

Arquitetura de Software

Strategy

Prof. MSc. Jader M. Caldonazzo Garbelini

jadergarbelini@utfpr.edu.br

Departamento de Computação
Universidade Tecnológica Federal do Paraná

Padrões Comportamentais

Padrões Comportamentais

Se preocupam com algoritmos e a atribuição de responsabilidades entre objetos.

Padrões Comportamentais

Não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles.

Padrões Comportamentais

Caracterizam fluxos de controle difíceis de seguir em tempo de execução. Eles afastam o foco do fluxo de controle para permitir que você se concentre somente na maneira como os objetos são interconectados.

Padrões Comportamentais

São divididos em dois tipos: **Classe** e **Objeto**

Padrões Comportamentais

Os padrões comportamentais de **classe** utilizam a herança para distribuir o comportamento entre classes.

Padrões Comportamentais

Exemplo: Template Method e Interpreter;

Padrões Comportamentais

Os padrões comportamentais de **objetos** utilizam a composição de objetos em vez da herança.

Padrões Comportamentais

- Strategy
- Chain of Responsibility
- Observer
- Command

Strategy

Padrão Comportamental

Intenção

Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis.

Intenção

Strategy permite que o algoritmo varie independente dos clientes que o utilizam.

Motivação

Considere os diferentes algoritmos para calcular o frete em um e-commerce. Vamos analisar o seguinte código.



Temos uma classe simples que resolve o problema do cálculo de cada tipo de frete.

Quais são os problemas?

```
602 public class Frete {
603
604     protected String frete;
605
606     public Frete (String frete){
607         this.frete = frete;
608     }
609
610     public double calcular(double distancia) {
611
612         if(frete.equals("sedex")){
613             return distancia * 0.45 + 12;
614         }
615         else if(frete.equals("normal")){
616             return distancia * 0.35 + 10;
617         }
618     }
619
620 }
```

1. Novos tipos: Sedex 10, E-Sedex, Transportadora, etc.

```
602 public class Frete {
603
604     protected String frete;
605
606     public Frete (String frete){
607         this.frete = frete;
608     }
609
610     public double calcular(double distancia) {
611         if(frete.equals("sedex")){
612             return distancia * 0.45 + 12;
613         }
614         else if(frete.equals("normal")){
615             return distancia * 0.35 + 10;
616         }
617     }
618 }
619
620 }
```


Esta inclusão pode gerar muita manutenção na classe, aumentando a possibilidade de erros.

```
602 public class Frete {
603
604     protected String frete;
605
606     public Frete (String frete){
607         this.frete = frete;
608     }
609
610     public double calcular(double distancia) {
611         if(frete.equals("sedex")){
612             return distancia * 0.45 + 12;
613         }
614         else if(frete.equals("normal")){
615             return distancia * 0.35 + 10;
616         }
617     }
618 }
619
620 }
```

Esta manutenção também pode aumentar de acordo com a complexidade dos algoritmos.

```
602 public class Frete {
603
604     protected String frete;
605
606     public Frete (String frete){
607         this.frete = frete;
608     }
609
610     public double calcular(double distancia) {
611
612         if(frete.equals("sedex")){
613             return distancia * 0.45 + 12;
614         }
615         else if(frete.equals("normal")){
616             return distancia * 0.35 + 10;
617         }
618     }
619
620 }
```

Se uma operação é complexa e crítica ao sistema, não é interessante permitir alterações frequentes na classe.

```
602 public class Frete {  
603  
604     protected String frete;  
605  
606     public Frete (String frete){  
607         this.frete = frete;  
608     }  
609  
610     public double calcular(double distancia) {  
611  
612         if(frete.equals("sedex")){  
613             return distancia * 0.45 + 12;  
614         }  
615         else if(frete.equals("normal")){  
616             return distancia * 0.35 + 10;  
617         }  
618     }  
619  
620 }
```

A dica de POO (SOLID) é Responsabilidade Única. A classe deve ser responsável por uma única tarefa.

```
602 public class Frete {
603
604     protected String frete;
605
606     public Frete (String frete){
607         this.frete = frete;
608     }
609
610     public double calcular(double distancia) {
611
612         if(frete.equals("sedex")){
613             return distancia * 0.45 + 12;
614         }
615         else if(frete.equals("normal")){
616             return distancia * 0.35 + 10;
617         }
618     }
619
620 }
```

Como resolver?

Como resolver?

- 1 Definir uma família de algoritmos;
- 2 Encapsular estes algoritmos;
- 3 Tornar estes algoritmos intercambiáveis.

Como resolver?

- 1 Definir uma família de algoritmos;
- 2 Encapsular estes algoritmos;
- 3 Tornar estes algoritmos intercambiáveis.

Como resolver?

- 1 Definir uma família de algoritmos;
- 2 Encapsular estes algoritmos;
- 3 Tornar estes algoritmos intercambiáveis.

1

Definir uma família de algoritmos;



2

Encapsular estes algoritmos

```
626 public class Sedex {  
627     public double calcular (double distancia){  
628         return distancia * 0.45 + 12  
629     }  
630 }  
631  
632  
633  
634 public class Pac {  
635     public double calcular (double distancia){  
636         return distancia * 0.25 + 10  
637     }  
638 }  
639  
640  
641  
642 public class Transportadora {  
643     public double calcular (double distancia){  
644         return distancia * 0.50 + 15  
645     }  
646 }  
647  
648 }
```

3

Tornar os algoritmos intercambiáveis

```
626 public class Sedex {  
627     [public double calcular (double distancia){  
628         return distancia * 0.45 + 12  
629     }  
630 }  
631  
632  
633  
634 public class Pac {  
635     [public double calcular (double distancia){  
636         return distancia * 0.25 + 10  
637     }  
638 }  
639  
640  
641  
642 public class Transportadora {  
643     [public double calcular (double distancia){  
644         return distancia * 0.50 + 15  
645     }  
646 }  
647  
648 }
```

Repare que a assinatura dos métodos é igual.

3

Tornar os algoritmos intercambiáveis

```
626 public class Sedex {  
627     [public double calcular (double distancia){  
628         return distancia * 0.45 + 12  
629     }  
630 }  
631  
632  
633  
634 public class Pac {  
635     [public double calcular (double distancia){  
636         return distancia * 0.25 + 10  
637     }  
638 }  
639  
640  
641  
642 public class Transportadora {  
643     [public double calcular (double distancia){  
644         return distancia * 0.50 + 15  
645     }  
646 }  
647  
648 }
```

Então, podemos
definir uma interface
única para estes
algoritmos

3

Tornar os algoritmos intercambiáveis

```
653 interface Frete {
654
655     public double calcular (double distancia);
656 }
657
658 public class Sedex implements Frete {
659
660     public double calcular (double distancia){
661
662         return distancia * 0.45 + 12
663     }
664 }
665
666 public class Cliente {
667
668     public void frete(double distancia, Frete frete){
669
670         System.out.println("O frete é:" + frete.calcular(distancia));
671     }
672 }
```

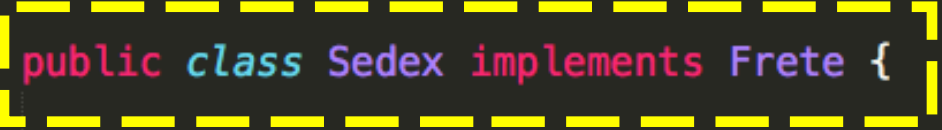
3

Tornar os algoritmos intercambiáveis

```
653 interface Frete {  
654     public double calcular (double distancia);  
656 }  
657  
658 public class Sedex implements Frete {  
659     public double calcular (double distancia){  
660         return distancia * 0.45 + 12  
663     }  
664 }  
665  
666 public class Cliente {  
667     public void frete(double distancia, Frete frete){  
669         System.out.println("O frete é:" + frete.calcular(distancia));  
671     }  
672 }
```

3

Tornar os algoritmos intercambiáveis

```
653 interface Frete {  
654     public double calcular (double distancia);  
655 }  
657  public class Sedex implements Frete {  
659     public double calcular (double distancia){  
660         return distancia * 0.45 + 12  
661     }  
662 }  
663  
664  
665  
666 public class Cliente {  
667     public void frete(double distancia, Frete frete){  
668         System.out.println("O frete é:" + frete.calcular(distancia));  
669     }  
670 }  
671  
672 }
```

3

Tornar os algoritmos intercambiáveis

```
653 interface Frete {  
654     public double calcular (double distancia);  
656 }  
657  
658 public class Sedex implements Frete {  
659     public double calcular (double distancia){  
660         return distancia * 0.45 + 12  
663     }  
664 }  
665  
666 public class Cliente {  
667     public void frete(double distancia, Frete frete){  
669         System.out.println("O frete é:" + frete.calcular(distancia));  
671     }  
672 }
```

O Cliente deve utilizar um objeto do tipo Frete.

Assim, pode alterar o algoritmo quando desejar

Aplicabilidade

Aplicabilidade

Muitas classes relacionadas diferem somente no seu comportamento. As estratégias possibilitam configurar uma classe com um dentre muitos comportamentos;

Aplicabilidade

Quando você precisar de variações de um algoritmo.

Por exemplo, considere os filtros de fotos do Instagram: Sépia, Preto e Branco, Vintage, entre outros.

Aplicabilidade

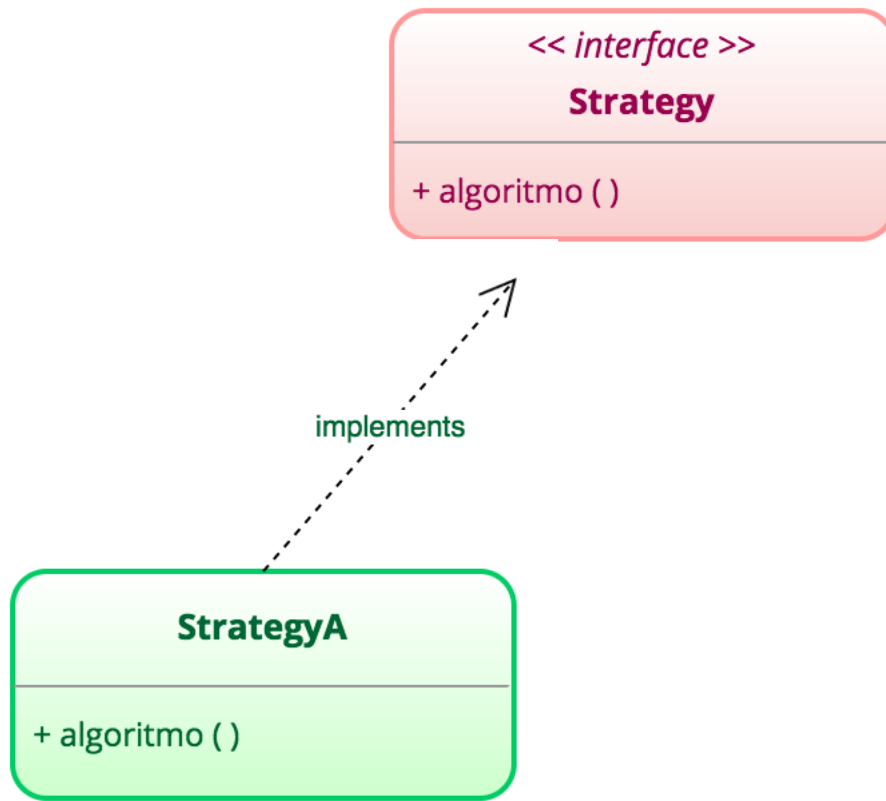
Um algoritmo usa dados dos quais os clientes não devem ter conhecimento. Use o padrão Strategy para evitar a exposição das estruturas de dados complexas.

Estrutura

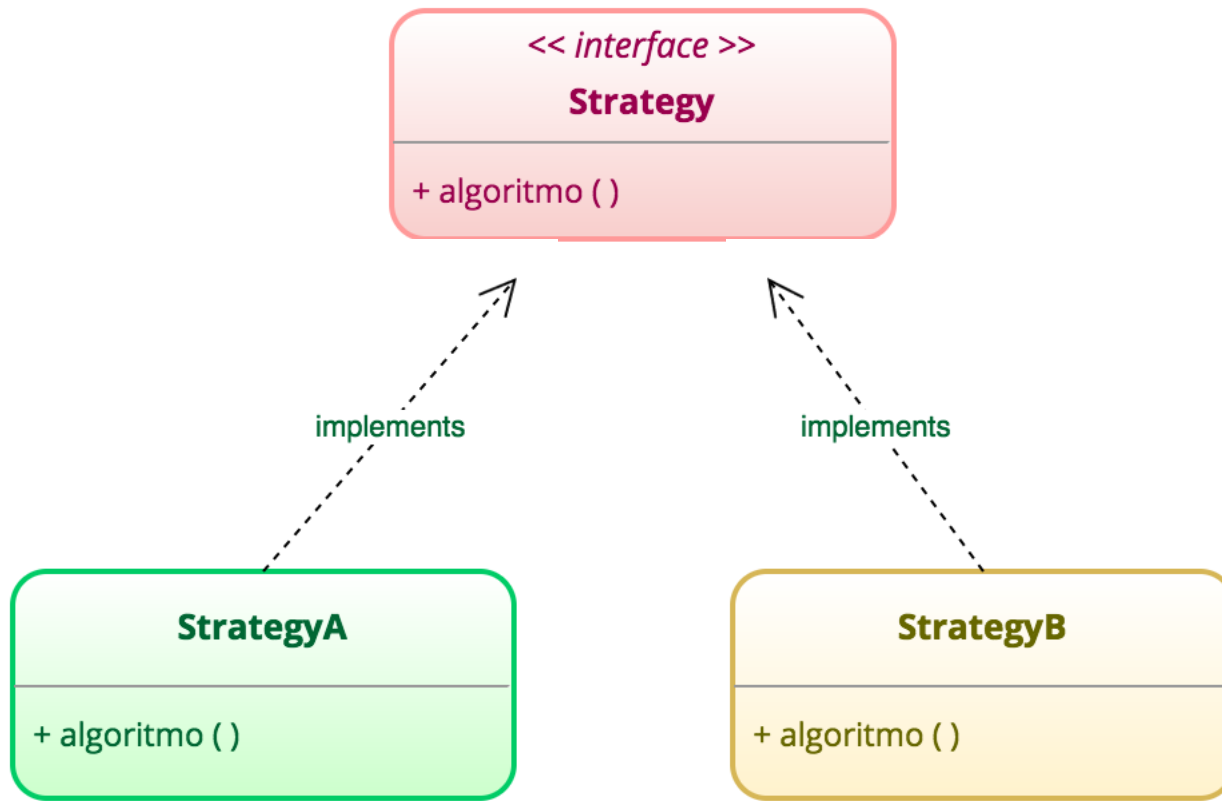
Estrutura



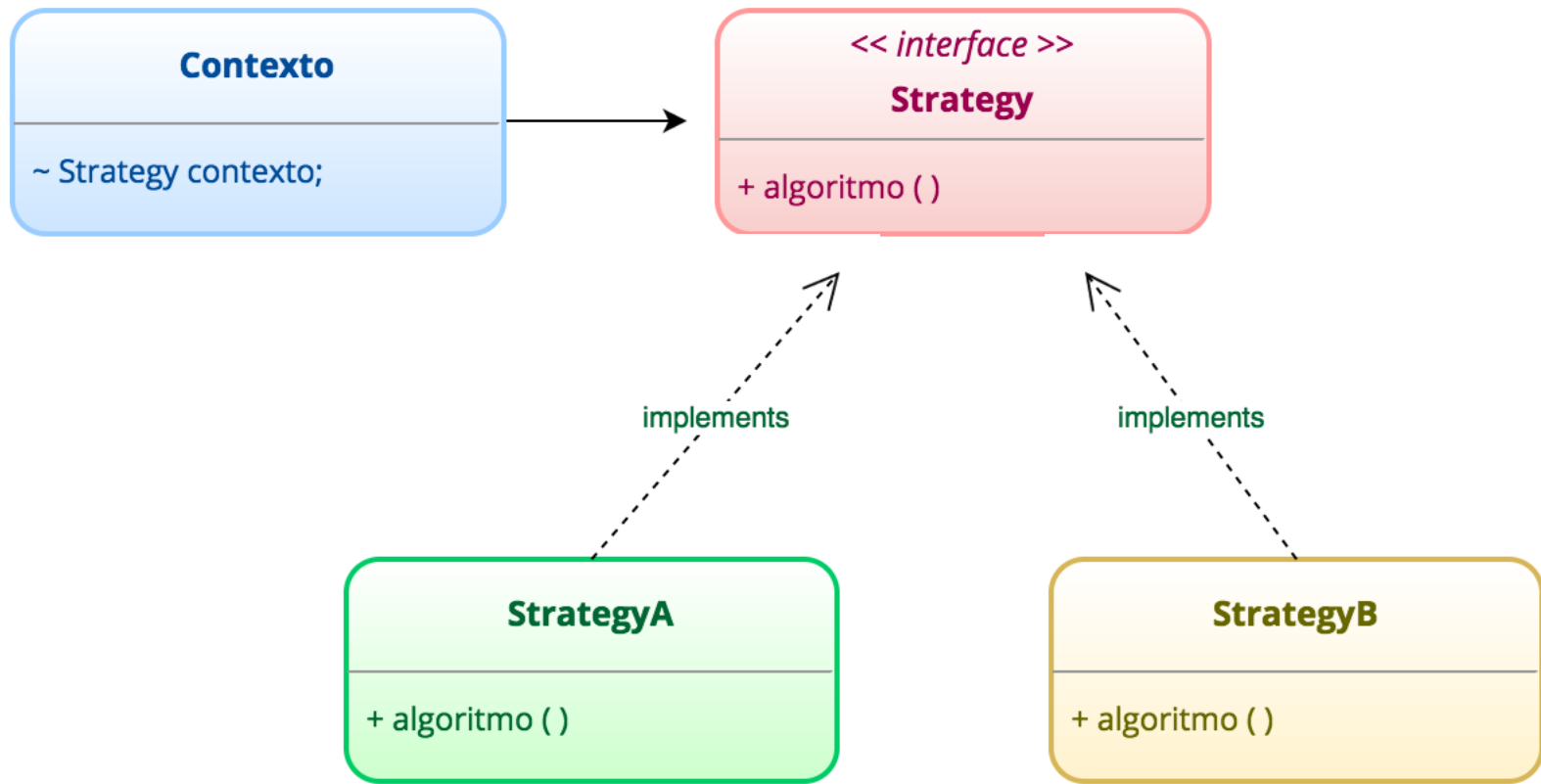
Estrutura



Estrutura



Estrutura



Participantes

Strategy: define a interface para todos os algoritmos relacionados (suportados).

Participantes

Strategy Concreta: implementa o algoritmo a partir da interface definida.

Participantes

Contexto: é configurado com um objeto Strategy Concreto.

Participantes

Contexto: mantém uma referência a um objeto
Strategy.

Colaborações

Strategy e **Contexto** interagem para implementar o algoritmo escolhido. Um contexto pode passar todos os dados requeridos pelo algoritmo para a estratégia quando o algoritmo é chamado.

Colaborações

Alternativamente, o contexto pode passar a si próprio como argumento para operações de Strategy. Isto permite à estratégia chamar de volta o contexto conforme requerido. **Exemplo:**
Frete.

Colaborações

Um contexto repassa solicitações dos seus clientes para sua estratégia. Os clientes usualmente criam e passam um objeto **Strategy Concreto** para o contexto; após isso, interagem exclusivamente com o contexto.

Exemplo

Exemplo

Vamos considerar os algoritmos para fazer o **Log** de erros em uma aplicação.

Exemplo

Considere os algoritmos: **LogFile**, **LogDatabase**
e **LogAPI**

Exemplo

Então, vamos definir uma interface única para o log, chamada **Logger**.

```
679 interface Logger {  
680  
681     // Método recebe uma string mensagem, então  
682     // faz a gravação do log e, em seguida,  
683     // retorna verdadeiro ou falso.  
684  
685     public boolean log (String mensagem);  
686  
687 }
```

Exemplo

Em seguida, implementamos cada algoritmo seguindo a interface especificada.

```
690 public class LogFile implements Logger {
691
692     public boolean log (String mensagem){
693
694         System.out.println("Mensagem: " + mensagem + " gravada no arquivo!");
695         return true;
696     }
697 }
698
699 public class LogDatabase implements Logger {
700
701     public boolean log (String mensagem){
702
703         System.out.println("Mensagem: " + mensagem + " gravada no Banco de Dados!");
704         return true;
705     }
706 }
707
```

Exemplo

Desta forma temos o encapsulamento e possibilitamos que os algoritmos sejam intercambiáveis.

```
690 public class LogFile implements Logger {
691
692     public boolean log (String mensagem){
693
694         System.out.println("Mensagem: " + mensagem + " gravada no arquivo!");
695         return true;
696     }
697 }
698
699 public class LogDatabase implements Logger {
700
701     public boolean log (String mensagem){
702
703         System.out.println("Mensagem: " + mensagem + " gravada no Banco de Dados!");
704         return true;
705     }
706 }
```

Exemplo

A seguir, definimos o contexto em nosso cliente. Neste exemplo, vamos utilizar o objeto como parâmetro.

```
711 public class Cliente {
712
713     // Definimos o contexto, que contem uma estratégia genérica.
714     public void gravarLog(String mensagem, Logger logger){
715
716         logger.log(mensagem);
717     }
718
719     public static void main(String args[]){
720
721         // Nosso cliente pode utilizar diferentes estratégias para
722         // o mesmo problema: Log
723         new Cliente().gravarLog("Teste", new LogFile());
724     }
725 }
```

Consequências

Consequências

Famílias de algoritmos relacionados.

Cada algoritmo é encapsulado em uma classe específica, facilitando a manutenção e o uso no cliente.

Consequências

Variar um algoritmo em tempo de execução.

Ao usar uma interface genérica e utilizar objetos que implementam esta interface, o cliente tem a possibilidade de alterar um algoritmo dinamicamente.

Consequências

O Strategy elimina a necessidade de comandos condicionais. Os algoritmos são encapsulados em classes próprias, não sendo necessários condicionais.

Consequências

Aumento no número de objetos. Ao usar diversas estratégias em uma aplicação, uma desvantagem pode ser um grande número de objetos criados.

Alguns usos

- Inteligência Artificial;
- Processamento de Imagem;
- Reconhecimento de padrões (texto e imagem);
- Criptografia;
- Autenticação ...

Exercícios

Exercícios

- Qual o objetivo do padrão Strategy?
- Descreva outra situação onde o Strategy pode ser usado.
- É possível usar os padrões Strategy e Factory Method juntos?
- Implemente e teste os exemplos apresentados na aula.