Engenharia de Softwares Escaláveis

Design Patterns e Domain-Driven Design com Java

Agenda

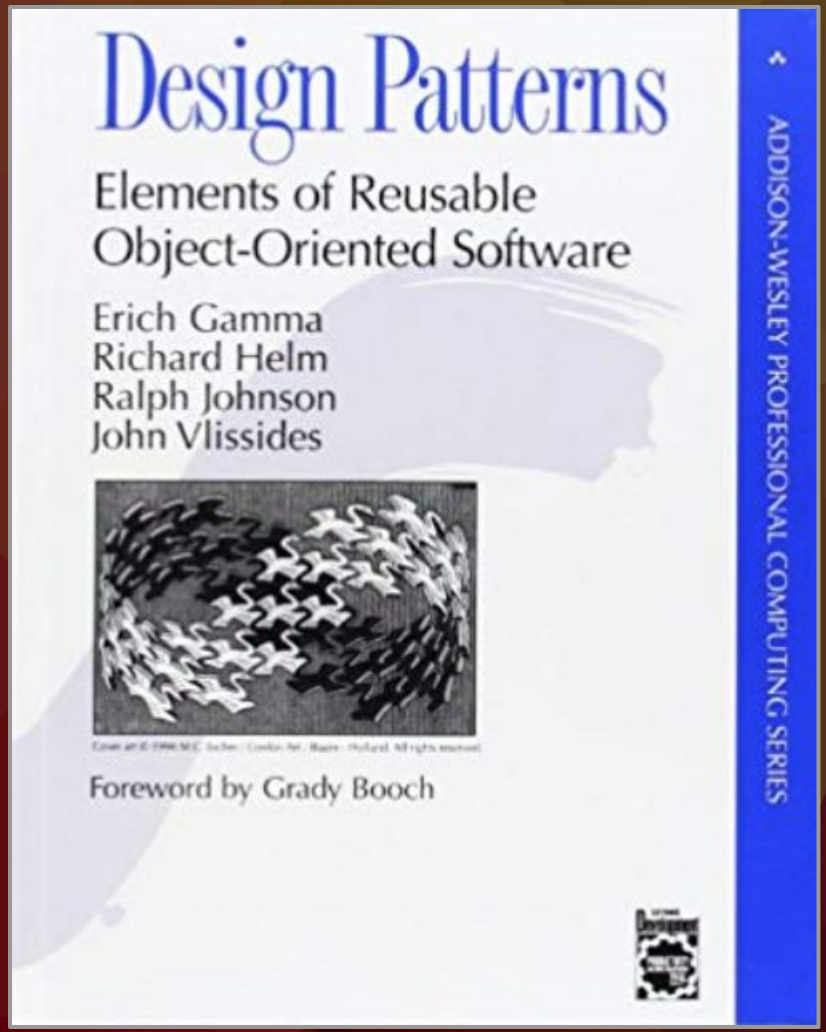
Etapa 2: Desenvolver Software Aplicando Design Patterns.

- Design Patterns.
- Padrões Criacionais.
- Padrões Estruturais.

Design Patterns

A ideia de **Design Patterns** pode ser descrita como um conjunto de abordagens de codificação reutilizáveis que **resolvem os problemas mais comuns** encontrados durante o desenvolvimento de aplicativos.

Essas abordagens estão alinhadas com os conceitos **APIE** e **SOLID** e têm um impacto incrivelmente positivo em trazer transparência, legibilidade e testabilidade ao caminho de desenvolvimento.



Propósito				
Escopo		Creational	Structural	Behavioral
	Classe	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade (ou Facade) Business Delegate Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

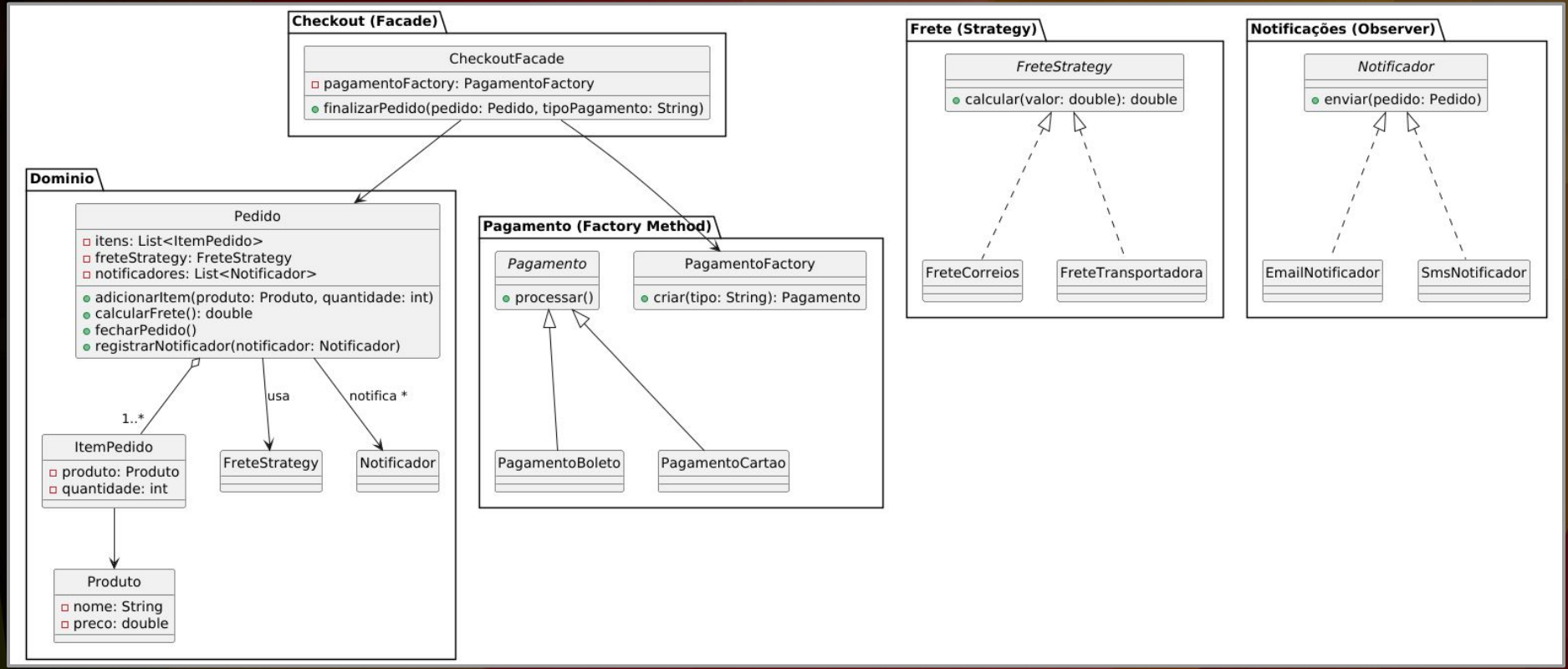
Design Patterns foram concebidos para colocar em prática os princípios **SOLID** e ajudar a reduzir acoplamento, aumentar coesão e permitir mudanças seguras no software.

Princípio SOLID	Padrões Relacionados	Justificativa
Single Responsibility (SRP)	*Builder*, *Command*, *Facade*, *Mediator*	Ajudam a isolar responsabilidades (ex: separar construção de objetos complexos, ou orquestração de tarefas).
Open/Closed (OCP)	*Strategy*, *Decorator*, *Factory Method*, *Command*	Permitem adicionar comportamentos sem alterar código existente (uso de polimorfismo e composição).
Liskov Substitution (LSP)	*Template Method*, *Strategy*, *State*	Padrões baseados em herança ou interfaces, onde substituição de subtipos é essencial.
Interface Segregation (ISP)	*Adapter*, *Proxy*, *Composite*	Permitem criar interfaces específicas e adaptadores que evitam dependência de métodos desnecessários.
Dependency Inversion (DIP)	*Abstract Factory*, *Observer*, *Strategy*, *Bridge*	Todos favorecem abstrações (interfaces), desacoplando implementações concretas.

A melhor forma de aprender **Design Patterns** é combinando teoria estruturada, prática guiada e aplicação em problemas reais.

É essencial aprimorar o domínio sobre **Orientação a Objetos** (Abstração, Encapsulamento, Herança e Polimorfismo), Princípios **SOLID** (muitos padrões existem para apoiar esses princípios) e modelagem de objetos e responsabilidades.

E-Commerce

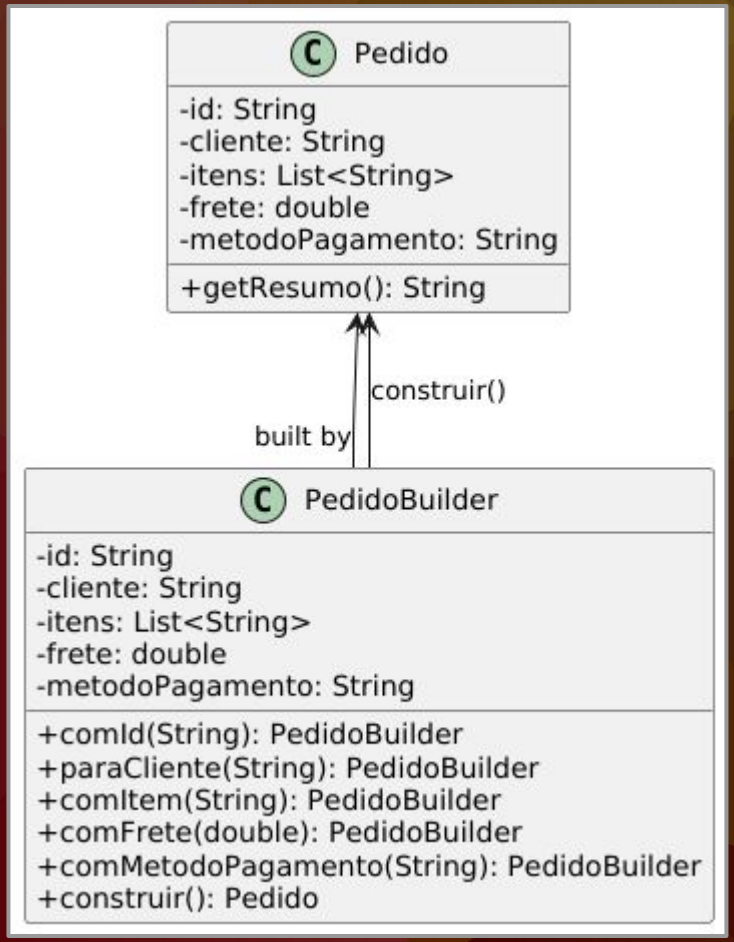

Checkout (Facade)
CheckoutFacade
- pagamentoFactory: PagamentoFactory
+ finalizarPedido(pedido: Pedido, tipoPagamento: String)
Dominio
Pedido
- itens: List<ItemPedido>
- freteStrategy: FreteStrategy
- notificadores: List<Notificador>
+ adicionarItem(produto: Produto, quantidade: int)
+ calcularFrete(): double
+ fecharPedido()
+ registrarNotificador(notificador: Notificador)
1..*
usa
notifica *
ItemPedido
- produto: Produto
- quantidade: int
FreteStrategy
Notificador
Produto
- nome: String
- preco: double
Pagamento (Factory Method)
Pagamento
+ processar()
PagamentoFactory
+ criar(tipo: String): Pagamento
PagamentoBoleto
PagamentoCartao
Frete (Strategy)
FreteStrategy
+ calcular(valor: double): double
FreteCorreios
FreteTransportadora
Notificações (Observer)
Notificador
+ enviar(pedido: Pedido)
EmailNotificador
SmsNotificador

Padrões Criacionais

Builder

Esse padrão serve para construir objetos complexos passo a passo, isolando o processo de construção da representação final do objeto.

Ele é muito útil quando um objeto tem muitos atributos opcionais ou requer um processo de montagem mais complexo.



```java
public class Pedido {
    private List<ItemPedido> itens;
    private FreteStrategy freteStrategy;

    public Pedido(List<ItemPedido> itens, FreteStrategy freteStrategy) {
        this.itens = itens;
        this.freteStrategy = freteStrategy;
    }

    public double getValorTotal() {
        return itens.stream().mapToDouble(ItemPedido::getSubtotal).sum();
    }

    public double calcularFrete() {
        return freteStrategy.calcular(this);
    }

    public List<ItemPedido> getItens() { return itens; }
}
```

```java
public class PedidoBuilder {
    private List<ItemPedido> itens = new ArrayList<>();
    private FreteStrategy freteStrategy;

    public PedidoBuilder comItem(Produto produto, int quantidade) {
        this.itens.add(new ItemPedido(produto, quantidade));
        return this;
    }

    public PedidoBuilder comFrete(FreteStrategy strategy) {
        this.freteStrategy = strategy;
        return this;
    }

    public Pedido build() {
        return new Pedido(itens, freteStrategy);
    }
}
```

```java
public class PedidoTest {

    @Test
    public void deveCalcularValorTotalCorretamente() {
        Produto camiseta = new Produto("Camiseta Polo", 80);
        Produto tenis = new Produto("Tênis Esportivo", 120);

        Pedido pedido = new PedidoBuilder()
                .comItem(camiseta, 1)
                .comItem(tenis, 1)
                .comFrete(new FretePadrao())
                .build();

        assertEquals(200.0, pedido.getValorTotal());
    }
}
```

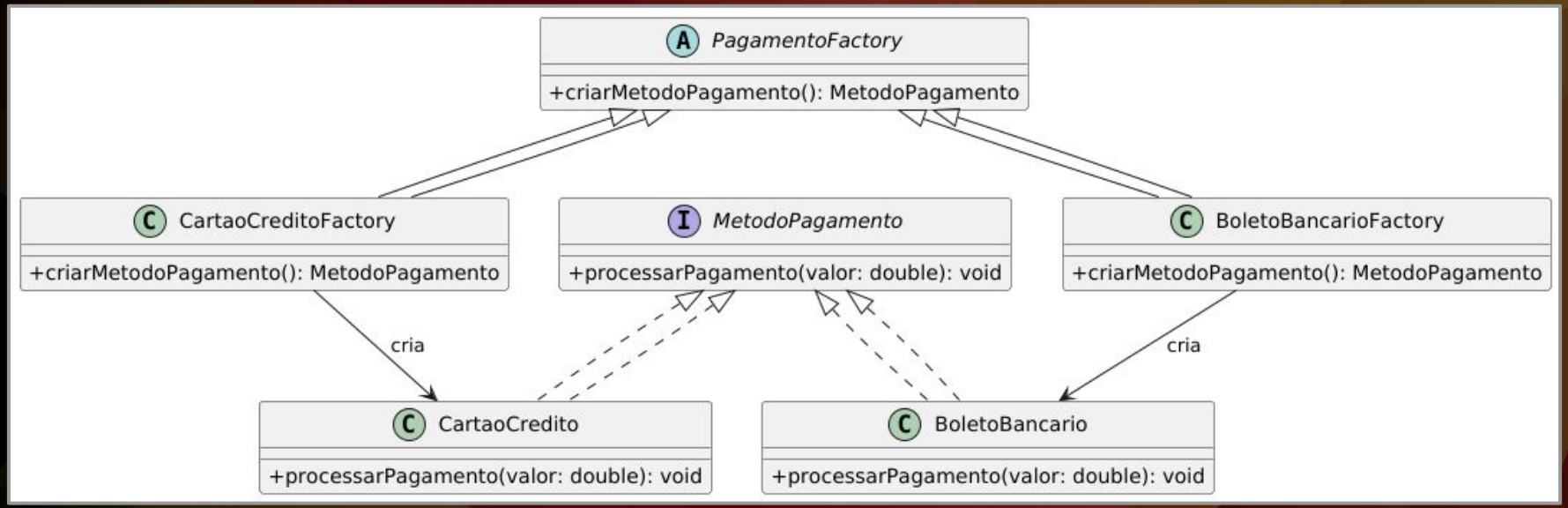
SRP: Com o Builder, o Pedido foca em sua lógica de negócio (como calcular total, aplicar cupom, etc.), enquanto o PedidoBuilder se concentra em como o pedido é criado (ordem, validação de campos, default values etc.).

OCP: O uso de Builder permite estender a forma como um objeto é montado sem alterar a estrutura da classe principal.

Factory Method

O padrão Factory Method define uma interface para criar objetos, mas delega às subclasses a decisão sobre qual classe instanciar.

Ele evita o acoplamento direto ao usar new, centralizando a lógica de criação.



```java
public interface Pagamento {
    void processar();
}

public class PagamentoCartao implements Pagamento {
    public void processar() {
        System.out.println("Pagamento com cartão processado.");
    }
}

public class PagamentoFactory {
    public static Pagamento criar(String tipo) {
        switch (tipo.toLowerCase()) {
            case "cartao": return new PagamentoCartao();
            case "boleto": return new PagamentoBoleto();
            default: throw new IllegalArgumentException("Tipo de pagamento inválido");
        }
    }
}
```

```java
public class PedidoTest {

    @Test
    public void deveProcessarPagamentoCartao() {
        Pagamento pagamento = PagamentoFactory.criar("cartao");
        assertTrue(pagamento instanceof PagamentoCartao);
    }

    @Test
    public void deveProcessarPagamentoBoleto() {
        Pagamento pagamento = PagamentoFactory.criar("boleto");
        assertTrue(pagamento instanceof PagamentoBoleto);
    }
}
```

OCP: novas formas de pagamento são adicionadas sem alterar Pedido nem PagamentoFactory.

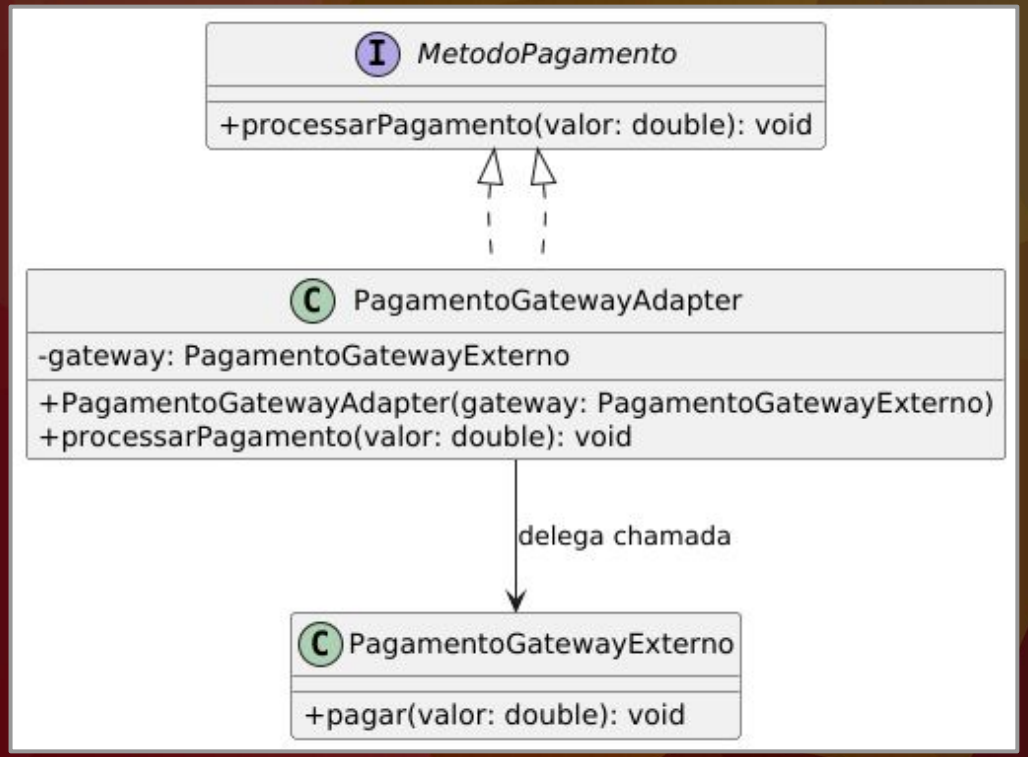
DIP: Pedido depende apenas da abstração Pagamento, permitindo inversão de controle.

Padrões Estruturais

Adapter

O Adapter permite que interfaces incompatíveis colaborem, adaptando uma interface existente para outra esperada pelo cliente, sem alterar o código original.

Você tem um sistema que espera uma interface GatewayPagamento, mas quer usar uma biblioteca externa com a interface ServicoPagSeguro.



```java
public interface GatewayPagamento {
    boolean processarPagamento(Pedido pedido);
}

public class ServicoPagSeguro {
    public boolean realizarTransacao(Pedido p, String token) { ... }
}

public class PagSeguroAdapter implements GatewayPagamento {
    private ServicoPagSeguro servico;

    public boolean processarPagamento(Pedido pedido) {
        return servico.realizarTransacao(pedido, "TOKEN123");
    }
}
```

```
public class PagamentoTest {

    @Test
    public void deveProcessarPagamento() {
        GatewayPagamento gateway = new PagSeguroAdapter();
        boolean ok = pedido.processarPagamento(gateway);
        assertTrue(ok);
    }
}
```

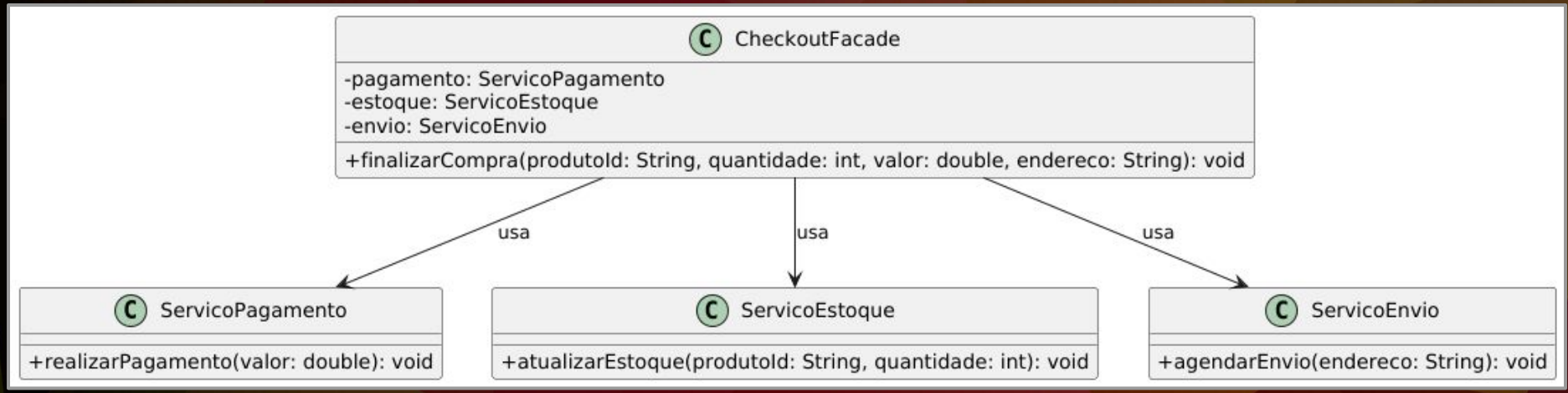
OCP: Adiciona compatibilidade com novas APIs externas sem alterar o cliente.

ISP: Cria interfaces específicas e enxutas para o cliente, evitando dependência em métodos desnecessários que vêm com as APIs de terceiros.

Facade / Façade

O Facade fornece uma interface simplificada e unificada para um conjunto complexo de classes que formam um subsistema.

Ele atua como uma fachada entre o cliente e a lógica interna.



```
public class PedidoFacade {
    private EstoqueService estoque;
    private PagamentoService pagamento;
    private NotaFiscalService notaFiscal;

    public void finalizarPedido(Pedido pedido) {
        estoque.baixarEstoque(pedido);
        pagamento.processar(pedido);
        notaFiscal.emitir(pedido);
    }
}
```

```
public class PedidoController {
    private PedidoFacade pedidoFacade;

    public void confirmarCompra(Pedido pedido) {
        pedidoFacade.finalizarPedido(pedido);
    }
}
```

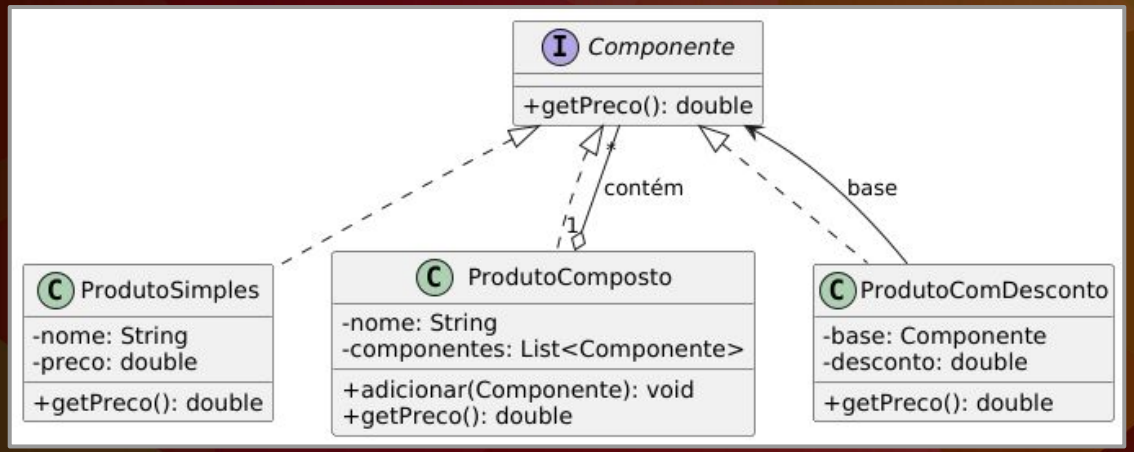
SRP: Isola a responsabilidade de orquestrar operações complexas sem acumular regras de negócio.

DIP: Permite que clientes dependam de uma interface de alto nível, desacoplando-os de implementações internas

Composite

Permite tratar objetos individuais e coleções de objetos de forma uniforme, usando uma interface comum.

Um produto pode ser simples (uma cadeira) ou composto (um conjunto com mesa e 4 cadeiras). Ambos implementam a interface Produto.



```java
public interface Componente {
    double getPreco();
}

public class ProdutoSimples implements Componente {
    public double getPreco() { return preco; }
}

public class ProdutoComposto implements Componente {
    private List<Componente> componentes;
    public double getPreco() {
        return componentes.stream().mapToDouble(Componente::getPreco).sum();
    }
}
```

```
Carrinho carrinho = new Carrinho();
carrinho.adicionarItem(new ProdutoSimples("Mouse", 100.0));
carrinho.adicionarItem(new ProdutoComposto("Kit Teclado + Mouse",
Arrays.asList(
    new ProdutoSimples("Teclado", 200.0),
    new ProdutoSimples("Mouse", 100.0)
)));
System.out.println(carrinho.total()); // Trata todos como Componentes
```

LSP: Produtos simples e compostos são tratados de forma uniforme e intercambiável.

OCP: Novos tipos de componentes podem ser adicionados sem alterar os consumidores.