Documentation: Observer and Factory Pattern Implementations in Airline Reservation System

1. Introduction

This documentation presents a comprehensive explanation of how the Observer and Factory design patterns were ingeniously implemented in the context of an Airline Reservation System. These design patterns were skillfully applied to enhance the adaptability, flexibility, and maintainability of the system.

2. Observer Pattern

2.1 Explanation

The Observer Pattern is a behavioral design pattern that facilitates a one-to-many relationship between objects. It allows a subject to maintain a list of observers and notify them of state changes without creating tight coupling.

2.2 Scenario Description

In our Airline Reservation System, the Observer Pattern is utilized to notify passengers about seat availability and booking status in real-time. Passengers, acting as observers, have a keen interest in receiving immediate updates about seat availability and bookings.

2.3 Implementation

Subject Interface (AirlineReservationSystem.java): This interface defines methods for managing observers.

public interface AirlineReservationSystem {

void addObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers(String message);

}

Concrete Subject (Flight.java): This class represents a flight, maintains a list of available seats, and notifies observers when seat availability changes.

public class Flight extends Observable implements AirlineReservationSystem { ... }

Observer Interface (Passenger.java): This interface defines the update method that observers implement to react to notifications.

public interface Passenger extends Observer {

void update(String message);

}

Concrete Observer (PassengerImpl.java): This class implements the update method to receive and display notifications.

public class PassengerImpl implements Passenger { ... }

2.4 Code Screenshots and Key Components Explanation

Key components of the Observer Pattern implementation:

Flight class: This concrete subject maintains a list of available seats and notifies observers about changes in seat availability.

Flight class

PassengerImpl class: This concrete observer reacts to notifications by implementing the update method, displaying messages to passengers.

PassengerImpl class

2.5 Challenges and Benefits

Challenges:

Complexity: Implementing the Observer Pattern can lead to complex code, particularly when managing multiple observers and subjects.

Benefits:

Flexibility: The Observer Pattern allows for the addition or removal of observers without altering the subject, promoting a flexible and scalable system.

Real-time Updates: Passengers receive real-time updates on seat availability and bookings, enhancing the user experience.

3. Factory Pattern

3.1 Explanation

The Factory Pattern is a creational design pattern that provides an interface for creating objects, allowing subclasses to change the type of objects created. It abstracts the process of object creation, enhancing flexibility and extensibility.

3.2 Scenario Description

In the Airline Reservation System, the Factory Pattern is applied to create various types of airline tickets, including economy and business class tickets. This pattern streamlines object creation, enabling the generation of objects with diverse behaviors while concealing the intricacies of the creation process.

3.3 Implementation

Product Interface (Ticket.java): This interface defines the product interface for different ticket types.

public interface Ticket {

void printTicket();

}

Concrete Products (EconomyTicket.java, BusinessTicket.java): These classes are implementations of specific ticket types.

public class EconomyTicket implements Ticket { ... }

public class BusinessTicket implements Ticket { ... }

Factory Interface (TicketFactory.java): This interface defines the factory interface for creating tickets.

public interface TicketFactory {

Ticket createTicket(String seatNumber);

}

Concrete Factories (EconomyTicketFactory.java, BusinessTicketFactory.java): These classes are implementations of the factory interface for creating specific types of tickets.

public class EconomyTicketFactory implements TicketFactory { ... }

public class BusinessTicketFactory implements TicketFactory { ... }

3.4 Code Screenshots and Key Components Explanation

Key components of the Factory Pattern implementation:

EconomyTicket class: This concrete product implements the printTicket method to display information about an economy class ticket.

EconomyTicket class

EconomyTicketFactory class: This concrete factory implements the createTicket method to create economy class tickets.

EconomyTicketFactory class

3.5 Challenges and Benefits

Challenges:

Increased Abstraction: The Factory Pattern introduces an additional layer of abstraction, which can initially complicate the codebase.

Benefits:

Extensibility: The Factory Pattern simplifies the addition of new ticket types without modifying existing code.

Encapsulation: It encapsulates the ticket creation process, making the system more modular and maintainable.

4. Conclusion

In conclusion, the unique implementation of the Observer and Factory design patterns in our Airline Reservation System significantly enhances its flexibility, scalability, and maintainability. The Observer Pattern keeps passengers informed about seat availability and bookings, while the Factory Pattern simplifies the creation of diverse ticket types. These patterns contribute to a more adaptable and user-friendly system, despite the associated challenges of increased complexity and abstraction.