ENSF 480: Principles of Software Design

L01: Dr. Ginde

Term Project: Design Document

Flight Flow

Team 5:
Aamna Ghimire (30206981)
Aditi Jain (30208786)
Himla Rahman (30204202)
Miriam Zeresenai(30214522)

Table of Contents

System Overview

The Flight Flow application is a stand-alone software system designed and developed using Object-Oriented principles, Software Design Patterns and a layered architecture learned during the Course ENSF 480. The purpose of this application is to stimulate the core operations of a flight booking environment.

It provides three distinct user roles: Customer, Flight Agent, and System Administrator. Each of these roles are tailored with capabilities such as searching and booking flights for the customers, managing reservations and customer data for flight agents, and administering flight schedules and system information for the admins. Also, the application is integrated with a simple Java-based GUI for usability in the presentations package, a business logic package to handle core domain operations, and a database package connected to a local database supported by SQLite for persistence.
Some of the key functionalities include role based login authentication, flight search and viewing, booking management, customer and payment record maintenance, and administrative control over flight data. The system emphasizes modularity, maintainability and scalability which ensures that the design can be extended to support future enhancements such as online booking or client-server deployment.

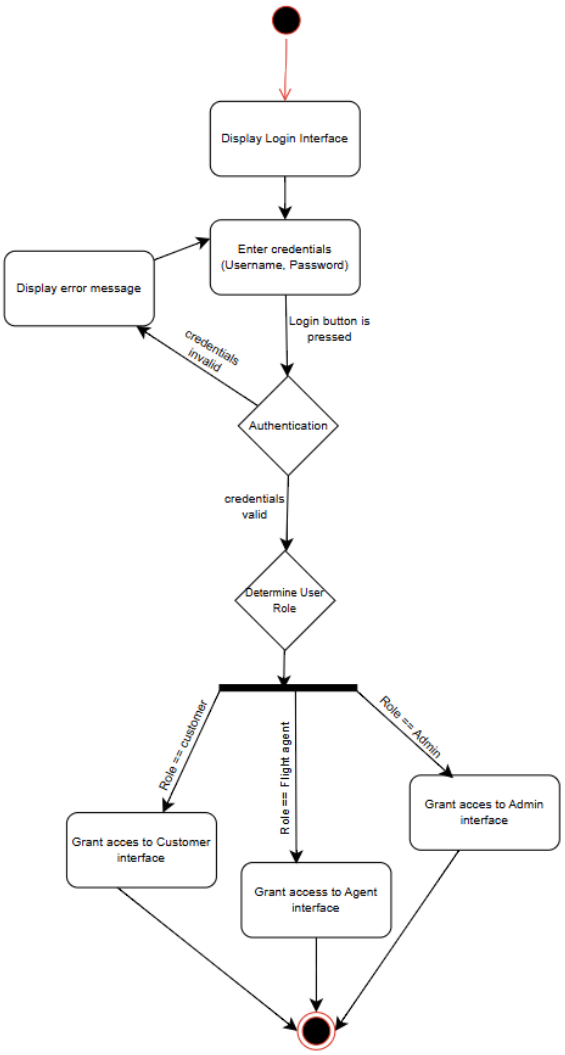
How Does The System Work?

Login

Description

The Login Process begins when the system is initialized, leading to the Display Login Interface activity where the user is prompted to Enter credentials (Username, Password). Once the Login button is pressed, the system initiates the critical Authentication step by checking the submitted credentials against the database. If the authentication fails, the flow branches to display an error message, effectively looping the user back to re-enter their credentials. However, upon successful authentication, the process immediately proceeds to the Determine User Role activity, which is essential for security and separation of concerns. Finally, the process uses the determined role (Customer, Flight Agent, or Admin) to grant access exclusively to the appropriate role-specific Customer Interface, Agent Interface, or Admin Interface. This role-based branching ensures that each user is directed to the appropriate, segregated segment of the system based on their privileges, thus completing the login activity.

Activity Diagram



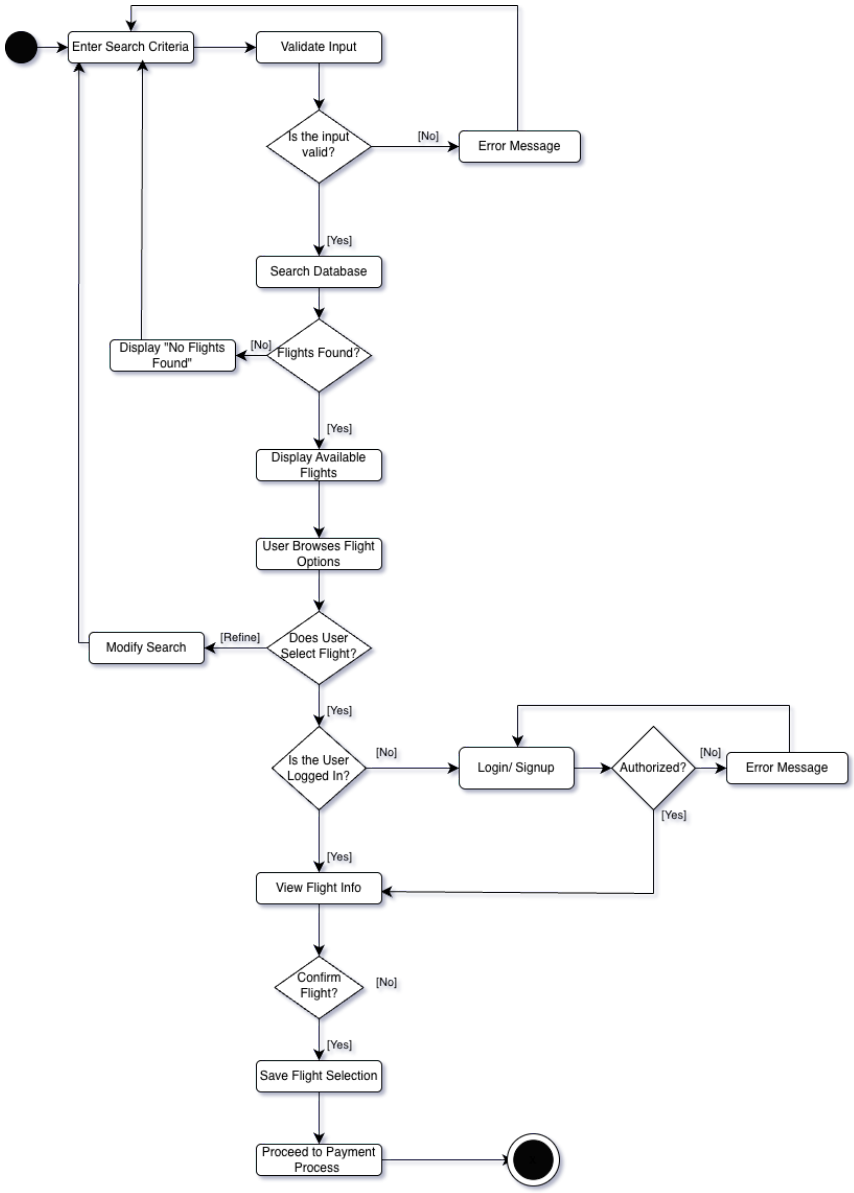
For full detailed view : **Activity Diagram - login **

Browsing/selecting flights

Description

The process of browsing flights and selecting a flight is a fundamental functionality that allows the customers to find and choose a flight for booking. It begins with the user entering search criteria such as the origin city, destination city, etc. The system then validates this input and queries its database to find all matching flights. The results are displayed in a list, showing all key information. The user can browse this list and select a specific flight to view in more detail. Upon selection, the system verifies the user's authentication status. If the user is not logged in, they are guided through a login or registration process. Once authenticated, the user can confirm their flight choice, which saves the selection and transitions them to the booking payment workflow. This process is illustrated in the diagram below.

Activity Diagram



For full detailed view : **Activity Diagram - flight-browse-select **

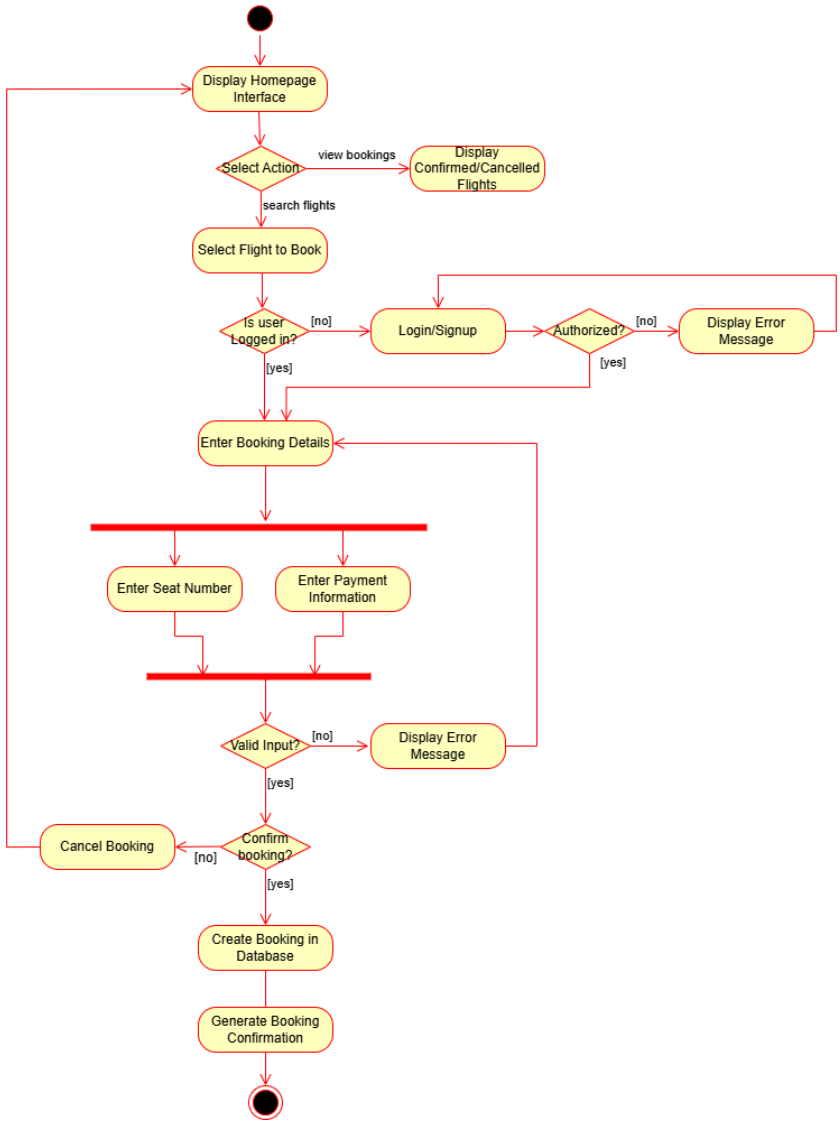
Booking a flight-ticket

Description

The booking management process is a core functionality that enables users to create, cancel, and view flight reservations. The process begins when the user accesses the homepage where the user can select

one of these available actions; search flights and view bookings. If the user chooses to search for flights, they must select a flight to create a new booking. However, the user must login/signup to be able to book the flight. Once login/signup completed, users begin booking their flight by entering their seat number and providing their payment information. The user's input must be valid in order to successfully confirm their booking. If not, they will be prompted to retype their seat number and payment information. Upon confirmation, the system creates a new reservation in the database, generates a booking confirmation, and displays it to the user. For cancelling a reservation, the user must cancel the booking instead of confirming. Upon cancellation, the reservation is removed from the database. If the user chooses to view their bookings, they can do so by going to “My Bookings”, where booking details and confirmation of booking is displayed.Throughout all paths, the system validates user input and displays appropriate error messages if the users input is invalid.

Activity Diagram



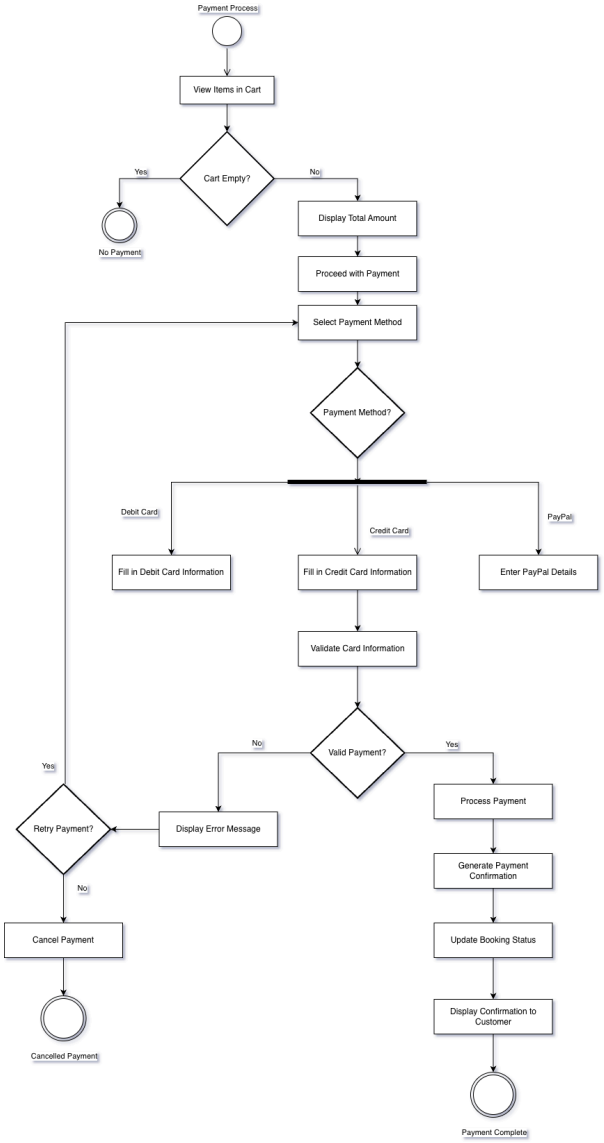
For full detailed view : **Activity Diagram - flight-booking**

Making payment

Description

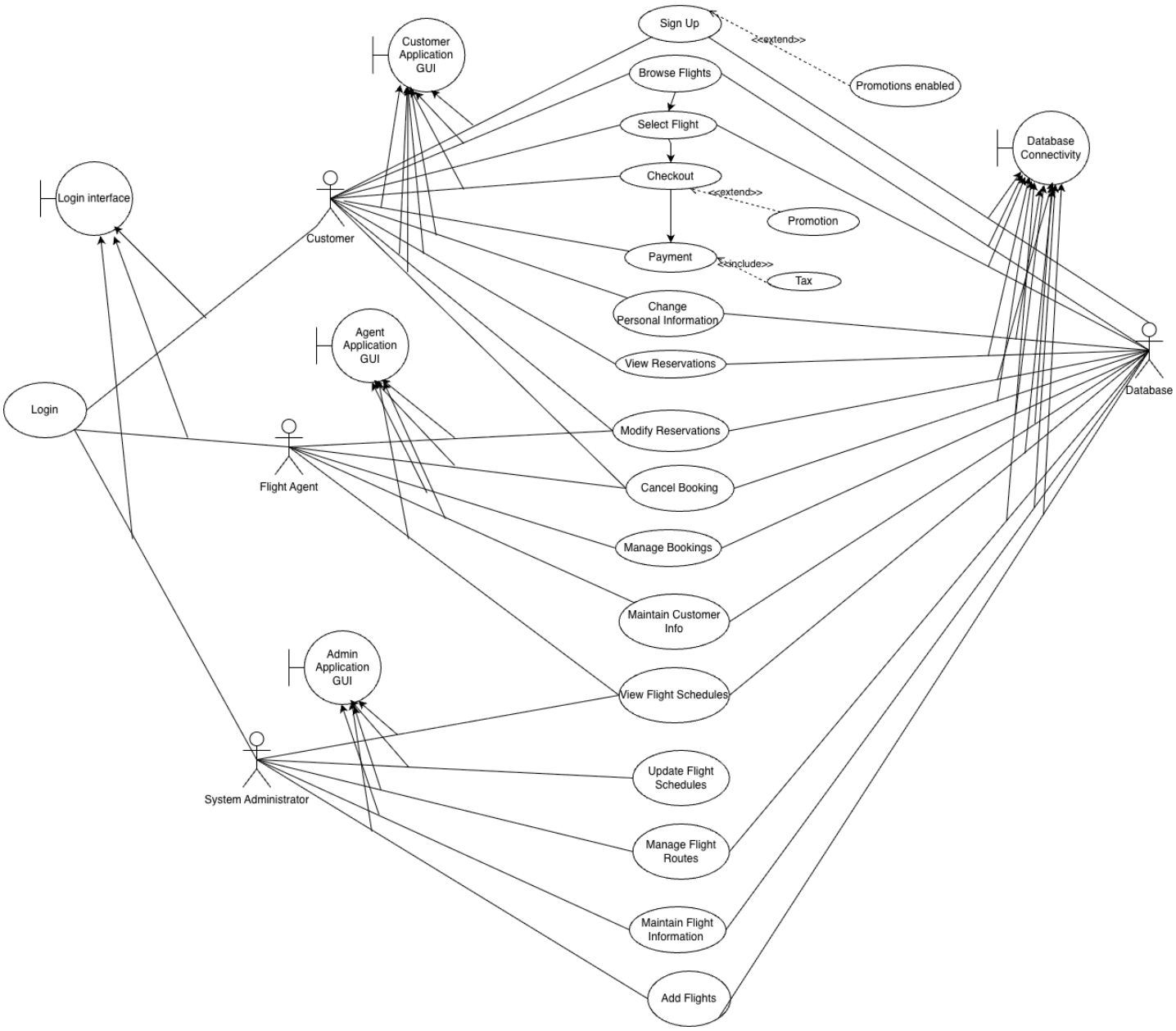
The payment process allows customers to complete their flight booking purchase. When a customer proceeds to checkout, the system displays their selected flight(s) and the total amount due. The customer then selects a payment method (Credit Card, Debit Card, or PayPal) and enters their payment details. The system validates the entered information to ensure it is correct and complete. If validation fails, the customer can retry with corrected information or cancel the transaction. Upon successful validation, the system processes the payment, records the transaction details in the database, updates the booking status to “confirmed”, and displays a payment confirmation to the customer with their booking details.

Activity Diagram



For full detailed view : **Activity Diagram - payment **

System Use Case Diagram



*for full detailed view : **System Use Case Diagram** **

Scenarios for Each Use Cases

Legend

Underlined → objects
Double underlined(underline and bold) → operations

Use case 1: Scenario for Use Case: Browse Flights

Actor: Customer, Agent, Guest
Goal: To search for and view available flights based on search criteria
Pre-condition: The FlightSearchPanel is displayed.
Post-conditions: A list of matching flights is displayed to the user.
Main flow:

1. The User enters search criteria (origin, destination, date) into the FlightSearchPanel.
2. The User initiates the **handleSearch** operation.
3. The FlightSearchPanel sends the criteria to the FlightSearchController.
4. The FlightSearchController calls the **searchFlights** operation on FlightDAO.
5. The FlightDAO retrieves matching Flight entities from DatabaseSQLite.
6. The FlightSearchController returns the list of flights to the FlightSearchPanel.
7. The FlightSearchPanel calls **displayFlights** to show the list of available flights with details.

Alternate flow: If no flights match the criteria, the FlightSearchPanel displays a “No Flights Found” message.

Use case 2: Login

Actor: Customer, Agent, Admin
Goal: To successfully gain authorized access to the system based on their role.
Pre-condition: The LoginPanel is displayed.
Post-conditions: The User is granted access to the role specific MainFrame.
Main flow:

1. The User enters credentials (email, password) into the LoginPanel and initiates the **handleLogin** operation.
2. The LoginPanel sends the credentials to the AuthenticationController via **login(email, password)**.
3. The AuthenticationController calls the **hashPassword** to hash the password.
4. AuthenticationController calls the **authenticateUser(email, password)** operation on UserDAO.
5. The UserDAO retrieves the matching User entity from the **DatabaseSQLite**.
6. The AuthenticationController validates the User and returns the role to the LoginPanel.
7. The LoginPanel creates and displays the appropriate CustomerInterface, AgentInterface or AdminInterface .

Alternate flow: If the UserDAO cannot find a match, the AuthenticationController returns a failed status, and the LoginPanel displays an Error Message.

Use case 3: Sign up

Actor: Customer
Goal: To successfully create a new Customer profile in the system.
Pre-condition: The SignUpPanel is displayed.
Post-conditions: A new Customer entity is created in the DatabaseSQLite with the role Customer..
Main flow:

1. The Customer selects the SignUpPanel and enters the required personal details and new credentials.
2. The Customer initiates the **submitRegistration** operation.
3. The SignUpPanel sends the details to the **AuthenticationController**.
4. The AuthenticationController performs **validateNewUser** (checking for unique username).
5. The AuthenticationController uses the UserFactory to **createUser** and sets the role to Customer.
6. The AuthenticationController calls the **saveUser** operation on the UserDAO.
7. The SignUpPanel confirms the successful creation and redirects to the LoginPanel.

Alternate flow: If the AuthenticationController finds the username is not unique, it displays an Error Message on the SignUpPanel.

Use case 4: Customer Cancel Booking

Use case 4: Customer Cancel Booking
Actor: Customer
Goal: Allow a Customer to cancel an existing Booking.
Pre-condition:

- Customer is logged in and authenticated through AuthenticationController
- Customer is viewing their BookingHistoryPanel
- Valid Bookings exist in the DatabaseSQLite

Post-conditions:

- The selected Booking is marked as “CANCELLED” in the DatabaseSQLite.
- Customer receives a cancellation confirmation message.
-

Main flow:

1. The Customer opens the BookingHistoryPanel from the CustomerDashboard.
2. The BookingHistoryPanel calls **loadBookings** operation.
3. The BookingHistoryPanel sends **getUserBookings** request to BookingController with the Customer’s userId.
4. The BookingController forwards the request to BookingDAO via **getUserBookings**.
5. BookingDAO retrieves all Booking for the Customer from DatabaseSQLite and **returns** them.
6. The BookingHistoryPanel **displays** the Booking list in the JTable.
7. The Customer selects a Booking from the table.
8. The Customer clicks the “Cancel booking” button.
9. The BookingHistoryPanel validates that the Booking is not already “CANCELLED”.
10. The BookingHistoryPanel **displays** a confirmation dialog asking the Customer to confirm cancellation.

11. The Customer confirms the cancellation.
12. The BookingHistoryPanel calls **cancelBooking** operation on BookingController with bookingId.
13. The BookingController retrieves the Booking details via **getBookingbyId** from BookingDAO.
14. The BookingController calls **cancelBooking** on BookingDAO.
15. The BookingDAO updates the Booking status to “CANCELLED” in DatabaseSQLite.
16. The BookingController calls **incrementAvailableSeats** on FlightDAO to return the seat to inventory.
17. The FlightDAO updates the flight's available seats in DatabaseSQLite
18. The BookingController returns success status to BookingHistoryPanel.
19. The BookingHistoryPanel **displays** a success message dialog to the Customer.
20. The BookingHistoryPanel refreshes the Booking list via **loadBookings**.

Alternate flow:

- 4a. If the BookingDAO cannot retrieve bookings due to database error:
    - The BookingController **returns** an empty list.
    - The BookingHistoryPanel **displays** "No bookings found" label.
- 7a. If no booking is selected:
    - The BookingHistoryPanel **displays** a warning message dialog: "Please select a booking to cancel".
- 9a. If the booking status is already "CANCELLED":
    - The BookingHistoryPanel **displays** an information message dialog: "This booking is already cancelled".
    - Flow ends.
- 11a. If the Customer cancels the confirmation dialog:
    - No cancellation is processed.
    - Flow ends.
- 14a. If the BookingDAO fails to update the database:
    - The BookingController **returns** failure status.
    - The BookingHistoryPanel **displays** an error message dialog: "Failed to cancel booking. Please try again."

Use case 5: Checkout

Actor: Customer
Goal: To confirm the selected flight and proceed to payment.
Pre-condition: The Customer has selected a Flight and reached the Checkout page.
Post-conditions: The booking summary is generated and the Customer is directed to the Payment process.
Main flow:

1. The Customer selects “Checkout” from the FlightSearchPanel.
2. The FlightSearchPanel calls **bookFlight** operation.
3. The system creates a new BookingDialog with the select Flight.
4. The BookingDialog calls **getFlightById** on FlightDAO via BookingController to verify the flight's availability.
5. The FlightDAO retrieves the Flight details from DatabaseConnection.
6. The BookingDialog calls **hasAvailableSeats** on the Flight object.

7. The BookingDialog displays the booking summary via **createFlightDetailsPanel** (flight details, price, date).
8. The BookingDialog displays **createSeatSelectionPanel** and **createPaymentPanel** for the Customer to proceed.

Alternate flow: If the Flight is sold out or no longer available, the system displays an error message and returns the Customer to the FlightSearchPanel.

Use case 6: Select flights

Actor: Customer
Goal: To select a specific Flight from the list of available flights.
Pre-condition: The Customer has successfully completed the Browse Flights use case and is viewing the list of results in FlightSearchPanel.
Post-conditions: The selected Flight is stored and the Customer can proceed to Checkout.
Main flow:

1. The Customer reviews the list of available Flight objects displayed in the flightTable.
2. The Customer selects a desired Flight from the displayed options by clicking on a row.
3. The FlightSearchPanel highlights the selected flight and stores it temporarily.
4. The FlightSearchPanel enables the bookButton.
5. The Customer clicks the bookButton to initiate **bookFlight**.
6. The FlightSearchPanel retrieves the selected Flight object from the table model.
7. The Checkout option becomes available via BookingDialog.

Alternate flow: If no flight is selected, the bookButton remains disabled and the system does not allow progression to Checkout.

Use case 7: Manage flights

Actor: Admin
Goal: To add, update, or delete flights within the system.
Pre-condition: The Admin is authenticated and viewing the AdminDashboard.
Post-conditions: The system's flight data is updated in the DatabaseSQLite via DatabaseConnection.
Main flow:

1. The Admin selects "Manage Flights" on the AdminDashboard.
2. The AdminDashboard displays **createManageFlightsPanel** with options to add, edit or delete.
3. The Admin chooses the following:
   Add: Clicks add button, AdminDashboard calls **showAddFlightDialog**
   Edit: Selects a flight, AdminDashboard calls **showEditFlightDialog (flight)**
   Delete: Selects a flight and confirms deletion
4. The AdminDashboard sends the request to the AdminController:
   For add: **addFlight(flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price)**
   For update: **updateFlight(flightId, flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price)**

For delete: **deleteFlight(flightId)**

5. The AdminController interacts with the FlightDAO to apply the changes via corresponding **addFlight**, **updateFlight**, or **deleteFlight** operations.
6. The FlightDAO updates the DatabaseConnection and returns success/failure status.
7. The AdminDashboard calls **getAllFlights** to refresh the display and confirms the operation.

Alternate flow: If the database update fails, the system notifies the Admin and no changes are applied.

Use case 8: Payment

Actor: Customer

Goal: To complete payment for a booking.

Pre-condition:

- The Customer has completed Checkout.
- A booking has been created in DatabaseSQLite with status "CONFIRMED".
- PaymentPanel is **displayed** in a dialog.

Post-conditions:

- The payment is processed and stored in DatabaseSQLite.
- The booking is confirmed.
- A confirmation message is **displayed** to the Customer.

Main Flow:

1. The PaymentPanel **displays** the total amount from the flight price.
2. The Customer selects a payment method from the **paymentMethodCombo**
3. The Customer enters payment details: nameOnCard in nameOnCardField, cardNumber in cardNumberField, cvv in cvvField, and expiry in expiryField.
4. The Customer **clicks** "OK" in the confirmation dialog.
5. The PaymentPanel calls **processPayment** operation.
6. The PaymentPanel validates that nameOnCard is not empty.
7. The PaymentPanel validates that all payment fields are filled.
8. The PaymentPanel calls **validateCardNumber** on PaymentController with the cardNumber.
9. The PaymentController validates that the cardNumber is 16 digits.
10. The PaymentPanel calls **validateCVV** on PaymentController with the cvv.
11. The PaymentController validates that the cvv is 3 digits.
12. The PaymentPanel validates that the expiry format matches the "MM/YY" pattern.
13. The PaymentPanel calls **processPayment** on PaymentController with bookingId, amount, paymentMethod, and cardNumber.
14. The PaymentController calls **maskCardNumber** to store only the last 4 digits.
15. The PaymentController calls **createPayment** on PaymentDAO.
16. The PaymentDAO inserts the payment record into DatabaseSQLite and **returns** paymentId.
17. The PaymentController **returns** the paymentId to PaymentPanel.
18. The PaymentPanel **returns** true to BookingPanel.
19. The BookingPanel creates a success panel with booking confirmation details.
20. The BookingPanel **displays** the success panel in a message dialog: "Booking Confirmed!".
21. The BookingPanel closes the booking dialog.

Alternate Flow:

- 4a. If the Customer clicks "Cancel": -
    - The BookingPanel calls **cancelBooking** on BookingController to remove the temporary booking.
    - Flow ends.
- 6a. If nameOnCard is empty: -
    - The PaymentPanel **displays** an error message dialog: "Please enter name on card".
    - The PaymentPanel **returns** false.
    - Flow ends.
- 7a. If any payment field is empty: -
    - The PaymentPanel **displays** an error message dialog: "Please fill in all payment fields".
    - The PaymentPanel **returns** false.
    - Flow ends.
- 9a. If cardNumber validation fails:
    - The PaymentPanel displays an error message dialog: "Invalid card number. Must be 16 digits."
    - The PaymentPanel **returns** false.
    - Flow ends.
- 11a. If cvv validation fails:
    - The PaymentPanel **displays** an error message dialog: "Invalid CVV. Must be 3 digits."
    - The PaymentPanel **returns** false.
    - Flow ends.
- 12a. If expiry format validation fails:
    - The PaymentPanel **displays** an error message dialog: "Invalid expiry format. Use MM/YY".
    - The PaymentPanel **returns** false.
    - Flow ends.
- 16a. If PaymentDAO fails to insert:
    - The PaymentDAO **returns** -1.
    - The PaymentController **returns** -1.
    - The BookingPanel calls **cancelBooking** to remove the booking.
    - The BookingPanel **displays** an error message dialog: "Payment failed. Booking cancelled."

Use case 9: Cancel booking

Actor: Customer, Agent
Goal: To cancel an existing booking and remove it from the system.
Pre-condition: A valid Booking exists as well as the Customer and Agent is authenticated.
Post-conditions: The Booking is removed, and the system updates the booking record.
Main flow:

1. The Customer/Agent opens the BookingHistory panel.
2. The panel calls **loadBookings** or **getAllBookings** to display the booking list.
3. The Customer/Agent selects a Booking to cancel from the displayed list.
4. The Customer/Agent clicks the cancel button to initiate **cancelBooking** operation.

5. The panel sends the cancellation request to the BookingController via **cancelBooking(bookingId)**.
6. The BookingController calls **cancelBooking(bookingId)** on BookingDAO.
7. The BookingDAO updates the Booking status to "CANCELLED" in the DatabaseConnection.
8. The BookingDAO calls **incrementAvailableSeats(flightId)** on FlightDAO to restore the seat.
9. The system confirms the cancellation and the panel calls **loadBookings** to update the interface.

Alternate flow: If the booking cannot be cancelled due to airline rules, the system displays an error message.

Use case 10: View Booking History

Actor: Customer
Goal: Allow a customer to view their past and upcoming bookings
Pre-conditions: Customer is logged in
Post-conditions: None (read-only operations)
MainFlow:

1. The Customer selects "My Bookings" from the CustomerDashboard.
2. The CustomerDashboard displays the BookingPanel.
3. The BookingPanel calls **initializeUI** to set up the interface.
4. The BookingPanel calls **loadBookings** operation.
5. The BookingPanel requests data from BookingController via **getUserBookings**(userId).
6. The BookingController calls **getUserBookings**(userId) on BookingDAO.
7. The BookingDAO retrieves Booking entities from DatabaseConnection.
8. The BookingDAO calls **extractBookingFromResultSet** for each record.
9. The BookingPanel displays the booking list in the bookingsTable.

Alternate flow: If no bookings exist, the BookingPanel displays "No bookings found" message.

Use case 11: Promotion

Actor: Customer
Goal: To receive and view promotional offers.
Pre-condition: Customer has opted in to receive promotions via receivePromotions attribute
Post-conditions: Promotional message is delivered to the Customer.
Main Flow:

1. During signup, the Customer checks the **receivePromotionsCheckBox** in SignupPanel.
2. The SignupPanel sets **receivePromotions** to true.
3. The SignupPanel calls signup on AuthenticationController with receivePromotions parameter.
4. The AuthenticationController calls **createUser** on UserFactory.
5. The UserFactory creates a Customer object with **receivePromotions** set to true.
6. The UserFactory checks if **receivePromotions** is true.
7. The UserFactory calls subscribe on PromotionManager singleton with the Customer object.
8. The PromotionManager adds the Customer to its subscribers list.

9. The PromotionManager logs: "Subscriber added: [customer email]".
10. When an Admin or system sends a promotion, the PromotionManager calls **notifyAllSubscribers** with a **promotionMessage**.
11. The PromotionManager iterates through all PromotionObservers in the subscribers list.
12. The PromotionManager calls **receivePromotion** on each Customer.
13. The Customer checks if **receivePromotions** is true via **receivePromotions**.
14. The Customer logs the promotion: "Promotion received by [email]: [message]".
15. When the Customer opens the ProfilePanel, it checks if **receivePromotions** is true.
16. If true, the ProfilePanel calls **createPromotionDisplayPanel.**
17. The ProfilePanel creates a promotion panel with a yellow background and promotional text.
18. The ProfilePanel displays current promotions in the promotionDisplayArea JTextArea.
19. The Customer views the promotional offers.

Alternate Flow:

- 1a. If the Customer does not check the **receivePromotionsCheckBox**:
  - **receivePromotions** is set to false.
  - The UserFactory does not call subscribe on PromotionManager.
  - The Customer is not added to the subscribers list.
  - No promotions are displayed in ProfilePanel.
- 13a. If Customer's **receivePromotions** is false:
  - The **receivePromotion** method does nothing.
  - The promotion is not logged.

Use case 12: Modify Reservations

Actor: Customer
Goal: Allow a Customer to update an existing booking (e.g., change seat number).
Pre-condition:

- Customer is logged in.
- Customer has at least one active booking (status = "CONFIRMED").
- BookingHistoryPanel is displayed.

Post-conditions:

- Booking updates are stored in DatabaseSQLite.
- Updated Booking details are **displayed**.

Main Flow:

1. The Customer opens the BookingHistoryPanel from CustomerDashboard.
2. The BookingHistoryPanel loads and **displays** the Customer's **bookings** in the JTable.
3. The Customer selects a booking from the table.
4. The Customer clicks a "Modify Booking" **button**
5. The BookingHistoryPanel validates that a booking is selected.
6. The BookingHistoryPanel retrieves the bookingId from the selected row.
7. The BookingHistoryPanel calls **getBookingById** on BookingController.
8. The BookingController forwards the request to BookingDAO.
9. The BookingDAO retrieves the Booking object from DatabaseSQLite.
10. The BookingController validates that the booking status is "CONFIRMED".

11. The BookingHistoryPanel **displays** a modification dialog with current booking details.
12. The Customer modifies the seat number in a text field.
13. The Customer clicks "Save Changes" button.
14. The BookingHistoryPanel validates the new seat number input.
15. The BookingHistoryPanel calls **modifyBooking** on BookingController with bookingId and newSeatNumber.
16. The BookingController retrieves the booking via **getBookingById** from BookingDAO.
17. The BookingController validates that the booking exists and status is "CONFIRMED".
18. The BookingController updates the booking's seatNumber using **setSeatNumber**.
19. The BookingController would call **updateBooking** on BookingDAO (Note: this method needs to be implemented).
20. The BookingDAO would update the booking record in DatabaseSQLite.
21. The BookingController returns success status to BookingHistoryPanel.
22. The BookingHistoryPanel displays a success message dialog: "Booking modified successfully!".
23. The BookingHistoryPanel refreshes the booking list via **loadBookings.**

Alternate Flow:

- 5a. If no booking is selected:
  - The BookingHistoryPanel displays a warning message dialog: "Please select a booking to modify".
  - Flow ends.
- 10a. If the booking status is "CANCELLED":
  - The BookingController returns an error status.
  - The BookingHistoryPanel displays an error message dialog: "Cannot modify a cancelled booking".
  - Flow ends.
- 14a. If the new seat number is empty or invalid:
  - The BookingHistoryPanel displays an error message dialog: "Please enter a valid seat number".
  - Flow returns to step 12. -
- 17a. If the booking is not found or status changed:
  - The BookingController returns false.
  - The BookingHistoryPanel displays an error message dialog: "Booking not found or cannot be modified".

Use case 13: Change Personal Information

Actor: Customer
Goal: To update stored Customer profile details.
Pre-condition:

- Customer is logged in and authenticated.
- CustomerDashboard is displayed.

Post-conditions:

- Updated personal information is saved in DatabaseSQLite.
- Customer object is updated with new values.

Main Flow:

1. The Customer clicks the "Profile" button in the CustomerDashboard navigation panel.
2. The CustomerDashboard switches the CardLayout to display the ProfilePanel.
3. The ProfilePanel retrieves current Customer data and populates the form fields.
4. The ProfilePanel **displays** email in emailField (read-only, with gray background).
5. The ProfilePanel **displays** firstName in firstNameField.
6. The ProfilePanel **displays** lastName in lastNameField.
7. The ProfilePanel **displays** address in addressField.
8. The ProfilePanel **displays** phone in phoneField.
9. The ProfilePanel **displays** receivePromotions status in promotionsCheckbox.
10. The Customer modifies their firstName, lastName, address, or phone in the respective fields.
11. The Customer clicks "Update Profile" button.
12. The ProfilePanel calls **updateProfile** operation.
13. The ProfilePanel retrieves the values from firstNameField, lastNameField, addressField, and phoneField using **getText** and trim.
14. The ProfilePanel validates that firstName and lastName are not empty.
15. The ProfilePanel updates the Customer object using **setFirstName, setLastName, setAddress,** and **setPhone.**
16. The ProfilePanel would call **updateUserInfo** on UserDAO
17. The UserDAO would **execute** an UPDATE SQL statement on DatabaseSQLite.
18. The UserDAO would **return** true if successful.
19. The ProfilePanel displays a success message dialog: "Profile updated successfully!".

Alternate Flow:

- 14a. If firstName or lastName is empty:
  - The ProfilePanel displays an error message dialog: "First name and last name are required".
  - Flow returns to step 10.
- 18a. If UserDAO fails to update the database:
  - The UserDAO returns false.
  - The ProfilePanel displays an error message dialog: "Failed to update profile. Please try again."

Use case 14: Manage Bookings

Actor: Flight Agent
Goal: Manage customer bookings on their behalf.
Pre-condition: Agent is authenticated and viewing AgentDashboard.
Post-conditions: Booking records are updated in DatabaseConnection.
Main flow:

1. The Agent selects "Manage Bookings" from the AgentDashboard.
2. The AgentDashboard calls **createViewAllBookingsPanel** to display all bookings.
3. The AgentDashboard calls **getAllBookings** on AgentController.
4. The AgentController calls **getAllBookings** on BookingDAO.
5. The BookingDAO retrieves all Booking entities from DatabaseConnection.

6. The Agent views customer bookings in the displayed table.
7. The Agent selects a booking and chooses to modify or cancel it.
8. For cancellation, the AgentDashboard calls **cancelCustomerBooking(bookingId)** on AgentController.
9. The AgentController calls **cancelBooking(bookingId)** on BookingDAO.
10. The BookingDAO updates the Booking status in DatabaseConnection.
11. The panel refreshes to show updated data.

Alternate flow: If the operation is invalid or fails, the system displays an error message.

Use case 15: Maintain Customer Info

Actor: Flight Agent
Goal: To update customer information.
Pre-condition: Agent is authenticated and viewing AgentDashboard
Post-conditions: Customer information is updated in DatabaseConnection.
Main flow:

1. The Agent selects "Manage Customers" from the AgentDashboard.
2. The AgentDashboard calls **createManageCustomersPanel**.
3. The Agent enters a customer email in the search field.
4. The AgentDashboard calls **searchCustomerByEmail**(email) on AgentController.
5. The AgentController calls **getUserByEmail**(email) on UserDAO.
6. The UserDAO retrieves the User record from DatabaseConnection.
7. The Agent selects the Customer profile from results.
8. The AgentDashboard calls **showEditCustomerDialog**(customer) to display editable fields.
9. The Agent edits the customer details and confirms.
10. The dialog calls **updateUserProfile**(userId, firstName, lastName, address, phone) on UserDAO.
11. The UserDAO updates the User record in DatabaseConnection.
12. The system displays a success confirmation.

Alternate flow: If the information is invalid or the customer is not found, an error message is displayed.

Use case 16: Update Schedules

Actor: System Administrator
Goal: To update flight schedules.
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: Flight schedule changes are stored and viewing in DatabaseConnection.
Main flow:

1. The Admin selects "Manage Flights" from the AdminDashboard.
2. The AdminDashboard calls **createManageFlightsPanel** to display current flights.

3. The AdminDashboard calls **getAllFlights** on AdminController.
4. The AdminController calls **getAllFlights** on FlightDAO.
5. The Admin selects a Flight to update schedule.
6. The AdminDashboard calls **showEditFlightDialog(flight)**.
7. The Admin updates the departureTime and/or arrivalTime fields.
8. The Admin confirms the changes.
9. The dialog calls **updateFlight(flightId, flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price)** on AdminController.
10. The AdminController calls **updateFlight** on FlightDAO.
11. The FlightDAO updates the Flight record in DatabaseConnection.
12. The system confirms the update and refreshes the display.

Alternate flow:If the schedule update is invalid (e.g., departure after arrival), an error is displayed.

Use case 17: Manage Flight Routes

Actor: System Administrator
Goal: Maintain and update flight routes (origin and destination cities).
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: Routes are updated in DatabaseConnection.
Main flow:

1. The Admin selects "Manage Flights" from the AdminDashboard.
2. The AdminDashboard displays the flight management interface via **createManageFlightsPanel**.
3. The Admin selects a Flight to modify its route.
4. The AdminDashboard calls **showEditFlightDialog(flight).**
5. The Admin updates the origin and/or destination fields.
6. The Admin confirms the changes.
7. The dialog calls **updateFlight(flightId, flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price)** on AdminController.
8. The AdminController calls **updateFlight** on FlightDAO.
9. The FlightDAO updates the Flight record in DatabaseConnection.
10. The system confirms the route change and refreshes the display.

Alternate flow: If the route update is invalid or database error occurs, an error message is displayed.

Use case 18: Maintain Flight Information

Actor: System Administrator
Goal: Update core flight information such as capacity, aircraft assignment, or pricing.
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: Updated flight details are stored in DatabaseConnection.

Main flow:

1. The Admin selects "Manage Flights" from AdminDashboard.

2. The AdminDashboard calls **getAllFlights** via AdminController to display flights.
3. The Admin selects a Flight to update.
4. The AdminDashboard calls **showEditFlightDialog**(flight).
5. The Admin updates flight details (totalSeats, price, aircraft type, etc.).
6. The Admin confirms changes.
7. The dialog calls **updateFlight** on AdminController.
8. The AdminController calls **updateFlight** on FlightDAO.
9. The FlightDAOupdates the Flight in DatabaseConnection.
10. The system confirms and refreshes the display.

Alternate flow: If validation fails, an error

Use case 19: Add Flights

Actor: System Administrator
Goal: Add new flights to the system.
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: New Flight record is stored in DatabaseConnection.

Main flow:

1. The Admin selects "Add Flight" from the AdminDashboard.
2. The AdminDashboard calls **showAddFlightDialog**.
3. The Admin enters new flight details (flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price) in the dialog fields.
4. The Admin confirms the addition.
5. The dialog calls **addFlight**(flightNumber, departureCity, arrivalCity, departureTime, arrivalTime, totalSeats, price) on AdminController.
6. The AdminController calls **addFlight**on FlightDAO.
7. The FlightDAO inserts the new Flight record into DatabaseConnection and returns the flightId.
8. The AdminDashboard calls **getAllFlights** to refresh the display.
9. The system displays a success confirmation.

Alternate flow: If the flight details are invalid or flightNumber already exists, an error is displayed and the flight is not added.

Use case 20: Remove Flights

Actor: Admin
Goal: Remove flights from the schedule.
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: Flight is removed from the DatabaseConnection.

Main flow:

1. The Admin selects "Manage Flights" from AdminDashboard.
2. The AdminDashboard displays flights via **getAllFlights**.
3. The Admin selects a Flight to delete.
4. The Admin clicks the delete button to initiate deletion.
5. The system displays a confirmation dialog.
6. The Admin confirms the deletion.
7. The AdminDashboard calls **deleteFlight**(flightId) on AdminController.
8. The AdminController calls **deleteFlight**(flightId) on FlightDAO.
9. The FlightDAO removes the Flight from DatabaseConnection.
10. The AdminDashboard refreshes the display via **getAllFlights.**.

Alternate flow: If the Flight has existing Booking records, the system blocks deletion and displays an error message stating that flights with bookings cannot be deleted.

Use case 21: Manage Aircraft

Actor: Admin
Goal: Maintain aircraft information used in flights.
Pre-condition: Admin is authenticated and viewing AdminDashboard.
Post-conditions: Aircraft information is updated in DatabaseConnection.

Main flow:

1. The Admin selects "Manage Aircraft" from AdminDashboard.
2. The system displays current aircraft records.
3. The Admin chooses to add, edit, or remove an aircraft record.
4. For adding: The Admin enters aircraft details (type, capacity, etc.).
5. For editing: The Admin selects an aircraft and modifies details.
6. For removing: The Admin selects an aircraft and confirms deletion.
7. The AdminDashboard sends the request to AdminController.
8. The AdminController interacts with the appropriate DAO to update aircraft data in DatabaseConnection.
9. The system confirms the operation and refreshes the display.

Alternate flow: If an aircraft is assigned to active flights, deletion is blocked and an error is displayed.

Use case 22: Database Connectivity - Database

Actor: Database (external entity)
Goal: Provide all system components with database access.
Pre-condition: DatabaseConnection is active via **getInstance** operation.
Post-conditions: Components successfully read/write data.

Main flow:

1. Any system component (DAO) requests database access.
2. The DAO calls **getInstance** on DatabaseConnection (Singleton pattern).
3. The DatabaseConnection returns its connection object via **getConnection**.
4. The DAO executes SQL queries using the connection.
5. The DatabaseConnection returns query results to the DAO.
6. The DAO processes results and returns data to the calling controller.

Alternate flow: If **getConnection** fails (database not available), the system displays a database connectivity error and operations are blocked.

Use case 23: Database Connectivity - Customer

Actor: Customer
Goal: Access the graphical interface to interact with system features.
Pre-condition: GUI is launched via MainFrame.
Post-conditions: Customer interacts with system functionalities.

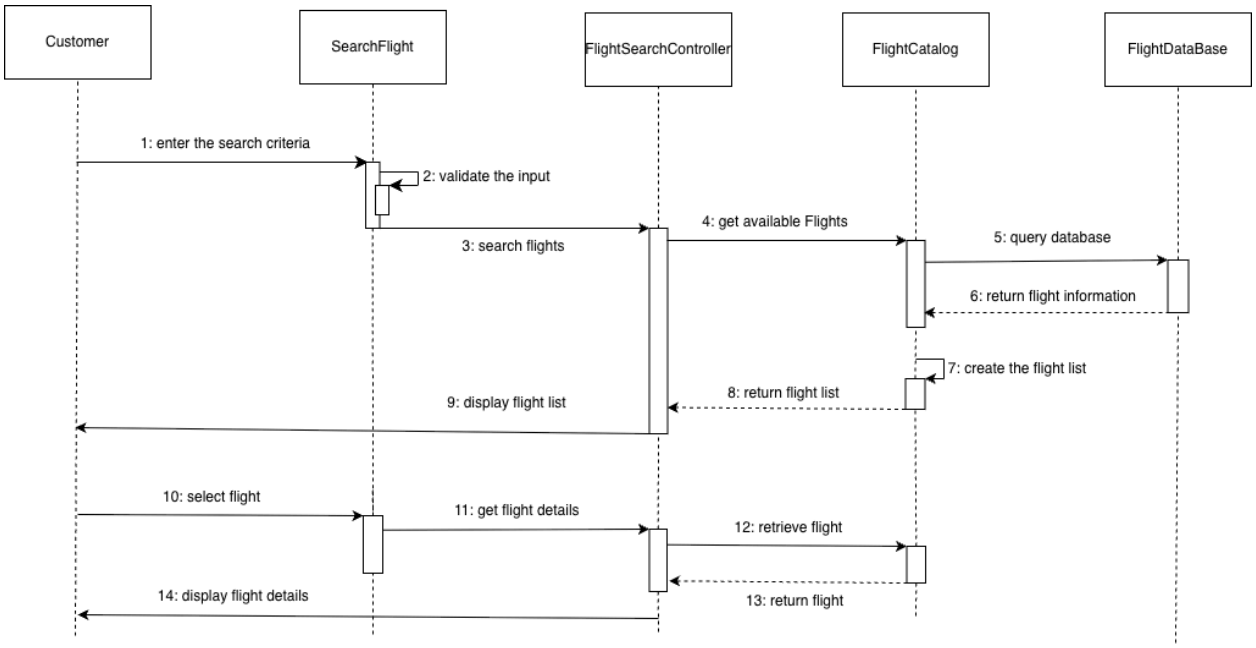
Main flow:

1. The Customer launches the application by running MainFrame.
2. The MainFrame calls **main**(args) operation which initializes the system.
3. The system calls **initializeDatabase** on DatabaseConnection to set up tables.
4. The MainFrame displays the HomePage.
5. The HomePage calls **initializeUI** to display role selection options.
6. The Customer selects their role or guest mode.
7. The HomePage calls **openAuthFrame**(role) or **openGuestMode**.
8. The appropriate interface (AuthFrame, GuestDashboard, or dashboard) is displayed.
9. The Customer interacts with available features through boundary objects.

Alternate flow: If the database initialization fails, an error dialog is displayed and the application exits.

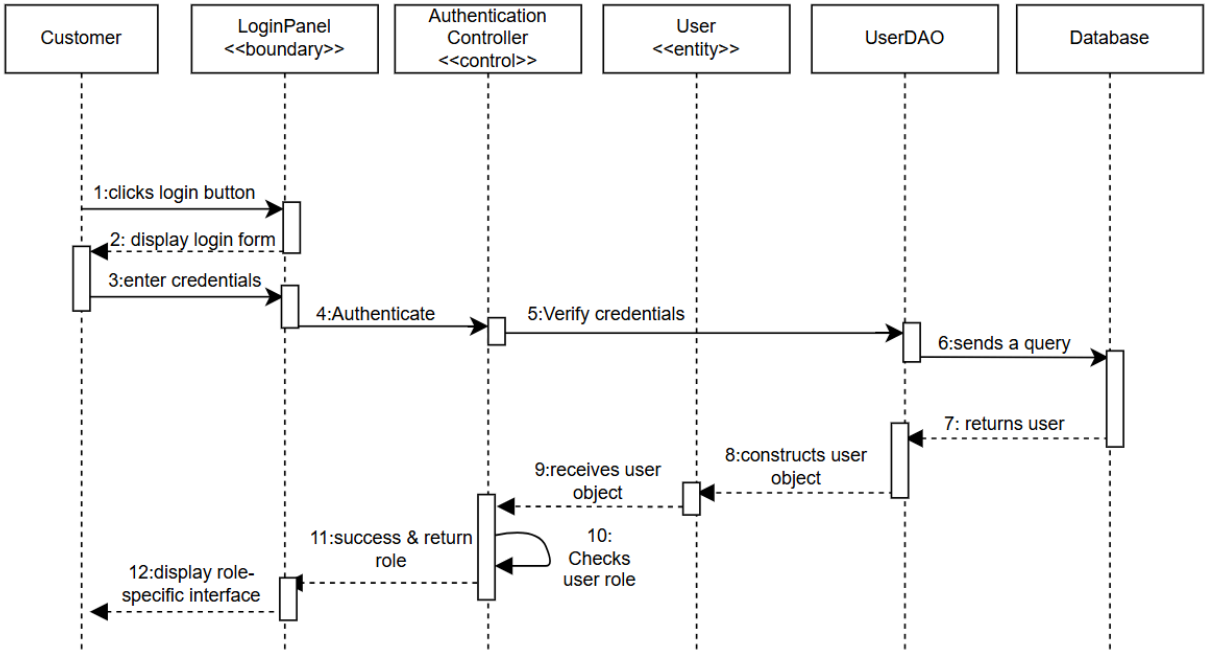
Interaction/Sequence Diagrams

Major Use case 1: Browsing



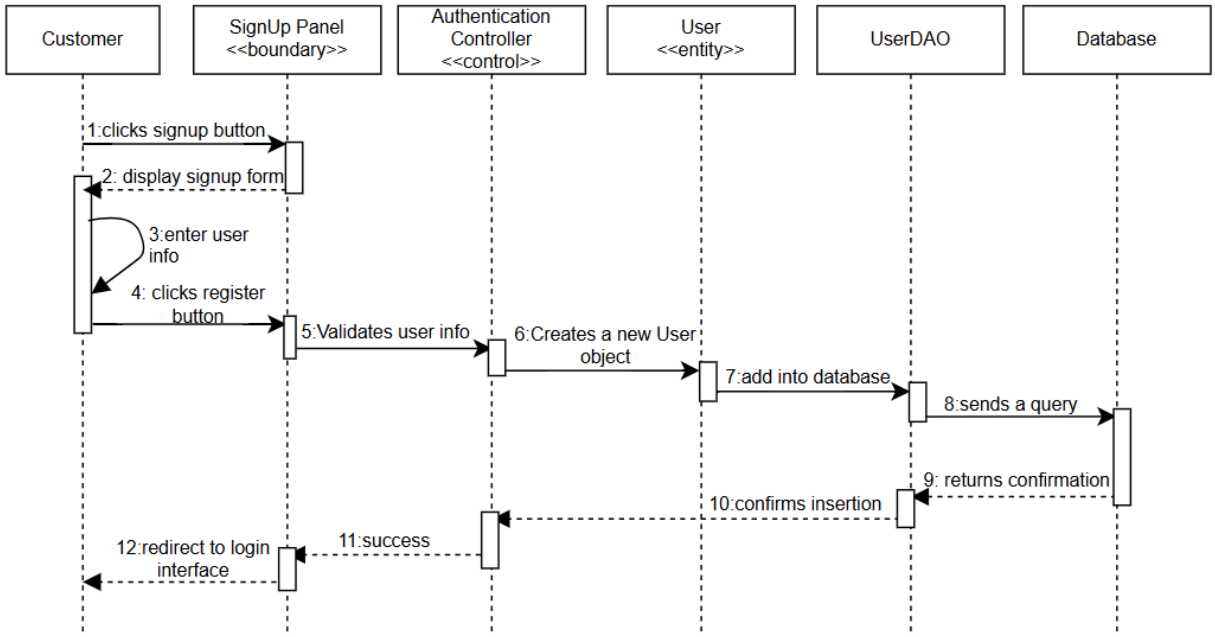
For full detailed view : **Sequence Diagram - Browsing **

Major Use case 2: Login



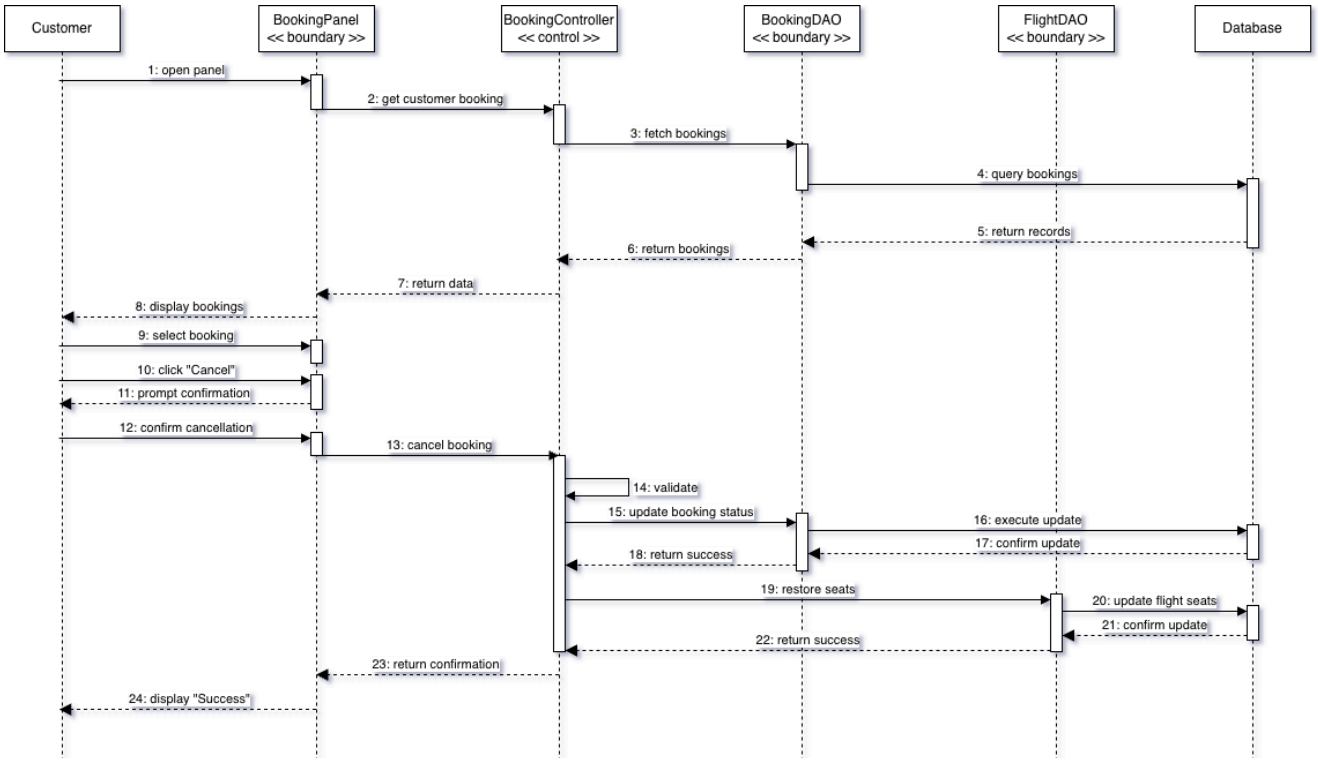
** For full detailed view : **Sequence Diagram - Login** **

Major Use case 3: Sign up



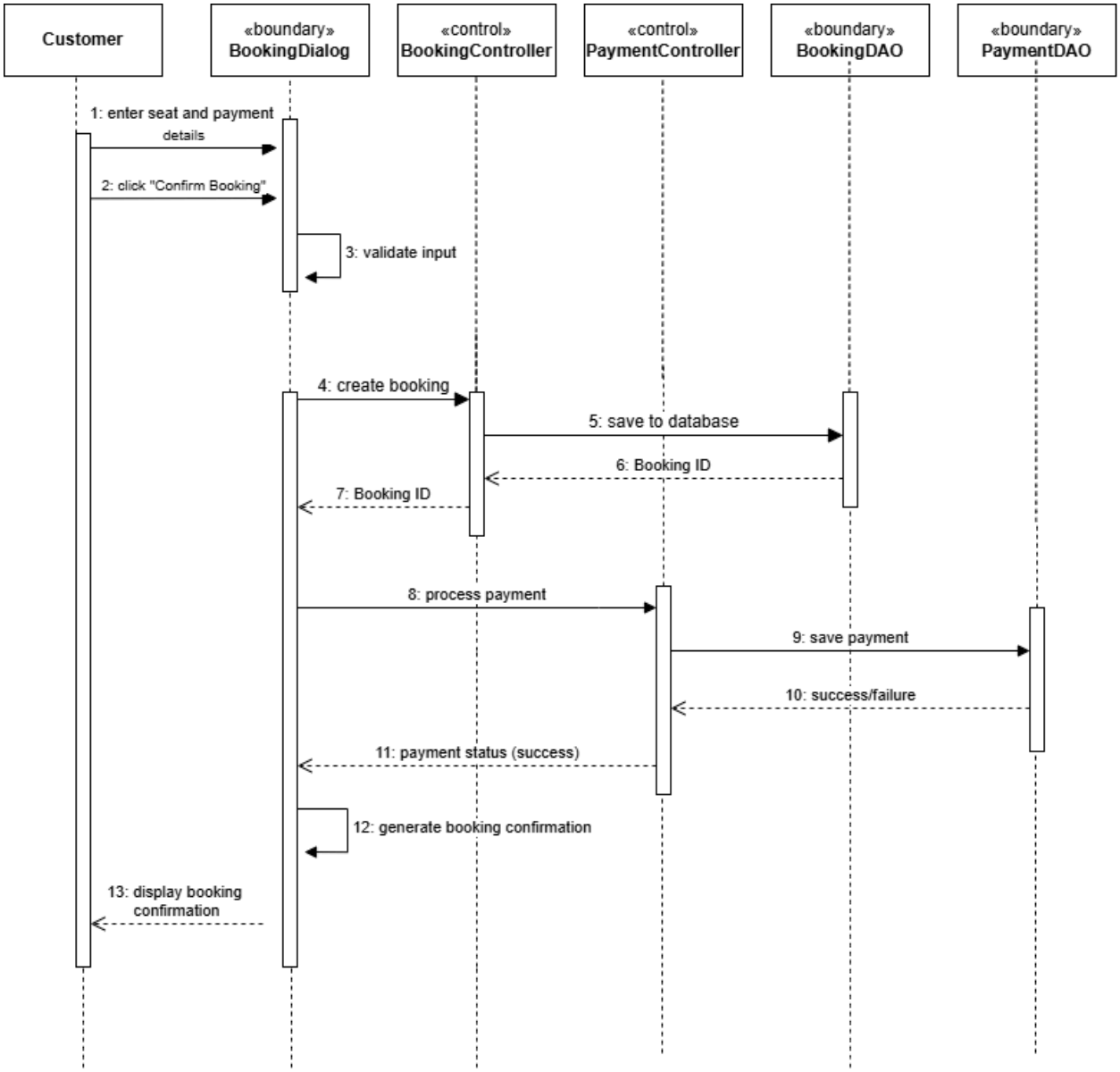
** For full detailed view : **Sequence Diagram - Sign up** **

Major Use case 4: Customer Cancel Bookings



** For full detailed view : **Sequence Diagram - Customer Cancel Bookings **

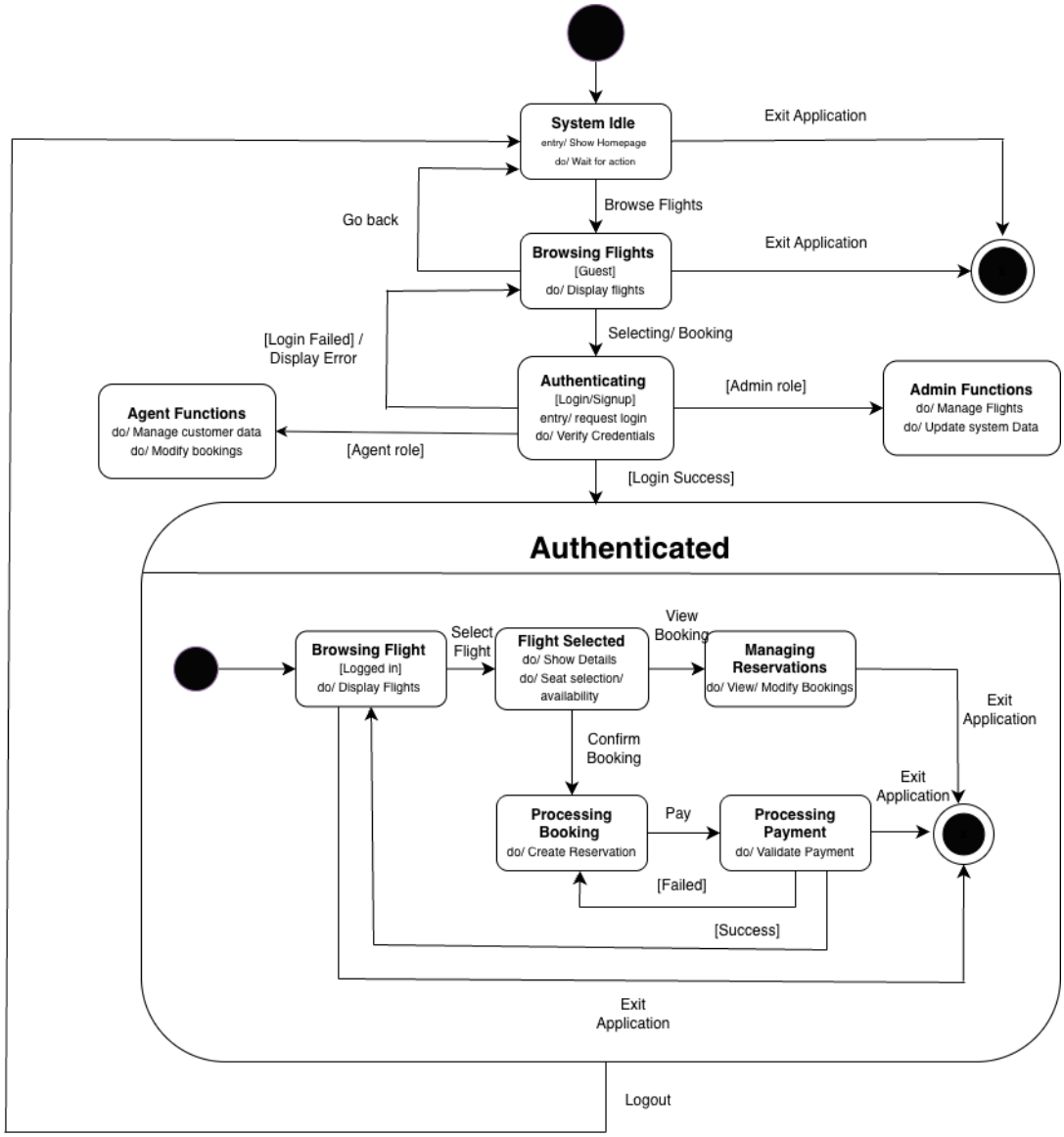
Major Use case 5: Checkout



** For full detailed view : **Sequence Diagram - Checkout** **

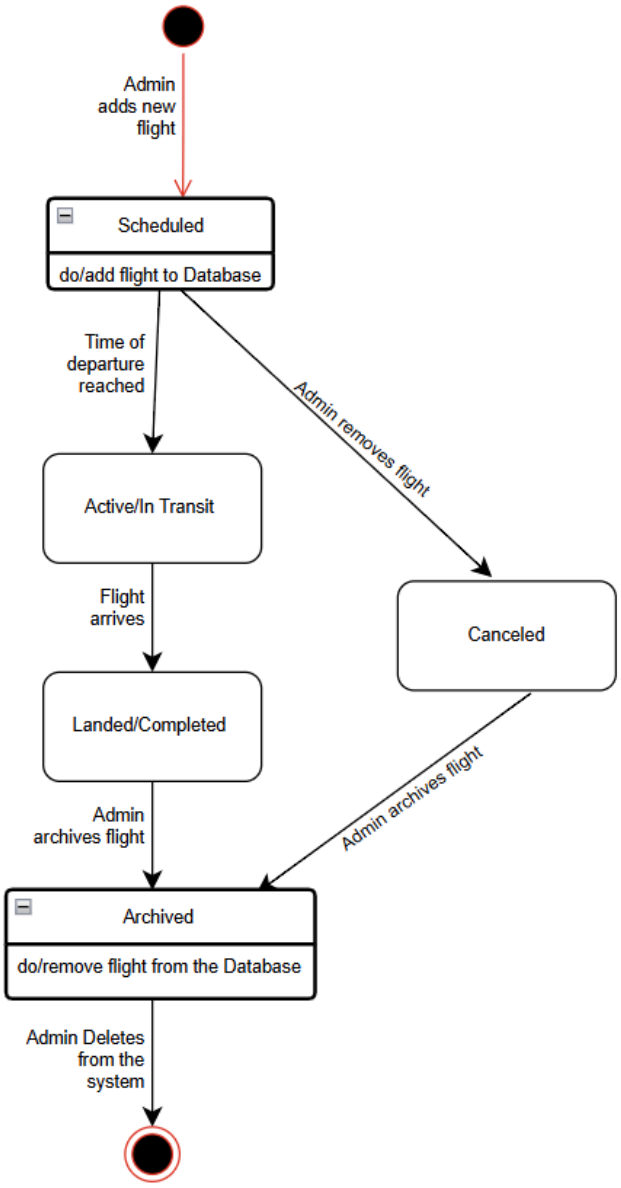
State Transition Diagrams

System



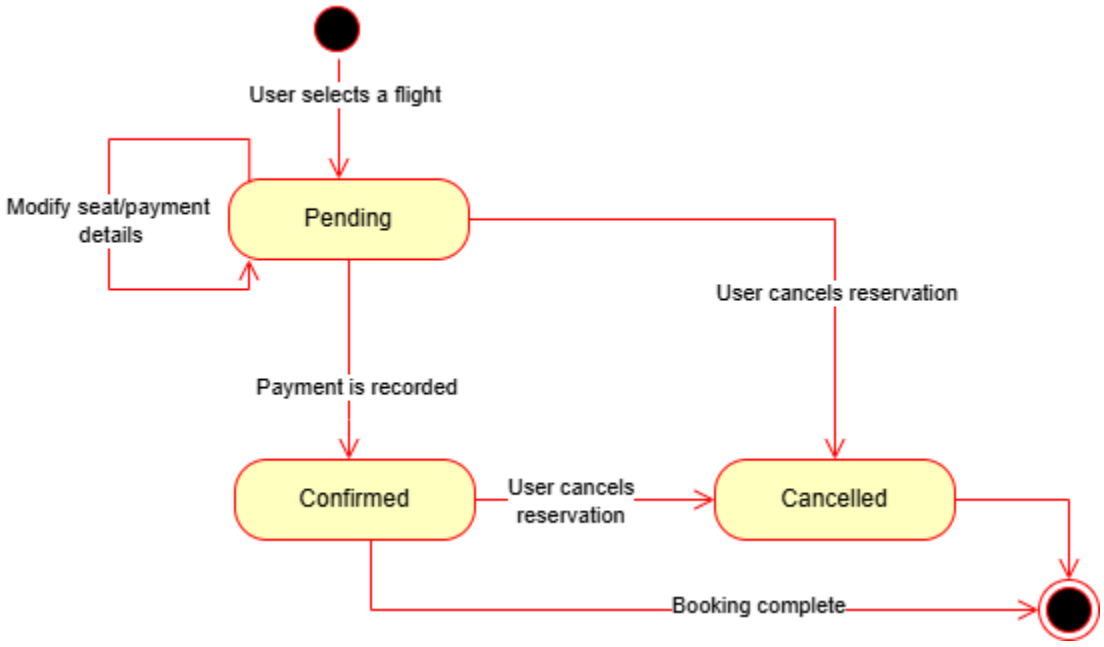
For full detailed view : **State Transition Diagram - System **

Flight Object



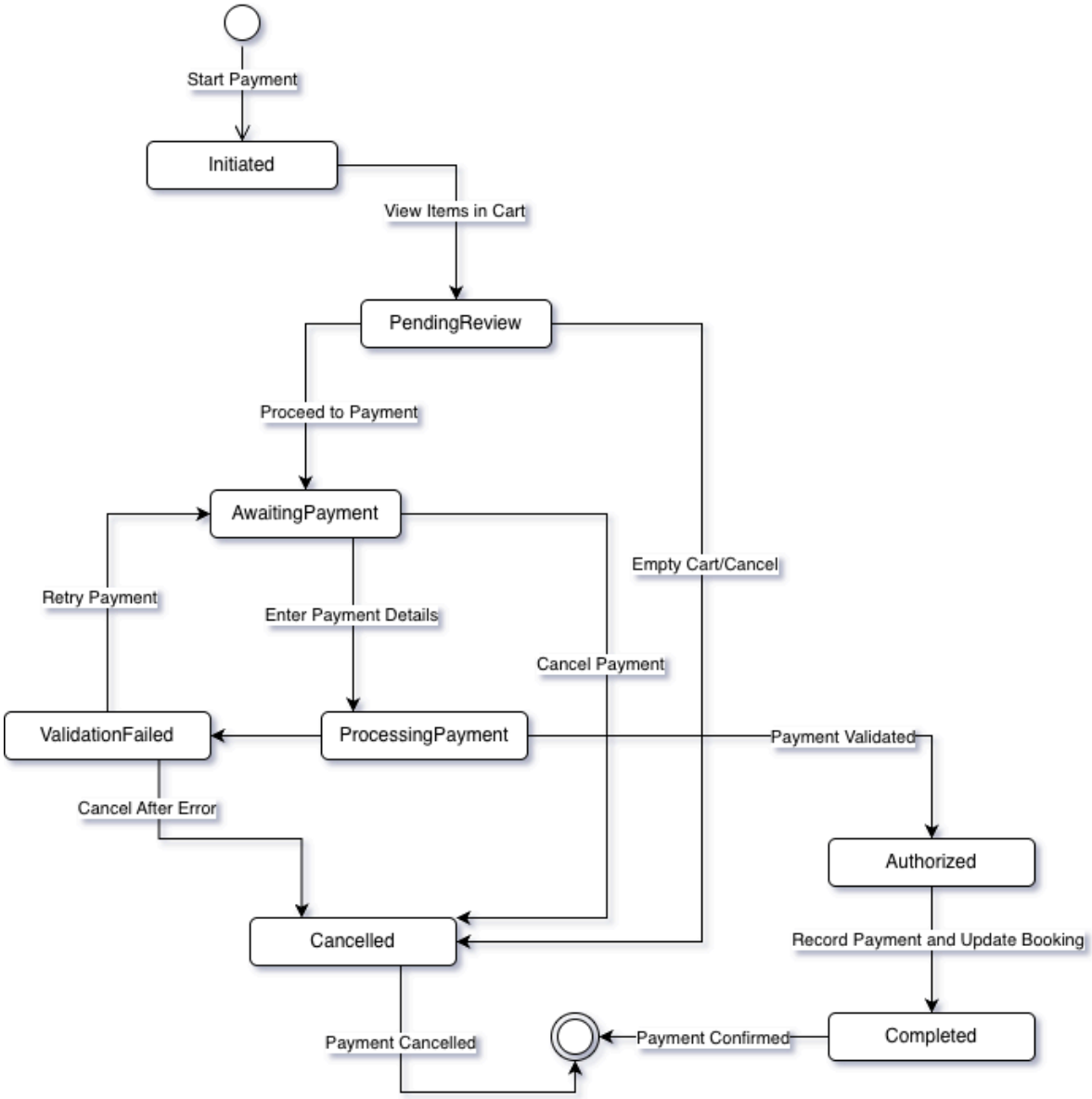
For full detailed view : **State Transition Diagram - Flight **

Reservation Object



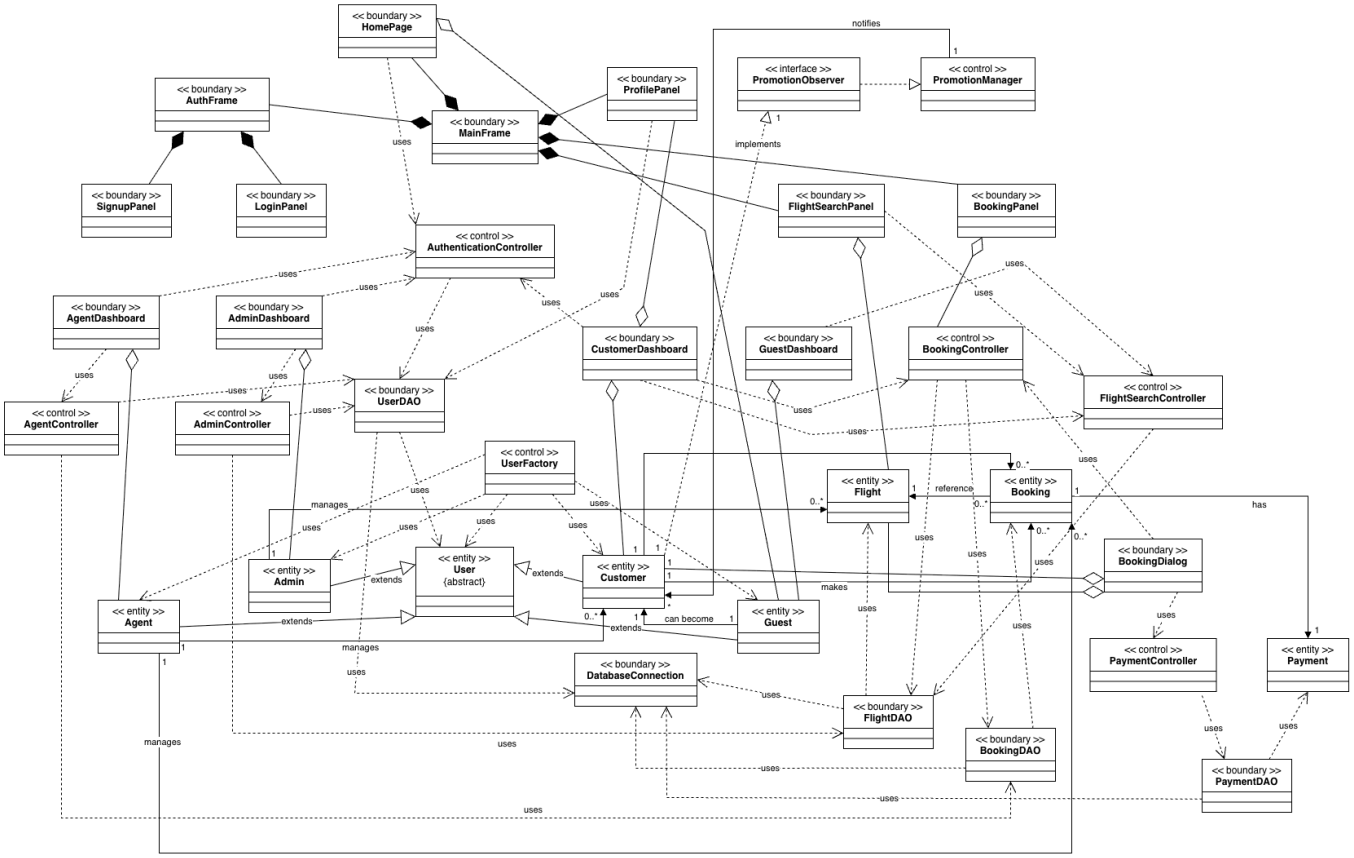
For full detailed view : **State Transition Diagram - Reservation **

Making Payments



For full detailed view : **State Transition Diagram - Payment **

Class Diagram



**For full detailed view : [Class Diagram **]

Class Diagram - Alphabetical Ordered Classes

.....

Admin

«entity»

Attributes

None (inherits from User)

Methods

- displayDashboard(): void
- toString(): String

AdminController

«control»

Attributes

- flightDAO: FlightDAO
- userDAO: UserDAO

Methods

- addFlight(flightNumber: String, departureCity: String, arrivalCity: String, departureTime: LocalDateTime, arrivalTime: LocalDateTime, totalSeats: int, price: double): int
- updateFlight(flightId: int, flightNumber: String, departureCity: String, arrivalCity: String, departureTime: LocalDateTime, arrivalTime: LocalDateTime, totalSeats: int, price: double): boolean
- deleteFlight(flightId: int): boolean
- getAllFlights(): List<Flight>
- getTotalUsers(): int
- getTotalFlights(): int

AdminDashboard

«boundary»

Attributes

- admin: Admin
- adminController: AdminController
- bookingController: BookingController
- flightSearchController: FlightSearchController
- cardLayout: CardLayout
- cardPanel: JPanel

Methods

- initializeUI(): void
- createNavButton(text: String): JButton
- createDashboardPanel(): JPanel
- createManageFlightsPanel(): JPanel
- createViewBookingsPanel(): JPanel
- createReportsPanel(): JPanel
- showAddFlightDialog(): void
- showEditFlightDialog(flight: Flight): void
- addFormField(dialog: JDialog, gbc: GridBagConstraints, row: int, label: String, field: JTextField): void
- styleButton(button: JButton, bgColor: Color): void
- logout(): void

Agent

«entity»

Attributes

None (inherits from User)

Methods

- displayDashboard(): void
- toString(): String

AgentController

«control»

Attributes

- userDAO: UserDAO
- bookingDAO: BookingDAO

Methods

- searchCustomerByEmail(email: String): User
- getCustomerBookings(userId: int): List<Booking>
- cancelCustomerBooking(bookingId: int): boolean
- getAllBookings(): List<Booking>

AgentDashboard

«boundary»

Attributes

- agent: Agent
- agentController: AgentController
- cardLayout: CardLayout
- cardPanel: JPanel

Methods

- initializeUI(): void
- createNavButton(text: String): JButton
- createDashboardPanel(): JPanel
- createSearchCustomerPanel(): JPanel
- createManageCustomersPanel(): JPanel
- createViewAllBookingsPanel(): JPanel
- showEditCustomerDialog(customer: User): void
- addFormField(dialog: JDialog, gbc: GridBagConstraints, row: int, label: String, field: JTextField): void
- styleButton(button: JButton, bgColor: Color): void
- logout(): void

AuthenticationController

«control» (Singleton)

Attributes

- instance: AuthenticationController (static)
- currentUser: User
- userDAO: UserDAO

Methods

- getInstance(): AuthenticationController (static)
- login(email: String, password: String): User
- signup(email: String, password: String, firstName: String, lastName: String, address: String, phone: String, role: String, receivePromotions: boolean): User
- logout(): void
- getCurrentUser(): User
- isLoggedIn(): boolean
- hashPassword(password: String): String
- isValidEmail(email: String): boolean
- isValidPassword(password: String): boolean

AuthFrame

«boundary»

Attributes

- cardLayout: CardLayout
- cardPanel: JPanel
- loginPanel: LoginPanel
- signupPanel: SignupPanel
- role: String

Methods

- initializeUI(): void
- showLoginPanel(): void
- showSignupPanel(): void

Booking

«entity»

Attributes

- bookingId: int
- userId: int
- flightId: int

- bookingDate: LocalDateTime
- status: String
- seatNumber: String
- customerName: String
- customerEmail: String
- flightNumber: String
- route: String

Methods

- toString(): String

BookingController

«control»

Attributes

- bookingDAO: BookingDAO
- flightDAO: FlightDAO

Methods

- createBooking(userId: int, flightId: int, seatNumber: String): int
- cancelBooking(bookingId: int): boolean
- getUserBookings(userId: int): List<Booking>
- getAllBookings(): List<Booking>
- getBookingById(bookingId: int): Booking
- modifyBooking(bookingId: int, newSeatNumber: String): boolean

BookingDAO

«boundary»

Attributes

None

Methods

- createBooking(userId: int, flightId: int, seatNumber: String): int
- cancelBooking(bookingId: int): boolean
- getUserBookings(userId: int): List<Booking>
- getAllBookings(): List<Booking>
- getBookingById(bookingId: int): Booking
- extractBookingFromResultSet(rs: ResultSet): Booking

BookingDialog

«boundary»

Attributes

- customer: Customer
- flight: Flight
- bookingController: BookingController
- paymentController: PaymentController
- seatNumberField: JTextField
- paymentMethodCombo: JComboBox<String>
- cardNumberField: JTextField
- cvvField: JTextField
- expiryField: JTextField
- nameOnCardField: JTextField
- bookingSuccessful: boolean

Methods

- initializeUI(): void
- createFlightDetailsPanel(): JPanel
- createSeatSelectionPanel(): JPanel
- createPaymentPanel(): JPanel
- confirmBooking(): void
- isBookingSuccessful(): boolean
- createBoldLabel(text: String): JLabel
- createLabel(text: String): JLabel
- styleTextField(field: JTextField): void

BookingPanel

«boundary»

Attributes

- customer: Customer
- bookingController: BookingController
- paymentController: PaymentController
- bookingsTable: JTable
- tableModel: DefaultTableModel
- viewDetailsButton: JButton
- cancelBookingButton: JButton
- refreshButton: JButton

Methods

- initializeUI(): void
- loadBookings(): void
- viewBookingDetails(): void
- createBookingDetailsPanel(booking: Booking, payment: Payment): JPanel
- createBoldLabel(text: String): JLabel

- createLabel(text: String): JLabel
- cancelBooking(): void

Customer

«entity»

Attributes

None (inherits from User)

Methods

- displayDashboard(): void
- receivePromotion(promotionMessage: String): void
- toString(): String

CustomerDashboard

«boundary»

Attributes

- customer: Customer
- cardLayout: CardLayout
- cardPanel: JPanel

Methods

- initializeUI(): void
- createNavButton(text: String): JButton
- logout(): void

DatabaseConnection

«boundary»

Attributes

- instance: DatabaseConnection (static)
- URL: String (static final)
- connection: Connection

Methods

- getInstance(): DatabaseConnection (static)
- getConnection(): Connection
- initializeDatabase(): void
- close(): void

Flight

«entity»

Attributes

- flightId: int
- flightNumber: String
- origin: String
- destination: String
- departureTime: LocalDateTime
- arrivalTime: LocalDateTime
- airline: String
- availableSeats: int
- price: double
- aircraftType: String
- totalSeats: int

Methods

- hasAvailableSeats(): boolean
- getDepartureCity(): String
- getArrivalCity(): String
- toString(): String

FlightDAO

«boundary»

Attributes

None

Methods

- initializeSampleData(): void
- insertSampleFlights(): void
- addFlight(flightNumber: String, origin: String, destination: String, departureTime: LocalDateTime, arrivalTime: LocalDateTime, availableSeats: int, price: double): int
- updateFlight(flightId: int, flightNumber: String, origin: String, destination: String, departureTime: LocalDateTime, arrivalTime: LocalDateTime, availableSeats: int, price: double): boolean
- deleteFlight(flightId: int): boolean
- searchFlights(origin: String, destination: String, date: String, airline: String): List<Flight>
- getFlightById(flightId: int): Flight
- getAllFlights(): List<Flight>
- getAllOrigins(): List<String>
- getAllDestinations(): List<String>
- getAllAirlines(): List<String>
- updateAvailableSeats(flightId: int, newSeatCount: int): boolean
- decrementAvailableSeats(flightId: int): boolean

- incrementAvailableSeats(flightId: int): boolean
- getTotalFlightCount(): int
- extractFlightFromResultSet(rs: ResultSet): Flight

FlightSearchController

«control»

Attributes

- flightDAO: FlightDAO

Methods

- searchFlights(origin: String, destination: String, date: LocalDate): List<Flight>
- searchFlights(origin: String, destination: String, date: String, airline: String): List<Flight>
- getAllFlights(): List<Flight>
- getFlightById(flightId: int): Flight
- getAllOrigins(): List<String>
- getAllDestinations(): List<String>
- getAllAirlines(): List<String>
- getAllCities(): List<String>

FlightSearchPanel

«boundary»

Attributes

- searchController: FlightSearchController
- customer: Customer
- departureCityCombo: JComboBox<String>
- arrivalCityCombo: JComboBox<String>
- airlineCombo: JComboBox<String>
- dateField: JTextField
- flightTable: JTable
- tableModel: DefaultTableModel
- searchButton: JButton
- bookButton: JButton
- clearButton: JButton

Methods

- initializeUI(): void
- loadFilters(): void
- clearFilters(): void
- loadAllFlights(): void
- searchFlights(): void
- displayFlights(flights: List<Flight>): void
- bookFlight(): void

Guest

«entity»

Attributes

None (inherits from User)

Methods

- displayDashboard(): void
- canBook(): boolean
- isGuest(): boolean
- toString(): String

GuestDashboard

«boundary»

Attributes

- guest: Guest

Methods

- initializeUI(): void

HomePage

«boundary»

Attributes

None

Methods

- initializeUI(): void
- createRoleButton(role: String, description: String): JButton
- openGuestMode(): void
- openAuthFrame(role: String): void

LoginPanel

«boundary»

Attributes

- emailField: JTextField

- passwordField: JPasswordField
- loginButton: JButton
- switchToSignupButton: JButton
- parentFrame: AuthFrame
- preSelectedRole: String

Methods

- initializeUI(): void
- handleLogin(): void
- clearFields(): void

MainFrame

«boundary»

Attributes

None

Methods

- main(args: String[]): void (static)

Payment

«entity»

Attributes

- paymentId: int
- bookingId: int
- amount: double
- paymentMethod: String
- cardNumber: String
- paymentDate: LocalDateTime

Methods

- getMaskedCardNumber(): String
- toString(): String

PaymentController

«control»

Attributes

- paymentDAO: PaymentDAO

Methods

- processPayment(bookingId: int, amount: double, paymentMethod: String, cardNumber: String): int
- getPaymentByBookingId(bookingId: int): Payment
- maskCardNumber(cardNumber: String): String
- validateCardNumber(cardNumber: String): boolean
- validateCVV(cvv: String): boolean

PaymentDAO

«boundary»

Attributes

None

Methods

- createPayment(bookingId: int, amount: double, paymentMethod: String, cardNumber: String): int
- getPaymentByBookingId(bookingId: int): Payment

ProfilePanel

«boundary»

Attributes

- customer: Customer
- userDAO: UserDAO
- promotionManager: PromotionManager
- emailField: JTextField
- firstNameField: JTextField
- lastNameField: JTextField
- addressField: JTextField
- phoneField: JTextField
- promotionsCheckbox: JCheckBox
- promotionDisplayArea: JTextArea
- promotionPanel: JPanel

Methods

- initializeUI(): void
- createPromotionDisplayPanel(): JPanel
- togglePromotionDisplay(): void
- createLabel(text: String): JLabel
- styleTextField(field: JTextField): void
- updateProfile(): void
- refreshPanel(): void

PromotionManager

«control» (Singleton + Observer)

Attributes

- instance: PromotionManager (static)
- subscribers: List<PromotionObserver>

Methods

- getInstance(): PromotionManager (static)
- subscribe(observer: PromotionObserver): void
- unsubscribe(observer: PromotionObserver): void
- notifyAllSubscribers(promotionMessage: String): void
- sendMonthlyPromotion(): void
- getSubscriberCount(): int
- isSubscribed(userId: int): boolean

PromotionObserver

«interface»

Methods

- receivePromotion(promotionMessage: String): void

SignupPanel

«boundary»

Attributes

- emailField: JTextField
- firstNameField: JTextField
- lastNameField: JTextField
- addressField: JTextField
- phoneField: JTextField
- passwordField: JPasswordField
- confirmPasswordField: JPasswordField
- receivePromotionsCheckBox: JCheckBox
- parentFrame: AuthFrame
- preSelectedRole: String

Methods

- initializeUI(): void
- addLabelAndField(labelText: String, field: JTextField, gbc: GridBagConstraints): void
- handleSignup(): void
- clearFields(): void

User

«entity» (Abstract)

Attributes

- userId: int
- email: String
- password: String
- firstName: String
- lastName: String
- address: String
- phone: String
- role: String
- receivePromotions: boolean

Methods

- displayDashboard(): void (abstract)
- isGuest(): boolean
- canBook(): boolean
- getFullName(): String
- toString(): String

UserDAO

«boundary»

Attributes

None

Methods

- authenticateUser(email: String, hashedPassword: String): User
- emailExists(email: String): boolean
- insertUser(email: String, hashedPassword: String, firstName: String, lastName: String, address: String, phone: String, role: String, receivePromotions: boolean): int
- getUserById(userId: int): User
- getUserByEmail(email: String): User
- updatePromotionPreference(userId: int, receivePromotions: boolean): boolean
- updateUserProfile(userId: int, firstName: String, lastName: String, address: String, phone: String): boolean
- getTotalUserCount(): int
- extractUserFromResultSet(rs: ResultSet): User

UserFactory

«control» (Factory Pattern)

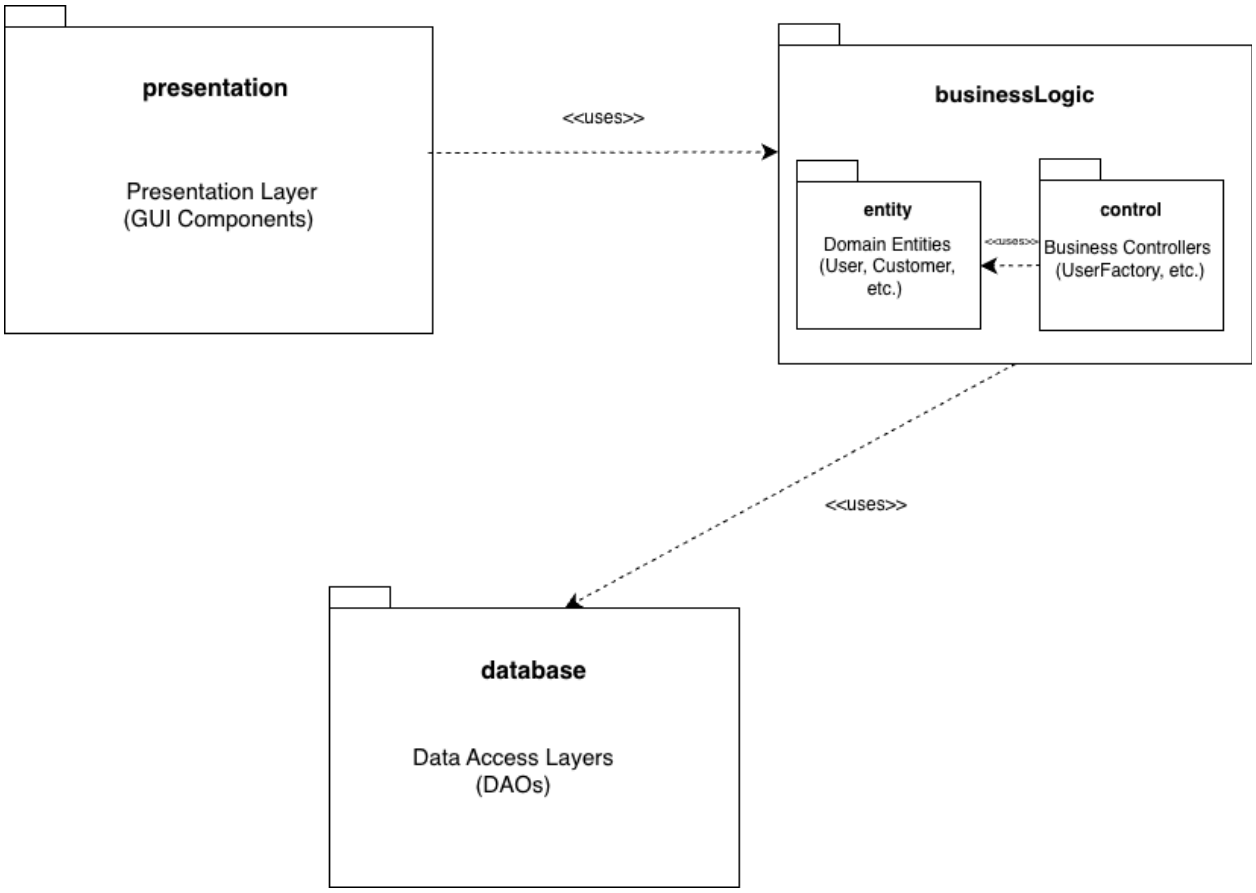
Attributes

None

Methods

- createUser(userId: int, email: String, password: String, firstName: String, lastName: String, address: String, phone: String, role: String, receivePromotions: boolean): User (static)
- createUser(userId: int, email: String, password: String, firstName: String, lastName: String, address: String, phone: String, role: String, receivePromotionsInt: int): User (static)

Package Diagram



For full detailed view : **Package Diagram **

Table with the list of classes in each package:

Package Name	Class Name	Description
presentation		**Presentation Layer**
	HomePage	Main entry point displaying welcome screen and role selection for guests vs. registered users
	MainFrame	Main Application window container managing panel switching and navigation between different views
	AuthFrame	Authentication window frame containing login and signup panel
	LoginPanel	Login form panel with email/password fields and authentication login
	SignupPanel	Sign up form panel for creating new customer accounts
	FlightSearchPanel	Search interface allowing users to search flights by origin, destination, date, and other criteria
	FlightDisplayPanel	Results display panel showing available flights
	BookingPanel	Booking form panel showing for selecting seats and passenger details
	BookingHistoryPanel	Booking form panel showing previous passenger details associated with past flights
	Profile	Displays customer details that were established during the sign up process.
	CustomerDashboard	Dashboard for logged-in customers that shows flight options for flight search, booking history, etc.
	AgentDashboard	Dashboard for flight agents that includes tools for managing customer bookings and viewing schedules
	AdminDashboard	Dashboard for system administrators which allows for flight, route, aircraft

		management.
	FlightSearchPanel	Search interface for users to input criteria and view available flights. Used by customers and agents.
	BookingDialog	Interface for making, modifying, or canceling flight reservations.
	PaymentPanel	Simulated payment processing interface that records payment details via PaymentController.
	GuestDashboard	Limited dashboard for guest users that only allows flight search and viewing (no booking).
	GuestFlightSearchPanel	Search panel for guests that disables booking function
businessLogic.entity		**Domain Entities**
	User	Abstract base class for all users containing common attributes (email, password, name, etc.)
	Customer	Customer user type extending User; can book flights and receive promotions
	Agent	Flight agent user type extending User; can manage customer bookings and reservations
	Admin	System administrator user type extending User; can manage flights, routes, and system data
	Guest	Represents a guest user extends user but with limited permission.
	PromotionObserver	Observer interface for promotions
	Flight	Flight data model containing flight details
	Booking	Booking/reservation data model linking customers to flights with seat and status information
	Payment	Payment data model storing transaction details (amount, card info, status info)

	Promotion	Promotion entity containing promotional offer details (title, description, discount, dates)
businessLogic.control		**Business Logic Controllers**
	AuthenticationController	Handles user authentication
	UserFactory	Creates user objects (Factory pattern)
	PromotionManager	Manages promotional notifications (Observer pattern)
	FlightSearchController	Handles search logic for flights including filtering by criteria and availability checking
	BookingController	Manages booking operations (create, cancel, modify reservations) and seat allocation
	PaymentController	Process payments using Strategy pattern for different methods and validate transactions
	AdminController	Handles admin only operations
	AgentController	Handles agent only operations
database		**Data Access Layer**
	DatabaseConnection	Singleton class managing SQLight database connection
	DatabaseInitializer	Handles the creation of default admin, agent, and customer users
	UserDAO	Data Access Object for flight-related database operations
	FlightDAO	Data Access Object for flight-related database operations (add, update, delete, search flights)
	BookingDAO	Data Access Object for booking/reservation database operations (create, cancel, update, query)
	PaymentDAO	Data Access Object for payment-related database operations (record, retrieve transaction history)

Package Dependencies

Source Package	Dependency Type	Target Package	Description
presentation	<<uses>>	businessLogic	The presentation layer uses controllers and entities.
businessLogic	<<uses>>	database	Business logic uses DAOs for maintaining data.
businessLogic.control	<<uses>>	businessLogic.entity	Controllers use domain entities.

Accessing the Implementation

The code can be accessed through: https://github.com/aditi-j25/ENSF_480_Term_project

---