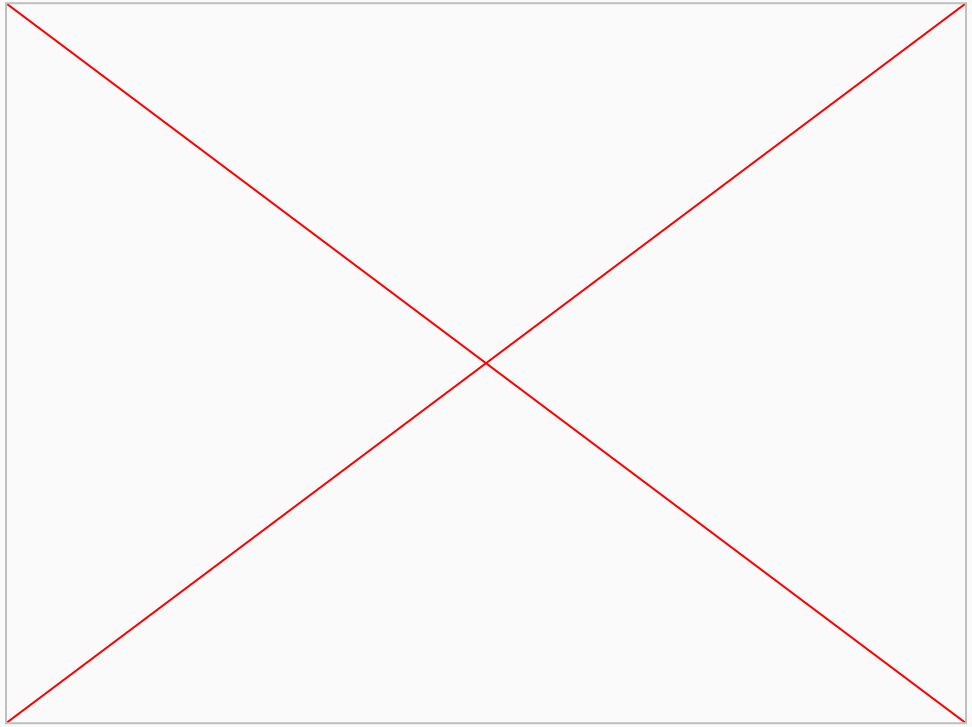
DriveShare Application

Abraham Abdulkarim, Ali Almuthafar, Alexis Whisnant

Demonstration

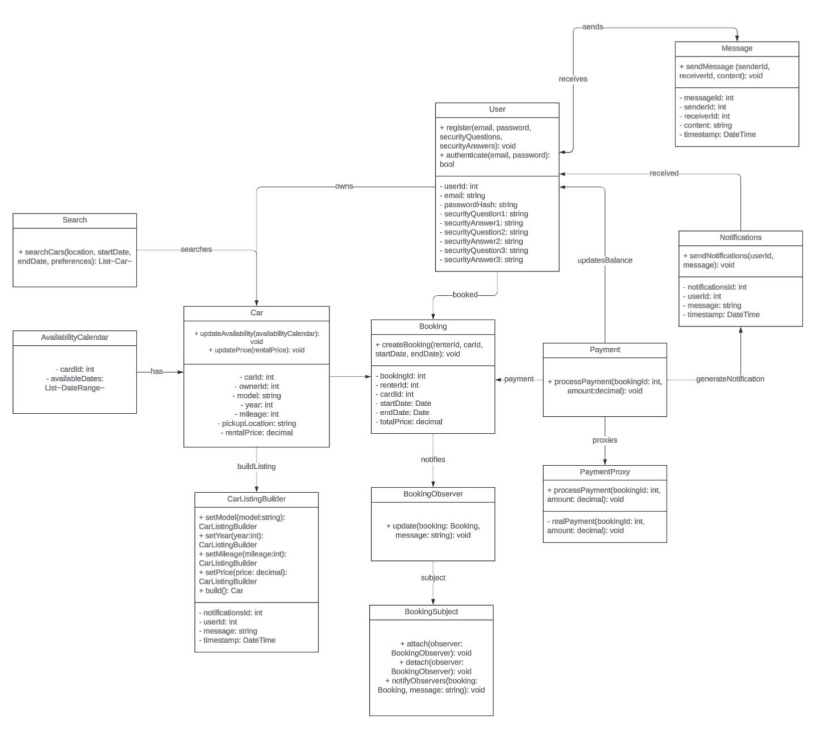


LINK

Demonstration Video

Design Explanation

Class Diagram



Description

User: Manages user accounts, authentication, and security.

Car: Represents a rental car with details and availability.

Booking: Handles car reservations between renters and owners.

AvailabilityCalendar: Tracks car availability for rentals.

Search: Enables users to find available cars based on criteria.

CarListingBuilder: Creates car listing objects with flexible attributes.

Notification: Represents and sends booking and other updates.

Payment: Processes rental payments.

PaymentProxy: Securely manages payment transactions.

BookingObserver/Subject: Notifies users of booking updates using the Observer pattern.

Message: Facilitates communication between users.

User Creation

Singleton Pattern

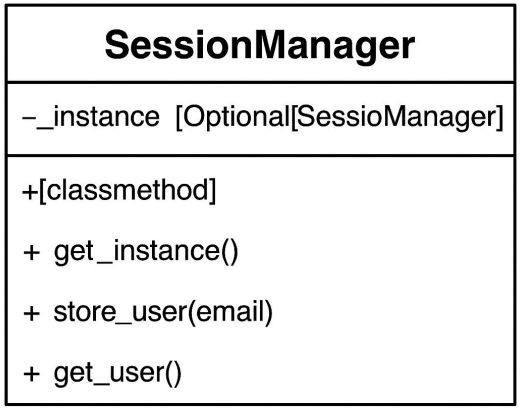
Singleton Pattern in User Authentication: SessionManager

The Singleton pattern ensures a class has **only one instance globally** and provides a **single point of access** to it. In DriveShare, this is crucial for managing the currently logged-in user securely across all screens of the GUI.
Class: `SessionManager:`

Singleton class that:

- Stores the **current user object** (host or guest)
- Prevents multiple conflicting user sessions

Is used by **all GUI modules** to access session data (like `user.email`, `user.role`)



Booking Notifications

Observer Pattern

The **Observer Pattern** establishes a **one-to-many relationship** between objects so that when one object changes state, all its **subscribers (observers)** are notified automatically.

In **DriveShare**, this pattern is used to send **notifications** to the user (host or guest) when:

- A **booking is made**
- A **payment is processed**
- A **review is submitted**

Key Classes Involved

1. **Subject (Publisher):** `BookingService`
2. **Observer (Listener):** `BookingObserver`

Booking Notifications

Observer Pattern

`BookingService` **(Subject)**

Maintains a list of observers and notifies them when a booking-related event occurs.

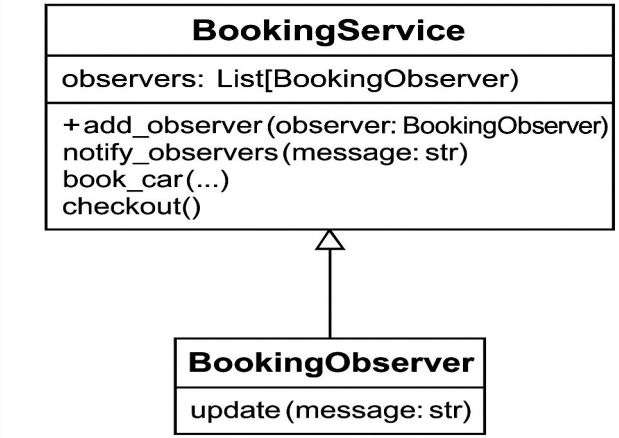
BookingObserver (Observer)
Located in the same file, `BookingObserver` listens for updates and displays them in the GUI or logs them.

Example in code:

```
class BookingObserver:

    def update(self, message):

        print(f"[NOTIFICATION]: {message}")

        # Could be extended to show GUI messages, log to file, or send email
```

Output:

The observer receives the update and acts on it (currently prints to console, can be extended for GUI messages or logging).

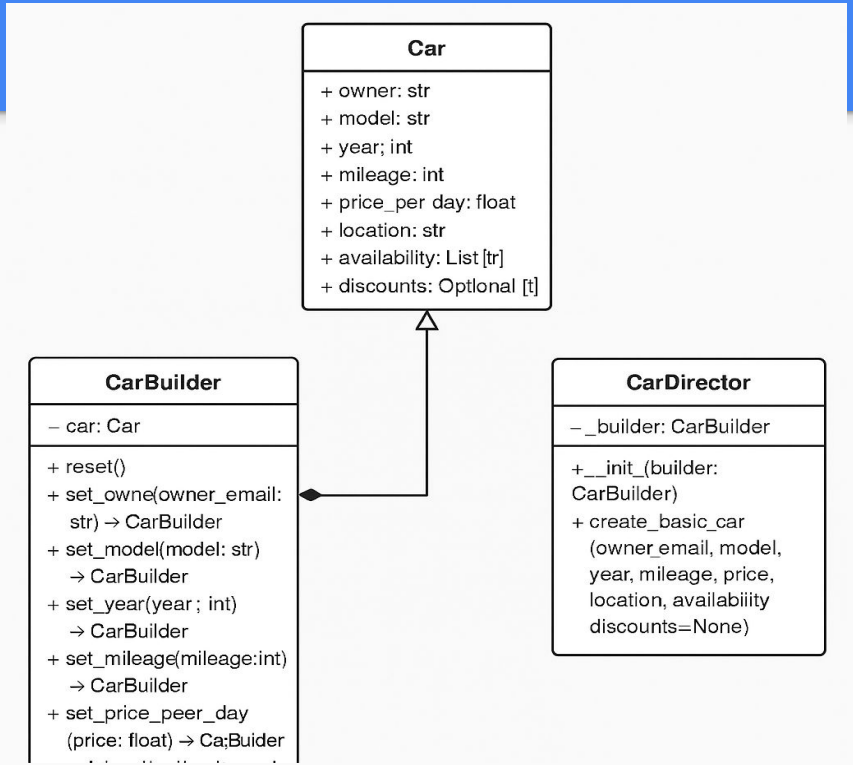


Listing Cars:

Builder Pattern

The **Builder Pattern** is used to construct complex objects **step-by-step**, allowing different representations of the same object. In DriveShare, it's used to build a `Car` object with all required fields like model, year, price, availability, etc.

This pattern separates the **construction logic** from the **representation**, improving modularity and code reusability.



Builder Pattern: Implementation

In our DriveShare project, the Builder Pattern is used to streamline and modularize the creation of car listings by encapsulating three key classes: `Car`, `CarBuilder`, and `CarDirector`. The `Car` class represents the final product, holding all relevant attributes like model, year, mileage, and availability. The `CarBuilder` provides step-by-step setter methods to construct the car object, enabling flexible and reusable logic. To further standardize the process, the `CarDirector` defines a predefined sequence for assembling a complete `Car` using the builder. This encapsulation allows us to separate the car creation logic from the GUI, ensuring that the object construction is clean, maintainable, and scalable as the application grows.

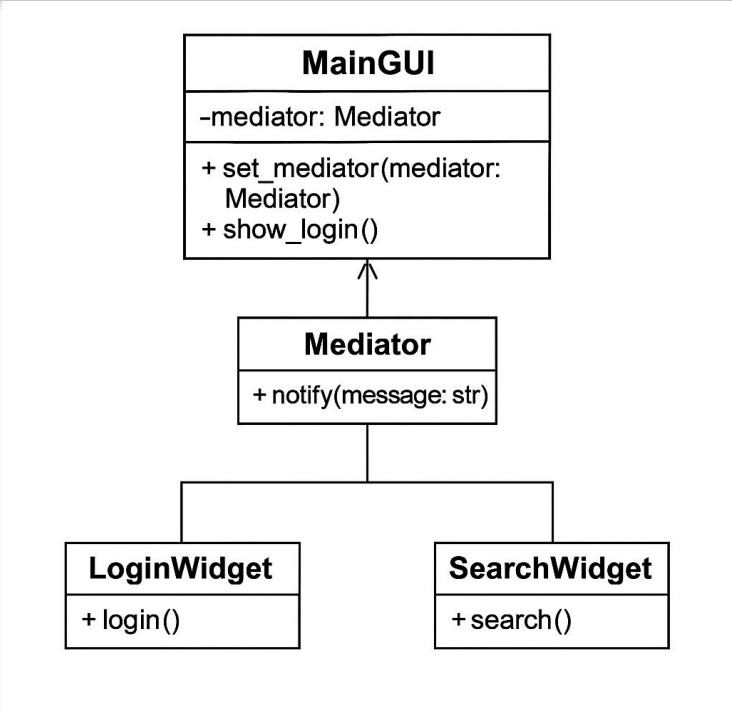
Mediator Pattern:

Key Mediator Class: `DriveShareGUI`

Located in: `main_gui.py`

This class serves as the **mediator** between:

- GUI buttons and actions
- User roles (host/guest)
- External services (`AuthService`, `CarService`, `BookingService`, etc.)
- Modular UI forms (imported functions from `guest_features.py`, `vehicle_forms.py`, `host_features.py`)



Mediator Pattern: Key Features That Rely on Mediator:

Role-Based Dashboards
Dynamically displays host vs guest interface after login using centralized control.

Authentication & Session Flow
Handles login/logout, session reset, and dashboard redirection in one place.

Modular Feature Navigation
Routes actions like booking, messaging, and payments to the correct form without components knowing about each other.

Consistent Frame Management
Maintains a shared `main_frame`, preventing GUI overlap and layout conflicts.

Proxy Pattern:

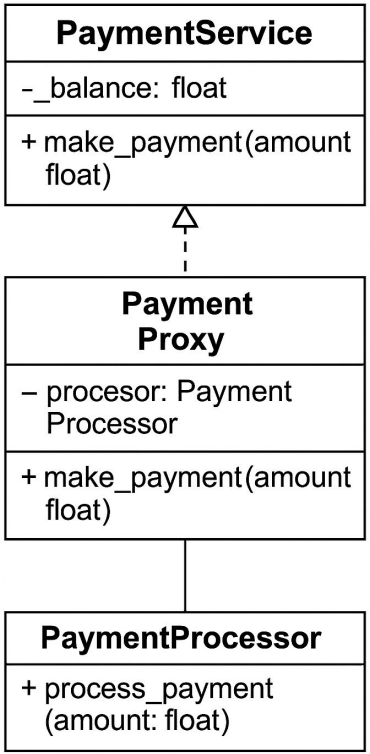
In DriveShare, this pattern is used to handle **secure and controlled access to the payment system**, without interacting with real financial APIs.

Key Class: `PaymentProxy`

Located in: `app/services/payment_proxy.py`

This class acts as a **middle layer** between the user and the simulated payment transaction, allowing the system to:

- Process a fake payment securely
- Log or print payment details
- Trigger notifications

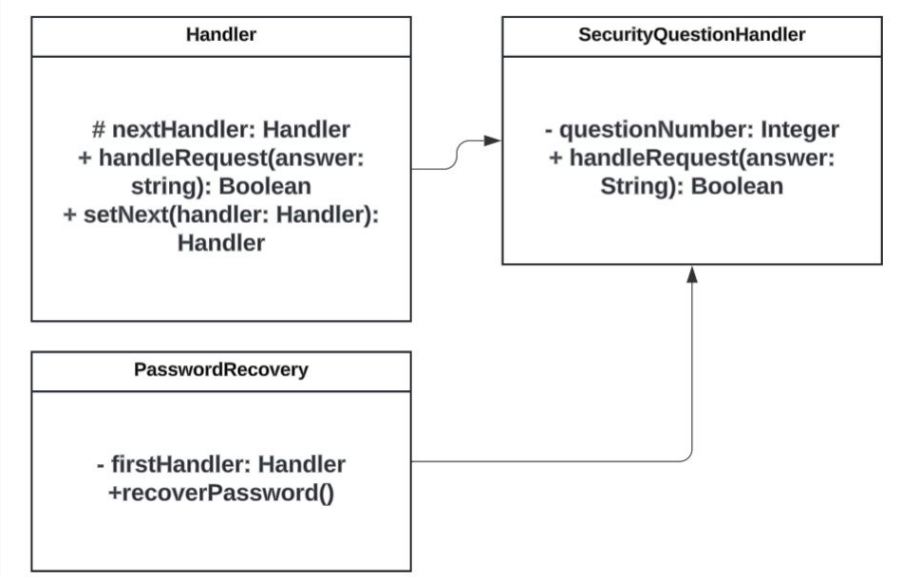


Chain of Responsibility Pattern:

This pattern allows a request to **pass through a chain of handlers**, where each handler can either process the request or pass it to the next one. In DriveShare, it's used to validate **three security questions** during password recovery in a step-by-step manner.

Core Concept

Each security question is handled by a separate "link" in the chain. If a user answers one correctly, the request is passed to the next handler. If any answer is wrong, the chain stops.



Password Recovery:

Chain of Responsibility Pattern

Pattern: Chain of Responsibility Pattern

The Chain of Responsibility pattern allows a request to pass through a chain of handlers, where each handler can either process the request or pass it to the next one. In DriveShare, this is used to validate security questions during password recovery in a step-by-step manner.

Core Concept:

Each security question is handled by a separate "link" in the chain. If a user answers one correctly, the request is passed to the next handler. If any answer is wrong, the chain stops.

Handler:

- Defines the interface for all concrete handlers.
- Methods: `handleRequest(answer: String, questionNumber: Integer): Boolean`, `setNext(handler: Handler): Handler`

SecurityQuestionXHandler: (e.g., `SecurityQuestion1Handler`, `SecurityQuestion2Handler`, `SecurityQuestion3Handler`)

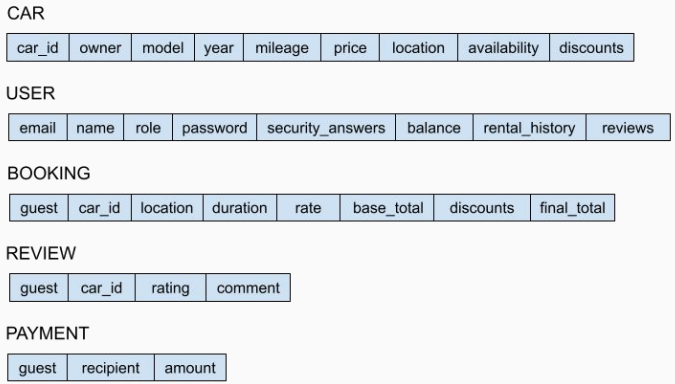
- Implements the `Handler` interface.
- Handles the validation of a specific security question.
- Methods: `handleRequest(answer: String, questionNumber: Integer): Boolean`

PasswordRecoveryService:

- Initiates the password recovery process.
- Manages the chain (often).
- Methods: `initiateRecovery()`

Database Schema:

The DriveShare application uses a database schema to manage data for users, cars, bookings, and payments. Key tables include Users (for user accounts), Cars (for rental car details), Bookings (for reservations), Availability Calendar (for car availability), and Payments (for transactions). Relationships between these tables are crucial; for example, `owner_id` in Cars links to Users, and `car_id` in Bookings links to Cars. This schema ensures data integrity and supports the application's core functionality.



Task List

- Abraham Abdulkarim
  - User Registration
  - Searching and Booking
  - Messaging and Communication
  - Rental History
- Ali Almuthafar
  - Car Listing and Management
  - UI Mediator
  - Payment Proxy
- Alexis Whisnant
  - Password Recovery
  - Reviews
  - Developed Class Diagrams