Airline Booking System - Design Patterns and Code Walkthrough

Overview

This document provides a high-level walkthrough of the Airline Booking System. The system demonstrates modularity, scalability, and real-world problem-solving using Object-Oriented Programming and Design Patterns.

High-Level Code Walkthrough

The Airline Booking System is designed to handle complex functionalities like booking flights, applying upgrades, managing states dynamically, and saving/loading data. Below is the implementation of each design pattern used in the project along with key code snippets.

Singleton + Factory Pattern

Purpose: Ensures a single instance of Airline and provides a consistent way to create airline objects.

Code Snippet:

```
public class AirlineFactory {
    private static final Airline airlineInstance = new
Airline();

    public static Airline getInstance() {
        return airlineInstance; // Singleton instance
    }
}
```

Builder Pattern

Purpose: Simplifies the construction of complex objects like Flight and Customer.

Code Snippet:

```
FlightBuilder flightBuilder = new FlightBuilder();
Flight flight = flightBuilder.setFlightNumber("DL123")
                             .setSource("Boston")
                             .setDestination("New York")
                             .setBasePrice(200)
                             .createFlight();
```

Strategy Pattern

Purpose: Enables runtime selection of discount strategies.

Code Snippet:

```
DiscountStrategyAPI discountStrategy = new
ChristmasDiscountStrategy();
double discountedPrice =
discountStrategy.calculateDiscount(flight.getBasePrice());
System.out.println("Discounted Price: " + discountedPrice);
```

Decorator Pattern
Purpose: Dynamically adds features like extra legroom or luggage without altering the base class.
Code Snippet:
```
FlightUpgrade baseFlight = new
FlightUpgradeImplementation(200);
baseFlight = new ExtraLegRoomUpgrade(baseFlight);
baseFlight = new ExtraLuggageUpgrade(baseFlight);
System.out.println("Total Price: " + baseFlight.getPrice());
```

Command Pattern
Purpose: Encapsulates requests for booking and cancellation into objects.
Code Snippet:
```
Booking booking = new Booking(flight, customer);
Command bookCommand = new BookTicketCommand(booking);
TicketInvoker invoker = new TicketInvoker();
invoker.setCommand(bookCommand);
invoker.executeCommand();
```

Observer Pattern
Purpose: Notifies customers of flight status changes in real-time.
Code Snippet:
```
Flight flight = new Flight();
Observer emailObserver = new
EmailObserver("john.doe@example.com");
flight.addObserver(emailObserver);
flight.notifyObservers("Flight delayed by 1 hour.");
```

State Pattern
Purpose: Changes flight behavior dynamically based on its state.
Code Snippet:
```
Flight flight = new Flight();
flight.setState(new FlightScheduled());
flight.performAction(); // Output: "Flight is scheduled."
```

Prototype Pattern
Purpose: Allows cloning of bookings for reuse.
Code Snippet:
```
Booking originalBooking = new Booking(flight, customer);
Booking clonedBooking = originalBooking.cloneTicket();
System.out.println("Cloned Booking: " + clonedBooking);
```

Adapter Pattern
Purpose: Converts flight prices into multiple currencies.
Code Snippet:
```
CurrencyAdapter adapter = new CurrencyAdapter(new
USCurrency());
double priceInINR = adapter.convertTo("INR", 200);
System.out.println("Price in INR: " + priceInINR);
```

Bridge Pattern
Purpose: Decouples abstraction (ticket features) from implementation (meal preferences, seat changes).
Code Snippet:
```
TicketFeature feature = new MealPreferenceFeature(new
SeatChangeFeature());
feature.addFeature();
```

Facade Pattern
Purpose: Simplifies saving and loading data with a single interface.
Code Snippet:
```
SaveAndLoadFacadeAPI facade = new SaveAndLoadToLocal();
facade.saveBooking(booking);
facade.loadBookings();
```