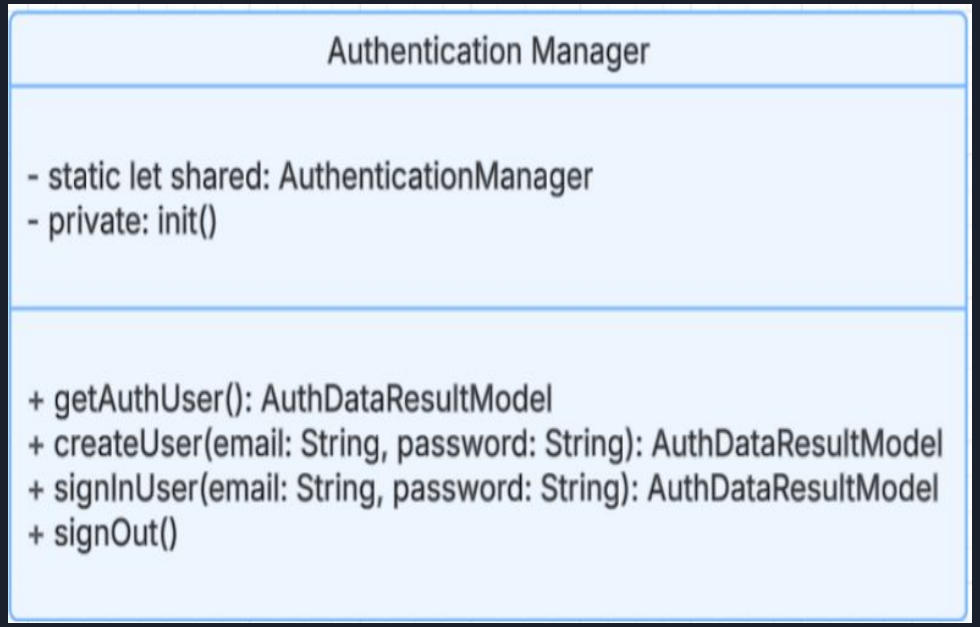
DriveShare

CIS-476

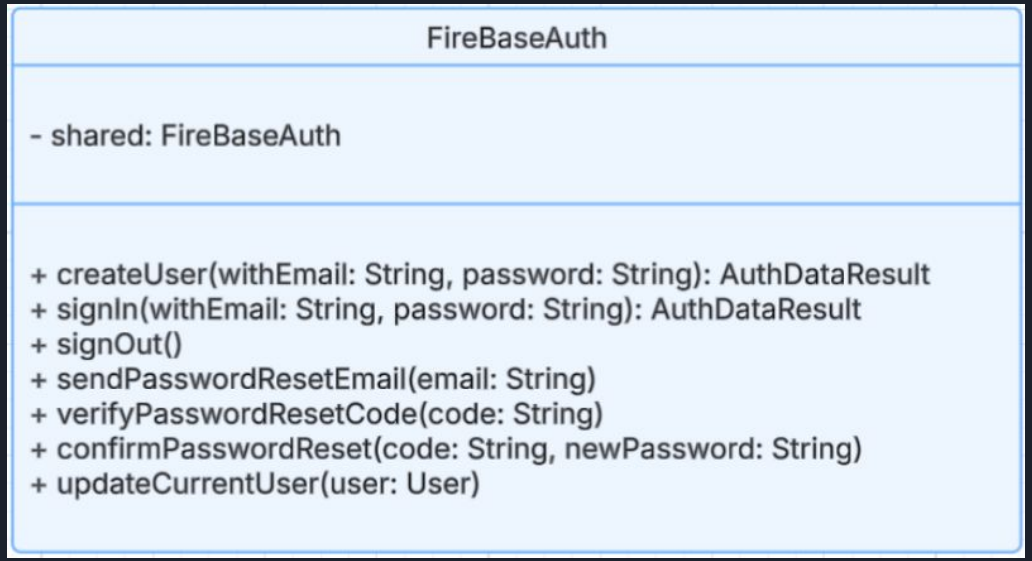
Austin Abro, Donovan McCarthy, Christopher Woods

Singleton Implementation



Description:
The AuthenticationManager class follows the Singleton design pattern, ensuring that only one instance is created and shared throughout the application. This class provides methods for user authentication, including retrieving the current authenticated user, creating an account, signing in, and signing out. The private init() prevents external instantiation, while the shared instance allows global access.

Singleton Implementation Continued



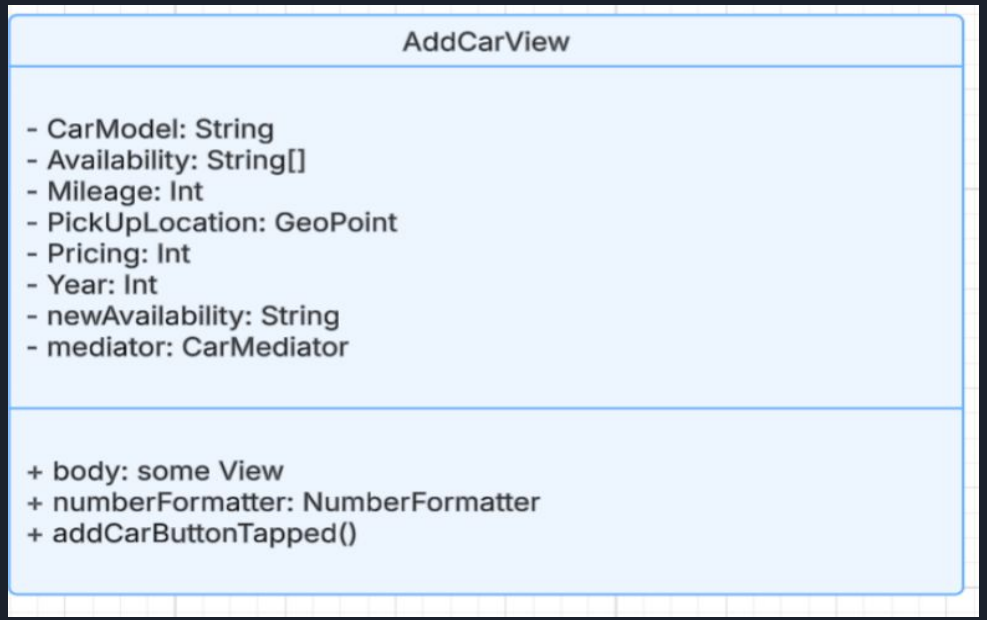
Description:
The FireBaseAuth class provides an abstraction layer for handling user authentication using Firebase. It includes methods for creating a user (createUser), signing in (signIn), and signing out (signOut). Additionally, it supports password management with functionality for sending password reset emails (sendPasswordResetEmail), verifying reset codes (verifyPasswordResetCode), and confirming password resets (confirmPasswordReset). The class also allows for updating the current user information through the updateCurrentUser method. The shared instance ensures a single point of access for these authentication operations.

Observer Implementation

Description:

The BookingViewModel class implements the Observer design pattern, acting as the Subject that manages a list of observers and notifies them when the bookingComplete state changes. It provides methods like addObserver(), removeObserver(), and notifyObservers(), and uses setBookingComplete() to update the state and trigger notifications. The BookingConfirmation class is a ConcreteObserver that listens to changes in bookingComplete via the update() method and updates the UI with booking details when the state is set to true. The Observer protocol defines the update() method, enabling observers like BookingConfirmation to respond to state changes in BookingViewModel.

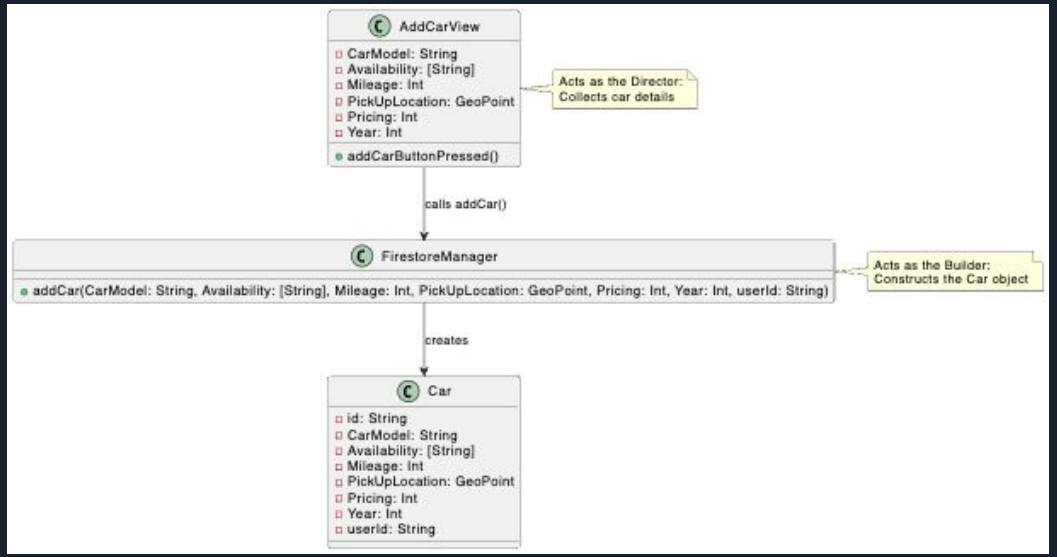
Mediator Implementation



Description:
AddCarView represents the user interface (UI) component that allows the user to input car details in the application. It is a View in SwiftUI that includes form fields for the car model, availability, mileage, pickup location, pricing, and year. This class provides the UI for users to interact with and fill in the necessary data to add a car.

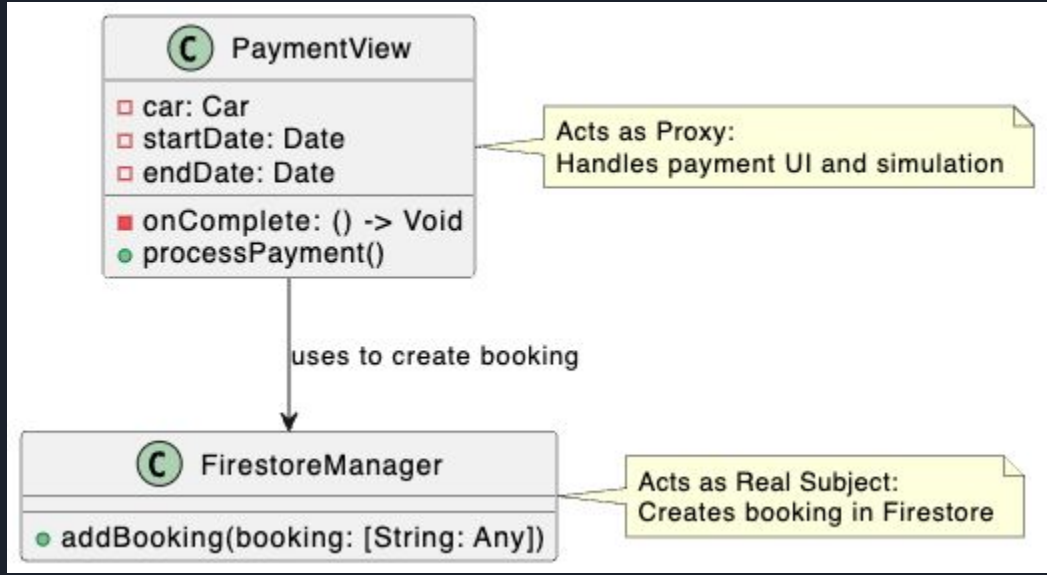
Builder Pattern for Car List Creation

In the DriveShare application, the Builder Pattern is employed to streamline the creation of car listings. The AddCarView assumes the role of the director, systematically gathering critical attributes of a car listing—such as model, availability dates, mileage, pickup location, pricing, and year—via user input fields. Upon the user's command to finalize the addition, these attributes are transmitted to the FirestoreManager, which operates as the builder. The FirestoreManager subsequently constructs the car listing object or generates a corresponding document within the Firestore database. This architectural approach distinctly separates the responsibility of data collection from the process of object construction.



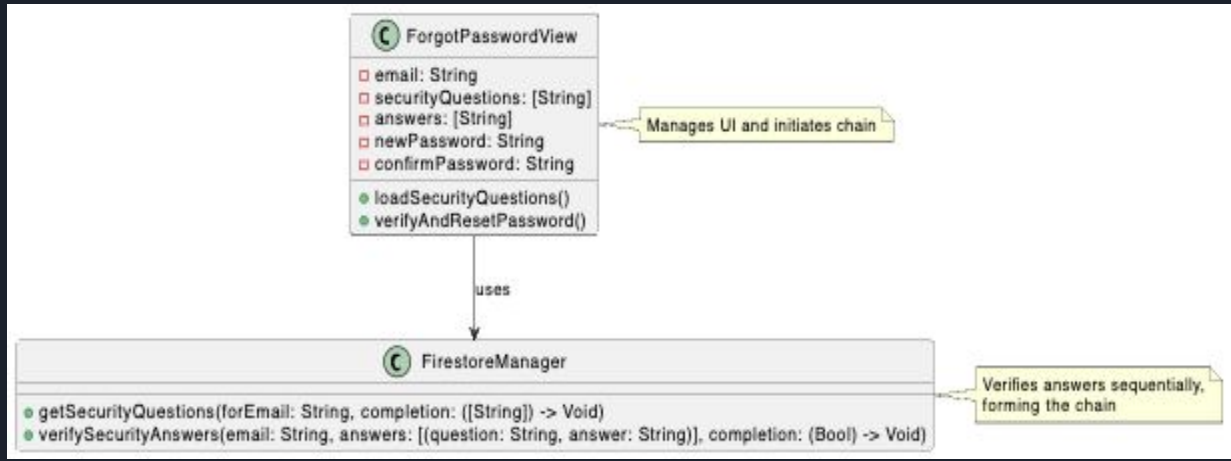
Proxy Pattern for Payment Integration

The Proxy Pattern is applied within the payment integration framework of the DriveShare application. The PaymentView serves as a proxy, overseeing the user interface dedicated to payment processing and simulating the transactional workflow. It is tasked with collecting and validating essential payment details—including card number, cardholder name, expiry date, and CVV—prior to authorizing further action. Upon successful validation, the PaymentView delegates the creation of a booking to the FirestoreManager, which executes the requisite database operations. This abstraction establishes a controlled intermediary layer, ensuring that access to the booking system is mediated through the proxy. Consequently, bookings are instantiated only after the payment simulation has been satisfactorily completed, reinforcing security.



Chain of Responsibility for Password Recovery

The Chain of Responsibility Pattern governs the password recovery mechanism within the DriveShare application. The ForgotPasswordView orchestrates this process, initiating the sequence with email verification to retrieve a set of predefined security questions from the FirestoreManager. The user is subsequently required to provide accurate responses to these questions, with each answer subjected to sequential verification. This establishes a chain-like structure wherein each verification step must be successfully completed to advance to the next, culminating in the authorization of a password reset only upon the correct resolution of all questions.



Proxy Pattern Code

```swift
class PasswordRecoveryProxy {
    private let firestoreManager: FirestoreManager
    private let authManager: AuthenticationManager
    private var recoveryAttempts: [String: Int] = [:]
    private let maxAttempts = 3

    init(firestoreManager: FirestoreManager, authManager: AuthenticationManager) {
        self.firestoreManager = firestoreManager
        self.authManager = authManager
    }

    func initiatePasswordRecovery(email: String, completion: @escaping (Bool, [String]) -> Void) {
            // Check for too many attempts
            if let attempts = self.recoveryAttempts[email], attempts >= self.maxAttempts {
                completion(false, [])
                return
            }
            // Get security questions
            self.firestoreManager.getSecurityQuestions(forEmail: email) { questions in
                completion(!questions.isEmpty, questions)
            }
        }
    func verifyAnswersAndResetPassword(email: String, answers: [(question: String, answer: String)],
                                       newPassword: String, completion: @escaping (Bool, String) -> Void) {
        // Increment attempt counter
        recoveryAttempts[email] = (recoveryAttempts[email] ?? 0) + 1

        firestoreManager.verifySecurityAnswers(email: email, answers: answers) { isVerified in
            guard isVerified else {
                completion(false, "Security answers do not match our records.")
                return
            }

            // Reset password if verified
            self.authManager.resetPassword(email: email, newPassword: newPassword) { success, error in
                if success {
                    // Clear attempt counter on success
                    self.recoveryAttempts.removeValue(forKey: email)
                    completion(true, "Password has been reset successfully.")
                } else {
                    completion(false, error ?? "Failed to reset password.")
                }
            }
        }
    }
}
```

Singleton Pattern Code

```swift
final class AuthenticationManager {
    static let shared = AuthenticationManager()
    private init() {

    }
    func getAuthUser() throws -> AuthDataResultModel{
        guard let user = Auth.auth().currentUser else{
            throw URLError(.badServerResponse)
        }
        return AuthDataResultModel(user: user)
    }

    @discardableResult
    func createUser(email: String, password: String) async throws -> AuthDataResultModel{
        let authResult = try await Auth.auth().createUser(withEmail: email, password: password)
        return AuthDataResultModel(user: authResult.user)
    }

    @discardableResult
    func signInUser(email: String, password: String) async throws -> AuthDataResultModel {
        let authDataResult = try await Auth.auth().signIn(withEmail: email, password: password)
        return AuthDataResultModel(user: authDataResult.user)
    }
    func signOut() throws{
        try Auth.auth().signOut()
    }
    // In AuthenticationManager class
    func resetPassword(email: String, newPassword: String, completion: @escaping (Bool, String?) -> Void) {
        // Use Firebase's password reset functionality directly
        Auth.auth().sendPasswordReset(withEmail: email)
    }

    // Add this new method for security question based reset
    func resetPasswordDirectly(email: String, newPassword: String, completion: @escaping (Bool, String?) -> Vo

        print("Would reset password for \(email) to \(newPassword)")
        completion(true, "Password reset successfully")
    }


}
```

Observer Pattern Code

```
class BookingViewModel: ObservableObject {
    @Published var bookingComplete: Bool = false
    private var observers: [Observer] = []
    func addObserver(observer: Observer) {
        observers.append(observer)
    }
    func removeObserver(observer: Observer) {
        observers.removeAll { $0 === observer }
    }
    func notifyObservers() {
        for observer in observers {
            observer.update()
        }
    }
    func setBookingComplete(value: Bool) {
        bookingComplete = value
        notifyObservers()
    }
}
protocol Observer: AnyObject {
    func update()
    struct BookingConfirmationView: View, Observer {
        @ObservedObject var viewModel: BookingViewModel
        var body: some View {
            VStack {
                if viewModel.bookingComplete {
                    Image(systemName: "checkmark.circle.fill")
                        .resizable()
                        .frame(width: 100, height: 100)
                        .foregroundColor(.green)
                    Text("Booking Confirmed!")
                        .font(.title)
                        .bold()
                    Text("Your car is reserved from \(startDate, style: .date) to \(endDate, style: .date)")
                        .multilineTextAlignment(.center)
                    Text("Booking Reference: \(generateBookingReference())")
                        .font(.headline)
                        .padding()
                        .background(Color.gray.opacity(0.1))
                        .cornerRadius(8)
                    Button("Done") {
                        dismiss()
                    }
                    .frame(maxWidth: .infinity)
                    .padding()
                    .background(Color.blue)
                    .foregroundColor(.white)
                    .cornerRadius(10)
                }
            }
            .onAppear {
                viewModel.addObserver(observer: self)
            }
            .onDisappear {
                viewModel.removeObserver(observer: self)
            }
        }
        func update() {
            // This method gets called whenever
            bookingComplete

            // Handle UI updates based on the new state here.
        }
    }
    private func processPayment() {
        // Validate input
        guard !cardNumber.isEmpty, !cardholderName.isEmpty,
              !expiryDate.isEmpty, !cvv.isEmpty else {
```

```
    private func processPayment() {
        // Validate input
        guard !cardNumber.isEmpty, !cardholderName.isEmpty,
              !expiryDate.isEmpty, !cvv.isEmpty else {
            return
        }
        // Simulate payment processing
        isProcessing = true
        // Fake processing delay
        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            isProcessing = false
            viewModel.setBookingComplete(value: true) // This triggers the Observer to update
            // Send message to car owner about the booking
            if let carOwnerId = car.userId {
                let dateFormatter = DateFormatter()
                dateFormatter.dateStyle =
                    .medium
                let bookingMessage = "I've booked your \(car.CarModel) from
                \(dateFormatter.string(from: startDate)) to \(dateFormatter.string(from: endDate)). Booking
                reference: \(generateBookingReference())"
                firestoreManager.sendMessage(
                    to: carOwnerId,
                    content: bookingMessage,
                    relatedCarId: car.id
                )
            }
        }
    }
}
```