



GRUPO POO 07

PIEDADE

- 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)

TABLE OF CONTENTS

01

Mediator
Design Pattern

02

Memento
Design Pattern

03

Observer
Design Pattern

04

Referências



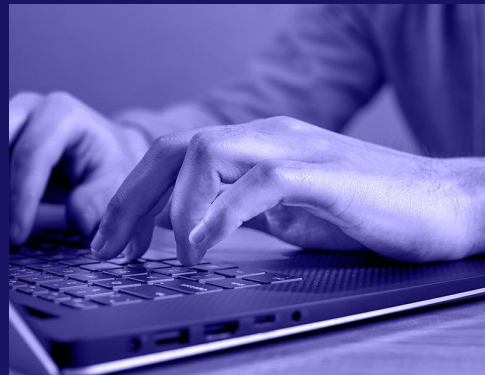
01

Mediator Design Pattern

CONCEITO

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.



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.

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.

PROPÓSITO

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.

VANTAGENS

- Desacopla objetos que poderiam estar firmemente acoplados;
- Relacionamentos passam a ser de um para muitos;
- A política de comunicação fica centralizada no mediador que organiza;
- Facilita a manutenção do código;
- Facilita a reutilização de objetos;
- Facilita a adição de novos mediadores e classes participantes na comunicação;
- Encapsula a comunicação entre objetos;
- Torna o sistema mais flexível.

DESVANTAGENS

- A possibilidade de gargalos de desempenho devido à centralização da política de comunicação;
- Tendência do aumento da complexidade do código com o tempo.

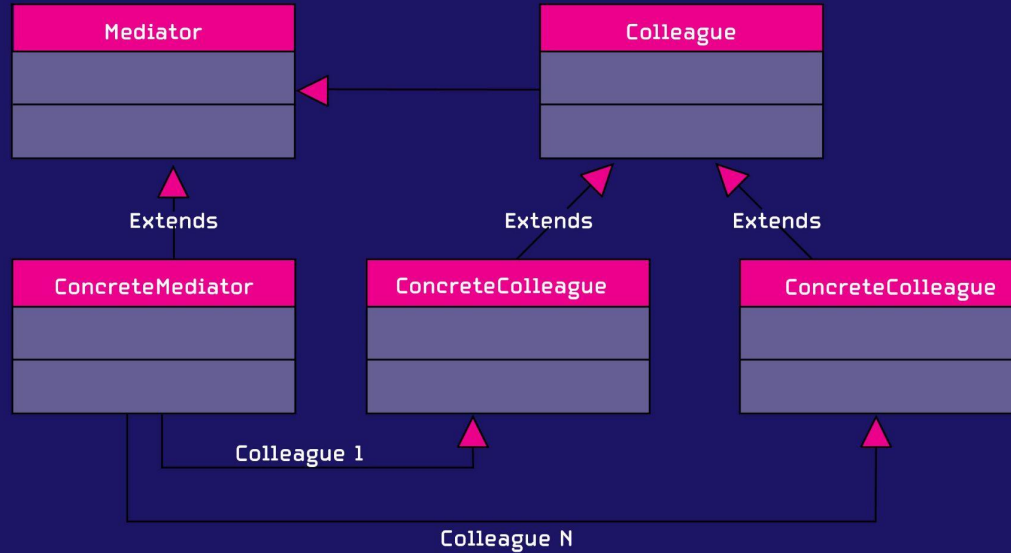
ESTRUTURA

Composta por apenas quatro elementos:

- Mediator;
- Concrete Mediator;
- Colleague;
- Concrete Colleague.

UML Representation:

We can represent the mediator pattern with the help of following UML diagram





Exemplo:


Chat de Mensagens



```
public interface ChatMediator {  
    public void enviarMensagem(String mensagem, Usuario user);  
  
    public void adicionarUsuario(Usuario user);  
}
```



```
public abstract class Usuario {  
    protected String nome;  
    protected ChatMediator mediator;  
  
    public Usuario(String nome, ChatMediator mediator) {  
        this.nome = nome;  
        this.mediator = mediator;  
    }  
  
    public abstract void enviarMensagem(String mensagem);  
  
    public abstract void receberMensagem(String mensagem);  
}
```



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

```
// Mediator Implementação
```

```
public class ChatMediatorImp implements ChatMediator {
    private List<Usuario> usuarios;

    public ChatMediatorImp() {
        this.usuarios = new ArrayList<Usuario>();
    }

    @Override
    public void enviarMensagem(String mensagem, Usuario user) {
        // Percorre a lista de usuários
        for(Usuario u : this.usuarios) {
            // Se o usuário for diferente do que está usando
            // a mensagem é enviada.
            if(u != user) {
                u.receberMensagem(mensagem);
            }
        }
    }

    @Override
    public void adicionarUsuario(Usuario user) {
        usuarios.add(user);
    }
}
```

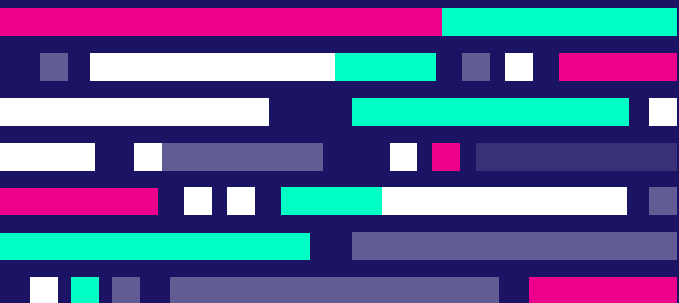


```
// Usuário Implementação
```

```
public class UsuarioImp extends Usuario {  
    public UsuarioImp(String nome, ChatMediator mediator) {  
        super(nome, mediator);  
    }  
  
    @Override  
    public void enviarMensagem(String mensagem) {  
        System.out.println(super.nome + " - Enviando mensagem: " +  
            mensagem);  
        super.mediator.enviarMensagem(mensagem, this);  
    }  
  
    @Override  
    public void receberMensagem(String mensagem) {  
        System.out.println(super.nome + " - Recebendo mensagem: " +  
            mensagem);  
    }  
}
```



```
public class Principal {  
    public static void main(String[] args) {  
        ChatMediator chat = new ChatMediatorImp();  
  
        Usuario u1 = new UsuarioImp("Usuario 1", chat);  
        Usuario u2 = new UsuarioImp("Usuario 2", chat);  
        Usuario u3 = new UsuarioImp("Usuario 3", chat);  
        Usuario u4 = new UsuarioImp("Usuario 4", chat);  
  
        chat.adicionarUsuario(u1);  
        chat.adicionarUsuario(u2);  
        chat.adicionarUsuario(u3);  
        chat.adicionarUsuario(u4);  
  
        u1.enviarMensagem("Boa noite");  
    }  
}
```



```
Console x Problems Debug Shell Coverage
<terminated> Principal (14) [Java Application] C:\APPS\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64.jre\bin\java.exe
Usuario 1 - Enviando mensagem: Boa noite
Usuario 2 - Recebendo mensagem: Boa noite
Usuario 3 - Recebendo mensagem: Boa noite
Usuario 4 - Recebendo mensagem: Boa noite
```



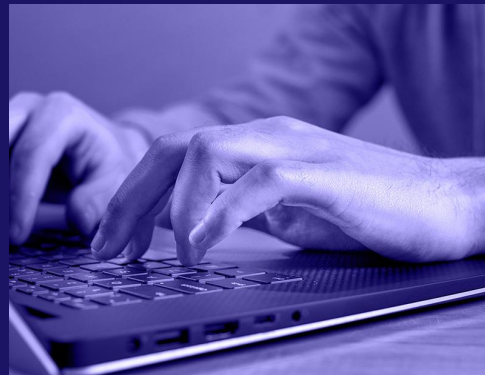

02

Memento Design Pattern

CONCEITO

O Memento é um Design Pattern que consiste em criar e utilizar estados de objetos antigos para recobrar um ponto anterior.

Portanto, ao utilizar um Memento é possível desfazer ações de forma segura e confiável no ponto de vista de boas práticas, sem quebrar o encapsulamento da aplicação.



PROPÓSITO

A intenção deste padrão de software é salvar o estado atual de um objeto em um Memento, um Originador é responsável pela captura e por se comunicar.

De tal forma quando um mecanismo pede para reverter alguma ação, o Originador entra em contato com o Memento e toma o estado anterior daquele objeto.

PROPÓSITO

Alguns objetivos do Memento são:

1. O salvamento do instantâneo de (partes) do estado de um objeto, para que possa ser devolvido em algum dado momento.
2. Quando há uma interface ligada diretamente para a obtenção do estado, expondo detalhes de implementação.

VANTAGENS

- A possibilidade de reaver estados anteriores de um objeto;
- A criação de barreiras capazes de evitar a quebra do encapsulamento;
- Definição de duas interfaces, mínima e ampla.
- O controle dos estados dos objetos.

DESVANTAGENS

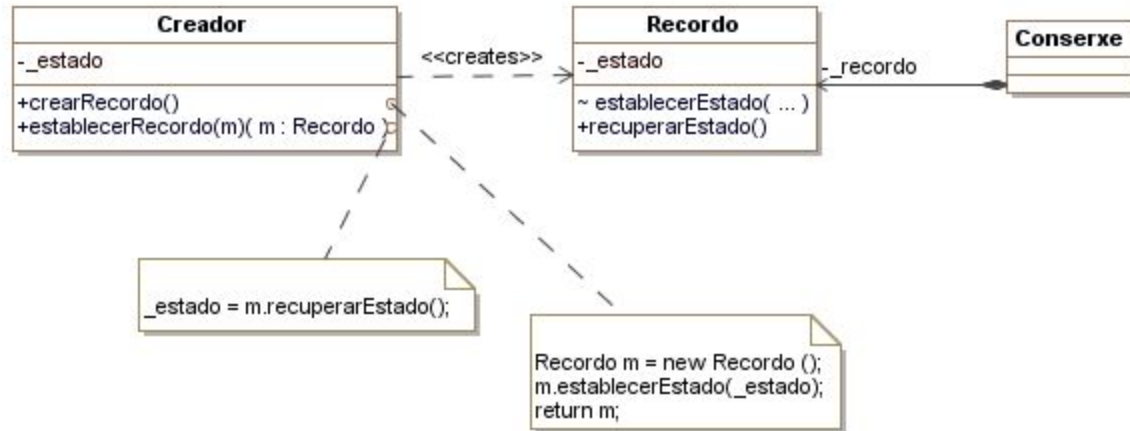
- O salvamento excessivo de Mementos no CareTaker pode ocasionar lentidão;
- Computacionalmente caro, caso a quantidade de informações a serem salva seja alto;
- Grande custo de armazenamento caso o CareTaker seja leve.

ESTRUTURA

Composta por três elementos:

- Memento;
- Originator;
- CareTaker

Memento UML Representation:





Exemplo:

Memento de textos



```
public class TextoMemento {  
    protected String estadoTexto;  
  
    public TextoMemento(String texto) {  
        estadoTexto = texto;  
    }  
  
    public String getTextoSalvo() {  
        return estadoTexto;  
    }  
}
```



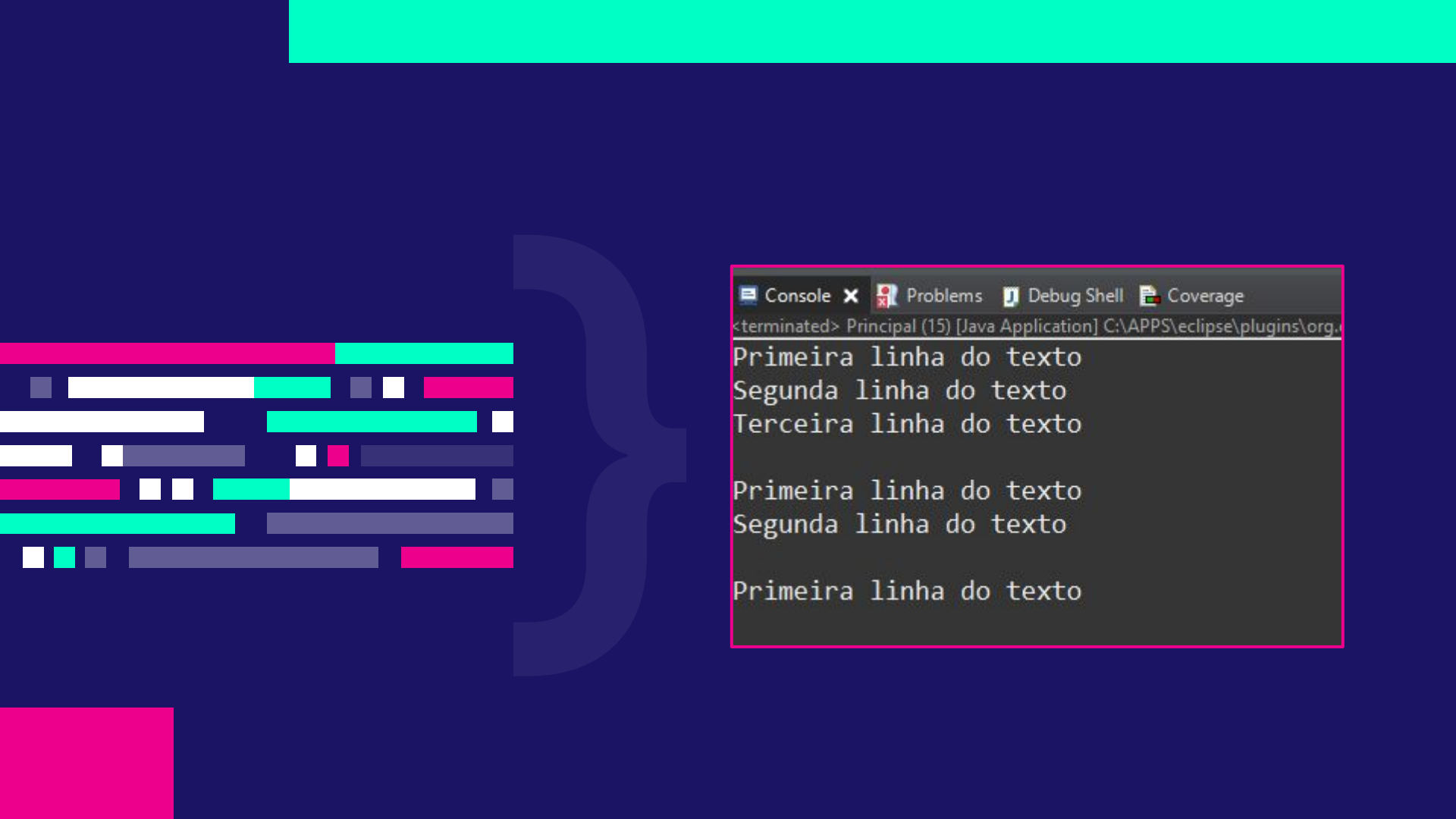
```
public class TextoCareTaker {  
    protected ArrayList<TextoMemento> estados;  
  
    public TextoCareTaker() {  
        estados = new ArrayList<TextoMemento>();  
    }  
  
    public void adicionarMemento(TextoMemento memento){  
        estados.add(memento);  
    }  
  
    public TextoMemento getUltimoEstadoSalvo() {  
        if (estados.size() <= 0) {  
            return new TextoMemento("");  
        }  
        TextoMemento estadoSalvo = estados.get(estados.size() - 1);  
        estados.remove(estados.size() - 1);  
        return estadoSalvo;  
    }  
}
```



```
public class Texto {  
    protected String texto;  
    Textocaretaker caretaker;  
  
    public Texto() {  
        caretaker = new Textocaretaker();  
        texto = new String();  
    }  
  
    public void escreverTexto(String novoTexto) {  
        caretaker.adicionarMemento(new Textomemento(texto));  
        texto += novoTexto;  
    }  
  
    public void desfazerEscrita() {  
        texto = caretaker.getUltimoEstadoSalvo().getTextoSalvo();  
    }  
  
    public void mostrarTexto() {  
        System.out.println(texto);  
    }  
}
```



```
public static void main(String[] args) {  
    Texto texto = new Texto();  
  
    texto.escreverTexto("Primeira linha do texto\n");  
    texto.escreverTexto("Segunda linha do texto\n");  
    texto.escreverTexto("Terceira linha do texto\n");  
    texto.mostrarTexto();  
    texto.desfazerEscrita();  
    texto.mostrarTexto();  
    texto.desfazerEscrita();  
    texto.mostrarTexto();  
    texto.desfazerEscrita();  
    texto.mostrarTexto();  
    texto.desfazerEscrita();  
    texto.mostrarTexto();  
}
```



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

Primeira linha do texto
Segunda linha do texto

Primeira linha do texto
```

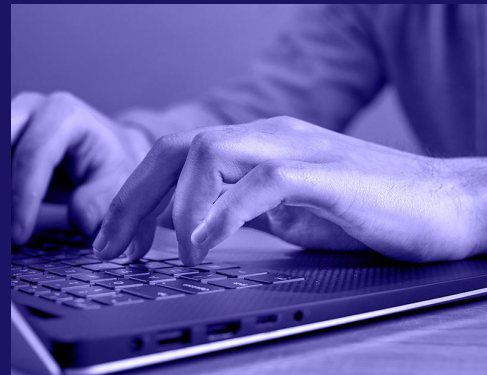


03

Observer Design Pattern

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.



PROPÓSITO

O objetivo principal e mais relevante de tal Pattern é definir um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

Além de desacoplar objetos emissores e objetos receptores pela definição de uma interface para sinalizar mudanças em subjects.

VANTAGENS

- Tanto observadores quanto sujeitos observados podem ser reutilizados e ter sua interface e implementação alteradas sem afetar o sistema;
- Relacionamentos passam a ser de um para muitos;
- Acoplamento forte é reduzido com o uso de interfaces e classes abstratas

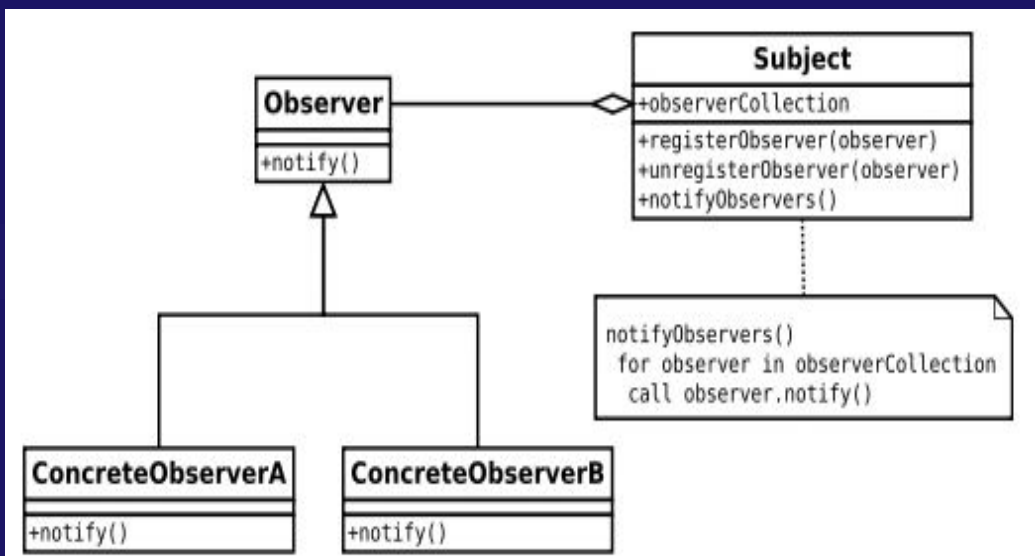
DESVANTAGENS

- O abuso pode causar sério impacto na performance. Sistema onde todos notificam todos a cada mudança ficam inundados de requisições.

ESTRUTURA

Composta pelos seguintes elementos:

- Sujeito (Subject);
- Observador (Observer);
- Sujeito Concreto (ConcreteSubject);
- Observador Concreto(ConcreteObserver).





Exemplo:

Criação de um editor de
mensagens



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



```
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);
        }
    }
}
```



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



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




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



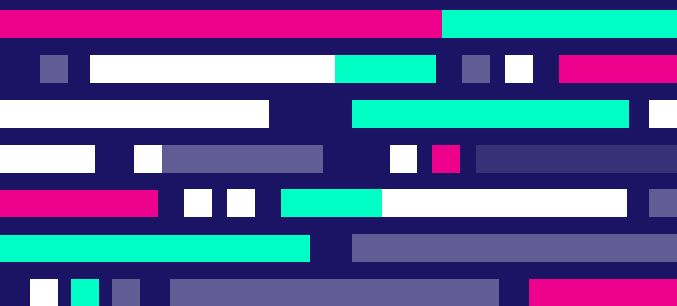
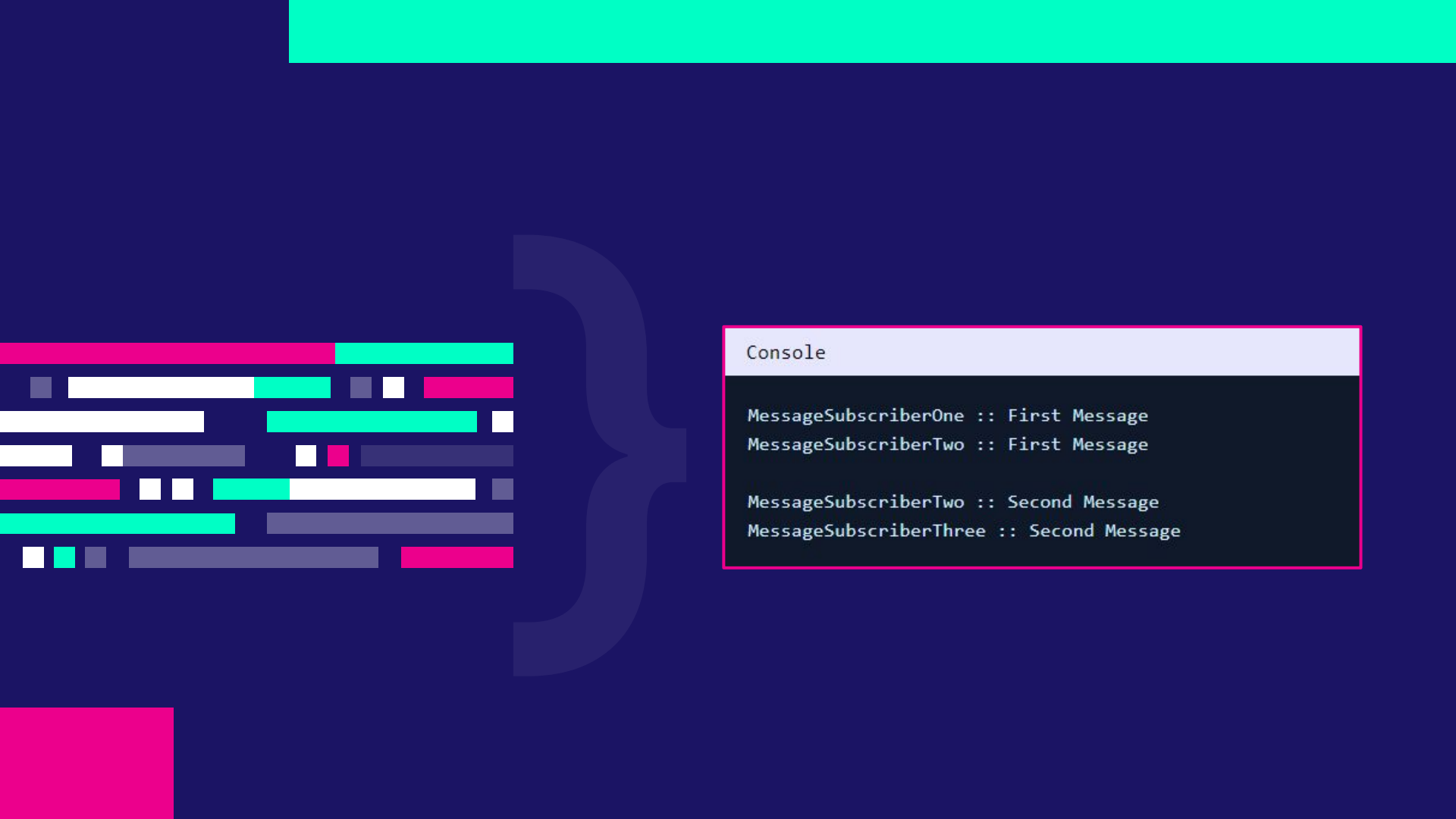
```
public class MessageSubscribeThree implements Observer {  
    @Override  
    public void update(Message m) {  
        System.out.println("MessageSubscribeThree :: "  
            + m.getMessageContent());  
    }  
}
```



```
public class Message {  
    final String messageContent;  
  
    public Message(String m) {  
        this.messageContent = m;  
    }  
  
    public String getMessageContent() {  
        return messageContent;  
    }  
}
```



```
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"));  
  
        p.detach(s1);  
        p.attach(s3);  
  
        p.notifyUpdate(new Message("Second Message"));  
    }  
}
```



Console

```
MessageSubscriberOne :: First Message  
MessageSubscriberTwo :: First Message  
  
MessageSubscriberTwo :: Second Message  
MessageSubscriberThree :: Second Message
```

REFERÊNCIAS

- Comportamental - Mediator - YouTube, CRISTIANO ALMEIDA "<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/>", 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>", acessado em maio de 2021.
- Mediator Teoria - Padrões de Projeto - Parte 35/45 - YouTube, OTÁVIO MIRANDA, "<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-proieto-mediator-na-pratica>", acessado em maio de 2021.
- GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

REFERÊNCIAS

- 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.
- Padrão de Projeto Observer (2012), Luiza Uira, "<https://pt.slideshare.net/luizauira/padro-de-projeto-observer>", acessado em maio de 2021.
- Observer (2018), Refactoring Guru, "<https://refactoring.guru/pt-br/design-patterns/observer#:~:text=Prop%C3%B3sito,objeto%20que%20eles%20est%C3%A3o%20observando.>", acessado em maio de 2021.
- Padrões em Java: Observer (2015), Diogo Souza, "<http://www.linhadecodigo.com.br/artigo/3643/padrees-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.

FIM