Lecture Notes:

1. “Intro To System Design”

0. Preliminary

System Design and AI Relevance

System Design is an essential skill for developers and engineers, enabling the creation of scalable, efficient, and resilient systems. With Artificial Intelligence (AI) becoming a cornerstone of modern applications, mastering system design is crucial for AI engineers to effectively deploy and scale their models, particularly in Pakistan’s fast-evolving tech landscape.

What is System Design?

System Design involves defining the architecture, components, modules, interfaces, and data flows to meet specific requirements. It focuses on scalability, data management, and seamless interaction between components like APIs, databases, and caching systems.

How System Design Integrates with AI

System Design enables AI engineers to deploy, scale, and integrate machine learning models into real-world applications, such as e-commerce platforms, healthcare systems, or smart city solutions, ensuring performance and reliability.

Real-Life AI + System Design Examples:

- E-Commerce Recommendation System: ML models provide personalized product suggestions, with APIs and Redis caching for real-time delivery.
- Traffic Management System: AI predicts traffic patterns in cities like Karachi or Lahore, with backend systems handling real-time data from IoT devices.
- Healthcare Appointment Scheduler: ML optimizes hospital schedules, integrated with MongoDB for patient data and FastAPI for bookings.
- Smart Agriculture Advisor: AI analyzes soil and weather data for farmers, with a cloud-based system delivering insights via mobile apps.

Common Tools Used

- Model Serving: TensorFlow Serving, TorchServe, ONNX, Triton
- APIs: FastAPI, Flask, Node.js
- Storage: MongoDB, PostgreSQL, S3
- Messaging/Streaming: Kafka, RabbitMQ
- Deployment: Docker, Kubernetes, CI/CD tools

1. Foundations:

Low-Level Design (LLD) and Object-Oriented Programming (OOP) form the foundation for creating software that is scalable, maintainable, and efficient. LLD focuses on detailing a system's internal structure to address real-world challenges, such as building ride-sharing platforms like Careem or food delivery apps like Cheetay.

OOP models real-world entities through classes and objects, enabling secure, reusable, and future-ready code. Visual tools like UML diagrams, such as Class Diagrams for relationships or Sequence Diagrams for workflows, help bring clarity to these designs.

The SOLID principles, particularly the Liskov Substitution Principle (LSP), provide a framework for writing clean and adaptable code, making it easier to design systems that excel in real-world applications.

2. OOP and Real-World Example

OOP simplifies software design. For instance, consider a system for managing a fleet of Suzuki and Honda cars for a ride-sharing app like Careem.

Key Points:

- Real-world entities like Car, Driver, and Ride can be modeled as classes.
- Using UML diagrams, we communicate software structure and behavior visually.

3. OOP Pillars

3.1 Abstraction

Abstraction hides implementation complexity. For example, starting a car engine requires just calling startEngine().

```
class Car {
public:
    virtual void startEngine() = 0;
    virtual void accelerate() = 0;
};
```

3.2 Encapsulation

Encapsulation protects data by restricting direct access using access modifiers.

```
class CareemCar {
private:
    string model;
    int speed;
public:
```

```
    void accelerate() { speed += 10; }
    int getSpeed() { return speed; }
};
```

3.3 Inheritance

Inheritance enables reuse. Example: ElectricCar inherits from Car.

```
class ElectricCar : public Car {
public:
    void chargeBattery() { /*...*/ }
};
```

3.4 Polymorphism

Polymorphism allows method overriding. Example: Ali's HybridCar overrides startEngine().

```
class HybridCar : public Car {
public:
    void startEngine() override { cout << "Hybrid engine started"; }
};
```

4. Low-Level Design (LLD)

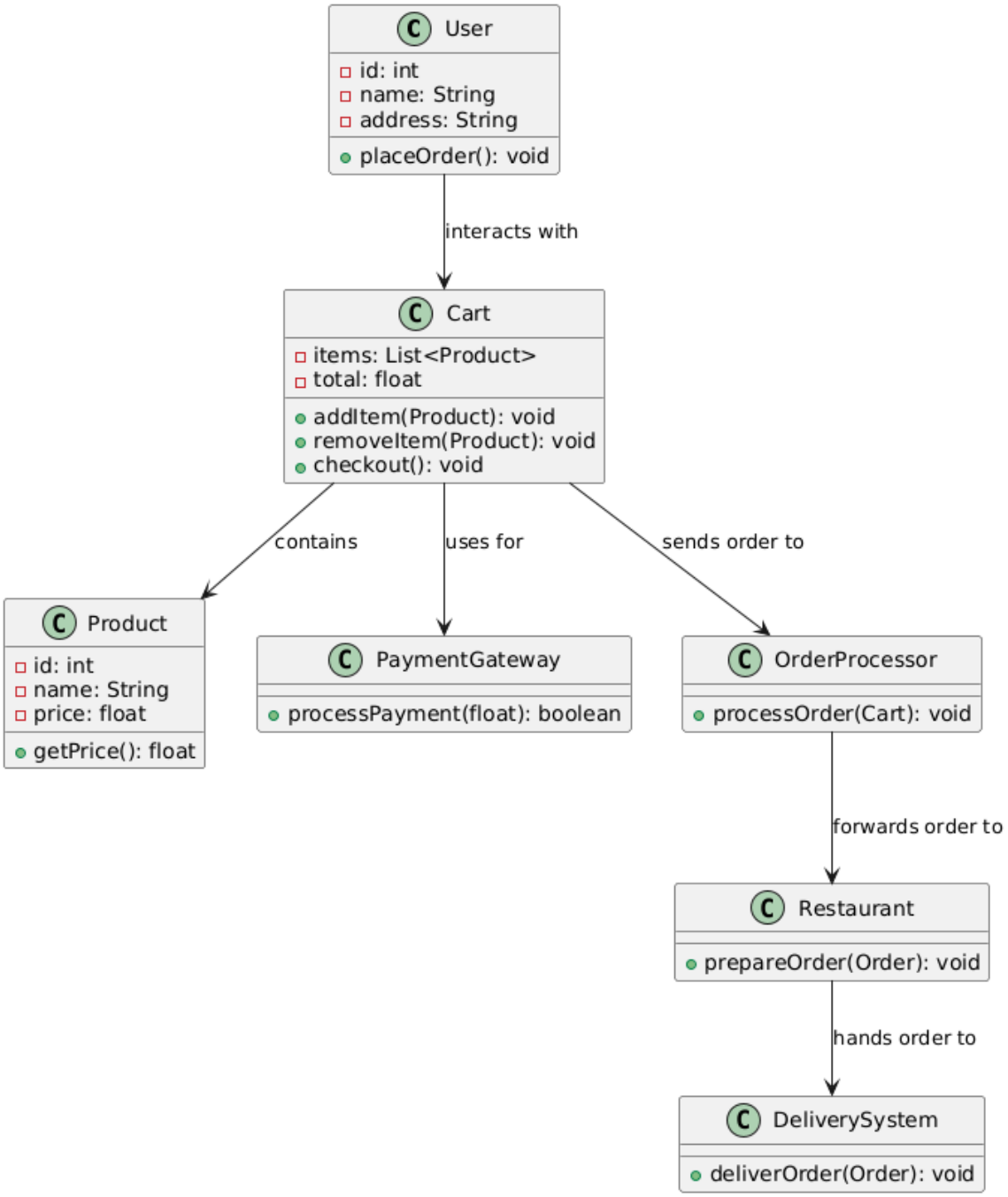
Low-Level Design (LLD) delves into the specifics of system architecture by detailing class-level structures and the underlying logic of software applications. In the platforms like Daraz or Rozee.pk, LLD helps define how individual components interact to deliver a seamless user experience. It focuses on creating clear, precise blueprints for development teams, ensuring consistency and scalability in implementation.

5. UML Diagrams

Unified Modeling Language (UML) diagrams are powerful tools for visually representing the structure and behavior of a system. They bridge the gap between conceptual design and actual implementation, making complex systems easier to understand.

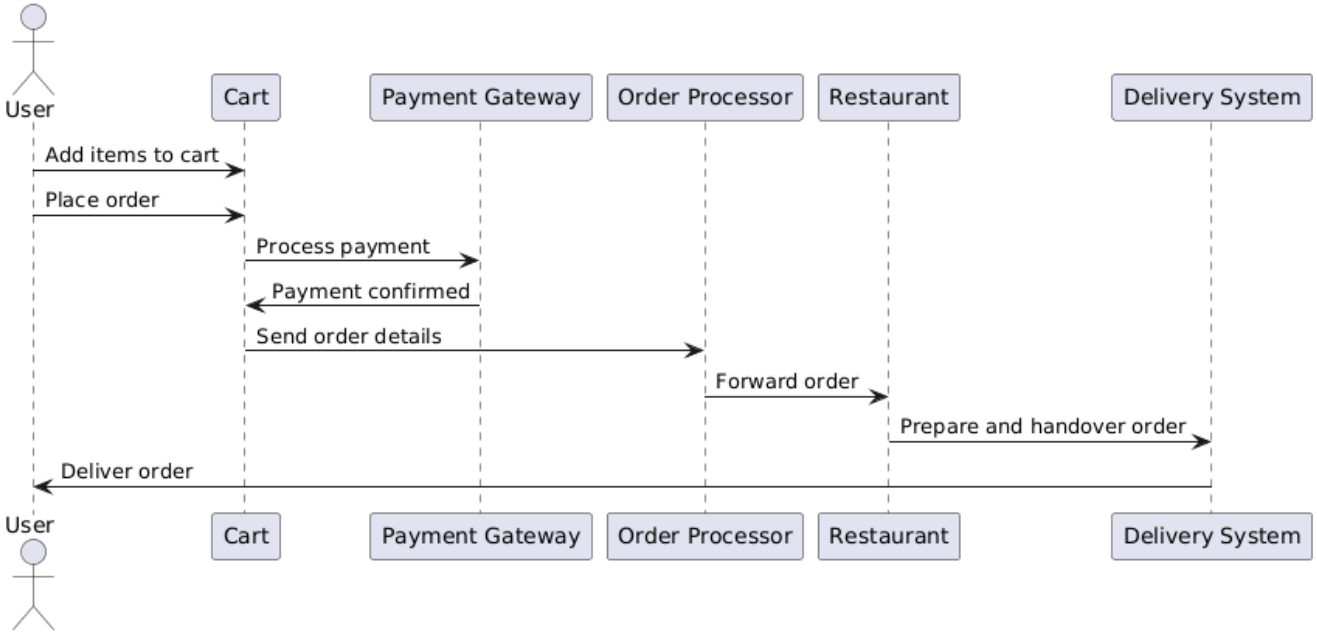
5.1 Class Diagram

Class Diagrams illustrate the static structure of a system. They define classes, their attributes, methods, and the relationships between them (e.g., inheritance, associations, or dependencies). For instance, in a ride-sharing platform, you could depict classes like `Driver`, `Rider`, and `Ride`, along with their attributes (e.g., name, location) and methods (e.g., startRide, endRide).



5.2 Sequence Diagram

Sequence Diagrams capture the dynamic interactions between objects over time, focusing on message exchange. For example, in a food delivery app like Cheetay, a Sequence Diagram could depict the flow of actions when a user places an order—starting with the user selecting items, placing the order, payment processing, and finally, the restaurant and delivery system receiving the order.



6. SOLID Principles

The SOLID principles are a set of five guidelines designed to create software that is scalable, maintainable, and easy to extend. Following these principles ensures that code remains clean, modular, and adaptable to future requirements.

1. Single Responsibility Principle (SRP)

- **Definition**: Each class should have only one reason to change, meaning it should only do one job.
- **Example**: In a ride-sharing app, a `Ride` class should handle only ride-related logic, like start and end times, but not payment processing or user notifications.

2. Open/Closed Principle (OCP)

- **Definition**: Classes should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.
- **Example**: If a Car class is implemented, new car types (e.g., `ElectricCar` or `HybridCar`) can be added by extending the base class, without modifying the original `Car` class.

3. Liskov Substitution Principle (LSP)

- **Definition**: Subclasses should be replaceable by their parent classes without breaking the application.
- **Example**: An `ElectricCar` subclass can replace a `Car` superclass in any context, ensuring that it behaves as expected, fulfilling all the contracts of Car.

4. Interface Segregation Principle (ISP)

- **Definition**: Classes should not be forced to implement interfaces they don't use. It's better to have smaller, more specific interfaces than one large general-purpose interface.
- **Example**: In a ride-sharing app, separate interfaces for `Payment` and `Ride` ensure that a class managing rides does not need to implement payment-specific methods it doesn't require.

5. Dependency Inversion Principle (DIP)

- **Definition**: Depend on abstractions, not concrete implementations. This makes your code more flexible and easier to test.
- **Example**: A Car class should depend on an `IEngine` interface instead of a specific Engine class, allowing different types of engines (e.g., `ElectricEngine`, `PetrolEngine`) to be used interchangeably.

Principle	Definition	Example
SRP	One class = one job	`Ride` class handles ride logic only
OCP	Open for extension, closed for modification	Add new car type without editing Car
LSP	Subclass should be replaceable by parent	`ElectricCar` should replace Car
ISP	Don't force classes to implement unused interfaces	Separate Payment and Ride interfaces
DIP	Depend on abstraction	Car depends on `IEngine` , not Engine

6.1 Liskov Substitution Principle (LSP)

Example of Dog replacing Animal properly:

```
class Animal {
public:
    virtual void makeSound() { cout << "Sound" << endl; }
```

```
};

class Dog : public Animal {
public:
    void makeSound() override { cout << "Bark" << endl; }
};
```

7. Conclusion

System Design is a cornerstone of modern software engineering, especially as it applies to Artificial Intelligence (AI) and large-scale applications. By exploring its core concepts and practical applications, we uncovered how it plays a vital role in creating scalable, efficient systems, particularly in the context of Artificial Intelligence (AI) and large-scale solutions.

1. **The Importance of System Design and AI Integration**
   AI technologies, while transformative, require well-architected systems for deployment and scaling. Concepts like scalability, reliability, and modularity form the backbone of these integrations, enabling engineers to address complex challenges like e-commerce personalization, urban traffic management, and smart agriculture solutions.

2. **Foundations of Object-Oriented Programming (OOP) in Design**
   OOP concepts such as abstraction, encapsulation, inheritance, and polymorphism were introduced to highlight their role in modeling real-world scenarios. By representing entities like cars or drivers in a ride-sharing app, we explored how OOP simplifies problem-solving and enhances code maintainability.

3. **Low-Level Design (LLD) Techniques**
   Focusing on the finer details of class interactions, this week covered UML diagrams like class and sequence diagrams, demonstrating their use in visualizing system architecture for applications like Daraz or Rozee.pk.

4. **SOLID Principles for Clean Architecture**
   Adhering to SOLID principles ensures scalability and maintainability. Examples like the Liskov Substitution Principle illustrated how adhering to these guidelines fosters reliable and flexible codebases.

5. **Practical Applications in Pakistan's Ecosystem**
   Real-life examples emphasized the relevance of system design in addressing local challenges, such as improving healthcare appointment systems, optimizing traffic flows, and supporting smart agricultural practices. These case studies served as a bridge between theoretical concepts and practical implementation.

6. **Tools and Technologies**
   A curated list of tools, including TensorFlow Serving for AI model deployment, FastAPI for API development, and Kubernetes for scalable deployments, provided a clear roadmap for engineers to implement efficient system designs.

Project: Low-Level Design of a Ride-Sharing App

Why LLD for a Ride-Sharing App?

In Pakistan, ride-sharing apps like Careem and Bykea are super popular, solving real-world problems like affordable transport in cities like Karachi, Lahore, or Islamabad. Low-Level Design (LLD) helps us create a modular, scalable system for such apps, ensuring components like ride booking, driver management, and trip tracking work seamlessly. For AI engineers, this is relevant for integrating features like AI-driven ride pricing or route optimization (e.g., avoiding traffic in Saddar, Karachi).

What is the Project?

We're designing a **Ride-Sharing App** that allows:

- Users to book a ride (e.g., bike or car).
- Adding vehicles (e.g., Mehran, Honda bike).
- Tracking ride status (e.g., "Driver Assigned", "Ride Completed").
- Saving ride details to a file (scalable for database storage).

The system must be **scalable** to support new vehicle types (e.g., rickshaws) or features like fare estimation using AI.

Tools Used:

- **Language**: C++ (for this example).
- **Design Tool**: UML diagrams (Class, Sequence).
- **Storage**: File-based (scalable to MongoDB/PostgreSQL).
- **Future AI Integration**: TensorFlow for fare prediction, FastAPI for real-time APIs.

1. Approach to LLD

We'll use the **Bottom-Up Approach** for LLD, starting with small components (e.g., vehicles, rides) and building up to the main app. This is ideal for interviews as it ensures modularity and scalability. We'll:

1. Create a bad design (one class doing everything).
2. Identify flaws using SOLID principles.
3. Refactor into a better design with UML diagrams and code.

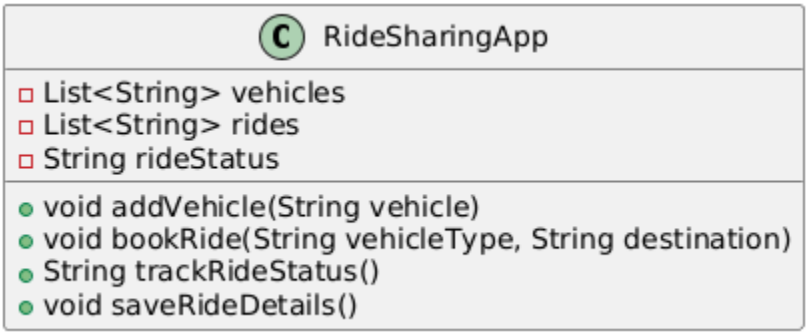
2. Bad Design

Let's start with a bad design where a single class, RideSharingApp, handles everything: storing vehicles, booking rides, tracking status, and saving ride details. This violates SOLID principles like **Single Responsibility Principle (SRP)** and **Open-Closed Principle (OCP)**.

2.1 Features

- Add vehicles (e.g., "Honda Bike", "Suzuki Mehran").
- Book a ride with a vehicle type and destination (e.g., "Gulberg to Mall Road").
- Track ride status (e.g., "Driver Assigned").
- Save ride details to a file.

2.2 UML Diagram (Bad Design)



2.3 Code (Bad Design)

Code can be accessed on my GitHub:
https://github.com/MuhammadTahaNasir/Systems_Design-Codes-Notes/blob/main/Week-1/Bad_Design/Bad_Design.cpp

2.4 Output

```
Ride Status: Driver Assigned
Ride details saved successfully!
```

File Content (ride_details.txt):

```
Honda Bike to Gulberg to Mall Road
```

2.5 Issues with Bad Design

1. **SRP Violation**: `RideSharingApp` handles vehicle management, ride booking, status tracking, and file saving means multiple responsibilities.
2. **OCP Violation**: Adding a new vehicle type (e.g., rickshaw) or status (e.g., "Ride Completed") requires modifying the class.
3. **No Abstraction**: Uses strings for vehicles and rides, with no polymorphism.
4. **Tightly Coupled**: Logic for ride booking and saving is hardcoded, making it hard to extend (e.g., for database storage or AI-based fare calculation).

5. **Scalability Issues**: Can't easily add features like Urdu notifications or JazzCash payments.

3. Improved Design

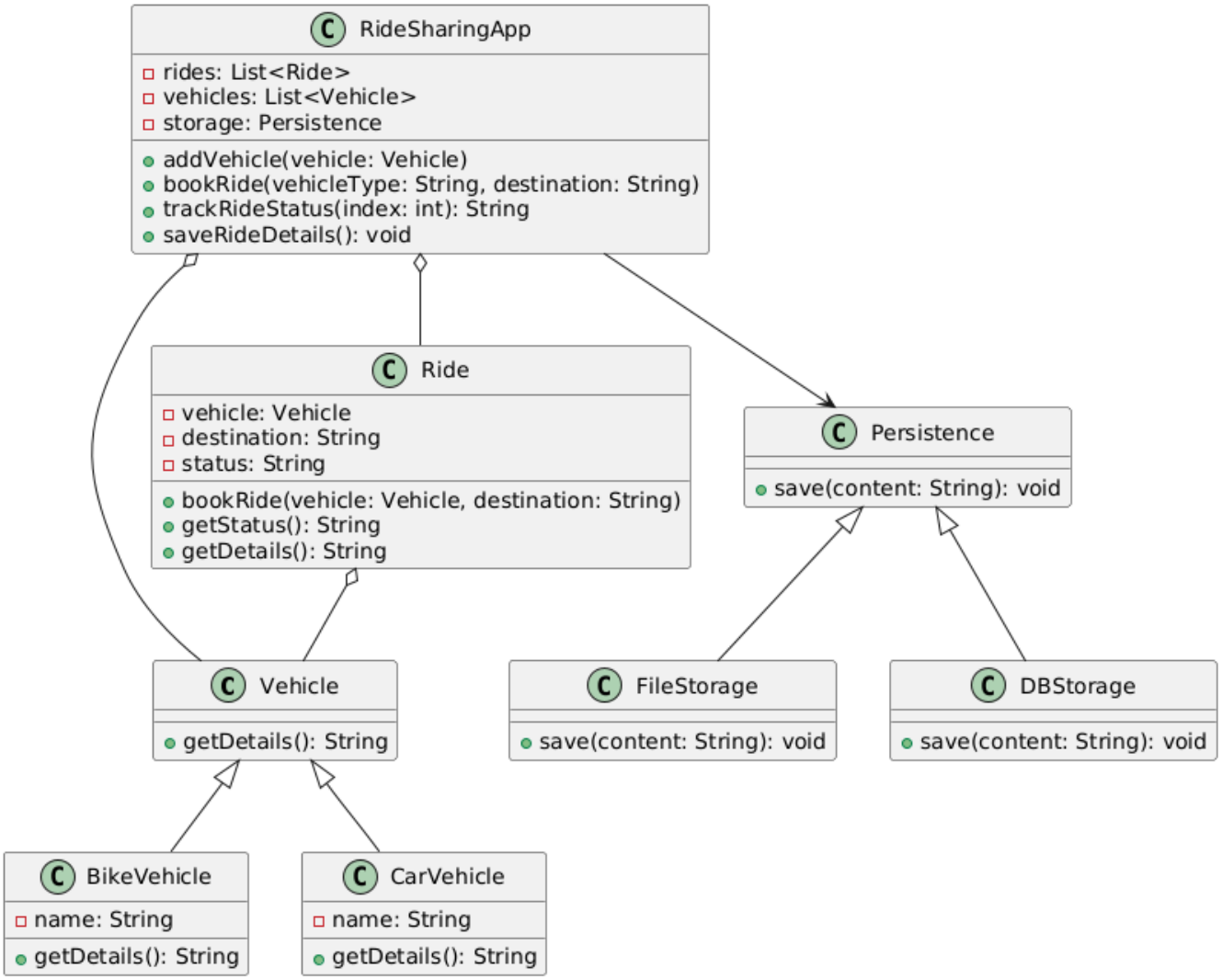
We'll refactor the design using SOLID and OOP principles to make it modular, scalable, and maintainable. We'll:

- Use polymorphism for vehicle types.
- Separate responsibilities (vehicle management, ride booking, persistence).
- Introduce abstractions for extensibility.

3.1 Components

1. **Vehicle (Abstract Class)**: Represents vehicles (e.g., bike, car) with a `getDetails()` method.
2. **BikeVehicle, CarVehicle**: Concrete vehicle types.
3. **Ride**: Manages ride details (vehicle, destination, status).
4. **Persistence (Abstract Class)**: Handles saving ride details.
5. **FileStorage, DBStorage**: Concrete storage implementations.
6. **RideSharingApp**: Coordinates client requests, delegating to Ride and Persistence.

3.2 UML Diagram (Improved Design)



3.3 Code (Improved Design)

Code can be accessed on my GitHub:
https://github.com/MuhammadTahaNasir/Systems_Design-Codes-Notes/blob/main/Week-1/Good_Design/Good_Design.cpp

3.4 Output

```
Ride Status: Driver Assigned
Ride details saved successfully!
```

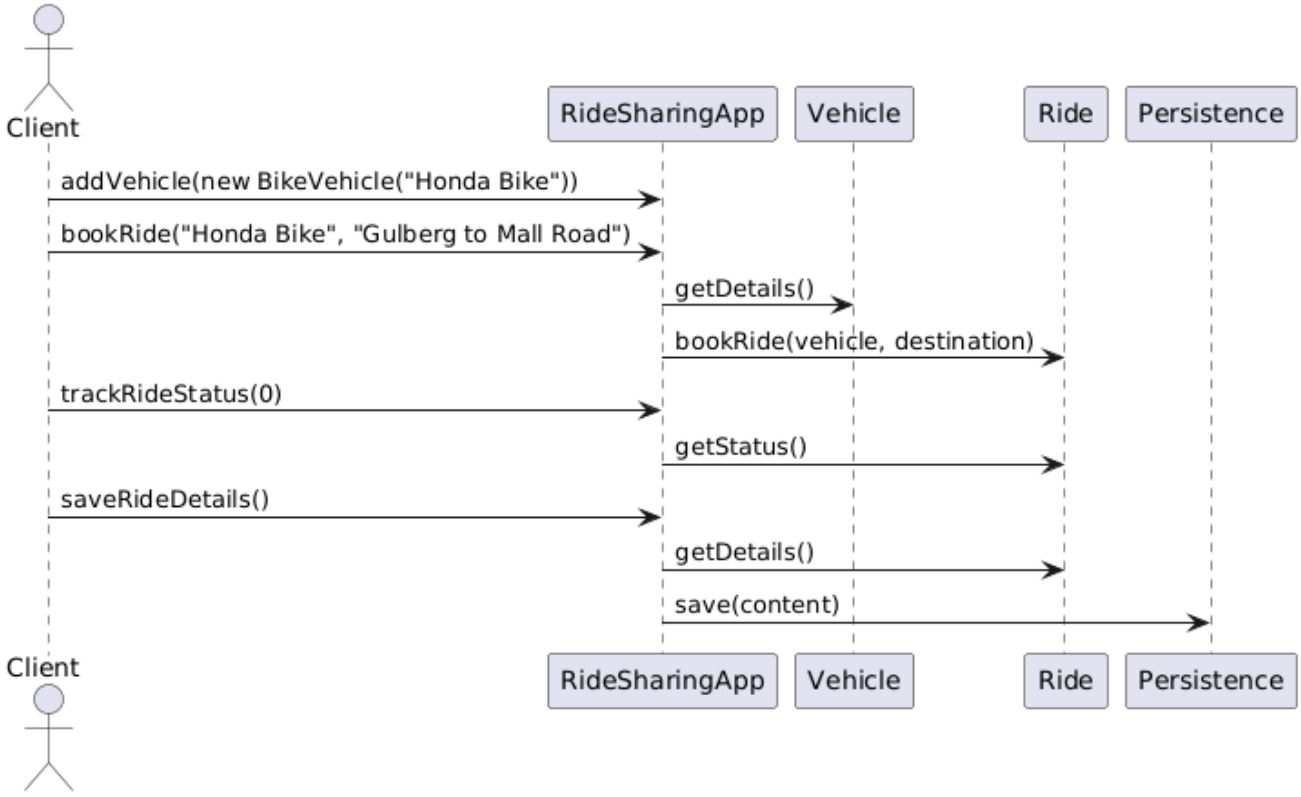
File Content (ride_details.txt):

```
Bike: Honda Bike to Gulberg to Mall Road (Driver Assigned)
```

3.5 Why This Design is Better

1. **Single Responsibility Principle (SRP)**:
   - `Vehicle` handles vehicle details.
   - `Ride` manages ride booking and status.
   - `Persistence` handles saving.
   - `RideSharingApp` coordinates client requests.
2. **Open-Closed Principle (OCP)**:
   - Add new vehicle types (e.g., `RickshawVehicle`) by extending `Vehicle` without modifying existing code.
   - Add new storage (e.g., cloud storage) by extending `Persistence`.
3. **Liskov Substitution Principle (LSP)**:
   - `BikeVehicle` or `CarVehicle` can replace `Vehicle` in `Ride` without issues.
4. **Interface Segregation Principle (ISP)**:
   - Vehicle has only `getDetails()`, and `Persistence` has only `save()`.
5. **Dependency Inversion Principle (DIP)**:
   - `RideSharingApp` depends on `Persistence` abstraction, not concrete `FileStorage`.

3.6 Sequence Diagram

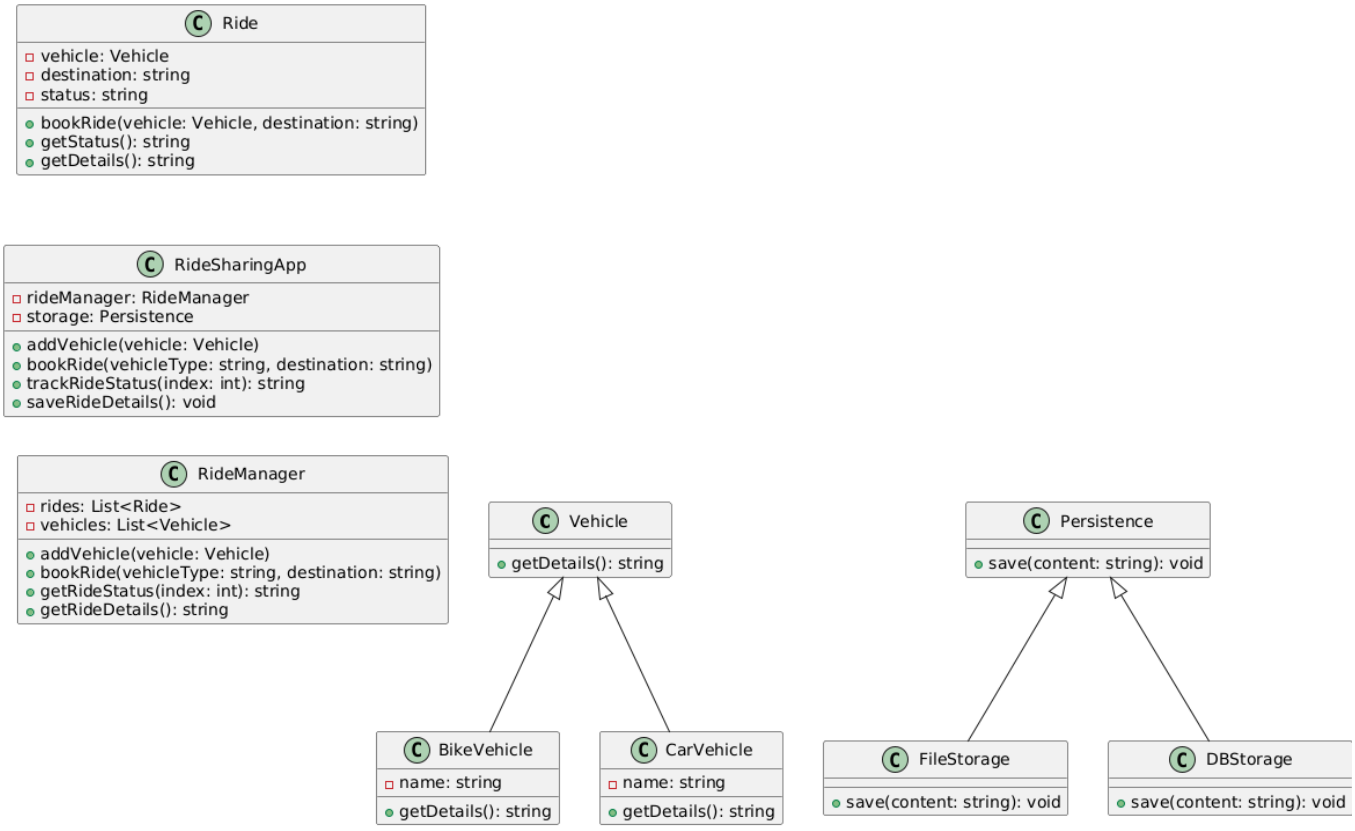


4. Further Improvement

Concern: The `RideSharingApp` class still handles vehicle management, ride booking, and saving, potentially violating SRP and the **Principle of Least Knowledge** (it knows about Vehicle and Ride internals).

Solution: Introduce a `RideManager` class to handle ride-related operations, leaving `RideSharingApp` as a facade for client interaction.

4.1 Updated UML Diagram



4.2 Updated Code (`RideManager`)

Code can be accessed on my GitHub:
https://github.com/MuhammadTahaNasir/Systems_Design-Codes-Notes/blob/main/Week-1/Good_Design/Good_Design_Updated.cpp

4.3 Benefits

- **SRP**: `RideManager` handles ride and vehicle management; `RideSharingApp` is a facade for client interaction.
- **Principle of Least Knowledge**: `RideSharingApp` only interacts with `RideManager` and `Persistence`, not their internals.
- **Trade-Off**: Adds a new class, increasing complexity but improving modularity.

---

5. SOLID Principles in the Design

Principle	How It's Applied
SRP	Vehicle handles vehicle details, Ride manages ride data, Persistence saves data, RideManager manages rides/vehicles, RideSharingApp coordinates.
OCP	Add RickshawVehicle or CloudStorage without modifying existing code.
LSP	BikeVehicle or CarVehicle can replace Vehicle in Ride.
ISP	Vehicle has only getDetails(), Persistence has only save().
DIP	RideSharingApp depends on Persistence and RideManager abstractions.

6. Conclusion

Project Summary:

- Designed a **Pakistani Ride-Sharing App** inspired by Careem/Bykea, focusing on ride booking, vehicle management, and saving ride details.
- Started with a bad design (single class), then refactored into a modular, scalable system using SOLID and OOP principles.
- Used UML diagrams (Class, Sequence) to visualize structure and interactions.

Why It's Relevant:

- Addresses local needs (e.g., bike rides in congested areas).
- Scalable for AI features like fare prediction or Urdu notifications.
- Simple yet practical for LLD interviews.

Interview Tips:

- Start with a bad design to show flaws.
- Improve iteratively, justifying with SOLID principles.
- Use UML diagrams to explain your design.
- Discuss trade-offs (e.g., complexity vs. modularity).

This project shows how to approach LLD for a ride-sharing app. Practice this for interviews, and let's meet in the next week's notes with another project.