

# **Análise e Projeto de software**

**Renan Thiago da Silva Rosa**

**201304940032**

Faculdade de Computação

Universidade Federal do Pará

[renannojosa@gmail.com](mailto:renannojosa@gmail.com)

## **1. Introdução**

Atividade Avaliativa do tema Padrões de Projeto

Responder as questões abaixo da seguinte forma:

a) um relatório feito em editor de texto contendo, para cada questão:

- modelos UML com a descrição da solução
- a explicação dos principais trechos de código-fonte
- evidência de execução (com duas telas pelo menos)

b) código fonte de todas as soluções (como arquivos texto)

## **2. Solução da 1ª questão**

### **State Pattern**

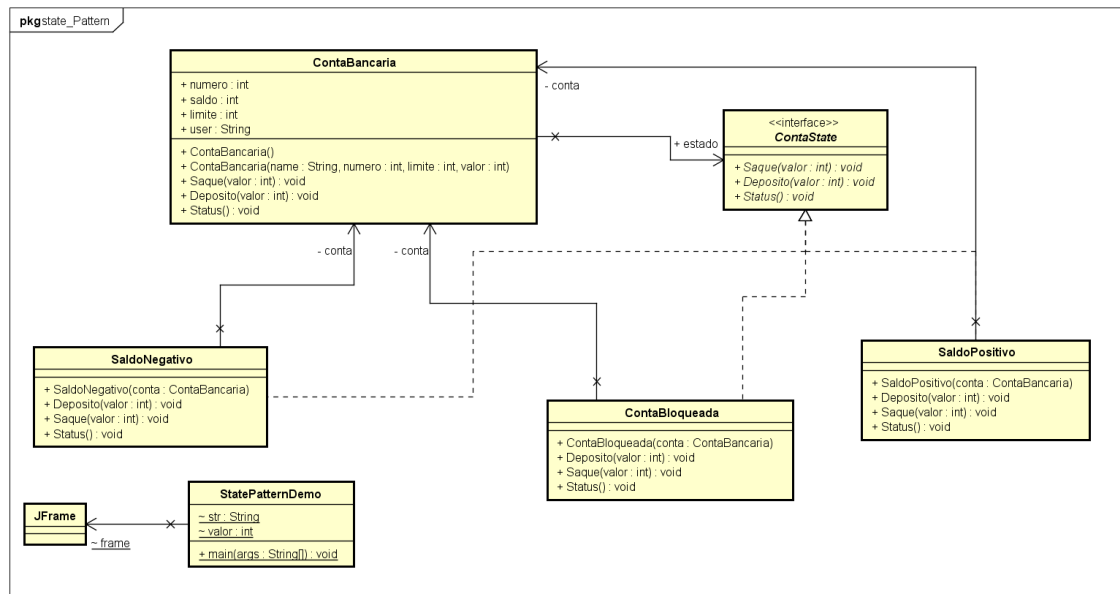
No padrão State, um comportamento de classe muda com base em seu estado. Este tipo de padrão de projeto vem sob o padrão de comportamento.

No padrão State, criamos objetos que representam vários estados e um objeto de contexto cujo comportamento varia conforme seu objeto de estado muda.

### **Implementação**

Vamos criar uma interface de estado definindo uma ação e classes de estado concretas implementando a interface de estado. Contexto é uma classe que transporta um Estado.

StatePatternDemo, nossa classe de demonstração, usará `ContaBancaria` e objetos de estado para demonstrar a mudança no comportamento de Contexto com base no tipo de estado em que se encontra.



**Figura 2.0 Diagrama UML State Pattern**

```
public interface ContaState {
    void Saque(int valor);
    void Deposito(int valor);
    void Status();
}
```

A interface definida acima possui métodos para saque, deposito e também um método que retorna o atual estado do objeto.

Criamos classes concretas implementando a mesma interface.

```
public class SaldoPositivo implements ContaState {
    private ContaBancaria conta;

    public SaldoPositivo(ContaBancaria conta)
    {
        this.conta = conta;
    }
}
```

Todas as classes que implementam `ContaState` recebem um objeto do tipo `ContaBancaria`. Este objeto terá seus atributos manipulados pelos métodos da classe `SaldoPositivo`.

Assim, o objeto pode mudar de estado caso algum método verifique que é necessário:

```
public void Deposito(int valor)
{
    this.conta.saldo+=valor;
    showAlertDialog(null,"Foi depositado R$ "+ valor);
    if ( this.conta.saldo < 0 )
    {
        if ( this.conta.saldo < -1.0*this.conta.limite)
        {
            this.conta.estado = new ContaBloqueada(this.conta);
        }
        else
        {

```

```

        this.conta.estado = new SaldoPositivo(this.conta);
    }
}

```

O código abaixo mostra partes da classe que terá seu estado mudado:

```

public class ContaBancaria {
    public int limite;
    public ContaState estado;
    public String user;

    public void Saque(int valor)
    {
        this.estado.Saque(valor);
        this.estado.Status();
    }
}

```

Observe que o método Saque passa um valor para que outro objeto realize as operações. Na classe StatePatternDemo podemos ver mudanças no comportamento quando o Estado muda.

```

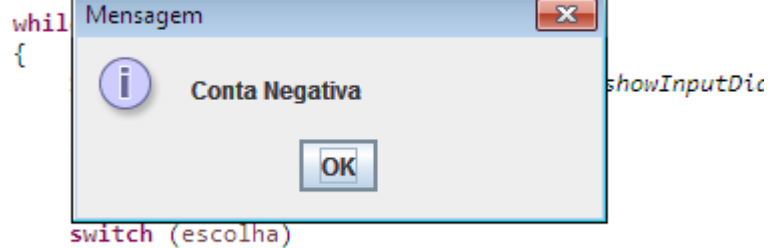
ContaBancaria bradesco = new ContaBancaria(user,7592,1

while
{
    Bem Vindo!
    Escolha uma opção !!!!
    Saque
    Saque
    Deposito
    Status
    Sair
    case
    str = JOptionPane.showInputDialog(null, "[
    valor = Integer.parseInt( str.trim() );
    bradesco.Saque(valor);
    String[] opcoes = { Saque, Deposito, Status,
    String user = JOptionPane.showInputDialog(null, "Di
    ContaBancaria bradesco = new ContaBancaria(user,759

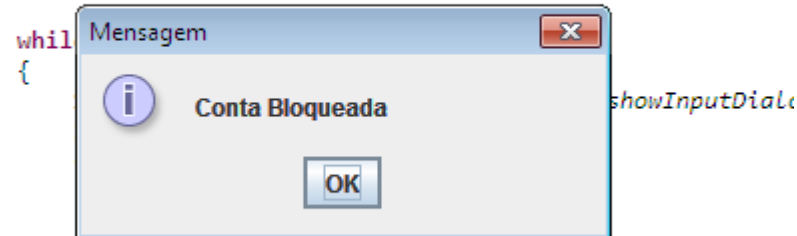
while
{
    Mensagem
    Foi sacado R$ 100
    OK
    switch (escolha)
    {
        case "Saque":
            str = JOptionPane.showInputDialog(null,

```

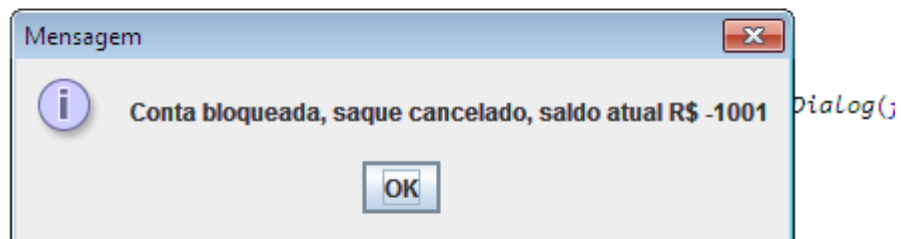
```
String user = JOptionPane.showInputDialog(null, "Digite seu nome");
ContaBancaria bradesco = new ContaBancaria(user, 7592, 1000);
```



```
switch (escolha) {
    case "Saque":
        str = JOptionPane.showInputDialog(null, "Digite o valor a sacar");
        bradesco.sacar(str);
        JOptionPane.showMessageDialog(null, "Saque realizado com sucesso!");
        break;
    case "Deposito":
        str = JOptionPane.showInputDialog(null, "Digite o valor a depositar");
        bradesco.depositar(str);
        JOptionPane.showMessageDialog(null, "Deposito realizado com sucesso!");
        break;
    case "Status":
        JOptionPane.showMessageDialog(null, "Saldo atual R$ " + bradesco.getSaldo());
        break;
    case "Sair":
        System.exit(0);
        break;
}
```



```
switch (escolha) {
    case "Saque":
        str = JOptionPane.showInputDialog(null, "Digite o valor a sacar");
        bradesco.sacar(str);
        JOptionPane.showMessageDialog(null, "Saque realizado com sucesso!");
        break;
    case "Deposito":
        str = JOptionPane.showInputDialog(null, "Digite o valor a depositar");
        bradesco.depositar(str);
        JOptionPane.showMessageDialog(null, "Deposito realizado com sucesso!");
        break;
    case "Status":
        JOptionPane.showMessageDialog(null, "Saldo atual R$ " + bradesco.getSaldo());
        break;
    case "Sair":
        System.exit(0);
        break;
}
```



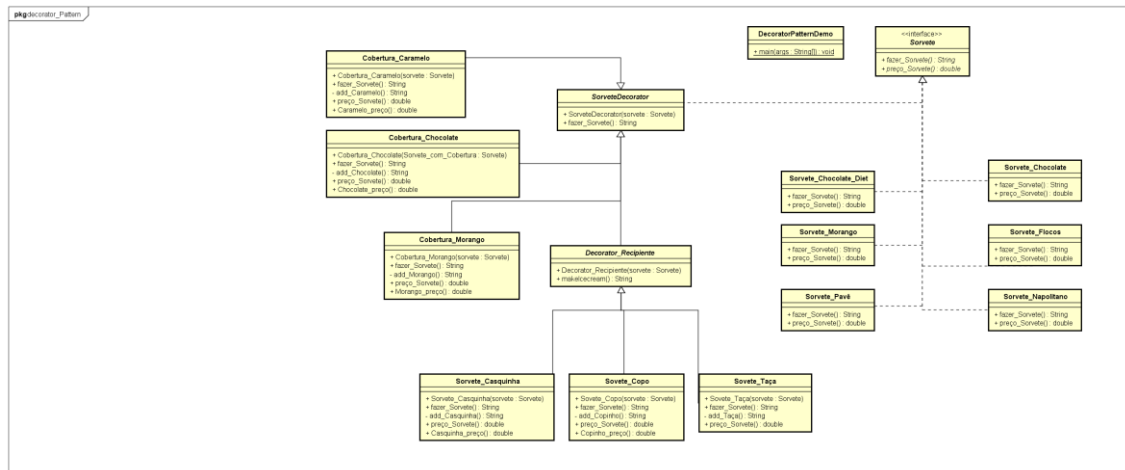
```
switch (escolha) {
    case "Saque":
        str = JOptionPane.showInputDialog(null, "Digite o valor a sacar");
        bradesco.sacar(str);
        JOptionPane.showMessageDialog(null, "Saque realizado com sucesso!");
        break;
    case "Deposito":
        str = JOptionPane.showInputDialog(null, "Digite o valor a depositar");
        bradesco.depositar(str);
        JOptionPane.showMessageDialog(null, "Deposito realizado com sucesso!");
        break;
    case "Status":
        JOptionPane.showMessageDialog(null, "Saldo atual R$ " + bradesco.getSaldo());
        break;
    case "Sair":
        System.exit(0);
        break;
}
```

### 3. Solução da 2ª questão

#### Implementação

O padrão Decorator permite que um usuário adicione a funcionalidade nova a um objeto existente sem alterar sua estrutura. Este tipo de padrão de projeto vem sob padrão estrutural como este padrão age como um invólucro para a classe existente.

Esse padrão cria uma classe de decorador que envolve a classe original e fornece funcionalidade adicional mantendo a assinatura de métodos de classe intacta.



Vamos criar uma interface `Sorvete` e classes concretas implementando a interface `Sorvete`. Em seguida, criaremos uma classe de decorador abstrato `SorveteDecorator` implementando a interface `Sorvete` e tendo o objeto `Sorvete` sorvete como sua variável de instância.

```

public interface Sorvete {

    public String fazer_Sorvete();
    public double preçoSorvete();

}

public abstract class SorveteDecorator implements Sorvete {

    protected Sorvete sorvete;

}

```

As classes abaixo implementam a interface acima e serão decoradas por outros objetos que estenderem a classe abstrata acima.

```

public class Sorvete_Chocolate implements Sorvete {

    public String fazer_Sorvete() {return "Sorvete de Chocolate ";}

    public double preçoSorvete () {return 1.50;}

}

public class Sorvete_Flocos implements Sorvete {

    public String fazer_Sorvete() {return "Sorvete de Flocos ";}

    public double preçoSorvete () {return 1.50;}

}

```

A classe abaixo decora objetos do tipo sorvete, ou seja, acrescenta novas funcionalidade sem alterar a estrutura da classe

```
public class Cobertura_Caramelo extends SorveteDecorator {  
    public Cobertura_Caramelo(Sorvete sorvete)  
    {  
        super(sorvete);  
    }  
  
    public String fazer_Sorvete()  
    {  
        return sorvete.fazer_Sorvete() + add_Caramelo();  
    }  
  
    private String add_Caramelo(){  
        return "com cobertura de Caramelo";  
    }  
  
    public double preçoSorvete ()  
    {  
        return sorvete.preçoSorvete () + Caramelo_preço ();  
    }  
  
    public double Caramelo_preço () {  
        return 0.50;  
    }  
}
```

Agora usamos a classe SorveteDecorator e retornamos o resultado

```
Sorvete sorvete = new Sorvete_preço ();  
System.out.println(sorvete.fazer_Sorvete());  
  
Sorvete sorvete_com_Cobertura = new  
Cobertura_Chocolate(sorvete);
```

```
JOptionPane.showMessageDialog(null,sorvete.fazer_Sorvete());
```

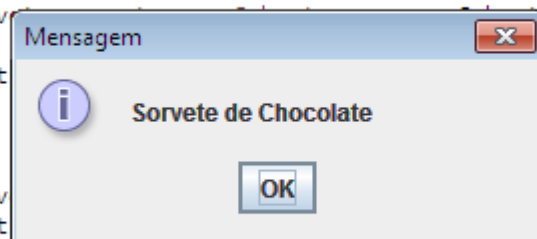
```
Sorvete sorvete_com_Cobertura = new Cobertura_Chocolate(sorvete);
```

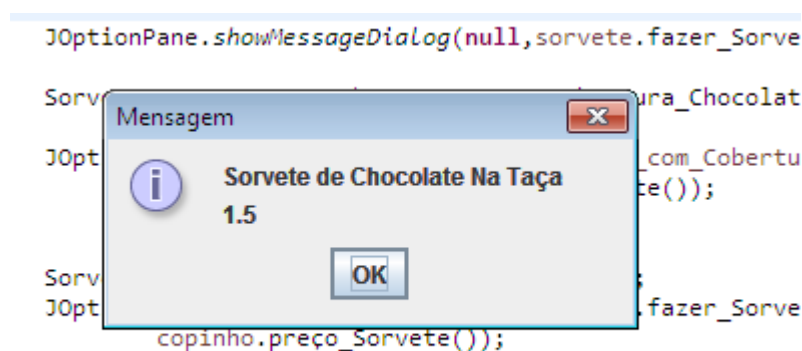
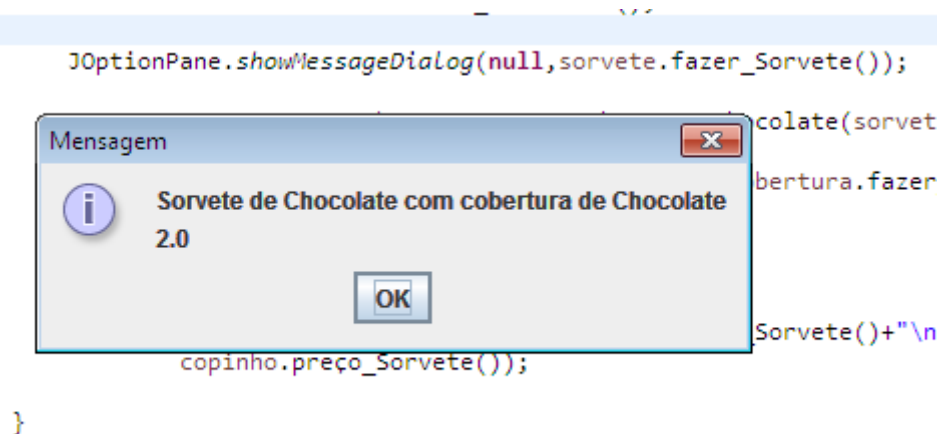
```
JOptionPane.showMessageDialog(null,sorvete_com_Cobertura.fazer_Sorvete());
```

```
Sorvete sorvete_com_Cobertura = new Cobertura_Chocolate(sorvete);
```

```
JOptionPane.showMessageDialog(null,sorvete_com_Cobertura.fazer_Sorvete());
```

```
}
```





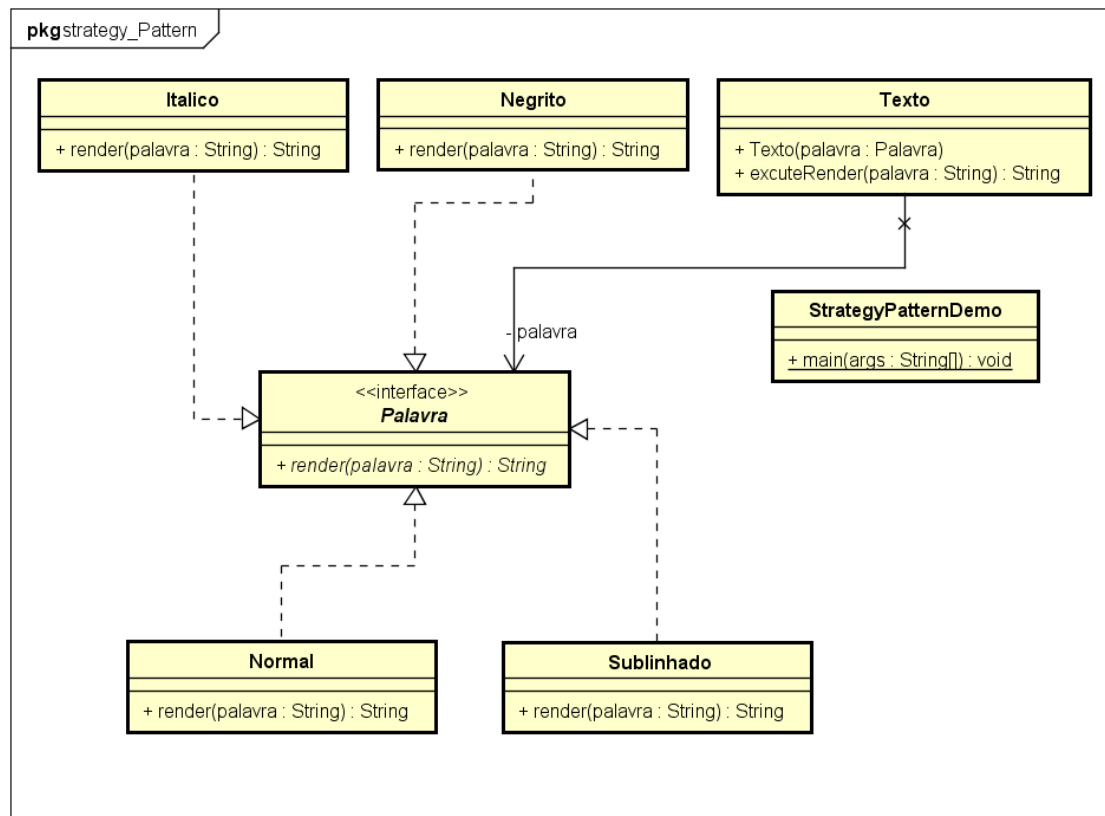
#### 4. Solução da 3ª questão

##### Strategy Pattern

No padrão Strategy, um comportamento de classe ou seu algoritmo pode ser alterado em tempo de execução. Este tipo de padrão de projeto vem sob o padrão de comportamento.

No padrão Strategy, criamos objetos que representam várias estratégias e um objeto de contexto cujo comportamento varia de acordo com seu objeto de estratégia. O objeto de estratégia altera o algoritmo de execução do objeto de contexto.

##### Implementação:



**Figura 4.0 Diagrama UML Strategy Pattern**

Vamos criar uma interface Palavra definindo uma ação e classes de Palavra concretas implementando a interface da Palavra. Texto é uma classe que usa uma Estratégia.

```

public interface Palavra {
    public String render(String palavra);
}
  
```

O método acima define uma estratégia que será usada e sobrescrita em todas as classes.

```

public class Normal implements Palavra {
    public String render(String palavra)
    {
        return "<span>"+palavra+"</span>";
    }
}
  
```

As classe concretas que implementam a interface, recebem uma palavra e a retorna em um formato apropriado com está abaixo.

```

public class Normal implements Palavra {
  
```



```

    public String render(String palavra)
    {
        return "<span>"+palavra+"</span>";
    }
}

```

A classe `Texto` possui como atributo um objeto da classe `Palavra` que será usado para executar os métodos das classes concretas de `Palavra`. O código abaixo exemplifica este processo.

```

public class Texto {
    private Palavra palavra;

    public Texto(Palavra palavra)
    {
        this.palavra = palavra;
    }

    public String executeRender(String palavra)
    {
        return this.palavra.render(palavra);
    }
}

```

`StatePatternDemo`, nossa classe de demonstração, usará objetos de `Texto` e `Palavra` para demonstrar mudanças no comportamento do contexto com base na estratégia que ele implementa ou usa.

```

Texto texto;
String[] palavras = {"norma", "italico", "negrito", "Subl

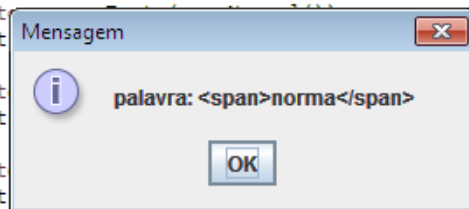
text
JOpt
a: " + textc

text
JOpt
a: " + textc

text
JOpt
a: " + textc

texto = new Texto(new Sublinhado());
JOptionPane.showMessageDialog(null, "palavra: " + textc

```



```

Texto texto;
String[] palavras = {"norma", "italico", "negrito", "Sul


text
JOpt
text
JOpt
text
JOpt

texto = new Texto(new Sublinhado());
JOOptionPane.showMessageDialog(null, "palavra: " + tex

texto = new Texto(new Sublinhado());
JOOptionPane.showMessageDialog(null, "palavra: " + tex

texto = new Texto(new Sublinhado());
JOOptionPane.showMessageDialog(null, "palavra: " + tex

```



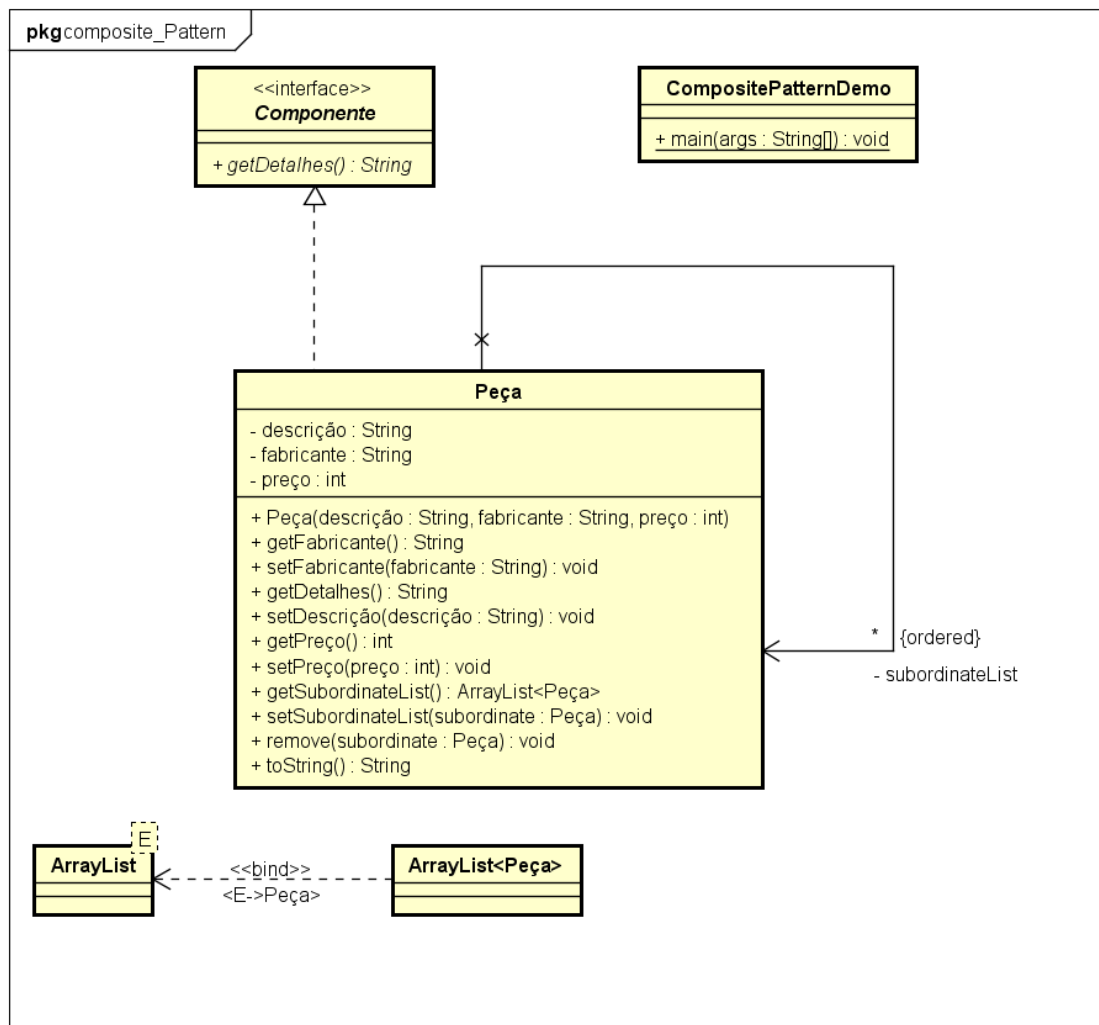
#### 4. Solução da 4ª questão

##### Composite Pattern

Padrão Composite é usado onde temos de tratar um grupo de objetos em forma semelhante como um único objeto. O padrão composto compõe objetos em termos de uma estrutura de árvore para representar parte assim como a hierarquia inteira. Este tipo de padrão de projeto vem sob o padrão estrutural como este padrão cria uma estrutura de árvore do grupo de objetos.

Esse padrão cria uma classe que contém o grupo de seus próprios objetos. Essa classe fornece maneiras de modificar seu grupo de mesmos objetos.

##### Implementação



**Figura 4.0 Diagrama UML Composite Pattern**

Definimos uma interface que possui apenas um método que descreve os detalhes de um componente caso o objeto Peça os tenha.

```

public interface Componente {
    public String getDetalhes();
}

```

Temos uma classe Peça que atua como classe de ator padrão composta. Composite Pattern Demo, ou classe demo usará classe Peça para adicionar hierarquia de nível de componente e imprimir todos os detalhes do mesmo.

```

public class Peça implements Componente {
    private String      descrição;
    private String      fabricante;
    private int         preço;
    private ArrayList< Peça > subordinateList = new ArrayList< Peça
>();

    public void setSubordinateList(Peça subordinate) {

```

```

        this.subordinateList.add(subordinate);
    }

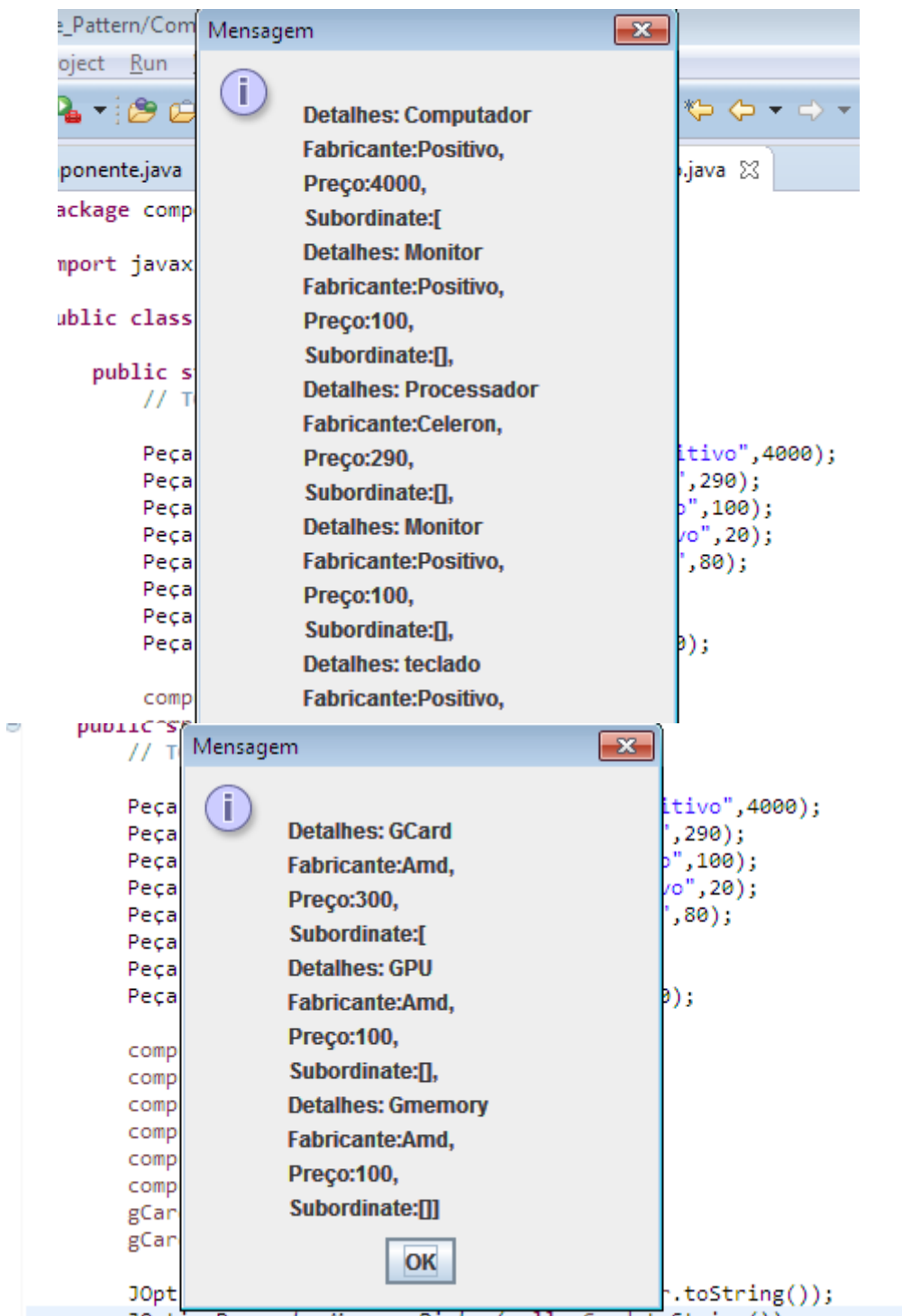
```

```

public void remove(Peça subordinate) {
    subordinateList.remove(subordinate);
}

```

Com o método `setSubordinateList()` podemos acrescentar outros objetos do tipo `Peça` para representar componentes subordinados na hierarquia.



## Visitor Pattern

No padrão Visitor, usamos uma classe visitante que altera o algoritmo de execução de uma classe de elemento. Desta forma, o algoritmo de execução do elemento pode variar conforme o visitante varia. Este padrão vem sob a categoria padrão de comportamento. Conforme o padrão, o objeto elemento tem que aceitar o objeto visitante de modo que o objeto visitante manipula a operação no objeto elemento.

## Implementação

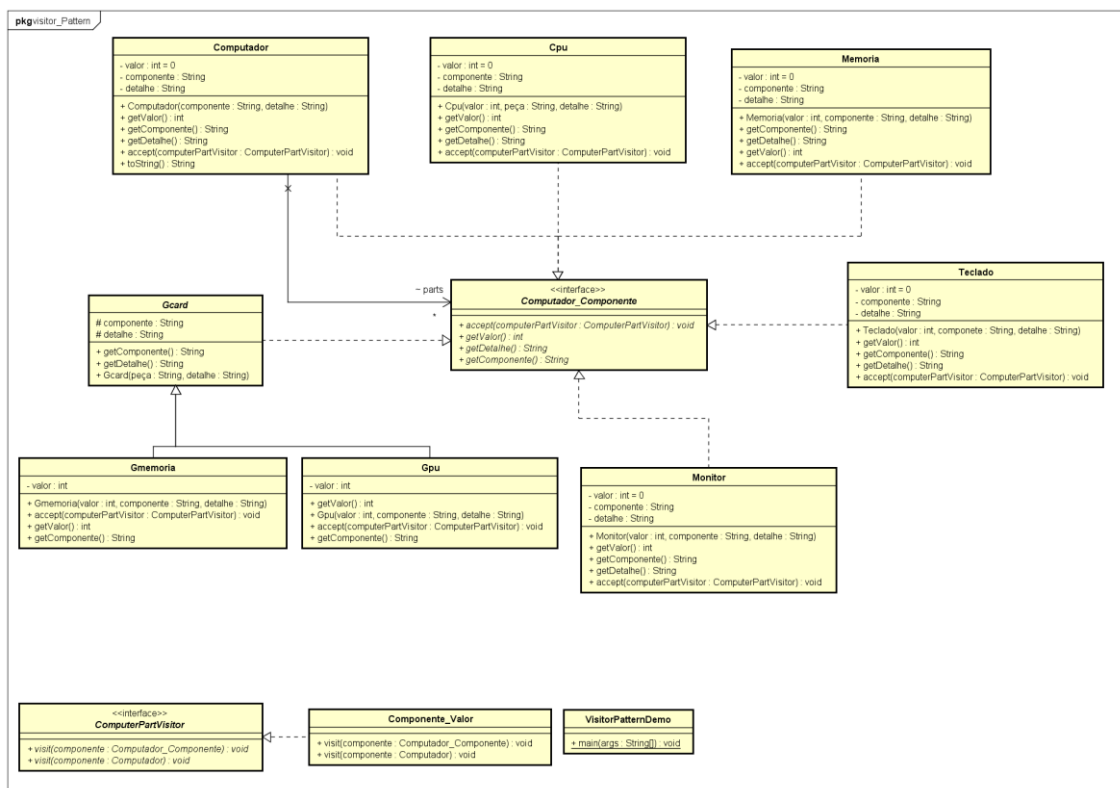


Figura 4.1 Diagrama UML Visitor Pattern

Vamos criar uma interface `Computador_Componente` definindo o método `accept`.

`Teclado`, `Memoria`, `Monitor`, `Gcard` e `Computador` são classes concretas implementando interface `Computador_Componente`. Vamos definir uma outra interface `ComputerPartVisitor` que irá definir uma classe visitante. `Computador` usa o método `visit` para fazer a ação correspondente.

A interface abaixo define um método que recebe um objeto que executará uma ação de acordo como o tipo do objeto concreto que implementa essa interface.

```
public interface Computador_Componente {

    public void accept(ComputerPartVisitor
computerPartVisitor);
```

Assim, um exemplo de objeto concreto é descrito abaixo.

```
public class Teclado implements Computador_Componente {

    private int valor = 0;
    private String componente;
    private String detalhe;

    public Teclado(int valor,String componete,String detalhe)
    {
        this.valor = valor;
        this.detalhe = detalhe;
        this.componente = componete;
    }

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor)
    {
        computerPartVisitor.visit(this);
    }
```

O método visit da classe ComputerPartVisitor verifica se o objeto passado como argumento é uma memória, se for verdade o método retorna o atributo valor.

```
public void visit(Computador_Componente componente)
{
    if ( componente instanceof Memoria || componente
instanceof Gmemoria )
        JOptionPane.showMessageDialog(null,"Memoria:
"+componente.getDetalhe()+
                                           "\nTipo:
"+componente.getComponente()+
                                           "\ncusta:
"+componente.getValor());
}
```

VisitorPatternDemo, nossa classe de demonstração, usará objetos de Computador\_Componente e Componente\_Valor para demonstrar uma resposta.

