CIS 476 Term Project

Team Alpha

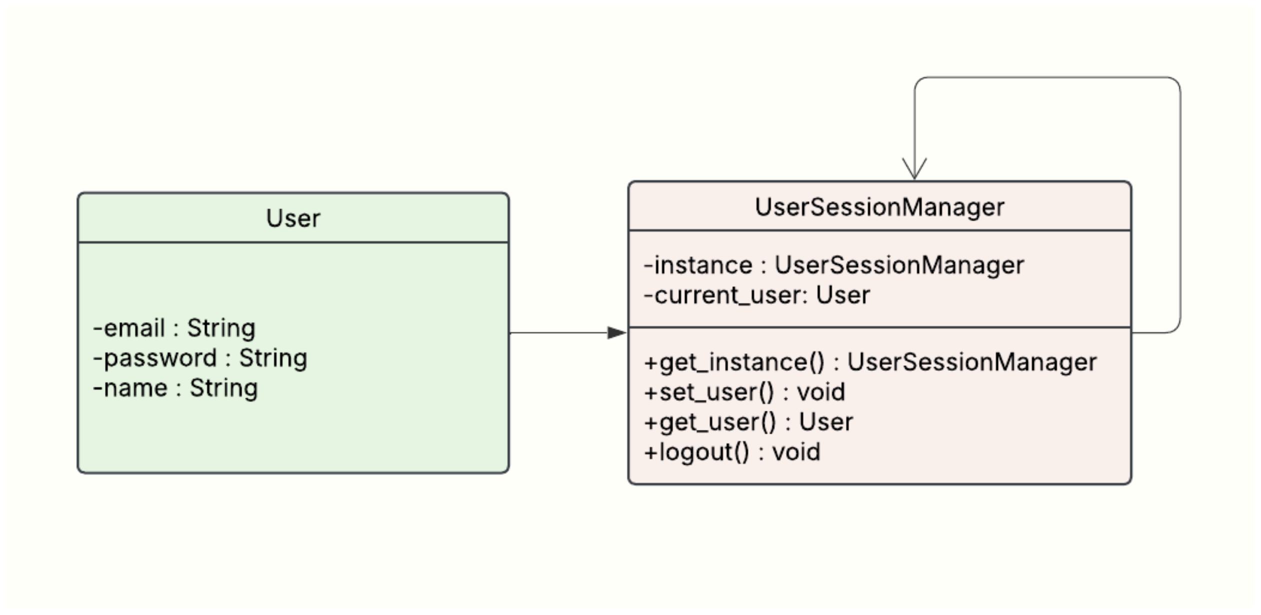
Leah Mirch, Zaynab Mourtada,
Souad Omar, Firas Abueida

UML Class diagrams

Singleton Pattern

The Singleton pattern is used to manage user sessions across the DriveShare platform. In the class diagram, the UserSessionManager class ensures that only one instance of the session manager exists during the application's runtime. This guarantees that session-related data (like the currently logged-in user) is consistent and globally accessible.

The User class contains user attributes such as email, password, and name. The UserSessionManager holds a reference to the current User and exposes methods to set and retrieve this user, as well as to handle logout operations. The method get_instance() returns the single instance of the session manager, allowing other parts of the application to interact with the session securely and reliably.

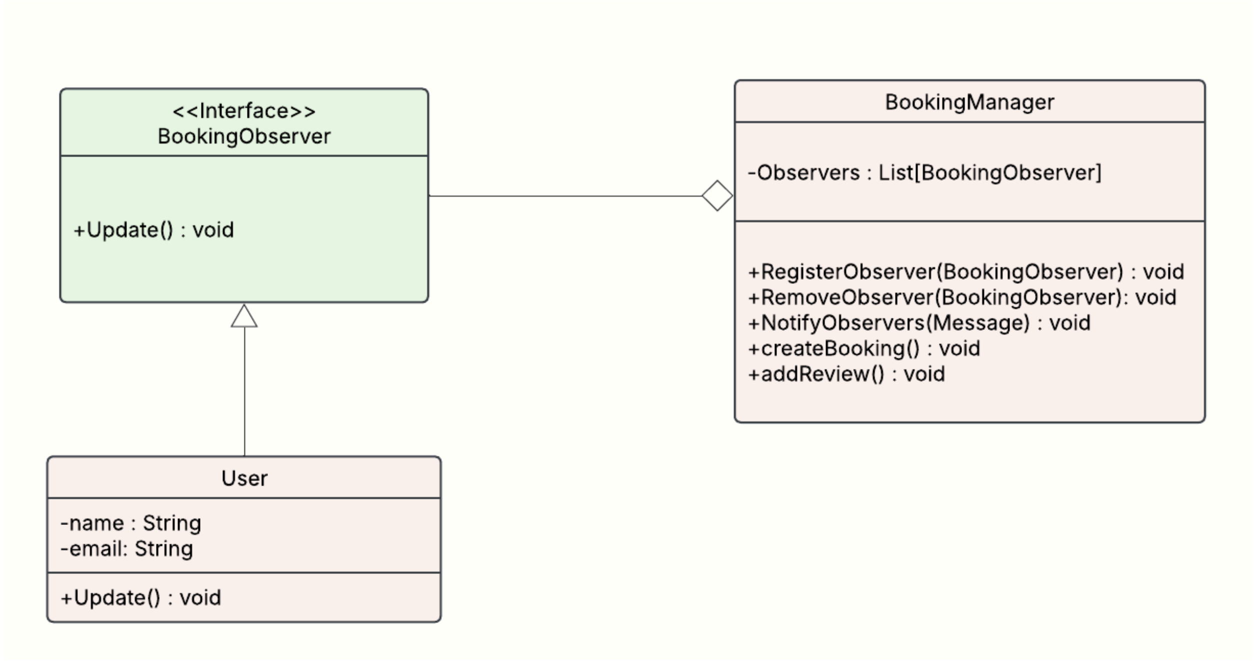


Observer Pattern

The Observer pattern is used to manage booking and review notifications within the DriveShare platform. In this diagram, the BookingManager class acts as the subject that maintains a list of registered observers, components interested in being notified of updates.

The BookingObserver interface defines a single method, Update(), which observers must implement to receive notifications. The User class implements this interface, allowing each user to be notified when relevant events occur, such as a new booking confirmation or a new review being added.

The BookingManager provides methods to register or remove observers and to broadcast messages using NotifyObservers(). When actions like createBooking() or addReview() are performed, all registered users are notified accordingly. This approach ensures a clean separation between the booking logic and the notification system.

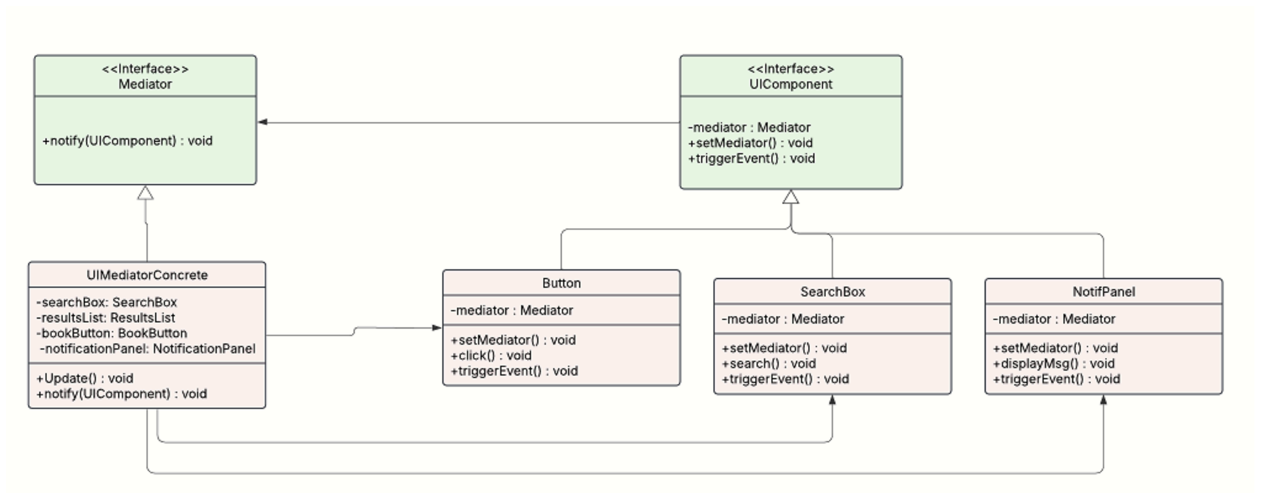


Mediator Pattern

The Mediator pattern is used to manage communication between UI components in DriveShare, ensuring a clean and decoupled architecture. Instead of UI elements interacting directly with one another, each component communicates through a central mediator, reducing dependencies and complexity.

In the diagram, the Mediator interface defines a notify() method that all concrete mediators must implement. The UIMediatorConcrete class serves as the central hub, coordinating communication between various UI components like SearchBox, Button, and NotificationPanel.

All UI components implement the UIComponent interface, which includes a reference to a Mediator and defines methods such as setMediator() and triggerEvent(). Each component also has its own unique behavior—for example, SearchBox handles user queries, Button can trigger actions like bookings, and NotificationPanel displays system messages. This design makes it easy to add or update UI components without affecting others.

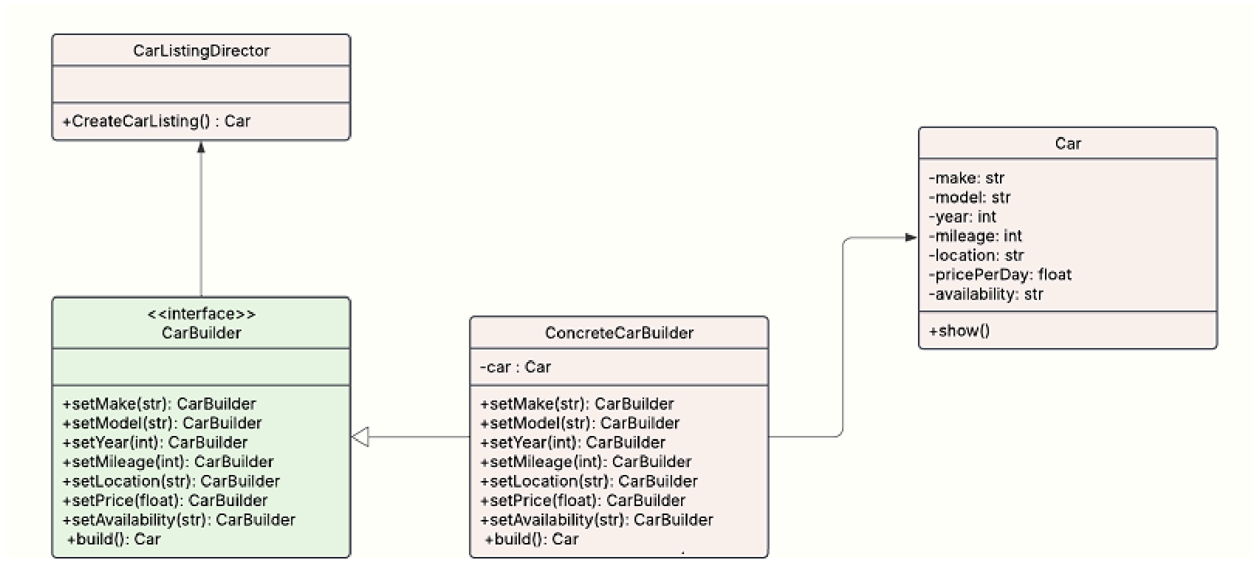


Builder Pattern

The Builder pattern is used in DriveShare to manage the flexible and modular creation of car listings. Each car listing contains multiple optional and required fields such as make, model, year, mileage, location, price, and availability. Instead of using a large constructor with many parameters, the Builder pattern enables a step-by-step and readable object creation process.

In the diagram, the CarBuilder interface defines the structure for setting each car attribute and a build() method to return the completed Car object. The ConcreteCarBuilder class implements these methods and internally constructs the Car instance.

The Car class itself is the final product containing all listing details and a method show() to display them. Optionally, the CarListingDirector class acts as the director, controlling how specific car listings are assembled (e.g., creating a standard or luxury listing). This separation of concerns makes the code cleaner, easier to maintain, and allows for reusable builder logic across different parts of the system.

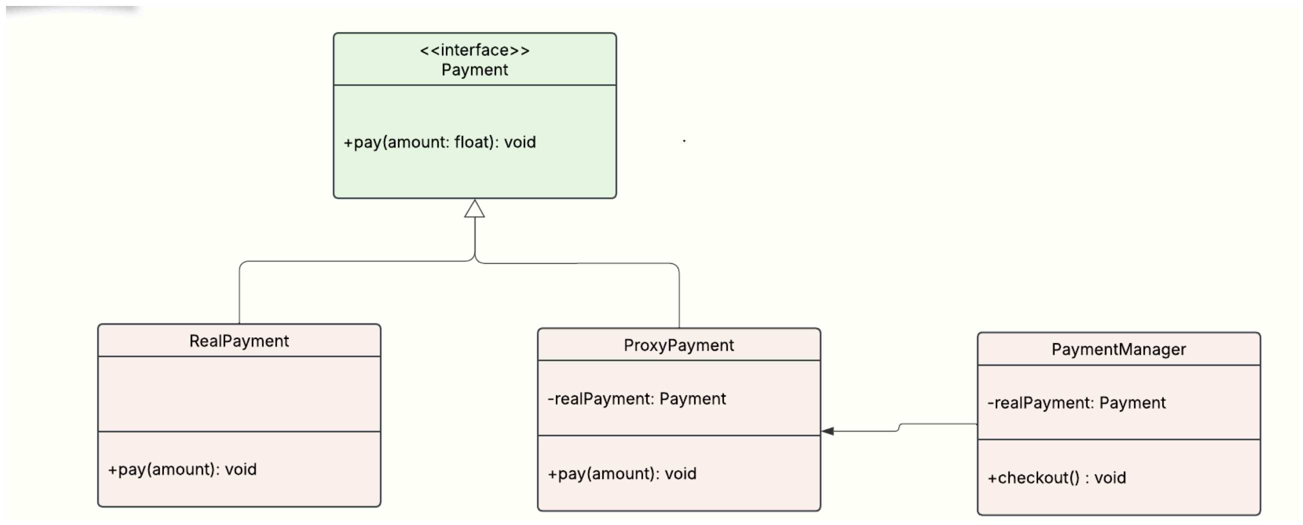

CarListingDirector
+CreateCarListing() : Car
<<interface>>
CarBuilder
+setMake(str): CarBuilder
+setModel(str): CarBuilder
+setYear(int): CarBuilder
+setMileage(int): CarBuilder
+setLocation(str): CarBuilder
+setPrice(float): CarBuilder
+setAvailability(str): CarBuilder
+build(): Car
ConcreteCarBuilder
-car : Car
+setMake(str): CarBuilder
+setModel(str): CarBuilder
+setYear(int): CarBuilder
+setMileage(int): CarBuilder
+setLocation(str): CarBuilder
+setPrice(float): CarBuilder
+setAvailability(str): CarBuilder
+build(): Car
Car
-make: str
-model: str
-year: int
-mileage: int
-location: str
-pricePerDay: float
-availability: str
+show()

Proxy Pattern

The Proxy pattern is used in DriveShare to handle payment processing in a secure and controlled manner. Rather than having the system directly interact with the real payment service, a proxy acts as an intermediary to manage communication, allowing for additional functionality like access control, logging, and simulation.

The Payment interface defines a standard pay() method. Both RealPayment and ProxyPayment classes implement this interface. The RealPayment class contains the actual logic for processing payments, while ProxyPayment holds a reference to a RealPayment object and delegates the payment operation after performing any necessary checks or setup.

The PaymentManager class interacts only with the Payment interface and uses it to initiate a transaction through its checkout() method. This design keeps the system flexible and testable, as real payments can be simulated during development or restricted without changing the core logic of the application.

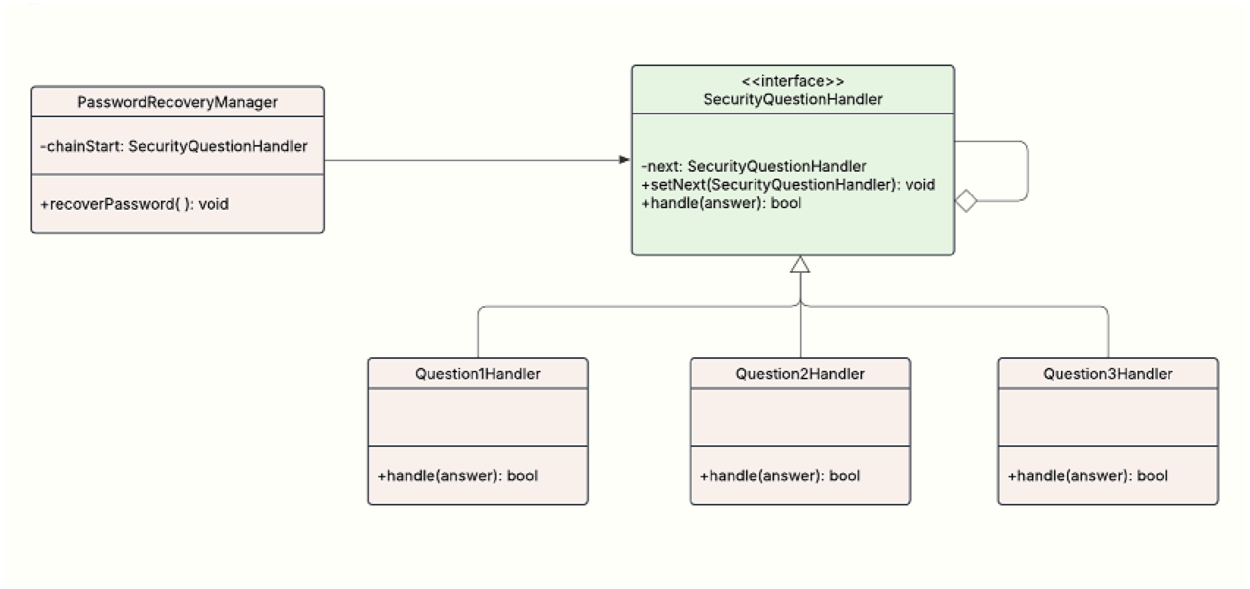


Chain of Responsibility

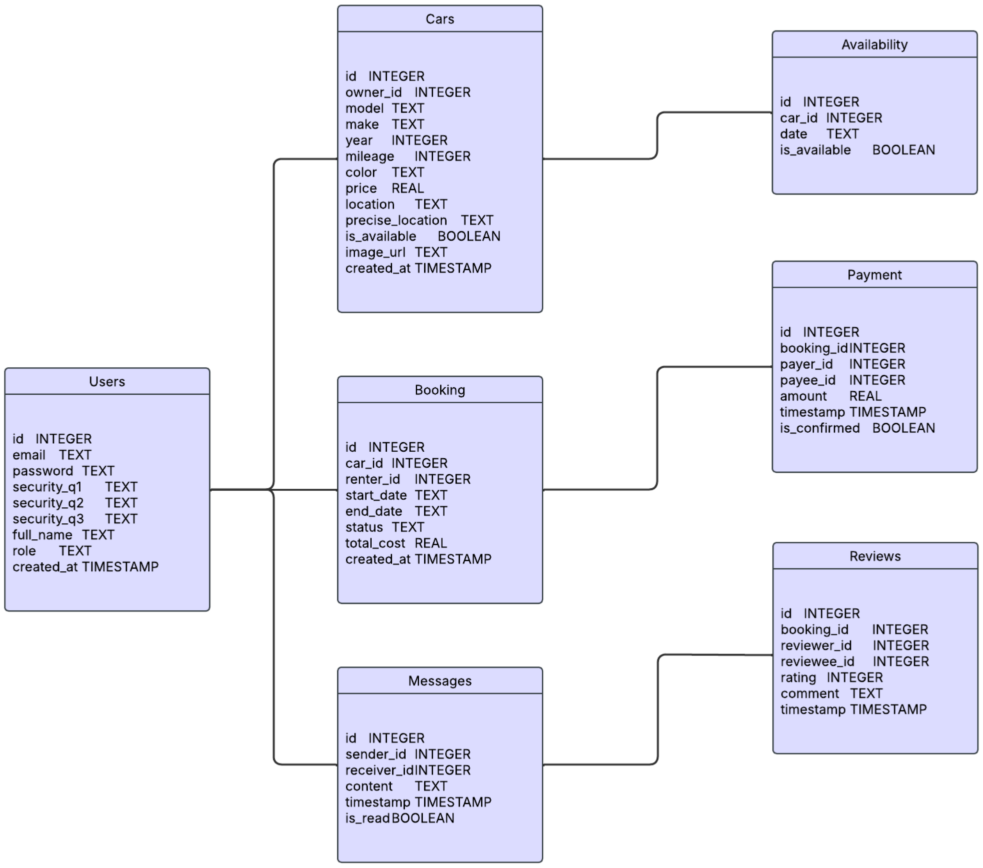
The Chain of Responsibility pattern is used in DriveShare to implement a secure and step-by-step password recovery process. This approach allows multiple security questions to be handled in sequence, where each question acts as a link in the chain. If a user answers a question correctly, the request is passed to the next handler in the chain. If any answer is incorrect, the chain stops, and recovery fails.

The SecurityQuestionHandler interface defines the structure for each handler in the chain, including a handle() method and a setNext() method to establish the next link. Each concrete class, Question1Handler, Question2Handler, and Question3Handler, implements the logic to verify a specific security question.

The PasswordRecoveryManager class acts as the client, initiating the chain through its recoverPassword() method by calling the first handler. This pattern enhances modularity, allows easy extension, and ensures that the password recovery process remains secure and maintainable.



Database Schema


Cars
id INTEGER
owner_id INTEGER
model TEXT
make TEXT
year INTEGER
mileage INTEGER
color TEXT
price REAL
location TEXT
precise_location TEXT
is_available BOOLEAN
image_url TEXT
created_at TIMESTAMP
Availability
id INTEGER
car_id INTEGER
date TEXT
is_available BOOLEAN
Payment
id INTEGER
booking_idINTEGER
payer_id INTEGER
payee_id INTEGER
amount REAL
timestamp TIMESTAMP
is_confirmed BOOLEAN
Users
id INTEGER
email TEXT
password TEXT
security_q1 TEXT
security_q2 TEXT
security_q3 TEXT
full_name TEXT
role TEXT
created_at TIMESTAMP
Booking
id INTEGER
car_id INTEGER
renter_id INTEGER
start_date TEXT
end_date TEXT
status TEXT
total_cost REAL
created_at TIMESTAMP
Reviews
id INTEGER
booking_id INTEGER
reviewer_id INTEGER
reviewee_id INTEGER
rating INTEGER
comment TEXT
timestamp TIMESTAMP
Messages
id INTEGER
sender_id INTEGER
receiver_idINTEGER
content TEXT
timestamp TIMESTAMP
is_readBOOLEAN