INSTITUTO FEDERAL
Paraíba
Campus Campina Grande

Padrões de Projetos

Sistema de Vendas

Alunos:
José Domingos de Oliveira Neto
Lucas Cordeiro Vieira

Principais funcionalidades

- Realizar pedidos
- Gerar orçamentos(produto/ serviço)
- Calcular impostos
- Calcular descontos
- Realizar pagamentos
- Gerenciar produtos

Diagrama de classes UML (versão 1)

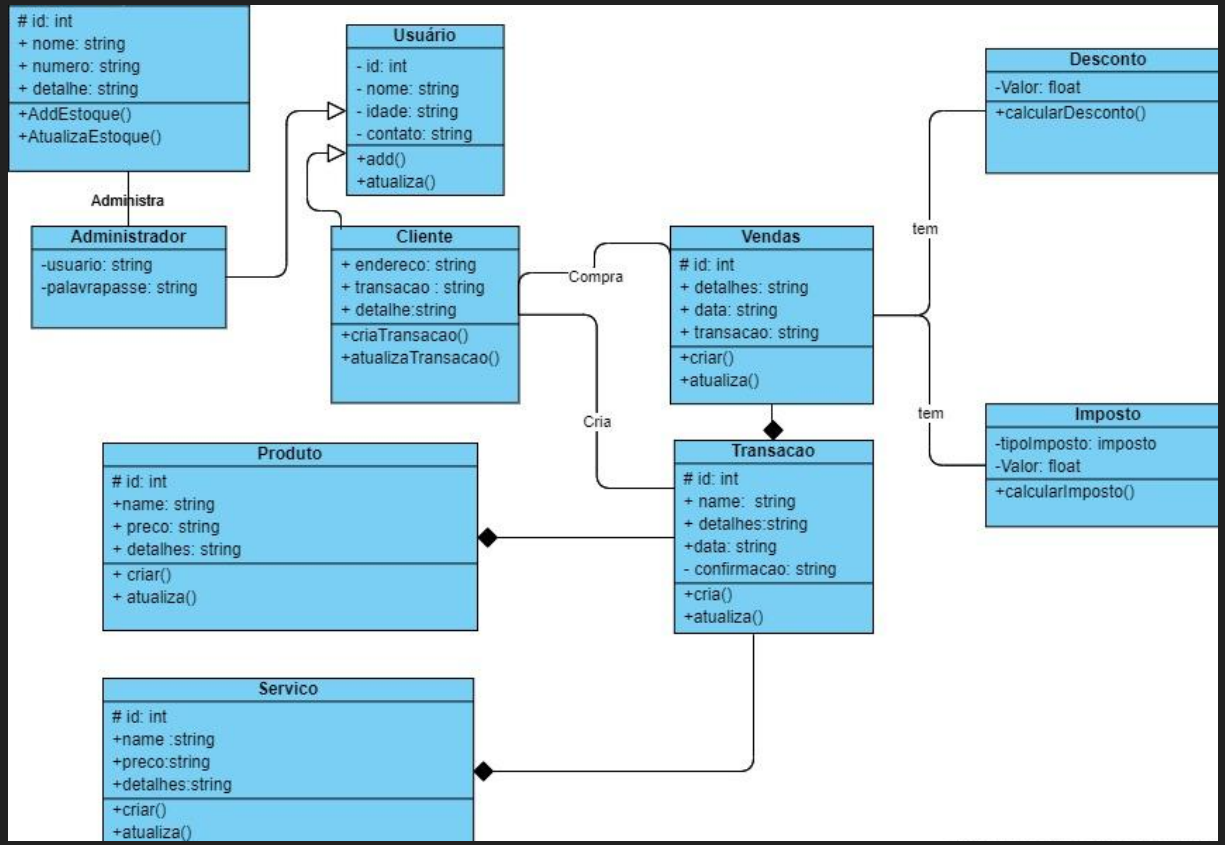
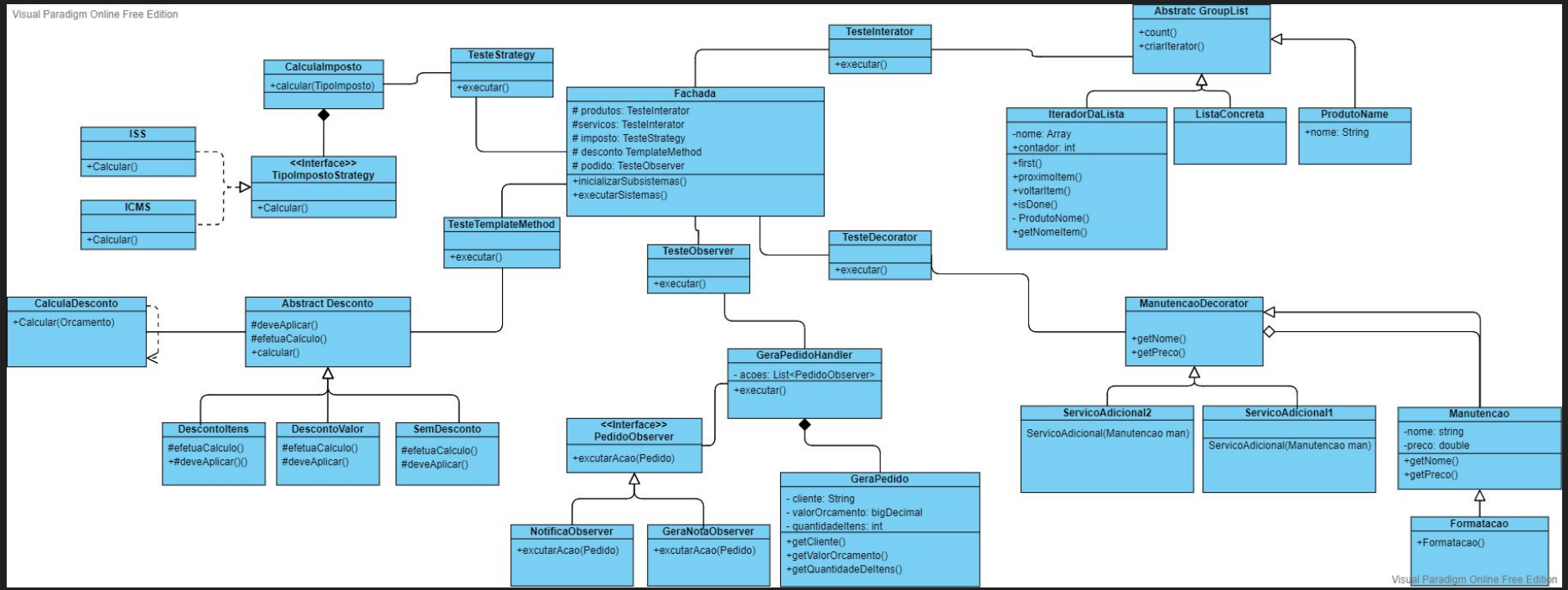
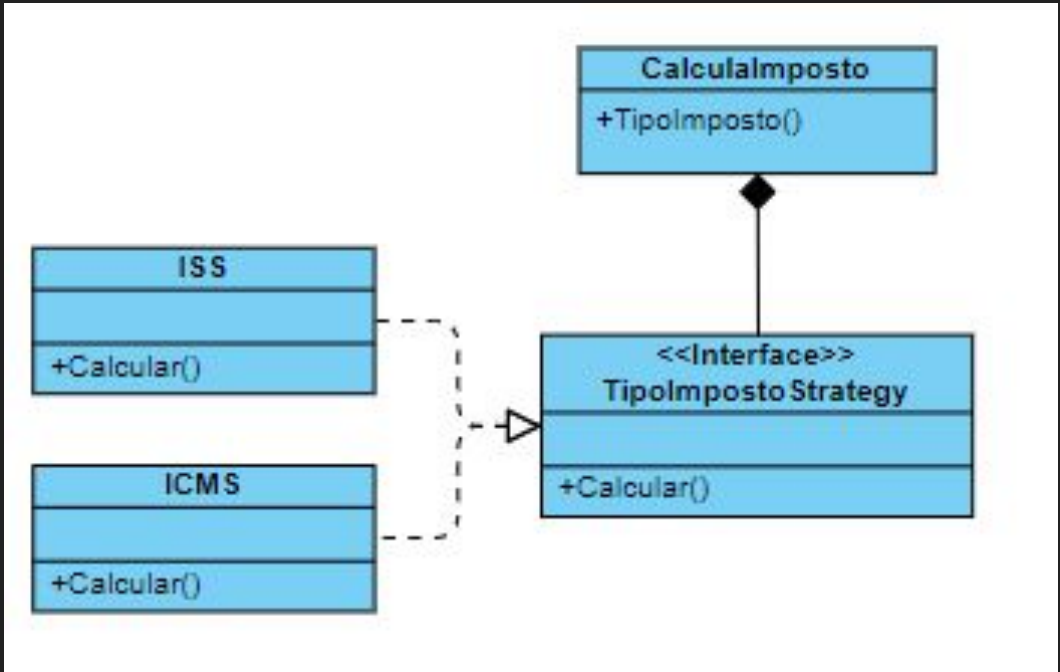


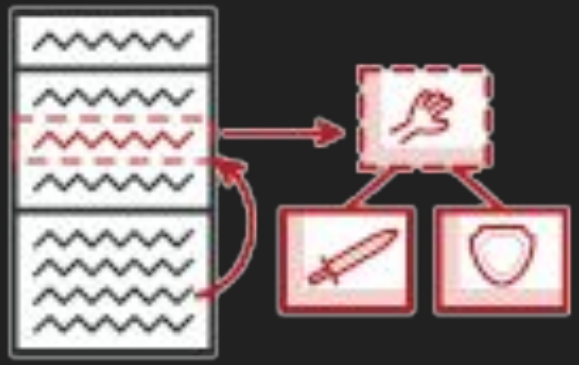
Diagrama de classes UML (Versão Final)



Strategy

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.





```java
public interface TipoImpostoStrategy {
    BigDecimal calcular(Orcamento orcamento);
}
```

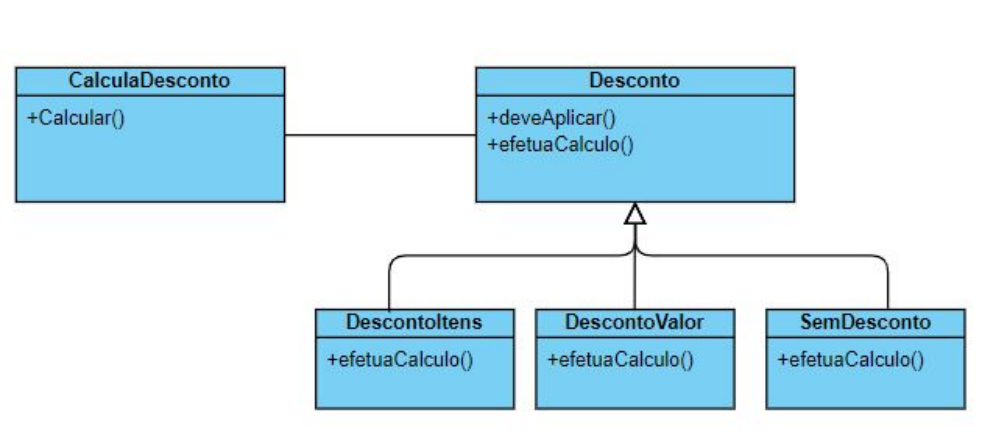
```java
public class CalculaImposto {
    public BigDecimal calcular(Orcamento orcamento, TipoImpostoStrategy tipoImposto){
        return tipoImposto.calcular(orcamento);
    }
}
```

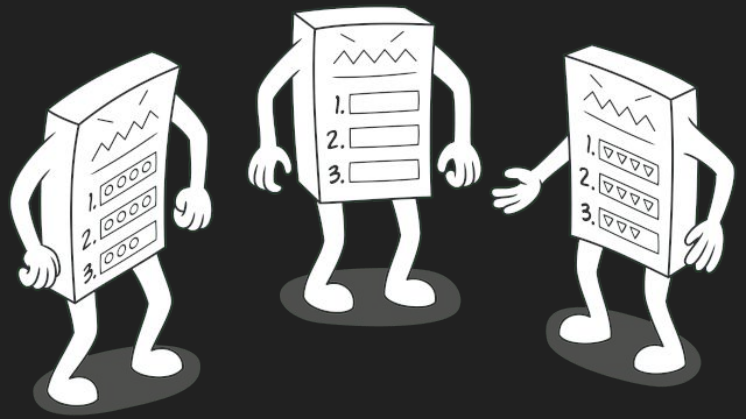
```java
public class ICMS implements TipoImpostoStrategy {
    public BigDecimal calcular(Orcamento orcamento){
        System.out.print("ICMS: ");
        return orcamento.getValor().multiply(new BigDecimal("0.17"));
    }

}
```

```java
public class ISS  implements TipoImpostoStrategy{
    public BigDecimal calcular(Orcamento orcamento){
        System.out.print("ISS: ");
        return orcamento.getValor().multiply(new BigDecimal("0.05"));
    }
}
```

Template Method

Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.





```java
public abstract class Desconto {
    protected Desconto proximo;
    public Desconto( Desconto proximo){
        this.proximo = proximo;
    }
    public BigDecimal calcular(Orcamento orcamento){
        if (deveAplicar(orcamento)){
            return efetuarCalculo(orcamento);
        }
        return proximo.calcular(orcamento);
    }
    protected abstract boolean deveAplicar(Orcamento orcamento);

    protected abstract BigDecimal efetuarCalculo(Orcamento orcamento);

}
```

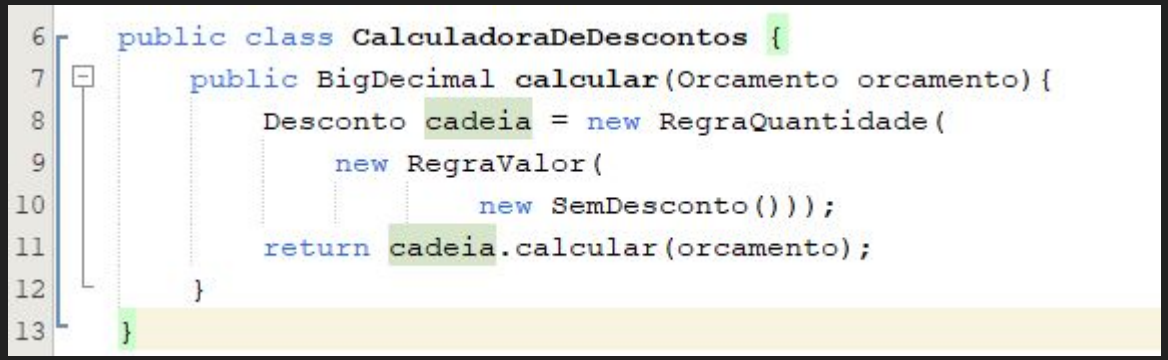
```java
public class RegraQuantidade extends Desconto{
    public RegraQuantidade(Desconto proximo){
        super(proximo);
    }
        public BigDecimal efetuarCalculo(Orcamento orcamento) {
                return orcamento.getValor().multiply(new BigDecimal("0.1"));
        }

        @Override
        public boolean deveAplicar(Orcamento orcamento) {
                return orcamento.getQuantidadeItens() > 10;
        }

}
```

```java
public class SemDesconto extends Desconto {

        public SemDesconto() {
                super(null);
        }
        public BigDecimal efetuarCalculo(Orcamento orcamento) {
                return BigDecimal.ZERO;
        }
        @Override
        public boolean deveAplicar(Orcamento orcamento) {
                return true;
        }
}
```

```java
public class RegraValor extends Desconto{
    public RegraValor(Desconto proximo){
        super(proximo);
    }
    public BigDecimal efetuarCalculo(Orcamento orcamento) {
        return orcamento.getValor().multiply(new BigDecimal("0.1"));
        }

        @Override
    public boolean deveAplicar(Orcamento orcamento) {
        return orcamento.getValor().compareTo(new BigDecimal("500")) > 0;
        }
```

Chain of Responsibility

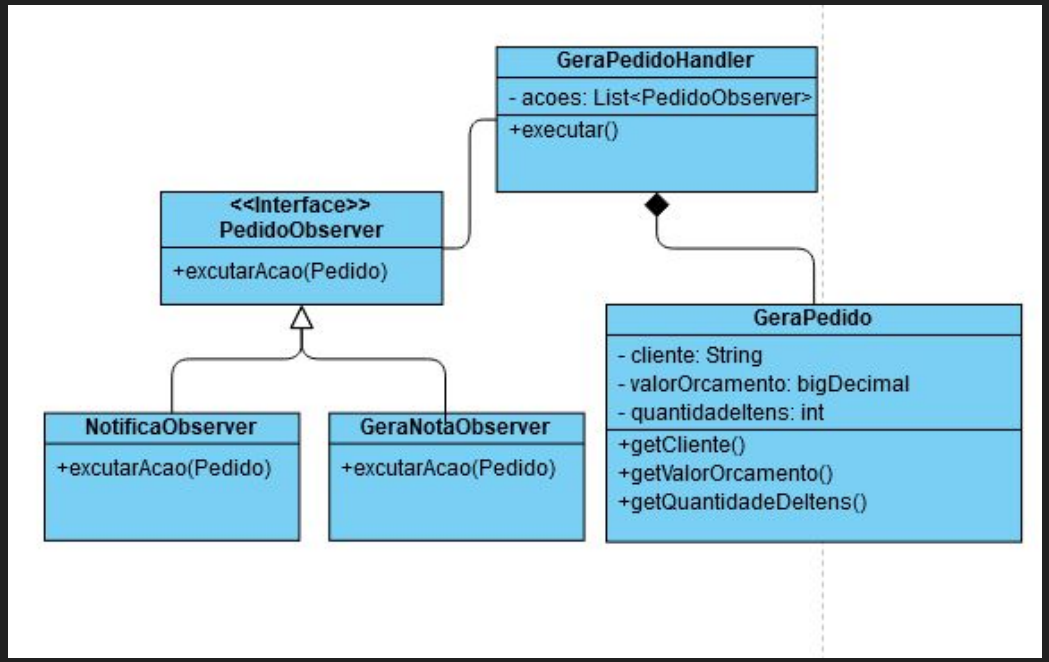
é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.



```
6   public class CalculadoraDeDescontos {
7       public BigDecimal calcular(Orcamento orcamento){
8           Desconto cadeia = new RegraQuantidade(
9               new RegraValor(
10                  new SemDesconto()));
11          return cadeia.calcular(orcamento);
12      }
13  }
```

Observer

é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



```java
public class GeraPedido {

	private String cliente;
	private BigDecimal valorOrcamento;
	private int quantidadeItens;

	public GeraPedido(String cliente, BigDecimal valorOrcamento, int quantidadeItens) {
		this.cliente = cliente;
		this.valorOrcamento = valorOrcamento;
		this.quantidadeItens = quantidadeItens;
	}

	public String getCliente() {
		return cliente;
	}

	public BigDecimal getValorOrcamento() {
		return valorOrcamento;
	}

	public int getQuantidadeItens() {
		return quantidadeItens;
	}
```

```java
public class Pedido {

	private String cliente;
	private LocalDateTime data;
	private Orcamento orcamento;

	public Pedido(String cliente, LocalDateTime data, Orcamento orcamento) {
		this.cliente = cliente;
		this.data = data;
		this.orcamento = orcamento;
	}
	public String getCliente() {
		return cliente;
	}
	public LocalDateTime getData() {
		return data;
	}
	public Orcamento getOrcamento() {
		return orcamento;
	}

}
```

```java
public class GeraPedidoHandler {

	private List<PedidoObserver> Acoes;

	public GeraPedidoHandler(List<PedidoObserver> acoesAposGerarPedidos) {
		Acoes = acoesAposGerarPedidos;
	}

	public void executar(GeraPedido geraPedido) {
		Orcamento orcamento = new Orcamento(geraPedido.getValorOrcamento(), geraPedido.getQuantidadeItens());
		Pedido pedido = new Pedido(geraPedido.getCliente(), LocalDateTime.now(), orcamento);

		this.Acoes.forEach(a -> a.executarAcao(pedido));
	}

}
```

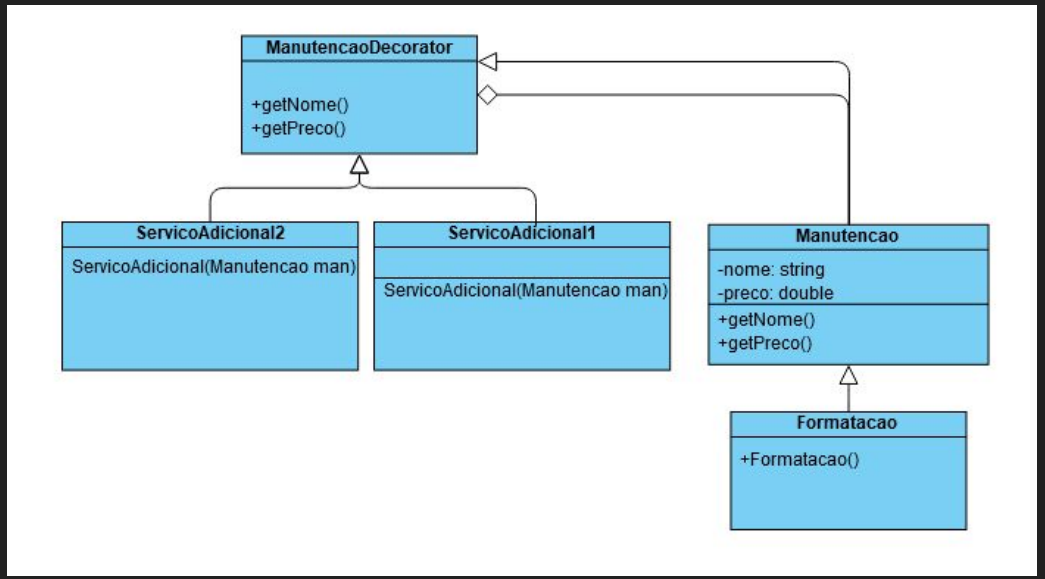
```java
public interface PedidoObserver {

    void executarAcao(Pedido pedido);

}
```

```java
public class NotificarObserver implements PedidoObserver {

    public void executarAcao(Pedido pedido) {
        System.out.println("Compra realizada com sucesso!");
    }

}
```

```java
public class GerarNotaObserver implements PedidoObserver {

    public void executarAcao(Pedido pedido) {
        System.out.println("GERAR NOTA FISCAL");
    }

}
```

Decorator

é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.



```java
package br.edu.ifpb.padroes.sevico;
public abstract class Manutencao {
    String nome;
    double preco;
    public String getNome(){
        return nome;
    }
    public double getPreco(){
        return preco;
    }
}

```

```java
package br.edu.ifpb.padroes.sevico;
public class Formatacao extends Manutencao {
    public Formatacao(){
        nome = "Formataçao";
        preco = 100;
    }

}

```

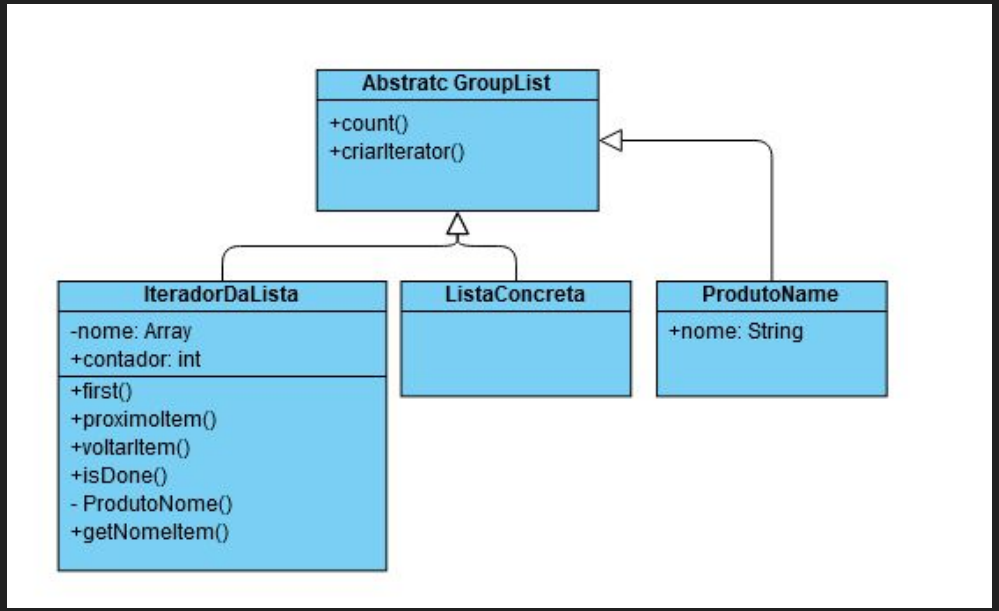
```java
public class ManutencaoDecorator extends Manutencao{
    Manutencao manutencao;
    public ManutencaoDecorator(Manutencao man){
        manutencao = man;
    }
    public String getNome(){
        return manutencao.getNome()+ " + " + nome;
    }
    public double getPreco(){
        return manutencao.getPreco() + preco;
    }
}
```

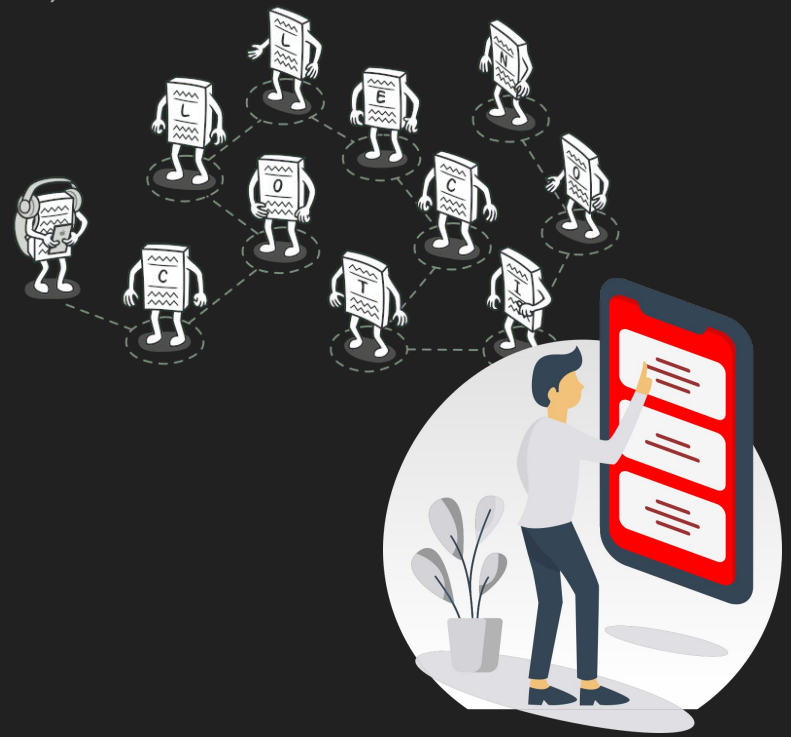
```java
package br.edu.ifpb.padroes.sevico;
public class ServicoAdicional_1_Decorator extends ManutencaoDecorator{
    public ServicoAdicional_1_Decorator(Manutencao man){
        super(man);
        nome = "Limpeza";
        preco = 30;
    }

}
```

```java
package br.edu.ifpb.padroes.sevico;
public class ServicoAdicional_2_Decorator extends ManutencaoDecorator{
    public ServicoAdicional_2_Decorator(Manutencao man){
        super(man);
        nome = "Backup";
        preco = 50;
    }

}
```

Interator

é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.)





```java
import java.util.ArrayList;

public abstract class GroupLists {
        protected ArrayList<ProdutoName> produtos;
        public GroupLists() {
                produtos = new ArrayList<ProdutoName>();
        }
        public int count() {
                return produtos.size();
        }
        public IteradorDaLista criarIterator() {
                return new IteradorDaLista(produtos);
        }
}
```

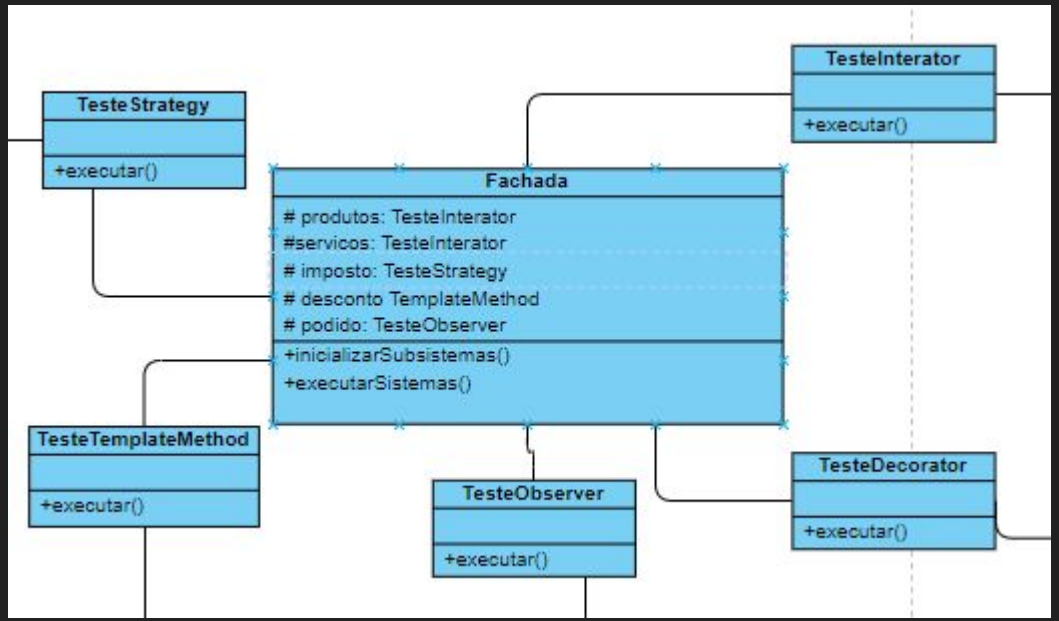
```java
public class ProdutoName {
        String nome;

        public ProdutoName(String nome) {
                this.nome = nome;
        }
}
```

```java
public class ListaConcreta extends GroupLists {
        public ListaConcreta() {
                produtos.add(new ProdutoName("Monitor"));
                produtos.add(new ProdutoName("Computador"));
                produtos.add(new ProdutoName("Mouse"));
                produtos.add(new ProdutoName("Teclado"));
                produtos.add(new ProdutoName("Memoria"));
        }

}
```

```java
        ArrayList<ProdutoName> lista;
        int contador;

        protected IteradorDaLista(ArrayList<ProdutoName> lista) {
                this.lista = lista;
                contador = 0;
        }

        public void first() {
                contador = 0;
        }

        public void proximoItem() {
                contador++;
        }

        public void voltarItem() {
                contador--;
        }

        public boolean isDone() {
                return contador == lista.size();
        }

        private ProdutoName currentItem() {
                if (isDone()) {
                        contador = lista.size() - 1;
                } else if (contador < 0) {
                        contador = 0;
                }
                return lista.get(contador);
        }

        public String getNomeItem() {
                return currentItem().nome;
        }
```

Fachada

é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.



```java
public class SistemasFacade {
    protected TesteInterator produtos;
    protected TesteDecorator servicos;
    protected TesteStrategy imposto;
    protected TesteTemplateMethod desconto;
    protected TestesObserver pedido;



    public void inicializarSubsistemas(){
        System.out.println("##### Configurando subsistemas #####\n");
        produtos = new TesteInterator();
        servicos = new TesteDecorator();
        imposto = new TesteStrategy();
        desconto = new TesteTemplateMethod();
        pedido = new TestesObserver();
    }
    public void executarSistemas(){
        produtos.executar();
        servicos.executar();
        imposto.executar();
        desconto.executar();
        pedido.executar();
    }
}
```

```java
public class Cliente {
    public static void main(String[] args) {
        SistemasFacade fachada = new SistemasFacade();
        fachada.inicializarSubsistemas();
        fachada.executarSistemas();
        }
}
```

Considerações finais

- Banco de dados
- Aplicação