COMSATS UNIVERSITY ISLAMABAD, DHAMTHOR CAMPUS

SOFTWARE DESIGN AND ARCHITECTURE

PROJECT

CAR RENTAL APP

Rehman Ghani

Syed Muhammad Usman

Zain Ul Abideen

Version : 1.0

Lab : 3

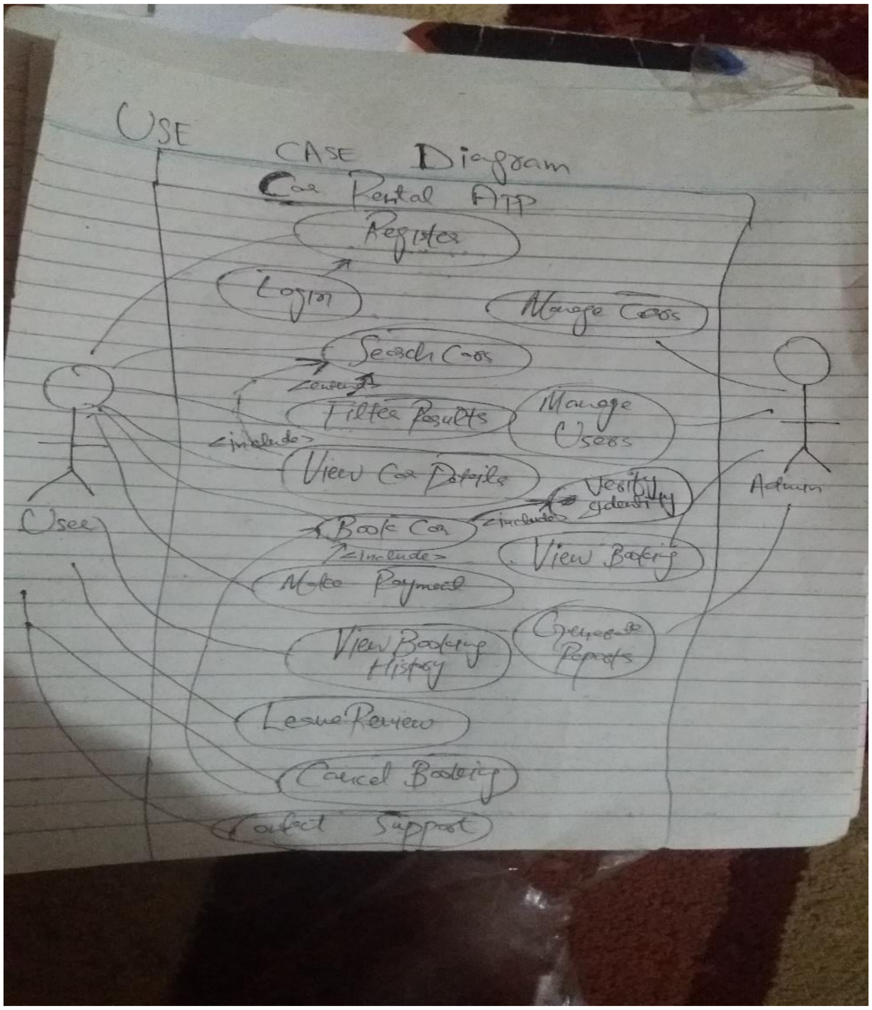
Submitted To: Mukhtiar Zamin

DEPARTEMENT OF SOFTWARE ENGINEERING

CAR RENTAL APP:

A car rental app, which can enhance user experience and streamline the rental process for both customers and operators. Users can create an account and log in using email or phone number. Users can search for available cars based on date, time, and car type. Users can filter results by price range, car brand, type (SUV, sedan, etc.), fuel type, and features. Detailed descriptions of each car, including make, model, year, mileage, and features. Users can select a car, choose rental dates, and reserve the vehicle. Immediate booking confirmation via email or app notification. Options for one-way rentals, hourly, daily, or weekly rates. Support for cash payment and bank transfers. Implementation of secure payment gateways to protect user information. Clear display of rental costs, insurance fees, and taxes. Assistance in case of car breakdowns or accidents. Users can review and sign rental agreements electronically. Admins can add, update, and remove vehicles from the inventory. Users can rate and review their rental experience and vehicles.

USE CASE DIAGRAM:



FULLY DRESSED USE CASE DIAGRAM:

Fully Dressed Use Case: BOOK CAR

Use Case Name: Book Car
Primary Actor: User (Customer)
Secondary Actors: Payment Gateway
Stakeholders and Interests:

- **User:** Wants to book a car for a specified time period.
- **Car Rental Company:** Wants to ensure cars are rented smoothly and payments are collected securely.
- **Administrator:** Wants to track bookings for managing the fleet.

Pre-conditions:

1. The user is logged into the app.
2. The user has searched for and selected a car that is available for rental.
3. The selected car is not already booked for the specified time period.

Post-conditions:

1. The car is successfully reserved for the specified time period.
2. Payment is processed and confirmed.
3. The booking is stored in the system with the user and car details.
4. The user receives a booking confirmation notification (email or app notification).

Main Success Scenario (Basic Flow):

1. The user selects the car they want to book.
2. The system displays the car details, including rental dates, price, and additional fees (if any).
3. The user selects the desired rental start and end dates.
4. The system checks the car's availability for the specified dates.
5. The user reviews the rental details and selects "Book Now."
6. The system displays the payment options.
7. The user chooses a payment method and provides payment details.
8. The system processes the payment through a payment gateway.
9. The payment is confirmed, and the system generates a rental agreement.
10. The system updates the car's availability status to "Booked" for the selected dates.
11. The system sends a booking confirmation notification to the user, including the rental agreement.
12. The use case ends successfully.

Alternative Flows:

Alternative Flow 1: Car Not Available

- 4a. If the car is not available for the specified dates:
    1. The system informs the user that the car is unavailable.
    2. The user can choose different dates or select another car.
    3. The use case continues from step 2.

Alternative Flow 2: Payment Failure

- 8a. If the payment fails:
    1. The system informs the user of the payment failure and prompts them to retry or choose a different payment method.
    2. The user retries the payment process.
    3. The use case continues from step 8 if the payment succeeds. If the payment fails again, the use case ends with a failure notification to the user.

Alternative Flow 3: User Cancels the Booking Process

- 5a. If the user decides to cancel the booking process:
    1. The system cancels the booking operation and returns the user to the car details page.
    2. The use case ends.

Trigger:

- The user decides to rent a car and selects "Book Now" after reviewing car details.

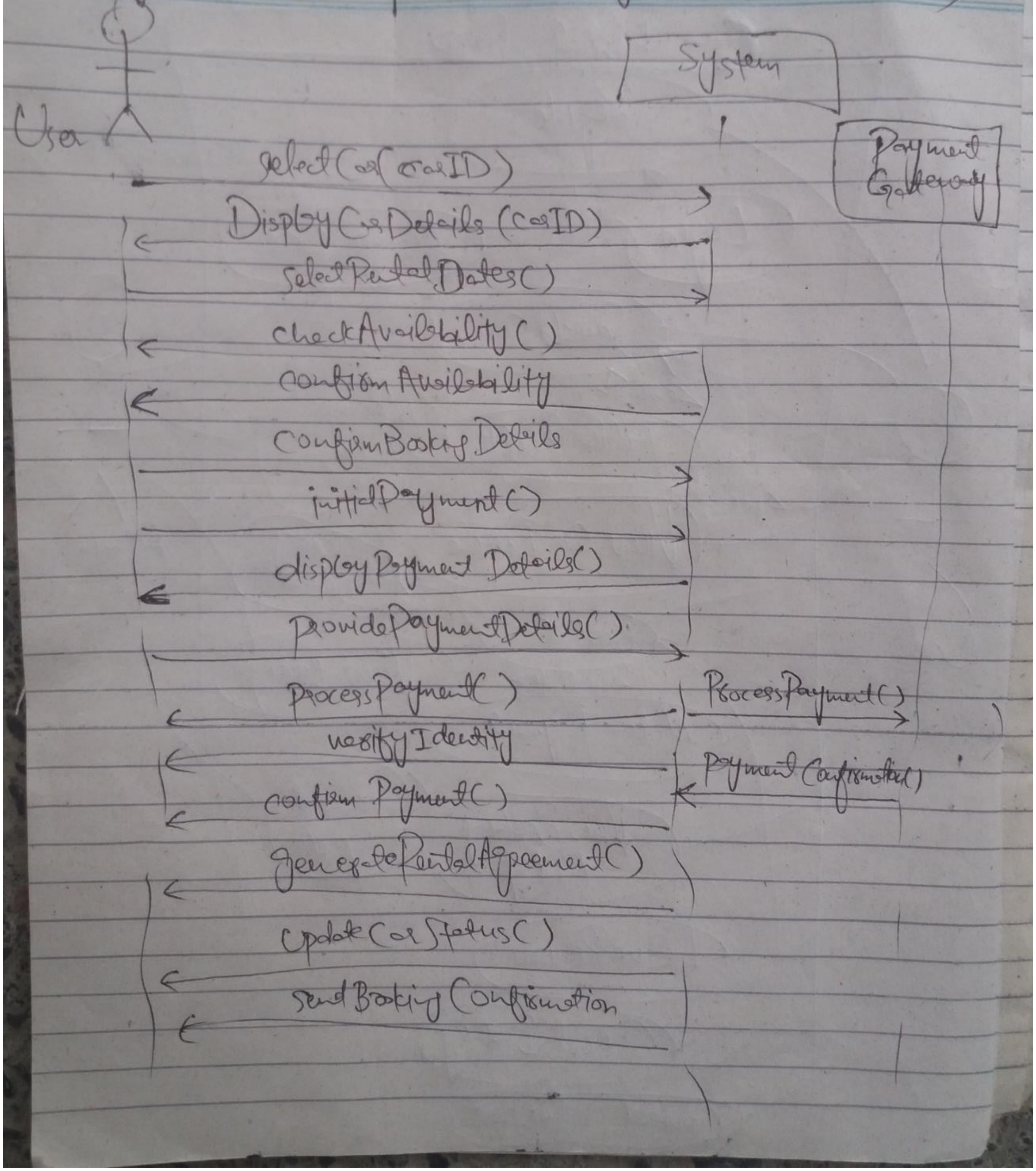
SEQUENCE OF EVENTS FOR "BOOK CAR" USE CASE:

1. **User selects the car to book:**
    - **User:** Chooses a car from the search results or car listings.
    - **System:** Displays the selected car’s details, including rental price, specifications, and availability.
2. **User selects rental start and end dates:**
    - **User:** Inputs the desired rental start and end dates.
    - **System:** Checks the car's availability for the selected time period.
3. **System verifies car availability:**
    - **System:** Confirms whether the car is available for the specified dates.
    - **If available:** The system proceeds to the next step.
    - **If not available:** The system informs the user that the car is unavailable for those dates and suggests alternative dates or cars.
4. **User confirms booking details:**
    - **User:** Reviews the booking details, including dates, total cost, and any additional fees (e.g., insurance).
    - **System:** Displays a "Book Now" button for the user to proceed.
5. **User initiates payment:**
    - **User:** Clicks "Book Now" to proceed with the payment.
    - **System:** Displays payment options, allowing the user to select a payment method.
6. **User provides payment details:**
    - **User:** Inputs payment details (e.g., credit card).
    - **System:** Processes the payment through the payment gateway.

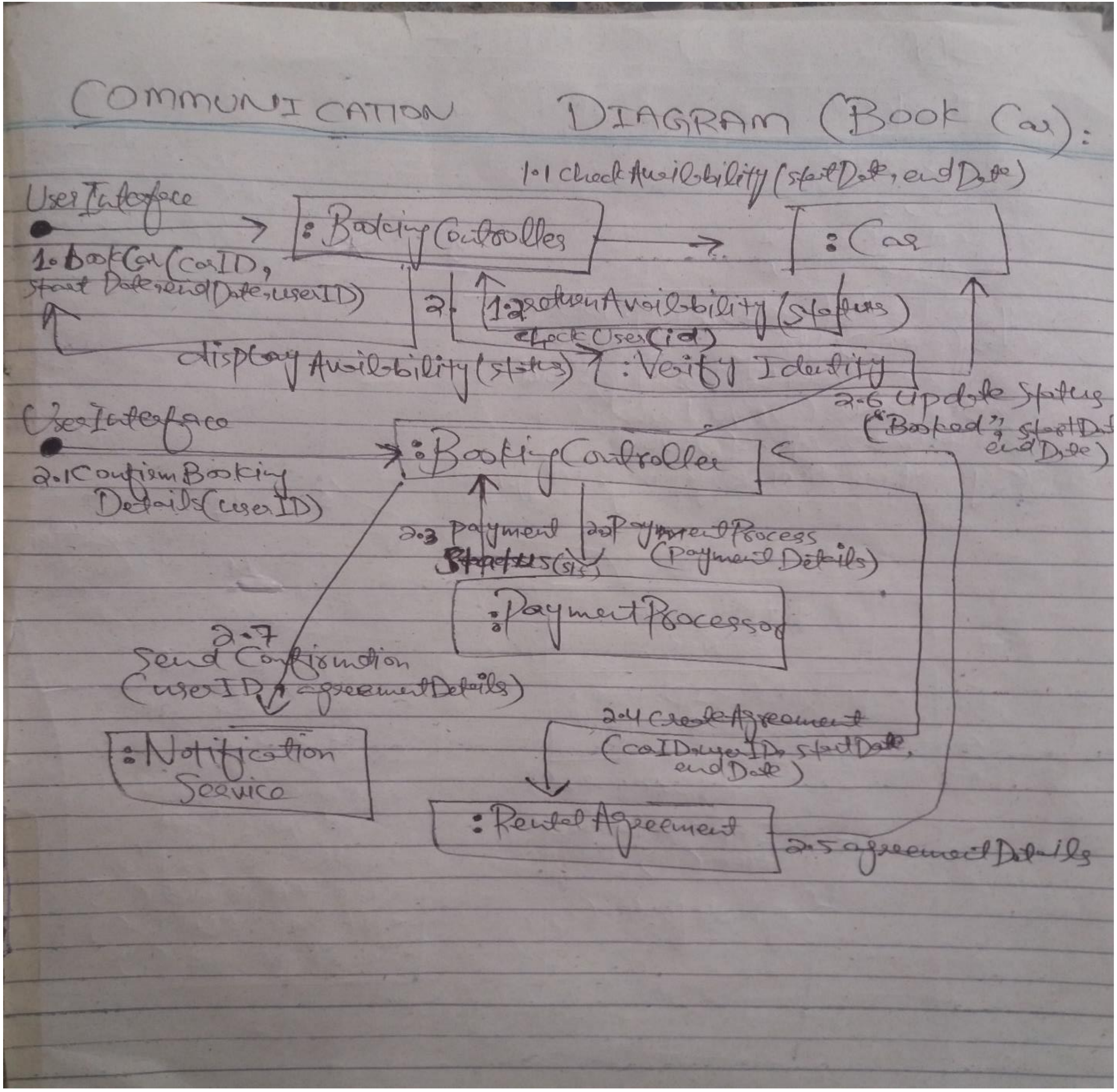
7. **System processes payment:**
    - **System:** Sends payment information to the payment gateway.
    - **If payment is successful:** The system confirms the payment and continues.
    - **If payment fails:** The system notifies the user of the payment failure and allows them to retry or use a different payment method.
8. **System generates rental agreement:**
    - **System:** Creates a rental agreement based on the user’s booking details and updates the car's status to "Booked" for the specified dates.
9. **System sends confirmation:**
    - **System:** Sends a booking confirmation notification to the user via email or in-app notification, including the rental agreement.
10. **User receives confirmation:**
    - **User:** Receives the booking confirmation and can view the booking details in their account.

SYSTEM SEQUENCE DIAGRAM:

System Sequence Diagram (Book Car)



COMMUNICATION DIAGRAM:


COMMUNICATION DIAGRAM (Book Car):
UserInterface
1. bookCar(carID, startDate, endDate, userID)
:BookingController
1.1 checkAvailability(startDate, endDate)
:Car
1.2 returnAvailability(status)
displayAvailability(status)
2.
checkUser(id)
:Verify Identity
UserInterface
2.1 ConfirmBookingDetails(userID)
:BookingController
2.2 PaymentProcess(PaymentDetails)
2.3 paymentStatus(sts)
:PaymentProcessor
2.4 CreateAgreement(carID, userID, startDate, endDate)
:RentalAgreement
2.5 agreementDetails
2.6 Update Status("Booked", startDate, endDate)
2.7 Send Confirmation(userID, agreementDetails)
:Notification Service

A communication diagram (interaction diagram) for the "Book Car" use case, outlining the programming interfaces rather than the implementation. The diagram will follow GRASP (General Responsibility Assignment Software Patterns) principles for assigning responsibilities to various objects. Key GRASP principles used here include:

1. **Controller:** Manages the interactions between the user interface and the domain logic.
2. **Information Expert:** The object that has the necessary information to fulfill a responsibility.
3. **Low Coupling:** Keeping dependencies between objects low.
4. **High Cohesion:** Ensuring that each class is focused on a specific responsibility.

Objects Involved:

1. **UserInterface:** Represents the user interaction layer.
2. **BookingController:** Acts as a controller to manage booking actions.
3. **Car:** Represents the car entity, with information about availability.
4. **PaymentProcessor:** Handles the payment processing.
5. **RentalAgreement:** Manages the creation of the rental agreement.
6. **NotificationService:** Sends notifications to the user.

Communication Diagram Sequence (Textual Representation):

1. **UserInterface → BookingController:** `bookCar(carID, startDate, endDate, userID)`
   - The user initiates the booking process by selecting a car and rental dates.
2. **BookingController → Car:** `checkAvailability(startDate, endDate)`
   - The controller asks the `Car` object to check if it is available for the specified dates.
3. **Car → BookingController:** `returnAvailability(status)`
   - The `Car` object responds with its availability status.
4. **BookingController → UserInterface:** `displayAvailability(status)`
   - The controller notifies the user of the car's availability.
5. **UserInterface → BookingController:** `confirmBookingDetails(userID)`
   - The user confirms the booking details.
6. **BookingController → PaymentProcessor:** `processPayment(paymentDetails)`
   - The controller sends payment details to the `PaymentProcessor` to handle the payment.
7. **PaymentProcessor → BookingController:** `paymentStatus(success/failure)`
   - The `PaymentProcessor` returns the status of the payment.
8. **BookingController → RentalAgreement:** `createAgreement(carID, userID, startDate, endDate)`
   - If the payment is successful, the controller creates a rental agreement.
9. **RentalAgreement → BookingController:** `agreementDetails`
   - The `RentalAgreement` object returns the agreement details.
10. **BookingController → Car:** `updateStatus("Booked", startDate, endDate)`

- The controller updates the status of the car to "Booked" for the selected rental period.

11. **BookingController → NotificationService:** `sendConfirmation(userID, agreementDetails)`

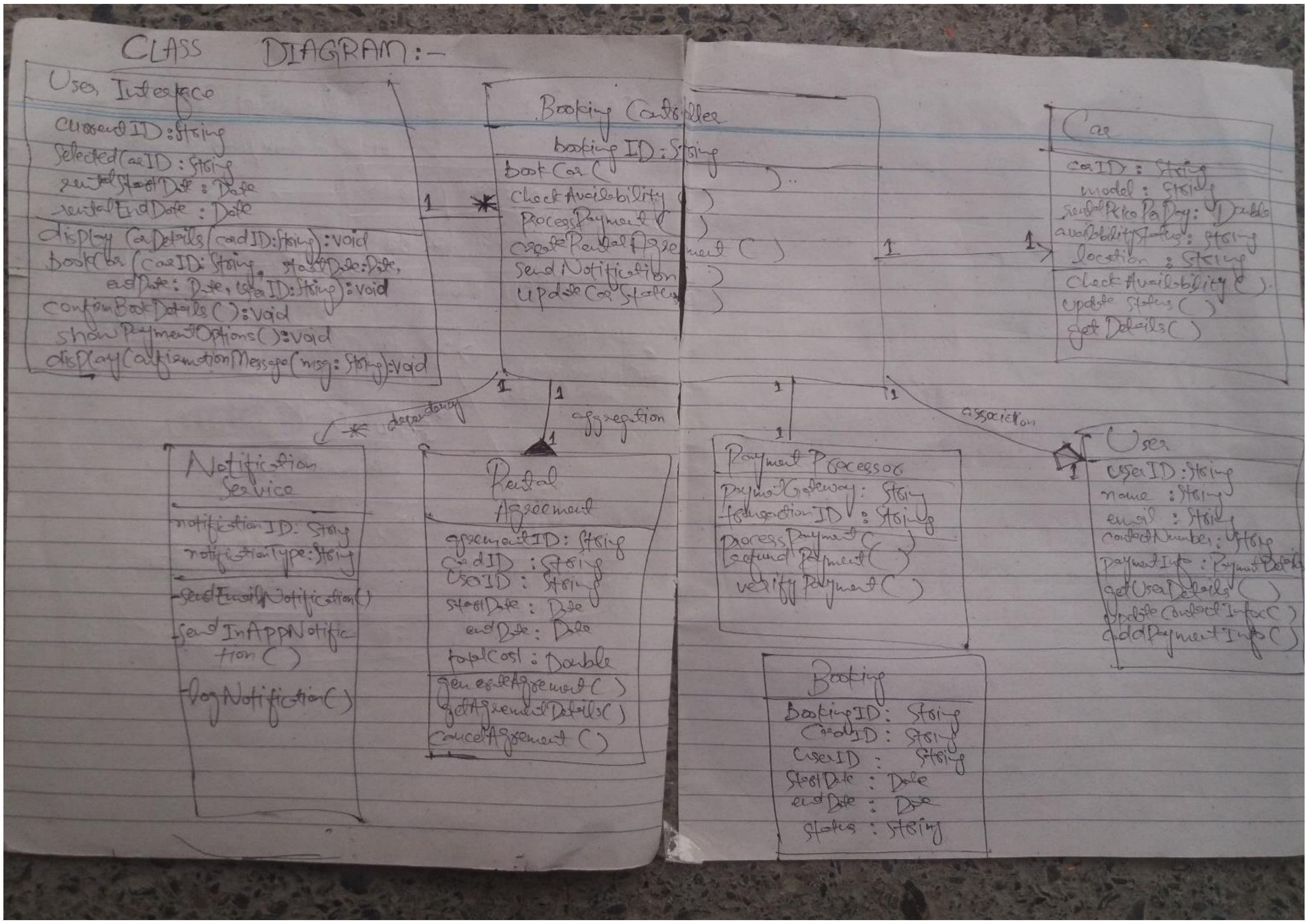
- The controller sends a booking confirmation to the user.

GRASP Principles Application:

1. **Controller (BookingController):** Acts as the mediator between the `UserInterface` and other components like `Car`, `PaymentProcessor`, and `RentalAgreement`. It coordinates the flow.
2. **Information Expert (Car, PaymentProcessor, RentalAgreement):** Each object is responsible for its own data. For instance, `Car` knows about its availability, and `PaymentProcessor` knows how to handle payments.

3. **Low Coupling:** Dependencies are minimized between unrelated objects. For example, the `UserInterface` does not interact directly with the `PaymentProcessor`.
4. **High Cohesion:** Each object has a single responsibility. `Car` only manages availability, `PaymentProcessor` handles payment, and `RentalAgreement` manages the agreement details.

CLASS DIAGRAM:



A comprehensive class diagram for the "Book Car" use case in a car rental application, following GRASP principles. This diagram illustrates the classes involved, their associations, aggregation, dependencies, and other relationships. It incorporates key GRASP principles such as Controller, Information Expert, Low Coupling, and High Cohesion.

Classes and GRASP Principles

1. **UserInterface:** Responsible for user interaction.
2. **BookingController:** The controller coordinating actions between the UI and domain logic.
3. **Car:** Represents a car available for rental.

4. **User:** Represents a customer who rents cars.
5. **PaymentProcessor:** Handles payment transactions.
6. **RentalAgreement:** Represents the booking contract between the user and the rental service.
7. **NotificationService:** Manages notifications sent to the user.
8. **Booking:** Represents the actual booking of a car.

Relationships and Associations

1. **UserInterface - BookingController:**
    - **Association:** The `UserInterface` uses the `BookingController` to initiate the booking process.
    - **Multiplicity:** 1-to-1 relationship (1 `UserInterface` interacts with 1 `BookingController`).
2. **BookingController - Car:**
    - **Association:** The `BookingController` checks the availability of a `Car`.
    - **Dependency:** The `BookingController` depends on the `Car` to confirm availability.
3. **BookingController - User:**
    - **Association:** The `BookingController` accesses the `User` details for booking.
    - **Multiplicity:** 1-to-1 relationship (each booking involves a single user).
4. **BookingController - PaymentProcessor:**
    - **Association:** The `BookingController` initiates payment processing.
    - **Dependency:** The `BookingController` depends on the `PaymentProcessor` for handling payment.
5. **BookingController - RentalAgreement:**
    - **Aggregation:** `BookingController` aggregates `RentalAgreement`, indicating that the agreement is created as part of the booking process but exists independently.
    - **Multiplicity:** 1-to-1 relationship (each booking generates one agreement).
6. **BookingController - NotificationService:**
    - **Dependency:** The `BookingController` uses the `NotificationService` to send confirmation notifications.
7. **Car - Booking:**
    - **Association:** `Booking` is associated with a `Car`, indicating which car has been reserved.
    - **Multiplicity:** 1-to-1 relationship (each booking corresponds to a single car).
8. **User - Booking:**
    - **Aggregation:** A `User` aggregates multiple `Booking` instances, representing different rentals made by the same user.
    - **Multiplicity:** 1-to-many relationship (a user can have multiple bookings).

Relationships Explained

1. **UserInterface → BookingController (association):** `UserInterface` interacts with the controller to start the booking.

2. **BookingController → Car (dependency):** Checks the car's availability for the given dates.
3. **BookingController → User (association):** Uses the user’s data to perform the booking.
4. **BookingController → PaymentProcessor (dependency):** Handles payment operations.
5. **BookingController → RentalAgreement (aggregation):** Manages the rental agreement as part of the booking process.
6. **Car - Booking (association):** Represents the car that is booked for a rental.
7. **User - Booking (aggregation):** A user can have multiple bookings.
8. **BookingController → NotificationService (dependency):** Sends notifications to confirm bookings.

GRASP Principles Applied

1. **Controller (BookingController):** Coordinates actions between objects.
2. **Information Expert (Car, PaymentProcessor, RentalAgreement):** Each class manages its own data, reducing redundancy.
3. **Low Coupling:** Dependencies are minimized between unrelated classes.
4. **High Cohesion:** Classes are focused on specific responsibilities.

1. UserInterface

- **Attributes:**
  - `currentUserID: String`
  - `selectedCarID: String`
  - `rentalStartDate: Date`
  - `rentalEndDate: Date`
- **Methods:**
  - `displayCarDetails(carID: String): void`
  - `bookCar(carID: String, startDate: Date, endDate: Date, userID: String): void`
  - `confirmBookingDetails(): void`
  - `showPaymentOptions(): void`
  - `displayConfirmationMessage(message: String): void`

2. BookingController

- **Attributes:**
  - `bookingID: String`
- **Methods:**
  - `bookCar(carID: String, startDate: Date, endDate: Date, userID: String): void`
  - `checkAvailability(carID: String, startDate: Date, endDate: Date): Boolean`
  - `processPayment(paymentDetails: PaymentDetails): Boolean`
  - `createRentalAgreement(carID: String, userID: String, startDate: Date, endDate: Date): RentalAgreement`
  - `sendNotification(userID: String, message: String): void`
  - `updateCarStatus(carID: String, status: String, startDate: Date, endDate: Date): void`

3. Car

- **Attributes:**
  - `carID: String`
  - `model: String`
  - `rentalPricePerDay: Double`
  - `availabilityStatus: String`
  - `location: String`
- **Methods:**
  - `checkAvailability(startDate: Date, endDate: Date): Boolean`
  - `updateStatus(status: String, startDate: Date, endDate: Date): void`
  - `getDetails(): CarDetails`

4. User

- **Attributes:**
  - `userID: String`
  - `name: String`
  - `email: String`
  - `contactNumber: String`
  - `paymentInfo: PaymentDetails`
- **Methods:**
  - `getUserDetails(userID: String): UserDetails`
  - `updateContactInfo(email: String, contactNumber: String): void`
  - `addPaymentInfo(paymentInfo: PaymentDetails): void`

5. PaymentProcessor

- **Attributes:**
  - `paymentGateway: String`
  - `transactionID: String`
- **Methods:**
  - `processPayment(paymentInfo: PaymentDetails): Boolean`
  - `refundPayment(transactionID: String): Boolean`
  - `verifyPaymentStatus(transactionID: String): Boolean`

6. RentalAgreement

- **Attributes:**
  - `agreementID: String`
  - `carID: String`
  - `userID: String`
  - `startDate: Date`
  - `endDate: Date`
  - `totalCost: Double`
- **Methods:**
  - `generateAgreement(carID: String, userID: String, startDate: Date, endDate: Date): RentalAgreement`
  - `getAgreementDetails(agreementID: String): AgreementDetails`
  - `cancelAgreement(agreementID: String): void`

7. NotificationService

- **Attributes:**
  - `notificationID: String`
  - `notificationType: String`
- **Methods:**
  - `sendEmailNotification(email: String, message: String): void`
  - `sendInAppNotification(userID: String, message: String): void`
  - `logNotification(notificationID: String): void`

8. Booking

- **Attributes:**
  - `bookingID: String`
  - `carID: String`
  - `userID: String`
  - `startDate: Date`
  - `endDate: Date`
  - `status: String`
- **Methods:**
  - `createBooking(carID: String, userID: String, startDate: Date, endDate: Date): Booking`
  - `updateBookingStatus(bookingID: String, status: String): void`
  - `getBookingDetails(bookingID: String): BookingDetails`

Relationships Recap

1. **UserInterface - BookingController:** Uses `BookingController` to initiate booking.
2. **BookingController - Car:** Checks car availability and updates status.
3. **BookingController - User:** Gets user details and uses payment information.
4. **BookingController - PaymentProcessor:** Processes payments.
5. **BookingController - RentalAgreement:** Creates a rental agreement.
6. **BookingController - NotificationService:** Sends notifications to the user.
7. **Car - Booking:** Associates with a specific car.
8. **User - Booking:** A user can have multiple bookings.

GRASP Principles Reflected

- **Controller:** The `BookingController` coordinates all tasks.
- **Information Expert:** Classes like `Car`, `User`, and `PaymentProcessor` hold the data and know how to handle their responsibilities.
- **Low Coupling:** Reducing dependencies by using controller classes to mediate between UI and other classes.
- **High Cohesion:** Each class is focused on a specific responsibility.