Facade and Behavioural Design Patterns

Key terms

Facade

> A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

Behavioural Design Patterns

> Behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Observer

> The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

Strategy

> The strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family.

Facade

> Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Facade means "face" in French. It is a front-facing building that is the main entrance to a building. The facade is the first thing that a visitor sees when they enter a building. The facade hides the complexity of the building from the visitor. The facade provides a simple interface to the building. The facade is a single point of entry to the building.

Problem

Let us take the example of an e-commerce application. The application has a lot of functionality. It has a product catalog, a shopping cart, a payment system, a shipping system, etc. The application has a lot of classes and a lot of dependencies between them. The application is complex and it is hard to understand how all the classes work together. When you make an order, you have to do the following:

- Call payment gateway to charge the credit card.
- Update the inventory.
- Email the customer.
- Add the order to the shipping queue.

- Update analytics.

The above steps are not trivial. The application has a lot of classes and a lot of dependencies between them. The application is complex and it is hard to understand how all the classes work together. The application is also hard to maintain. If you want to change the way the application sends emails, you will have to change the code in multiple places. If you want to add a new feature, you will have to change the code in multiple places. Imagine how the class looks:

```
public class Order {
    private PaymentGateway paymentGateway;
    private Inventory inventory;
    private EmailService emailService;
    private ShippingService shippingService;
    private AnalyticsService analyticsService;

    public void checkout() {
        paymentGateway.charge();
        inventory.update();
        emailService.send();
        shippingService.add();
        analyticsService.update();
    }
}
```

Here we have a lot of dependencies, some of which might be external vendors. The business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain. The Order class is hard to test. You will have to mock all the dependencies. The Order class is also hard to reuse. If you want to reuse the Order class in another application, you will have to change the code. Every time one of the logic changes, you will have to change the code in multiple places and hence violating SOLID principles.

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

Solution

The Facade pattern suggests that you wrap a complex subsystem with a simpler interface. The Facade pattern provides a higher-level interface that makes the subsystem easier to use. The Facade pattern is implemented by simply creating a new class that encapsulates the complex logic of the existing classes. For our example above, we will move the complex logic to a new class called OrderProcessor.

```
public class OrderProcessor {
    private PaymentGateway paymentGateway;
    private Inventory inventory;
```

```
    private EmailService emailService;
    private ShippingService shippingService;
    private AnalyticsService analyticsService;

    public void process() {
        paymentGateway.charge();
        inventory.update();
        emailService.send();
        shippingService.add();
        analyticsService.update();
    }
}
```

Now we can use the OrderProcessor class in our Order class and delegate the complex logic to the OrderProcessor class.

```
public class Order {
    private OrderProcessor orderProcessor;

    public void checkout() {
        orderProcessor.process();
    }
}
```

The Order class is now much simpler. It has a single responsibility of creating an order. The Order class is also easier to test. You can mock the OrderProcessor class. The Order class is also easier to reuse. You can reuse the Order class in another application without changing the code.

Observer Pattern

Imagine it is iPhone season again, where Apple releases a new version of the iPhone and millions of individuals can't wait to get there hands on it. The stock at the local store has not yet been updated, but you want to be the first to know when the new iPhone is available. You can go to the store every day to check if the stock has been updated, but that is wasteful. Another option is that the store sends everyone an email when the new phones come in. Again, it is wasteful since not everyone is interested in the new iPhone. The best option is that you register or subscribe to the store's mailing list. When the new iPhone comes in, the store sends an email to everyone on the mailing list. This is the motivation behind the Observer pattern.

We now want to build a Bitcoin tracking application that sends out emails or tweets when the price of Bitcoin changes. We have a data model for Bitcoin that contains the current price of Bitcoin. Apart from this we have a `BitcoinTracker` class that is responsible for setting the price of the Bitcoin.

```
public class BitcoinTracker {
    private Bitcoin bitcoin;

    public void setPrice(double price) {
        bitcoin.setPrice(price);
```

```
    }
}
```

Now we want to send an email when the price of Bitcoin changes. We can do this by calling the `sendEmail` method in the setter method of the `BitcoinTracker` class.

```
public class BitcoinTracker {
    private Bitcoin bitcoin;

    public void setPrice(double price) {
        bitcoin.setPrice(price);
        sendEmail();
    }
}
```

The above implementation works but it is not ideal. The `BitcoinTracker` class has two responsibilities. It is responsible for setting the price of Bitcoin and it is responsible for sending an email. The `BitcoinTracker` class violates the Single Responsibility Principle. Similarly, we can also send a tweet when the price of Bitcoin changes. We can do this by calling the `sendTweet` method in the setter method of the `BitcoinTracker` class. This now violates the Open-Closed Principle. We will have to change the code in multiple places if we want to add a new feature.

```
public class BitcoinTracker {
    private Bitcoin bitcoin;

    public void setPrice(double price) {
        bitcoin.setPrice(price);
        sendEmail();
        sendTweet();
    }
}
```

Another option is to have a secondary class fetch the price of Bitcoin and send an email when the price changes. This is known as polling. The problem with any polling approach is that it is wasteful. The secondary class will have to poll the BitcoinTracker class every few seconds to check if the price has changed.

```
public class BitcoinPoller {
    private BitcoinTracker bitcoinTracker;
    private Bitcoin previousBitcoin;

    public void poll() {
        Bitcoin currentBitcoin = bitcoinTracker.getBitcoin();
        if (currentBitcoin.getPrice() != previousBitcoin.getPrice()) {
            sendEmail();
        }
```

```
            this.previousBitcoin = currentBitcoin;
        }
    }
```

The two approaches we have discussed so far are not ideal. The first approach violates the Single Responsibility Principle. The second approach is wasteful. The Observer pattern suggests that you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Implementation

1. `Observable interface` - This interface defines the methods that the subject class must implement. The subject class is responsible for notifying the observers when the state of the subject changes.

```
public interface class Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

2. `Observer interface` - This interface defines the methods that the observer class must implement. The observer class is responsible for updating itself when the state of the subject changes.

```
public interface class Observer {
    void notify();
}
```

3. `Concrete observables` - These are the classes that implement the `Observable` interface. The `BitcoinTracker` class is a concrete observable. The `BitcoinTracker` class is responsible for notifying the observers when the state of the subject changes.

```
public class BitcoinTracker implements Observable {
    private Bitcoin bitcoin;

    public void setPrice(double price) {
        bitcoin.setPrice(price);
        notifyObservers();
    }
}
```

To simplify the code and provide better interfaces, we can borrow from the registry pattern and register observers and even add utility methods to the `Observable` interface.

```
public abstract class Observable {

    List<Observer> observers = new ArrayList<>();

    public void register(Observer observer) {
        observers.add(observer);
    }

    public void deregister(Observer observer) {
        observers.remove(observer);
    }

    public void notifyChange() {
        for (Observer observer : observers) {
            observer.notifyChange();
        }
    }
}
```

4. `Concrete observers` - These are the classes that implement the `Observer` interface. The `EmailSender` class is a concrete observer. The `EmailSender` class is responsible for updating itself when the state of the subject changes.

```
public class EmailSender implements Observer {
    private Bitcoin bitcoin;

    public void notifyChange() {
        sendEmail();
    }
}
```

5. `Client` - The client is responsible for creating the subject and the observers. The client is also responsible for registering the observers with the subject.

```
public class Client {
    public static void main(String[] args) {
        BitcoinTracker bitcoinTracker = new BitcoinTracker();
        EmailSender emailSender = new EmailSender();

        bitcoinTracker.register(emailSender);
    }
}
```

Recap

- There are often times when you want to notify other objects when the state of an object changes. The Observer pattern suggests that you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- In-place updates and polling are not ideal. In-place updates violate the Single Responsibility Principle whereas polling is wasteful.
- To implement the Observer pattern, define an `Observable` interface that defines the methods that the subject class must implement.
- Define an `Observer` interface that defines the methods that the observer class must implement. The subject class is responsible for notifying the observers when the state of the subject changes. The observer class is responsible for updating itself when the state of the subject changes.

Design Patterns in different languages

Observer Pattern

Python

- Observer Pattern - I
- Observer Pattern - II
- Observer Pattern - III
- Observer Pattern - IV

JavaScipt

- Observer Pattern - I
- Observer Pattern - II
- Observer Pattern - III
- Observer Pattern - IV
- Observer Pattern - V
- Observer Pattern - VI