Design Patterns Activity Report

Chain of Responsibilities

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

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

```
// 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
    }
}
```

```
// 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

```java
// 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";
    }

    // ...
}
```

```java
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