Software Engineering - Class Activity - 6

Team - 10: Event-Driven Engineers

Strategy Design Pattern

Modified Files:

- `flight.reservation.plane.*.java`
- `flight.reservation.flight.Flight.java`
- `flight.reservation.flight.ScheduleFlight.java`

Why Did We Use the Strategy Design Pattern?

- There are different types of flights with varying methods and attributes. Maintaining all of them in a single class is difficult, and extending functionality in the future becomes cumbersome.
- To overcome these issues, we use the **Strategy Design Pattern**, which helps encapsulate different behaviors and allows easy extensibility.

How Did We Use the Strategy Design Pattern?

- We first created an interface called `Vehicle` containing a few functions.
- This enforces all new aircraft classes to implement certain mandatory functionalities.
- We then modified the existing classes: `Helicopter`, `PassengerDrone`, and `PassengerPlane`, ensuring they conform to the `Vehicle` interface.
- We utilized **polymorphism**, allowing the same interface to be implemented differently in each subclass.

Code Changes Made:

```java
// Vehicle interface
public interface Vehicle {
    public String getModel();
    public int getCrewCapacity();
    public int getPassengerCapacity();
}

// Helicopter class
public class Helicopter implements Vehicle {
    @Override
    public String getModel() {
        return model;
    }
    @Override
    public int getCrewCapacity() {
        return crewCapacity;
    }

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

// PassengerDrone class
public class PassengerDrone implements Vehicle {
    public PassengerDrone(String model) {
        this.model = model;
    }

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

    @Override
    public int getCrewCapacity() {
        return crewCapacity;
    }

    @Override
    public int getPassengerCapacity() {
        return 4;
    }
}

// PassengerPlane class
public class PassengerPlane implements Vehicle {
    public PassengerPlane(String model) {
        this.model = model;
    }

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

    @Override
    public int getCrewCapacity() {
        return this.crewCapacity;
    }

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

// Changes made in Flight.java
private boolean isAircraftValid(Airport airport) {
    if (!(this.aircraft instanceof Vehicle)) {
        throw new IllegalArgumentException("Aircraft is not recognized");
    }
    String model = ((Vehicle) this.aircraft).getModel();
    return Arrays.stream(airport.getAllowedAircrafts()).anyMatch(x -> x.equals(model));
}

// Changes made in ScheduleFlight.java
public int getCrewMemberCapacity() throws NoSuchFieldException {
    if (this.aircraft instanceof Vehicle) {
        return ((Vehicle) this.aircraft).getCrewCapacity();
    }
    throw new NoSuchFieldException("This aircraft has no information about its crew capacity
}

public int getCapacity() throws NoSuchFieldException {
    if (this.aircraft instanceof Vehicle) {
        return ((Vehicle) this.aircraft).getPassengerCapacity();
    }
    throw new NoSuchFieldException("This aircraft has no information about its capacity");
}
```

Adapter Design Pattern

Modified Files:

- `flight.reservation.payment.*.java`
- `flight.reservation.flight.FlightOrder.java`

Why Did We Use the Adapter Design Pattern?

- There were different payment functionalities in the codebase.
- Dedicated functions were written for each payment method, making the code difficult to maintain and extend.

- This also caused high coupling, making changes cumbersome.
- To overcome these issues, we use the **Adapter Design Pattern**, which helps standardize interfaces for different payment methods.

How Did We Use the Adapter Design Pattern?

- We created an interface for the `PaymentGateway`.
- This interface defines the basic functionality required for all adapter classes.
- Each adapter class implements this interface according to the specific payment method.
- We implemented `CreditCardAdapter` and `PaypalAdapter`, encapsulating payment processing logic.
- From the client side, users provide the payment method as input, instantiate the appropriate adapter, and use the `pay()` function, regardless of the underlying implementation.

Code Changes Made:

```java
// PaymentGateway interface
public interface PaymentGateway {
    public boolean pay(double amount);
}

// Paypal adapter
public class PaypalAdapter implements PaymentGateway {
    private PaypalInstance paypalInstance;

    public PaypalAdapter(PaypalInstance paypal) {
        this.paypalInstance = paypal;
    }

    @Override
    public boolean pay(double amount) {
        // Process payment with PayPal
        System.out.println("Paying " + amount + " using PayPal.");
        return true;
    }
}

// CreditCard adapter
public class CreditCardAdapter implements PaymentGateway {
    private CreditCard creditCard;

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

    @Override
    public boolean pay(double amount) {
        // Process payment with Credit Card
        System.out.println("Paying " + amount + " using Credit Card.");
        double remainingAmount = this.creditCard.getAmount() - amount;
        if (remainingAmount < 0) {
            System.out.printf("Card limit reached - Balance: %f%n", remainingAmount);
            throw new IllegalStateException("Card limit reached");
        }
        this.creditCard.setAmount(remainingAmount);
        return true;
    }
}

// Paying through Credit Card
public boolean payWithCreditCard(CreditCard card, double amount) throws IllegalStateExceptio
    if (cardIsPresentAndValid(card)) {
        // Pay
        PaymentGateway paymentGateway = new CreditCardAdapter(card);
        return paymentGateway.pay(amount);
    } else {
        return false;
    }
}

// Paying through PayPal
public boolean payWithPayPal(String email, String password, double amount) throws IllegalSta
    if (email.equals(Paypal.DATA_BASE.get(password))) {
        // Pay
        PaymentGateway paymentGateway = new PaypalAdapter(new PaypalInstance(email, passwor
        return paymentGateway.pay(amount);
    } else {
        return false;
    }
}
```

Factory Design Pattern

Modified Files:

- `flight.reservation.Customer.java`
- `flight.reservation.order.FlightOrderFactory.java`
- `flight.reservation.order.OrderFactory.java`

- `flight.reservation.order.FlightOrder.java`

Why Did We Use the Factory Design Pattern?

- We have only flight class. but if there is a possibility for other classes to come in future, we can just implement a factory Design pattern. so that it is easily extensible.
- The creation of `FlightOrder` contained logic that should be centralized for maintainability and scalability.
- The `Customer` class was responsible for too many things, including validation, order creation, and linking passengers, violating the **Single Responsibility Principle (SRP)**.
- Introducing a **Factory Pattern** allows better encapsulation of object creation and provides a structured way to create `FlightOrder` objects while ensuring validation.

How Did We Use the Factory Design Pattern?

- We introduced an **interface `OrderFactory`** that defines a standard contract for creating orders.
- We implemented **`FlightOrderFactory`**, a concrete factory class that follows `OrderFactory` and encapsulates the logic of creating `FlightOrder` objects.
- We modified **`Customer.java`** so that it no longer handles order creation directly but instead delegates it to `FlightOrderFactory`.
- This ensures **flexibility**, making it easy to introduce new order types in the future (e.g., `HotelOrderFactory`, `TrainOrderFactory`).

Code Changes Made:

OrderFactory Interface

```
// OrderFactory.java
public interface OrderFactory {
    Order createOrder(Customer customer, List<String> passengerNames, List<ScheduledFlight>
}
```

FlightOrderFactory Implementation

```
// FlightOrderFactory.java
public class FlightOrderFactory implements OrderFactory {
    @Override
    public FlightOrder createOrder(Customer customer, List<String> passengerNames, List<Sche
        if (!isOrderValid(customer, passengerNames, flights)) {
            throw new IllegalStateException("Order is not valid");
        }
        FlightOrder order = new FlightOrder(flights);
        order.setCustomer(customer);
        order.setPrice(price);

        List<Passenger> passengers = new ArrayList<>();
        for (String name : passengerNames) {
            passengers.add(new Passenger(name));
        }
        order.setPassengers(passengers);

        for (ScheduledFlight scheduledFlight : order.getScheduledFlights()) {
            scheduledFlight.addPassengers(passengers);
        }

        customer.getOrders().add(order);
        return order;
    }
}
```

Customer Class Modification

```
// Customer.java
public class Customer {
    private final OrderFactory orderFactory;

    public Customer(String name, String email) {
        this.name = name;
        this.email = email;
        this.orders = new ArrayList<>();
        this.orderFactory = new FlightOrderFactory(); // Inject factory instance
    }

    public FlightOrder createOrder(List<String> passengerNames, List<ScheduledFlight> flight
        return (FlightOrder) orderFactory.createOrder(this, passengerNames, flights, price)
    }
}
```

Observer Design Pattern

Added/modified Files:

- `flight.reservation.flight.FlightSubject.java`
- `flight.reservation.flight.ObservableFlight.java`
- `flight.reservation.NotifiedPassenger.java`
- `flight.reservation.FlightObserver.java`

Why Did We Use the Observer Design Pattern?

- In a flight reservation system, passengers may want to receive updates about their flight status (e.g., “On Time”, “Delayed”, “Cancelled”).
- Instead of manually checking the flight status repeatedly, passengers can subscribe to flight notifications.
- The **Observer Design Pattern** allows the system to efficiently notify multiple passengers (observers) when the flight status changes.
- This pattern decouples the flight management logic from the notification mechanism, making it more maintainable and scalable.

How Did We Use the Observer Design Pattern?

- Created the FlightObserver Interface:
  - This defines a method update(String flightNumber, String status) that all observers must implement.
  - Any class implementing this interface will receive flight status updates.
- Created the FlightSubject Interface:
  - This interface provides methods to add, remove, and notify observers.
  - Any class implementing this interface can manage a list of observers.
- Modified ObservableFlight Class (The Subject):
  - Extends the Flight class and implements FlightSubject.
  - Manages a list of registered observers (List).
  - When the flight status changes (delayed, cancelled, etc.), it notifies all registered observers.
- Created the NotifiedPassenger Class (An Observer):
  - Extends Passenger and implements FlightObserver.
  - When notified, it prints a message for the passenger about the flight status change.
- We implemented flight cancellation and delay mechanisms by simply updating the status message. Since there is no actual implementation for canceling or delaying flights, the status update acts as a notification system for passengers.

Code Changes Made:

```
// FlightSubject interface
interface FlightSubject {
    void addObserver(FlightObserver observer);
    void removeObserver(FlightObserver observer);
    void notifyObservers();
}

// ObservableFlight class (Subject)
public class ObservableFlight extends Flight implements FlightSubject {
    private String status; // Flight status (e.g., On Time, Delayed, Cancelled)
```

```java
    private List<FlightObserver> FlightObservers;

    public ObservableFlight(int number, Airport departure, Airport arrival, Object aircraft)
        super(number, departure, arrival, aircraft); // Call the existing Flight constructo
        this.status = "On Time"; // Default status
        this.FlightObservers = new ArrayList<>();
    }

    public void cancelFlight() {
        this.status = "Cancelled";
        notifyObservers();
    }

    public void delayFlight(String reason) {
        this.status = "Delayed: " + reason;
        notifyObservers();
    }

    public void setStatus(String status) {
        this.status = status;
        notifyObservers();
    }

    public String getStatus() {
        return status;
    }

    @Override
    public void addObserver(FlightObserver observer) {
        FlightObservers.add(observer);
    }

    @Override
    public void removeObserver(FlightObserver observer) {
        FlightObservers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (FlightObserver observer : FlightObservers) {
            observer.update("Flight " + getNumber(), status);
        }
    }
}

// FlightObserver interface
```

```java
public interface FlightObserver {
    void update(String flightNumber, String status);
}

// NotifiedPassenger class (Observer)
public class NotifiedPassenger extends Passenger implements FlightObserver {
    public NotifiedPassenger(String name) {
        super(name);
    }

    @Override
    public void update(String flightNumber, String status) {
        System.out.println("Notification for " + getName() + ": " + flightNumber + " is now
    }
}
```

by the way, if this project gets into production stage, I would use Paypal method to order flights. because it doesn't deduct money from your account at all =). But, wait a minute!. I think there are only 2 accounts who have the permission. (-_-)