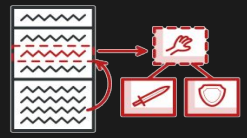
Padrões de projeto comportamentais

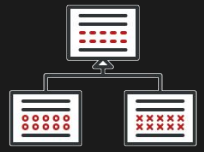
Matheus Barbosa
matheus.barbosa@dcx.ufpb.br

Estes padrões são voltados aos algoritmos e a designação de responsabilidades entre objetos.



Strategy

Permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

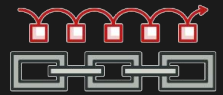


Template Method

Define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.

Observer

Permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

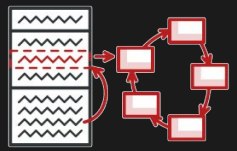


Chain of Responsibility

Permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou passa para o próximo handler da corrente.

Iterator

Permite que você percorra elementos de uma coleção sem expor as representações estruturais deles (lista, pilha, árvore, etc.)



State

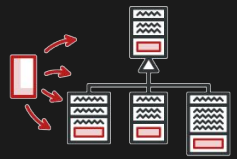
Permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.

Mediator

Permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.

Momento

Permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.



Visitor

Permite que você separe algoritmos dos objetos nos quais eles operam.

Strategy

Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis. Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam

Problema

Você tem diferentes métodos de cálculo de frete: frete padrão, expresso e internacional.

Se você usa `if/else` para escolher o cálculo, o código fica difícil de manter e de estender quando novos tipos de frete forem adicionados.

```java
class ShippingCalculator {
    public double calculate(String type, double weight) {
        if (type.equals("standard")) {
            return weight * 5;
        } else if (type.equals("express")) {
            return weight * 10;
        } else if (type.equals("international")) {
            return weight * 20;
        }
        return 0;
    }
}
```

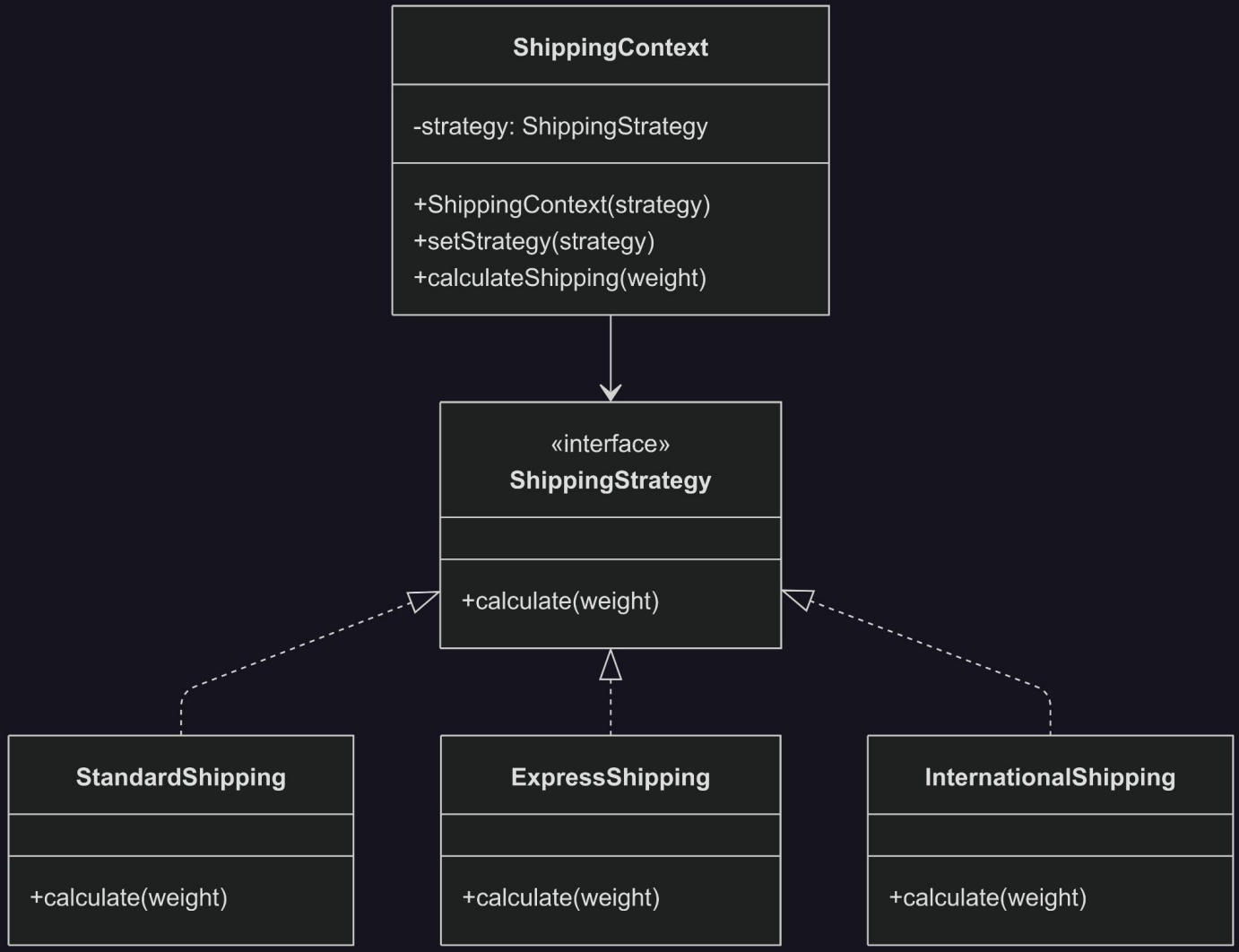
Fere SRP (Princípio da responsabilidade única) e OCP (Princípio Aberto-Fechado)

Obsessão por tipos primitivos

(https://refactoring.guru/smells/primitive-obsession)

Solução

O Strategy separa diferentes maneiras de executar uma tarefa em classes de estratégia. O contexto (A classe original) mantém uma referência a uma estratégia e delegar a execução a ela, sem se preocupar com qual algoritmo específico está sendo usado. O cliente escolhe a estratégia, e o contexto interage com ela apenas por meio de uma interface comum.


ShippingContext
-strategy: ShippingStrategy
+ShippingContext(strategy)
+setStrategy(strategy)
+calculateShipping(weight)
«interface»
ShippingStrategy
+calculate(weight)
StandardShipping
+calculate(weight)
ExpressShipping
+calculate(weight)
InternationalShipping
+calculate(weight)

```
interface ShippingStrategy {
    double calculate(double weight);
}

class StandardShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 5;
    }
}

class ExpressShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 10;
    }
}

class InternationalShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 20;
    }
}
```

```
class ShippingCalculator {
    private ShippingStrategy strategy;

    public ShippingContext(ShippingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(ShippingStrategy strategy) {
        this.strategy = strategy;
    }

    public double calculateShipping(double weight) {
        return strategy.calculate(weight);
    }
}
```

Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.

Template Method

Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.

Problema

Muitas vezes temos algoritmos que seguem passos semelhantes, mas cada variação precisa personalizar apenas um ou dois passos.

Se copiamos e colamos o código em várias classes, o sistema fica cheio de duplicação e difícil de manter.

Cada empresa de transporte (Correios, FedEx, Transportadora Local) segue um processo parecido para realizar a entrega:

1. Preparar pacote.
2. Transportar até o destino.
3. Confirmar entrega.
4. Registrar histórico no sistema.

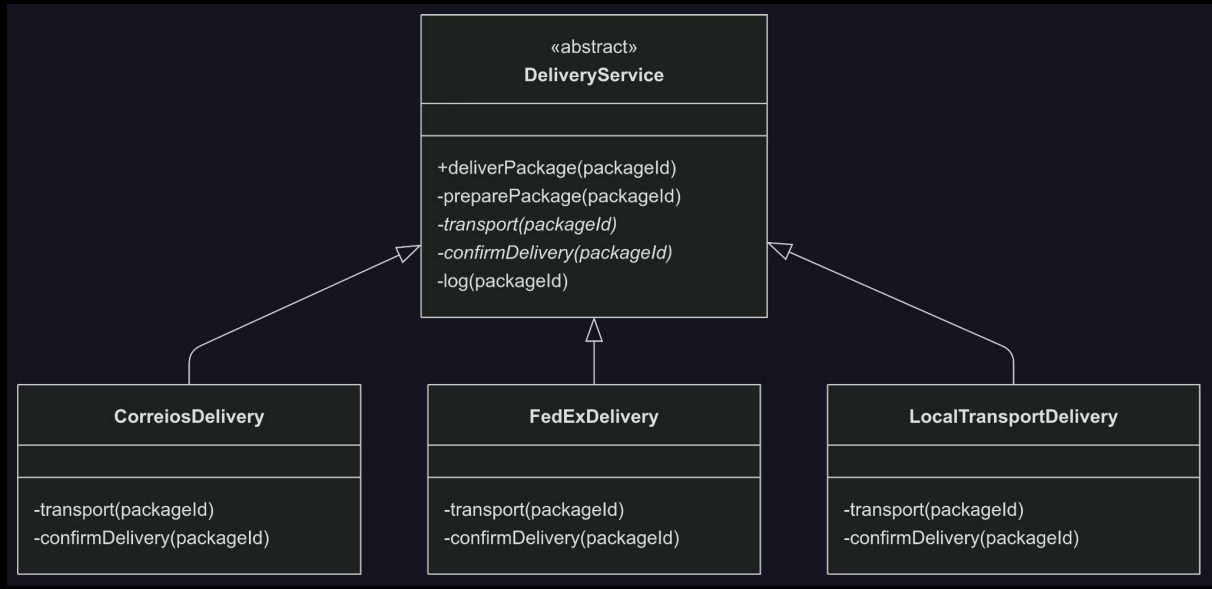
```
class CorreiosDelivery {
    public void deliver(String packageId) {
        preparePackage(packageId); //Preparando pacote..
        transport(packageId); // Levando pacote de caminhão dos Correios...
        confirmDelivery(packageId); // Confirmando entrega com assinatura física...
        log(packageId); // Registrando entrega no sistema.
    }
}

class FedExDelivery {
    public void deliver(String packageId) {
        preparePackage(packageId); //Preparando pacote..
        transport(packageId); // Levando pacote de de avião da FedEx...
        confirmDelivery(packageId); // Confirmando entrega com código digital...
        log(packageId); // Registrando entrega no sistema.
    }
}
```

O que muda é como cada empresa transporta e confirma a entrega, mas a sequência geral é sempre a mesma.

Solução

Criamos uma classe base `DeliveryService` com o método `deliverPackage` (template method).

Subclasses personalizam apenas o transporte e a forma de confirmação.



```java
abstract class DeliveryService {
    // Template Method
    public final void deliverPackage(String packageId) {
        preparePackage(packageId);
        transport(packageId);
        confirmDelivery(packageId);
        log(packageId);
    }

    private void preparePackage(String packageId) {
        System.out.println("Preparando pacote " + packageId);
    }

    protected abstract void transport(String packageId);
    protected abstract void confirmDelivery(String packageId);

    private void log(String packageId) {
        System.out.println("Registrando entrega do pacote "
           + packageId + " no sistema.");
    }
}
```

```java
class CorreiosDelivery extends DeliveryService {
    protected void transport(String packageId) {
        System.out.println("Levando pacote " + packageId + " de caminhão dos Correios...");
    }

    protected void confirmDelivery(String packageId) {
        System.out.println("Confirmando entrega " + packageId + " com assinatura física.");
    }
}

class FedExDelivery extends DeliveryService {
    protected void transport(String packageId) {
        System.out.println("Levando pacote " + packageId + " de avião da FedEx...");
    }

    protected void confirmDelivery(String packageId) {
        System.out.println("Confirmando entrega " + packageId + " com código digital.");
    }
}

class LocalTransportDelivery extends DeliveryService {
    protected void transport(String packageId) {
        System.out.println("Levando pacote " + packageId + " de moto da transportadora
local...");
    }

    protected void confirmDelivery(String packageId) {
        System.out.println("Confirmando entrega " + packageId + " por foto no aplicativo.");
    }
}
```

Implementações do padrão Template Method tendem a ser mais difíceis de se manter quanto mais etapas eles tiverem

Observer

Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente

Problema

Quando um evento acontece (ex.: pacote “Pagamento confirmado”), várias partes do sistema podem precisar reagir:

- O cliente deve receber uma notificação com recibo.
- Estoque debe ser atualizado.
- Liberar pedido para separação/envio.
- Atualizar o dashboard financeiro.

Se a classe `Payment` tiver que chamar manualmente cada uma dessas ações, o código fica acoplado, difícil de manter e de estender.

```java
class Payment {
    private String orderId;

    public Payment(String orderId) {
        this.orderId = orderId;
    }

    public void confirmPayment() {
        System.out.println("Pagamento do pedido " + orderId + " confirmado.");

        // Chamando manualmente cada ação
        System.out.println("Enviando recibo ao cliente...");
        System.out.println("Atualizando estoque...");
        System.out.println("Liberando pedido para separação...");
        System.out.println("Atualizando dashboard financeiro...");
    }
}
```

Solução

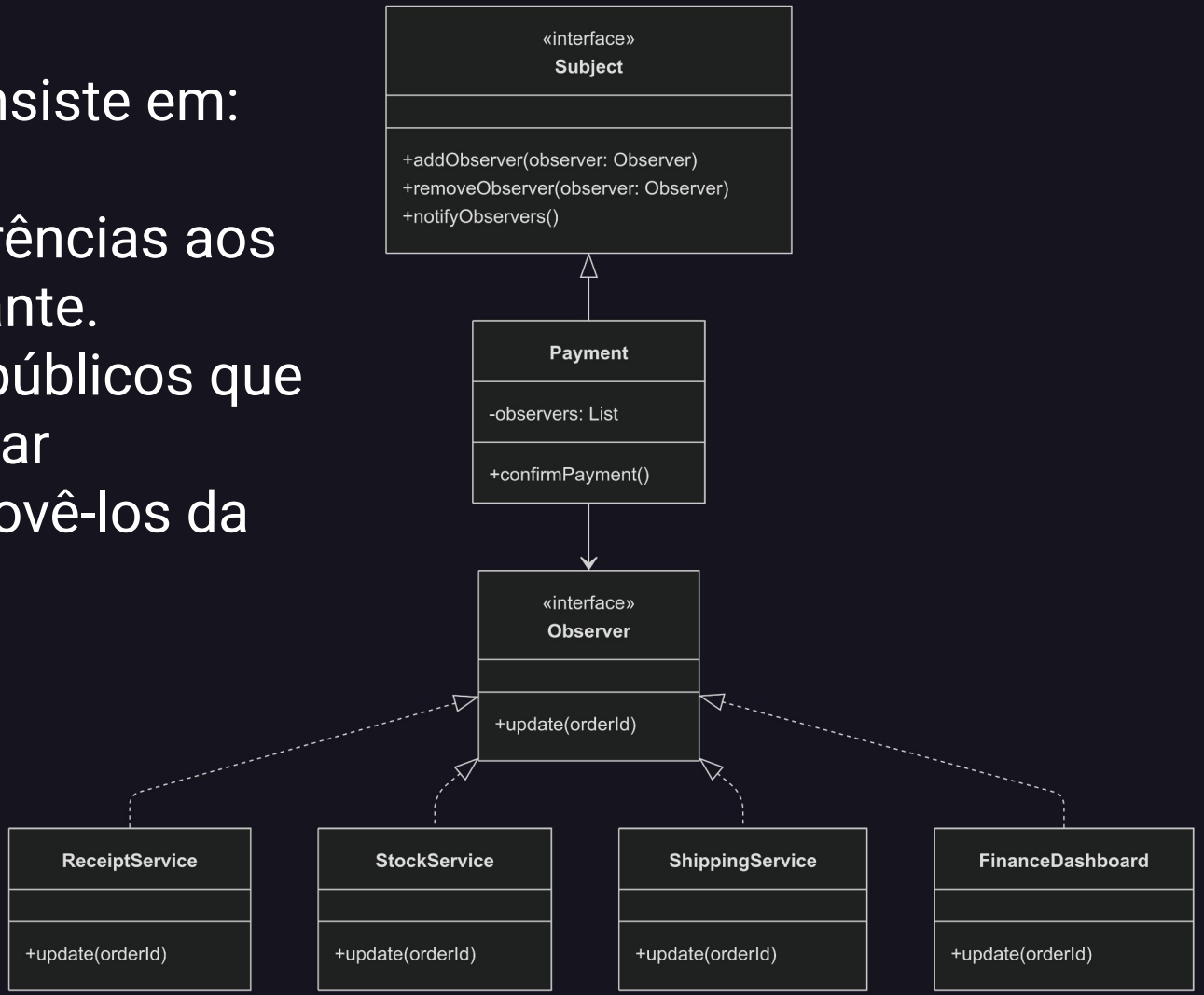
O objeto que tem um estado interessante é quase sempre chamado de *sujeito*

Todos os outros objetos que querem saber das mudanças do estado do *sujeito* são chamados de *observadores*.

O padrão **Observer** sugere que você adicione um mecanismo de assinatura para que objetos *observadores* possam assinar ou desassinar uma corrente de eventos vindo daquela classe *sujeito*

Esse mecanismo consiste em:

1. Uma lista de referências aos objetos do assinante.
2. alguns métodos públicos que permitem adicionar assinantes e removê-los da lista.



```java
interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

class Payment implements Subject {
    private String orderId;
    private List<Observer> observers = new ArrayList<>();

    public Payment(String orderId) {
        this.orderId = orderId;
    }

    public void confirmPayment() {
        System.out.println("Pagamento do pedido " + orderId + " confirmado.");
        notifyObservers(); // Notifica todos os observadores...
    }

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    private void notifyObservers() {
        for (Observer o : observers) {
            o.update(orderId);
        }
    }
}
```

```java
// Observer
interface Observer {
    void update(String orderId);
}

// Observers concretos
class ReceiptService implements Observer {
    public void update(String orderId) {
        System.out.println("Enviando recibo para pedido " + orderId);
    }
}

class StockService implements Observer {
    public void update(String orderId) {
        System.out.println("Atualizando estoque para pedido " + orderId);
    }
}

class ShippingService implements Observer {
    public void update(String orderId) {
        System.out.println("Liberando pedido " + orderId + " para separação e envio");
    }
}

class FinanceDashboard implements Observer {
    public void update(String orderId) {
        System.out.println("Atualizando dashboard financeiro com pedido " + orderId);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Payment payment = new Payment("ORDER123");

        payment.addObserver(new ReceiptService());
        payment.addObserver(new StockService());
        payment.addObserver(new ShippingService());
        payment.addObserver(new FinanceDashboard());

        payment.confirmPayment();
    }
}
```

Assinantes são notificados em ordem aleatória