**Name: Abdulmajeed Almaweri**

**Date: 04/05/2025**

**Class: CIS 476**

**GitHub URL:** [aalmaweri12/DriveShareCIS476](https://github.com/aalmaweri12/DriveShareCIS476)

**Video URL:** <https://youtu.be/6tO2lQJELks>

**Video URL:** [**https://1drv.ms/v/c/c85f45401e6132f6/ERT0A4TYrlVFlPjxAdvAtcYB0LfDiJKoUVDOwiaZdh2ABQ?e=6jDudf**](https://1drv.ms/v/c/c85f45401e6132f6/ERT0A4TYrlVFlPjxAdvAtcYB0LfDiJKoUVDOwiaZdh2ABQ?e=6jDudf)

**Design Patterns in Car Sharing Application**

**Introduction**

This report documents the design and implementation of a car sharing application named "DriveShare". The application is built using modern software design patterns to ensure maintainability, extensibility, and robustness. The system allows users to list their cars for rent, search for available cars, make bookings, process payments, and communicate with car owners.

The following design patterns have been implemented in the system:

* Builder Pattern for Car Listing creation
* Chain of Responsibility Pattern for Password Recovery
* Mediator Pattern for UI Components
* Observer Pattern for Notification System
* Proxy Pattern for Payment Processing
* Singleton Pattern for User Session Management

**Class Diagrams and Descriptions**

**Builder Pattern**

The Builder pattern is used to construct complex Car objects step by step. This pattern is particularly useful for creating car listings with various configurations while keeping the construction process consistent.

A screenshot of a computer

AI-generated content may be incorrect.

**Description:**

* **Car**: The product class representing a car listing with various attributes.
* **CarListingBuilder**: Provides step-by-step construction of Car objects with fluent interface (method chaining).
* **CarListingDirector**: Defines common building sequences for standard car types (economy, luxury, custom).

The Builder pattern separates the construction of complex Car objects from their representation, allowing the same construction process to create different representations. This pattern is especially useful as car listings have many optional parameters that would make a constructor with many parameters unwieldy.

**Chain of Responsibility Pattern**

The Chain of Responsibility pattern is implemented for the password recovery system. It allows multiple security questions to be processed in sequence, with each handler deciding either to process the request or pass it to the next handler.

A screenshot of a computer screen

AI-generated content may be incorrect.

**Description:**

* **PasswordRecoveryHandler**: Abstract base class defining the chain structure and interface.
* **QuestionHandlers**: Concrete handlers that validate specific security questions.
* **PasswordResetHandler**: Final handler that performs the actual password reset.

This pattern allows security questions to be processed in sequence, with each handler deciding whether to pass the request to the next handler. The chain is only as strong as its weakest link, ensuring that all security questions must be answered correctly before the password is reset.

**Mediator Pattern**

The Mediator pattern is used for coordinating communication between UI components. This pattern reduces direct dependencies between components by having them communicate indirectly through a mediator object.

A diagram of a computer

AI-generated content may be incorrect.

**Description:**

* **UIMediator**: Central hub that coordinates communication between components.
* **BaseComponent**: Abstract base class for all UI components that need to communicate.
* **Concrete Components**: SearchComponent, BookingComponent, and MessageComponent that perform specific tasks.

The Mediator pattern decouples components by eliminating direct communication between them. Instead, they communicate indirectly through the mediator, which centralizes control logic. This makes the system more maintainable as components don't need to know about each other's implementation details.

**Observer Pattern**

The Observer pattern is implemented for the notification system. It enables multiple notification methods (email, in-app) to respond to changes in the system state without tight coupling between the notification system and business logic.

A screenshot of a computer

AI-generated content may be incorrect.

**Description:**

* **NotificationObserver**: Abstract base class for notification mechanisms.
* **Concrete Observers**: EmailNotifier and AppNotifier that implement specific notification channels.
* **NotificationSubject**: Maintains the list of observers and notifies them of changes.
* **BookingManager**: Concrete subject that triggers notifications on booking events.

This pattern allows for a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It enables the system to send both email and in-app notifications in response to booking events without tight coupling between the booking system and notification mechanisms.

**Proxy Pattern**

The Proxy pattern is used for payment processing. It provides a surrogate or placeholder for another object to control access to it, adding functionality such as security checks and notifications.

A screenshot of a computer

AI-generated content may be incorrect.

**Description:**

* **IPaymentProcessor**: Interface defining the contract for payment processors.
* **RealPaymentProcessor**: Performs the actual payment processing logic.
* **PaymentProxy**: Adds security checks before and notifications after delegating to the real processor.

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. In this case, the PaymentProxy adds security checks before allowing the payment to proceed and notifies users after successful payment processing, without changing the RealPaymentProcessor's code.

**Singleton Pattern**

The Singleton pattern is used for user session management. It ensures that only one user session exists at a time within the application.

A close-up of a diagram

AI-generated content may be incorrect.

**Description:**

* **UserSession**: Class with static instance management to ensure only one instance exists.
* **Methods**: Handles user login/logout and session state management.

The Singleton pattern guarantees that a class has only one instance and provides a global point of access to that instance. This pattern is appropriate for user session management as it ensures consistent session state across the application.

**Database Schema**

The application uses a relational database with the following schema:

A diagram of a computer

AI-generated content may be incorrect.

**Description:**

* **User**: Stores user account information and security questions.
* **Car**: Contains car listings with details like model, price, availability.
* **Booking**: Records bookings made by users for cars with start/end dates.
* **Payment**: Tracks payments for bookings.
* **Message**: Stores communication between users regarding bookings.
* **Notification**: Records notifications sent to users.

**User Interface Mockups**

**Login page**

**A screenshot of a login page

AI-generated content may be incorrect.**

**Sign up Page**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Home Page**

A screenshot of a computer

AI-generated content may be incorrect.

The home page features a search component allowing users to find available cars by location and dates. It also displays featured car listings and user testimonials.

**Car Listing Page**

A screenshot of a computer

AI-generated content may be incorrect.

The car listing page displays detailed information about a car, including photos, specifications, availability calendar, and pricing. Users can book the car directly from this page.

**Booking Management**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The booking management interface allows car owners to view and respond to booking requests. They can confirm or reject bookings and communicate with potential renters.

**Payment Processing**

A screenshot of a chat

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a web page

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The payment interface provides a secure way for users to pay for their bookings. It includes fraud detection and confirmation notifications.

**Messaging System**

A screenshot of a chat

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

The messaging system enables communication between car owners and renters. It is context-aware and organized by booking, making it easy to track conversations.

**Password Recovery**

A screenshot of a login page

AI-generated content may be incorrect.

A screenshot of a login page

AI-generated content may be incorrect.

A screenshot of a identity

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a login screen

AI-generated content may be incorrect.

The password recovery interface implements the Chain of Responsibility pattern, guiding users through a series of security questions before allowing password reset.

**References**

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
2. Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.
3. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
4. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
5. Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.