Assignment 2 - ICA4 Reservation System

Builder pattern helps in the creation of complex objects that require a laborious, step-by-step initialization of many fields and nested objects.
Initialization code is usually buried inside a monstrous constructor with lots of parameters or scattered all over the client code.
Creating subclasses to cover all combinations of parameters is not an efficient solution and will require creating a considerable number of subclasses.
Creating a giant constructor in the base class with all possible parameters will eliminate the need for subclasses but will make constructor calls pretty ugly with most parameters being unused.
Builder pattern solves these problems by separating the construction of a complex object from its representation, allowing for step-by-step creation of the object, control over the creation process, and avoiding the need for creating multiple subclasses or monstrous constructors.

A Prototype By

Anusha Nath Roy, 2020101124

Venika Sruthi, 2020101072

Freyam Mehta, 2020101114

Aditya Harikrish, 2020111009

Kunwar Singh, 2019101075

Code: https://github.com/AnushaNathRoy/Reservation-System-Starter

---

Table of Contents

ICA4 - Reservation System

The ICA4 Reservation System is a simple application that provides a ticket reservation service for flights and helicopters. The system consists of entities for various types of air travel, planes, airports, flights and schedules. It also includes customer and passenger entities, classes for the reservation and payment of tickets.

The system allows for the creation of flights from one airport to another with a specific aircraft, scheduling flights for a specific date and time, booking a reservation for a customer on a specific scheduled flight, processing the payment (by credit card or PayPal) and closing or finalizing the order.

Our task here is to refactor the existing codebase using various design patterns learnt in class. We have discussed our methods in the sections below.

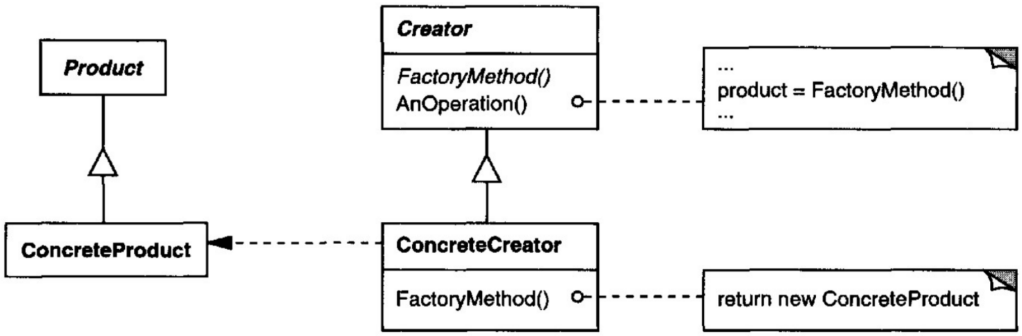
Refactoring the Code

Factory Method 🏭

Problem ☹️

- The system lacks flexibility.
- Extending the current system of creating a `plane` is tedious.
- The existing code has too many `switch` cases, i.e., the code smell: `Switch Statements`.

Structure 👷



Creator: `PlaneFactoryInterface`

ConcreteCreator: `PlaneFactory`

Product: `Plane`

ConcreteProduct: `Helicopter` , `PassengerDrone` , `PassengerPlane`

FactoryMethod : `createPlane()`

The pattern defines an interface for creating an object called `PlaneFactoryInterface` and the concrete creator that implements it is called PlaneFactory.

The product that is created using this pattern is called `Plane`, which can have different concrete implementations: `Helicopter`, `PassengerDrone`, and `PassengerPlane`.

The Factory Method itself is defined in the `createPlane()` method, which is responsible for creating the Plane object. By using this pattern, the system can be more flexible in terms of creating objects and managing their lifecycle.

file structure:

```
plane:
- Helicopter.java
- PassengerDrone.java
- PassengerPlane.java
- PassengerPlaneFactory.java
- Plane.java
- PlaneFactory.java
- PlaneFactoryInterface.java
```

Refactored Code 👩‍💻

💡 We have added an additional method `getAirHostessRequired()` to show how easily extensible this refactored code is to add different types of attributes in the future.

```
// Plane.java

package flight.reservation.plane;

public interface Plane {
    String getModel();
```

```
    int getPassengerCapacity();
    int getCrewCapacity();
    int getAirHostessRequired();
}
```

```
// Helicopter.java

package flight.reservation.plane;
public class Helicopter implements Plane {
    private final String model;
    private final int passengerCapacity;
    private final int crewCapacity = 1;
    private final int airHostessRequired;

    public Helicopter(String model) {
        this.model = model;
        if (model.equals("H1")) {
            passengerCapacity = 4;
            airHostessRequired = 1;
        } else if (model.equals("H2")) {
            passengerCapacity = 6;
            airHostessRequired = 2;
        } else {
            throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }

    public String getModel() {
        return model;
    }

    public int getPassengerCapacity() {
        return passengerCapacity;
    }

    public int getCrewCapacity() {
        return crewCapacity;
    }

    public int getAirHostessRequired() {
        return airHostessRequired;
    }
}
```

```
// PassengerDrone.java

package flight.reservation.plane;
public class PassengerDrone implements Plane {
    private final String model;
    private final int passengerCapacity = 2;
    private final int crewCapacity = 0;
    private final int airHostessRequired = 0;

    public PassengerDrone(String model) {
        if (model.equals("HypaHype")) {
            this.model = model;
        } else {
            throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }

    public String getModel() {
        return model;
    }

    public int getPassengerCapacity() {
        return passengerCapacity;
    }

    public int getAirHostessRequired() {
        return airHostessRequired;
    }

    public int getCrewCapacity() {
        return crewCapacity;
    }
```

```
}
```

```
// PassengerPlane.java

package flight.reservation.plane;

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

    public PassengerPlane(String model) {
        this.model = model;
        switch (model) {
            case "A380":
                passengerCapacity = 500;
                crewCapacity = 42;
                airHostessRequired = 10;
                break;
            case "A350":
                passengerCapacity = 320;
                crewCapacity = 40;
                airHostessRequired = 7;
                break;
            case "Embraer 190":
                passengerCapacity = 25;
                crewCapacity = 5;
                airHostessRequired = 2;
                break;
            case "Antonov AN2":
                passengerCapacity = 15;
                crewCapacity = 3;
                airHostessRequired = 1;
                break;
            default:
                throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }

    public String getModel() {
        return model;
    }

    public int getPassengerCapacity() {
        return passengerCapacity;
    }

    public int getCrewCapacity() {
        return crewCapacity;
    }

    public int getAirHostessRequired() {
        return airHostessRequired;
    }
}
```

```
// PlaneFactoryInterface.java

package flight.reservation.plane;

public interface PlaneFactoryInterface {
    Plane createPlane(String model);
}
```

```
// PlaneFactory.java

package flight.reservation.plane;

public class PlaneFactory implements PlaneFactoryInterface{
```

```
    public Plane createPlane(String model) {
        switch (model) {
            case "A380":
                return new PassengerPlane(model);
            case "A350":
                return new PassengerPlane(model);
            case "Embraer 190":
                return new PassengerPlane(model);
            case "Antonov AN2":
                return new PassengerPlane(model);
            case "H1":
                return new Helicopter(model);
            case "H2":
                return new Helicopter(model);
            case "HypaHype":
                return new PassengerDrone(model);
            default:
                throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}
```

```
// Runner.java

PlaneFactory planeFactory = (PlaneFactory) new PlaneFactory()
planeFactory.createPlane("A380")
```

Benefits of the Refactored Code 💡

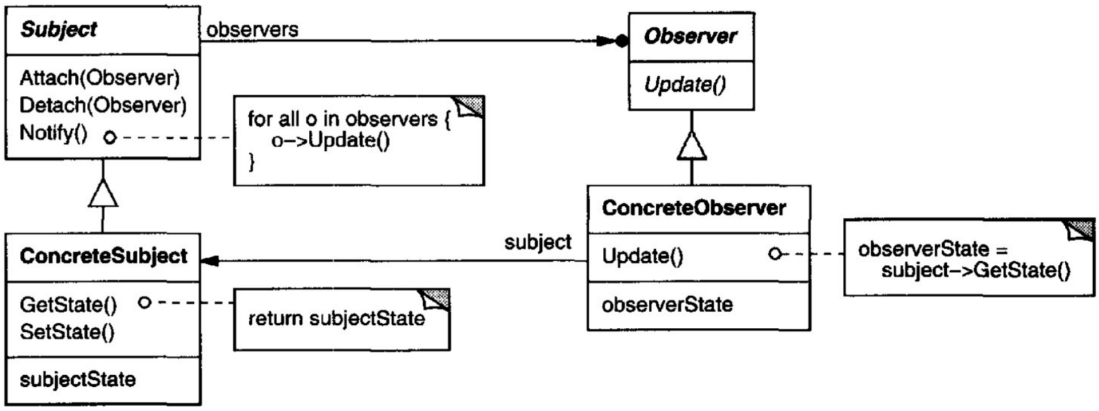
- Single Responsibility Principle. The code for `plane` creation is separate from the actual object `plane`. This will help in maintaining code.
- As seen above, we could easily extend the code due to this decoupling by adding a new `getAirHostessRequired()` very easily.
- Leads to less confusing `switch` cases spread across various functions. Now only one `switch` case exists in the factory method to create the different `planes`.
- Open/Closed Principle. We can add new types of `plane` such as `CargoPlane` easily without breaking existing code.

Observer

Problem ☹️

- Currently, there is no communication between `Customer` and `Order` . So a customer class cannot do anything upon change in order or the process to integrate such a function will be tedious without the use of the `Observer` pattern.
  Eg: doing an action by the customer on the completion of the processing of an order like updating seats and miles earned by the customer.

Structure 👷



Subject: `Order` . It has functions: `addObserver`, `notifyObservers` , `removeObserver`

ConcreteSubject: `FlightOrder`

Observer: `OrderObserver` . It has a `update()` method.

Concrete Observer:

- `CustomerObserver` . This observer/listener notifies its subscribers when their order is processed, i.e., when the `setClosed()` method is called upon completion of the order all the observers are notified.
- `OrderPriceObserver`. This type of subscription will allow the listeners interested in getting notified about price changes in tickets. The observer exists but its implementation is not provided by us can be integrated easily upon extension.

> 💡 The **Client** (who creates publisher and subscriber objects separately and then registers subscribers for publisher updates) is the `Customer` here. Customer creates the `Order` and assigns the particular order the subscriptions based on their preference.

file structure:

```
order:
- Order.java
- FlightOrder.java
- OrderObserver.java
- CustomerOrder.java
- OrderPriceObserver.java
```

Refactored Code 👩‍💻

> 💡 Our code currently offers two types of subscriptions:
> - `Customer` subscription to be notified when order is processed completely with payment.
> - `OrderPrice` subscription to be notified if the requested flight order prices change.

```
//Order.java

package flight.reservation.order;

import flight.reservation.Customer;
```

```
import flight.reservation.Passenger;

import java.util.ArrayList;
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 List<OrderObserver> observers = new ArrayList<>();

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

    public UUID getId() {
        return id;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

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

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

    public boolean isClosed() {
        return isClosed;
    }

    public void setClosed() {
        isClosed = true;
        notifyObservers();
    }

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

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

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

```
// FlightOrder.java

package flight.reservation.order;

import flight.reservation.Customer;
import flight.reservation.flight.ScheduledFlight;
import flight.reservation.payment.CreditCard;
import flight.reservation.payment.Paypal;

import java.util.Arrays;
```

```
import java.util.Date;
import java.util.List;

public class FlightOrder extends Order {
    private final List<ScheduledFlight> flights;
    static List<String> noFlyList = Arrays.asList("Peter", "Johannes");

    public FlightOrder(List<ScheduledFlight> flights) {
        this.flights = flights;
    }

    public static List<String> getNoFlyList() {
        return noFlyList;
    }

    public List<ScheduledFlight> getScheduledFlights() {
        return flights;
    }

    private boolean isOrderValid(Customer customer, List<String> passengerNames, List<ScheduledFlight> flights) {
        boolean valid = true;
        valid = valid && !noFlyList.contains(customer.getName());
        valid = valid && passengerNames.stream().noneMatch(passenger -> noFlyList.contains(passenger));
        valid = valid && flights.stream().allMatch(scheduledFlight -> {
            try {
                return scheduledFlight.getAvailableCapacity() >= passengerNames.size();
            } catch (NoSuchFieldException e) {
                e.printStackTrace();
                return false;
            }
        });
        return valid;
    }

    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("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("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("Card limit reached - Balance: %f%n", remainingAmount);
                throw new IllegalStateException("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;
        }
    }
}
```

```
// OrderObserver.java

package flight.reservation.order;
public interface OrderObserver {
    void update(Order order);
}
```

```
// CustomerObserver.java
package flight.reservation.order;
import flight.reservation.Customer;
public class CustomerObserver implements OrderObserver {
    Customer customer;

    @Override
    public void update(Order order) {
        System.out.println("Order " + order.getId() + " process is closed.");
    }
}
```

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

public class OrderPriceObserver implements OrderObserver {

    @Override
    public void update(Order order) {
        System.out.println("Order price updated: " + order.getPrice());
    }
}
```

Benefits of the Refactored Code 💡

- This type of pattern gives the `Customer` to select what subscription or notification they would like to receive
- A relationship between `Customer` and `Order` will be established during runtime which will help the classes to do actions that depend on each other much easily and provide extra functionality.
- Open/Closed Principle. A new subscription can be easily added like subscription `noFlyListUpdated` which will inform `Customer` class on change of the `noFlyList` .

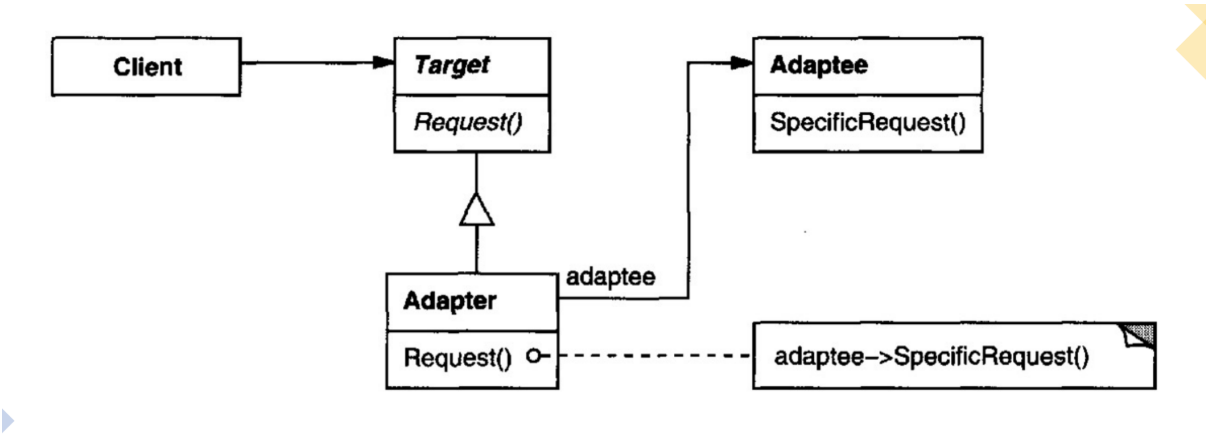
Adapter

Problem ☹️

It seems that the provided code defines three different classes (`Helicopter` , `PassengerDrone` , and `PassengerPlane`) that implement the same `Plane` interface, but each class has its own unique way of defining its properties (such as passenger capacity, crew capacity, and air hostess requirement).

Suppose, the `ReservationSystem` needs details of `Plane` as `Strings` and as an additional method

Structure 👷



- **Target:** The `FlightDetails` interface defines the target interface that the `ReservationSystem` class will use to interact with planes.
- **Client:** The `ReservationSystem` is the client that interacts with the planes through the `FlightDetails` interface.
- **Adaptee:** The `Plane` interface and its implementations (`Helicopter` , `PassengerDrone` , and `PassengerPlane`) are the adaptees that provide the original interface that the `ReservationSystem` class cannot use directly.
- **Adapter:** The `PlaneToFlightDetails` class is the adapter that adapts the `Plane` interface to the `FlightDetails` interface, allowing the `ReservationSystem` class to work with the different types of planes.

Refactored Code 👩‍💻

> 💡 We are assuming that the reservation system needs data in `String` format and a `totalCount` .

```
// FlightDetails.java

package flight.reservation.flight;

public interface FlightDetails {
    public String getFlightModel();
    public String getPassengerCount();
    public String getCrewCount();
    public String getAirHostessCount();
    public String totalCount();
}
```

```
// PlaneToFlightDetails.java

package flight.reservation.flight;
import flight.reservation.plane.Plane;

public class PlaneToFlightDetails implements FlightDetails {
    private final Plane plane;

    public PlaneToFlightDetails(Plane plane) {
```

```
        this.plane = plane;
    }

    public String getFlightModel() {
        return plane.getModel();
    }

    public String getPassengerCount() {
        return String.valueOf(plane.getPassengerCapacity());
    }

    public String getCrewCount() {
        return String.valueOf(plane.getCrewCapacity());
    }

    public String getAirHostessCount() {
        return String.valueOf(plane.getAirHostessRequired());
    }

    public String totalCount() {
        return String.valueOf(plane.getPassengerCapacity() + plane.getCrewCapacity() + plane.getAirHostessRequired());
    }

}
```

We can then use this adapter class to adapt each `Plane` object to the `FlightDetails` interface. For example:

```
Plane helicopter = new Helicopter("H1");
FlightDetails helicopterFlightDetails = new PlaneToFlightDetails(helicopter);
```

Similarly, we can create flight objects for `PassengerDrone` and `PassengerPlane` as well by using the `PlaneToFlightDetails` **adapter** class.

Benefits of the Refactored Code 💡

If the adapter pattern is used to refactor the code, it brings the following benefits:

- Loose coupling: The adapter pattern decouples the `ReservationSystem` class from the implementation details of the `Plane` interface. It creates a new interface that the `ReservationSystem` class can work with, and provides a way to adapt the different types of planes to this interface. This loose coupling between the classes makes the code more flexible and easier to maintain.
- Better encapsulation: The adapter pattern allows each type of plane to encapsulate its own implementation details, and exposes a standard interface to the `ReservationSystem` class. This makes the code more modular, as each type of plane can be developed and tested independently of the `ReservationSystem` class.
- Extensibility: The adapter pattern allows for easy extensibility of the system. If a new type of plane is added, it can be adapted to the standard interface using a new adapter, without the need to modify the `ReservationSystem` or any of the existing adapters.
- Reusability: The adapter pattern promotes the reusability of code. Adapters can be reused in different parts of the system that require the same interface, which saves development time and effort.
- Maintainability: The adapter pattern makes the code easier to maintain, as changes to one type of plane do not affect the rest of the system. This reduces the risk of introducing bugs and makes it easier to debug and troubleshoot the code.

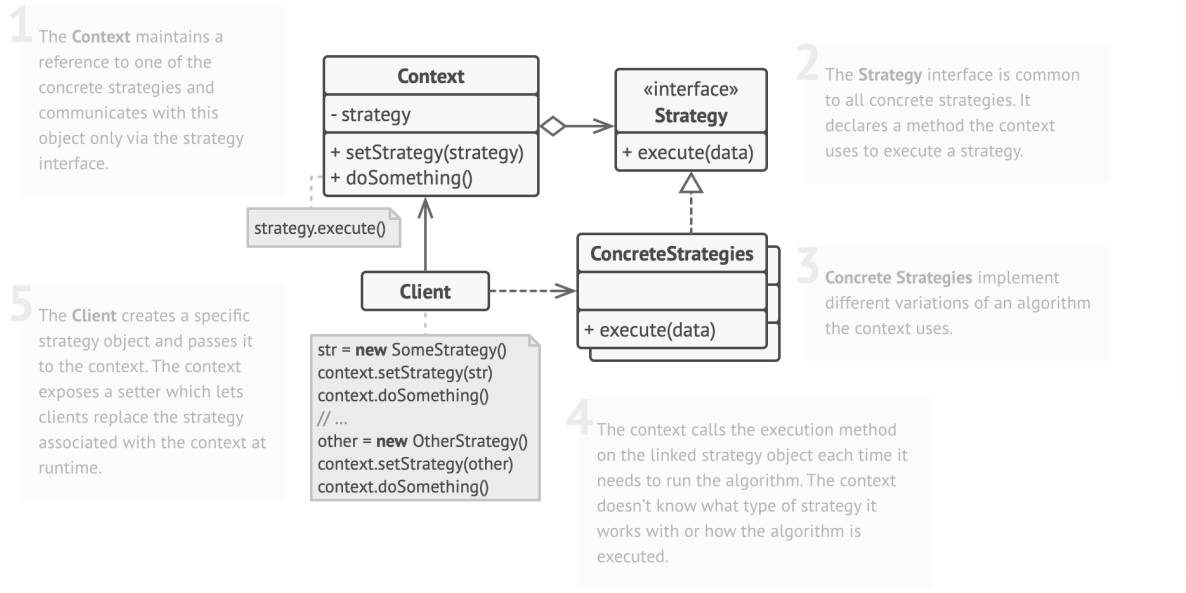
Strategy

Problem ☹️

- It was difficult to modify or extend the `Payment` functionality because payment logic was firmly coupled with the `FlightOrder` class.

- The `FlightOrder` class contained multiple payment methods (credit card and PayPal) which made the code more difficult to comprehend and maintain.
- The validation logic for each payment method was combined into the same methods, making the code more difficult to modify and more complicated. - `Large Class` and `Divergent Change` code smell.

Structure 👷



Context: `FlightOrder` . This class contains the `setPaymentStrategy()` method which sets strategy.

Strategy: `PaymentStrategy` . The execute function here is `pay(amount)` method.

ConcreteStrategies: `CreditCardStrategy` and `PaypalStrategy` which implement the respective different strategies of payment.

Client: Here the client is the `Customer` class. It acts as a client through the following way:

```
public FlightOrder createOrder(List<String> passengerNames, List<ScheduledFlight> flights, double price) {
       // existing code ..
        FlightOrder order = new FlightOrder(flights);
        order.setCustomer(this);
        order.setPrice(price);
        order.setPaymentStrategy(paymentStrategy);

        // existing code ..
        return order;
    }
```

We can implement this during runtime by:

```
FlightOrder order = customer.createOrder(Arrays.asList("Max"), Arrays.asList(scheduledFlight), 100);
order.setPaymentStrategy(new CreditCardStrategy(creditCard));
order.processOrder();
```

file structure:

```
payment:
- CreditCard.java
- CreditCardStrategy.java
- PaymentStrategy.java
- Paypal.java
- PaypalStrategy. java
```

Refactored Code 👩‍💻

```
// PaymentStrategy.java
package flight.reservation.payment;

public interface PaymentStrategy {
    boolean pay(double amount);
}
```

💡 Each strategy has its own validation.

```
// CreditCardStrategy.java
package flight.reservation.payment;

import java.util.Date;

public class CreditCardStrategy implements PaymentStrategy {

    CreditCard creditCard;

    public CreditCardStrategy(String number, Date expirationDate, String cvv) {
        this.creditCard = new CreditCard(number, expirationDate, cvv);
    }

    private boolean cardIsPresentAndValid() {
        return this.creditCard != null && this.creditCard.isValid();
    }

    public CreditCardStrategy(CreditCard creditCard) {
        this.creditCard = creditCard;
    }

    @Override
    public boolean pay(double amount) throws IllegalStateException {

        if (!cardIsPresentAndValid()) {
            throw new IllegalStateException("Payment information is not set or not valid.");
        }

        if (creditCard.isValid()) {
            System.out.println("Paying " + amount + " using Credit Card.");
            double remainingAmount = creditCard.getAmount() - amount;
            if (remainingAmount < 0) {
                System.out.printf("Card limit reached - Balance: %f%n", remainingAmount);
                throw new IllegalStateException("Card limit reached");
            }
            creditCard.setAmount(remainingAmount);
            return true;
        } else {
            throw new IllegalStateException("Credit card is not valid");
        }
    }
}
```

```
// PaypalStrategy.java

package flight.reservation.payment;

public class PaypalStrategy implements PaymentStrategy {
    private String email;
    private String password;
    Paypal paypal;

    public PaypalStrategy(String email, String password) {
        this.email = email;
        this.password = password;
        this.paypal = new Paypal();
```

```
    }

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

```
// CreditCard.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.cvv = cvv;
        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("000");
    }
}
```

```
// Paypal.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("amanda1985", "amanda@ya.com");
        DATA_BASE.put("qwerty", "john@amazon.eu");
    }
}
```

💡 Look at how much neater and cleaner the `FlightOrder` class looks. The payment strategy is decoupled from this class.

```
// FlightOrder.java

package flight.reservation.order;

import flight.reservation.Customer;
import flight.reservation.flight.ScheduledFlight;
import flight.reservation.payment.PaymentStrategy;

import java.util.Arrays;
import java.util.Date;
import java.util.List;

public class FlightOrder extends Order {

    private PaymentStrategy paymentStrategy;

   // existing methods

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }


   // existing methods

    public boolean processOrder() throws IllegalStateException {
        if (paymentStrategy == null) {
            throw new IllegalStateException("Payment strategy is not set");
        }

        if(paymentStrategy.pay(this.getPrice())) {
           setClosed();
           return true;
        }

        return false;
    }
}
```

Benefits of the Refactored Code 💡

- The Strategy Pattern enabled the `Payment` functionality to be encapsulated in distinct `PaymentStrategy` classes, which were readily modifiable and extendable without influencing the `FlightOrder` class. Eg: A new payment strategy like `GooglePay` can be easily added.
- `FlightOrder` was refactored to use a PaymentStrategy interface, allowing the payment logic to be shifted in and out at runtime. Eg: A customer could change their payment strategy anytime during runtime.
- Each `PaymentStrategy` class was responsible for its own validation logic, which simplified the `FlightOrder` class and made it more amenable to modification and extension.
- Reduction of code complexity and enhancing its maintainability and extensibility.

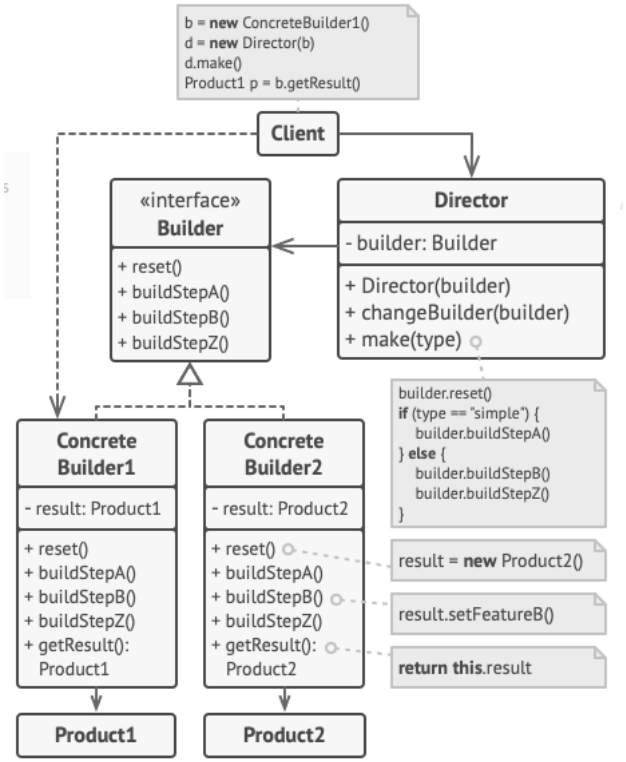
Builder Pattern

Problem ☹️

- `PassengerPlane` class has several attributes that can be set at construction time, such as the model type, passenger capacity, crew capacity, and air hostess requirement.
- Using a builder pattern for this class can make it more flexible and easier to use by allowing users to set only the attributes they need, and to do so in a clear and readable way.
- Additionally, the `PassengerPlane` class has a large number of possible attribute values, so using a builder pattern can help avoid the need for a large number of constructors with different parameter lists.

- Builder pattern helps in the creation of complex objects that require a laborious, step-by-step initialization of many fields and nested objects.
- Initialization code is usually buried inside a monstrous constructor with lots of parameters or scattered all over the client code.
- Creating subclasses to cover all combinations of parameters is not an efficient solution and will require creating a considerable number of subclasses.
- Creating a giant constructor in the base class with all possible parameters will eliminate the need for subclasses but will make constructor calls pretty ugly with most parameters being unused.
- Builder pattern solves these problems by separating the construction of a complex object from its representation, allowing for step-by-step creation of the object, control over the creation process, and avoiding the need for creating multiple subclasses or monstrous constructors.

Structure 👷



Director: `PlaneFactory`

ConcreteBuilder: `Builder`

Product: `PassengerPlane`

The `PlaneFactory` class can be considered a director for the `PassengerPlane` class, as it creates and returns instances of `PassengerPlane` using the builder pattern.

The `Builder` class in `PassengerPlane` can be considered the concrete builder, responsible for building the `PassengerPlane` object step-by-step.

The `PassengerPlane` class itself can be considered the product that the builder constructs.

Refactored Code 👩‍💻

💡 A `Builder` class has been added to PassengerPlane.

```
package flight.reservation.plane;

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

    private PassengerPlane(Builder builder) {
        this.model = builder.model;
        this.passengerCapacity = builder.passengerCapacity;
        this.crewCapacity = builder.crewCapacity;
        this.airHostessRequired = builder.airHostessRequired;
    }

    public String getModel() {
        return model;
    }

    public int getPassengerCapacity() {
        return passengerCapacity;
    }

    public int getCrewCapacity() {
        return crewCapacity;
    }

    public int getAirHostessRequired() {
        return airHostessRequired;
    }

    public static class Builder {
        private final String model;
        private int passengerCapacity;
        private int crewCapacity;
        private int airHostessRequired;

        public Builder(String model) {
            this.model = model;
            switch (model) {
                case "A380":
                    passengerCapacity = 500;
                    crewCapacity = 42;
                    airHostessRequired = 10;
                    break;
                case "A350":
                    passengerCapacity = 320;
                    crewCapacity = 40;
                    airHostessRequired = 7;
                    break;
                case "Embraer 190":
                    passengerCapacity = 25;
                    crewCapacity = 5;
                    airHostessRequired = 2;
                    break;
                case "Antonov AN2":
                    passengerCapacity = 15;
                    crewCapacity = 3;
                    airHostessRequired = 1;
                    break;
                default:
                    throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
            }
        }

        public Builder passengerCapacity(int passengerCapacity) {
            this.passengerCapacity = passengerCapacity;
            return this;
        }

        public Builder crewCapacity(int crewCapacity) {
            this.crewCapacity = crewCapacity;
            return this;
        }

        public Builder airHostessRequired(int airHostessRequired) {
            this.airHostessRequired = airHostessRequired;
            return this;
        }
```

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

To create an instance of `PassengerPlane` using the builder pattern, you would do something like this:

```
PassengerPlane plane = new PassengerPlane.Builder("A380")
.passengerCapacity(550)
.crewCapacity(44)
.airHostessRequired(12)
.build();
```

Benefits of the Refactored Code 💡

1. **Improved readability:** The use of the builder pattern makes the code more readable and easier to understand.
2. **Simplified object creation:** The builder pattern simplifies the process of creating a `PassengerPlane` object by breaking down the process into smaller, more manageable steps.
3. **Flexibility:** The builder pattern allows for the creation of different variations of the `PassengerPlane` object without having to create new classes or methods.
4. **Easy maintenance:** With the builder pattern, it is easy to maintain and modify the `PassengerPlane` class as changes can be made to the builder class without affecting the `PassengerPlane` class.
5. Improved readability: The use of the builder pattern makes the code more readable and easier to understand.
6. Simplified object creation: The builder pattern simplifies the process of creating a PassengerPlane object by breaking down the process into smaller, more manageable steps.
7. Flexibility: The builder pattern allows for the creation of different variations of the PassengerPlane object without having to create new classes or methods.
8. Easy maintenance: With the builder pattern, it is easy to maintain and modify the PassengerPlane class as changes can be made to the builder class without affecting the PassengerPlane class.

Composite Pattern

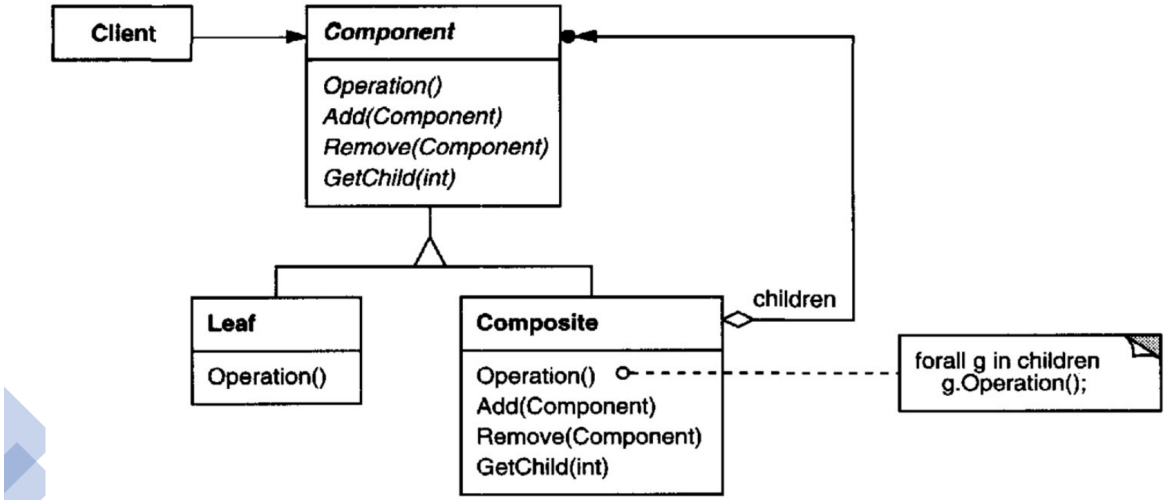
The composite pattern is utilised in situations where the boxes-and-products analogy holds: that is, situations that demand recursive iteration through a “box” and its contents. The recursion stops when it reaches a product, which cannot contain other products.

The composite pattern tackles this issue by instead creating a common interface for both, the metaphorical “box” and the “product”. Since they are treated as instances of the same interface, now recursive iteration does not require checking the data type of the current vertex (if one considers the recursive iteration to be a tree whose root is the “biggest box”, where each node is a box or a product). The leaves of such a graph are the products.

Problem ☹️

- The code base by itself doesn’t have any instance of recursive iteration through a class’s contents.
- So, we created a class called CompositeOrder to implement the composite pattern. The idea is that this class can account for the entire order (”box”) while its components can be different sub-charges like luggage costs, meals, etc.
- The class LeafOrder, as the name suggests, represents those orders that cannot have sub-orders (”product”).
- Both of these classes implement the OrderDetailsComponent interface.

Structure 👷



- **Composite (”box”):** CompositeOrder
- **Leaf (”product”):** LeafOrder
- **Common Interface:** OrderDetailsComponent

Refactored Code 👩‍💻

> 💡 We are adding an extra functionality of a composite order which can be a combination of leaf orders. Eg: A leaf order can be a food item, seat spacing, set booking, or extra luggage. A composite order can then add all the prices of the individual leaf orders and give the total price.

```
// OrderDetailsComponent.java
package flight.reservation.order;
import java.util.ArrayList;
import java.util.List;

public interface OrderDetailsComponent {
    void add(OrderDetailsComponent orderComponent);
    void remove(OrderDetailsComponent orderComponent);
    List<OrderDetailsComponent> getChildren();
    double getPrice();
}
```

```
// CompositeOrderDetails.java
package flight.reservation.order;
import java.util.ArrayList;
import java.util.List;

public class CompositeOrder implements OrderDetailsComponent {

    private List<OrderDetailsComponent> orderDetailsComponents = new ArrayList<>();

    public void add(OrderDetailsComponent orderDetailsComponent) {
        orderDetailsComponents.add(orderDetailsComponent);
    }

    public void remove(OrderDetailsComponent orderDetailsComponent) {
        orderDetailsComponents.remove(orderDetailsComponent);
    }

    public List<OrderDetailsComponent> getChildren() {
```

```
        return orderDetailsComponents;
    }

    public double getPrice() {
        double price = 0;
        for (OrderDetailsComponent OrderDetailsComponent : orderDetailsComponents) {
            price += OrderDetailsComponent.getPrice();
        }
        return price;
    }
}
```

```
// LeafOrder.java
package flight.reservation.order;
import java.util.ArrayList;
import java.util.List;

public class LeafOrder implements OrderDetailsComponent {

   public double price;

    public LeafOrder(double price) {
        this.price = price;
    }

    public void add(OrderDetailsComponent orderDetailsComponent) {
        throw new UnsupportedOperationException();
    }

    public void remove(OrderDetailsComponent orderDetailsComponent) {
        throw new UnsupportedOperationException();
    }

    public List<OrderDetailsComponent> getChildren() {
        throw new UnsupportedOperationException();
    }

    public double getPrice() {
        return this.price;
    }
}
```

Benefits of the Refactored Code 💡

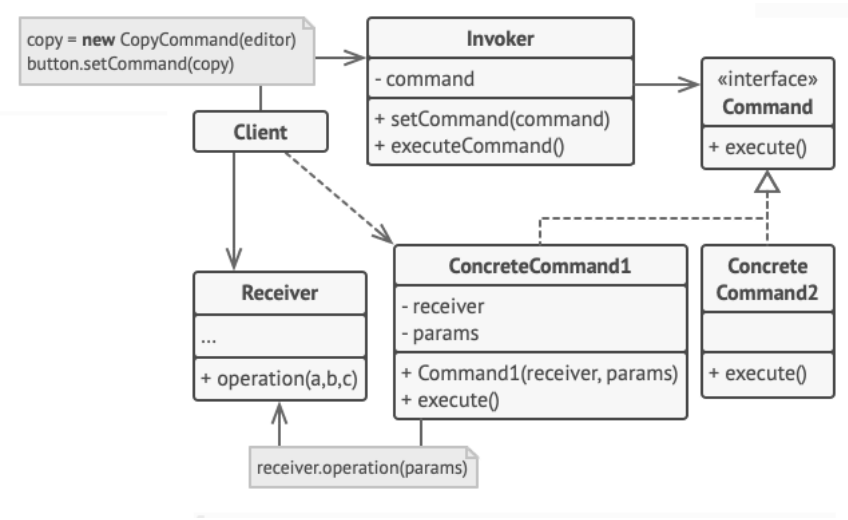
1. **Greater extensibility**. The composite pattern ensures greater extensibility as one does not have to code up the functionality for the composites and the leaves on a case-by-case basis anymore; the involved classes have the functions and we take advantage of polymorphism instead. This allows for easier extensibility. For instance, if we wanted to add a new charge (say convenience fee) all we need to do is create its object and add it to a CompositeOrder's list of OrderDetailsComponent objects.
2. **Fulfills the open/closed principle.** Modification to the nodes in the recursive tree is easier now that polymorphism and a common interface for all the nodes grants easier extensibility. At the same time, it is closed for modification as new functionality belongs to new code (see the above example of convenience fee) and not modifications to the existing code.

Command

Problem ☹️

- The original implementation provides basic functionality for scheduling, removing, and searching flights.
- Without the Command Pattern, it would be challenging to add additional functionality to the `Schedule` class, as any new methods would need to be added directly to the class itself. This could result in the class becoming more complex and harder to maintain over time, particularly as more methods are added.
- By using Command Patterns, we can add new functionality to the `Schedule` class in a more modular and maintainable way. Commands can be created as separate classes that encapsulate a particular behavior, and these commands can be executed by the `Schedule` class when needed.

Structure 👷



Invoker: The `Schedule` class itself acts as the invoker in this implementation. It receives commands and executes them when requested.

Receiver: The receiver in this case is the `scheduledFlights` list, which is where the scheduled flights are stored. The receiver is the object that the command is ultimately acting upon.

Command: The `Command` interface defines the common operations that all commands must implement, such as execute() and undo(). This interface acts as the command abstraction in this implementation.

Concrete Command: The `ScheduleFlightCommand`, `RemoveFlightCommand`, and `ClearScheduleCommand` classes are the concrete command classes in this implementation. Each of these classes implements the Command interface and defines a specific behavior that the `Schedule` class can execute.

The Command Pattern provides a way to encapsulate a request as an object, which allows the request to be treated as a first-class object in the system. This approach helps to decouple the invoker from the receiver, which can make the system more modular and easier to maintain over time.

Refactored Code 👩‍💻

💡 The following changes were made:

1. Created a concrete `ScheduleCommand` class that implements the `Command` interface and has a reference to a `Schedule` instance.
2. Modified the `scheduleFlight`, `removeFlight`, `removeScheduledFlight`, and `clear` methods in the `Schedule` class to create instances of the `ScheduleCommand` class and add them to a list of commands.
3. Added a new `executeCommands` method to the `Schedule` class that executes all of the commands in the list.
4. Removed the code that executed commands from the `scheduleFlight`, `removeFlight`, `removeScheduledFlight`, and `clear` methods.

```
package flight.reservation.flight;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Schedule {

    private List<ScheduledFlight> scheduledFlights;
    private List<Command> commandHistory;
```

```
    public Schedule() {
        scheduledFlights = new ArrayList<>();
        commandHistory = new ArrayList<>();
    }

    public List<ScheduledFlight> getScheduledFlights() {
        return scheduledFlights;
    }

    public void scheduleFlight(Flight flight, Date date) {
        ScheduledFlight scheduledFlight = new ScheduledFlight(flight.getNumber(), flight.getDeparture(), flight.getArrival(), flight.g
        Command command = new ScheduleFlightCommand(this, scheduledFlight);
        executeCommand(command);
    }

    public void removeFlight(Flight flight) {
        List<ScheduledFlight> tbr = new ArrayList<>();
        for (ScheduledFlight scheduledFlight : scheduledFlights) {
            if (scheduledFlight == flight ||
                    (flight.getArrival() == scheduledFlight.getArrival() &&
                            flight.getDeparture() == scheduledFlight.getDeparture() &&
                            flight.getNumber() == scheduledFlight.getNumber())) {
                tbr.add(scheduledFlight);
            }
        }
        Command command = new RemoveFlightCommand(this, tbr);
        executeCommand(command);
    }

    public void removeScheduledFlight(ScheduledFlight flight) {
        Command command = new RemoveScheduledFlightCommand(this, flight);
        executeCommand(command);
    }

    public ScheduledFlight searchScheduledFlight(int flightNumber) {
        return scheduledFlights.stream()
                .filter(f -> f.getNumber() == flightNumber)
                .findFirst()
                .orElse(null);
    }

    public void clear() {
        Command command = new ClearScheduleCommand(this);
        executeCommand(command);
    }

    public void executeCommand(Command command) {
        command.execute();
        commandHistory.add(command);
    }

    public void undoLastCommand() {
        if (commandHistory.size() > 0) {
            Command lastCommand = commandHistory.remove(commandHistory.size() - 1);
            lastCommand.undo();
        }
    }
}

interface Command {
    void execute();
    void undo();
}

class ScheduleFlightCommand implements Command {
    private Schedule schedule;
    private ScheduledFlight scheduledFlight;

    public ScheduleFlightCommand(Schedule schedule, ScheduledFlight scheduledFlight) {
        this.schedule = schedule;
        this.scheduledFlight = scheduledFlight;
    }

    public void execute() {
        schedule.getScheduledFlights().add(scheduledFlight);
    }

    public void undo() {
        schedule.getScheduledFlights().remove(scheduledFlight);
    }
}

class RemoveFlightCommand implements Command {
    private Schedule schedule;
    private List<ScheduledFlight> toBeRemoved;

    public RemoveFlightCommand(Schedule schedule, List<ScheduledFlight> toBeRemoved) {
```

```
        this.schedule = schedule;
        this.toBeRemoved = toBeRemoved;
    }

    public void execute() {
        schedule.getScheduledFlights().removeAll(toBeRemoved);
    }

    public void undo() {
        schedule.getScheduledFlights().addAll(toBeRemoved);
    }
}

class RemoveScheduledFlightCommand implements Command {
    private Schedule schedule;
    private ScheduledFlight scheduledFlight;

    public RemoveScheduledFlightCommand(Schedule schedule, ScheduledFlight scheduledFlight) {
        this.schedule = schedule;
            this.scheduledFlight = scheduledFlight;
        }

        public void execute() {
            schedule.getScheduledFlights().remove(scheduledFlight);
        }

        public void undo() {
            schedule.getScheduledFlights().add(scheduledFlight);
        }
    }

class ClearScheduleCommand implements Command {
    private Schedule schedule;
    private List<ScheduledFlight> previousState;

    public ClearScheduleCommand(Schedule schedule) {
        this.schedule = schedule;
        this.previousState = new ArrayList<>(schedule.getScheduledFlights());
    }

    public void execute() {
        schedule.getScheduledFlights().clear();
    }

    public void undo() {
        schedule.getScheduledFlights().addAll(previousState);
    }

}
```

Benefits of the Refactored Code 💡

There are several benefits to using the Command Pattern in the refactored `Schedule` class:

1. Separation of Concerns: The Command Pattern separates the concerns of requesting an action and executing the action. This separation allows the `Schedule` class to be decoupled from the details of how the commands are executed, which can make the code easier to maintain and modify.
2. Flexibility: Because the Command Pattern encapsulates requests as objects, it allows for greater flexibility in how requests are handled. New commands can be easily added to the system without changing the existing code.
3. Undo/Redo Functionality: The Command Pattern provides a natural way to implement undo/redo functionality in a system. By keeping track of a history of executed commands, the system can easily undo or redo previous actions as needed.

Overall, the Command Pattern can make code more modular, flexible, and testable, which can help to improve its overall quality and maintainability over time.

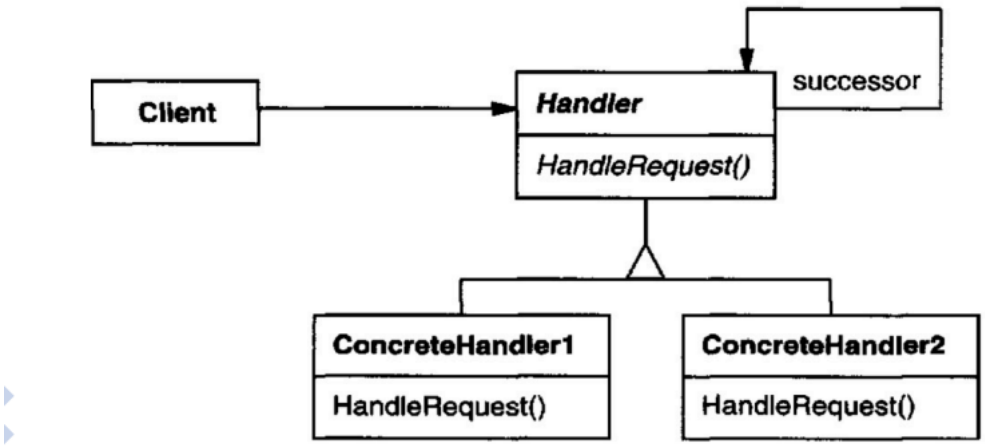
Chain Of Responsibility

Problem ☹️

- The `FlightOrder` class was responsible for order validation, payment processing and order completion. Violates the Single Responsibility Principle.

- This leads to maintenance issues as changing order processing logic will lead to lots of changes in the order class.
- The `FlightOrder` class was tightly coupled to the `PaymentStrategy` class, making it difficult to change payment processing logic. Less flexibility in this way.

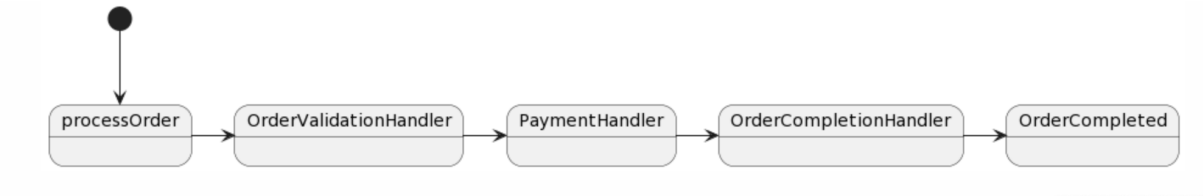
Structure 👷



Client: `FlightOrder` has the `processOrder()` method which initiates a request to the `orderValidationHandler`

Handler: `OrderHandler` which has `processOrder(FlightOrder order)` method which is like the `HandleRequest()` method. It also sets the next handler through `setNextHandler(OrderHandler)` .

ConcreteHandler: `OrderValidationHandler` , `PaymentHandler` and `OrderCompletionHandler` .



Refactored Code 👩‍💻

```
// OrderHandler.java

package flight.reservation.order;

public interface OrderHandler {
    void setNextHandler(OrderHandler nextHandler);
    boolean processOrder(FlightOrder order);
}
```

```
// OrderValidationHandler.java


public class OrderValidationHandler implements OrderHandler {

    private OrderHandler nextHandler;
    private Customer customer;
    private List<Passenger> passengers;
    private List<ScheduledFlight> flights;
    private List<String> noFlyList;


    @Override
    public void setNextHandler(OrderHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public boolean processOrder(FlightOrder order) {
        this.customer = order.getCustomer();
        this.passengers = order.getPassengers();
        this.flights = order.getScheduledFlights();
        this.noFlyList = FlightOrder.getNoFlyList();

        boolean valid = true;
        valid = valid && !noFlyList.contains(customer.getName());
        valid = valid && passengers.stream().noneMatch(passenger -> noFlyList.contains(passenger));
        valid = valid && flights.stream().allMatch(scheduledFlight -> {
            try {
                return scheduledFlight.getAvailableCapacity() >= passengers.size();
            } catch (NoSuchFieldException e) {
                e.printStackTrace();
                return false;
            }
        });

        if (valid) {
            return nextHandler.processOrder(order);
        } else {
            return false;
        }
    }
}
```

```
// PaymentHandler.java

public class PaymentHandler implements OrderHandler {

    private OrderHandler nextHandler;

    @Override
    public void setNextHandler(OrderHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public boolean processOrder(FlightOrder order) {
        if (order.getPaymentStrategy().pay(order.getPrice())) {
            System.out.println("Payment processed successfully.");
            return nextHandler.processOrder(order);
        } else {
            System.out.println("Payment failed.");
            return false;
        }
    }
}
```

```
// OrderCompletionHandler.java

public class OrderCompletionHandler implements OrderHandler {

    private OrderHandler nextHandler;

    @Override
    public void setNextHandler(OrderHandler nextHandler) {
        this.nextHandler = nextHandler;
```

```
    }

    @Override
    public boolean processOrder(FlightOrder order) {
        order.setClosed();
        System.out.println("Order fulfilled.");
        return true;
    }
}
```

```
// FlightOrder.java

public class FlightOrder extends Order {
    private final List<ScheduledFlight> flights;
    private PaymentStrategy paymentStrategy;
    static List<String> noFlyList = Arrays.asList("Peter", "Johannes");

    public FlightOrder(List<ScheduledFlight> flights) {
        this.flights = flights;
    }

    public static List<String> getNoFlyList() {
        return noFlyList;
    }

    public List<ScheduledFlight> getScheduledFlights() {
        return flights;
    }

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public PaymentStrategy getPaymentStrategy() {
        return paymentStrategy;
    }


    public boolean process() {
        OrderHandler orderHandler = new OrderValidationHandler();
        OrderHandler paymentHandler = new PaymentHandler();
        OrderHandler orderFulfillmentHandler = new OrderCompletionHandler();

        orderHandler.setNextHandler(paymentHandler);
        paymentHandler.setNextHandler(orderFulfillmentHandler);

        return orderHandler.processOrder(this);
    }


}
```

Benefits of the Refactored Code 💡

- Improved code maintainability and reduced complexity. The code separation of concerns was refactored into many classes, each responsible for a single component of order processing.
- `OrderHandler` class abstracts away the method of processing order - improved code quality.
- Loose coupling: The handlers were isolated from each other and from the `FlightOrder` class, making it easy to update or add handlers in the future without affecting the existing code.
  Eg: We can add a new handler very easily now, eg: adding an `OrderConfirmationHandler` that will send out a confirmation after `OrderCompletionHandler`.

> 💡 Below is the code before refactoring. It looks so cluttered and also the class Order and the methods of processing order: `isOrderValid()`, and `processOrder()` for payment and order completions, make the responsibility of the `Order` class unnecessarily coupled too deeply with the logic for processing.

```
public class FlightOrder extends Order {


    // actual order attributes
    private final List<ScheduledFlight> flights;
    private PaymentStrategy paymentStrategy;
    static List<String> noFlyList = Arrays.asList("Peter", "Johannes");

    public FlightOrder(List<ScheduledFlight> flights) {
        this.flights = flights;
    }

    public static List<String> getNoFlyList() {
        return noFlyList;
    }

    public List<ScheduledFlight> getScheduledFlights() {
        return flights;
    }

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }


    // cluttered order processing methods
    private boolean isOrderValid(Customer customer, List<String> passengerNames, List<ScheduledFlight> flights) {
        boolean valid = true;
        valid = valid && !noFlyList.contains(customer.getName());
        valid = valid && passengerNames.stream().noneMatch(passenger -> noFlyList.contains(passenger));
        valid = valid && flights.stream().allMatch(scheduledFlight -> {
            try {
                return scheduledFlight.getAvailableCapacity() >= passengerNames.size();
            } catch (NoSuchFieldException e) {
                e.printStackTrace();
                return false;
            }
        });
        return valid;
    }

    public boolean processOrder() throws IllegalStateException {
        if (paymentStrategy == null) {
            throw new IllegalStateException("Payment strategy is not set");
        }

        if(paymentStrategy.pay(this.getPrice())) {
           setClosed();
           return true;
        }

        return false;
    }
}
```

💡 Now let's look at the refactored code especially at the method `process()`. It looks so much cleaner and abstracts out and handles the processing order logic including order validation, payment and order completion.

```
public class FlightOrder extends Order {
    private final List<ScheduledFlight> flights;
    private PaymentStrategy paymentStrategy;
    static List<String> noFlyList = Arrays.asList("Peter", "Johannes");

    public FlightOrder(List<ScheduledFlight> flights) {
        this.flights = flights;
    }

    public static List<String> getNoFlyList() {
        return noFlyList;
    }

    public List<ScheduledFlight> getScheduledFlights() {
        return flights;
    }
```

```
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public PaymentStrategy getPaymentStrategy() {
        return paymentStrategy;
    }


    public boolean process() {
        OrderHandler orderHandler = new OrderValidationHandler();
        OrderHandler paymentHandler = new PaymentHandler();
        OrderHandler orderFulfillmentHandler = new OrderCompletionHandler();

        orderHandler.setNextHandler(paymentHandler);
        paymentHandler.setNextHandler(orderFulfillmentHandler);

        return orderHandler.processOrder(this);
    }


}
```

💡 Also integrating this pattern into the already existing code in `ScenarioTest.java` was very easy which might be the result of all the previous design patterns. It also makes the implementation code much more meaningful and comprehensive. Order processing is simply just calling the method `process()` on it.

```
ScheduledFlight scheduledFlight = schedule.searchScheduledFlight(flight.getNumber());
FlightOrder order = customer.createOrder(Arrays.asList("Amanda", "Max"), Arrays.asList(scheduledFlight), 180);
order.setPaymentStrategy(new PaypalStrategy("amanda@ya.com", "amanda1985"));
boolean isProcessed = order.process();
```

Bonus Exploring: Code Metrics

Did the refactoring actually work? How do we know it? Thankfully, as we have all learnt in class we can investigate using code metrics. Here are some interesting observations after running an IntelliJ plugin called MetricsTree using both the original and refactored code.

Original Code Metrics

Halstead Metric Set:
Halstead Effort: 162533.5096
Halstead Errors: 1.9223

MOOD Metrics Set:
Attribute Hiding Factor: 89.6154%
Attribute Inheritance Factor: 2.4390%
Coupling Factor: 30.7692%
Method Hiding Factor: 6.5789%
Method Inheritance Factor: 17.3913%
Polymorphism Factor: 5.8824%

Statistics:
Lines Of Code: 402
Non-Commenting Source Statements: 232
Number Of Abstract Classes: 0
Number Of Concrete Classes: 14

Number Of Interfaces: 0
Number Of Static Classes: 0

Maintainability Index:
Maintainability Index: 16.3117

Refactored Code Metrics

Halstead Metric Set:
Halstead Effort: 155994.58
Halstead Errors: 2.3531

MOOD Metrics Set:
Attribute Hiding Factor: 97.5929%
Attribute Inheritance Factor: 1.3333%
Coupling Factor: 12.6894%
Method Hiding Factor: 12.2596%
Method Inheritance Factor: 12.7517%
Polymorphism Factor: 62.7451%

Statistics:
Lines Of Code: 589
Non-Commenting Source Statements: 326
Number Of Abstract Classes: 0
Number Of Concrete Classes: 27
Number Of Interfaces: 6
Number Of Static Classes: 1

Maintainability Index:
Maintainability Index: 11.7478