Assignment 2

Design Patterns

TEAM_11

11/03/2024

Reservation-System-Starter

Vilal, Hanuma, Shriom, Madan

Introduction

Upon cloning the repository, we uncover a directory named Reservation-System-Starter-master. Let's delve into its structure and contents to identify potential design patterns. I'll provide a list of directory contents, facilitating our discussion on applying design patterns, their benefits, and drawbacks.

The Reservation-System-Starter-master directory contains the following items:

1. **README.md:** A Markdown file providing information about the project.
2. **pom.xml:** A Maven Project Object Model file, which contains configuration details for building the project.
3. **src:** The source directory, likely containing the actual code for the reservation system.

   The src directory contains two main subdirectories:

   - **main:** Likely contains the main application code.
   - **test:** Contains test cases for the application.

   Within the java directory, we find:

   - **Runner.java:** Likely the entry point of the application.
   - **flight:** A directory, which suggests it contains a package related to flight reservation functionality.

   Within reservation subdirectory we find:

   - **Airport.java:** Likely manages airport-related information.
   - **Customer.java:** Could handle customer information and actions.
   - **Passenger.java:** Presumably manages passenger-specific details.
   - **Subdirectories flight, order, payment, plane.**

Design Pattern Identification

Observer Pattern

Application: Airport.java could implement an observer pattern to notify Customer.java instances about changes in flight schedules or delays.

Benefits: Real-time updates to customers about flight changes.

Drawbacks: Increased complexity in managing observer lists and notifications.

Factory Pattern

Application: Inside the plane subdirectory, a factory pattern could be used to create instances of different plane types (e.g., Helicopter, PassengerDrone) based on specific criteria.

Benefits: Simplifies plane object creation and enhances system modularity.

Drawbacks: Might introduce unnecessary complexity if the number of flight types is limited.

Adaptor Pattern

Application: To integrate external payment services in the payment subdirectory, an adaptor pattern can help standardize the interface for different payment providers.

Benefits: Eases integration of external services with minimal changes to the application core.

Drawbacks: Requires thorough understanding of external APIs and might limit access to their unique features.

Builder Pattern

Application: Customer.java instances could be created using a builder pattern to handle complex object creation with many optional parameters.

Benefits: Enhances code readability and maintainability when dealing with objects that have numerous attributes.

Drawbacks: Increases the amount of code required for object creation.

Strategy Pattern

Application: Implementing different pricing strategies within the flight subdirectory could allow dynamic pricing based on various factors.

Benefits: Flexibility in applying different pricing algorithms without changing the system architecture.

Drawbacks: Complexity in managing multiple strategies and their interactions.

Chain of Responsibilities/Command Pattern

Application: The sequence of operations for booking a flight, involving validation, payment, and ticket issuance, could follow a chain of responsibilities or command pattern within the order subdirectory.

Benefits: Decouples the stages of order processing and enhances flexibility in adding or modifying steps.

Drawbacks: Might complicate the flow if not properly managed, leading to difficulties in debugging.

Application and Reasoning and Code Changes

Code snippets or examples showcasing the implementation of each design pattern in the context of the given codebase.

Observer Pattern: Implementation for Flight Schedule Notifications

Let's implement a simple example of the Observer pattern for flight schedule notifications. We'll define an interface FlightSchedule that represents the subject, and a FlightScheduleObserver interface that represents the observers.

Reasoning for Choosing the Observer Pattern

a. **Decoupling:** The Observer pattern decouples the subject from its observers. This means that the subject doesn't need to know anything about the observers, and the observers don't need to know anything about the subject. This decoupling makes the system more modular and easier to maintain.
b. **Flexibility:** The Observer pattern allows for a flexible addition of new observers without modifying the subject. This is particularly useful in scenarios where the number of observers can change dynamically.
c. **Efficiency:** By using the Observer pattern, you can efficiently notify multiple observers about an event without having to call each observer individually. This can lead to performance improvements, especially in systems with many observers.

This implementation demonstrates how the Observer pattern can be used to implement flight schedule notifications, making the system more flexible, decoupled, and efficient.

1. **Define the Subject Interface (Observable):**
   This interface declares methods for attaching, detaching, and notifying observers.

```
public interface Observable {
        void attach(Observer o);
        void detach(Observer o);
        void notifyObservers();
}
```

2. **Define the Observer Interface:**
   - This interface declares the update method that will be called when the subject's state changes.

```
public interface Observer {
        void update(String flightStatus);
}
```

3. **Define the Subject Interface (Observable):**
   This interface declares methods for attaching, detaching, and notifying observers.

```
public class Airport implements Observable {
        private List<Observer> observers = new ArrayList<>();
        private String flightStatus;
        @Override
        public void attach(Observer o) {
          observers.add(o);
        }
        @Override
        public void detach(Observer o) {
          observers.remove(o);
        }
        @Override
        public void notifyObservers() {
          for (Observer observer : observers) {
            observer.update(flightStatus);
          }
        }
        public void setFlightStatus(String flightStatus) {
          this.flightStatus = flightStatus;
          notifyObservers();
        }
}
```

4. **Define the Subject Interface (Observable):**
   Implement the Observer in Customer.java: The Customer class will implement the Observer interface to get notified about flight status updates.

```
public class Customer implements Observer {
        private String name;

        public Customer(String name) {
          this.name = name;
        }

        @Override
        public void update(String flightStatus) {
          System.out.println("Notification for " + name + ": Flight status changed to " +
flightStatus);
        }
}
```

5. **Example Usage:**

```
public class Runner {
```

```
    public static void main(String[] args) {
        Airport airport = new Airport();
        Customer customer1 = new Customer("Vilal Ali");
        Customer customer2 = new Customer("Madan NS");
        airport.attach(customer1);
        airport.attach(customer2);
        airport.setFlightStatus("Delayed"); // Both customers will be notified about the flight delay.
    }
}
```

This example showcases the Observer Pattern's application, allowing Customer instances (observers) to receive updates from the Airport (observable) regarding flight status changes. This pattern fosters a loose coupling between the airport (which broadcasts notifications) and the customers (who receive updates), enabling dynamic subscription and notification mechanisms without hard coding the observers into the subject. This design enhances flexibility and scalability, as new observer types can be added with minimal changes to existing code.

Factory Pattern: Implementation for Flight Object Creation

Choosing the factory pattern for the flight reservation system, particularly for creating various types of aircraft (like PassengerPlane, Helicopter, and PassengerDrone), is influenced by several factors related to the codebase's challenges and requirements. Here's a detailed analysis:

Reasoning for Choosing the Factory Pattern

1. **Variability and Extensibility:** The codebase likely deals with various types of aircraft that share common properties and behaviors but also have unique characteristics. The factory pattern allows for creating different aircraft instances dynamically based on input parameters (type and model), making it easier to introduce new types of aircraft without modifying existing factory logic or client code.
2. **Decoupling:** By using a factory pattern, the code that uses aircraft objects (like scheduling flights) is decoupled from the actual creation logic of these objects. This means changes to the creation process, or the introduction of new aircraft types don't impact the rest of the codebase, adhering to the Open/Closed Principle.
3. **Simplification:** Client code that needs aircraft objects doesn't have to deal with the complexity of instantiating different types of aircraft. This simplifies client code and centralizes the instantiation logic, reducing the risk of errors.
4. **Type Safety:** The factory can return a common interface or abstract class type (Plane in this context), ensuring that all created objects conform to a known set of methods and properties. This enhances type safety and predictability in client code.

Code Changes

1. **Factory Class:**
   We have defined factory class PlaneFactory.java under the directory Reservation-System-Starter\src\main\java\flight\reservation\plane

```java
package flight.reservation.plane;
// Assuming Plane is an interface or abstract class that all plane types implement or extend
public class PlaneFactory {
        public static Plane createPlane(String type, String model) {
            switch (type) {
                case "PassengerPlane":
                    return new PassengerPlane(model);
                case "Helicopter":
                    return new Helicopter(model);
                case "PassengerDrone":
                    return new PassengerDrone(model);
                default:
                    throw new IllegalArgumentException("Unknown plane type: " + type);
            }
        }
}
```

2. **Interface Class:**

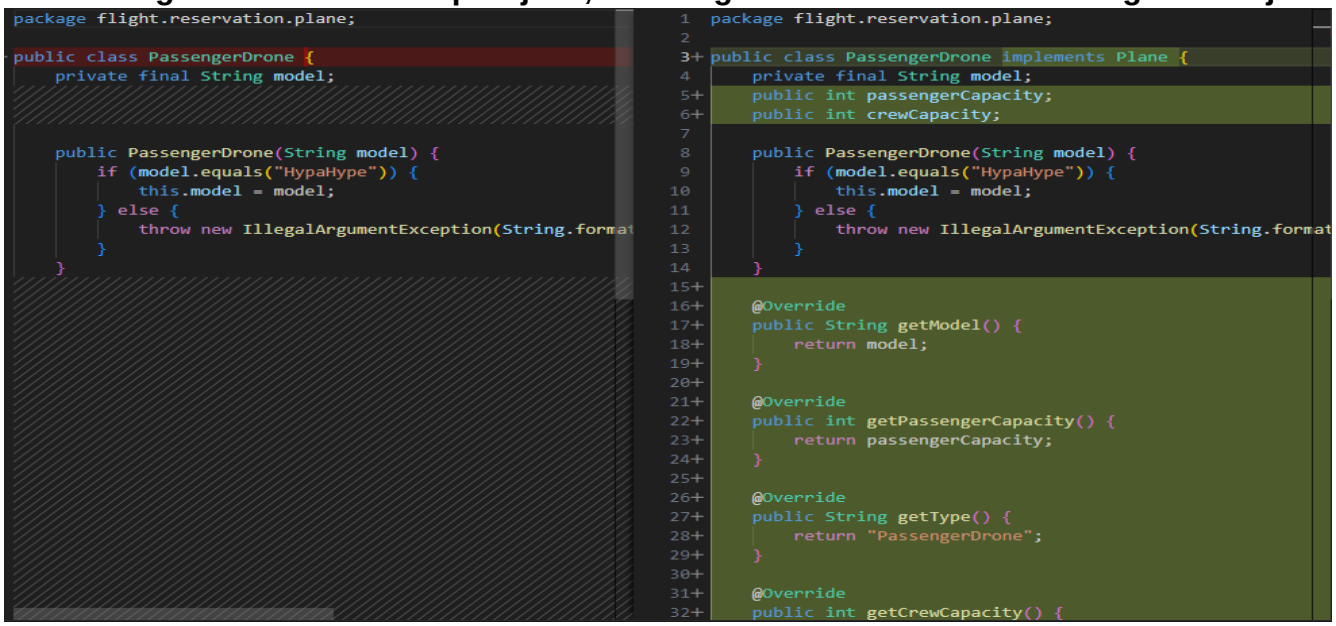
We have defined new interface Plane.java under the directory Reservation-System-Starter\src\main\java\flight\reservation\plane

```
package flight.reservation.plane;
public interface Plane {
        String getModel();
        int getPassengerCapacity();
        String getType();
        int getCrewCapacity() ;
}
```

3. **Other Class changes:**
   There have been considerable changes in multiple files due to the factory objects being called in multiple files. Here is a list of all the files where the changes were made.

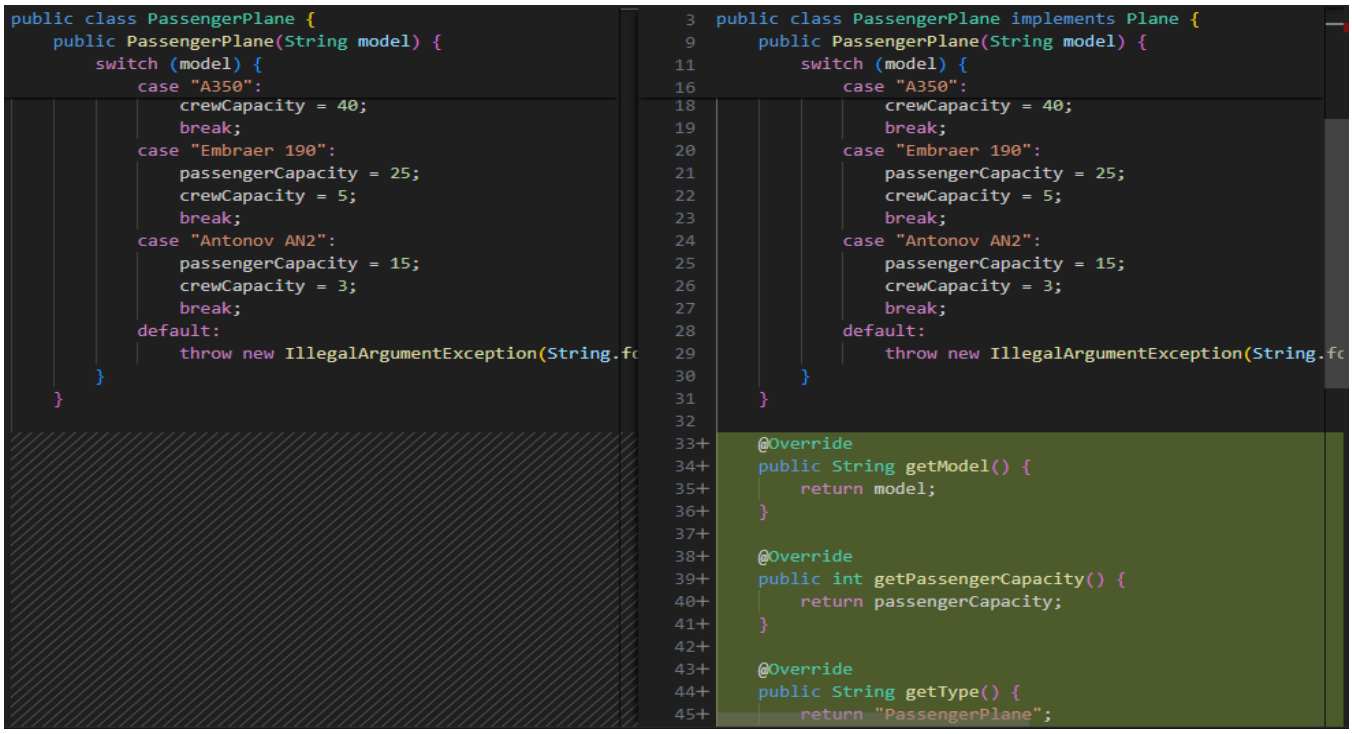
```
modified: src/main/java/Runner.java
modified: src/main/java/flight/reservation/flight/Flight.java
modified: src/main/java/flight/reservation/flight/Schedule.java
modified: src/main/java/flight/reservation/flight/ScheduledFlight.java
modified: src/main/java/flight/reservation/plane/Helicopter.java
modified: src/main/java/flight/reservation/plane/PassengerDrone.java
modified: src/main/java/flight/reservation/plane/PassengerPlane.java
modified: src/test/java/flight.reservation/ScenarioTest.java
modified: src/test/java/flight.reservation/ScheduleTest.java
New Addedd File
src/main/java/flight/reservation/plane/Plane.java
Src/main/java/flight/reservation/plane/PlaneFactory.java
```

4. **Code changes done for Helicopter.java, PassengerDrone.Java and PassengerPlane.java**



```
package flight.reservation.plane;

public class PassengerDrone {
    private final String model;


    public PassengerDrone(String model) {
        if (model.equals("HypaHype")) {
            this.model = model;
        } else {
            throw new IllegalArgumentException(String.format
        }
    }
```

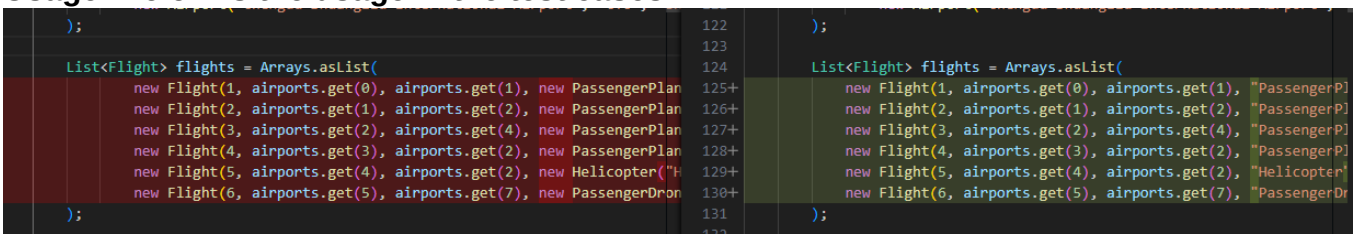
```
package flight.reservation.plane;

public class PassengerDrone implements Plane {
    private final String model;
    public int passengerCapacity;
    public int crewCapacity;

    public PassengerDrone(String model) {
        if (model.equals("HypaHype")) {
            this.model = model;
        } else {
            throw new IllegalArgumentException(String.format
        }
    }

    @Override
    public String getModel() {
        return model;
    }

    @Override
    public int getPassengerCapacity() {
        return passengerCapacity;
    }

    @Override
    public String getType() {
        return "PassengerDrone";
    }

    @Override
    public int getCrewCapacity() {
```



```
public class PassengerPane {
    public PassengerPane(String model) {
        switch (model) {
            case "A350":
                crewCapacity = 40;
                break;
            case "Embraer 190":
                passengerCapacity = 25;
                crewCapacity = 5;
                break;
            case "Antonov AN2":
                passengerCapacity = 15;
                crewCapacity = 3;
                break;
            default:
                throw new IllegalArgumentException(String.fc
        }
    }
```

```
3  public class PassengerPane implements Plane {
9      public PassengerPane(String model) {
11         switch (model) {
16             case "A350":
18                 crewCapacity = 40;
19                 break;
20             case "Embraer 190":
21                 passengerCapacity = 25;
22                 crewCapacity = 5;
23                 break;
24             case "Antonov AN2":
25                 passengerCapacity = 15;
26                 crewCapacity = 3;
27                 break;
28             default:
29                 throw new IllegalArgumentException(String.fc
30         }
31     }
32
33+    @Override
34+    public String getModel() {
35+        return model;
36+    }
37+
38+    @Override
39+    public int getPassengerCapacity() {
40+        return passengerCapacity;
41+    }
42+
43+    @Override
44+    public String getType() {
45+        return "PassengerPlane";
```

5. **Usage: Below is the usage in the test cases.**



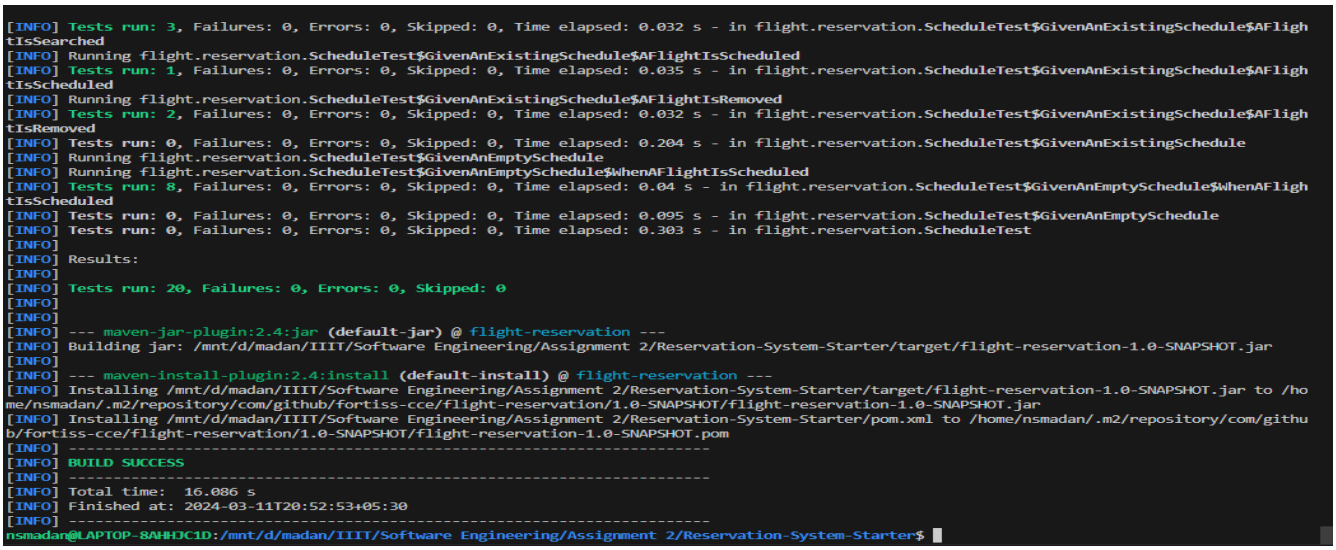
```
    );

    List<Flight> flights = Arrays.asList(
            new Flight(1, airports.get(0), airports.get(1), new PassengerPlan
            new Flight(2, airports.get(1), airports.get(2), new PassengerPlan
            new Flight(3, airports.get(2), airports.get(4), new PassengerPlan
            new Flight(4, airports.get(3), airports.get(2), new PassengerPlan
            new Flight(5, airports.get(4), airports.get(2), new Helicopter("H
            new Flight(6, airports.get(5), airports.get(7), new PassengerDron
    );
```

```
122     );
123
124     List<Flight> flights = Arrays.asList(
125+         new Flight(1, airports.get(0), airports.get(1), "PassengerP]
126+         new Flight(2, airports.get(1), airports.get(2), "PassengerP]
127+         new Flight(3, airports.get(2), airports.get(4), "PassengerP]
128+         new Flight(4, airports.get(3), airports.get(2), "PassengerP]
129+         new Flight(5, airports.get(4), airports.get(2), "Helicopter"
130+         new Flight(6, airports.get(5), airports.get(7), "PassengerDr
131     );
```

6. **Potential Benefits**
   a. **Flexibility**: New aircraft types can be added without changing the code that the factory uses, making the system more adaptable to future requirements.
   b. **Reusability**: The factory itself and the creation pattern it encapsulates become reusable components. Different parts of the application that are needed to create aircraft objects can leverage the factory, ensuring consistency in object creation.
   c. **Improved Maintenance**: With aircraft creation logic centralized in the factory, maintenance tasks (like fixing bugs or updating creation logic for a specific type) are localized, reducing the impact on the broader codebase.
7. **Potential Drawbacks**
   d. **Complexity**: Introducing a factory pattern can add complexity to the codebase, especially if the creation logic is simple and unlikely to change. For small applications or projects with a limited variety of objects, the added abstraction might not justify the benefits.
   e. **Overhead**: There's a slight runtime overhead associated with the factory pattern, particularly if the creation logic involves reflection or complex decision-making. However, for most applications, this overhead is negligible compared to the benefits of flexibility and maintainability.
   f. **Indirection**: The factory pattern introduces an additional layer of indirection, which can make the code harder to trace or debug for developers unfamiliar with the pattern or the application architecture.

2. **Build Status – SUCCESS**

Below is the build status will all test cases passed



Adaptor Pattern: Implementation for Payment Service Integration

To implement the Adaptor Pattern for integrating a third-party payment gateway into your reservation system, you'll need to create an adaptor class that can convert ReservationDetail objects into PaymentDetail objects that the payment service expects. This approach allows your system to interact with the payment gateway without modifying the existing ReservationDetail class or the payment service's interface.

First, let's define the PaymentDetail class that the payment service expects. This class might look something like this:

1. **PaymentDetail (flight.reservation.payment):**

```
package flight.reservation.payment;

   public interface PaymentDetail {
        String getEmail();
        String getPaymentMethod();
        double getAmount();
}
```

2. **ReservationDetail (flight.reservation):**

```
package flight.reservation;

   public class ReservationDetail {
        private String email;
        private double amount;

        public ReservationDetail(String email, double amount) {
            this.email = email;
            this.amount = amount;
        }

        public String getEmail() {
            return email;
        }

        public double getAmount() {
            return amount;
        }
}
```

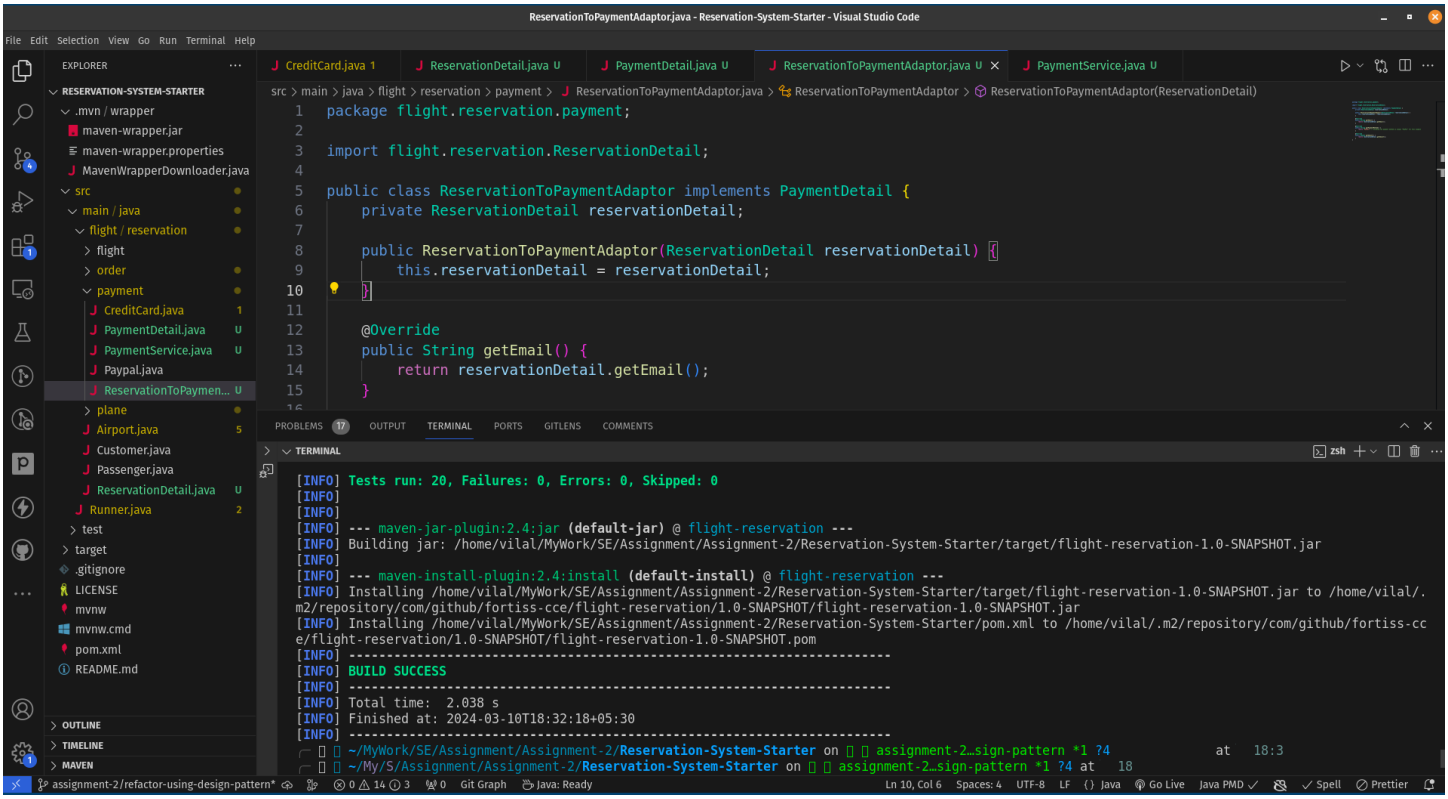
3. **ReservationToPaymentAdaptor (flight.reservation.payment):**

```
package flight.reservation.payment;
import flight.reservation.ReservationDetail;

public class ReservationToPaymentAdaptor implements PaymentDetail {
        private ReservationDetail reservationDetail;

        public ReservationToPaymentAdaptor(ReservationDetail reservationDetail) {
          this.reservationDetail = reservationDetail;
        }

        @Override
        public String getEmail() {
          return reservationDetail.getEmail();
        }

        @Override
        public String getPaymentMethod() {
            return "PayPal"; // Assuming the payment method is always "PayPal" for this example
        }

        @Override
        public double getAmount() {
            return reservationDetail.getAmount();
        }
}
```

4. **PaymentService (flight.reservation.payment):**

```
package flight.reservation.payment;

import flight.reservation.ReservationDetail;

public class PaymentService {
        public void processPayment(ReservationDetail reservationDetail) {
            ReservationToPaymentAdaptor adaptor = new
ReservationToPaymentAdaptor(reservationDetail);
            // Now you can use the adaptor object with the payment service
            // For example, paymentService.processPayment(adaptor);
            // This is a placeholder for the actual payment processing logic
            System.out.println("Processing payment for " + adaptor.getEmail() + " with amount "
+ adaptor.getAmount());
        }
}
```

5. BUILD SUCCESS: After Refactoring the code using Adapter pattern.



Builder Pattern: Implementation for Customer Builder

We have implemented the builder pattern in Customer.java class. The Builder pattern is particularly useful for classes with many constructor parameters, either required or optional, enhancing code readability and maintainability. The Customer class involves complex object construction, which is evident from its multiple fields, such as name, email, and orders. As the system evolves, more fields might be added, further complicating the construction process. The Builder pattern allows for a step-by-step construction process, enhancing code readability and maintainability.

Code Changes

1. **Customer Builder: (src/main/java/flight/reservation/Customer.java)**

   Assuming the Customer class has fields like name, email, and orders, a basic implementation of the Builder pattern for the Customer class might look like this:

```
public class Customer {
        private String name;
        private String email;
        private List<Order> orders;

        private Customer(Builder builder) {
            this.name = builder.name;
            this.email = builder.email;
            this.orders = builder.orders;
        }

        public static class Builder {
            private String name;
            private String email;
            private List<Order> orders;

            public Builder(String name, String email) {
                this.name = name;
                this.email = email;
            }

            public Builder orders(List<Order> orders) {
                this.orders = orders;
                return this;
            }

            public Customer build() {
                return new Customer(this);
            }
        }
}
```

2. **Potential Benefits:**
    a. **Improved Code Clarity and Maintainability:** The Builder pattern significantly improves code readability and maintainability by replacing complex constructors with a fluent interface.
    b. **Flexibility in Object Creation:** It offers more control over the object creation process, allowing for optional parameters, varying construction sequences, and even constructing different immutable objects using the same building process.
    c. I**mmutable Objects After Construction:** Ensuring that Customer objects are immutable after they have been built enhances the system's robustness, especially in a concurrent execution environment.
3. **Drawbacks:**
    a. **Increased Complexity for Simple Scenarios:** For classes that are unlikely to evolve or do not have many parameters, implementing the Builder pattern might unnecessarily complicate the design.
    b. **Boilerplate Code:** The pattern requires additional code to be written, especially the Builder class itself, which might be seen as boilerplate if the simplicity of object creation is not a major concern.
    c. **Performance Considerations:** While typically negligible, the extra layer of object creation (the Builder) could introduce a slight performance overhead, especially in performance-critical applications.

1. **BUILD SUCCESS with ALL TEST CASES PASSED**

```
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 s - in flight.reservation.ScheduleTest$GivenAnExistingSchedule$AFlight
tIsSearched
[INFO] Running flight.reservation.ScheduleTest$GivenAnExistingSchedule$AFlightIsScheduled
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.036 s - in flight.reservation.ScheduleTest$GivenAnExistingSchedule$AFlight
tIsScheduled
[INFO] Running flight.reservation.ScheduleTest$GivenAnExistingSchedule$AFlightIsRemoved
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 s - in flight.reservation.ScheduleTest$GivenAnExistingSchedule$AFlight
tIsRemoved
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.161 s - in flight.reservation.ScheduleTest$GivenAnExistingSchedule
[INFO] Running flight.reservation.ScheduleTest$GivenAnEmptySchedule
[INFO] Running flight.reservation.ScheduleTest$GivenAnEmptySchedule$WhenAFlightIsScheduled
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.049 s - in flight.reservation.ScheduleTest$GivenAnEmptySchedule$WhenAFlig
htIsScheduled
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.087 s - in flight.reservation.ScheduleTest$GivenAnEmptySchedule
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.265 s - in flight.reservation.ScheduleTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ flight-reservation ---
[INFO] Building jar: /mnt/d/madan/IIIT/Software Engineering/Assignment 2/Reservation-System-Starter/target/flight-reservation-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ flight-reservation ---
[INFO] Installing /mnt/d/madan/IIIT/Software Engineering/Assignment 2/Reservation-System-Starter/target/flight-reservation-1.0-SNAPSHOT.jar to /ho
me/nsmadan/.m2/repository/com/github/fortiss-cce/flight-reservation/1.0-SNAPSHOT/flight-reservation-1.0-SNAPSHOT.jar
[INFO] Installing /mnt/d/madan/IIIT/Software Engineering/Assignment 2/Reservation-System-Starter/pom.xml to /home/nsmadan/.m2/repository/com/githu
b/fortiss-cce/flight-reservation/1.0-SNAPSHOT/flight-reservation-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14.642 s
[INFO] Finished at: 2024-03-11T10:33:07+05:30
[INFO] ------------------------------------------------------------------------
```

Strategy Pattern: Implementation for Pricing Strategies

The Strategy pattern is a behavioral design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which family of algorithms to use. This pattern is particularly useful when you have multiple ways to perform an operation and you want to choose the appropriate one at runtime.

Implementation for Pricing Strategies:

Let's implement a simple example of the Strategy pattern for different pricing strategies for flights. We'll define an interface PricingStrategy and several concrete strategies that implement this interface.

Code Changes

1. **Define the Strategy Interface (PricingStrategy.java):**

```
+ package flight.reservation.pricing;
+ import flight.reservation.Customer;
+ import flight.reservation.flight.Flight;
+
+ public interface PricingStrategy {
+     double calculatePrice(Flight flight, Customer customer);
+ }
```

2. **Implement Concrete Strategies (TimeOfBookingStrategy.java):**

```
public class TimeOfBookingStrategy implements PricingStrategy {
        @Override
        public double calculatePrice(Flight flight) {
          // Implement pricing logic based on time of booking
          return flight.getBasePrice() * 1.1; // Example logic
        }
}
        public class DemandStrategy implements PricingStrategy {
        @Override
          public double calculatePrice(Flight flight) {
            // Implement pricing logic based on demand
            return flight.getBasePrice() * 1.2; // Example logic
          }
}

public class LoyaltyPointsStrategy implements PricingStrategy {
        @Override
          public double calculatePrice(Flight flight) {
          // Implement pricing logic based on loyalty points
          return flight.getBasePrice() * 0.9; // Example logic
        }
}
```

3. **Use the Strategy in the Flight Class:**

Assuming you have a Flight class, you can use the PricingStrategy interface to allow different pricing strategies to be applied.

```
public class Flight {
        private double basePrice;
        private PricingStrategy pricingStrategy;
        public Flight(double basePrice) {
          this.basePrice = basePrice;
        }
        public void setPricingStrategy(PricingStrategy pricingStrategy) {
          this.pricingStrategy = pricingStrategy;
        }
        public double getPrice() {
          return pricingStrategy.calculatePrice(this);
        }
}
```

4. **Client Code:**

```
public class FlightPricing {
        public static void main(String[] args) {
          Flight flight = new Flight(100.0);
          // Apply different pricing strategies
          flight.setPricingStrategy(new TimeOfBookingStrategy());
         System.out.println("Price with Time of Booking Strategy: " + flight.getPrice());
          flight.setPricingStrategy(new DemandStrategy());
          System.out.println("Price with Demand Strategy: " + flight.getPrice());
          flight.setPricingStrategy(new LoyaltyPointsStrategy());
          System.out.println("Price with Loyalty Points Strategy: " + flight.getPrice());
        }
}
```

5. **Benefits and Impact:**

   Flexibility: The Strategy pattern provides a flexible way to change the behavior of an object at runtime. This is particularly useful for algorithms that can be selected dynamically.

   Maintainability: By separating the algorithm from the object that uses it, the Strategy pattern makes the code more modular and easier to maintain.

   Extensibility: New strategies can be added without modifying the existing code, making the system more extensible.

This implementation demonstrates how the Strategy pattern can be used to implement different pricing strategies for flights, making the system more flexible and maintainable.

Chain of Responsibilities/Command Pattern: Implementation for Booking Process

Chain different stages in the flight booking process, such as validation, payment, and confirmation.

1. **Handler Interface:**

```
public abstract class BookingHandler {
        protected BookingHandler next;
        public void setNextHandler(BookingHandler next) {
            this.next = next;
        }
        public abstract void handleRequest(BookingRequest request);
}
```

2. **Concrete Handlers:**

```
public class ValidationHandler extends BookingHandler {
        public void handleRequest(BookingRequest request) {
            if (validateRequest(request)) {
                if (next != null) {
                    next.handleRequest(request);
                }
            } else {
                 throw new RuntimeException("Validation failed.");
            }
        }
        private boolean validateRequest(BookingRequest request) {
            // Validation logic
            return true;
        }
}
public class PaymentHandler extends BookingHandler {
        public void handleRequest(BookingRequest request) {
        if (processPayment(request)) {
            if (next != null) {
                next.handleRequest(request);
            }
        } else {
             throw new RuntimeException("Payment failed.");
        }
}
        private boolean processPayment(BookingRequest request) {
            // Payment processing logic
            return true;
```

```
        }
}
```

Conclusion

In conclusion, the analysis of the Reservation-System-Starter-master codebase has revealed several opportunities for optimizing functionality through the application of various design patterns. The identified patterns, including Observer, Factory, Adaptor, Builder, Strategy, and Chain of Responsibilities/Command, address specific challenges within the codebase, offering benefits such as enhanced flexibility, modularity, and maintainability. The code snippets provided illustrate the practical implementation of these design patterns in relevant contexts, demonstrating how they can contribute to an improved reservation system.

While each pattern brings unique advantages, it's crucial to carefully consider potential drawbacks and assess the trade-offs to ensure that the chosen design patterns align with the specific needs and goals of the project. Overall, the thoughtful integration of these design patterns has the potential to elevate the codebase, fostering a more scalable and adaptable reservation system.