Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

agosto 31, 2011

Mão na massa: Strategy

Como post inicial da série Mão na Massa: Padrões de Projeto vamos falar sobre o padrão Strategy. O objetivo desta série de posts é apresentar problemas "reais" de utilização do padrão em discussão, assim vamos começar apresentando o problema a ser discutido.

Problema

Suponha uma empresa, nesta empresa existem um conjunto de cargos, para cada cargo existem regras de cálculo de imposto, determinada porcentagem do salário deve ser retirada de acordo com o salário base do funcionário. Vamos as regras:

- O Desenvolvedor deve ter um imposto de 15% caso seu salário seja maior que R$ 2000,00 e 10% caso contrário;
- O Gerente deve ter um imposto de 20% caso seu salário seja maior que R$ 3500,00 e 15% caso contrário;
- O DBA deve ter um imposto de de 15% caso seu salário seja maior que R$ 2000,00 e 10% caso contrário;

Uma solução?

Até ai tudo bem, uma solução bem simples seria criar uma classe para representar um funcionário e dentro dele um campo para guardar seu cargo e salário. No método de cálculo de imposto utilizaríamos um switch para selecionar o cargo e depois verificaríamos o salário, para saber qual a porcentagem de imposto que deve ser utilizada. Vamos dar uma olhada no método que calcula o salário do funcionário aplicando o imposto:

```
public double calcularSalarioComImposto() {
        switch (cargo) {
        case DESENVOLVEDOR:
            if (salarioBase >= 2000) {
                return salarioBase * 0.85;
            } else {
                return salarioBase * 0.9;
            }
        case GERENTE:
            if (salarioBase >= 3500) {
                return salarioBase * 0.8;
            } else {
                return salarioBase * 0.85;
            }
        case DBA:
            if (salarioBase >= 2000) {
                return salarioBase * 0.85;
            } else {
                return salarioBase * 0.9;
            }
        default:
            throw new RuntimeException("Cargo não encontrado :/");
        }
    }
```

Este método é uma ótima prova do porque existem tantas vagas para desenvolvimento de software por ai 😀
Pense na seguinte situação: Surgiu um novo cargo que precisa ser cadastro, este cargo deve utilizar as mesmas regras de negócio do cargo DBA. O que seria necessário para incluir esta nova funcionalidade? Um novo case com novos if e else. Fácil não?
Imagine agora que depois de todo o trabalho para inserir todos os possíveis cargos de uma empresa, e uma regra muda? Seria awesome dar manutenção neste código não?

Padrão Strategy

O Padrão Strategy tem como Intenção:

“Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam” [1]

Ou seja, o padrão sugere que algoritmos parecidos (métodos de cálculo de Imposto) sejam separados de que os utiliza (Funcionário).

Certo, e como fazer isso? Bom, a primeira parte é encapsular todos os algoritmos da mesma família. No nosso exemplo a família de algoritmos é a que calcula salários com impostos, então para encapsulá-las criamos uma classe interface (Java) ou abstrata pura (C++). Vamos lá então:

```
interface CalculaImposto {
    double calculaSalarioComImposto(Funcionario umFuncionario);
}
```

Uma vez definida a classe que encapsula os algoritmos vamos definir as estratégias concretas de cálculo de imposto, a seguir o código para calculo de imposto de 15% ou 10%:

```
public class CalculoImpostoQuinzeOuDez implements CalculaImposto {
    @Override
    public double calculaSalarioComImposto(Funcionario umFuncionario) {
        if (umFuncionario.getSalarioBase() > 2000) {
            return umFuncionario.getSalarioBase() * 0.85;
        }
        return umFuncionario.getSalarioBase() * 0.9;
    }
}
```

As outras estratégias seguem este mesmo padrão, então vamos partir agora para as alterações na classe Funcionário. Esta classe **depende** da classe CalculoImposto, ou seja, ela utiliza um objeto CalculoImposto. Mas, como eu vou utilizar um objeto interface? Simples, este objeto será instanciado em tempo de execução e, de acordo com o Cargo dele a estratégia de cálculo correta será utilizada. No construtor, de acordo com o cargo nós configuramos a estratégia de cálculo correta:

```
 public Funcionario(int cargo, double salarioBase) {
         this.salarioBase = salarioBase;
         switch (cargo) {
         case DESENVOLVEDOR:
             estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
             cargo = DESENVOLVEDOR;
             break;
         case DBA:
             estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
             cargo = DBA;
             break;
         case GERENTE:
             estrategiaDeCalculo = new CalculoImpostoVinteOuQuinze();
             cargo = GERENTE;
             break;
         default:
             throw new RuntimeException("Cargo não encontrado :/");
         }
      }
```

E agora a única coisa que precisamos fazer para calcular o salário com imposto é:

```
public double calcularSalarioComImposto() {
    return estrategiaDeCalculo.calculaSalarioComImposto(this);
}
```

Agora sim está simples. 😀

Um pouco de teoria:

O padrão Strategy, além de encapsular os algoritmos da mesma família também permite a reutilização do código. No exemplo a regra para cálculo do imposto do Desenvolvedor e do DBA são as mesmas, ou seja, não será necessário escrever código extra.

Outra vantagem é a facilidade para extensão das funcionalidades. Caso seja necessário incluir um novo cargo basta implementar sua estratégia de cálculo de imposto ou reutilizar outra. Nenhuma outra parte do código precisa ser alterada.

O livro Padrões de Projeto da série Use a Cabeça fala do padrão Strategy como se fossem comportamentos, ou seja, uma família de algoritmos que simulam determinado comportamento. Neste livro é dado o exemplo da classe genérica Duck que possui os comportamentos de Fly e Quack. Assim cada tipo de Duck utiliza um comportamento Fly e Quack próprio.

Esta é outra nomenclatura para o padrão. As classes de estratégia são chamadas de Comportamento e a classe que utiliza o comportamento é chamada de Contexto. Ou seja, para um determinado Contexto você pode aplicar um conjunto de comportamentos.

Problemas com o Strategy

Quando outra pessoa está utilizando seu código ela pode escolher qualquer comportamento para o contexto que ela deseja aplicar. Isso pode ser visto como um potencial problema, pois o usuário do seu código deve conhecer bem a diferença entre as estratégias para saber escolher qual se aplica melhor ao contexto dele.

Outro potencial problema é a comunicação entre a classe de Contexto e a classe de Comportamento. Suponha um conjunto de dados (contexto) e vários algoritmos de ordenação (comportamento), caso a passagem do conjunto de dados para o algoritmo não seja eficiente a execução do algoritmo vai acabar sendo prejudicada. Da mesma forma, o contexto pode desperdiçar tempo passando dados para um contexto que não precisa deles. Ou seja, vale gastar algum tempo pensando bem em como a comunicação Contexto-Comportamento será feita.

Como nem tudo é totalmente ruim e nem totalmente bom, o que a nossa primeira solução tem de melhor em relação a solução que utiliza o padrão Strategy? Todo o cálculo é feito na classe funcionário, ou seja, apenas um objeto funcionário seria necessário para realizar todas as operações! Essa também é uma das desvantagens do padrão Strategy, precisamos incluir outro objeto que fica responsável com calcular o salário com imposto. No exemplo dos patos, para cada comportamento precisamos criar um objeto diferente. Ou seja, utilizar o padrão Strategy aumenta o número de objetos no programa.

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto (https://github.com/MarcosX/Padr-es-de-Projeto).

Os arquivos estão como um projeto do eclipse, então basta colocar no seu Workspace e fazer o import.

Se gostou do post compartilhe com seus amigos e colegas, senão, comente o que pode ser melhorado. Encontrou algum erro no código? Comente também. Possui alguma outra opinião ou alguma informação adicional? Comenta ai! 😀

Referências:

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

Padrões de Projeto, Strategy Java, Padrões, Projeto, Strategy 8 Comentários

8 comentários sobre “Mão na massa: Strategy”

1. junho 3, 2014 às 8:33 AM
   Anônimo
   Olá Marcos, como ficaria o diagrama ?

   Responder
   junho 3, 2014 às 9:49 AM
   Marcos Brizeno
   Opa, foi mal não ter adicionado a imagem. Pode olhar aqui e dar uma conferida em como seria, o exemplo é diferente mas a ideia é a mesma
   http://sourcemaking.com/design_patterns/strategy

   Responder
2. julho 24, 2014 às 3:11 PM
   Jhonathan Maia
   Marcos, boa tarde.

   Muito boa a explicação e o exemplo, a parte teórica deu pra entender porem quando mudo o cenário da prática complica um pouco. Pergunto o padrão Strategy se adequa a implementação Pessoa, Pessoa Fisica, Pessoa Juridica e Cliente que pode ser juridica ou fisica? Segundo, você poderia criar um exemplo disto na prática?

   Meu e-mail: jhol360@hotmail.com

   Agradeceria muito.

   Abraço.

   Responder
   julho 25, 2014 às 6:26 AM
   Marcos Brizeno
   Olá Jhonatha, tudo depende da necessidade do seu design. Uma dia é se, em algum ponto do código, você precisar ficar fazendo vários ifs pra saber se é uma Pessoa, Pessoa Física, etc. acho que cabe sim usar o Strategy.
   A ideia do Strategy é evitar ter uma classe que sabe tudo sobre todos os objetos, e deixar as responsabilidades distribuídas.

   Responder
3. setembro 21, 2015 às 12:34 PM
   Anônimo
   Olá Marcos, parabéns pela explicação! Só uma dúvida, seria interessante a aplicação também do factory method nesse projeto para a criação do funcionário? Agradeço desde já ☺

   Responder

setembro 24, 2015 às 1:18 PM

Marcos Brizeno

Olá! Sim, combinar padrões é bastante comum. Só tenha cuidado de pensar se é realmente necessário aplicar o padrão, pois ao utilizá-los você gera uma carga extra grande. Cabe avaliar se essa carga extra vale a pena ou não ☺

Responder

4. novembro 19, 2015 às 8:55 PM

Ricardo Assis

Olá Marcos, está faltando parte do código, não? O metodo Funcionario que deveria ser uma classe ou um construtor cita salarioBase como um atributo que não existe e acima cima o metodo getSalarioBase que não existe.

Responder

novembro 20, 2015 às 6:01 AM

Marcos Brizeno

Olá Ricardo, você pode conferir o código completo nesse link: https://github.com/MarcosX/Padr-es-de-Projeto

Responder

__PRESENT