In Class Activity: 23/24

1- Observer Pattern:

Potential Use Case: Notifying customers or systems when a flight's status changes (e.g., delays, cancellations). This could involve modifying classes such as Flight or ScheduledFlight to become subjects, with observers being the Customer instances or a system monitoring flight statuses.

We have updated

Reservation-System-Starter/src/main/java/flight/reservation/flight/ScheduledFlight.java

1. **Define the Observer Interface:** This interface will be implemented by any class that wants to observe the ScheduledFlight.

```
public interface FlightObserver {
    void update(ScheduledFlight flight);
}
```

2. **Modify the ScheduledFlight Class:** Add methods to add and remove observers, and a method to notify all observers when the flight's status changes.

```
import java.util.ArrayList;
import java.util.List;

public class ScheduledFlight extends Flight {
    // Existing fields and methods...

    private List<FlightObserver> observers = new ArrayList<>();

    public void addObserver(FlightObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(FlightObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        for (FlightObserver observer : observers) {
            observer.update(this);
        }
    }

    // Example method that changes the flight's status and notifies observers
    public void changeStatus(String newStatus) {
        // Change the status of the flight
        // notifyObservers(); // Notify observers about the status change
    }
}
```

2. Facto Factory Pattern:

Potential Use Case: Creating different types of Plane objects (Helicopter, PassengerDrone, PassengerPlane) based on criteria. Implementing a PlaneFactory class could streamline object creation, especially if the instantiation logic is complex or if there are many different types of planes. ry Pattern:

Based on the analysis, the Factory Pattern is the most evident pattern in the provided code. The Runner.java file uses Arrays.asList to create a list of Flight objects, which is a simple form of the factory pattern.

To refactor the code using the Factory Pattern, we can create a FlightFactory.java class that will be responsible for creating Flight objects. This way, we can encapsulate the creation logic and make it more flexible and maintainable.

Here's how you can refactor the Runner.java file to use a FlightFactory:

First, create a FlightFactory class:

```
public class FlightFactory {
    public static Flight createFlight(int flightNumber, Airport departureAirport, Airport arrivalAirport, Object aircraft) {
        return new Flight(flightNumber, departureAirport, arrivalAirport, aircraft);
    }
}
```

Then, in the Runner.java file, replace the direct creation of Flight objects with calls to the FlightFactory:

```
public class Runner {
    // ... (other code remains unchanged)

    static List<Flight> flights = Arrays.asList(
            FlightFactory.createFlight(1, airports.get(0), airports.get(1), aircrafts.get(0)),
            FlightFactory.createFlight(2, airports.get(1), airports.get(2), aircrafts.get(1)),
            FlightFactory.createFlight(3, airports.get(2), airports.get(4), aircrafts.get(2)),
            FlightFactory.createFlight(4, airports.get(3), airports.get(2), aircrafts.get(3)),
            FlightFactory.createFlight(5, airports.get(4), airports.get(2), aircrafts.get(4)),
            FlightFactory.createFlight(6, airports.get(5), airports.get(7), aircrafts.get(5))
    );

    // ... (rest of the class remains unchanged)
}
```

After Implementing the code: Build Success!!

```
[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: /home/vilal/MyWork/SE/ClassActvity/ClassActivity-4/Reservation-System-Starter/target/flight-reservation-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ flight-reservation ---
[INFO] Installing /home/vilal/MyWork/SE/ClassActvity/ClassActivity-4/Reservation-System-Starter/target/flight-reservation-1.0-SNAPSHOT.jar to /home/vilal/.m
2/repository/com/github/fortiss-cce/flight-reservation/1.0-SNAPSHOT/flight-reservation-1.0-SNAPSHOT.jar
[INFO] Installing /home/vilal/MyWork/SE/ClassActvity/ClassActivity-4/Reservation-System-Starter/pom.xml to /home/vilal/.m2/repository/com/github/fortiss-cce
/flight-reservation/1.0-SNAPSHOT/flight-reservation-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.683 s
[INFO] Finished at: 2024-02-20T15:14:41+05:30
[INFO] ------------------------------------------------------------------------
╭─  ~/My/S/Cl/ClassActivity-4/Reservation-System-Starter on  master        took
╭─  ~/MyWork/SE/ClassActvity/ClassActivity-4/Reservation-System-Starter on  master                                took  5s at   15:14:41 ─╮
```

Builder Pattern:

Potential Use Case: Simplifying the creation of complex Order objects, which might involve multiple steps to set up various aspects of a flight reservation (e.g., selecting flights, adding passengers, choosing payment methods). A builder class could help encapsulate this complexity.

We have updated

src/main/java/flight/reservation/order/Order.java

```java
package flight.reservation.order;

import flight.reservation.Customer;
import flight.reservation.Passenger;

import java.util.List;
import java.util.UUID;

public class Order {

    private final UUID id;
    private double price;
    private boolean isClosed = false;
    private Customer customer;
    private List<Passenger> passengers;

    private Order(OrderBuilder builder) {
        this.id = builder.id;
        this.price = builder.price;
        this.isClosed = builder.isClosed;
        this.customer = builder.customer;
        this.passengers = builder.passengers;
    }

    public UUID getId() {
        return id;
    }
      Sidx-sys, 12 months ago • Starter Code
    public double getPrice() {
        return price;
    }

    public Customer getCustomer() {
        return customer;
    }
```

```java
public List<Passenger> getPassengers() {
    return passengers;
}

public boolean isClosed() {
    return isClosed;
}

public static class OrderBuilder {

    private final UUID id;
    private double price;
    private boolean isClosed = false;
    private Customer customer;
    private List<Passenger> passengers;

    public OrderBuilder() {
        this.id = UUID.randomUUID();
    }

    public OrderBuilder price(double price) {
        this.price = price;
        return this;
    }

    public OrderBuilder isClosed(boolean isClosed) {
        this.isClosed = isClosed;
        return this;
    }

    public OrderBuilder customer(Customer customer) {
        this.customer = customer;
        return this;
    }

    public OrderBuilder passengers(List<Passenger> passengers) {
        this.passengers = passengers;
        return this;
    }

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

With this modification, you can now use the OrderBuilder class to create Order objects with a more fluent and readable syntax:

Order order = new Order.OrderBuilder()

.price(100.0)

.isClosed(false)

.customer(customer)

.passengers(passengers)

.build();