

Trabalho - Grupo 07 - Mediator, Memento, Observer

Alunos:

- 01- Vinicius Rodrigues Cardoso Silva (202000632)**
- 02- Paloma Corrêa Alves (202000716)**
- 03- Matheus Barros Crisóstomo (202058447)**
- 04- Vinicius Maia Azevedo de Oliveira (202003404)**
- 05- Victor Laranjeira de Oliveira (201902162)**
- 06- Filipe Melo da Silva (202000240)**
- 07- Mylena Rodrigues Soares do Monte (202004142)**
- 08- Luís Eduardo da Silva Gomes (202004091)**
- 09- Dennys Gabriel Vasconcelos de Oliveira (202005054)**
- 10- João Victor Lemos (202002341)**

Turno: Manhã - CCO Piedade

Professor:

Diogenes Carvalho Matias

Sumário

Mediator Design Pattern

1.1 Conceito	2
1.2 Propósito	2
1.3 Vantagens e Desvantagens	2
1.4 Estrutura	3
1.5 Aplicação Prática	3
1.6 Referências	7

Memento Design Pattern

2.1 Conceito	7
2.2 Propósito	7
2.3 Vantagens e desvantagens	8
2.4 Estrutura	8
2.5 Implementação	9
2.6 Referências	11

Observer Design Pattern

3.1 Conceito	11
3.2 Propósito	12
3.3 Vantagens e Desvantagens	12
3.4 Estrutura	12
3.5 Aplicações Práticas	14
3.6 Referências	22

1. Mediator Design Pattern

1.1 Conceito

Para facilitar o desenvolvimento, a manutenção e manter o código limpo e legível, grandes aplicações procuram seguir princípios SOLID, padrões de projetos e outras recomendações de boas práticas, como a desacoplação dos objetos.

Durante o desenvolvimento de um software, é comum ocorrer situações em que precisamos referenciar algumas classes para consumir seus atributos ou métodos. Porém, à medida que a complexidade da funcionalidade expande, novas classes são referenciadas, criando uma dependência cada vez maior entre elas.

O Mediator consiste em um **Design Pattern** que provê um mecanismo de encapsulamento das interações que ocorrem entre diferentes objetos. Dessa forma, quando dois objetos precisam interagir, a comunicação é realizada exclusivamente por meio do Mediator.

1.2 Propósito

O objetivo dessa abordagem é alcançar um baixo nível de acoplamento na arquitetura do software, já que os objetos passam a não referenciar diretamente outros objetos. Podemos afirmar, portanto, que o Mediator é adequado para cenários em que muitas classes referenciam muitas classes. Quando isso ocorre, uma simples alteração pode gerar um impacto significativo no software, uma vez que o acoplamento é elevado.

Dessa forma, o Mediator Pattern gerencia as interações de diferentes objetos, por meio de uma classe mediadora que centraliza todas as interações entre os objetos, visando diminuir o acoplamento e a dependência entre eles. Com isso, neste padrão, os objetos não conversam diretamente entre eles, toda comunicação precisa passar pela classe mediadora.

Usa-se o mediator quando se tem como objetivo:

1. Diminuir ou extinguir o acoplamento direto entre as classes que poderiam estar diretamente acopladas.
2. Simplificar comunicações de muitos-para-muitos para comunicações um-para-muitos.

Resumidamente, o mediator é um objeto que visa centralizar a comunicação de vários objetos que, se não houvesse o mediator, estariam intimamente acoplados.

1.3 Vantagens e Desvantagens

Ao utilizar o padrão mediator os participantes da rede de comunicação são desacoplados, ou seja, desacopla objetos que poderiam estar firmemente acoplados. Os relacionamentos passam a ser de um para muitos e a política de comunicação fica centralizada no mediador

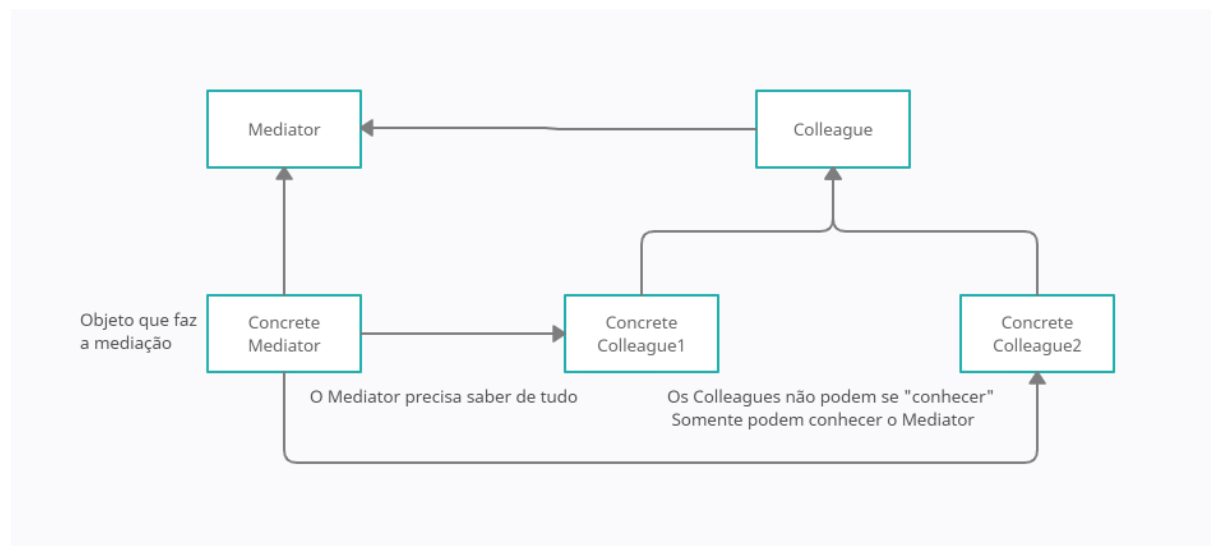
que organiza e facilita a manutenção do código. Nesse sentido, facilita a reutilização de objetos; facilita a adição de novos mediadores e classes participantes na comunicação; encapsula a comunicação entre objetos. Assim, torna o sistema mais flexível. No entanto, alguns dos problemas que surgem com a implementação é a possibilidade de gargalos de desempenho devido à centralização da política de comunicação, e também tendência do aumento da complexidade do código com o tempo.

1.4 Estrutura

O padrão Mediator consiste de duas figuras principais: o Mediator e o Colleague. O Mediator recebe mensagens de um Colleague, define qual protocolo utilizar e então envia a mensagem. O Colleague define como receberá uma mensagem e envia uma mensagem para um Mediator.

A estrutura deste padrão de projeto é de fácil compreensão, composta por apenas quatro elementos:

- **Mediator:** Interface que define os métodos de comunicação entre os objetos;
- **Concrete Mediator:** Classe concreta que implementa a Interface Mediator, está ciente de todos os colegas e de suas intercomunicações. Qualquer comunicação entre colegas acontece apenas por meio de um mediador
- **Colleague:** Interface que define os métodos referente às “intenções” dos objetos, ou seja, que representa os componentes do nosso sistema
- **Concrete Colleague:** Classe concreta que implementa a Interface Colleague e estão dispostas a interagir umas com as outras.



1.5 Aplicação Prática

Como exemplo, iremos criar um software de chat.

Primeiro, cria-se a interface. Nós a chamamos de ChatMediator. Nele, vão ter dois métodos: enviarMensagem que recebe a mensagem de um usuário e o adicionarUsuario onde um usuário será adicionado a uma lista.

```
01| public interface ChatMediator {
02|     public void enviarMensagem(String mensagem, Usuario user);
03|
04|     public void adicionarUsuario(Usuario user);
05| }
```

Cria-se uma classe abstrata Usuario e iremos adicionar atributos e métodos onde todo usuário deverá ter. Criamos dois atributos: o nome e um atributo que faz referência ao mediator, ou seja, todo usuário irá usar esse mediator para realizar comunicação. Depois criamos um construtor para inicializar os atributos. O usuário deverá enviar e receber a mensagem, então por isso, criamos dois métodos, uma para enviar a mensagem e outra para receber a mensagem.

```
01| public abstract class Usuario {
02|     protected String nome;
03|     protected ChatMediator mediator;
04|
05|     public Usuario(String nome, ChatMediator mediator) {
06|         this.nome = nome;
07|         this.mediator = mediator;
08|     }
09|
10|     public abstract void enviarMensagem(String mensagem);
11|
12|     public abstract void receberMensagem(String mensagem);
13| }
```

Agora iremos criar a classe para implementação da classe ChatMediator. Teremos que implementar a classe com os dois métodos feitos no ChatMediator e modificá-los para que sejam implementados, substituindo o “ponto e vírgula” por “chaves”. Antes de cada método, temos que colocar o @Override para sobrescrever o método. Criamos um atributo lista para armazenar os usuários (importe a biblioteca java.util.List para usá-lo) e geramos um construtor para inicializar os usuários em um ArrayList (importe a biblioteca java.util.ArrayList para usá-lo). No método adicionarUsuario colocamos o comando .add para adicionar, no caso, o usuário na lista. No método enviarMensagem, criamos um laço que irá percorrer toda a lista de usuários. Dentro do laço, colocamos uma condição que se o usuário for diferente daquele que enviou a mensagem, esses usuários irão receber essa mensagem.

```
01| import java.util.ArrayList;
02| import java.util.List;
03|
04| // Mediator Implementação
05|
```

```
06| public class ChatMediatorImp implements ChatMediator {
07|     private List<Usuario> usuarios;
08|
09|     public ChatMediatorImp() {
10|         this.usuarios = new ArrayList<Usuario>();
11|     }
12|
13|     @Override
14|     public void enviarMensagem(String mensagem, Usuario user) {
15|         // Percorre a lista de usuários
16|         for(Usuario u : this.usuarios) {
17|             // Se o usuário for diferente do que está usando
18|             // a mensagem é enviada.
19|             if(u != user) {
20|                 u.receberMensagem(mensagem);
21|             }
22|         }
23|     }
24|
25|     @Override
26|     public void adicionarUsuario(Usuario user) {
27|         usuarios.add(user);
28|     }
29|
30| }
```

Agora criamos a implementação da classe Usuario. Essa classe irá estender nossa classe abstrata Usuario, tendo que trazer os métodos da classe Usuario e que, ao colocar os métodos na nossa classe, teremos que retirar a menção abstrata e fazê-los métodos implementáveis. Lembrar de colocar `@Override` antes de cada método para dizer que irá sobrescrever. Agora criamos um construtor, recebendo os mesmos atributos do nosso construtor da classe Usuario, e passá-los ao construtor da superclasse através do `super`. No método `enviarMensagem`, iremos mostrar na tela o nome do usuário e mostrar que aquele usuário está enviando a mensagem. Depois iremos invocar o mediator enviando a mensagem e o, como referência de classe, o usuário. Já no método `receberMensagem`, iremos mostrar na tela o nome(s) do(s) usuário(s) que irá(ão) receber a mensagem.

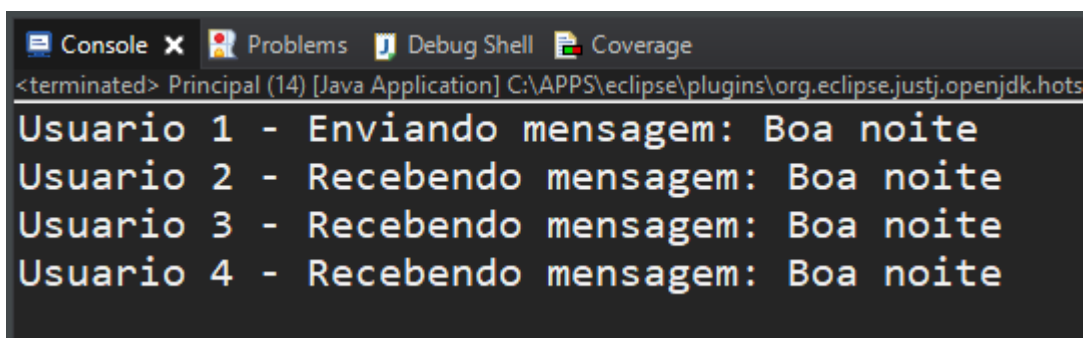
```
01| // Usuário Implementação
02|
03| public class UsuarioImp extends Usuario {
04|     public UsuarioImp(String nome, ChatMediator mediator) {
05|         super(nome, mediator);
06|     }
07|
08|     @Override
09|     public void enviarMensagem(String mensagem) {
10|         System.out.println(super.nome + " - Enviando mensagem: " +
mensagem);
```

```
11|         super.mediator.enviarMensagem(mensagem, this);
12|     }
13|
14|     @Override
15|     public void receberMensagem(String mensagem) {
16|         System.out.println(super.nome + " - Recebendo mensagem: " +
mensagem);
17|     }
18| }
```

Para finalizar, criaremos a classe Principal. Agora iremos instanciar todas as classes do nosso mediator. A primeira delas será o ChatMediator, que é a nossa interface, e vamos instanciar a um objeto concreto do tipo da implementação. Em seguida, vamos criar nossos usuários, passando o nome do usuário e o mediator que chamamos de “chat”. Vamos adicionar os usuários ao chat por meio do método adicionarUsuario que criamos anteriormente. Por fim, vamos simular uma situação onde o usuário 1 irá enviar uma mensagem para todos os outros.

```
01| public class Principal {
02|     public static void main(String[] args) {
03|         ChatMediator chat = new ChatMediatorImp();
04|
05|         Usuario u1 = new UsuarioImp("Usuario 1", chat);
06|         Usuario u2 = new UsuarioImp("Usuario 2", chat);
07|         Usuario u3 = new UsuarioImp("Usuario 3", chat);
08|         Usuario u4 = new UsuarioImp("Usuario 4", chat);
09|
10|         chat.adicionarUsuario(u1);
11|         chat.adicionarUsuario(u2);
12|         chat.adicionarUsuario(u3);
13|         chat.adicionarUsuario(u4);
14|
15|         u1.enviarMensagem("Boa noite");
16|     }
17| }
```

Executando o nosso código, o resultado será este:



```
<terminated> Principal (14) [Java Application] C:\APPS\eclipse\plugins\org.eclipse.justj.openjdk.hotspot
Usuario 1 - Enviando mensagem: Boa noite
Usuario 2 - Recebendo mensagem: Boa noite
Usuario 3 - Recebendo mensagem: Boa noite
Usuario 4 - Recebendo mensagem: Boa noite
```

1.6 Referências

Comportamental - Mediator - YouTube, CRISTIANO ALMEIDA,

["https://www.youtube.com/watch?v=sqEwg0rGqX"](https://www.youtube.com/watch?v=sqEwg0rGqX), acessado em maio de 2021.

Design Patterns GoF – Mediator (2017), ANDRÉ CELESTINO,

["https://www.andrecelestino.com/delphi-design-patterns-mediator/"](https://www.andrecelestino.com/delphi-design-patterns-mediator/), acessado em maio de 2021.

Mediator Pattern com MediatR no ASP.NET Core (2020), LEO PRANGE,

["https://www.treinaweb.com.br/blog/mediator-pattern-com-mediatr-no-asp-net-core"](https://www.treinaweb.com.br/blog/mediator-pattern-com-mediatr-no-asp-net-core), acessado em maio de 2021.

Mediator Teoria - Padrões de Projeto - Parte 35/45 - YouTube, OTÁVIO MIRANDA,

["https://www.youtube.com/watch?v=fb7NrdCo4Ko"](https://www.youtube.com/watch?v=fb7NrdCo4Ko), acessado em maio de 2021.

O padrão de projeto Mediator na prática (2011), JOSÉ CARLOS MACORATTI,

["https://imasters.com.br/dotnet/o-padrao-de-projeto-mediator-na-pratica"](https://imasters.com.br/dotnet/o-padrao-de-projeto-mediator-na-pratica), acessado em maio de 2021.

2. Memento Design Pattern

2.1 Conceito

Memento é um padrão de software responsável por reaver estados passados de um objeto, capturando e externalizando um estado interno sem quebrar o encapsulamento do programa, o objetivo deste padrão de software é garantir que haja a possibilidade de desfazer certas ações executadas pelo usuário sem comprometer seu trabalho.

2.2 Propósito

Ao utilizar um programa é de se esperar que em um dado momento você irá cometer algum erro, tendo isso em mente é desejável que haja alguma funcionalidade para que se possa retroceder a ação errônea, aproveitando-se de ponto ou (CheckPoints) onde é possível salvar automaticamente cada etapa executada para então retornar a um momento anterior.

As informações que são salvas não podem ser facilmente acessadas de forma comum pelo programa, pois caso isso aconteça o encapsulamento pode e irá ser comprometido, prejudicando a confiabilidade da aplicação, já que os estados terão de ser capturados diretamente das classes que deveriam estar "ocultas". Há vários cenários onde tal padrão é utilizado, considere, por exemplo, um editor de imagens onde você quer mesclar e conectar duas imagens uma por cima da outra, ao mover a imagem de cima, a que estiver embaixo irá se comportar conforme o desejado.

Para que a manipulação da imagem funcione é preciso um objetoCalculo(); responsável por manter a conexão e calcular todas as manipulações que estão ocorrendo, seja ela

opacidade, mudança na coloração, movimentação na tela, adição de mais imagens, etc. O objeto é capaz até mesmo de rearranjar os elementos ao estado anterior mantendo tudo de forma apropriada.

O suporte a operações de desfazer neste aplicativo não é tão fácil quanto parece. Uma maneira de desfazer a operação "mover" é armazenar a distância de movimento original e, em seguida, mover o objeto de volta para a distância equivalente. No entanto, isso não garante que todos os objetos aparecerão em suas posições anteriores. Suponha que a conexão esteja solta. Nesse caso, simplesmente mover a imagem de volta à sua posição original não produz necessariamente o efeito desejado.

Utilizar o `objetoCalculo()`; para resolver tais operações de desfazer pode não ser o suficiente, pois uma relação mais profunda com essa interface é precisa. Este problema pode ser resolvido usando o padrão memento, que é um objeto que armazena um instantâneo do estado interno de outro objeto, o mecanismo que é responsável por solicitar uma operação desfazer solicita um memento do originador, quando é preciso fazer um `checkPoint` do estado do originador. O Originador por sua vez chama o memento e recupera um estado anterior, podendo então retornar a estrutura prévia de seu estado ao `objetoCalculo()`; por exemplo.

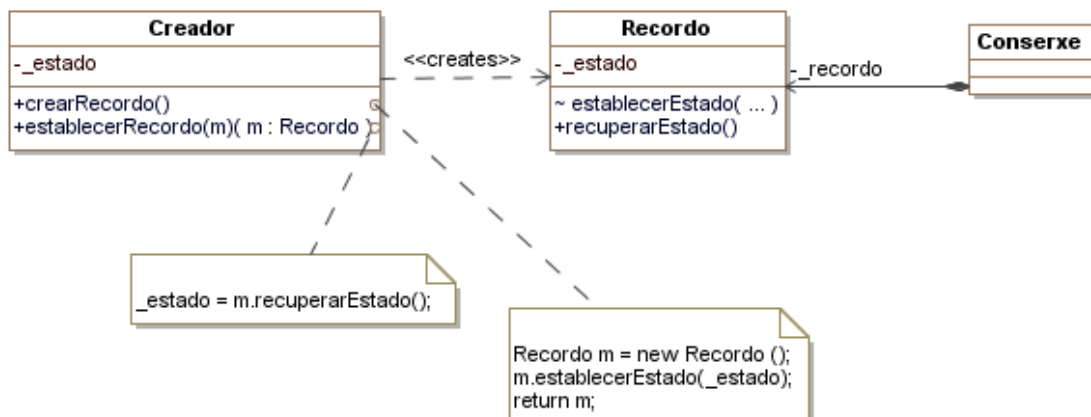
2.3 Vantagens e desvantagens

A medida que se usa uma aplicação como uma IDE, ela irá executar sua tarefa de manter o memento sempre "cheio" garantindo a confiabilidade de um possível retorno a um estado anterior, com a ajuda de um Originador para criar `checkPoint` e um Caretaker que mantém a guarda do memento. A grande desvantagem desse design pattern é a acumulação de mementos ao longo do tempo, causando um uso excessivo de memória podendo provocar lentidão no sistema.

2.4 Estrutura

As etapas para desfazer uma operação seguem esse ritmo:

1. O mecanismo de desfazer operações solicita um memento ao `objetoCalculo()`;
2. O `objetoCalculo()` cria e devolve à instância de uma classe `solverState()`; O `solverState()` contendo então estruturas de dados atuais do `objetoCalculo()`;
3. Quando qualquer tipo de operação for desfeita, o editor retorna o `solverState()`; ao `objetoCalculo()`;
4. O `objetoCalculo()`; por sua vez converte suas estruturas internas para o estado anterior ao ato de mesclar imagens por exemplo.



2.5 Implementação

A classe a seguir é o memento que irá armazenar um único estado do objeto originador, para simplificar, a classe `textoMemento` manterá apenas uma `String` e um getter representando o texto.

```

1      public class TextoMemento {
2          protected String estadoTexto;
3
4          public TextoMemento(String texto) {
5              estadoTexto = texto;
6          }
7
8          public String getTextoSalvo() {
9              return estadoTexto;
10         }
11     }
  
```

Após a criação do memento, iremos partir para o desenvolvimento do `CareTaker`, encarregado por manter o memento, o guardando por um certo período de tempo até que ele, caso seja pedido, devolver para o originador. Neste caso o `CareTaker` criará uma lista onde irá armazenar vários estados de memento, dando então ao Originador a possibilidade de executar mais de uma recuperação.

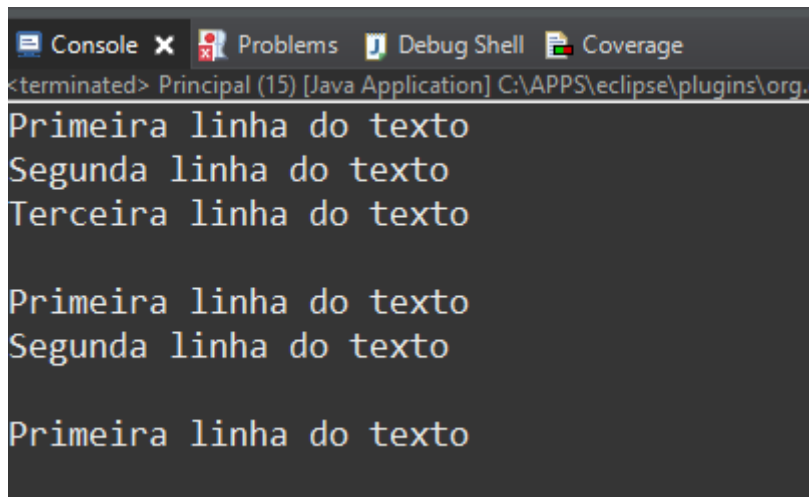
```
1 public class TextoCareTaker {
2     protected ArrayList<TextoMemento> estados;
3
4     public TextoCareTaker() {
5         estados = new ArrayList<TextoMemento>();
6     }
7
8     public void adicionarMemento(TextoMemento memento){
9         estados.add(memento);
10    }
11
12    public TextoMemento getUltimoEstadoSalvo() {
13        if (estados.size() <= 0) {
14            return new TextoMemento("");
15        }
16        TextoMemento estadoSalvo = estados.get(estados.size() - 1);
17        estados.remove(estados.size() - 1);
18        return estadoSalvo;
19    }
20 }
```

A classe texto a seguir permitirá escrever um texto, desfazê-lo e depois irá exibir o mesmo no terminal. Seguindo o padrão dos códigos acima, o texto irá ser salvo e então a alteração será executada, só então a ação de desfazer o texto poderá acontecer, o CareTaker será chamado que irá enviar o último estado salvo ao Originador, possibilitando a restauração do texto.

```
1 public class Texto {
2     protected String texto;
3     TextoCareTaker caretaker;
4
5     public Texto() {
6         caretaker = new TextoCareTaker();
7         texto = new String();
8     }
9
10    public void escreverTexto(String novoTexto) {
11        caretaker.adicionarMemento(new TextoMemento(texto));
12        texto += novoTexto;
13    }
14
15    public void desfazerEscrita() {
16        texto = caretaker.getUltimoEstadoSalvo().getTextoSalvo();
17    }
18
19    public void mostrarTexto() {
20        System.out.println(texto);
21    }
22 }
```

E por fim a interface que irá dar os comando para visualizar o memento funcionando de acordo com o esperado.

```
1 public static void main(String[] args) {  
2     Texto texto = new Texto();  
3     texto.escreverTexto("Primeira linha do texto\n");  
4     texto.escreverTexto("Segunda linha do texto\n");  
5     texto.escreverTexto("Terceira linha do texto\n");  
6     texto.mostrarTexto();  
7     texto.desfazerEscrita();  
8     texto.mostrarTexto();  
9     texto.desfazerEscrita();  
10    texto.mostrarTexto();  
11    texto.desfazerEscrita();  
12    texto.mostrarTexto();  
13    texto.desfazerEscrita();  
14    texto.mostrarTexto();  
15 }
```



```
<terminated> Principal (15) [Java Application] C:\APPS\eclipse\plugins\org...  
Primeira linha do texto  
Segunda linha do texto  
Terceira linha do texto  
  
Primeira linha do texto  
Segunda linha do texto  
  
Primeira linha do texto
```

2.6 Referências

GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

BRIZENO, Marcos. Memento: Desenvolvimento de software. *In*: Mão na massa: Memento. [S. l.], 5 nov. 2011. Disponível em: <https://brizeno.wordpress.com/2011/11/05/mao-na-massa-memento/>. Acesso em: 13 maio 2021.

<https://brizeno.wordpress.com/2011/11/05/mao-na-massa-memento/><https://bit.ly/3vYrGbO>

3. Observer Design Pattern

3.1 Conceito

O observer é mais um dos vários Design Patterns mais utilizados e aceitos dentro de projetos nas linguagens de programação mais modernas, além de trazer vários frameworks consigo para tratar de interfaces de usuário (UI) e atualização de estados de um objeto. Basicamente ele é um padrão de projetos que define um tipo de “audição” aos eventos selecionados para serem observados. Na maior parte dos frameworks e bibliotecas Java, temos a presença constante do “ouvinte de eventos” que capturam determinados eventos de usuário, tais como cliques, teclas pressionadas, focos e etc. Os ouvintes/observadores mais utilizados são:

- `java.util.Observer/java.util.Observable`
- `java.util.EventListener`
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

3.2 Propósito

- Desacoplar objetos emissores e objetos receptores pela definição de uma interface para sinalizar mudanças em subjects.
- Definir um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

3.3 Vantagens e Desvantagens

- Fornece suporte para comunicação do tipo broadcast. A notificação que um subject envia não precisa especificar seu receptor, ela é transmitida automaticamente para todos os objetos interessados que a subscreveram. O subject não se preocupa com quantos objetos interessados existem, sua única responsabilidade é notificar todos os seus observers. Isso dá a liberdade de acrescentar e remover observers a qualquer momento.
- Gera um acoplamento abstrato entre Subject e Observer, tudo que o subject sabe é que ele tem uma lista de observadores, cada um seguindo a interface simples da classe abstrata Observer. O subject não conhece a classe concreta de nenhum observer. Assim, o acoplamento entre o subject e os observers é abstrato e mínimo.

- Gera atualizações inesperadas, pelo fato de um observer não ter conhecimento da presença dos outros, assim eles podem ser cegos para o custo global de mudança do subject. Uma operação usada de forma inocente no subject pode causar uma cascata de atualizações nos observers e seus objetos dependentes.

3.4 Estrutura

O padrão de projeto Observer promove o acoplamento fraco entre um objeto-assunto e objetos observadores — um objeto-assunto notifica os objetos observadores quando o assunto altera o estado. Quando notificado pelo assunto, os observadores mudam em resposta. Um assunto pode notificar vários observadores; portanto, o assunto tem um relacionamento de um-para-muitos com os observadores.

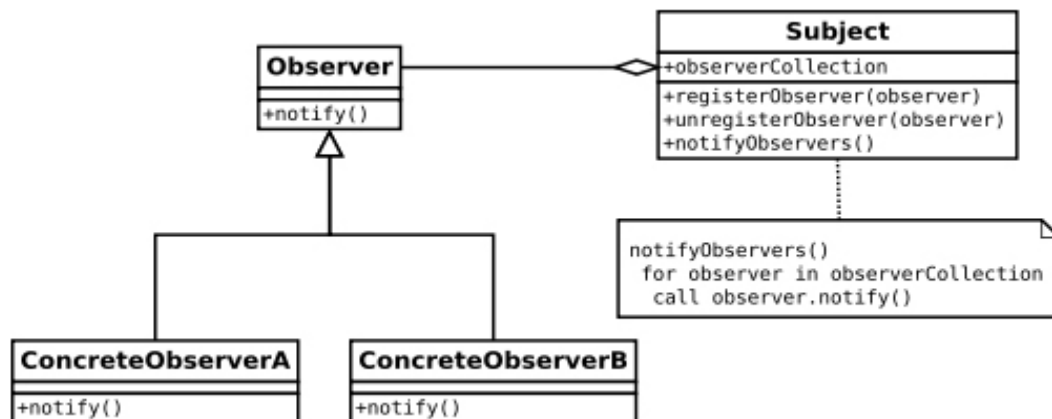
Elementos :

- Sujeito (Subject): – conhece seus observadores;
- Observador (Observer): – define uma interface para objetos que devem ser notificados de mudanças em um Subject;
- Sujeito Concreto (ConcreteSubject): – armazena estados de interesse dos objetos ConcreteObserver; – envia notificações para seus Observers quando esses estados mudam.
- Observador Concreto (ConcreteObserver): – mantém uma referência para um objeto ConcreteSubject;
 - armazena estado que deve ficar consistente com o do subject;
 - implementa a interface de atualização do Observer para manter-se consistente com o do Subject.
- Colaborações

O ConcreteSubject notifica seus observadores sempre que ocorre uma mudança que poderia fazer com que o estado dos observadores inconsistentes com o seu próprio estado. Que ao ser informado de mudanças em ConcreteSubject um objeto ConcreteObserver pode consultar o ConcreteSubject para obter informações e usá-las para reconciliar seu estado com o subject.

Esse padrão é bem comum no universo de desenvolvimento Java. Suponha, por exemplo, que você tenha um sistema que gerencia o seu regime. Com o passar dos dias você usará o sistema e informará diariamente o seu peso e as alterações do mesmo. Você terá uma classe A que receberá essa informação e que irá atrelar à ela mesma uma classe B (O observer), para notificar toda alteração que aconteça. Quando a mesma ultrapassar um limite definido pelo IMC de massa corpórea o observer será avisado e então tomará as medidas cabíveis. Este é apenas um de muitos exemplos possíveis para o padrão, e,

independente do caso, o modelo representado pode explicar melhor como suas classes poderão se dividir.



3.5 Aplicações Práticas

Exemplo 1: Criação de um editor de mensagens do tipo Subject e 3 assinantes Observer, com o Publisher publicando periodicamente um mensagem para os Observer inscritos ou anexados, assim imprimindo uma mensagem atualizada no console.

Subject.java

```

public interface Subject
{
    public void attach(Observer o);
    public void detach(Observer o);
    public void notifyUpdate(Message m);
}
    
```

MessagePublisher.java

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

public class MessagePublisher implements Subject {

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

    @Override
    public void attach(Observer o) {
        observers.add(o);
    }

    @Override
    public void detach(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyUpdate(Message m) {
        for(Observer o: observers) {
            o.update(m);
        }
    }
}
```

Observer.java

```
public interface Observer
{
    public void update(Message m);
}
```

MessageSubscriberOne.java

```
public class MessageSubscriberOne implements Observer
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberOne :: " + m.getMessageContent());
    }
}
```


MessageSubscriberTwo.java

```
public class MessageSubscriberTwo implements Observer
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberTwo :: " + m.getMessageContent());
    }
}
```

MessageSubscriberThree.java

```
public class MessageSubscriberThree implements Observer
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberThree :: " + m.getMessageContent());
    }
}
```

Após isso vem se o Objeto de estado que é um objeto imutável que, desse modo, não permite alteração por nenhum tipo de classe.

Message.java

```
public class Message
{
    final String messageContent;

    public Message (String m) {
        this.messageContent = m;
    }

    public String getMessageContent() {
        return messageContent;
    }
}
```

Main.java

```
public class Main
{
    public static void main(String[] args)
    {
        MessageSubscriberOne s1 = new MessageSubscriberOne();
        MessageSubscriberTwo s2 = new MessageSubscriberTwo();
        MessageSubscriberThree s3 = new MessageSubscriberThree();

        MessagePublisher p = new MessagePublisher();

        p.attach(s1);
        p.attach(s2);

        p.notifyUpdate(new Message("First Message")); //s1 and s2 will receive the message

        p.detach(s1);
        p.attach(s3);

        p.notifyUpdate(new Message("Second Message")); //s2 and s3 will receive the message
    }
}
```

Saída do Programa:

Console

```
MessageSubscriberOne :: First Message
MessageSubscriberTwo :: First Message

MessageSubscriberTwo :: Second Message
MessageSubscriberThree :: Second Message
```

3.6 Referências

Padrões em Java: Observer, Diogo Souza,
“<http://www.linhadecodigo.com.br/artigo/3643/padroes-em-java-observer.aspx>”,
acessado em maio de 2021.

Observer Pattern na Dev Media (2010), Caio Humberto,
“<https://www.devmedia.com.br/design-patterns-observer/16875>”,
acessado em maio de 2021.

Trabalhando com o Pattern Observer na Plataforma Java EE (2015), Higor Medeiros,
“<https://www.devmedia.com.br/trabalhando-com-o-pattern-observer-na-plataforma-java-ee/3149>”, acessado em maio de 2021.

Observer (2016), Patrick Helder Alvarenga
Belém, “<http://patrick.ison.com.br/questions/6/idea/18>”, acessado em maio de 2021.

Observer Design Pattern, Lokesh Gupta,
“<https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/>”,
acessado em maio de 2021.

Observer Design Pattern em Java, Pankaj.
“<https://www.journaldev.com/1739/observer-design-pattern-in-java>”,
Acessado em Maio de 2021.

Observer Design Pattern(How To Do In Java).
“<https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/>”
Acessado em Maio de 2021.

The Observer Pattern in Java, Predhag.
“<https://www.baeldung.com/java-observer-pattern>”
Acessado em Maio de 2021.