Car Rental System – System Documentation

1. System Architecture

The Car Rental System is implemented with a **Layered Architecture**, ensuring separation of concerns and modular design:

- **UI Layer**
  - Classes: `CarUI`, `CustomerUI`, `UserUI`, `RentalUI`, `LoginUI`, `MainMenu`.
  - Purpose: Handles user interactions, displays menus, and passes user input to the Business Layer.
- **Business Layer (Service Layer)**
  - Classes: `CarService`, `RentalService`, `UserService` (with interfaces `ICarService`, `IRentalService`, `IUserService`).
  - Purpose: Implements business rules (e.g., only available cars can be rented). Acts as the core logic of the application.
- **Domain Layer**
  - Models: `Car`, `User`, `Rental`, `Vehicle`.
  - DTOs: `CarDto`, `UserDto`, `RentalDto`.
  - Mappers: `CarMapper`, `UserMapper`, `RentalMapper`.
  - Purpose: Defines entities and data transformation between persistence and business logic.
- **Data Layer (Persistence)**
  - Repositories: `CarRepository`, `UserRepository`, `RentalRepository`.
  - `DBManager`: Singleton managing the database session.
  - Purpose: Handles persistence (CRUD operations) with SQLAlchemy and SQLite.

This layered separation improves **maintainability, testability, and scalability**.

---

2. Design Patterns in Use

Repository Pattern

- Classes: `CarRepository`, `UserRepository`, `RentalRepository`.
- Purpose: Abstract database access, providing a clean API for services.
- Benefit: Decouples business logic from persistence, enables mocking in tests.

Service Layer Pattern

- Classes: `CarService`, `RentalService`, `UserService`.
- Purpose: Encapsulates business rules, implements interfaces for flexibility.
- Example: RentalService ensures only available cars can be rented.

Data Transfer Object (DTO) Pattern

- DTOs: `CarDto`, `UserDto`, `RentalDto`.
- Purpose: Transfer data safely between layers without exposing database entities.
- Benefit: Enhances encapsulation, allows independent evolution of internal models.

Mapper / Adapter Pattern

- Classes: `CarMapper`, `UserMapper`, `RentalMapper`.
- Purpose: Converts between domain models and DTOs.
- Benefit: Isolates transformation logic in one place.

Dependency Inversion Principle (via Interfaces)

- Services depend on abstractions (`ICarService`, `IRentalService`) not implementations.
- Repositories follow the same principle.

Factory / Installer Pattern

- `ServiceInstaller`: Wires up repositories and services.
- Purpose: Acts as a manual Dependency Injector.

---

3. SOLID Principles

1. **Single Responsibility Principle (SRP)**
    - Each class has one purpose (e.g., `CarRepository` only persistence, `CarService` only logic).
2. **Open/Closed Principle (OCP)**
    - System can be extended (new repositories, new DB engines) without modifying existing code.
3. **Liskov Substitution Principle (LSP)**
    - Interfaces allow swapping implementations without breaking functionality.
4. **Interface Segregation Principle (ISP)**
    - Interfaces are small and specific (`ICarService` handles only car-related operations).
5. **Dependency Inversion Principle (DIP)**
    - High-level modules (`CarService`) depend on abstractions, not concrete repositories.

---

4. Best Practices Evident

- **Separation of Concerns**: UI, business, and data persistence layers are decoupled.
- **Encapsulation**: DTOs protect internal models from leaking into UI.
- **Testability**: Interfaces allow mocking repositories/services for unit testing.
- **Scalability**: Adding new services/entities is straightforward.
- **Maintainability**: Database changes don't directly impact business logic.

---

5. Areas for Improvement

1. **Dependency Injection Framework**
    - Replace manual `ServiceInstaller` with a DI container (e.g., FastAPI's dependency injection).
2. **Error Handling & Validation Layer**
    - Add validation services for input (e.g., rental dates, user data).
3. **Domain-Driven Design (DDD)**
    - Define aggregates (e.g., Rental aggregate with car + user rules).
4. **Event-Driven Architecture / Observer Pattern**
    - Example: Trigger "Car Availability Updated" event when a rental is created or completed.

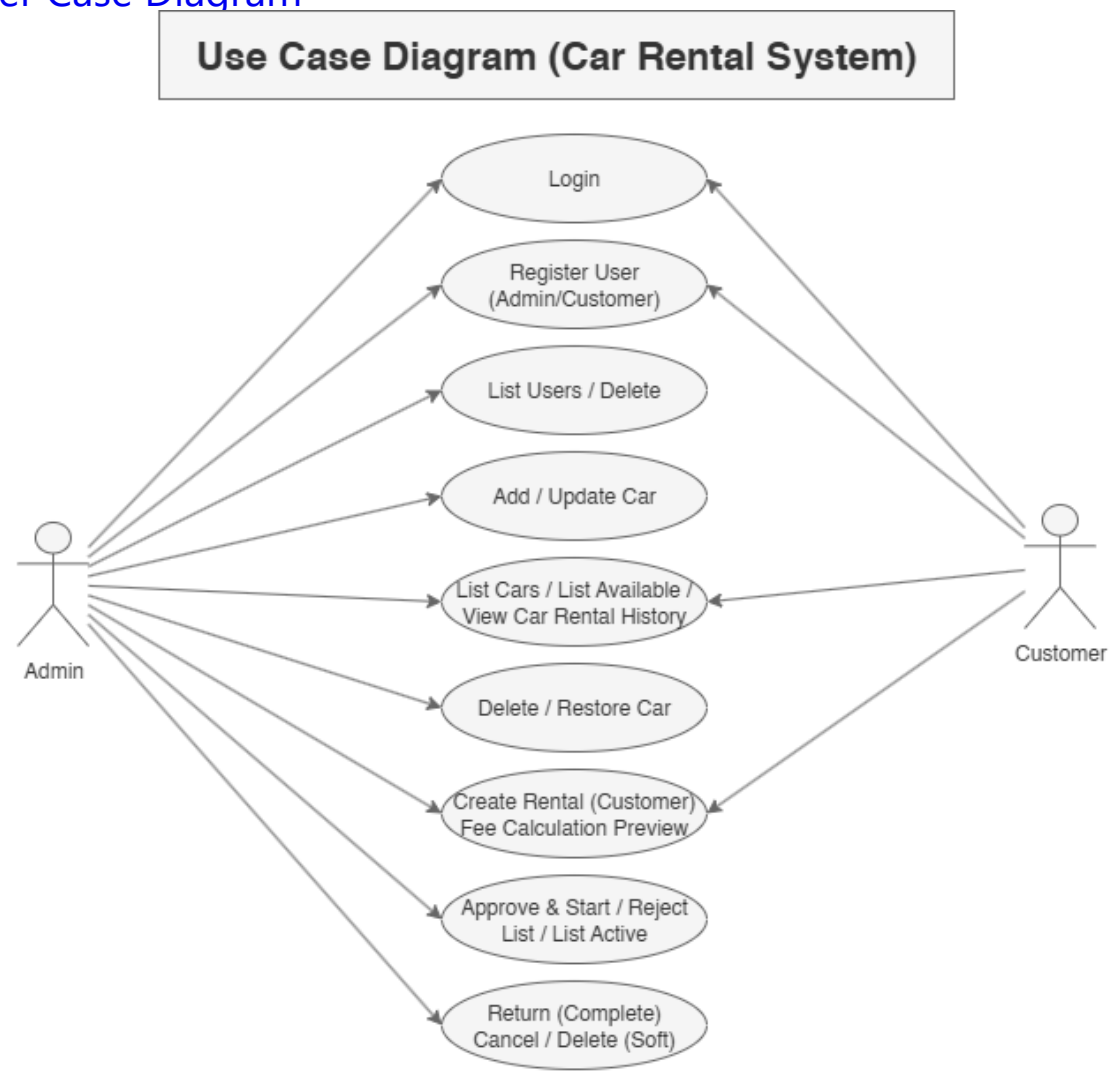
---

6. UML Diagrams

◈ Use Case Diagram

- **Actors**: Admin, Customer.
- **Admin Use Cases**: Manage Users, Manage Cars, Manage Rentals.
- **Customer Use Cases**: Register, Login, Browse Cars, Create Rentals.

User Case Diagram

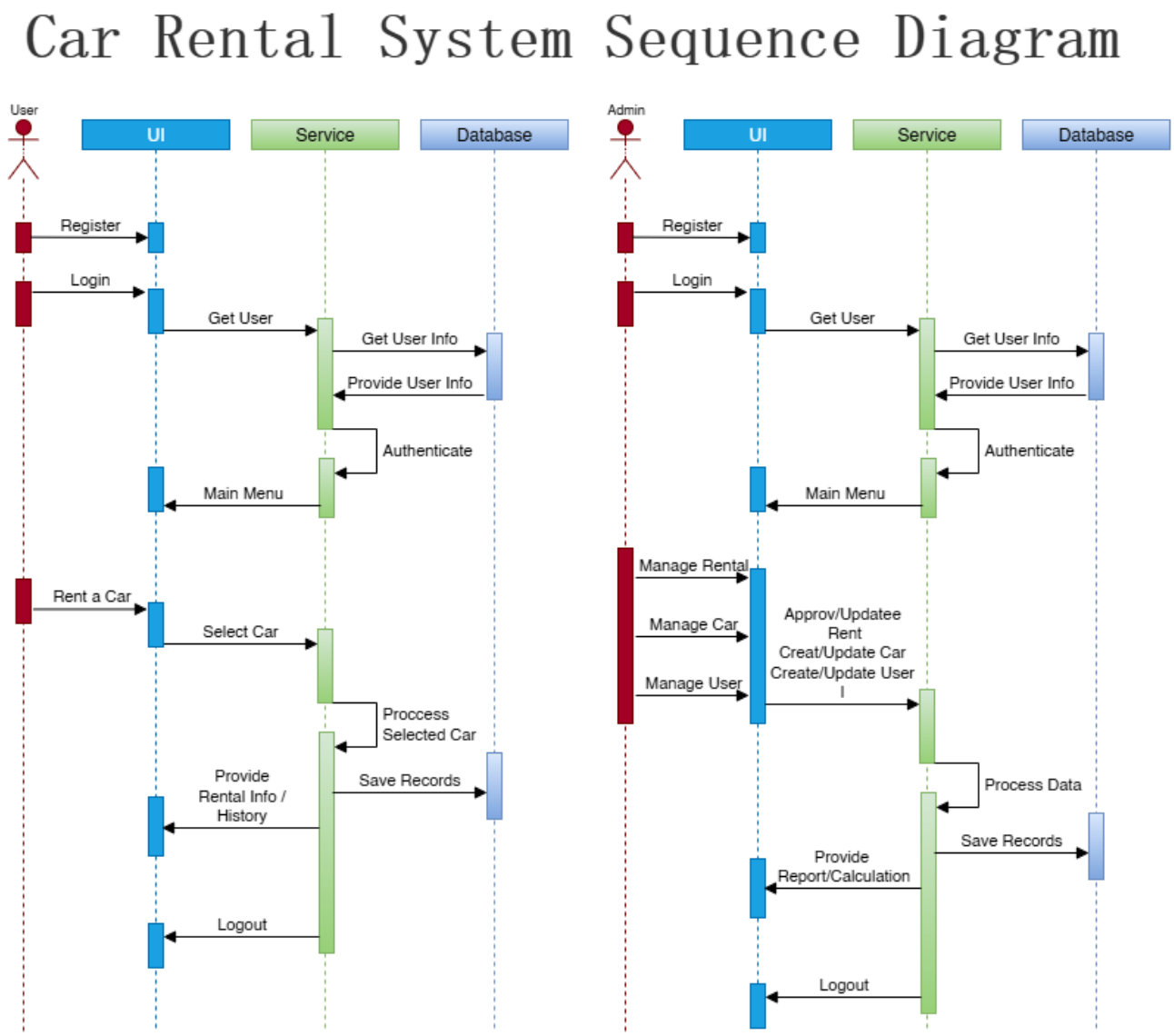


- The system boundary encloses all use cases provided by the application
- Admin: Manage Users (register/list/soft-delete), Manage Cars (add/list/soft-delete/restore), Manage Rentals (approve/reject/start/cancel/return).
- Customer: Register, Login, Browse Available Cars, Create Rental.

Sequence Diagram

- Shows flow between **User** → **UI** → **Services** → **Database**.
- Captures login, rental creation, approval, and return processes.

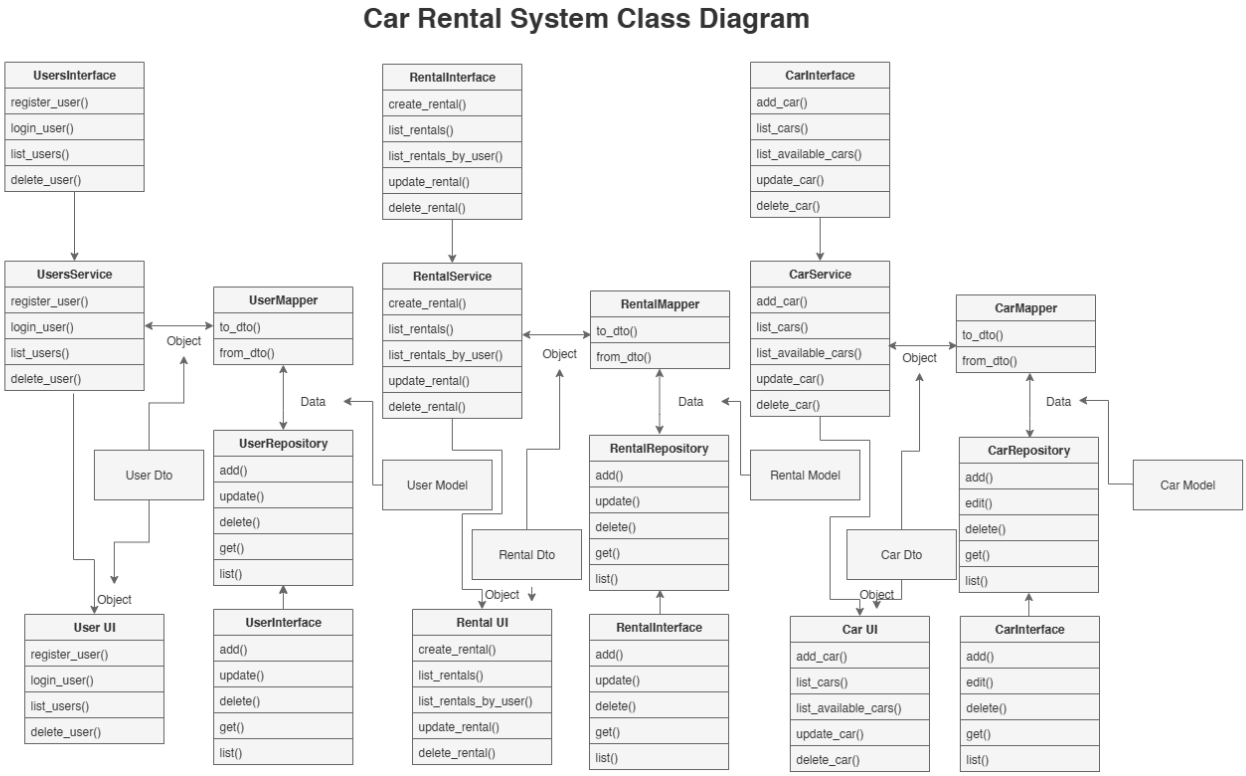
Sequence Diagram



Class Diagram

- **UI Classes ↔ Service Classes ↔ Repository Classes ↔ Domain Models & DTOs**.
- Demonstrates layered architecture and decoupling.

Class Diagram



7. Summary

The Car Rental System demonstrates:

- **Layered Architecture** for clean separation.
- Use of **Repository, Service Layer, DTO, Mapper, Dependency Inversion** patterns.
- Compliance with **SOLID principles**.
- Application of **best practices** in maintainability, scalability, and testability.

Together, the UML diagrams and architectural choices ensure the system is **robust, extensible, and professionally engineered**.