AA 19/20 – Relatório Design Patterns 2

João Marques (A81826), José Pereira (A82880), Ricardo Petronilho (A81744)

23 de Março de 2020

1 INTRODUÇÃO

No âmbito da Unidade Curricular Arquitecturas Aplicacionais do 4º ano do Mestrado integrado em Engenharia Informática da Universidade do Minho, foi-nos proposta uma pesquisa sobre design patterns estruturais e comportamentais.

O presente relatório relata a investigação sobre *Structural* e *Behavioral Design Patterns*. Desta forma, foram dados à escolha os *Design Patterns*, *Flyweight* e *Decorator* como *Structural Designs*, *Memento*, *Mediator* e *Visitor* como *Behavioral Designs*. Destes foram escolhidos um de cada tipo, isto é, o *Decorator* e *Mediator*.

Inicialmente, será ilustrada a explicação dos design patterns, elaborada a partir da pesquisa realizada sobre os mesmos.

2 STRUCTURAL DESIGN PATTERN - DECORATOR

O *Design Pattern* estrutural *Decorator* vem resolver um problema recorrente no desenvolvimento de software, que trata-se da adição de novos comportamentos a objectos, sem necessidade de alterar bastante código, ou aumentar o número de dependências.

Com a finalidade de uma melhor percepção do problema que este design pattern resolve, vai-se seguir um exemplo real referido no website refactoring.guru.

Imagine-se um código responsável pela notificação de alguma informação. Deste modo, existirá um cliente e uma classe responsável por essa notificação, sendo neste momento apenas por email. Após o lançamento do produto, verifica-se a necessidade de **adicionar** novas formas de notificar, tais como: sms, facebook, slack, etc. A primeira solução que poderá ocorrer será utilizar herança, criando subclasses, específicas a cada tipo de notificação, isto é, a classe Email, SMS, Slack, etc..., e estas estenderem a classe responsável pela notificação. No entanto, esta solução contém muitas dependências e um crescimento de subclasses infinito para cada tipo de combinação de notificações, isto é, se se quisesse enviar por exemplo a combinação de SMS + Slack, teria que se criar uma classe específica para tal. Para além destes problemas, muitas linguagens não permitem estender mais do que uma classe, logo utilizar herança será uma má ideia.

Com a finalidade de uma melhor percepção da solução do problema, vai-se usar uma analogia com a realidade. A nível de software a solução para a necessidade de adicionar novas funcionalidades a um objecto, consiste em colocá-lo no "interior" de outros, denominados por "*wrappers*". Em analogia com a realidade, pode-se pensar na boneca *Matriosca*, isto é, a funcionalidade base, consiste na notificação por **Email**, correspondendo à boneca mais pequena da *Matriosca*. Se for necessário adicionar uma funcionalidade adicional ao **Email**, por exemplo a notificação por **SMS**, então vai-se ter uma boneca maior que a do **Email**, que corresponde ao **SMS**, e colocar a boneca mais pequena (**Email**) dentro da boneca do **SMS**. Ou seja, significa que sempre que se enviar a boneca do **SMS**, dentro dela vai sempre a boneca do **Email**, isto é, sempre que se envia o **SMS**, também se envia o **Email**.

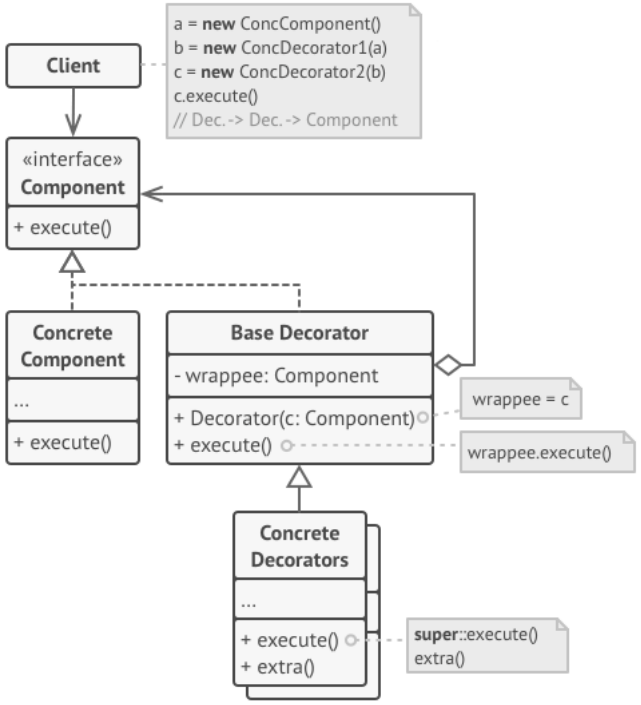

Figure 1. Implementação genérica do design pattern Decorator.

Após a analogia apresentada anteriormente, explicar-se-á agora em concreto a implementação do design pattern *Decorator*, seguindo a figura 1. A interface **Component** representa, tanto o **componente base**, bem como os seus decoradores ("**decorators**"). Isto é, existem as classes **ConcreteComponent** e **BaseDecorator**, que implementam a interface **Component**, ou seja, uniformizou-se tanto o componente, como os seus decoradores, facilitando desta forma a criação de camadas (análogo ao *design pattern Composite*), como irá ver-se mais adiante. Tal como foi referido, a classe **ConcreteComponent**, representa o componente base, ou seja, a funcionalidade base (**execute()**), que no exemplo apresentado consiste na notificação por **Email**. A classe **BaseDecorator**, representa todos os **decorators**, ou seja, faz o "embrulho"(*wrapped)* do componente, através de composição do mesmo (variável **wrappee**). A forma como se faz o embrulho ao componente é através do envio do mesmo pelo construtor. Importa realçar a importância de se ter definido a interface **Component**, pois a mesma é que permite a criação de camadas, uma vez que, a variável que está na classe **BaseDecorator** é do tipo **Component**, ou seja, tanto pode ser o componente (**ConcreteComponent**), como um decorador (**BaseDecorator**). A razão da existência da classe **BaseDecorator**, consiste, em que todos os **decorators**, contenham o componente, e a funcionalidade associada a este, uma vez que, o método **execute** da classe **BaseDecorator**, chama o **execute** do componente, tal como se pode verificar nas notas da figura 1. As classes concretas de cada **Decorator**, vão estender a superclasse **BaseDecorator**, herdando o método **execute()** (*override*), e adicionando a este a nova funcionalidade, com o método **extra()**. Importa realçar ainda, que cada nova funcionalidade extra que se pretenda, basta criar uma nova classe que estenda a **BaseDecorator**, redefinindo o método **execute()**, adicionando-lhe a nova funcionalidade.

Por fim, e não menos importante, o cliente, decide como utilizar os **componentes** e os respectivos **decoradores**, ou seja, se o cliente apenas necessitar de enviar notificação por email, apenas cria um **ConcreteComponent**, isto é, uma única camada. Por outro lado, se o cliente, quiser enviar a notificação por email e SMS, terá que criar o **ConcreteComponent**, de seguida, o decorador do SMS, **ConcreteDecoratorSMS**, e no construtor deste enviar o componente. De seguida, através da instância do **ConcreteDecoratorSMS**, chama o método **execute()**, criando desta forma, duas camadas. Analisando em concreto, o *flow* deste último exemplo, quando no **ConcreteDecoratorSMS**, chamar o **execute()**, este irá executar o **extra()** (funcionalidade adicional), e chamar o **execute()** do **BaseDecorator** (utilizando super.execute()), por sua vez, este irá chamar o **execute()** do **Component** (funcionalidade base), criando desta forma, duas camadas.

Assim, para uma melhor percepção do *design pattern*, a equipa implementou o mesmo em relação ao problema apresentado das notificações.

2.1 Notas em relação ao código do Decorator

O código do *Decorator* encontra-se no anexo A. Nos argumentos, bem como no retorno do método **execute()**, utilizou-se o tipo *Object*, de forma a facilitar a reutilização da interface **Component**.

Na classe **Client** mostra-se três situações de camadas diferentes, sendo respectivamente exemplos de uma, duas e três camadas. No primeiro exemplo, apenas tem-se o componente base, isto é, aquele que envia a notificação por email. De seguida, temos a adição de uma funcionalidade de notificar o cliente por Slack, ou seja, cria-se um decorador, onde se envia o componente base pelo construtor, criando-se duas camadas. Por fim, é feito um exemplo de três camadas, onde em relação ao caso anterior, cria-se mais um decorador, e envia-se, o decorador anterior como argumento, sendo que o decorador anterior, já trás o componente base "dentro"dele. Assim, através do último decorador, chamando o **execute()**, vai-se notificar em três camadas, isto é, Email, SMS, Slack.

3 BEHAVIORAL DESIGN PATTERN - MEDIATOR

O *Design Pattern* comportamental *Mediator* consiste no **desacoplamento** dos diversos componentes (objectos) entre si. Desta forma em vez de um objecto comunicar explicitamente com um outro, este fá-lo indirectamente através do **mediator**.

A implementação genérica do design pattern é a seguinte:

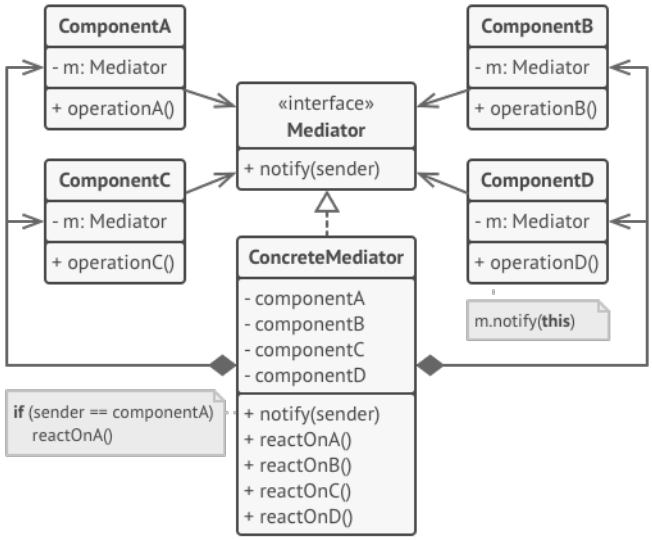

Figure 2. Implementação genérica do design pattern Mediator.

Partindo desta arquitectura decidiu-se implementar um exemplo em específico que consiste na gestão de componentes de uma interface gráfica, mais concretamente quando um utilizador clica num botão (Button) é obtido o seu nome, previamente introduzido numa caixa de texto de input (TextInput) e de seguida é apresentada uma mensagem de saudação - "Hello <nome>!- numa caixa de texto de output (TextBox).

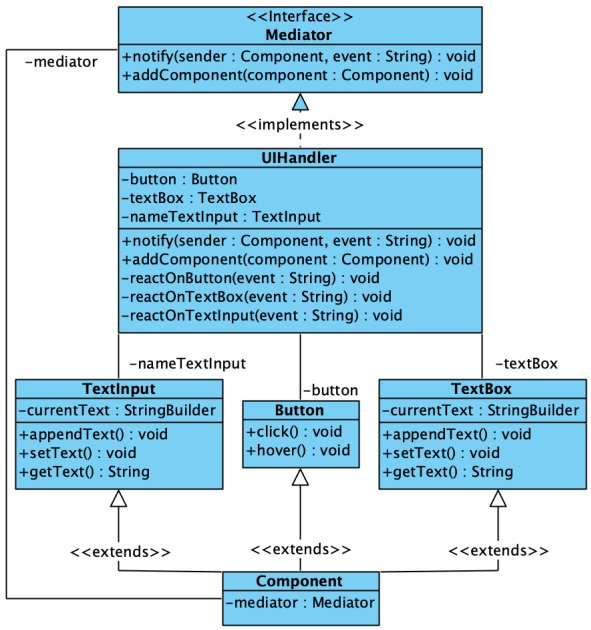

Figure 3. Diagrama de classes da implementação.

A implementação mais fácil, no entanto não a mais correta, seria o botão (Button) conhecer as caixas de texto (TextInput e TextBox), desta forma quando o mesmo fosse clicado simplesmente este acedia à caixa de texto de input (TextInput) para obter o nome e depois à caixa de texto de output (TextBox) para definir a mensagem. **No entanto, só neste pequeno exemplo, o botão (Button) já tem uma dependência de 1 para 2, sendo que se o botão interagisse com N componentes diferentes essa dependência também aumentava para N**. A manutenção deste código torna-se-ia cada vez mais complicada uma vez que a alteração de um componente ou da forma como esse componente interage com outro poderia exigir a alteração de várias classes.

Uma forma de resolver este problema é a introdução de um objecto mediador (Mediator) que fica responsável pela interacção dos vários componentes entre si. Assim todos os componentes (Button, TextInput e TextBox) **dependem unicamente do mediador (Mediator), reduzindo assim as dependências de 1 para N para 1 para 1**. Da mesma forma o mediador (Mediator) tem acesso a todos os componentes.

A implementação seria da seguinte maneira, o botão (Button) notifica o mediador (Mediator) que foi clicado (event), de seguida o mediador (Mediator) acede à caixa de texto de input (TextInput) para obter o nome e depois à caixa de texto de output (TextBox) para definir a mensagem. **Note-se que apesar de se continuar com uma dependência de 1 para N no Mediator, esta dependência é apenas neste objecto**.

4 ANEXOS

A DECORATOR

```java
public interface Component {
    Object execute(Object obj);
}

// ------------------------------------------------------------

public class ConcreteComponent implements Component {
    @Override
    public Object execute(Object obj) {
        System.out.println("Notificação por Email: " + obj);
        return null;
    }
}

// ------------------------------------------------------------

public class BaseDecorator implements Component{
    private Component wrappee;

    public BaseDecorator(Component c){
        this.wrappee = c;
    }

    @Override
    public Object execute(Object obj) {
        this.wrappee.execute(obj);
        return null;
    }
}

// ------------------------------------------------------------

public class ConcreteDecoratorSlack extends BaseDecorator {
    public ConcreteDecoratorSlack(Component c) {
        super(c);
    }

    @Override
    public Object execute(Object obj){
        super.execute(obj);
        extra(obj);
        return null;
    }
```

```
    public Object extra(Object obj){
        System.out.println("Notificação por Slack: " + obj);
        return null;
    }
}

// ------------------------------------------------------------

public class ConcreteDecoratorSMS extends BaseDecorator {
    public ConcreteDecoratorSMS(Component c) {
        super(c);
    }

    @Override
    public Object execute(Object obj){
        super.execute(obj);
        extra(obj);
        return null;
    }

    public Object extra(Object obj){
        System.out.println("Notificação por SMS: " + obj);
        return null;
    }
}

// ------------------------------------------------------------

public class Client {
    public static void main(String[] args){
        //  criamos um componente, que apenas notifica por email, 1 layer
        Component c = new ConcreteComponent();
        c.execute("Hello world!");

        System.out.println("-------------------------------------------");

        //  vou adicionar comportamento extra ao componente c, que é enviar
        //  notificação também por slack, 2 layers
        Component d1 = new ConcreteDecoratorSlack(c);
        d1.execute("Hello world!");

        System.out.println("-------------------------------------------");

        //  por fim, vou adicionar ainda mais comportamento, isto é, mais
        //  uma layer, para enviar por SMS, 3 layers
        Component d2 = new ConcreteDecoratorSMS(d1);
        d2.execute("Hello world!");
    }
```

```
}

// ------------------------------------------------------------

Output:
Notificação por Email: Hello world!
-----------------------------------------
Notificação por Email: Hello world!
Notificação por Slack: Hello world!
-----------------------------------------
Notificação por Email: Hello world!
Notificação por Slack: Hello world!
Notificação por SMS: Hello world!
```

B MEDIATOR

```
public interface Mediator {
    void notify(Component sender, String event);
    void addComponent(Component component);
}

// ------------------------------------------------------------

public class UIHandler implements Mediator {
    private Button button;
    private TextBox textBox;
    private TextInput nameTextInput;

    @Override
    public void addComponent(Component component) {
        if (component instanceof Button) button = (Button) component;
        else if (component instanceof TextBox) textBox = (TextBox) component;
        else if (component instanceof TextInput) nameTextInput = (TextInput) component;
    }

    @Override
    public void notify(Component sender, String event) {
        if (sender.equals(button)) reactOnButton(event);
        else if (sender.equals(textBox)) reactOnTextBox(event);
        else if (sender.equals(nameTextInput)) reactOnTextInput(event);
    }

    private void reactOnButton(String event) {
        switch (event) {
            case "click":
                String name = nameTextInput.getText();
                textBox.setText("Hello " + name + "!");
                System.out.println(textBox.getText());
                break;
            case "hover":
                textBox.setText("Mouse is over button!");
                System.out.println(textBox.getText());
                break;
            default:
                break;
        }
    }

    private void reactOnTextBox(String event) {}

    private void reactOnTextInput(String event) {}
}
```

```
// ------------------------------------------------------------

public class Component {
    protected Mediator mediator;

    public Component(Mediator mediator) {
        this.mediator = mediator;
    }
}

// ------------------------------------------------------------

public class Button extends Component {

    public Button(Mediator mediator) {
        super(mediator);
    }

    public void click() {
        this.mediator.notify(this, "click");
    }

    public void hover() {
        this.mediator.notify(this, "hover");
    }
}

// ------------------------------------------------------------

public class TextBox extends Component {
    private StringBuilder currentText;

    public TextBox(Mediator mediator) {
        super(mediator);
        this.currentText = new StringBuilder();
    }

    public void appendText(String text) {
        this.currentText.append(text);
    }

    public void setText(String text) {
        this.currentText.setLength(0);
        this.currentText.append(text);
    }
```

```
    public String getText() {
        return this.currentText.toString();
    }
}

// ------------------------------------------------------------

public class TextInput extends Component {
    private StringBuilder currentText;

    public TextInput(Mediator mediator) {
        super(mediator);
        this.currentText = new StringBuilder();
    }

    public void appendText(String text) {
        this.currentText.append(text);
    }

    public void setText(String text) {
        this.currentText.setLength(0);
        this.currentText.append(text);
    }

    public String getText() {
        return this.currentText.toString();
    }
}

// ------------------------------------------------------------

public class Main {
    public static void main(String[] args) {

        Mediator mediator = new UIHandler(); /** Acts as a mediator between UI components
        Button button = new Button(mediator);
        TextBox textBox = new TextBox(mediator);
        TextInput textInput = new TextInput(mediator);
        mediator.addComponent(button);
        mediator.addComponent(textBox);
        mediator.addComponent(textInput);

        textInput.setText("Nestor"); /** Simular a introdução do nome na UI. */
        button.hover(); /** Simular passar o rato em cima do botão. */
        button.click(); /** Simular o clicar no botão. */
    }
}
```

```
// ------------------------------------------------------------

Output:
Mouse is over button!
Hello Nestor!
```