Design Patterns Activity Report

Chain of Responsibilities

```java
// Handler.java
package flight.reservation.order;

interface Handler {
    void setNext(Handler h);
    boolean handle(Order o);
}
```

```java
// PaymentHandler.java
package flight.reservation.order;

import flight.reservation.payment.PaymentStrategy;

public class PaymentHandler {
    Order order;
    Handler nextHandler;

    public void setNext(Handler h) {
        this.nextHandler = h;
    }

    public boolean handle(Order order) {
        this.order = order;
        return Pay(order.getPaymentStrategy());
    }

    public boolean Pay(PaymentStrategy paymentMethod){
        // pay
    }
}
```

```java
// ProcessingHandler.java
package flight.reservation.order;

public class ProcessingHandler implements Handler{

    Order order;
    Handler nextHandler;
// ...
    public boolean handle(Order order) {
        this.order = order;
        if (order.getPaymentStrategy().getClass() !=
CreditCard.class) {
            CreditCard card = (CreditCard)
order.getPaymentStrategy();
            return processOrderWithCreditCard(card);
        } else {
            String password = "";
            // get password from UI
            return
processOrderWithPayPal(order.getCustomer().getEmail(),
password);
        }
    }

    public boolean processOrderWithCreditCardDetail(String
number, Date expirationDate, String cvv) {
        // do processing
    }

    public boolean processOrderWithCreditCard(CreditCard
creditCard) {
        // do processing
    }
}
```

The payment of an order required multiple steps without much overlap and the possibility of failure at any step. Another problem was the crowding of responsibilities within the `FlightOrder` class.
Thus, we've employed a chain of responsibilities between a `PaymentHandler` class and a `ProcessingHandler` class that both implement the `Handler` interface.

Command

```java
// Command.java
package ui.command;

public interface Command {
    void exec();
}
```

```java
// ScheduleFlightCommand.java
package ui.command;

import java.util.Date;

import flight.reservation.flight.Flight;
import flight.reservation.flight.Schedule;

public class ScheduleFlightCommand implements Command{
    private Flight flight;
    private Date date;
    // ...
    public void exec() {
        Schedule schedule = new Schedule();
        // get schedule
        schedule.scheduleFlight(flight, date);
    }
}
```

```java
package ui;

import ui.command.Command;

public class Button {
    // ...

    private Command command;

    // ...

    public void clickButton() {
        command.exec();
    }
}
```

A use case was assumed - One where the UI for this software can have many button classes to do different operations. The multiple button subclasses are problematic while maintaining the base button class.
To fix this, a `Command` interface allows for the creation of command classes that can be attached to the same button class. And the many button subclasses can be removed.

Composite

```java
// ScheduledFlight.java
package flight.reservation.flight;

public class ScheduledFlight extends Flight {

    private final List<Passenger> passengers;
    private final Date departureTime;
    private double currentPrice = 100;
    private List<ScheduledFlight> layovers;

        // ...

    public double getCurrentPrice() {
        double total = currentPrice;
        for (ScheduledFlight flight: this.layovers) {
            total += flight.getCurrentPrice();
        }
        return total;
    }

        // ...

    public void addLayoverFlight(ScheduledFlight layover)
    {
        this.layovers.add(layover);
    }
}
```

A new use case was assumed - Consider that each `ScheduledFlight` may consist of many layover flights.

A list of `layovers` was added to `ScheduledFlight` and the price calculation was adjusted to check each of the flights in `layovers` and sum them.

Adapter

```
// Flight.java
package flight.reservation.flight;

class PassengerPlaneAdapter implements PassengerPlane {
    private final PassengerDrone passengerDrone;

    public PassengerPlaneAdapter(PassengerDrone
passengerDrone) {
        this.passengerDrone = passengerDrone;
    }

    @Override
    public String getModel() {
        return "HypaHype";
    }

    // ...
}
```

```
public class Flight {

  private int number;
  private Airport departure;
  private Airport arrival;
  protected Object aircraft;

  public Flight(int number, Airport departure, Airport
arrival, Object aircraft) throws IllegalArgumentException {
    this.number = number;
    this.departure = departure;
    this.arrival = arrival;
    this.aircraft = adaptAircraft(aircraft);
    checkValidity();
  }
  private Object adaptAircraft(Object aircraft){
    if(aircraft instanceof PassengerPlane || aircraft
instanceof Helicopter){
      return aircraft;
    }
    else if(aircraft instanceof PassengerDrone){
      return new
PassengerPlaneAdapter((PassengerDrone)aircraft);
    }else{
      throw new
IllegalArgumentException(String.format("Aircraft not
recognizable"));
    }
  }
  // ...
}
```

We introduced a new method `adaptAircraft` to the updated `Flight` class, which takes an `Object` parameter and returns an adapted aircraft. The `adaptAircraft` method verifies the type of the input aircraft and returns a new object of the same type. If the input aircraft is a `PassengerDrone`, a new `PassengerPlaneAdapter` object is created, which converts the `PassengerDrone` to the `PassengerPlane` interface. If the input aircraft is a `PassengerPlane` or `Helicopter`, the input object is returned directly.
In the modified code, the following changes have been made:

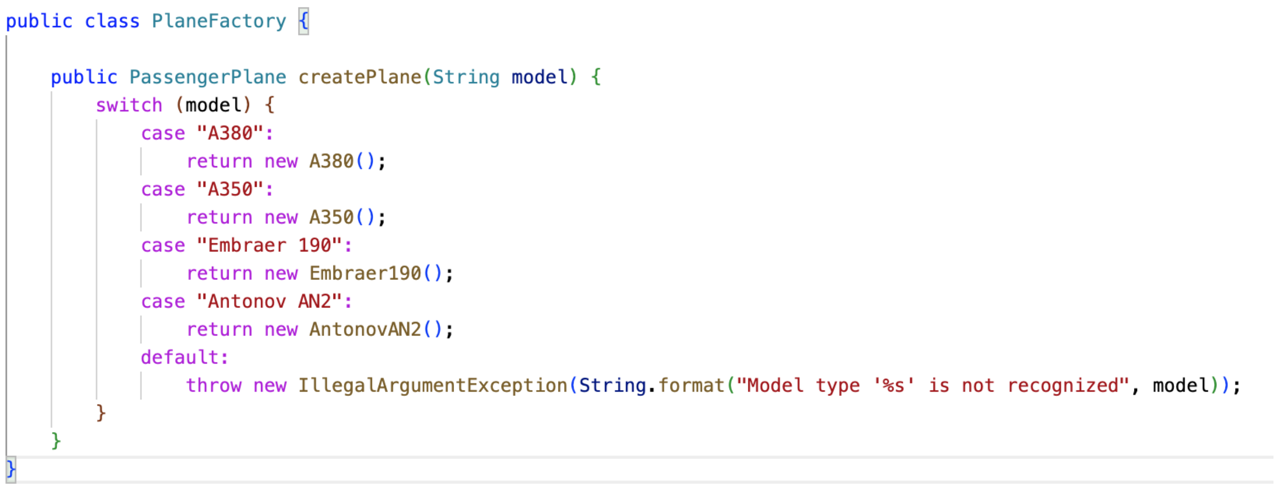
- The `Flight` Constructor now takes a `passengerPlane` object instead of an `Object`.
- The `isAircraftValid` method has been modified to handle `PassengerPlaneAdapter`, which returns the string `”HypaHype”` as model.
- Added an adapter class `PassengerPlaneadapter` that adapts the `PassengerDrone` class to the `PassengerPlane`

Factory and Observer

Issue1:
Current way of creating different models of planes is inextensible because if we add a new attribute for a plane then the same attribute should be updated in all the conditions of the CASE statement.

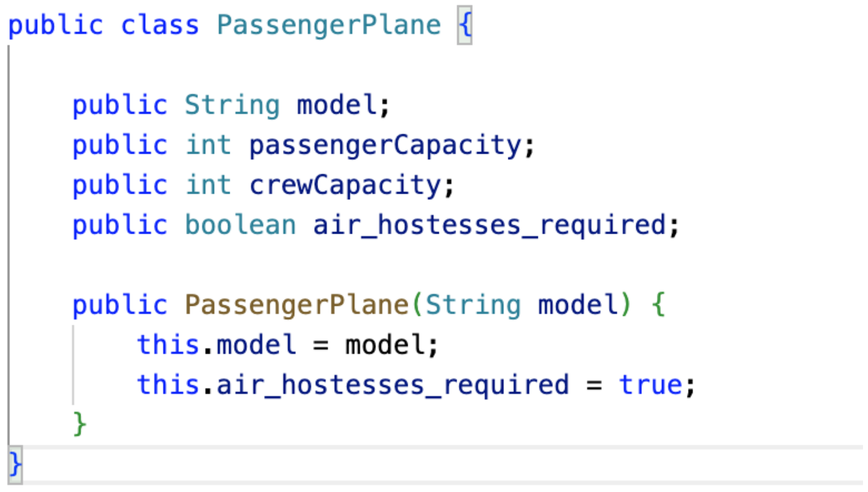
Resolution:

1. We can use Factory pattern in order to create different types of planes. With the factory pattern, we can encapsulate the logic of object creation and it'll make it easy to add new types of objects in the future without requiring changes to the client code.



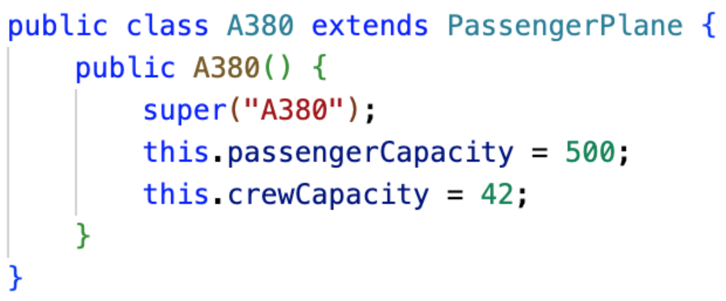
```java
public class PlaneFactory {

    public PassengerPlane createPlane(String model) {
        switch (model) {
            case "A380":
                return new A380();
            case "A350":
                return new A350();
            case "Embraer 190":
                return new Embraer190();
            case "Antonov AN2":
                return new AntonovAN2();
            default:
                throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}
```

2. Generally, we use an interface to define object behavior and implement that interface for object creation while setting respective attributes. But, in this case we've used super class PassengerPlane with all the common attributes.
3. Constructor of super class will set the default values for all objects of PassengerPlane.

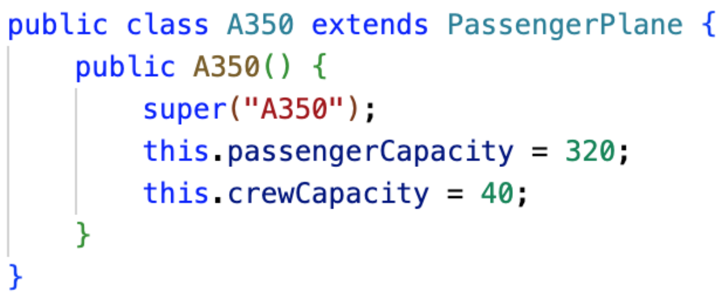


```java
public class PassengerPlane {

    public String model;
    public int passengerCapacity;
    public int crewCapacity;
    public boolean air_hostesses_required;

    public PassengerPlane(String model) {
        this.model = model;
        this.air_hostesses_required = true;
    }
}
```

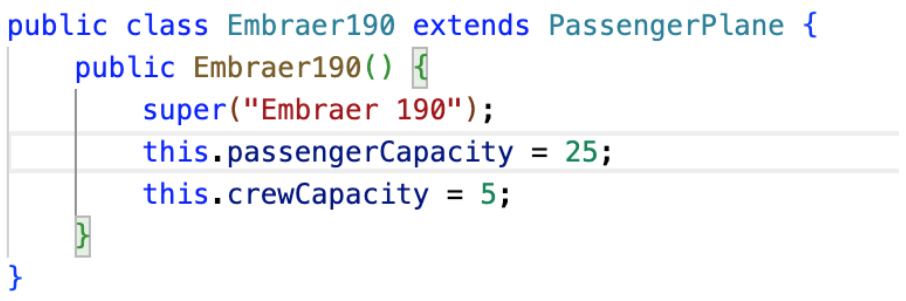
4. To add a new type of plane, we just need to create a new subclass of PassengerPlane and add a case to the factory's switch statement. In this way, we can make the object creation more extensible.



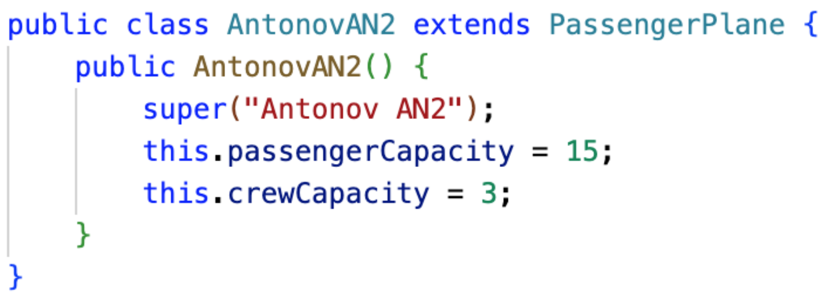
```
public class A380 extends PassengerPlane {
    public A380() {
        super("A380");
        this.passengerCapacity = 500;
        this.crewCapacity = 42;
    }
}
```



```
public class A350 extends PassengerPlane {
    public A350() {
        super("A350");
        this.passengerCapacity = 320;
        this.crewCapacity = 40;
    }
}
```



```
public class Embraer190 extends PassengerPlane {
    public Embraer190() {
        super("Embraer 190");
        this.passengerCapacity = 25;
        this.crewCapacity = 5;
    }
}
```



```
public class AntonovAN2 extends PassengerPlane {
    public AntonovAN2() {
        super("Antonov AN2");
        this.passengerCapacity = 15;
        this.crewCapacity = 3;
    }
}
```

Issue2:
Currently, there is no provision to notify the passengers about the flight updates like change in schedule, cancellations etc.

Resolution:
We can leverage the observer pattern in order to notify all the passengers of a flight.

1. Create an interface that defines the methods that the observers can use to register themselves, unregister themselves, and receive updates from the subject.

```
public interface FlightNotifications {
   public void addPassenger(Passenger passenger);
   public void removePassenger(Passenger passenger);
   public void notifyPassengers();
}
```

2. The existing class 'ScheduledFlight' which has a list of passengers can implement this interface to add, remove and notify the passengers. Following is the implementation of those methods:

```
public void addPassenger(Passenger passenger) {
    this.passengers.add(passenger);
}

public void removePassenger(Passenger passenger) {
    this.passengers.remove(passenger);
}

public void notifyPassengers() {
    for (Passenger passenger : this.passengers) {
        passenger.update(this);
    }
}

public void setDepartureTime(Date departureTime) {
    this.departureTime = departureTime;
    this.notifyPassengers();
}
```

3. Now, we need to have an interface for receiving the notification in Passenger. For that, we will create an interface for observer which will be implemented in Passenger.

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

4. This behavior will be implemented in Passenger in the following way:

```
public void update(ScheduledFlight flight) {
       System.out.println("Notification for user " + this.name + ":
Flight " + flight.getFlightNumber() + " has changed Departure Time to " +
flight.getDepartureTime());
   }
```

5. With this, we can achieve the notification functionality to passengers upon change in schedule of a flight.

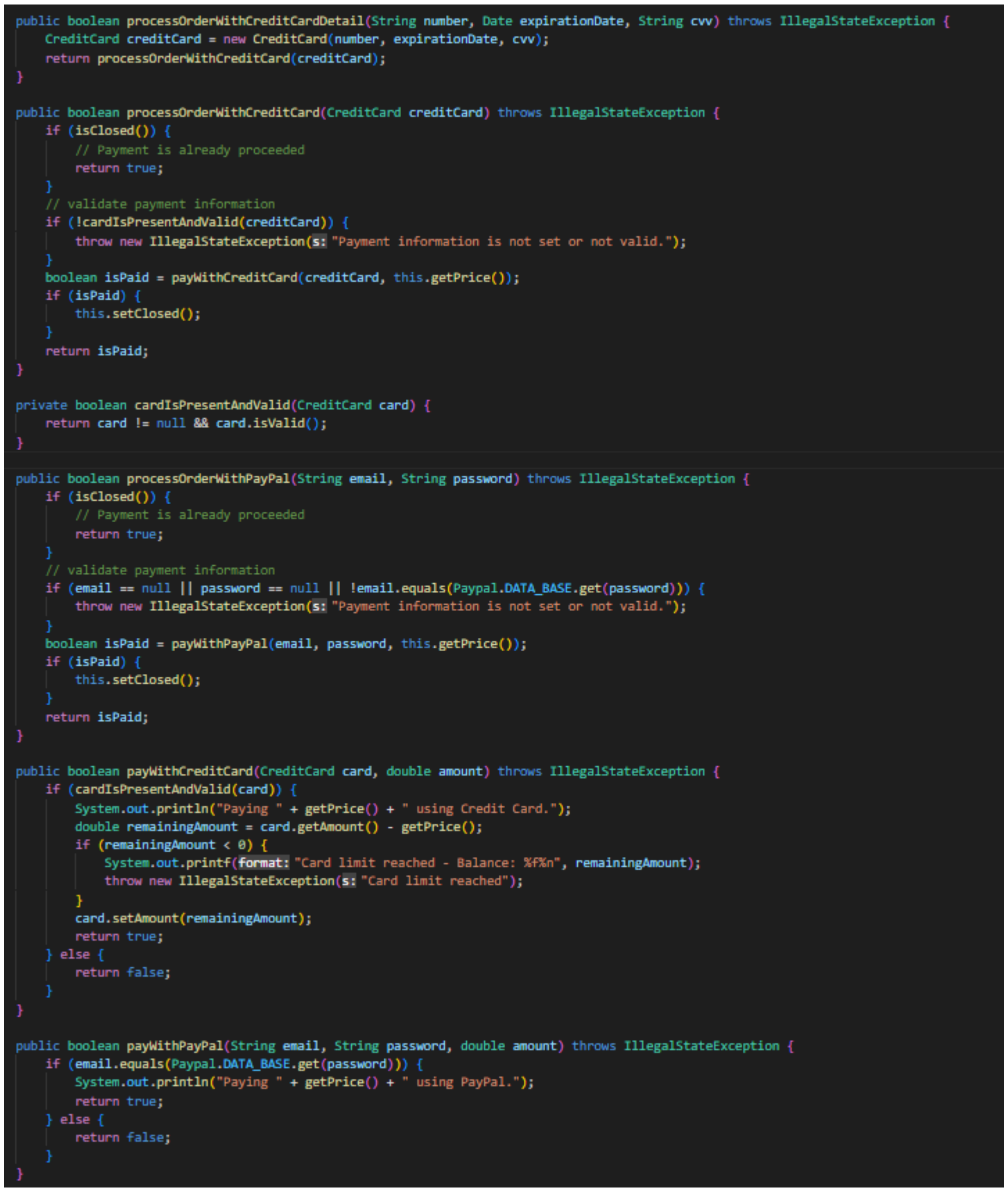
Strategy pattern

The payment system for a flight booking is as follows in the system

In the “FlightOrder” class the methods

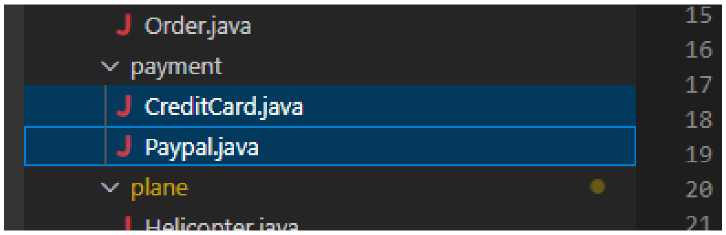
```
processOrderWithCreditCardDetail
processOrderWithCreditCard
payWithCreditCard
processOrderWithPayPal
payWithPayPal
```

are used to process a payment either with Cred card or paypal



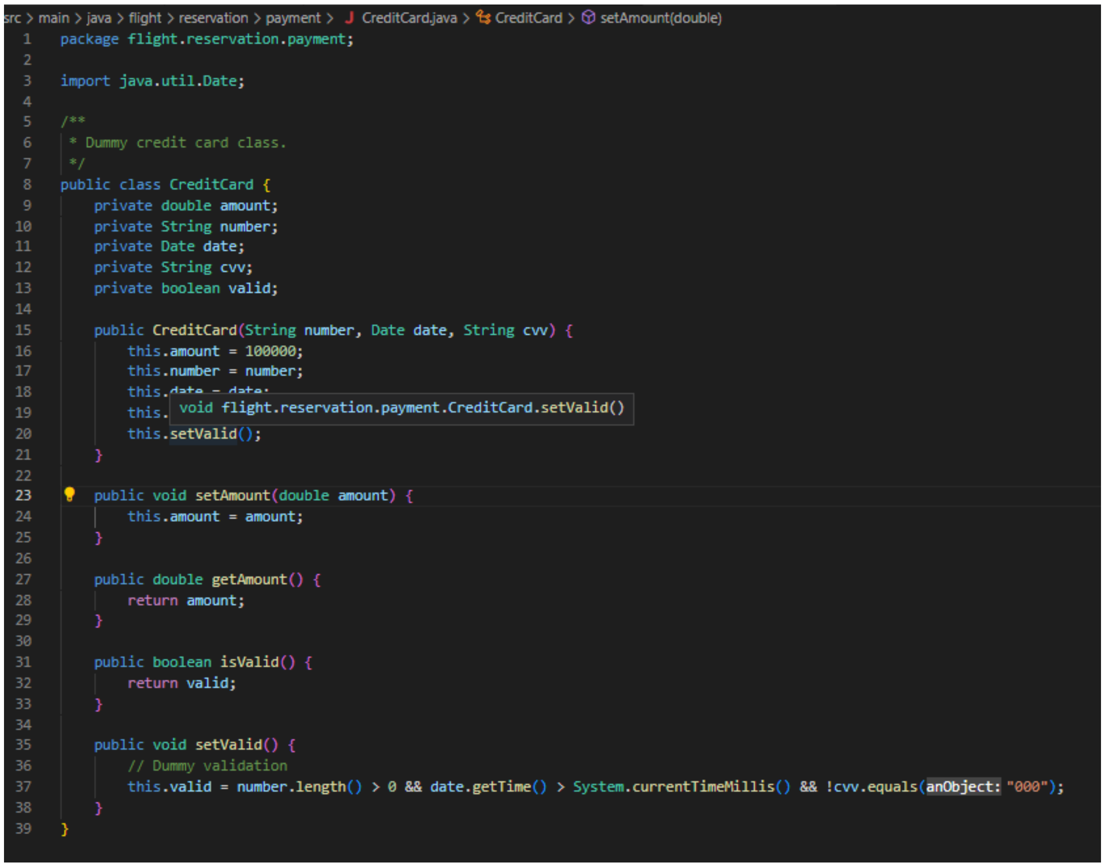
```java
public boolean processOrderWithCreditCardDetail(String number, Date expirationDate, String cvv) throws IllegalStateException {
    CreditCard creditCard = new CreditCard(number, expirationDate, cvv);
    return processOrderWithCreditCard(creditCard);
}

public boolean processOrderWithCreditCard(CreditCard creditCard) throws IllegalStateException {
    if (isClosed()) {
        // Payment is already proceeded
        return true;
    }
    // validate payment information
    if (!cardIsPresentAndValid(creditCard)) {
        throw new IllegalStateException(s: "Payment information is not set or not valid.");
    }
    boolean isPaid = payWithCreditCard(creditCard, this.getPrice());
    if (isPaid) {
        this.setClosed();
    }
    return isPaid;
}

private boolean cardIsPresentAndValid(CreditCard card) {
    return card != null && card.isValid();
}

public boolean processOrderWithPayPal(String email, String password) throws IllegalStateException {
    if (isClosed()) {
        // Payment is already proceeded
        return true;
    }
    // validate payment information
    if (email == null || password == null || !email.equals(Paypal.DATA_BASE.get(password))) {
        throw new IllegalStateException(s: "Payment information is not set or not valid.");
    }
    boolean isPaid = payWithPayPal(email, password, this.getPrice());
    if (isPaid) {
        this.setClosed();
    }
    return isPaid;
}

public boolean payWithCreditCard(CreditCard card, double amount) throws IllegalStateException {
    if (cardIsPresentAndValid(card)) {
        System.out.println("Paying " + getPrice() + " using Credit Card.");
        double remainingAmount = card.getAmount() - getPrice();
        if (remainingAmount < 0) {
            System.out.printf(format: "Card limit reached - Balance: %f%n", remainingAmount);
            throw new IllegalStateException(s: "Card limit reached");
        }
        card.setAmount(remainingAmount);
        return true;
    } else {
        return false;
    }
}

public boolean payWithPayPal(String email, String password, double amount) throws IllegalStateException {
    if (email.equals(Paypal.DATA_BASE.get(password))) {
        System.out.println("Paying " + getPrice() + " using PayPal.");
        return true;
    } else {
        return false;
    }
}
```

Where the creditcard and paypal classes are defined outside



```
Order.java
payment
    CreditCard.java
    Paypal.java
plane
    Helicopter.java
```

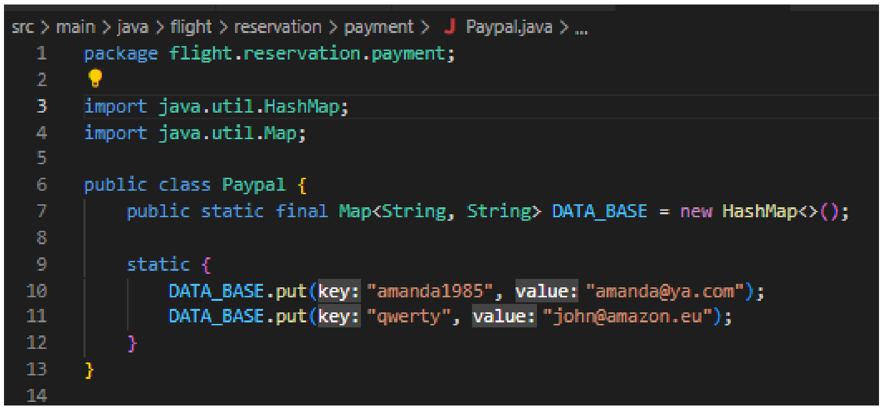
–credit card



src > main > java > flight > reservation > payment > CreditCard.java > CreditCard > setAmount(double)

```java
package flight.reservation.payment;

import java.util.Date;

/**
 * Dummy credit card class.
 */
public class CreditCard {
    private double amount;
    private String number;
    private Date date;
    private String cvv;
    private boolean valid;

    public CreditCard(String number, Date date, String cvv) {
        this.amount = 100000;
        this.number = number;
        this.date = date;
        this. void flight.reservation.payment.CreditCard.setValid()
        this.setValid();
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }

    public boolean isValid() {
        return valid;
    }

    public void setValid() {
        // Dummy validation
        this.valid = number.length() > 0 && date.getTime() > System.currentTimeMillis() && !cvv.equals(anObject: "000");
    }
}
```

-Paypal

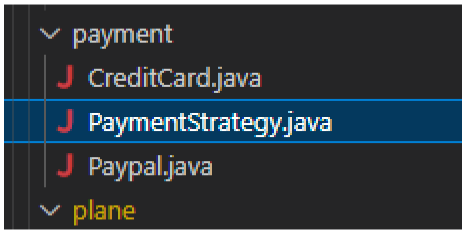


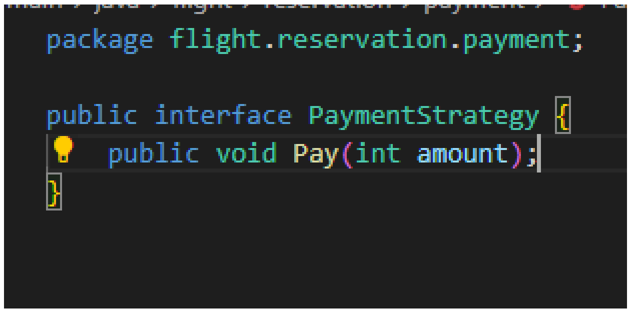
src > main > java > flight > reservation > payment > Paypal.java > ...

```java
package flight.reservation.payment;

import java.util.HashMap;
import java.util.Map;

public class Paypal {
    public static final Map<String, String> DATA_BASE = new HashMap<>();

    static {
        DATA_BASE.put(key: "amanda1985", value: "amanda@ya.com");
        DATA_BASE.put(key: "qwerty", value: "john@amazon.eu");
    }
}
```

A payment strategy could be implemented as follows

Adding a new interface for payment

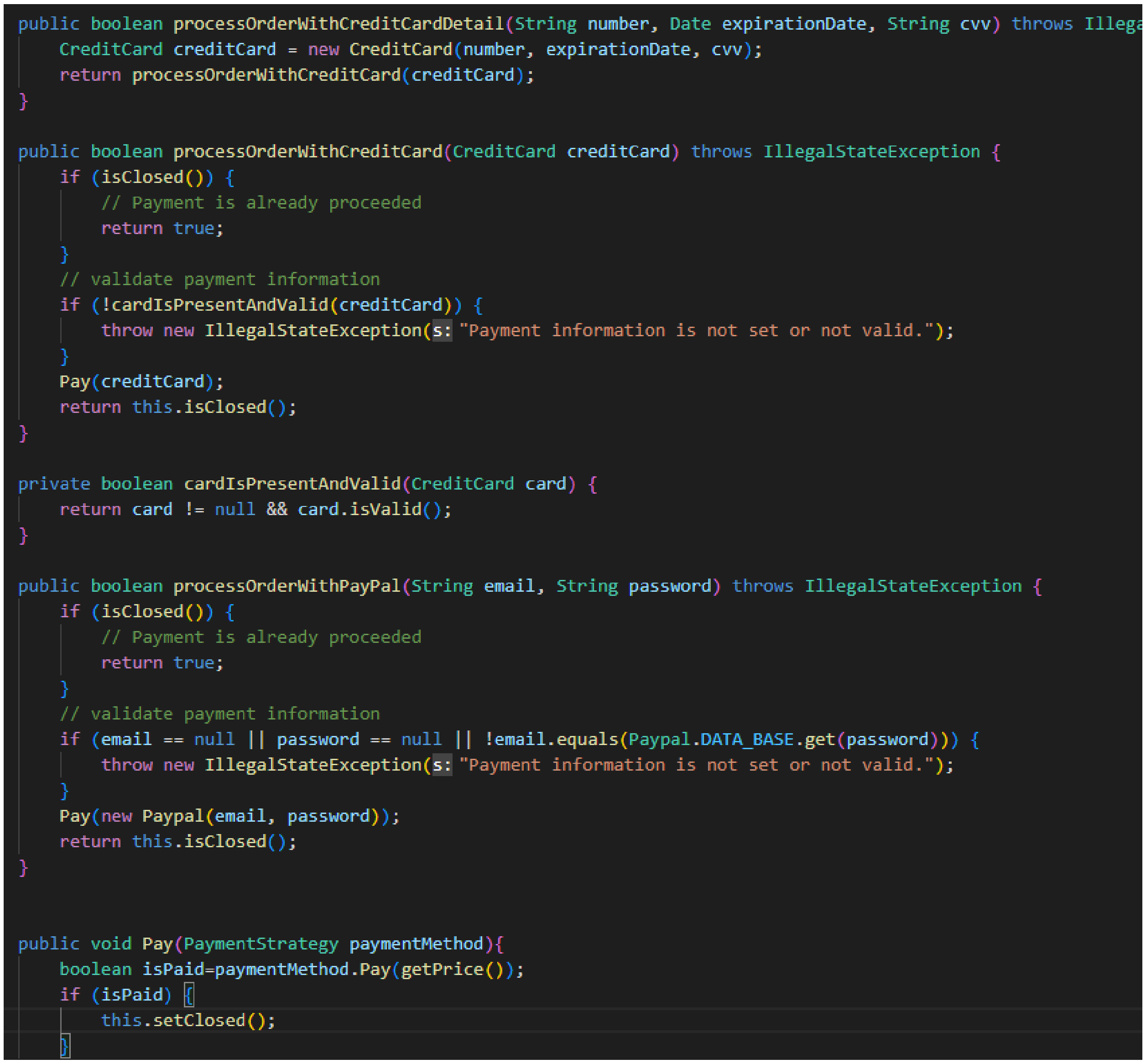




```java
package flight.reservation.payment;

public interface PaymentStrategy {
    public void Pay(int amount);
}
```

Now implement this interface in all payment classes

And the payments could be reduced to as follows



```java
public boolean processOrderWithCreditCardDetail(String number, Date expirationDate, String cvv) throws Illega
    CreditCard creditCard = new CreditCard(number, expirationDate, cvv);
    return processOrderWithCreditCard(creditCard);
}

public boolean processOrderWithCreditCard(CreditCard creditCard) throws IllegalStateException {
    if (isClosed()) {
        // Payment is already proceeded
        return true;
    }
    // validate payment information
    if (!cardIsPresentAndValid(creditCard)) {
        throw new IllegalStateException(s: "Payment information is not set or not valid.");
    }
    Pay(creditCard);
    return this.isClosed();
}

private boolean cardIsPresentAndValid(CreditCard card) {
    return card != null && card.isValid();
}

public boolean processOrderWithPayPal(String email, String password) throws IllegalStateException {
    if (isClosed()) {
        // Payment is already proceeded
        return true;
    }
    // validate payment information
    if (email == null || password == null || !email.equals(Paypal.DATA_BASE.get(password))) {
        throw new IllegalStateException(s: "Payment information is not set or not valid.");
    }
    Pay(new Paypal(email, password));
    return this.isClosed();
}


public void Pay(PaymentStrategy paymentMethod){
    boolean isPaid=paymentMethod.Pay(getPrice());
    if (isPaid) {
        this.setClosed();
    }
```

Builder Design Pattern in Scheduledflight.java

```
public static final class ScheduledFlightBuilder {
       private List<Passenger> passengers;
       private Date departureTime;
       private double currentPrice;

       private ScheduledFlightBuilder() {
       }

       public static ScheduledFlightBuilder
aScheduledFlight(List<Passenger> passengers, Date departureTime, double
currentPrice) {
           return new ScheduledFlightBuilder(passengers, departureTime,
currentPrice);
       }

       public ScheduledFlight build() {
           ScheduledFlight scheduledFlight = new
ScheduledFlight(passengers, departureTime, currentPrice);
           return scheduledFlight;
       }

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

       public void setPassengers(List<Passenger> passengers) {
           this.passengers = passengers;
       }

       public Date getDepartureTime() {
           return departureTime;
       }

       public void setDepartureTime(Date departureTime) {
           this.departureTime = departureTime;
       }

       public static final class ScheduledFlightBuilderBuilder {
           private List<Passenger> passengers;
```

Builder Design Pattern in Scheduledflight.java

```
            private Date departureTime;

            private ScheduledFlightBuilderBuilder() {
            }

            public static ScheduledFlightBuilderBuilder
aScheduledFlightBuilder() {
                return new ScheduledFlightBuilderBuilder();
            }

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

            public ScheduledFlightBuilderBuilder withDepartureTime(Date
departureTime) {
                this.departureTime = departureTime;
                return this;
            }

            public ScheduledFlightBuilder build() {
                ScheduledFlightBuilder scheduledFlightBuilder = new
ScheduledFlightBuilder();
                scheduledFlightBuilder.setPassengers(passengers);
                scheduledFlightBuilder.setDepartureTime(departureTime);
                return scheduledFlightBuilder;
            }
        }
    }
}
```

Creating a builder class ScheduledBuildFlighter, In this case it gives the passengers list, departure time, Arrival time. Which is expected to schedule a Flight.
Builder Interface declares the product construction steps which are common to all types of builders like Passengers list which will be traveling from one place to another place and the departure time, and the Price required to from A to B.

In the modified Code, `class ScheduledFlightBuilder`

Builder Design Pattern in Scheduledflight.java

- Now it takes all the passengers list as an input, and departure time, Current price required to travel.