Design Patterns Practice - Structural, Behavioral Patterns

Week 12

The use of design patterns should be the result of deep thinking, not a way to avoid it
– Bill Venners

Objectives

- Learn how to use structural patterns
- Learn how to use behavioral patterns

Contents

- Simple practice: Structural Patterns
  - Decorator
  - Proxy
  - Composite
- Simple practice: Behavioral Patterns
  - Visitor
  - Observer

Simple Practice: Structural Patterns

- We are going to implement simple code with structural patterns
- Please clone or pull the skeleton code [Link]
- We will use the following 3 android projects
  - `structuralPatterns/DecoratorExample`
  - `structuralPatterns/ProxyExample`
  - `structuralPatterns/CompositeExample`

Practice #1: Decorator

- Open `structuralPatterns/DecoratorExample` with Android Studio
- There are 6 classes
  - Client.java: Main function
  - Message.java: Interface about Message (Component)
  - SimpleMessage.java: Concrete component
  - EncryptedMessage.java: Concrete component
  - CompressedMessage.java: Concrete component
  - EncryptedAndCompressedMessage.java: Concrete component

Practice #1: Skeleton Code (1/3)

```java
public interface Message {

    String getContent();

}

public class SimpleMessage implements Message{
    private String content;

    public SimpleMessage(String content) {
        this.content = content;
    }
    @Override
    public String getContent() {
        return content;
    }
}
```

```java
class EncryptedMessage implements Message {
    private String content;

    public EncryptedMessage(String content) {
        this.content = encrypt(content);
    }

    private String encrypt(String content) {
        return "encrypted(" + content + ")";
    }
    @Override
    public String getContent() {
        return content;
    }
}
```

Practice #1: Skeleton Code (2/3)

```
class CompressedMessage implements Message {
    private String content;

    public CompressedMessage(String content) {
        this.content = compress(content);
    }

    private String compress(String content) {
        return "compressed(" + content + ")";
    }
    @Override
    public String getContent() {
        return content;
    }
}
```

```
public class EncryptedAndCompressedMessage implements Message{
    private String content;
    public EncryptedAndCompressedMessage(String content) {
        this.content = compress(encrypt(content));
    }
    private String encrypt(String content) {
        return "encrypted(" + content + ")";
    }
    private String compress(String content) {
        return "compressed(" + content + ")";
    }
    @Override
    public String getContent() {
        return content;
    }
}
```

Practice #1: Skeleton Code (3/3)

```
public class Client {
    public static void main(String[] args){
        Message simpleMessage = new SimpleMessage("Hello");
        System.out.println(simpleMessage.getContent());

        Message encryptedMessage = new EncryptedMessage("Hello");
        System.out.println(encryptedMessage.getContent());

        Message compressedMessage = new CompressedMessage("Hello");
        System.out.println(compressedMessage.getContent());

        Message encryptedAndCompressedMessage = new EncryptedAndCompressedMessage("Hello");
        System.out.println(encryptedAndCompressedMessage.getContent());
    }
}
```

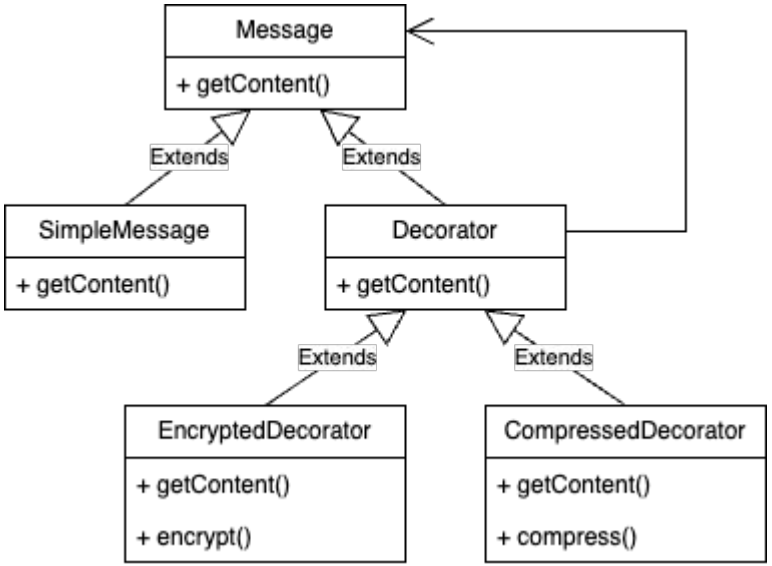
Practice #1: Implement Decorator

- In the skeleton code, the client should initialize the new object for other message
- When we need other operations for `Message`, we should extend many classes. (e.g., `HTMLMessage`, `HTMLAndEncrytedMessage` …)
- Implement the Decorator
  - Eliminate all concrete component except `SimpleMessage`
  - Define `Decorator` interface and `EncryptedDecorate`, `CompressedDecorator`

7 mins

Practice #1 : Decorator Diagram

- Define `Decorator` interface
- Define `EncryptedDecorate` and `CompressedDecorator`



Practice #2: Proxy

- Open `structuralPatterns/ProxyExample` with Android Studio
- There are 2 classes
  - Client.java: Main function
  - RealService.java: Real subject

Practice #2: Skeleton Code

```java
public class RealService {
    public void execute(){
        System.out.println("Execution start");
        for(int i = 0; i < 10; i++){
            System.out.println(i * i);
        }
        System.out.println("Execution end");
    }
}
```

```java
public class Client {
    public static void main(String[] args){
        RealService realService = new RealService();
        realService.execute();
    }
}
```

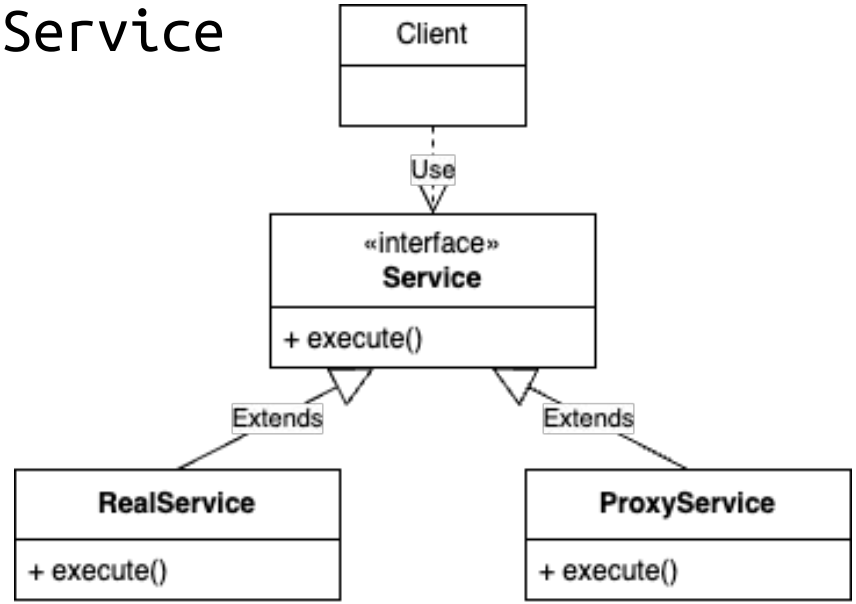
Practice #2: Implement Proxy

- In the skeleton code, the `execute()` method of a `RealService` does not have only one functionality
  - SRP violation
- When developers change the logging, they must change `RealService`
- Implement the Proxy
  - Define `Service` interface
  - Define `ProxyService`
  - Modify `RealService`

5 mins

Practice #2 : Proxy Diagram

- Define `Service` interface
- Define `ProxyService`
- Modify `RealService`



Practice #3: Composite

- Open `structuralPatterns/CompositeExample` with Android Studio
- There are 3 classes
  - Client.java: Main function
  - File.java
  - Directory.java

Practice #3: Skeleton Code (1/3)

```
class File {
    private String name;

    public File(String name) {
        this.name = name;
    }

    public void showDetails() {
        System.out.println("File: " + name);
    }
}
```

Practice #3: Skeleton Code (2/3)

```
class Directory {
    private List<File> files = new ArrayList<>();
    private List<Directory> directories =
                                new ArrayList<>();
    private String name;

    public Directory(String name) {
        this.name = name;
    }

    public void addFile(File file) {
        files.add(file);
    }
    ...
    ...
    public void addDirectory(Directory directory) {
        directories.add(directory);
    }
    public void showDetails() {
        System.out.println("Directory: " + name);
        for (File file : files) {
            file.showDetails();
        }
        for (Directory directory : directories) {
            directory.showDetails();
        }
    }
}
```

Practice #3: Skeleton Code (3/3)

```
public class Client {
    public static void main(String[] args) {
        File file1 = new File("File1.txt");
        File file2 = new File("File2.txt");
        Directory directory1 = new Directory("Directory1");

        directory1.addFile(file1);
        directory1.addFile(file2);
        ...
```

```
        ...
        File file3 = new File("File3.txt");
        Directory directory2 = new
Directory("Directory2");
        directory2.addFile(file3);
        directory2.addDirectory(directory1);

        directory2.showDetails();
    }
}
```

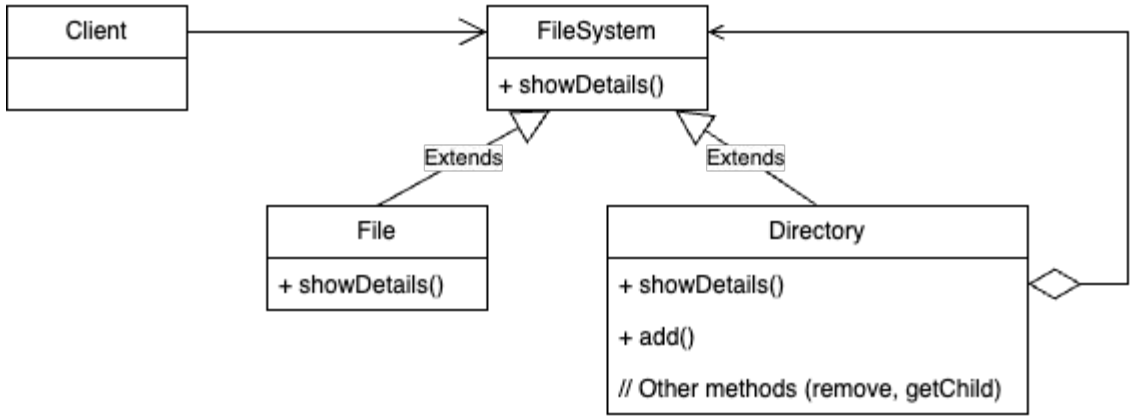
Practice #3: Implement Composite

- In the skeleton code, `File` and `Directory` should be handled differently
- To access the overall structure, we need to call a different interface, method, each time
  - Increasing mistakes
- Implement the Composite
  - Define `FileSystem` as Component interface
  - Modify `File` as Leaf class
  - Modify `Directory` as Composite class

7 mins

Practice #3 : Composite Diagram

- Define `FileSystem` as Component interface
- Modify `File` as Leaf class
- Modify `Directory` as Composite class



Contents

- ~~Simple practice: Structural Patterns~~
  - ~~Decorator~~
  - ~~Proxy~~
  - ~~Composite~~
- Simple practice: Behavioral Patterns
  - Visitor
  - Observer

Simple Practice: Behavioral Patterns

- We are going to implement simple code with behavioral patterns
- Please clone or pull the skeleton code [Link]
- We will use the following 2 android projects
  - `behavioralPatterns/VisitorExample`
  - `behavioralPatterns/ObserverExample`

Practice #4: Visitor

- Open `behavioralPatterns/VisitorExample` with Android Studio
- There are 6 classes
  - Client.java: Main function
  - ComputerPart.java: Interface about ComputerPart (Element)
  - CPU.java: Concrete element
  - GPU.java: Concrete element
  - RAM.java: Concrete element
  - Computer.java: Concrete element

Practice #4: Skeleton Code (1/2)

```java
public interface ComputerPart {
    void run();
}

public class CPU implements ComputerPart{
    @Override
    public void run() {
        System.out.println("Running CPU");
    }
}
```

```java
public class GPU implements ComputerPart{
    @Override
    public void run() {
        System.out.println("Running GPU");
    }
}

public class RAM implements ComputerPart{
    @Override
    public void run() {
        System.out.println("Running RAM");
    }
}
```

Practice #4: Skeleton Code (2/2)

```
class Computer implements ComputerPart {
    ComputerPart[] parts;
    public Computer() {
        parts = new ComputerPart[] {new CPU(),
                new GPU(), new RAM()};
    }
    @Override
    public void run() {
        for (int i = 0; i < parts.length; i++) {
            parts[i].run();
        }
        System.out.println("Running Computer");
    }
}
```

```
public class Client {
    public static void main(String[] args){
        ComputerPart computer = new Computer();
        computer.run();
    }
}
```

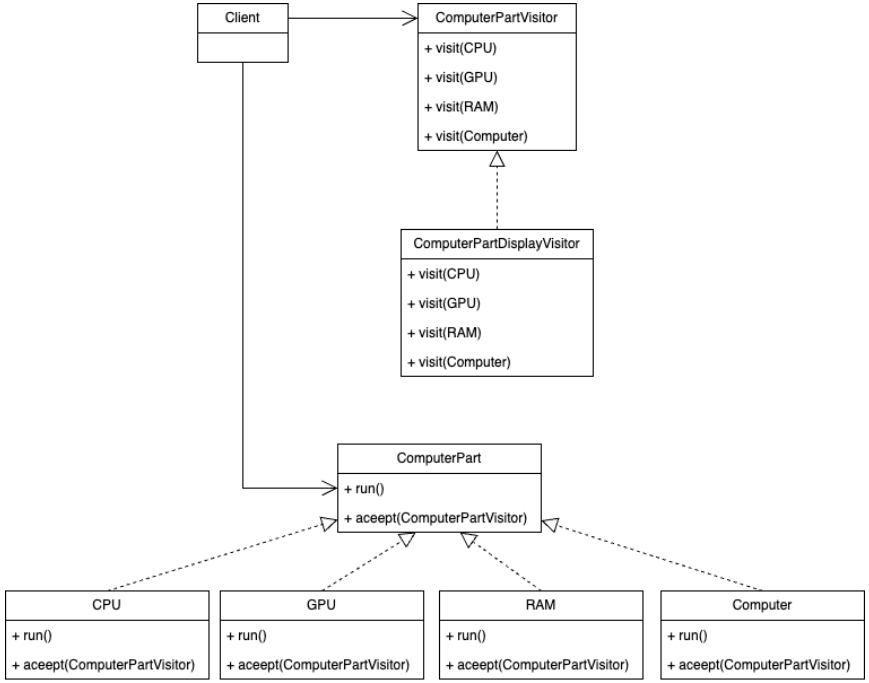
Practice #4: Implement Visitor

- In the skeleton code, we need to modify the classes (`ComputerPart`, `CPU`, `GPU`, `RAM`, `Computer`) to add new functionality to a `ComputerPart`
- This makes the code harder to manage
  - OCP violation
- Implement the Visitor
  - Add `accept()` in `ComputerPart`
  - Define `ComputerPartVisitor` interface
  - Define `ComputerPartDisplayVisitor` class to print message
    - Message : “Displaying CPU/GPU/RAM/Computer”

10 mins

Practice #4 : Visitor Diagram

- Add `accept()` in `ComputerPart`
- Define `ComputerPartVisitor` interface
- Define `ComputerPartDisplayVisitor` class to print message



Practice #5: Observer

- Open `behavioralPatterns/ObserverExample` with Android Studio
- There are 3 classes
  - Client.java: Main function
  - Display.java: Concrete observer
  - WeatherData.java: Concrete subject

Practice #5: Skeleton Code (1/3)

```
public class WeatherData {
    private int temperature;
    private int humidity;
    private int pressure;

    public void setMeasurements() {
        Random random = new Random();
        this.temperature = random.nextInt(0, 100);
        this.humidity = random.nextInt(0, 100);
        this.pressure = random.nextInt(0, 100);
    }
```

```
public int getTemperature() {
        return temperature;
    }

    public int getHumidity() {
        return humidity;
    }

    public int getPressure() {
        return pressure;
    }
}
```

Practice #5: Skeleton Code (2/3)

```
public class Display {
    private int temperature;
    private int humidity;
    private int pressure;
    public void update(int temperature, int humidity, int pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        display();
    }
    public void display() {
        System.out.println("Current conditions: " + temperature
                + "F degrees and " + humidity + "% humidity " + pressure + "
pressure");
    }
```

```
public int getTemperature() {
        return temperature;
    }

    public int getHumidity() {
        return humidity;
    }

    public int getPressure() {
        return pressure;
    }
}
```

Practice #5: Skeleton Code (3/3)

```
public class Client {
    public static void main(String[] args){
        WeatherData weatherData = new WeatherData();
        Display display1 = new Display();
        Display display2 = new Display();
        weatherData.setMeasurements();

        if(weatherData.getTemperature() != display1.getTemperature()){
            if(weatherData.getHumidity() != display1.getHumidity()){
                if(weatherData.getPressure() != display1.getPressure()){
                    display1.update(weatherData.getTemperature(), weatherData.getHumidity(), weatherData.getPressure());
                }
            }
        }
        ...
    }
}
```

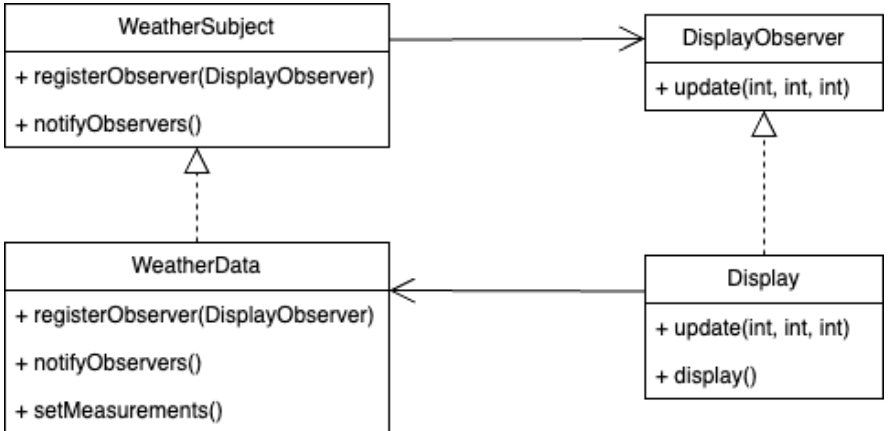
Practice #5: Implement Observer

- In the skeleton code, we need to keep checking to see if the value of `WeatherData` changes
- Implement the Observer
  - Define `WeatherSubject` (interface) and `DisplayObserver` (interface)
    - `WeatherSubject`: `registerObserver(DisplayObserver)`, `notifyObservers()`
    - `DisplayObserver`: `update(int, int, int)`
  - Modify `WeatherData`
    - Implement the `registerObserver(Display)`, and `notifyObservers()`
  - Modify `Display`

10 mins

Practice #5 : Observer Diagram

- Define `WeatherSubject` (interface) and `DisplayObserver` (interface)
  - `WeatherSubject`: `registerObserver(DisplayObserver)`, `notifyObservers()`
  - `DisplayObserver: update(int, int, int)`
- Modify `WeatheData`
  - Implement the `registerObserver(Display)`, and `notifyObservers()`
- Modify `Display`



Submissions

- Submit 2 zip files on eTL
  - Zip `structuralPatterns` directory
    - Submit to eTL
    - Week 12. Structural Patterns Practice Submission
  - Zip `behavioralPatterns` directory
    - Submit to eTL
    - Week 12. Behavioral Patterns Practice Submission
- Deadline: **11/26 23:59**

Thank You.
Any Questions?