## Instituto Federal de São Paulo – Campus São Paulo

# Projeto de Engenharia de Software III

Projeto consiste em mostrar alguns padrões de projetos do GoF, dentre eles, padrão de criação, padrão estrutural e padrão comportamental. Além disso, trazer diagrama de classes, diagrama de comunicação, sequência desses padrões, um exemplo de código dos padrões e por fim, indicar classes ou framework que utilizam algum desses padrões.

## Integrantes


| | | |
|---|---|---|
| Daniel Araujo de Oliveira | – | SP3082831 |
| Gustavo Nascimento Falconi | – | SP3097854 |
| Matheus Savoia Resende | – | SP3097781 |
| Wesley Vieira dos Santos | – | SP3083896 |


## Padrões Escolhidos

Padrões de Criação - Singleton e Prototype

Padrões Estruturais - Adapter e Decorator

Padrões Comportamentais - Visitor e Memento

**IFSP**

**SÃO PAULO**

**2022**

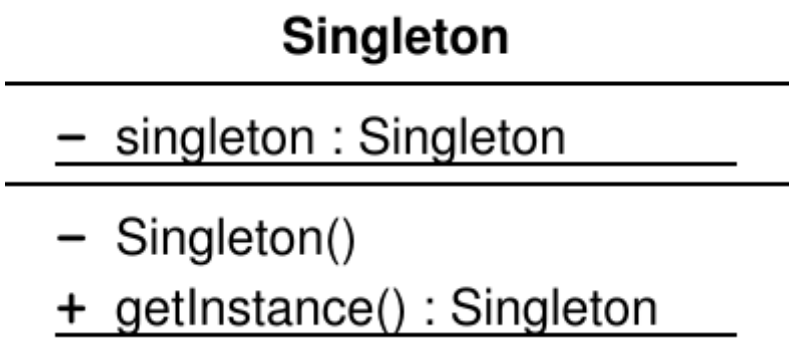
## Sumário

# 1. Padrões de Criação

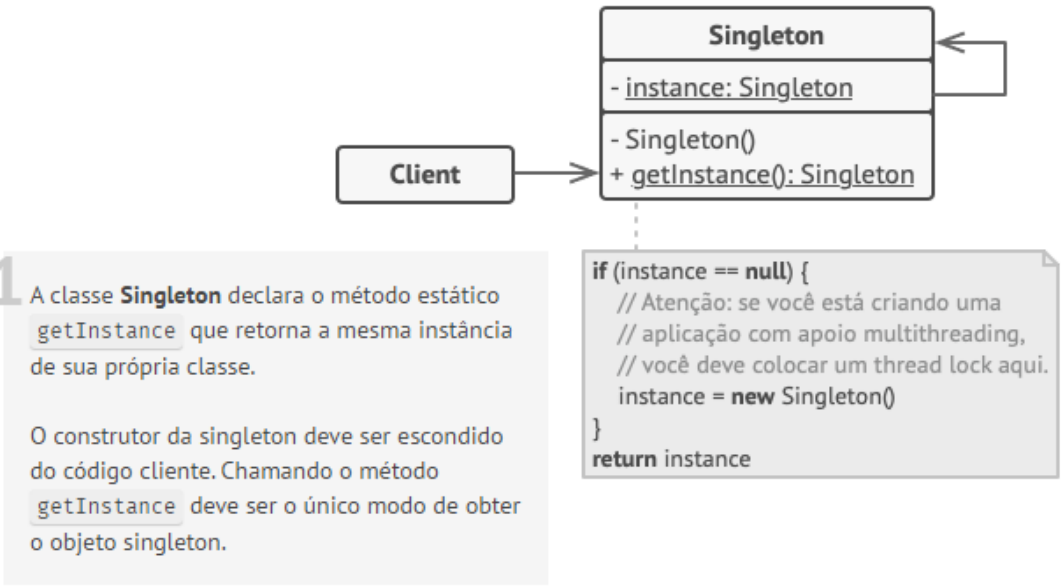
## 1.1 Padrão Singleton

O padrão Singleton permite criar objetos únicos para os quais há apenas uma instância. Este padrão oferece um ponto de acesso global, assim como uma variável global, porém sem as desvantagens das variáveis globais.

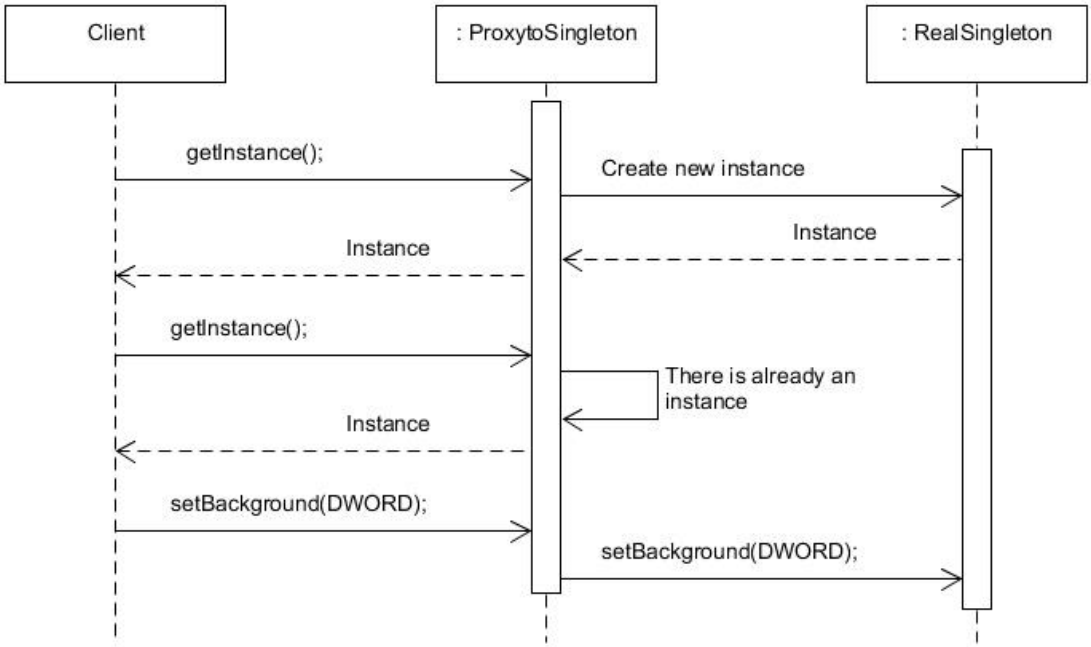
- **Diagrama de classes**




- **Diagrama de comunicação**



- **Diagrama de sequência**




- **Código do padrão**

```
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();

        return uniqueInstance;
    }
}
```

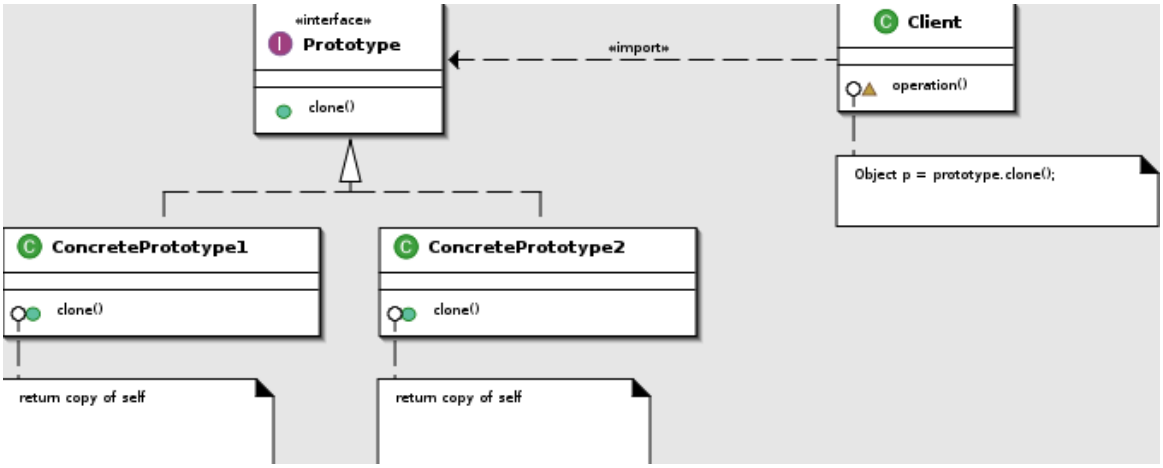
- **Classes / Framework que utilizam esse padrão**

Singleton normalmente é usado para logging, drivers objects, caching e criação de thread pool. Inclusive ele pode ser usado junto com outros padrões como o Abstract Factory, Builder, Prototype, Facade e assim vai. Inclusive ele é usado dentro do próprio core do Java em classes comojava.lang.Runtime, java.awt.Desktop.

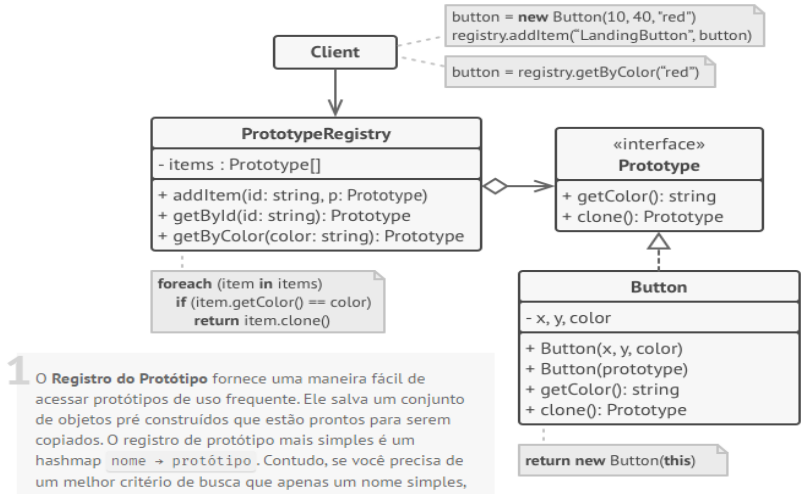
## 1.2 Padrão Prototype

O Prototype é um padrão de projeto criacional que permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes.

- **Diagrama de Classes**

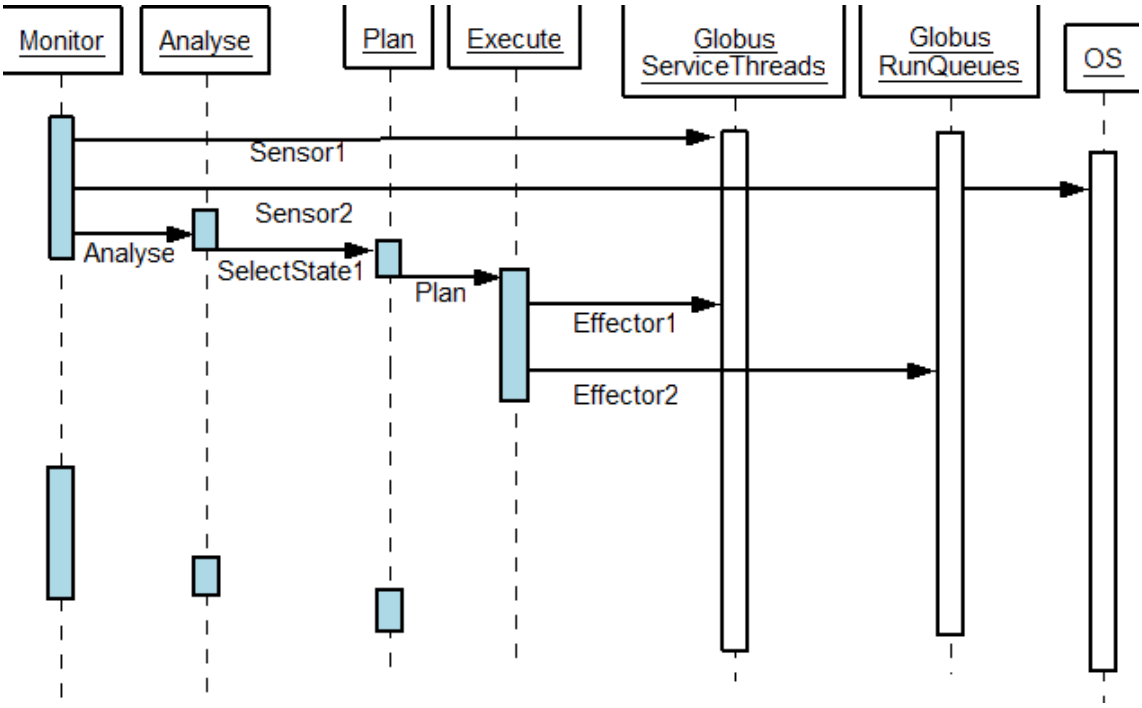



- **Diagrama de comunicação**




1 O **Registro do Protótipo** fornece uma maneira fácil de acessar protótipos de uso frequente. Ele salva um conjunto de objetos pré construídos que estão prontos para serem copiados. O registro de protótipo mais simples é um hashmap `nome → protótipo`. Contudo, se você precisa de um melhor critério de busca que apenas um nome simples, você pode construir uma versão muito mais robusta do registro.

- **Diagrama de sequência**




- **Classes / Framework que utilizam esse padrão**

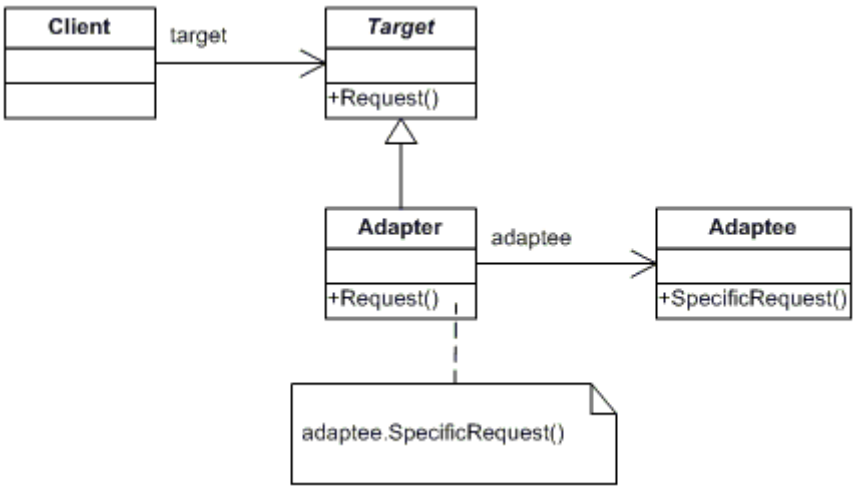
O padrão Prototype está disponível e pronto para uso em Java com a interface *Cloneable*. Qualquer classe pode implementar essa interface para se tornar clonável. *java.lang.Object#clone(),* a classe deve implementar a interface *java.lang.Cloneable*.

# 2. Padrões Estruturais

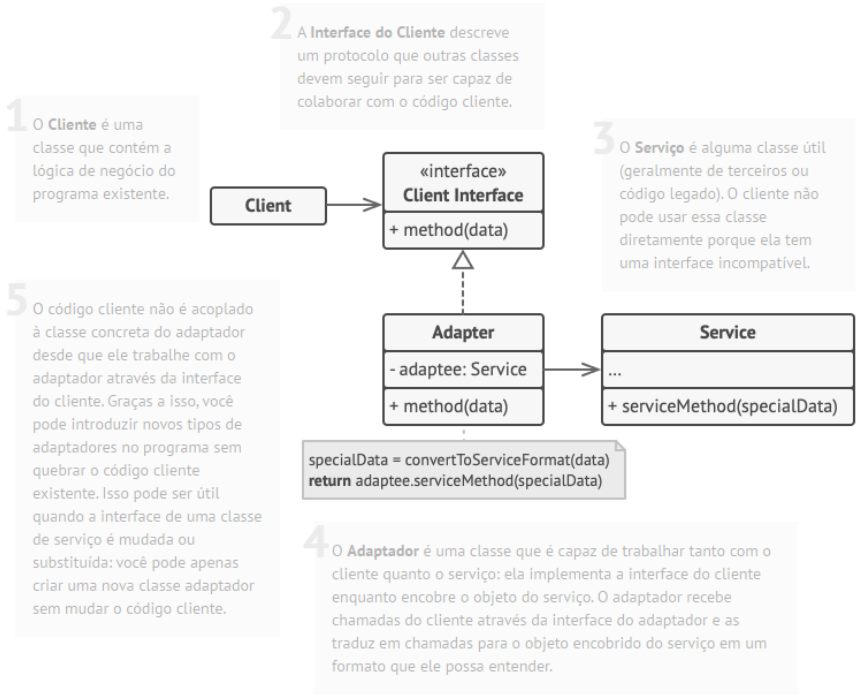
## 2.1 Padrão Adapter

O padrão Adapter é um padrão de projeto que permite que diferentes objetos com interfaces incompatíveis colaborem entre si. No mundo real, esse padrão seria como adaptadores de tomada, que permite que tomadas diferentes se conecte no mesmo lugar.

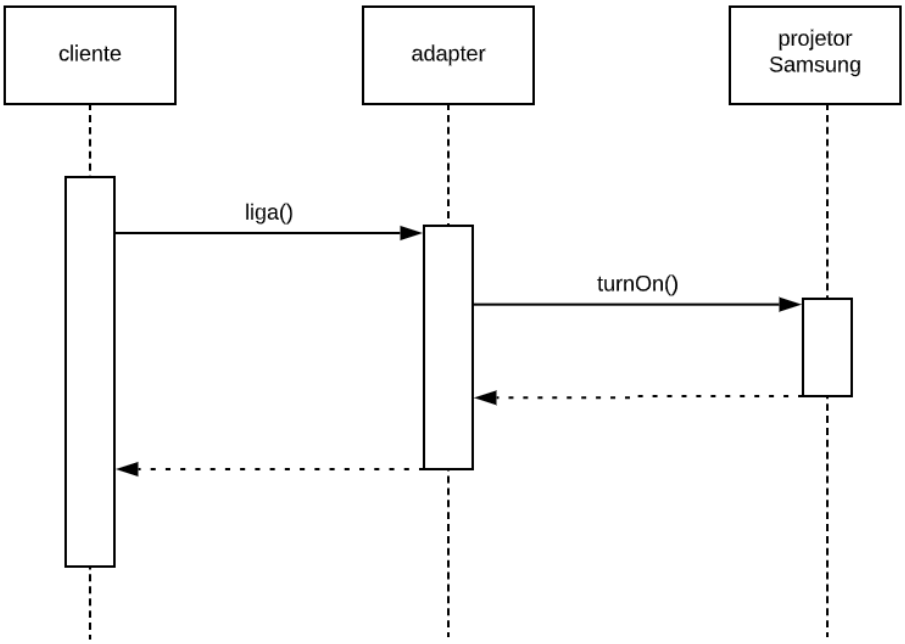
- **Diagrama de classes**




- **Diagrama de comunicação**



- **Diagrama de sequência**




- **Código do padrão**

```java
public class TomadaDeDoisPinos {
    public void ligarNaTomadaDeDoisPinos() {
        System.out.println("Ligado na Tomada de Dois Pinos");
    }
}

public class TomadaDeTresPinos {
    public void ligarNaTomadaDeTresPinos() {
        System.out.println("Ligado na Tomada de Tres Pinos");
    }
}

public class AdapterTomada extends TomadaDeDoisPinos {
    private TomadaDeTresPinos tomadaDeTresPinos;

    public AdapterTomada(TomadaDeTresPinos tomadaDeTresPinos) {
        this.tomadaDeTresPinos = tomadaDeTresPinos;
    }

    public void ligarNaTomadaDeDoisPinos() {
        tomadaDeTresPinos.ligarNaTomadaDeTresPinos();
    }
}
```

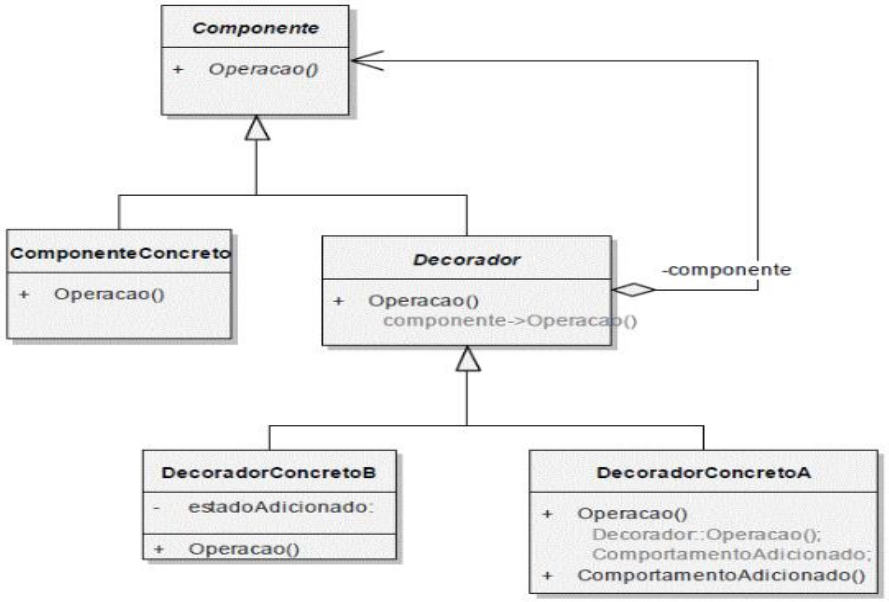
- **Classes / Framework que utilizam esse padrão**

As classes que contém adaptadores em Java podem ser encontradas nos pacotes java.awt.event, java.awt.dnd e javax.swing.event. Em classes como a classe WindowAdapter, do java.awt.event e a classe MouseInputAdapter, do pacote javax.swing.event.

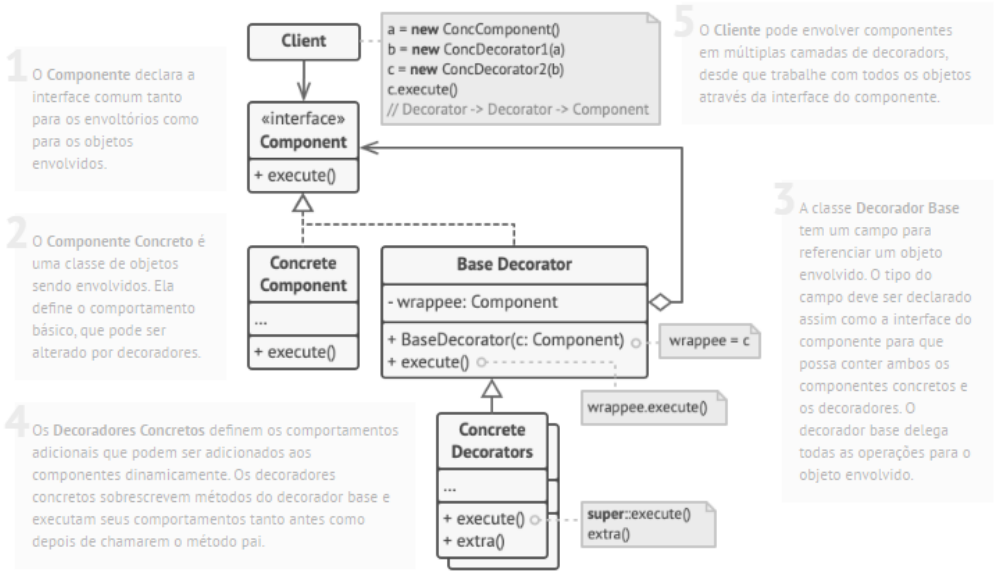
### 2.2 Padrão Decorator

O padrão de projeto Decorator permite acoplar responsabilidades adicionais a um objeto dinamicamente. É uma alternativa no uso de subclasses para extensão de funcionalidades. Utiliza a agregação ou composição ao invés da herança.

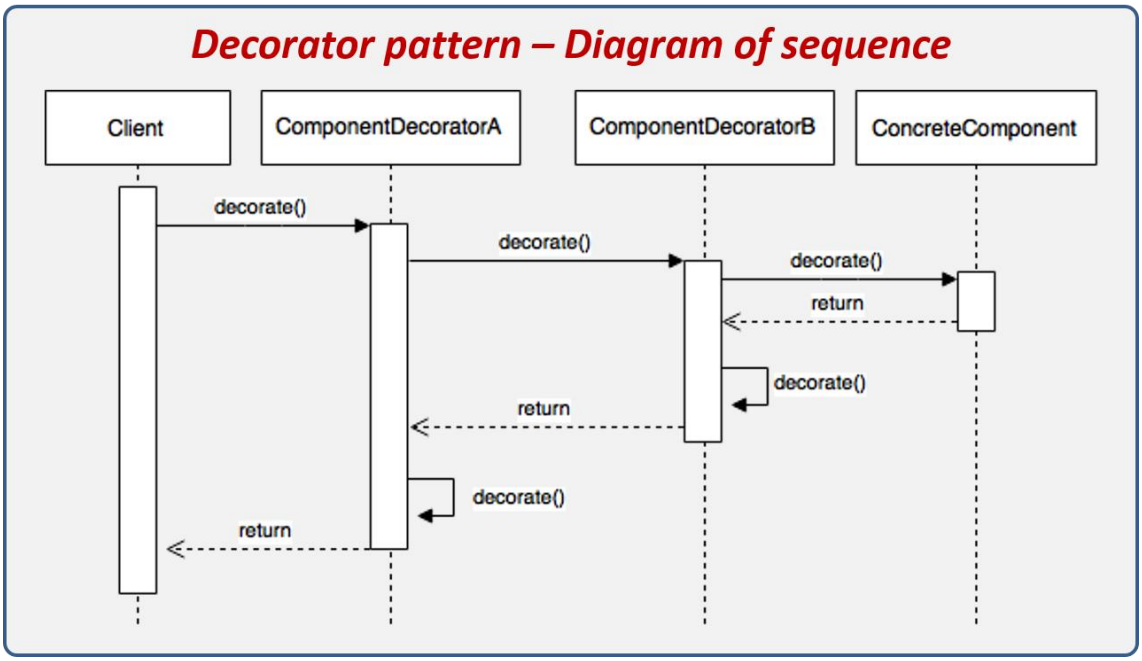
- **Diagrama de classes**



- **Diagrama de comunicação**




- **Diagrama de sequência**



- **Código padrão**

```
abstract class Janela {

    public abstract void draw();

}

class JanelaSimples extends Janela {

    public void draw() {
        System.out.println(“desenha uma janela”);
    }
}

abstract class JanelaDecorator extends Janela {

    protected Janela janelaDecorada;

    public JanelaDecorator(Janela janelaDecorada) {
        this.janelaDecorada = janelaDecorada;
    }

}
```

```
class DecoradorBarraVertical extends JanelaDecorator {

    public DecoradorBarraVertical(Janela janelaDecorada) {
        super(janelaDecorada);
    }

    public void draw() {
        drawBarraVertical();
        janelaDecorada.draw();
    }

    private void drawBarraVertical() {
        System.out.println(“desenha uma barra vertical na janela”);
    }

}
```

- **Classes / Framework que utilizam esse padrão**

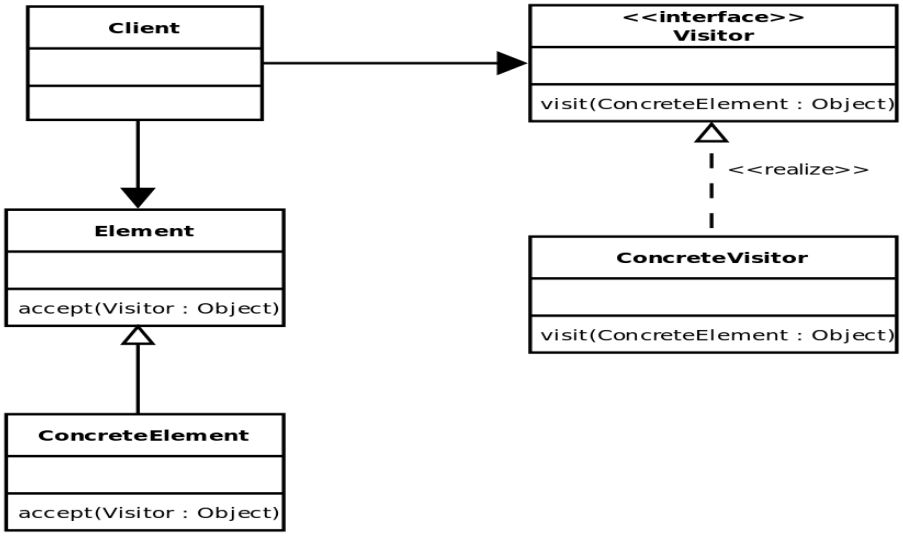
Em o Java a utilização do padrão Decorator é recorrente, em especial nos códigos relacionados a fluxos. Uma classe em Java que utiliza bastante o padrão de projeto Decorator é a java.IO, exemplo de objetos que a utilizam: LineNumberInputStream, BufferedInputStream e FileInputStream.

# 3. Padrões Comportamentais

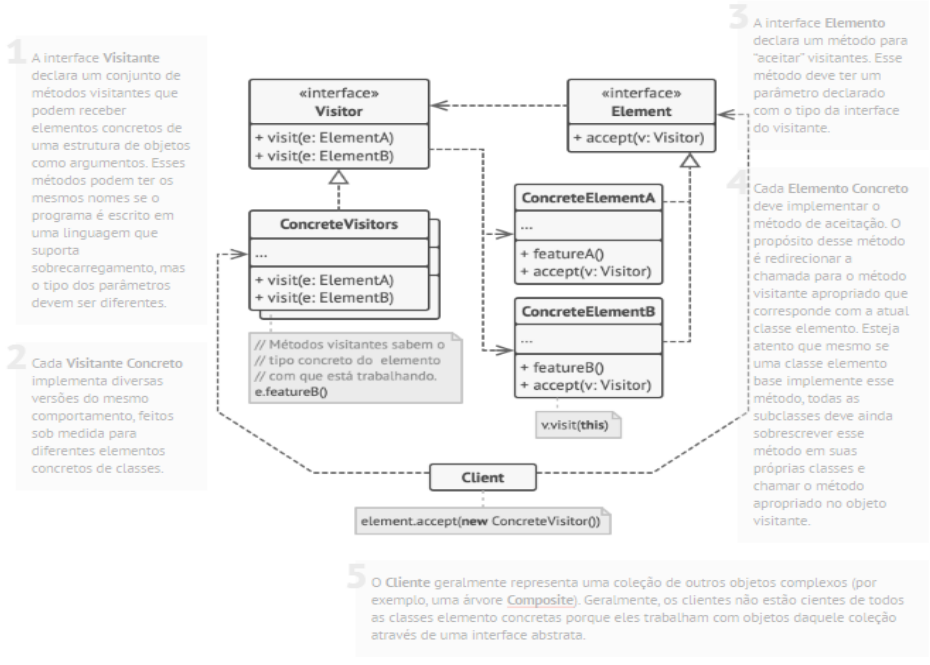
## 3.1 Padrão Visitor

O Visitor é um padrão de projeto comportamental que permite separar algoritmos dos objetos nos quais eles operam, colocando novos comportamentos em classes separadas chamadas visitantes, ao invés de tentar integrar estes comportamentos em classes já existentes.

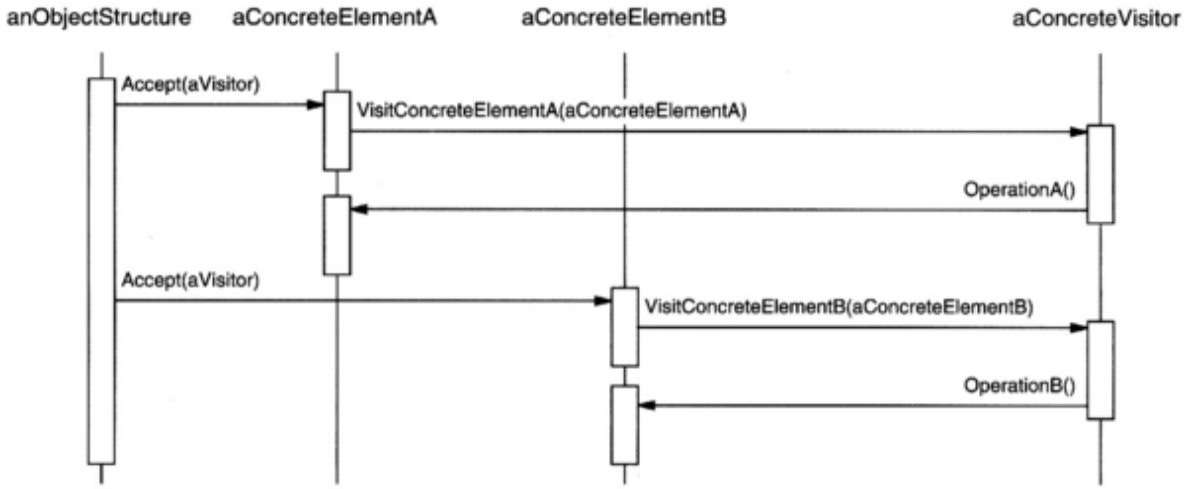
- **Diagrama de classes**




- **Diagrama de comunicação**



- **Diagrama de sequência**




- **Código padrão**

```
interface ItemElement
{
    public int accept(ShoppingCartVisitor visitor);
}

class Book implements ItemElement
{
    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice()
    {
        return price;
    }

    public String getIsbnNumber()
    {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}
```

```
class Fruit implements ItemElement
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}
```

```
interface ShoppingCartVisitor
{

    int visit(Book book);
    int visit(Fruit fruit);
}

class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{

    @Override
    public int visit(Book book)
    {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50)
        {
            cost = book.getPrice()-5;
        }
        else
            cost = book.getPrice();

        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit)
    {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }

}
```

```
class ShoppingCartClient
{

    public static void main(String[] args)
    {
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"),
                              new Book(100, "5678"), new Fruit(10, 2, "Banana"),
                              new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items)
    {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
        {
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

- **Classes / Framework que utilizam esse padrão**

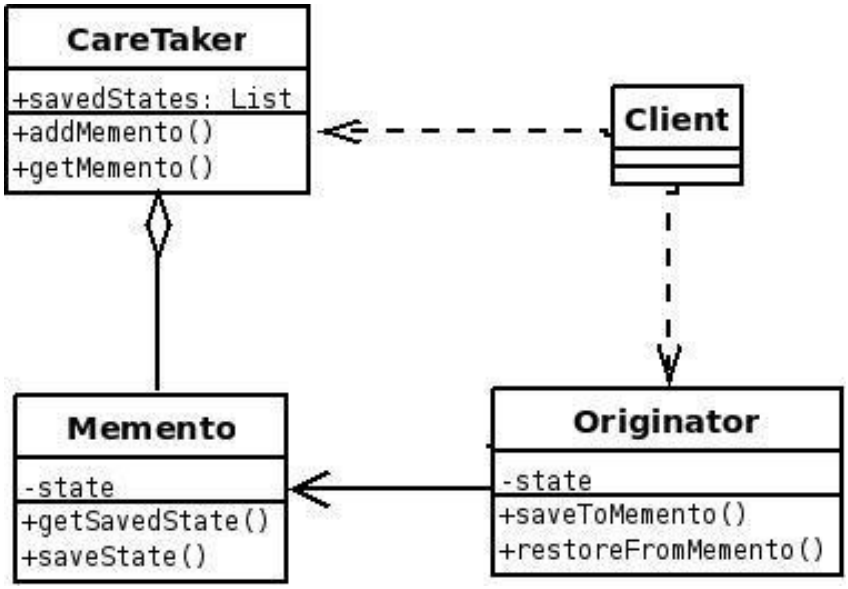
O Visitor não é um padrão muito comum devido à sua complexidade e aplicabilidade limitada. Alguns exemplos do padrão nas principais bibliotecas Java:

- javax.lang.model.element.AnnotationValue e AnnotationValueVisitor;
- javax.lang.model.element.Element e ElementVisitor;
- javax.lang.model.type.TypeMirror e TypeVisitor;
- java.nio.file.FileVisitor e SimpleFileVisitor;
- javax.faces.component.visit.VisitContext e VisitCallback;

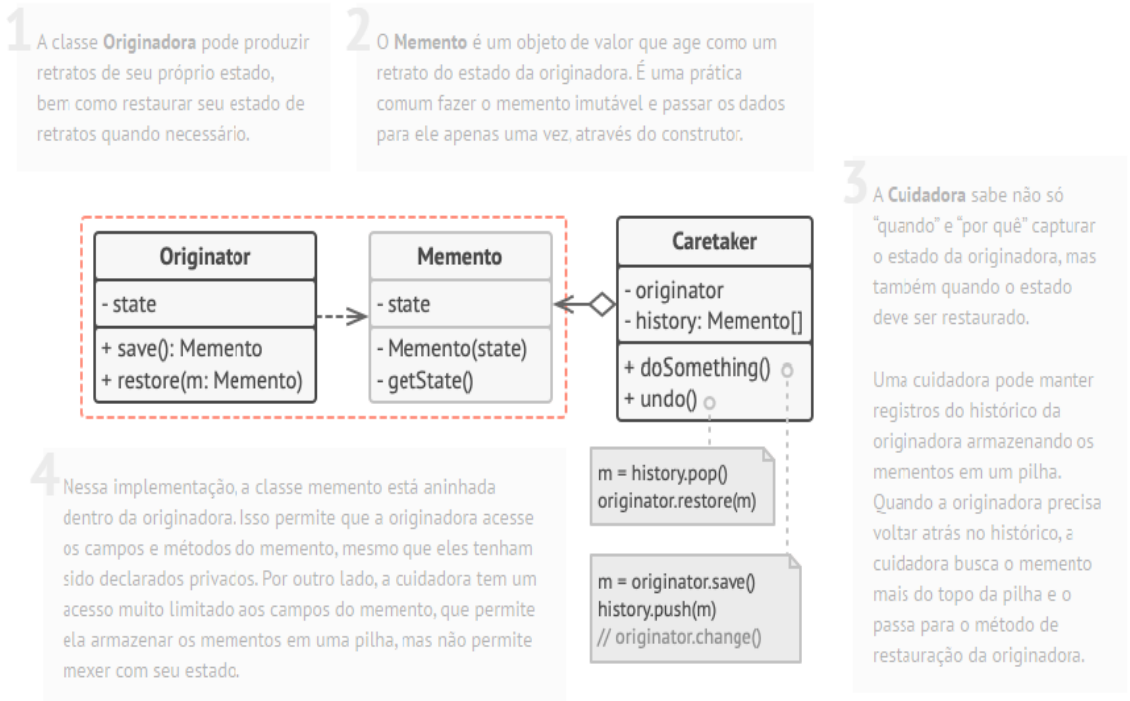
### 3.2 Padrão Memento

O Memento é um padrão de projeto comportamental que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação. Antes de realizar qualquer operação, a aplicação grava o estado de todos os objetos e salva eles em algum armazenamento. Mais tarde, quando um usuário decide reverter a ação, a aplicação busca o último retrato do histórico e usa ele para restaurar o estado de todos os objetos.

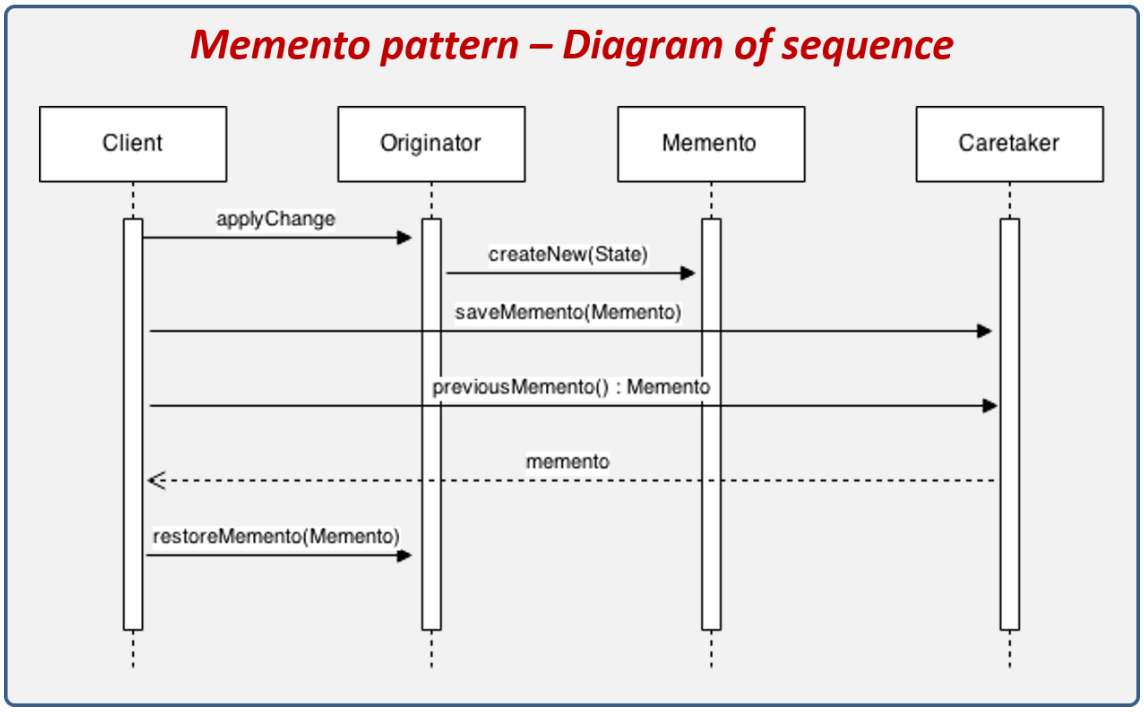
- **Diagrama de classes**




- **Diagrama de comunicação**



- **Diagrama de sequência**




- **Código padrão**

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

class Life
{
    private String time;

    public void set(String time)
    {
        System.out.println("Setting time to " + time);
        this.time = time;
    }

    public Memento saveToMemento()
    {
        System.out.println("Saving time to Memento");
        return new Memento(time);
    }

    public void restoreFromMemento(Memento memento)
    {
        time = memento.getSavedTime();
        System.out.println("Time restored from Memento: " + time);
    }

    public static class Memento
    {
        private final String time;

        public Memento(String timeToSave)
        {
            time = timeToSave;
        }

        public String getSavedTime()
        {
            return time;
        }
    }
}
```

```
class Design
{

    public static void main(String[] args)
    {

        List<Life.Memento> savedTimes = new ArrayList<Life.Memento>();

        Life life = new Life();

        //time travel and record the eras
        life.set("1000 B.C.");
        savedTimes.add(life.saveToMemento());
        life.set("1000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("2000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("4000 A.D.");

        life.restoreFromMemento(savedTimes.get(0));

    }
}
```

- **Classes / Framework que utilizam esse padrão**

Alguns exemplos do padrão nas principais bibliotecas Java:

- All java.io.Serializable implementações podem simular o Memento.
- Todas as implementações de javax.faces.component.StateHolder.

# 4. Referências Bibliográficas

## 4.1 Referências


- https://www.devmedia.com.br/
- https://refactoring.guru/
- https://deviniciative.wordpress.com/