Universidade Federal de Uberlândia
Faculdade de Computação
Programação Orientada a Objetos II
Prof. Fabiano Azevedo Dorça

Padrão Strategy (Estratégia)

Define uma família de comportamentos, que podem ser utilizados de forma intercambiável.

O padrão Strategy permite que o algoritmo varie independentemente dos clientes que o usam. Cada comportamento é encapsulado numa classe.

Para usar o padrão Strategy, deve-se perceber o que pode mudar no seu código e encapsular.

O padrão pode ser aplicado quando tem-se operações comuns a uma série de objetos de classes diferentes, podendo-se facilmente criar novos comportamentos sem alterar o que está pronto (aberto para extensão, fechado para modificação).

Participantes:

As classes e/ou objetos que participam no padrão são:

· *Strategy*
- Declara uma interface comum a todos os comportamentos suportados. Context utiliza esta interface para evocar o comportamento definido por um ConcreteStrategy.

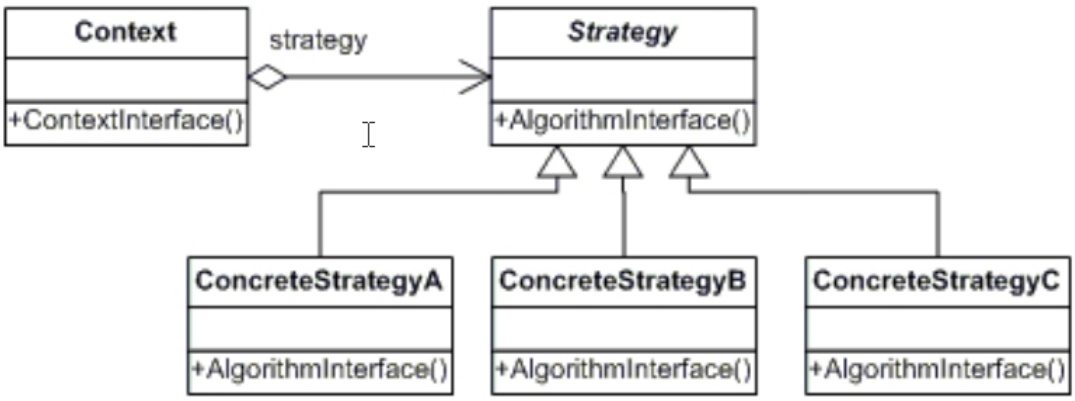
· *ConcreteStrategy*
- Implementa o algoritmo utilizando a interface Strategy.

· *Context*
- Utiliza os comportamentos (*ConcreteStrategy)*
- É composto com um objeto ConcreteStrategy.
- Mantém uma referência para um objeto Strategy.
- Pode definir uma interface para permitir o acesso de Strategy aos seus dados.

- As classes Context instanciam os objetos Strategy e invocam o método AlgorithmInterface passando os parâmetros solicitados;
- A interface Strategy decide qual das implementações ConcreteStrategy deve atender a chamada;
- Esse padrão tem como elementos participantes o Context, que tem seu "comportamento" ou parte dele definido pelo algoritmo implementado pela Strategy referenciada; o Strategy, que define a interface comum para todos os algoritmos; o ConcreteStrategy, que implementa os algoritmos definidos pela interface Strategy.

O padrão Strategy te conduz a seguinte orientação:

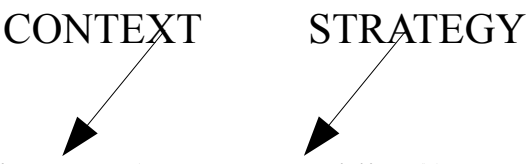
1 - Programe sempre para interfaces;
2 - Dê preferência a composição ao invés de herança;



Exemplo 01:

```
// Padrão Strategy

// Strategy
interface Preco {
	double algoritmo(double p);
}
```

```
// As diferentes estratégias
class PrecoPublico implements Preco {
	public double algoritmo(double p) {
		return p;
	}
}

class PrecoCredito implements Preco {
	public double algoritmo(double p) {
		return (p*1.2);
	}
}

class PrecoVip implements Preco {
	public double algoritmo(double p) {
		return (p*0.8);
	}
}

// O "Context" controla a estratégia
class DeterminaPreco {
	private Preco politicaPreco;
	public DeterminaPreco(Preco estrat) {
		politicaPreco = estrat;
	}

	double precoAplicavel(double p) {
		return politicaPreco.algoritmo(p);
	}

	void trocaAlgoritmo(Preco novoAlgoritmo) {
		politicaPreco = novoAlgoritmo;
	}
}
```




```
// A classe Cliente
public class testaPadrao {
	static DeterminaPreco dt =new DeterminaPreco(new PrecoPublico());
	static double preco = 40.0;
```

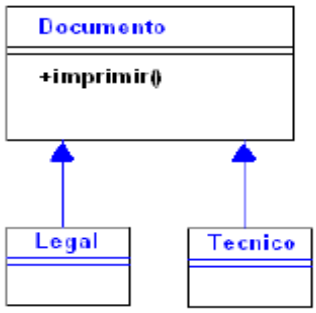
```
        public static void print(double p) {
                System.out.println(p);
        }

        public static void main(String args[]) {
                print(dt.precoAplicavel(preco));
                dt.trocaAlgoritmo(new PrecoVip());
                print(dt.precoAplicavel(preco));
        }
}
```

Resultado da execução:

```
/*
40.0
32.0
*/
```

Exemplo 02:

Imagine que temos um pequeno sistema que apresenta uma classe abstrata Documento
que possui um método chamado imprimir();



Tem-se duas classes:

1. Legal - que herda da classe Documento e sobrescreve o método imprimir() para imprimir documentos Legais;

2. Tecnico - que também herda de Documento e sobrescreve o método imprimir() para imprimir documentos técnicos;

PROBLEMA: Poderia se ter muitas outras classes herdando de documento e cada uma sobrescrevendo o método imprimir() para atender um requisito de impressão diferente como : HTML, PDF, TEXTO, RTF , etc.

O método imprimir() vai ser implementado de forma diferente para cada requisito de impressão e vai ter que ser sobrescrito de forma distinta em cada uma das classes que herdam da classe Documento.

Aqui é que entra o padrão Strategy e para aplicá-lo vamos usar as premissas:
1 - Programe sempre para interfaces;
2 - Dê preferência a composição ao invés de herança;
3 – Isole o comportamento que varia.

Vamos fazer isso removendo o método imprimir() da classe Documento e criar uma interface chamada IImprimir na solução definindo a assinatura do método imprimir():

```
public interface IImprimir{
        public abstract void imprimir(String conteudo);
}
```

A seguir vamos criar na solução duas novas classes que representam uma particularidade de impressão e que irão implementar a interface IImprimir():

```
public class PDF implements IImprimir{
        public void imprimir(String conteudo) {
                System.out.println("Imprimindo PDF: " + conteudo);
        }
}

public class HTML implements IImprimir{
        public void imprimir(String conteudo) {
                System.out.println("Imprimindo HTML: " + conteudo);
        }
}
```

Temos agora duas classes que estão implementando a partir da interface IImprimir a impressão de acordo com sua particularidade vamos então remover o método imprimir() da classe Documento e efetuar alguns ajustes conforme a seguir:

```
public abstract class Documento {
	protected Iimprimir iimprimir;
	public abstract void executaImprimir();

	public void setImprimir(Iimprimir iimprimir) {
		this.iimprimir = iimprimir;
	}
}

public class Legal extends Documento{
	public void executaImprimir(){
		iimprimir.imprimir("CONTEUDO LEGAL");

}

public class Tecnico extends Documento{
	public void executaImprimir(){
		iimprimir.imprimir("CONTEUDO TECNICO");

}
```

Finalmente para testar a estratégia vamos definir o código abaixo no módulo da solução:

```
public void main(String args[]){
	Documento doc = new Legal();
	doc.setImprimir(new PDF());
	doc.executaImprimir();
	doc.setImprimir(new HTML());
	doc.executaImprimir();
}
```

Saída:

```
Imprimindo PDF:  CONTEUDO LEGAL
Imprimindo HTML:  CONTEUDO LEGAL
```

No código anterior criamos uma instância da classe Legal e imprimimos PDF; em seguida imprimimos HTML provando que podemos alterar o comportamento em tempo de execução.

Exemplo 03:

Pretende-se aplicar uma política de descontos no preço dos seus produtos onde cada produto poderá, de acordo com as datas especiais do calendário: dia das mães , dia dos pais , dia das crianças , páscoa, natal , etc..., ter um preço promocional.

Neste cenário teríamos para cada data um tipo de desconto diferente que deverá ser implementado por um algoritmo diferente.

1- Primeiro definimos uma interface chamada IPromocao contendo a definição de um método chamado desconto();

As interfaces não podem ter variáveis, nem ter implementações básicas e nem possuir modificadores de acesso nos seus métodos dela; quem deve fazer isso é a classe que a implementa.

```
public interface IPromocao
{
	float desconto();
}
```

2- Em seguida criamos duas classes que irão implementar esta interface efetuando um desconto para uma das datas especiais. Como exemplo usei os nomes PromocaoNatal e PromocaoNamorados:

```
class PromocaoNatal implements IPromocao{
	public float desconto(){
		return 10;
	}
}
```

```
class PromocaoNamorados implements IPromocao{
	public float desconto(){
		return 15;
	}
}
```

3- Definimos uma classe abstrata chamada Produto que irá retornar o resultado do método desconto da Interface definida:

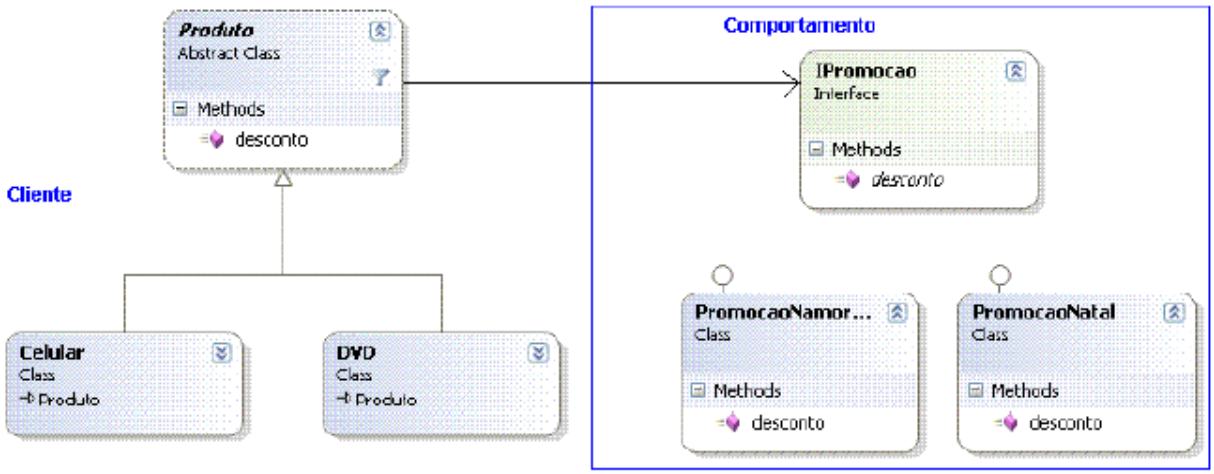
```
public abstract class Produto{
	protected IPromocao promocao;
	public float desconto(){
		return promocao.desconto();
	}
}
```

4- Finalmente criamos as classes DVD e Celular para cada produto que herda de Produto e define um tipo de promoção diferente. Havendo uma nova promoção basta definir a nova classe para a promoção e a classe do produto usar a nova promoção:

```
public class DVD extends Produto{
	public DVD(IPromocao p){
		this.promocao = p;
	}
}
public class Celular extends Produto{
	public Celular(IPromocao p){
		this.promocao = p;
	}
}
class Teste{
	public static void main(String args[]){
		Produto p = new DVD(new PromocaoNamorados());
		System.out.println(p.desconto());
		….
		p = new DVD(new PromocaoNatal());
		System.out.println(p.desconto());
	}
}
```

Caso se deseje alterar a política de desconto de um produto dinamicamente durante a execução, é necessário criar método set, conforme exemplo anterior.

A seguir temos o Diagrama de classes para as classes descritas acima :



Finalmente:

- Identifique um algoritmo, ou seja um comportamento, que deverá ser usado por um cliente;
- Defina uma assinatura para o algoritmo em uma interface;
- Efetua os detalhes da implementação em classes derivadas que implementam a interface;
- Defina classes concretas no cliente para usar o algoritmo.

Referências

FREEMAN, Eric; FREEMAN, Elisabeth. **Use a cabeça! padrões de projeto**. Rio de Janeiro: Atlas Books, 2005.

Macoratti.net. **Padrões de projeto - Usando Strategy**. Disponível em: http://www.macoratti.net