrased in active voice, and has clear acceptance criteria.
* **Feasibility:** Assess whether non-functional targets (performance, availability, security) are realistic given the technology stack.
* **Traceability:** Confirm that each functional requirement can be traced back to a business goal or pain point.

## Traceability Matrix

| **Requirement** | **Quality Attribute(s)** | **Assessment Metric** |
| --- | --- | --- |
| Customer can request and book a ride via the app/web | • Usability  • Performance  • Reliability | • <2 s for ride-request submission• ≥ 95% successful bookings• <1% booking errors |
| Driver can receive and accept ride requests | • Performance  • Usability | • Notification delivered <1 s after assignment• ≥ 98% driver-app uptime |
| Real-time driver tracking and ETA updates for customers | • Performance  • Reliability | • GPS location updated every 5 s• ≥ 99% location accuracy• <3 s ETA refresh |
| Customer can complete secure online payment | • Security  • Performance | • PCI-DSS compliance• <3 s transaction processing time• 0% unencrypted PII |
| Manager can generate operational reports on demand | • Scalability  • Performance | • Reports for 30 days of data generated <5 s• ≥ 99% data accuracy |
| System must support ≥ 10 000 concurrent users | • Scalability  • Availability | • Sustained throughput at 10 000 sessions/minute• 99.9% uptime in 30-day window |
| Notifications of ride status changes reach users reliably | • Reliability  • Performance | • ≥ 99.5% notification delivery success• Delivery latency <2 s |

## CRUD Check

| **Task / Actors** | **Customer** | **Driver** | **Cashier** | **System** | **Manager** |
| --- | --- | --- | --- | --- | --- |
| Register Account | C, R, U, D | C, R, U, D | R |  |  |
| Book a Ride | C, R | R |  | R, U | R |
| Accept Ride Request |  | C, R, U |  | R, U | R |
| Track Ride in Real-Time | R, U | R, U |  | R, U |  |
| Make Payment | C, R |  | R, U | C, R, U | R, U |
| Generate Receipt | R |  | R, U | C, R | R, U |
| Generate Reports |  |  |  | C, R | R, U, D |
| Manage Driver Information |  | C, R, U, D |  |  | R, U, D |

# Possible Solutions

## Problem 1: Ride Matching Delays During Peak Hours

**Description:**  
During high-demand periods (e.g., rush hour, weekends), customers may experience delays in being matched with available drivers. This can result in poor user experience, ride cancellations, and reduced customer retention.

**Possible Solutions:**

1. **Dynamic Pricing (Surge Pricing):**
   * Implement surge pricing algorithms that increase fare rates during peak hours to incentivize more drivers to become available.
   * Communicate surge pricing transparently to users before ride confirmation.
2. **Driver Incentive System:**
   * Offer bonuses or higher commissions for drivers accepting rides during high-demand windows.
   * Introduce gamification (e.g., reward points) to encourage driver availability.
3. **AI-based Demand Forecasting:**
   * Use machine learning models to predict demand spikes based on historical data, location, weather, and events.
   * Preemptively alert or position drivers in expected high-demand zones.

## Problem 2: Payment Failures or Discrepancies

**Description:**  
Users may face issues during payment processing due to connectivity errors, gateway downtime, or inconsistent fare calculations. This can lead to failed transactions, overcharge, or underpayment.

**Possible Solutions:**

1. **Multi-Gateway Payment Support:**
   * Integrate multiple payment gateway providers (e.g., Stripe, PayPal, Razorpay) with failover mechanisms to ensure continuity if one service fails.
2. **Pre-Authorization of Fare Estimate:**
   * Lock a fare estimate with a margin before ride starts; finalize fare at end with fare breakdown.
   * Handle discrepancies with automated dispute resolution or escalation to customer service.
3. **Payment Retry and Wallet Option:**
   * Offers retry mechanisms and fallback to in-app wallet balance for failed transactions.
   * Notify the user via app and email with detailed receipts and refund process if needed.

## Problem 3: Fake Registrations or Fraudulent Use

**Description:**  
The system may be vulnerable to botted accounts or malicious users creating fake rider or driver accounts to exploit promotions or conduct fraud.

**Possible Solutions:**

1. **Phone and Email Verification:**
   * Enforce OTP-based verification during registration.
   * Require verified email addresses and phone numbers for account creation.
2. **Document and Face Verification for Drivers:**
   * Implement KYC (Know Your Customer) checks including driver's license, ID proof, and real-time selfie matching.
   * Use third-party ID verification services (e.g., Onfido, Jumio).
3. **Behavioral Analysis & Fraud Detection Algorithms:**
   * Monitor suspicious patterns like high ride cancellation rate, location spoofing, or frequent promo code use.
   * Apply AI-driven fraud detection systems to flag and block such activities automatically.

# PART 2: OBJECT DESIGN

# Introduction

## Outlook of the Solution

In this phase we translate the high-level requirements into an initial object-oriented design that captures the static structure and collaboration of core system components. We will:

* Identify and define the principal **classes** (both domain and control/service objects) needed to realize ride-sharing workflows.
* Illustrate their inter-relationships via a UML class diagram.
* Describe each class’s responsibilities and collaborators through CRC (Class-Responsibility-Collaborator) cards.
* Show how objects are instantiated and composed at system bootstrap.
* Verify design adequacy through representative interaction scenarios.

This design aims to be both **flexible**—to accommodate future features such as pooled rides or promotions—and **robust**, enforcing clear separation of concerns and high cohesion.

## Documentation and Guidelines for Interfaces

All public interfaces (methods, events, service endpoints) must adhere to SESoft’s standard conventions:

* **Naming:** PascalCase for classes and methods; camelCase for parameters.
* **Parameters & Return Types:** Explicit, strongly-typed definitions with clear domain types (e.g., Ride, Location).
* **Error Handling:** Methods must throw domain-specific exceptions (e.g., NoDriversAvailableException) rather than generic errors.
* **Asynchronous Operations:** Long-running or I/O-bound methods (e.g., payment processing, GPS updates) must provide async variants.
* **Documentation:** Each interface element is documented with XML-style comments including <summary>, <param>, <returns>, and <exceptions>.

## Definitions, Acronyms and Abbreviations

| **Term / Acronym** | **Definition** |
| --- | --- |
| **ORSP** | Online Ride-Sharing Platform. |
| **CRC Card** | Class-Responsibility-Collaborator design tool. |
| **UML** | Unified Modeling Language—standard for visualizing object-oriented designs. |
| **DTO** | Data Transfer Object—simple data holder used for communication between layers. |
| **API** | Application Programming Interface—contract exposing class operations. |
| **GPS** | Global Positioning System—used for real-time location tracking. |
| **ETA** | Estimated Time of Arrival—driver’s projected arrival time. |
| **SRP** | Single Responsibility Principle—each class should have one reason to change. |
| **DTO** | Data Transfer Object—plain object for carrying data. |

# Problem Analysis

The primary problem that this project addresses is the development of an Online Ride-Sharing Platform (ORSP) for SmartRide, aimed at automating core operations from user registration through ride completion and payment. In more detail, the project context is as follows:

SmartRide currently operates a manual dispatch and payment system in a busy metropolitan area, leading to long customer wait times, missed ride opportunities, and cumbersome cash-based transactions. During peak hours, the imbalance between ride requests and available drivers causes customer frustration and lost revenue. Additionally, all payments are processed by hand, introducing delays and reconciliation errors.

## Assumptions

| **ID** | **Assumption** |
| --- | --- |
| A1 | It is assumed that the platform will support up to 10,000 concurrent users during peak hours without requiring additional infrastructure changes. |
| A2 | All Customers and Drivers possess basic smartphone literacy and have compatible GPS-enabled devices for running the SmartRide mobile app. |
| A3 | Drivers will use their own personal vehicles (cars or motorbikes) and smartphones; the project scope excludes procurement of any hardware devices. |
| A4 | Only two vehicle types (car and motorbike) are supported in the initial release; addition of new vehicle categories will be handled via future system updates. |
| A5 | The system integrates with a reliable third-party GPS/navigation service for real-time tracking and ETA calculations; no in-house map or routing engine is required. |
| A6 | Online payment transactions are processed through a PCI-compliant third-party payment gateway; SmartRide will not store raw payment card data. |
| A7 | Customer support (e.g., dispute resolution, refunds) is managed outside the ORSP and is not part of the software’s core functionality. |
| A8 | Managers, Drivers, and Customers must register and verify their accounts before using platform features; unverified users cannot book or accept rides. |

## Simplifications

During the analysis of the SmartRide Online Ride-Sharing Platform (ORSP), several simplifying assumptions were introduced to streamline the project’s scope and focus development on the core domain features, thereby enhancing clarity and feasibility (Rich & Waters, 1982).

* **Fixed Service Area and User Capacity:**  
  We assume SmartRide will initially operate within a single metropolitan region and support up to 10,000 concurrent users during peak periods. This allows infrastructure planning without immediately tackling multi-region scalability or extreme load scenarios.
* **Limited Vehicle Types:**  
  Only two vehicle categories, cars and motorbikes—are supported in the initial release. By fixing vehicle types, we avoid designing extensible fleet-management modules or dynamic vehicle classification until later phases.
* **Simplified Ride Rules:**  
  Shared rides (pooling) and surge-pricing mechanisms are out of scope for launch. Every ride is a single-passenger trip at a fixed fare structure, eliminating the need to model multi-leg journeys or complex dynamic fare calculations.
* **Third-Party Service Reliance:**  
  The system leverages an existing, reliable GPS/navigation API for mapping and ETA, and a PCI-compliant payment gateway for transactions. We do not develop in-house routing engines or direct handling of payment card details.
* **Basic User Onboarding:**  
  All users (customers, drivers, managers) must create and verify an account before accessing platform features. No guest checkout or unregistered booking is permitted, which simplifies authentication flows and data integrity.
* **Standard UI Guidelines:**  
  We adhere to SESoft Consulting’s existing User Interface Standards (Ref: SESoft-UI-Standards-2025) without creating new design guidelines. This ensures consistency across screens and accelerates front-end development.

By adopting these simplifications, the project team can concentrate on delivering a robust, end-to-end ride-sharing experience—covering registration, booking, dispatch, tracking, and payment—while deferring non-essential complexities to future iterations.

# Candidate Classes

## Candidate Class List

**Domain Entities (Core Data Classes)**

* **Customer**
* **Driver**
* **Manager**
* **Ride**
* **Payment**
* **Account**

**Data-Holder / Value Classes**

* **Vehicle**
* **Location**
* **Report**

**Controller / Coordinator Classes**

* **RideManager**
* **PaymentProcessor**
* **UserAccountManager**
* **ReportGenerator**

**External-Service Adapters / Interfaces**

* **NotificationService** (push/SMS/Email notifications)
* **NavigationService** (map routing & ETA)
* **PaymentGateway** (third-party payment integration)
* **GPSService** (real-time location feed)

## Class Justification

**Customer**: Represents the end user of the ride-sharing service who initiates ride requests, provides payment information, and consumes real-time tracking and notifications. Encapsulating customer behaviors in a dedicated class ensures clear separation of user-facing logic (e.g., booking, viewing rides) from underlying system workflows.

**Driver**: Models the service provider entity responsible for accepting ride assignments, updating trip status, and interacting with navigation services. A separate Driver class encapsulates availability management, identity verification, and performance tracking specific to drivers.

**Ride:** Serves as the core transactional entity linking Customers and Drivers. It holds state transitions (requested → accepted → in-progress → completed) and aggregates associated Location data and Payment details. Centralizing ride logic in its own class simplifies coordination of status updates and ETA calculations.

**Payment:** Encapsulates all aspects of fare computation, transaction processing, and receipt generation via a third-party gateway. Isolating payment logic supports extension to multiple payment methods (credit card, wallet) and compliance with security standards without polluting other domain classes.

**Manager:** Represents administrative users who require access to aggregated system data and reporting features. By treating managers as a distinct class, we can enforce role-based access, audit actions, and cleanly separate operational analytics from standard user workflows.

**Account:** Abstracts authentication and profile management for all user types (Customer, Driver, Manager). A unified Account class prevents duplication of login logic and credentials handling across multiple actor classes, adhering to DRY and Single Responsibility principles.

**Vehicle** (Data Holder)**:** Holds static details of the transport units (type, plate number) linked to a Driver. As a simple data structure, it cleanly separates vehicle attributes from driver behavior and supports future expansion (e.g., new vehicle categories).

**Location** (Data Holder)**:** Stores geospatial coordinates and address information for pickup and drop-off points. Encapsulating Location details in its own class enables reuse across ride booking, tracking, and ETA computations without redundancy.

**Report** (Data Holder)**:** Aggregates metrics on ride volumes, revenue, and driver performance for managerial review. As a distinct class, Report structures analytics data and formats it for dashboards or export, decoupling reporting concerns from real-time operations.

## UML Class Diagram

A diagram of a company

AI-generated content may be incorrect.

## CRC Cards

### Customer

| **Description** | A user who books rides, tracks status, and makes payments via the platform. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Book a new ride | Ride |
| View ride status | Ride |
| Make payment | Payment |
| View payment receipt | Payment |
| Update personal profile | Account |
| View ride history | Ride |

### Driver

| **Description** | A service provider who accepts ride requests, updates trip statuses, and navigates to pickup/drop-off. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Accept a ride assignment | Ride |
| Update ride status | Ride |
| Send GPS location updates | Location |
| View payment history | Payment |
| Update availability status | Account |
| Manage vehicle details | Vehicle |

### Ride

| **Description** | Represents a single trip from pickup to drop-off, managing state transitions and linking customer, driver, locations, and payment. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Create ride request | Customer |
| Assign driver to ride | Driver, RideManager |
| Track ride progress (status & ETA) | Location |
| Update ride status (accepted, completed) | Driver |
| Link to payment for fare collection | Payment |

### Payment

| **Description** | Handles fare calculation, transaction processing, and receipt generation through a third-party gateway. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Calculate ride fare | Ride |
| Process payment transaction | PaymentGateway |
| Generate digital receipt | Customer, Ride |
| Update payment status on ride | Ride |

### Manager

| **Description** | Administrative user who accesses reports, monitors operations, and manages accounts. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Generate operational reports | ReportGenerator, Report |
| View ride and payment analytics | Ride, Payment |
| Manage user accounts | Account |
| Monitor system performance metrics | Report |

### Account

| **Description** | Manages authentication credentials and profile data for all user types (Customer, Driver, Manager). |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Register new user account | Customer, Driver, Manager |
| Authenticate user login | Customer, Driver, Manager |
| Update credentials and profile data | Customer, Driver |
| Enforce role-based access control | Manager |

### Vehicle

| **Description** | Data-holder for a driver’s transport details. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Store vehicle information | — |
| Provide vehicle type | Driver |
| Validate registration status | — |

### Location

| **Description** | Data-holder for geographic coordinates and address details used in ride booking and tracking. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Hold pickup/drop-off coordinates | Ride |
| Provide address lookup | None or external GIS service |
| Supply data for ETA calculations | RideManager |

### Report

| **Description** | Data-holder for aggregated analytics and summaries used by managers. |
| --- | --- |
| **Responsibility** | **Collaborator** |
| Store ride volume and revenue data | ReportGenerator |
| Hold driver performance metrics | ReportGenerator |
| Format summary for display/export | Manager |

# Design Quality

## Design Heuristics

| **ID** | **Design Heuristic** | **Description** | **Application in SmartRide** |
| --- | --- | --- | --- |
| H1 | High Cohesion | Ensure each class has a single, well‐defined responsibility. | The Ride class handles trip lifecycle (status, ETA), Payment handles fare calculation and receipts, and Account manages authentication—each class focuses on one domain concept. |
| H2 | Low Coupling | Minimize dependencies between classes to reduce ripple effects of changes. | Customer, Driver, and Manager interact with Account via a simple association; Payment relies on PaymentGateway interface, avoiding direct dependency on gateway internals. |
| H3 | Encapsulation | Hide internal state; expose behavior only through public methods. | Attributes like rideStatus and paymentStatus are private; updates occur via methods (updateStatus(), markPaid()), preventing external objects from corrupting internal state. |
| H4 | Single Responsibility | A class should have one reason to change, handling only its own concern. | RideManager coordinates assignments and status changes, while ReportGenerator is solely responsible for analytics—separating dispatch logic from reporting logic. |
| H5 | Information Expert | Assign responsibility to the class with the necessary information. | The Location class calculates distance and ETA because it holds geocoordinates; Ride delegates ETA calculation to Location, leveraging the data holder’s knowledge of positions. |
| H6 | DRY (Don’t Repeat) | Avoid duplicating logic; centralize shared functionality. | Authentication logic is centralized in AccountManager rather than duplicated in Customer or Driver; likewise, notification delivery is handled by a single NotificationService. |
| H7 | Open/Closed Principle | Classes should be open for extension but closed for modification. | New payment methods can be added by implementing the PaymentStrategy interface without altering existing PaymentProcessor code. |

## Design Patterns

### Creation Patterns

#### Factory Pattern

SmartRide must support multiple types of rides (e.g., ImmediateRide vs. ScheduledRide) and multiple payment strategies (e.g., CreditCardPayment, WalletPayment, PayPalPayment).

**Intent:**  
Encapsulate the instantiation logic for these variants behind a common interface so clients simply request “a ride” or “a payment” without knowing the concrete class details.

**Structure:**

* RideFactory.createRide(type, customer, pickup, dropoff) returns either an ImmediateRide or ScheduledRide instance.
* PaymentFactory.createPayment(method, ride, amount) returns the appropriate Payment subclass (CreditCardPayment, WalletPayment, etc.).

A screenshot of a computer

AI-generated content may be incorrect.

**Benefits:**

* **Flexibility:** New ride types (e.g., pooled rides) or payment methods can be added without modifying client code.
* **Maintainability:** Centralizes creation logic, making it easier to update instantiation rules (e.g., fare adjustments, validation) in one place.
* **Extensibility:** Supports future features—like discounted or loyalty-based payments—by plugging in new factory products.

#### Singleton Pattern

Core coordination services such as **RideManager** and **PaymentProcessor** must have exactly one global instance to maintain consistent state and coordinate across all users and rides.

**Intent:**  
Ensure a single, shared instance of these service classes, preventing conflicting state and reducing resource overhead.

A computer screen shot of a program

AI-generated content may be incorrect.

**Benefits:**

* **Global Access Point:** All parts of the system—controllers, UI layers, background jobs—use the same RideManager instance.
* **Consistent State:** Prevents divergent dispatch queues or duplicate driver assignments by centralizing coordination.
* **Resource Efficiency:** Avoids unnecessary instantiation of heavyweight service classes, conserving memory and threads.

### Structural Patterns

#### MVC (Model - View - Controller) Pattern

The SmartRide platform exposes both a web portal (for managers and administrative staff) and mobile/web interfaces (for customers and drivers). To organize the presentation, interaction, and data layers cleanly, we adopt the MVC pattern across our web and mobile applications.

**Model**

* **Definition:** Encapsulates domain data and business logic.
* **Examples in SmartRide:**
  + Entity Framework Core classes (Customer, Ride, Payment, etc.)
  + Service-layer components (RideManager, PaymentProcessor) that enforce business rules (e.g., calculating fares, assigning drivers).

**View**

* **Definition:** Renders the user interface and presents data to end users.
* **Examples in SmartRide:**
  + **Web Portal:** React components (e.g., dashboard pages, ride-booking forms) or Razor views in ASP.NET Core MVC.
  + **Mobile App:** React Native screens for booking rides, tracking, and payments.

**Controller**

* **Definition:** Handles user input, orchestrates interactions between Model and View, and returns responses.
* **Examples in SmartRide:**
  + **API Controllers** in ASP.NET Core (RidesController, PaymentsController) that receive HTTP requests, invoke model services, and return JSON.
  + **React Router** configuration (web) or navigation logic (mobile) that maps URLs or UI actions to controller endpoints.

### Behavioral Patterns

#### Observer Pattern

Customers and drivers need real-time notifications whenever key events occur (ride assignment, status changes, payment completion).

**Usage:**

* The **Ride** object acts as the **Subject**.
* **CustomerApp** and **DriverApp** instances subscribe as **Observers** to ride events.
* When the ride status updates (e.g., “Assigned,” “Arrived,” “Completed”), the Ride object calls its notifyObservers() method, triggering each subscriber’s update callback.

**Benefits:**

* **Loose Coupling:** Ride logic does not need to know details of UI layers; it simply broadcasts events.
* **Scalability:** New observers (e.g., analytics service, SMS gateway) can subscribe without changing Ride code.
* **Responsiveness:** Customers and drivers receive instantaneous updates, improving trust and UX.

#### Queue Pattern

During peak demand, hundreds or thousands of ride requests arrive concurrently. We must ensure no request is lost and that assignments are processed in a fair, efficient order.

**Usage:**

* Incoming ride requests are enqueued in a **DispatchQueue**.
* A background **Dispatcher** service dequeues requests one at a time and invokes RideManager.assignDriver().
* Priority rules (e.g., scheduled rides before immediate, premium customers) can be implemented by inserting requests at different positions in the queue.

**Benefits:**

* **Reliability:** Guarantees every ride request is handled, even under heavy load.
* **Flexibility:** Priority logic can be adjusted (e.g., VIP customers or high-fare requests first) without disrupting the enqueue/dequeue mechanism.
* **Back-pressure Handling:** If downstream services (e.g., driver notification) are slow, the queue buffers requests rather than dropping them, allowing the system to smooth out spikes.

# Bootstrap Process

## Key Components

* **Main**: The entry point of the SmartRide ORSP responsible for initializing core services and starting the application.
* **Customer**: Represents end users who register, book rides, and make payments.
* **Driver**: Represents service providers who register, accept rides, and update trip status.
* **Account**: Manages authentication and profile information for Customers, Drivers, and Managers.
* **Ride**: Encapsulates a single trip’s details, state transitions, and links to Customer, Driver, and Location.
* **Location**: Holds geographic data (pickup and drop-off coordinates and addresses) used for booking and tracking.
* **Vehicle**: Data-holder for Driver’s transport details (type, license plate).
* **RideManager**: Coordinates ride request processing, driver assignment, and status updates.
* **PaymentProcessor**: Orchestrates fare calculation, transaction submission to a PaymentGateway, and receipt generation.
* **Payment**: Represents a fare transaction, its status, and associated receipt details.
* **ReportGenerator**: Aggregates ride and payment data to produce management reports.
* **Report**: Data-holder for summarized metrics (ride volumes, revenue, driver performance).
* **NotificationService**: Sends real-time alerts to Customers and Drivers (e.g., status changes, payment confirmations).
* **NavigationService**: Interfaces with external GPS/navigation APIs to compute routes and ETAs.
* **PaymentGateway**: Adapter for a third-party payment service used by the PaymentProcessor.

## System Initialization (Main)

The Main system will first be initialized to handle the connection and

communication with the database, which is essential in the initialization of other

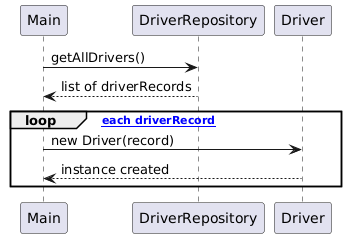
components of the RIS.

## Customer Initialization

A screenshot of a computer

AI-generated content may be incorrect.

## Driver Initialization



## Account Initialization

A screenshot of a computer

AI-generated content may be incorrect.

## Ride Initialization

A diagram of a ride record

AI-generated content may be incorrect.

## Location Initialization

A diagram of a service

AI-generated content may be incorrect.

## Vehicle Initialization

A screenshot of a computer

AI-generated content may be incorrect.

## Payment Initialization

A screenshot of a computer

AI-generated content may be incorrect.

## Report Initialization

A diagram of a payment generator

AI-generated content may be incorrect.

# Verification

The proposed design has been through rigorous verification processes with the

below user tasks.

## Customer Registers and Logs In

A screenshot of a computer

AI-generated content may be incorrect.

A new customer signs up (registration), receives a confirmation, then logs in to obtain a session token.

## Customer Books a Ride

A diagram with text and words

AI-generated content may be incorrect.

An authenticated customer specifies pickup & drop-off; the RideManager validates and assigns a driver.

## Customer Tracks Ride

A diagram of a service

AI-generated content may be incorrect.

During an active ride, the driver’s app pushes GPS updates to the server, which notifies the customer.

## Customer Processes Payment

A screenshot of a computer

AI-generated content may be incorrect.

After ride completion, the customer initiates payment; the PaymentProcessor interacts with the gateway and updates the ride.

## Manager Requests Report

A screenshot of a computer

AI-generated content may be incorrect.

A manager requests an operational report; the ReportGenerator aggregates data and returns a summary.

# PART 3: Object Design Implementation & Reflection

# Detailed Design

## Customer Class

A screen shot of a computer screen

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Implementation Notes:**

* Uses **Factory Pattern** (RideFactory) in BookRide.
* Validates location bounds via injected ILocationService.
* Raises domain events (RideRequestedEvent) that RideManager observes.

## Diver Class

A computer screen shot of blue text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Implementation Notes**

* **Observer Pattern**: subscribes to RideAssignedEvent to trigger UI notification.
* Location updates are throttled and sent to GPSService via a rate-limited queue.

## Ride Class

A screen shot of a computer program

AI-generated content may be incorrect.

A black screen with blue text

AI-generated content may be incorrect.

**Implementation Notes**

* **State transitions** validated internally (e.g., cannot complete before start).
* ETA calculation delegated to Location.CalculateDistance() and average speed constants.

## Payment Class

A screen shot of a computer code

AI-generated content may be incorrect.



**Implementation Notes**

* Implements **Strategy Pattern**: different IPaymentGateway strategies plugged in at runtime.
* Retries up to three times on transient failures, with exponential back-off.

## Account Class

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Implementation Notes**

* Passwords hashed with PBKDF2.
* JWT tokens include role-based claims for Customer, Driver, or Manager.

## Data Holder Classes

* **Vehicle**: simple POCO with Model, LicensePlate, Type.
* **Location**: holds Latitude, Longitude, Address; provides a CalculateDistanceTo(Location other) helper.
* **Report**: aggregates counts and sums (TotalRides, TotalRevenue, AverageDriverRating) for dashboards.

## Controller Classes

* **RideManager** (Singleton): Coordinates AssignDriver, retries on decline, and publishes ride-related events.
* **PaymentProcessor** (Singleton): Wraps Payment creation, gateway invocation, and receipt issuance.
* **ReportGenerator:** Queries repositories, aggregates data, and formats Report objects for UI or export.
* **Asp.NET Core Controllers** (RidesController, PaymentsController, AccountsController): Map HTTP routes to service calls, handle input validation (via data annotations), and return standardized responses.

# Discussion of Quality of Part 2 Design

The design strikes a strong balance between **modularity** and **pragmatic simplicity**. By leveraging design heuristics (high cohesion, low coupling, DRY) and patterns (Factory, Singleton, Observer, Queue, Strategy, MVC), we have:

* **Reduced complexity** in each class’s responsibilities.
* **Enabled future growth** without major rewrites.
* **Provided clear extension points** for new functionality (e.g., shared rides, dynamic pricing).
* **Ensured the core workflows** (registration, booking, tracking, payment, reporting) are well-supported and resilient under load.

This solid foundation sets up SmartRide for both immediate success and smooth evolution as business needs expand.

# Lessons Learnt

* **Prioritize Core Domain Understanding**  
  Early investment in clarifying domain concepts (e.g., differentiating Ride from RideRequest, or understanding the real-time nature of GPS updates) paid dividends in avoiding misaligned designs and rework.
* **Balance Granularity**  
  While fine-grained classes improve single responsibility, too many micro-objects (e.g., separate DTOs, adapters) can complicate code navigation. Striking the right balance between clarity and over-engineering is key.
* **Leverage Design Patterns Judiciously**  
  Applying patterns like Factory and Observer provided clear extension points and decoupling. However, overuse can add unnecessary indirection—patterns should solve concrete problems, not be used for their own sake.
* **Design for Testability from Day One**  
  Defining clear interfaces (e.g., IPaymentGateway, IGPSService) and keeping domain classes free of infrastructure dependencies enabled thorough unit testing and simplified mocking of external services.
* **Model Real-World Workflows Accurately**  
  Creating detailed workflow and sequence diagrams uncovered edge cases (e.g., ride cancellations, payment retries) that initial requirements had not fully addressed, leading to more robust error-handling plans.
* **Document Assumptions and Simplifications**  
  Explicitly stating what’s in and out of scope (e.g., no shared rides, fixed vehicle types) prevented feature creep and kept the initial implementation focused on delivering high-value functionality quickly.
* **Plan for Scalability Upfront**  
  Although the initial load requirements were modest, incorporating queue-based dispatch and rate-limiting early ensured the system could handle traffic spikes without retrofitting major architectural changes later.

# Implementation - Source Code

(To be added)

# Implementation- Compilation and execution

(to be added)

# Meeting the requirements of the assignment specification

| **Requirement** | **Location in Document** |
| --- | --- |
| **Part 1: Requirements Specification** |  |
| – Domain-level requirements (tasks users want to perform) | “Functional Requirements at Goal Level” (p. X) and “List of Tasks” (p. Y) |
| – Non-functional requirements (quality attributes) | “Non-Functional Requirements” (p. Z) |
| – Context of the new system | “Project Overview,” “Scope,” and “Context of the New System” sections |
| – Domain/data requirements (simple domain model) | “Domain Entities List” and “Domain Model in PlantUML” |
| – Four non-functional quality attributes specified | “Quality Attributes (Non-Functional Requirements)” table |
| – Workflow of events from reservation to payment | “Typical Workflow,” plus detailed **Swimlane Workflow Diagrams** for Registration, Booking, and Payment sections |
| **Part 2: Object Design** |  |
| – List of classes & UML class diagram (no methods/attributes) | “List of Classes” and “UML Class Diagram in PlantUML” |
| – CRC cards for each class | “CRC Cards” tables |
| – Design patterns and heuristics applied | “Design Patterns and Design Heuristics” sections |
| – Bootstrap/initialization process diagrams | “Component Initialization” and “Bootstrap Process” sequence diagrams |
| – Four interaction scenarios and verification diagrams | “Typical Interaction Scenarios” and **Sequence Diagrams** for booking, tracking, payment, and reporting |
| **Part 3: Object Design Implementation & Reflection** |  |
| – Detailed design (attributes, methods, patterns) | “Detailed Design” section with class skeletons and pattern usage |
| – Source code for core classes | “Implementation – Source Code” text document |
| – Reflection on design quality (cohesion, coupling, testability, etc.) | “Discussion of Quality of Part 2 Design” and “Lessons Learned” |
| – Verification that requirements are met through sequence diagrams | “Verification” section with sequence diagrams for all key interactions |

# Conclusion

This report has presented a comprehensive, end-to-end analysis and design for SmartRide’s Online Ride-Sharing Platform (ORSP). In **Part 1**, we captured the business drivers—reducing wait times, automating dispatch and payments, and enabling data-driven management—and translated them into clear functional and non-functional requirements. We documented the system context, domain vocabulary, major tasks, workflows, and quality attributes needed to guide development.

In **Part 2**, we used Responsibility-Driven Design to define the core classes—Customer, Driver, Ride, Payment, Manager, and Account—along with data-holder classes for Vehicle, Location, and Report. We expressed their relationships in UML class and domain model diagrams, articulated responsibilities through CRC cards, and applied key design patterns (Factory, Singleton, Observer, Queue, Strategy, and MVC) and heuristics (high cohesion, low coupling, SRP, DRY). A detailed bootstrap process and verification scenarios demonstrated that the design supports all critical user interactions.

**Part 3** bridged design and implementation. We detailed class attributes and methods, showed how patterns are realized in C# skeleton code, and reflected on design quality—highlighting testability, maintainability, and scalability. Sequence diagrams validated requirement coverage, while lessons learned underscore best practices for evolving the platform.

Together, these deliverables give SESoft Consulting a solid blueprint for cost estimation, resource planning, and incremental development. The ORSP design not only meets SmartRide’s immediate needs but also provides a flexible foundation for adding advanced features—such as shared rides, surge pricing, loyalty programs, and multi-city rollouts—in future phases. With this clear, validated architecture and implementation sketch, the project is well positioned to move into detailed development and deployment.