MAPÚA UNIVERSITY
SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Tentative Title:
Accessible Transportation Scheduler

CPE106L (Software Design Laboratory)

Member 1: JM Ajegram A. Esperanza
Member 2: Ralph Jed N. Lam
Member 3: Denise Kate L. Pagala

Group No.: **8**
Section: **E01**

Date of Submission: **October 13, 2025**

Analysis

1.0 Introduction

1.1 Project Overview

The project aims to develop an Accessible Transportation Scheduler that assists elderly individuals or persons with accessibility needs in scheduling transportation with local volunteers or ride services. The system automates ride matching, route optimization, and scheduling to ensure efficient and inclusive mobility support.

The system's core functionalities include route optimization using graph-based algorithms (e.g. Dijkstra's Algorithm for shortest paths), ride scheduling, and real-time notifications. The application is built upon a Model-View-Controller (MVC) architecture, with the backend implemented using FastAPI and the user interface developed with Flet for desktop deployment. MongoDB serves as the main data store for flexible management of user, driver, and ride data, while Google Maps API is used for distance and routing integration.

1.2 Project Scope

- Scheduling and Optimization Algorithms: Implement route optimization algorithms like Dijkstra's Algorithm to determine the shortest and most efficient routes. Schedule rides based on availability, proximity, and priority of users and accessibility needs.
- Object-Oriented System Design: Define classes for User, Driver, RideRequest, Schedule, and Route.
- MVC Design Pattern: Employ the Model-View-Controller pattern to separate presentation, logic, and data handling.
- Data Management and Visualization: Use [SQL] for dynamic data storage and Matplotlib for ride and service analytics.
- API Integrations: Integrate Google Maps API for routing and FastAPI for backend endpoints and notifications.
- 3-Tier MVC Architecture Implementation:
    - Presentation Layer: Flet desktop client for booking and managing rides.
    - Business Logic Layer: FastAPI backend for scheduling, route optimization, and notifications.
    - Data Access Layer: A relational SQL database to store user, driver, and ride information using well-defined relationships.

2.0 System Architecture

The system is designed with a 3-tier MVC architecture:

1. **Presentation Layer (Client Application):**
    - **Flet Desktop App:** Provides an accessible interface for users and drivers.
    - **User Functions:** Request rides, view schedules, and receive notifications.
    - **Driver Functions:** Accept rides, view optimized routes, and update trip status.
2. **Business Logic Layer (Backend):**
    - **FastAPI Server:** Hosts endpoints for ride scheduling, route optimization, and data synchronization.
    - **Algorithm Module:** Implements Dijkstra algorithm for route and schedule optimization.
    - **Service Modules:** Sends updates to users and drivers in real-time.
    - **Analytics Module:** Generate statistics and visualizations using Matplotlib.
3. **Data Access Layer (Database):**
    - **SQL Database:** Stores relational data for users, drivers, rides, and schedules.
    - **DAO (Data Access Object) or ORM (Object-Relational Mapping):** Handles CRUD operations for the user, driver, ride, and log data.

3.0 Requirements

3.1 Functional Requirements

3.1.1 Ride Scheduling and Optimization Use Case

- **FR-S1:** The system shall allow users to request rides by specifying pickup and drop-off locations.
- **FR-S2:** The system shall retrieve route and distance data from the Google Maps API.
- **FR-S3:** The system shall apply route optimization algorithms to determine the shortest available path.
- **FR-S4:** The system shall assign available drivers based on proximity, route efficiency, and user priority (e.g., elderly or accessibility needs).
- **FR-S5:** The system shall generate and update ride schedules in real-time.

3.1.2 Ride Management and Notification Use Case

- **FR-M1:** The system shall allow users to view, cancel, or modify active ride requests.
- **FR-M2:** The system shall notify drivers of new ride assignments through the client app interface.
- **FR-M3:** The system shall display optimized routes and estimated travel time to the driver.
- **FR-M4:** The system shall allow drivers to update ride status (e.g., “En Route,” “Completed”).
- **FR-M5:** The system shall send notifications to users regarding ride status changes.

3.1.3 Data Access Use Case

- **FR-D1:** The system shall store user, driver, and ride information in a relational SQL database.
- **FR-D2:** The Data Access Layer shall handle all CRUD operations through ORM.
- **FR-D3:** The system shall maintain logs for ride history, driver availability, and performance statistics.
- **FR-D4:** The system shall allow administrators to retrieve reports and visualizations of ride activity.

3.2 Non-Functional Requirements

- **NFR-1: Performance:** The system should process ride scheduling and route optimization requests within [time] seconds to ensure responsive operation.
- **NFR-2: Scalability:** The architecture should support future expansion, such as integrating more service areas, additional drivers, or new data sources.
- **NFR-3: Reliability:** The system should handle concurrent requests from multiple users without service interruption.
- **NFR-4: Maintainability:** Codebase should follow modular design and adhere to the MVC pattern to simplify updates and debugging.
- **NFR-5: Usability:** The desktop client interface (Flet) should be intuitive and accessible for elderly or mobility-impaired users.
- **NFR-6: Security:** Communication between the client app and server should be encrypted using HTTPS, and user credentials must be securely stored.
- **NFR-7: Data Integrity:** The database shall enforce referential integrity using primary and foreign key constraints.

3.3 Hardware & Software Constraints

- **Hardware:** Desktop or laptop computer hosting the FastAPI server and SQL database.
- **Software:** Python as main programming language, FastAPI for backend web framework, Flet for desktop GUI framework, SQL Database, [SQL ORM] for database access and mapping, Google Maps API for route and distance data, and Matplotlib for visual analytics and reporting.

Use Case Diagram

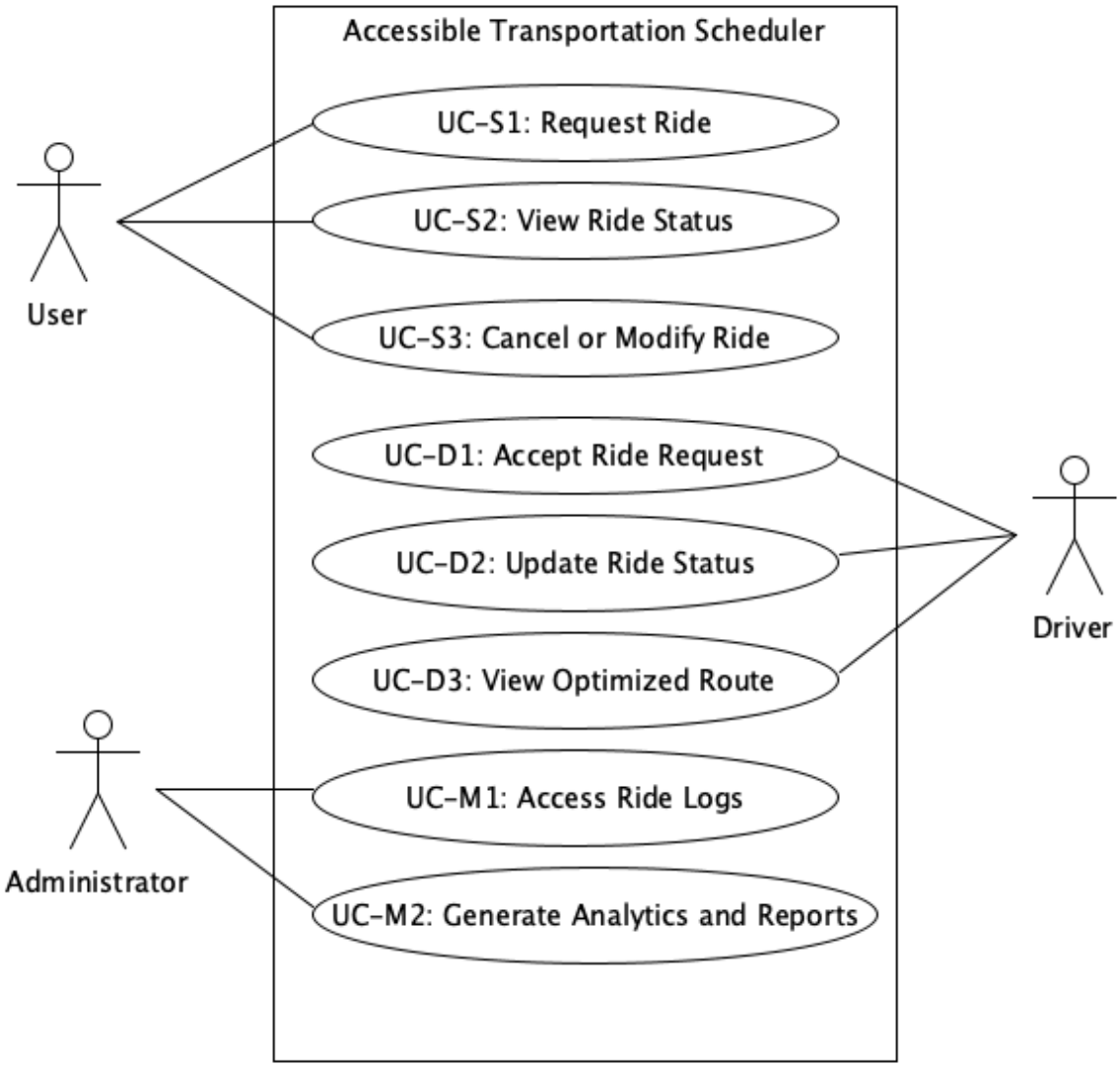

Figure 1. Use Case Diagram for Accessible Transportation Scheduler

Figure 1 shows the Use Case Diagram for the Accessible Transportation Scheduler system, which visually represents the interactions between the system's primary actors, User, Driver, and System Administrator, and the core functionalities of the system. The diagram outlines how each actor interacts with the system to perform various tasks.

The User, who represents elderly individuals or people with accessibility needs, can initiate a ride request by providing pickup and drop-off locations, view the current status of their ride, and cancel or modify the ride if necessary. These functionalities make sure

that the User has control and flexibility over their transportation experience. The Driver, who provides the transportation service, is responsible for accepting ride requests, updating the status of the ride, such as "En Route" or "Completed," and viewing the optimized route with estimated travel times to ensure timely and efficient service. The System Administrator, the third actor in the system, is tasked with accessing ride logs, viewing performance reports, and generating analytics and reports to track the system's overall performance and optimize its operations.

The diagram maps out these interactions, with each use case representing a distinct functionality within the system. The system boundary is clearly defined to enclose all the use cases, showing the main operations of the Accessible Transportation Scheduler. This boundary also helps to differentiate between what is part of the system and what lies outside of it.

Furthermore, the diagram highlights the real-time nature of the system, with features like ride status updates and the ability for Users to modify or cancel rides, ensuring a responsive experience for all users. The Administrator's ability to monitor and generate reports emphasizes the importance of system oversight and optimization. Overall, the diagram provides a high-level overview of the system's operation, offering a clear representation of the roles, responsibilities, and actions of each actor. This visual representation serves as a foundational tool for understanding the system's design and planning further enhancements or feature additions as the project evolves.

Class Diagram

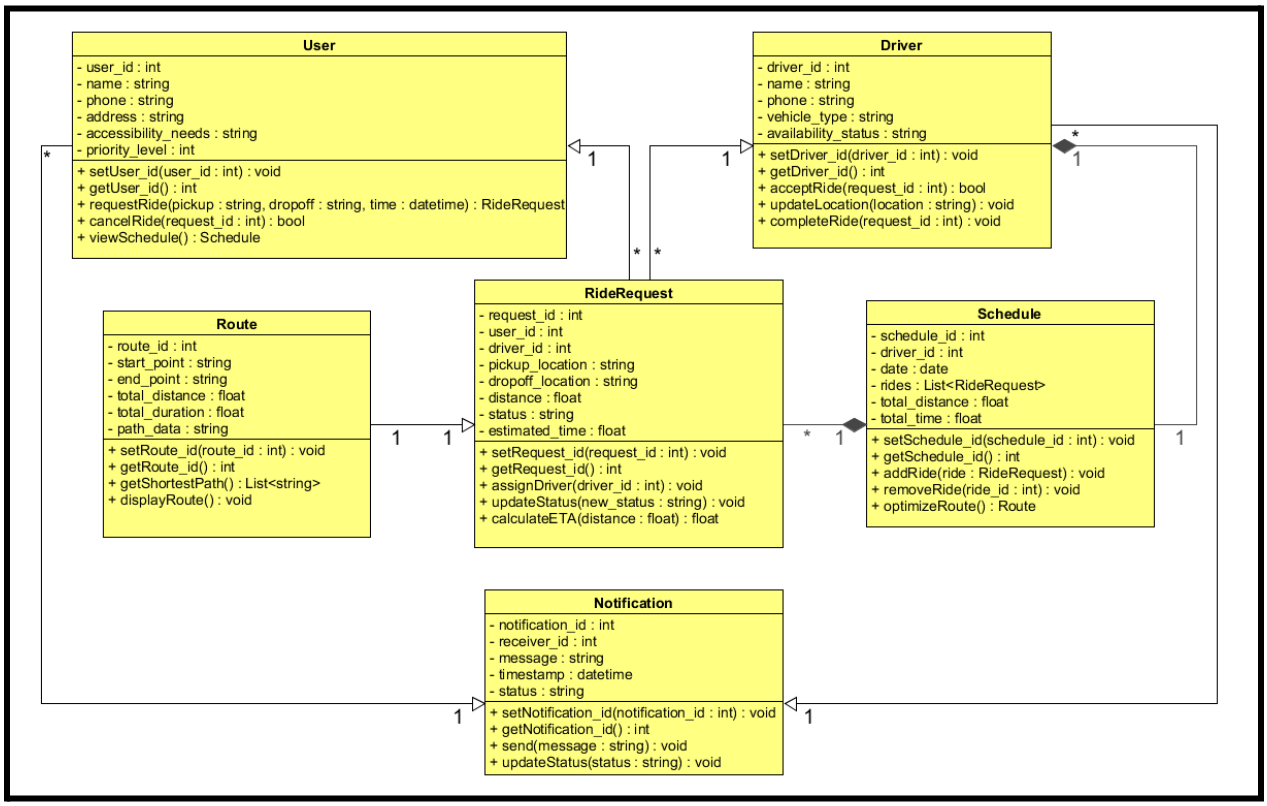

Figure 2. Class Diagram for Accessible Transportation Scheduler

Figure 2 shows the class diagram for the *Accessible Transportation Scheduler,* which illustrates the system's main components and their relationships. This design follows an object-oriented approach, ensuring modularity and alignment with the Python FastAPI backend structure. Furthermore, the class diagram includes five classes namely: User, Driver, RideRequest, Route, Schedule, and Notification.

The User class represents the passengers who request transportation, wherein it includes the following information, such as user_id, name, phone, address, accessibility_needs, and priority_level. The users can create ride requests, cancel bookings, and view their scheduled trips. On the other hand, the Driver class shows service drivers, which contains details like driver_id, name, phone, vehicle_type, and availability_status. The drivers can accept rides, update their location, and complete assigned trips. Both User and Driver classes include standard setter and getter methods for data access and encapsulation.

The RideRequest class serves as the central connection between users and drivers, which contains identifiers for both, along with ride details like pickup_location, dropoff_location, distance, estimated_time, and status. Each ride request is associated with a Route, which

stores routing data including the start_point, end_point, total_distance, total_duration, and path_data. This separation allows flexibility in computing routes using algorithms like Dijkstra's or A*, or through the Google Maps API integration. Conversely, the schedule class represents the driver's daily itinerary, which includes multiple RideRequest objects through a one-to-many relationship, along with fields like date, total_distance, and total_time. Each driver has one corresponding schedule, represented as an aggregation relationship, while a schedule is composed of multiple ride requests, indicated by a composition relationship in the diagram.

Lastly, the Notification class is responsible for communication between the system and its users or drivers. It includes attributes like notification_id, receiver_id, message, timestamp, and status. Its methods allow sending notifications and updating their status, ensuring that users are informed about the ride details. The relationships among these classes are defined through multiplicities to represent real-world interactions, which can be seen at the end of each arrow. A single user can make many ride requests, and a driver can fulfill multiple requests, which explains for the (1 -> *) or one-to-many. Each ride request has exactly one route, resulting in 1 -> 1, and a driver also has one schedule only, giving 1 -> 1. Furthermore, both drivers and users can receive multiple notifications, resulting to 1 -> *.

Overall, this class diagram demonstrates a structured and scalable design, with each class having a unique purpose. This organization supports the system's main object, which is to provide a reliable and accessible transportation scheduling service for elderly and differently-abled users.