CS6.401 Software Engineering Spring 2024

Assignment 2
Design Patterns

Team 23

Team Members:

Name	Roll Number
Karan Bhatt	2022202003
Madhusree Bera	2022202007
Vedashree Ranade	2022201073
Yash Maheshwari	2022201074
Piyush Rana	2022202012

Table of Contents

Following are the design patterns applied to the given codebase:

1. Builder Pattern

Description

- The Builder pattern is a creational design pattern.
- The Builder pattern is used to construct complex objects step by step, allowing you to create different representations of the same product.
- It helps avoid telescoping constructors with many optional parameters by providing a more readable and flexible approach to object construction.
- This pattern is beneficial when you need to build objects with multiple configurations or when the construction process involves similar steps that differ only in details.
- The Builder pattern also allows for the construction of Composite trees or other complex objects, and it ensures that the client code cannot access an incomplete or unfinished product.

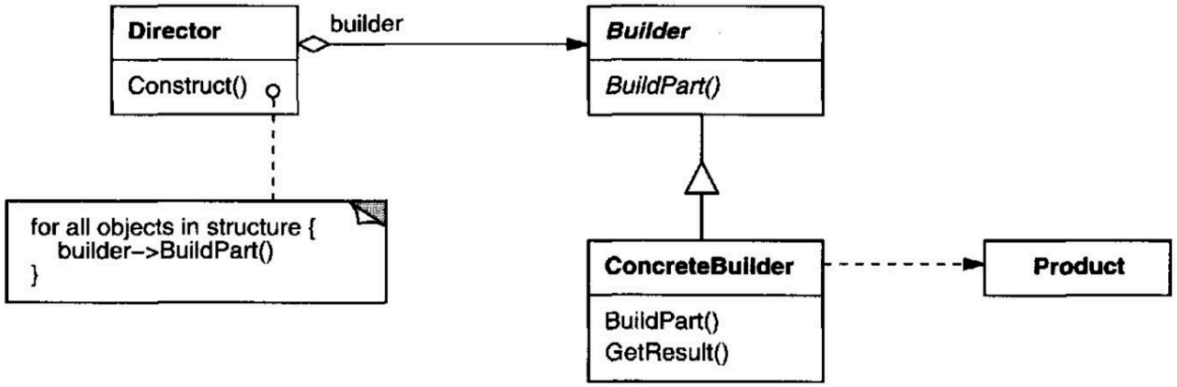
Potential benefits

- **Step-by-Step Construction:** You can construct objects step by step, allowing you to control and customise the construction process.
- **Code Reusability:** The same construction code can be reused to build various representations of products, reducing code duplication.
- **Single Responsibility Principle (SRP):** It helps isolate complex construction code from the business logic of the product, adhering to the SRP and improving code maintainability.

Drawbacks

- **Increased Complexity:** The pattern requires creating multiple new classes, which can increase the overall complexity of the codebase, especially for simpler object construction scenarios.

Structure



Case1: ScheduledFlight

Reasoning for application

- The ScheduledFlight class has two constructors for customising the creation of the ScheduledFlight objects. One constructor uses the currentPrice while the other does not.
- Both the constructors differ by a single parameter initialization logic. So the construction code can be reused.
- There is a scope for constructing the ScheduledFlight objects according to the need of optional parameters. Hence the builder design pattern can be applied in this case.

Benefits and impact on the codebase

- The ScheduledFlight objects can be created according to the need of the optional parameters.
- The complex construction code is isolated from the main logic which improves the code maintainability and also adheres to the SRP.
- The overall complexity of the codebase may increase slightly due to the creation of new classes.

Code snippets

ScheduledFlightBuilder.java

```
public abstract class ScheduledFlightBuilder {

    private List<Passenger> passengers ;
    private Date departureTime;
    protected double currentPrice = 100;
    private int number;
    private Airport departure;
    private Airport arrival;
    protected Object aircraft;

    public ScheduledFlightBuilder setNumber(int number) {
        this.number = number;
        return this;
    }
    public ScheduledFlightBuilder setDeparture(Airport departure) {
        this.departure = departure;
        return this;
    }
    public ScheduledFlightBuilder setArrival(Airport arrival) {
        this.arrival = arrival;
        return this;
    }
    public ScheduledFlightBuilder setAircraft(Object aircraft) {
        this.aircraft = aircraft;
        return this;
    }
    public ScheduledFlight build(){
        return new ScheduledFlight(this);
    }
    public List<Passenger> getPassengers() {
        return passengers;
    }
    public Date getDepartureTime() {
        return departureTime;
    }
    public double getCurrentPrice() {
        return currentPrice;
    }
    public int getNumber() {
        return number;
    }
    public Airport getDeparture() {
        return departure;
    }
    public Airport getArrival() {
        return arrival;
    }
    public Object getAircraft() {
        return aircraft;
    }
}
```

ScheduledFightWithoutPriceBuilder.java

```
public class ScheduledFightWithoutPriceBuilder extends
ScheduledFlightBuilder{
}
```

ScheduledFlightWithPriceBuilder.java

```
public class ScheduledFlightWithPriceBuilder extends
ScheduledFlightBuilder{

   public ScheduledFlightBuilder setCurrentPrice(double
currentPrice){
       super.currentPrice = currentPrice;
       return this;
   }
}
```

ScheduledFlightDirector.java

```
public class ScheduledFlightDirector {
   ScheduledFlightBuilder scheduledFlightBuilder;
   static Scanner sc = new Scanner(System.in);

   public ScheduledFlightDirector(ScheduledFlightBuilder
scheduledFlightBuilder) {
       this.scheduledFlightBuilder = scheduledFlightBuilder;
   }

   private ScheduledFlight
createScheduledFlightWithoutPrice(Airport departure, Airport
arrival, Object aircraft){
       System.out.print("Enter flight number: ");
       int flight_no = sc.nextInt();
       return
scheduledFlightBuilder.setNumber(flight_no).setDeparture(departure
).setArrival(arrival).setAircraft(aircraft).build();
   }

   private ScheduledFlight createScheduledFlightWithPrice(Airport
departure, Airport arrival, Object aircraft){
       System.out.print("Enter flight number: ");
       int flight_no = sc.nextInt();

       System.out.print("Enter price: ");
       double price = sc.nextDouble();

       ScheduledFlightBuilder temp =
scheduledFlightBuilder.setNumber(flight_no).setDeparture(departure
).setArrival(arrival).setAircraft(aircraft);
       return
((ScheduledFlightWithPriceBuilder)temp).setCurrentPrice(price).bui
```

```
ld();
    }

    public ScheduledFlight createScheduledFlight(Airport departure,
Airport arrival, Object aircraft){
        if(scheduledFlightBuilder instanceof
ScheduledFightWithoutPriceBuilder)
            return createScheduledFlightWithoutPrice( departure,
arrival,  aircraft);
        else if(scheduledFlightBuilder instanceof
ScheduledFlightWithPriceBuilder)
            return createScheduledFlightWithPrice( departure,
arrival,  aircraft);
        return null;
    }
}
```

scheduleFlight method in Schedule class

```
public void scheduleFlight(ScheduledFlightBuilder
scheduledFlightBuilder, Airport departure, Airport arrival, Object
aircraft){
    ScheduledFlightDirector scheduledFlightDirectorObj = new
ScheduledFlightDirector(new ScheduledFightWithoutPriceBuilder());

    ScheduledFlight scheduledFlight =
scheduledFlightDirectorObj.createScheduledFlight( departure,
arrival,  aircraft);
    scheduledFlights.add(scheduledFlight);
}
```

Flight constructor

```
public Flight(ScheduledFlightBuilder builder) throws
IllegalArgumentException{
    this.number = builder.getNumber();
    this.departure = builder.getDeparture();
    this.arrival = builder.getArrival();
    this.aircraft = builder.getAircraft();
    checkValidity();
}
```

ScheduledFlight Constructor

```
public ScheduledFlight(ScheduledFlightBuilder
scheduledFlightBuilder){
    super(scheduledFlightBuilder);
    this.departureTime = scheduledFlightBuilder.getDepartureTime();
    this.passengers = new ArrayList<>();
```

```
}
```

Case2: Airport

Reasoning for application

- The Airport class has two constructors for customising the creation of the airport objects. One constructor uses the list of allowed aircrafts provided as an argument while the other creats a fixed list if there it is not provided as an argument.
- Both the constructors differ by a single parameter initialization logic. So the construction code can be reused.
- There is a scope for constructing the Airport objects according to the need of optional parameters. Hence the builder design pattern can be applied in this case.

Benefits and impact on the codebase

- The Airport objects can be created according to the need of the optional parameters.

Code snippets

Airport constructor

```
public Airport(AirportBuilder airportBuilder){
    this.name = airportBuilder.getName();
    this.code = airportBuilder.getCode();
    this.location = airportBuilder.getLocation();
    this.allowedAircrafts = airportBuilder.getAllowedAircrafts();
}
```

AirportBuilder.java

```
public abstract class AirportBuilder {
    private String name;
    private String code;
    private String location;
    private List<Flight> flights;
    protected String[] allowedAircrafts;

    public AirportBuilder setName(String name) {
        this.name = name;
        return this;
    }
```

```
    public AirportBuilder setCode(String code) {
        this.code = code;
        return this;
    }

    public AirportBuilder setLocation(String location) {
        this.location = location;
        return this;
    }

    public AirportBuilder setFlights(List<Flight> flights) {
        this.flights = flights;
        return this;
    }

    abstract public AirportBuilder setAllowedAircrafts(String[]
allowedAircrafts);

    public Airport build(){
        return new Airport(this);
    }

    public String getName() {
        return name;
    }

    public String getCode() {
        return code;
    }

    public String getLocation() {
        return location;
    }

    public List<Flight> getFlights() {
        return flights;
    }

    public String[] getAllowedAircrafts() {
        return allowedAircrafts;
    }
}
```

AirportDirector.java

```
public class AirportDirector {

    AirportBuilder airportBuilder;

    public AirportDirector(AirportBuilder airportBuilder) {
        this.airportBuilder = airportBuilder;
    }
```

```
   private Airport createAirportWithoutAllowedAircrafts(String name, String
code, String location, String[] allowedAircrafts){
      return
airportBuilder.setName(name).setCode(code).setLocation(location).setAllowedAi
rcrafts().build();
   }

   private Airport createAirportWithAllowedAircrafts(String name, String
code, String location, String[] allowedAircrafts){
       AirportBuilder temp =
airportBuilder.setName(name).setCode(code).setLocation(location);
       return
((AirportWithAllowedAircraftsBuilder)temp).setAllowedAircrafts(allowedAircraf
ts).build();

   }

   public Airport createAirport(String name, String code, String location,
String[] allowedAircrafts){
       if(airportBuilder instanceof AirportWithAllowedAircraftsBuilder)
           return createAirportWithAllowedAircrafts(name, code, location,
allowedAircrafts);
       else if(airportBuilder instanceof
AirportWithoutAllowedAircraftsBuilder)
           return createAirport(name, code, location, null);
       return null;
   }
}
```

AirportWithAllowedAircraftsBuilder.java

```
public class AirportWithAllowedAircraftsBuilder extends AirportBuilder{
   public AirportBuilder setAllowedAircrafts(String[] allowedAircrafts) {
       this.allowedAircrafts = allowedAircrafts;
       return this;
   }
}
```

AirportWithoutAllowedAircraftsBuilder.java

```
public class AirportWithoutAllowedAircraftsBuilder extends AirportBuilder{

   public AirportBuilder setAllowedAircrafts(String[] allowedAircrafts) {
       this.allowedAircrafts = new String[]{"A380", "A350", "Embraer 190",
"Antonov AN2", "H1", "H2", "HypaHype"};
       return this;
   }
}
```

Note: The existing constructors of ScheduledFlight are not removed. The new constructor is added in the class for showing the implementation of the design pattern.

2. Factory Pattern

Description

- The Factory pattern is a creational design pattern.
- It is used when we don't know beforehand the exact types and dependencies of the objects our code should work with.
- It separates the construction of objects from the code that uses them, making it easier to extend the construction code independently.
- This pattern is useful when you want to provide users of your library or framework with a way to extend its internal components without modifying existing code.
- It is also beneficial for saving system resources by reusing existing objects instead of rebuilding them each time.
- Overall, the Factory Method pattern provides a flexible and reusable way to create objects in your codebase.

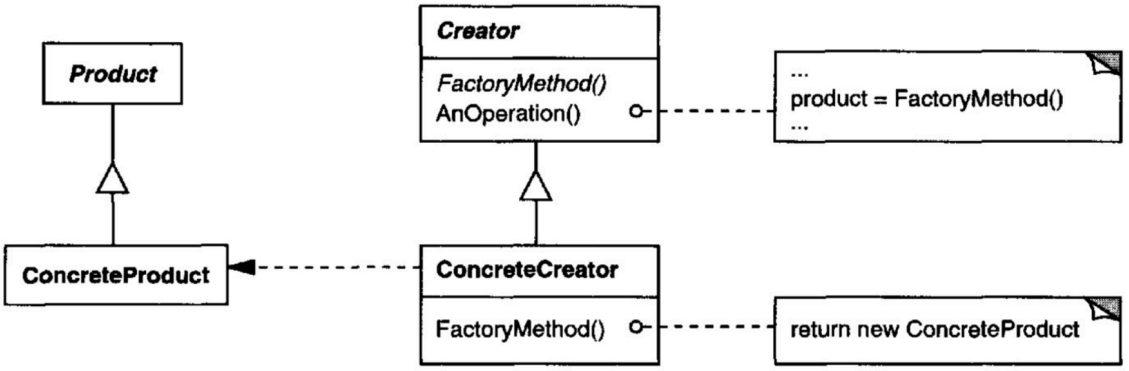
Potential benefits

- **Loose Coupling:** It helps avoid tight coupling between the creator and the concrete products, as the creator only deals with the abstract product interface.
- **Single Responsibility Principle (SRP):** The pattern allows you to move the product creation code into one place in the program, simplifying the code and adhering to the SRP.
- **Open/Closed Principle (OCP):** You can introduce new types of products into the program without modifying existing client code, promoting code extensibility and maintainability, adhering to the OCP.

Drawbacks

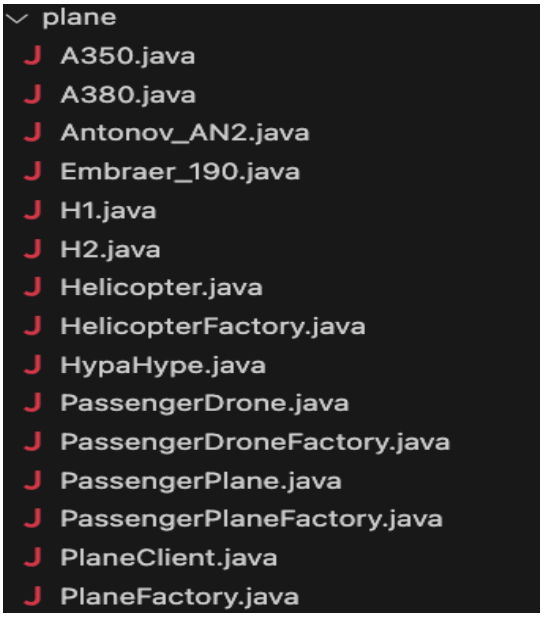
- **Increased Complexity:** Introducing a lot of new subclasses to implement the pattern can make the code more complicated, especially if you're adding it to an existing hierarchy of creator classes.
- **Potential Overhead:** There may be a slight runtime overhead associated with using the Factory Method pattern, as it involves dynamic object creation and method calls through interfaces.

Structure



Reasoning for application

- Planes initially have three classes, i.e., “Helicopter.java”, “PassengerDrone.java” and “PassengerPlane.java”.
- These classes check the specific Plane types, and construct another plane in the same Class using if-else.
- If we want to add another plane category, let’s say, fighter jet or spaceship, etc., we need to update accordingly in that code itself, which is not a good way in software development. So, we have to do something to tackle this issue. That’s why we use Factory Pattern here.
- Also, in a particular file, let’s say, “helicopter.java”, if we need to create new helicopters, with different functionalities, we can get stuck in the future. That’s where we use Factory Pattern again



Note: The previous codebase has not been deleted because it might be needed during the testing phase.

Benefits and impact on the codebase

- The major benefit we got by applying Factory Pattern is “Extensibility”, i.e., we can extend our code to create any number of classes, and the client will not get to know anything about it.
- Another benefit is that, since we make different files for different Plane type, we are following “Single Responsibility Principle”,
- Apart from it, we let our code “Open for Extension, but closed for Modification" , as we can create different “planes” whenever required.
- It is also following “Dependency Inversion Principle”, which signifies that, “Classes should depend on Interfaces, rather than concrete classes.”

Code snippets

PlaneFactory.java

```java
package flight.reservation.plane;

public interface PlaneFactory {
    public abstract PlaneFactory getPlane(String model);
    public abstract int getPassengerCapacity();
}
```

HelicopterFactory.java - Implemented from PlaneFactory

```java
package flight.reservation.plane;
import java.lang.String;

public abstract class HelicopterFactory implements PlaneFactory {
    private String model;
    private int passengerCapacity;

    public HelicopterFactory(String model) {
        this.model = model;
    }

    public abstract String getModel();
    public abstract int getPassengerCapacity();

    @Override
    public PlaneFactory getPlane(String model){
        if (model.equals("H1")) {
            return new H1(model);
        }
        else if (model.equals("H2")) {
            return new H2(model);
        }
        else {
            throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}

```

PassengerDroneFactory.java - Implemented from PlaneFactory

```java
public abstract class PassengerDroneFactory implements PlaneFactory {
    private final String model;

    public PassengerDroneFactory(String model) {
        this.model = model;
    }

    public abstract String getModel();

    @Override
    public int getPassengerCapacity(){
        return 0;
    }


    @Override
    public PlaneFactory getPlane(String model){
        if (model.equals("HypaHype")) {
            return new HypaHype(model);
        }
        else {
            throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}
```

PassengerPlaneFactory.java - Implemented from PlaneFactory

```java
package flight.reservation.plane;
import java.lang.String;

public abstract class PassengerPlaneFactory implements PlaneFactory {
    public String model;
    public int passengerCapacity;
    public int crewCapacity;

    public PassengerPlaneFactory(String model) {
        this.model = model;
    }

    public abstract int getPassengerCapacity();

    @Override
    public PlaneFactory getPlane(String model){
        if (model.equals("A380")) {
            return new A380(model);
        }
        else if (model.equals("A350")) {
            return new A350(model);
        }
        else if (model.equals("Embraer 190")) {
            return new Embraer_190(model);
        }
        else if (model.equals("Antonov AN2")) {
            return new Antonov_AN2(model);
        }
        else {
            throw new IllegalArgumentException(String.format("Model type '%s' is not recognized", model));
        }
    }
}
```

- After these PlaneFactory classes, we have implemented Factory Pattern within these classes, to make our code extensible with other SOLID Principles as well.

H1.java - extends from HelicopterFactory

```java
package flight.reservation.plane;

public class H1 extends HelicopterFactory {
    private final String model;
    private final int passengerCapacity;

    public H1(String model) {
        super(model);
        this.model = model;
        this.passengerCapacity = 4;
    }

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

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

H2.java - extends from HelicopterFactory

```java
package flight.reservation.plane;

public class H2 extends HelicopterFactory {
    private final String model;
    private final int passengerCapacity;

    public H2(String model) {
        super(model);
        this.model = model;
        this.passengerCapacity = 6;
    }

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

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

Hypahype.java - extends from PassengerDroneFactory

```java
package flight.reservation.plane;

public class HypaHype extends PassengerDroneFactory {
    private final String model;

    public HypaHype(String model) {
        super(model);
        this.model = model;
    }

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

A350.java - extends from PassengerPlaneFactor

```java
package flight.reservation.plane;

public class A350 extends PassengerPlaneFactory {
    public String model;
    public int passengerCapacity;
    public int crewCapacity;

    public A350(String model) {
        super(model);
        this.model = model;
        this.passengerCapacity = 320;
        this.crewCapacity = 40;
    }

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

Similarly, all other classes, like, “A380”, “Antonov AN2” and “Embraer 190” will be extended like this.

3. Strategy Pattern

Description

- The Strategy pattern is a behavioural design pattern.
- It is used to define a family of algorithms, encapsulate each one, and make them interchangeable. It enables the client to choose an algorithm at runtime.
- This pattern is beneficial when we have multiple algorithms that can be used interchangeably and when we want to avoid a large number of conditional statements in the code.
- It helps in isolating the business logic of a class from the implementation details of algorithms, promoting code reusability and maintainability.

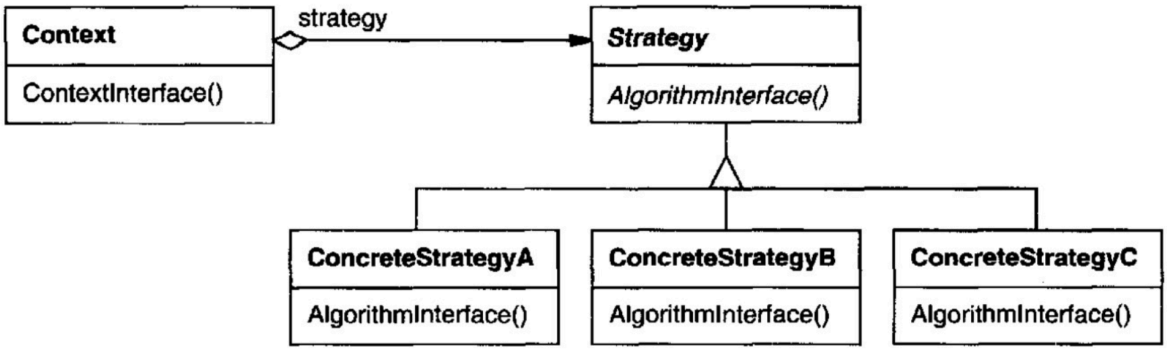
Potential benefits

- **Runtime Algorithm Swapping:** You can swap algorithms used inside an object at runtime, allowing for dynamic behaviour changes.
- **Isolation of Implementation Details:** The pattern isolates the implementation details of an algorithm from the code that uses it, promoting code modularity and maintainability.
- **Composition over Inheritance:** It enables you to replace inheritance with composition, as you can encapsulate algorithms into separate classes and compose them into the context object.
- **Open/Closed Principle (OCP):** You can introduce new strategies without modifying the context class, adhering to the OCP and promoting code extensibility.

Drawbacks

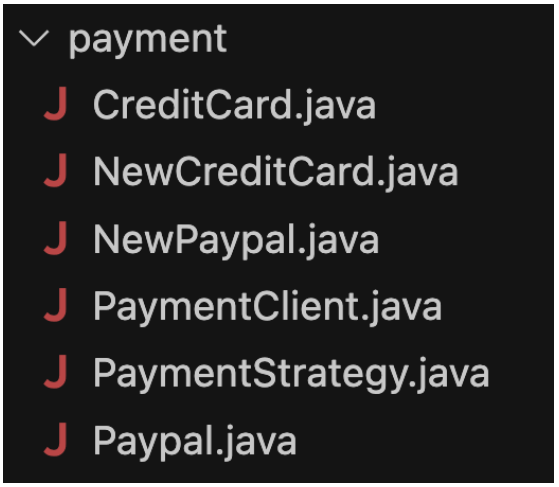
- **Overcomplication for Few Algorithms:** If you have only a couple of algorithms that rarely change, the pattern can overcomplicate the program with new classes and interfaces.
- **Client Awareness:** Clients must be aware of the differences between strategies to select the proper one, which can lead to increased complexity and potential confusion.
- **Functional Programming Alternative:** Modern programming languages with functional type support allow implementing different algorithm versions using anonymous functions. This can achieve similar results as the Strategy pattern without the need for extra classes and interfaces, reducing code bloat.

Structure



Reasoning for application

- Payment initially has two classes, i.e., “CreditCard.java” and “Paypal.java”.
- These classes have a specific working set, which might be updated later.
- But, what if we have to increase the Payment Gateways or change the behaviour of one of them at the run-time.
- We have similar objects here, but all the Payment Gateways might be implemented differently. That’s where we use Strategy Pattern.



Benefits and impact on the codebase

- The major benefit we got by applying Strategy Pattern is “Extensibility”, i.e., we can extend our code to create any number of classes(payment gateways in this case), and the client will not get to know anything about it.
- Another benefit is that, since we make different files for different Payment Type, we are following “Single Responsibility Principle”,

- Apart from it, we let our code “Open for Extension, but closed for Modification" , as we can create different “payment gateways” whenever required, with “pay” and “ValidatePaymentDetails” as the major abstract function.

Code snippets

PaymentStrategy.java

```java
package flight.reservation.payment;

public interface PaymentStrategy {
    public abstract boolean pay();
    public abstract boolean validatePaymentDetails();
}
```

NewPaypal.java - Implements from PaymentStrategy

```java
package flight.reservation.payment;

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

public class NewPaypal implements PaymentStrategy {
    private String emailId;
    private String userName;
    public static final Map<String, String> DATA_BASE = new HashMap<>();

    public NewPaypal() {
        this.userName = "user";
        this.emailId = "user@gmail.com";
        DATA_BASE.put(userName, emailId);
    }

    public NewPaypal(String userName){
        this.userName = userName;
        this.emailId = DATA_BASE.get(userName);
    }

    public NewPaypal(String userName, String emailId) {
        DATA_BASE.put(userName, emailId);
    }

   @Override
    public boolean validatePaymentDetails(){
        return true;
    }

   @Override
    public boolean pay (){
        System.out.println(" Amount Paid using Paypal");
        return true;
    }
}
```

NewCreditCard.java - Implements from PaymentStrategy

```java
package flight.reservation.payment;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

import flight.reservation.payment.PaymentStrategy;
import java.util.Date;

public class NewCreditCard implements PaymentStrategy {
    private double amount;
    private String number;
    private Date date;
    private String cvv;
    private boolean valid;

    public NewCreditCard(Double amount, String number, Date date, String cvv) {
        this.amount = amount;
        this.number = number;
        this.date = date;
        this.cvv = cvv;
        this.validatePaymentDetails();
    }

    public void setAmount(double amount) {this.amount = amount;}
    public double getAmount()    { return amount;}
    public boolean isValid()     { return valid; }

    @Override
    public boolean validatePaymentDetails(){
        return this.valid = number.length() > 0 &&
        date.getTime() > System.currentTimeMillis() &&
        cvv.length() == 3;
    }
    @Override
    public boolean pay (){
        System.out.println(" Amount Paid using Credit Card");
        return true;
    }
}
```

PaymentClient.java

```java
package flight.reservation.payment;

public class PaymentClient {
    public static void main(String[] args) {
        PaymentStrategy ps1 = new NewCreditCard();
        PaymentStrategy ps2 = new NewPaypal();

        ps1.pay();
        ps2.pay();
    }
}
```

4. Observer Pattern

Description

- The Observer is a behavioral design pattern.
- It is used when changes to the state of one object require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
- It is particularly useful in graphical user interface classes, where custom code needs to be executed when certain events occur, such as a button press.
- The pattern allows objects to subscribe to event notifications from other objects, enabling a dynamic and flexible relationship between publishers and subscribers.
- This pattern is suitable when objects need to observe others for a limited time or in specific cases, and it allows subscribers to join or leave the subscription list dynamically.

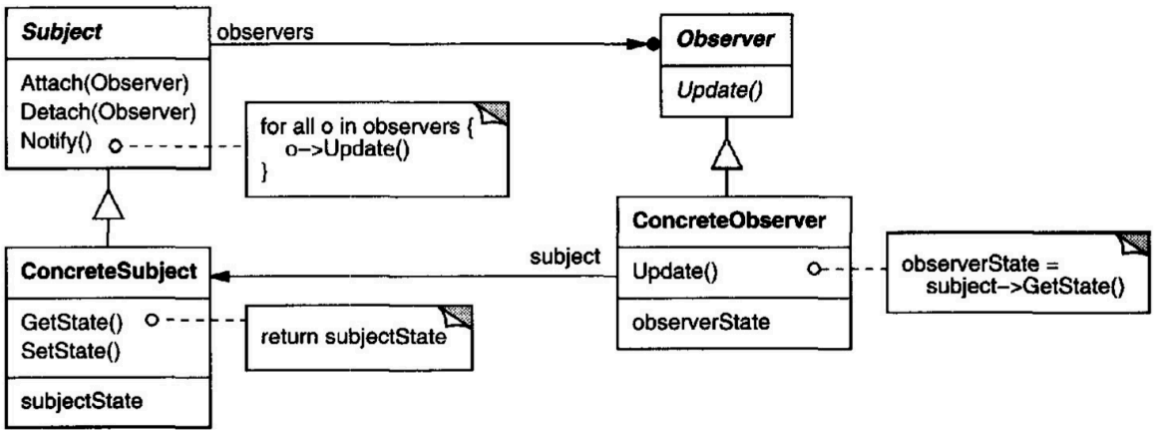
Potential benefits

- **Open/Closed Principle (OCP):** We can introduce new subscriber classes without modifying the publisher's code, and vice versa if there's a publisher interface. This promotes code extensibility and maintainability, adhering to the OCP.
- **Runtime Object Relations:** It allows us to establish relationships between objects at runtime, enabling dynamic and flexible interactions between publishers and subscribers.

Drawbacks

- **Random Notification Order:** Subscribers are notified in a random order, which can be a limitation if the order of notification is important for the application's functionality.

Structure



Reasoning for application

- In the given codebase, there is no feature to notify the Customers when a new Airport is added. So customers might miss important updates about available airports
- Customer objects need to be notified when a new Airport is added. For this reason, observer pattern will be applicable
- Customers are the subscribers and Airport data will be published

Benefits and impact on the codebase

- After implementing the notification feature, the Customers can easily be notified when new airports will be created.
- Using observer pattern for notification will help in adding as many Customers as subscribers thus making the solution extensible
- Other type of notifications can also be added using observer pattern making the code changeable for example: notifying passengers on change in flight details
- Subscribers can be easily added or removed
- Code can be easily maintained using this pattern

Code snippets

- First the AirportSubscriber interface is created

AirportSubscriber.java

```
package flight.reservation;

public interface AirportSubscriber {
   public void update(String value);

}
```

- Customer implements AirportSubscriber and adds description for update() method

Customer.java

```
public class Customer implements AirportSubscriber{

@Override
   public void update(String value) {
        System.out.println("Notifying " + this.name + ": New airport added\n" +
value);
   }
```

- AirportNotificationManager has the list of customer subscribers. It has the functionalities to add and notify subscribers, view list of subscribers.

AirportNotificationManager.java

```
package flight.reservation;

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

public class AirportNotificationManager {
    private static List<Customer> subscriberList = new ArrayList<>();
   public static boolean subscribe (Customer dataSubs)
   {
       subscriberList.add(dataSubs);
       System.out.println(" Added customer to list of subscribed customers ");
       return true;
   }

   public void notify(String value)
   {
       for (Customer customer : subscriberList) {
```

```
            customer.update(value);
        }
    }

    public List<Customer> getSubscriberList()
    {
        return subscriberList;
    }
}
```

- When new Customer is created, it subscribes to the AirportNotoficationManager

Customer.java

```
public class Customer implements AirportSubscriber{

    private String email;
    private String name;
    private List<Order> orders;

    public Customer(String name, String email) {
        this.name = name;
        this.email = email;
        this.orders = new ArrayList<>();
        AirportNotificationManager.subscribe(this);
    }
```

- AirportPublisher is composition of the AirportNotificationManager and airport data that is to be published. Whenever new airport data is set, the manager notifies all the subscribers

AirportPublisher.java

```
package flight.reservation;

public class AirportPublisher {
    public String airportData;
    public AirportNotificationManager manager;

    public AirportPublisher()
    {
        this.manager = new AirportNotificationManager();
    }

    public void setAirportData(String data)
```

```
    {
        this.airportData  = data;
        this.manager.notify(this.airportData);

    }

}
```

- The Airport class uses a static AirportPublisher object to publish the data wherever new Airport objects are created

Airport.java

```
package flight.reservation;

import flight.reservation.flight.Flight;

import java.util.List;

public class Airport {

   private final String name;
   private final String code;
   private final String location;
   private List<Flight> flights;
   private String[] allowedAircrafts;
   private static AirportPublisher ap = new AirportPublisher();

   public String toString(){
        return "{ \n Airport : " + name + "\n Code : " + code + "\n Location : " +
location + "\n}";
    }

   public Airport(String name, String code, String location) {
        this.name = name;
        this.code = code;
        this.location = location;
         this.allowedAircrafts = new String[]{"A380", "A350", "Embraer 190", "Antonov
AN2", "H1", "H2", "HypaHype"};

        ap.setAirportData(this.toString());
   }
```

```
        public Airport(String name, String code, String location, String[]
allowedAircrafts) {
        this.name = name;
        this.code = code;
        this.location = location;
        this.allowedAircrafts = allowedAircrafts;

        ap.setAirportData(this.toString());
    }
```

5. Command Pattern

Description

- The Command pattern is a behavioral design pattern.
- It is used to parametrize objects with operations, turning a specific method call into a stand-alone object.
- This allows for interesting uses such as passing commands as method arguments, storing them inside other objects, and switching linked commands at runtime.
- It is useful when you want to queue operations, schedule their execution, execute them remotely, or implement reversible operations like undo/redo functionality.
- The pattern enables the serialization of commands for delayed or scheduled execution, and it can be combined with the Memento pattern to manage the application's state efficiently.

Potential benefits

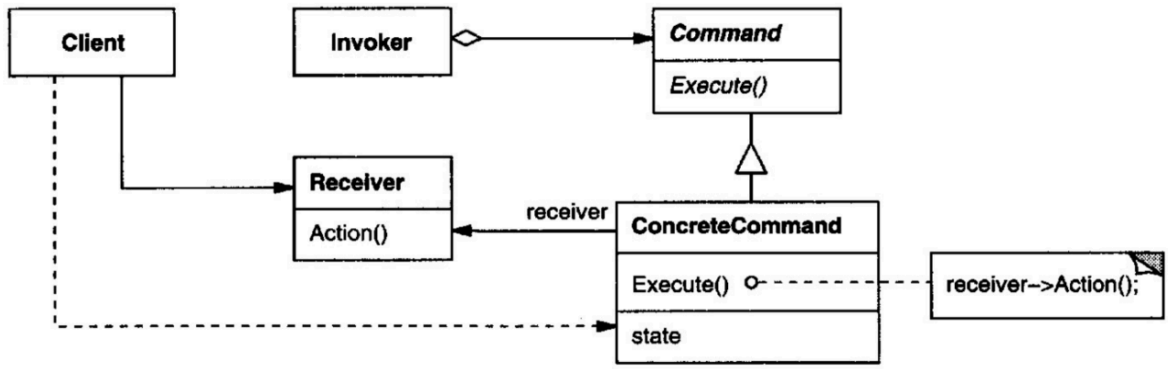
- **Single Responsibility Principle (SRP):** It allows you to decouple classes that invoke operations from classes that perform these operations, promoting a separation of concerns and adhering to the SRP.
- **Open/Closed Principle (OCP):** You can introduce new commands into the application without modifying existing client code, promoting code extensibility and maintainability, adhering to the OCP.
- **Undo/Redo Functionality:** The pattern facilitates the implementation of undo/redo functionality, allowing users to revert and reapply operations as needed.

- **Deferred Execution:** It enables deferred execution of operations, allowing you to schedule and execute commands at a later time or in a different context.
- **Complex Command Assembly:** You can assemble a set of simple commands into a complex one, enabling the creation of sophisticated behaviors and sequences of operations.

Drawbacks

- **Increased Complexity:** Introducing a whole new layer between senders and receivers can make the code more complicated, especially for simpler use cases where the additional abstraction may not be necessary.

Structure



Reasoning for application

- In the available codebase, there is no provision to add flights, aircrafts or airports using user input.
- Moreover, before adding a flight, airports and aircraft should be present.
- There is a need to execute these commands seamlessly using user input, keeping in mind the important validity checks. For this purpose, command pattern is used to add these functionalities.

Benefits and impact on the codebase

- Using the Command pattern, new commands that require complex steps can be easily abstracted from the user. Thus the code became extensible and changeable.
- Now there is a feature of adding airports, aircrafts and flights as per the need and as many times as possible without hardcoding.

Code snippets

- First, an AirportReservationSystem class is created which is composition of list of airports, aircrafts and flights.

AirportReservationSystem.java

```java
package flight.reservation;

import java.util.List;

import flight.reservation.flight.Flight;

public class AirportReservationSystem {
   private static List<Airport> airports;
   private static List<Object> aircrafts;
   private static List<Flight> flights;

   //getters
   public List<Airport> getAirports() {
       return airports;
   }

   public List<Object> getAircrafts() {
       return aircrafts;
   }

   public List<Flight> getFlights() {
       return flights;
   }


   public boolean addAirport(Airport airport) {
       return airports.add(airport);
   }

   public boolean addAircraft(Object aircraft) {
       return aircrafts.add(aircraft);
   }

   public boolean addFlight(Flight flight) {
       return flights.add(flight);
   }

}
```

- Next the Command interface is created with execute() method.

Command.java

```
package flight.reservation.command;
public interface Command {

   boolean execute();
}
```

- The command classes are created to add Airport, Aircraft and Flight that implements the Command interface.
- The AddAirportCommand class takes user input and creates Airport objects and adds to the list of airports.

AddAirportCommand.java

```
package flight.reservation.command;
import java.util.Scanner;
import flight.reservation.Airport;
import flight.reservation.AirportReservationSystem;

public class AddAirportCommand implements Command{
   static Scanner sc ;
   private final AirportReservationSystem airportReservationObject;

   public AddAirportCommand(AirportReservationSystem airportReservationObject){
       this.airportReservationObject = airportReservationObject;
   }
   @Override
   public boolean execute() {
       sc = new Scanner(System.in);
       System.out.println("Enter the airport name: ");
       String name = sc.nextLine();
       System.out.println("Enter the airport code: ");
       String code = sc.nextLine();
       System.out.println("Enter the airport location: ");
       String location = sc.nextLine();
       sc.close();

       Airport airport = new Airport(name, code, location);
       return airportReservationObject.addAirport(airport);
   }
}
```

- The AddAircraftCommand class takes type and model from user input, creates aircraft and adds to the list of aircrafts.

AddAircraftCommand.java

```
package flight.reservation.command;

import java.util.Scanner;

import flight.reservation.AirportReservationSystem;
import flight.reservation.plane.Helicopter;

public class AddAircraftCommand implements Command {
   static Scanner sc ;
   private final AirportReservationSystem airportReservationObject;

   public AddAircraftCommand(AirportReservationSystem airportReservationObject){
       this.airportReservationObject = airportReservationObject;
   }

   @Override
   public boolean execute() {



       sc = new Scanner(System.in);

       System.out.println("Add aircraft type");
       String type = sc.nextLine();
       System.out.println("Enter the aircraft model: ");
       String model = sc.nextLine();

       sc.close();

       //create object of aircraft
       Object aircraft;
       if(type.equals("Helicopter")){
           aircraft = new Helicopter(model);
       }
       else if(type.equals("PassengerPlane")){
           aircraft = new Helicopter(model);
       }
       else if(type.equals("PassengerDrone")){
           aircraft = new Helicopter(model);
       }
       else{
           System.out.println("Invalid aircraft type");
```

```
            return false;
        }

        return airportReservationObject.addAircraft(aircraft);
    }
}
```

- The AddFlightCommand class firstly displays the available airports and aircrafts. Then it allows the user to choose the serial number of airports and aircrafts available to create new flights. Necessary validity checks are done to avoid errors.

AddFlightCommand.java

```
package flight.reservation.command;

import java.util.Scanner;

import flight.reservation.AirportReservationSystem;
import flight.reservation.flight.Flight;

public class AddFlightCommand implements Command{
    static Scanner sc ;
    private final AirportReservationSystem airportReservationObject;

    public AddFlightCommand(AirportReservationSystem airportReservationObject){
        this.airportReservationObject = airportReservationObject;
    }
    @Override
    public boolean execute() {
        // adding flight requires to know the list of Airports
        System.out.println("List of Airports: ");
        for(int i=0; i<airportReservationObject.getAirports().size(); i++){
                                        System.out.println((i+1)    +    ".    "    +
    airportReservationObject.getAirports().get(i).getName());
        }
        // list of available aircrafts
        System.out.println("List of Aircrafts: ");
        for(int i=0; i<airportReservationObject.getAircrafts().size(); i++){
                                        System.out.println((i+1)    +    ".    "    +
    airportReservationObject.getAircrafts().get(i).toString());
        }

        sc = new Scanner(System.in);
```

```
        System.out.println("Enter the flight number: ");
        int number = sc.nextInt();
        System.out.println("Enter the serial number of departure airport: ");
        int departureIndex = sc.nextInt();
        System.out.println("Enter the serial number of arrival airport: ");
        int arrivalIndex = sc.nextInt();

        System.out.println("Enter the serial number of aircraft: ");
        int aircraftIndex = sc.nextInt();

        sc.close();

        if(departureIndex <= 0 || arrivalIndex <= 0 || aircraftIndex <= 0 ||
    departureIndex > airportReservationObject.getAirports().size() ||
    arrivalIndex > airportReservationObject.getAirports().size() ||
    aircraftIndex > airportReservationObject.getAircrafts().size()){
            System.out.println("Invalid serial number. Please try again.");
            return false;
        }

        Flight flight = new Flight(number,
    airportReservationObject.getAirports().get(departureIndex-1),
    airportReservationObject.getAirports().get(arrivalIndex-1),
    airportReservationObject.getAircrafts().get(aircraftIndex-1));
        return airportReservationObject.addFlight(flight);
     }
}
```

- Demo to execute the commands by creating objects

Runner.java

```
public static void executeCommands() {
        Scanner sc = new Scanner(System.in);
        AirportReservationSystem ars = new AirportReservationSystem();
        AddAirportCommand addAirportCommand = new AddAirportCommand(ars);
        AddAircraftCommand addAircraftCommand = new AddAircraftCommand(ars);
        AddFlightCommand addFlightCommand = new AddFlightCommand(ars);

        System.out.println("Enter 1 to add airports, 2 to add aircrafts, 3 to add
    flights: ");
        int choice = sc.nextInt();
        switch (choice) {
                case 1:
                        addAirportCommand.execute();
                        break;
```

```
                case 2:
                        addAircraftCommand.execute();
                        break;
                case 3:
                        addFlightCommand.execute();
                        break;
                default:
                        break;
        }
        sc.close();

}
```