

Marcos Brizeno

Desenvolvimento de Software #showmethecode

novembro 26, 2011

Mão na massa: Visitor

Problema:

É comum em projetos, onde você precisa criar sua própria estrutura de dados, que sejam necessários implementar várias operações sobre este conjunto de dados. Geralmente, esta responsabilidade é delegada para a própria classe que representa a estrutura.

Para discutirmos melhor, suponha que em um projeto você precisa lidar com uma estrutura de dados muito complexa, uma árvore binária por exemplo (tá, não é tão complexo mas serve pro exemplo). É comum que sejam implementados métodos para percorrer a árvore, como por exemplo: in-order, pre-order e post-order. [2]

O problema em implementar estes métodos diretamente na classe que representa a árvore é que é preciso ter um alto nível de certeza de que estas operações não mudarão, ou que serão utilizadas apenas nesta estrutura, pois o custo para realizar alterações seria muito grande, uma vez que o sistema dependerá da definição desta classe.

Outro problema é que, várias outras estruturas de dados vão utilizar uma abordagem semelhante. Por exemplo, caso seja implementado uma árvore AVL, que utiliza os mesmos dados e as mesmas operações de percurso, não seria possível reutilizar este método (reutilizar NÃO É ctrl+c ctrl+v!).

Vejamos então um pouco sobre o padrão Visitor, que vai nos ajudar bastante a resolver este problema.

Visitor

Como de costume, vamos a intenção do padrão:

“Representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.” [1]

Pela intenção já é possível ver como o padrão vai nos ajudar. A sua ideia é separar as operações que serão executadas em determinada estrutura de sua representação. Assim, incluir ou remover operações não terá nenhum efeito sobre a interface da estrutura, permitindo que o resto do sistema funcione sem depender de operações específicas.

Vamos então começar definindo a estrutura de dados a ser utilizada. A seguinte classe representa o nó da árvore:

```

1  public class No {
2      protected int chave;
3      No esquerdo, direito;
4
5      public No(int chave) {
6          this.chave = chave;
7          esquerdo = null;
8          direito = null;
9      }
10
11     public String toString() {
12         return String.valueOf(chave);
13     }
14
15     public int getChave() {
16         return chave;
17     }
18
19     public No getEsquerdo() {
20         return esquerdo;
21     }
22
23     public void setEsquerdo(No esquerdo) {
24         this.esquerdo = esquerdo;
25     }
26
27     public No getDireito() {
28         return direito;
29     }
30
31     public void setDireito(No direito) {
32         this.direito = direito;
33     }
34
35 }

```

Definimos então o nó básico da árvore, que contém a chave e os nós esquerdo e direito. Além disso o método “toString()” permite que a estrutura seja exibida na tela com o sysout. Agora vamos ver a estrutura árvore, que vai manter todos estes elementos.

```

1  public class ArvoreBinaria {
2      No raiz;
3      int quantidadeDeElementos;
4
5      public ArvoreBinaria(int chaveRaiz) {
6          raiz = new No(chaveRaiz);
7          quantidadeDeElementos = 0;
8      }
9
10     public void inserir(int chave) {
11         ...
12     }
13
14     public void remover(int chave){
15         ...
16     }
17
18     public void buscar(int chave){
19         ...
20     }
21
22     public void aceitarVisitante(ArvoreVisitor visitor) {
23         visitor.visitar(raiz);
24     }
25 }

```

A estrutura árvore vai então possuir o nó raiz da árvore e algumas outras informações sobre a árvore. Omiti a implementação dos métodos da árvore para simplificar o exemplo. O detalhe importante desta classe é o método “aceitarVisitante()”, ele recebe um objeto ArvoreVisitor e passa a sua raiz para ele. Ai começa a implementação do padrão de verdade.

A estrutura de dados vai possuir um método que recebe um objeto visitante. Deste método ela vai chamar o método visitar, do objeto visitante, e vai passar os seus dados para o objeto visitante. Dai em diante, o objeto visitante vai poder realizar as operações necessárias. Vamos então começar com a implementação de ArvoreVisitor:

```

1  public interface ArvoreVisitor {
2
3      void visitar(No no);
4
5  }

```

Esta classe vai apenas definir a interface de visita de um nó. Todas as operações vão receber um objeto No e a partir daí vão implementar suas operações. Por exemplo, vejamos a implementação de um método de percurso in-order.

```
1 public class ExibirInOrderVisitor implements ArvoreVisitor {
2
3     @Override
4     public void visitar(No no) {
5         if (no == null)
6             return;
7         this.visitar(no.getEsquerdo());
8         System.out.println(no);
9         this.visitar(no.getDireito());
10    }
11
12 }
```

Seguindo a ideia de implementação de percurso in-order [2], primeiro é feita a visita ao nó esquerdo, em seguida mostramos no terminal o nó e ao fim é feita a visita ao nó direito. Com esta simples implementação temos o método de percurso in-order da árvore, sem precisar alterar a sua estrutura.

Implementar várias outras operações fica muito simples ao utilizar este padrão. As implementações dos outros métodos de percurso ficam triviais. Uma implementação mais complexo, também fica mais simplificada. Por exemplo, caso seja necessário implementar um método que exibe os nós de uma maneira indentada, de acordo com seu nível na árvore.

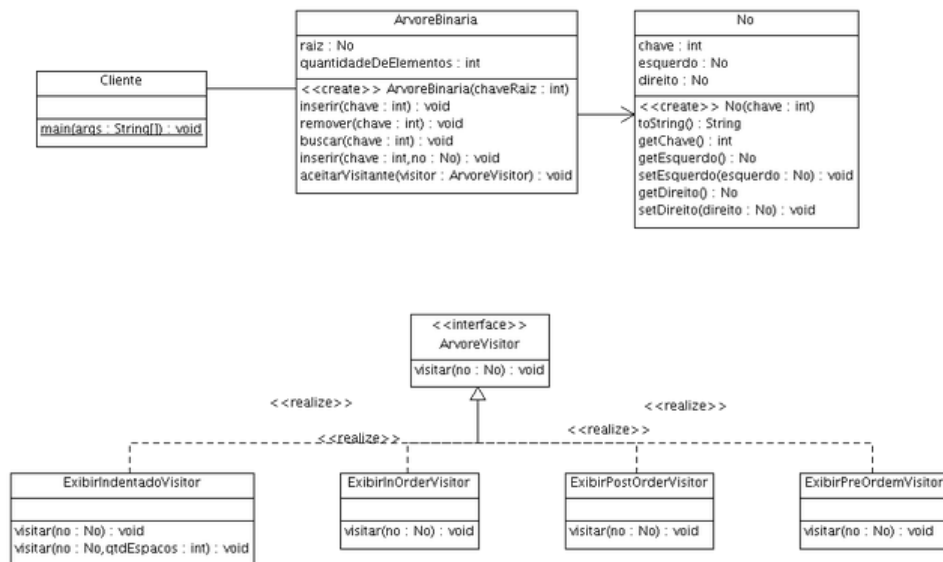
```
1 public class ExibirIndentadoVisitor implements ArvoreVisitor {
2
3     @Override
4     public void visitar(No no) {
5         if (no == null) {
6             return;
7         }
8         System.out.println(no);
9         visitar(no.getEsquerdo(), 1);
10        visitar(no.getDireito(), 1);
11    }
12
13    private void visitar(No no, int qtdEspacos) {
14        if (no == null) {
15            return;
16        }
17        for (int i = 0; i < qtdEspacos; i++) {
18            System.out.print("-");
19        }
20        System.out.println(no);
21        visitar(no.getEsquerdo(), qtdEspacos + 1);
22        visitar(no.getDireito(), qtdEspacos + 1);
23    }
24
25 }
```

Este método conta a quantidade de espaços para indentação a partir de cada nível de recursão do método. A única restrição das classes visitantes é que implementem o método para visitar. Quaisquer outras operações que precisem de suporte podem ser implementadas.

Um exemplo de cliente seria:

```
1 public static void main(String[] args) {
2     ArvoreBinaria arvore = new ArvoreBinaria(7);
3
4     arvore.inserir(15);
5     arvore.inserir(10);
6     arvore.inserir(5);
7     arvore.inserir(2);
8     arvore.inserir(1);
9     arvore.inserir(20);
10
11    System.out.println("### Exibindo em ordem ###");
12    arvore.aceitarVisitante(new ExibirInOrderVisitor());
13    System.out.println("### Exibindo pre ordem ###");
14    arvore.aceitarVisitante(new ExibirPreOrdemVisitor());
15    System.out.println("### Exibindo pós ordem ###");
16    arvore.aceitarVisitante(new ExibirPostOrderVisitor());
17    System.out.println("### Exibindo indentado ###");
18    arvore.aceitarVisitante(new ExibirIndentadoVisitor());
19 }
```

O diagrama UML que representa esta solução é o seguinte:



(<https://brizeno.files.wordpress.com/2011/11/visitor.png>).

Um pouco de teoria

Como foi exemplificado, o padrão Visitor oferece uma excelente alternativa quando é necessário realizar uma série de operações sobre um conjunto de dados, dado que estas operações são pouco estáveis, ou seja, sofrem alterações constantemente. Um outro exemplo que evidencia mais a aplicabilidade do padrão é na construção de compiladores, como mostrado em [1], onde existem vários tipos de nós da árvore de sintaxe abstrata, e o conjunto de operações é indefinido.

Assim, para decidir pela utilização do padrão Visitor é necessário ter certeza de que a estrutura dos elementos seja bem estável (não sofra alterações ao longo do projeto) e que a interface desta estrutura permita acesso suficiente para os objetos visitantes. Elementos podem ter interfaces diferentes, contanto que estas interface sejam estáveis e provejam acesso às classes visitantes.

Outro detalhes importante é que a estrutura de dados depende da interface das classes visitantes. Já a classe visitante precisa ter conhecimento sobre os vários tipo de elementos, pois cada um deles poderá ser visitado de uma maneira diferente. Para exemplificar esta afirmação, suponha o seguinte cenário:

São utilizadas várias estruturas de dados, como vetores, listas, árvores binárias, árvores B, heaps, etc. Todas estas classes oferecem uma interface simples: adicionar, remover e buscar elementos. Já um método visitante atuaria de maneiras diferentes em cada uma delas. O método de percurso de uma árvore binária é diferente do método de percurso de uma lista, ou de uma árvore B. Assim, na classe visitante, é preciso ter um método para visitar cada um destes tipos de estruturas, embora todas elas tenham uma interface em comum.

Se por um lado o padrão facilita a adição de novas operações sobre o conjunto de estruturas, fica muito difícil, uma vez que o projeto está utilizando o padrão, incluir novos tipos de elementos, pois cada novo elemento vai gerar alterações em todas as classes visitantes.

Padrões que utilizem estruturas de dados para representação podem utilizar o padrão Visitor, por exemplo, uma estrutura Composite (<http://wp.me/p1Mek8-M>) ou Interpreter (<http://wp.me/p1Mek8-4o>), pode oferecer um método de visita e atribuir para as classes visitantes a responsabilidade das operações sobre seu conjunto de dados.

Já que o padrão Visitor é utilizado para percorrer um conjunto de dados, qual a diferença dele e do padrão Iterator (<http://wp.me/p1Mek8-15>)? A intenção do padrão Iterator é fornecer uma maneira de percorrer os elementos de uma estrutura, expondo seus elementos e deixando para o cliente a responsabilidade de operar sobre estes. O padrão Visitor oferece operações sobre elementos, sem expor seu conteúdo, ou delegar responsabilidades extras para o cliente.

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 aí! 😊

Referências:

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

[2] WIKIPEDIA. Tree Traversal: http://en.wikipedia.org/wiki/Tree_traversal (http://en.wikipedia.org/wiki/Tree_traversal). Acesso em 24 de novembro de 2011.

☐ [Visitor](#) ☐ [Java, Padrões, Projeto, Visitor](#) ☐ [2 Comentários](#)

2 comentários sobre “Mão na massa: Visitor”

1. ☐ [janeiro 11, 2013 às 5:36 PM](#)

ELIAS

ÓTIMO OS SEUS CÓDIGOS. TAVA PRECISANDO MUITO. ME AJUDOU BASTANTE. QUE DEUS TE ABENÇOE.

☐ [Responder](#)

2. ☐ [maio 12, 2017 às 3:33 PM](#)

Anônimo

Estou achando errado ou parece um pouco com o Strategy?

☐ [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

setembro 18, 2011

Mão na massa: Template Method

Vamos mostrar agora o padrão Template Method!

Problema

Suponha um player de música que oferece várias maneiras de reproduzir as músicas de uma playlist. Para exemplificar suponha que podemos reproduzir a lista de músicas da seguinte maneira:

- Ordenado por nome da música
- Ordenado por nome do Autor
- Ordenado por ano
- Ordenado por estrela (preferência do usuário)

Uma ideia seria utilizar o padrão [Strategy](http://wp.me/s1Mek8-strategy) (<http://wp.me/s1Mek8-strategy>) e implementar uma classe que define o método de reprodução para cada tipo de reprodução da playlist. Esta seria uma solução viável, pois manteríamos a flexibilidade para implementar novos modos de reprodução de maneira bem simples.

No entanto, observe que, o algoritmo para reprodução de uma playlist é o mesmo, independente de qual modo esta sendo utilizado. A única diferença é a criação da playlist, que leva em consideração um dos atributos da música.

Para suprir esta dificuldade vamos ver o padrão Template Method!

Template Method

Como de costume, vejamos a intenção do padrão Template Method:

“Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passo de um algoritmo sem mudar a estrutura do mesmo.” [1]

Perfeito para o nosso problema! Precisamos definir o método de ordenação da Playlist mas só saberemos qual atributo utilizar em tempo de execução. Vamos definir então a estrutura de dados que define uma música:

```
1 public class MusicaMP3 {
2     String nome;
3     String autor;
4     String ano;
5     int estrelas;
6
7     public MusicaMP3(String nome, String autor, String ano, int estrela) {
8         this.nome = nome;
9         this.autor = autor;
10        this.ano = ano;
11        this.estrelas = estrela;
12    }
13 }
```

Para escolher como a playlist deve ser ordenada vamos criar uma pequena enumeração:

```

1 public enum ModoDeReproducao {
2     porNome, porAutor, porAno, porEstrela
3 }

```

Agora vamos escrever a nossa classe que implementa o método template para ordenação da lista:

```

1 public abstract class OrdenadorTemplate {
2     public abstract boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2);
3
4     public ArrayList<MusicaMP3> ordenarMusica(ArrayList<MusicaMP3> lista) {
5         ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();
6         for (MusicaMP3 musicaMP3 : lista) {
7             novaLista.add(musicaMP3);
8         }
9
10        for (int i = 0; i < novaLista.size(); i++) {
11            for (int j = i; j < novaLista.size(); j++) {
12                if (!isPrimeiro(novaLista.get(i), novaLista.get(j))) {
13                    MusicaMP3 temp = novaLista.get(j);
14                    novaLista.set(j, novaLista.get(i));
15                    novaLista.set(i, temp);
16                }
17            }
18        }
19
20        return novaLista;
21    }
22 }

```

Basicamente definimos um método de ordenação, no caso o método Bolha, e deixamos a comparação dos atributos para as subclasses. Essa é a ideia de utilizar o método template, definir o esqueleto e permitir a personalização dele nas subclasses. Veja o exemplo da classe a seguir que ordena as músicas por nome:

```

1 public class OrdenadorPorNome extends OrdenadorTemplate {
2
3     @Override
4     public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {
5         if (musica1.nome.compareToIgnoreCase(musica2.nome) <= 0) {
6             return true;
7         }
8         return false;
9     }
10 }
11 }

```

Para implementar as outras formas de reprodução da lista basta definir, na subclasse, o método que compara uma música com outra e diz se é necessário trocar.

O exemplo a seguir define a ordenação baseado no nível de preferência do usuário (aquelas estrelinhas dos players de música)

```

1 public class OrdenadorPorEstrela extends OrdenadorTemplate {
2
3     @Override
4     public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {
5         if (musica1.estrelas > musica2.estrelas) {
6             return true;
7         }
8         return false;
9     }
10 }
11 }

```

Pronto, definimos o algoritmo padrão e suas variações. Agora vamos ver a classe que manipula a playlist:

```

1  public class Playlist {
2      protected ArrayList<MusicaMP3> musicas;
3      protected OrdenadorTemplate ordenador;
4
5      public Playlist(ModoDeReproducao modo) {
6          musicas = new ArrayList<MusicaMP3>();
7          switch (modo) {
8              case porAno:
9                  ordenador = new OrdenadorPorAno();
10                 break;
11             case porAutor:
12                 ordenador = new OrdenadorPorAutor();
13                 break;
14             case porEstrela:
15                 ordenador = new OrdenadorPorEstrela();
16                 break;
17             case porNome:
18                 ordenador = new OrdenadorPorNome();
19                 break;
20             default:
21                 break;
22         }
23     }
24
25     public void setModoDeReproducao(ModoDeReproducao modo) {
26         ordenador = null;
27         switch (modo) {
28             case porAno:
29                 ordenador = new OrdenadorPorAno();
30                 break;
31             case porAutor:
32                 ordenador = new OrdenadorPorAutor();
33                 break;
34             case porEstrela:
35                 ordenador = new OrdenadorPorEstrela();
36                 break;
37             case porNome:
38                 ordenador = new OrdenadorPorNome();
39                 break;
40             default:
41                 break;
42         }
43     }
44
45     public void adicionarMusica(String nome, String autor, String ano,
46         int estrela) {
47         musicas.add(new MusicaMP3(nome, autor, ano, estrela));
48     }
49
50     public void mostrarListaDeReproducao() {
51         ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();
52         novaLista = ordenador.ordenarMusica(musicas);
53
54         for (MusicaMP3 musica : novaLista) {
55             System.out.println(musica.nome + " - " + musica.autor + "\n Ano: "
56                 + musica.ano + "\n Estrelas: " + musica.estrelas);
57         }
58     }
59 }

```

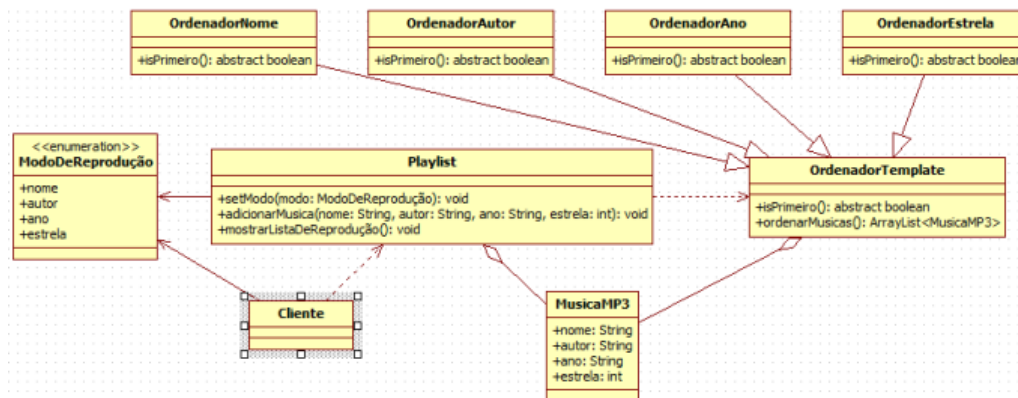
Definimos métodos para inserir músicas e exibir a playlist e, de acordo com o parâmetro passado, criamos uma playlist. O código cliente ficaria da seguinte maneira:


```

1 public static void main(String[] args) {
2
3     Playlist minhaPlaylist = new Playlist(ModoDeReproducao.porNome);
4     minhaPlaylist.adicionarMusica("Everlong", "Foo Fighters", "1997", 5);
5     minhaPlaylist.adicionarMusica("Song 2", "Blur", "1997", 4);
6     minhaPlaylist.adicionarMusica("American Jesus", "Bad Religion", "1993",
7     3);
8     minhaPlaylist.adicionarMusica("No Cigar", "Milencollin", "2001", 2);
9     minhaPlaylist.adicionarMusica("Ten", "Pearl Jam", "1991", 1);
10
11     System.out.println("=== Lista por Nome de Musica ===");
12     minhaPlaylist.mostrarListaDeReproducao();
13
14     System.out.println("\n=== Lista por Autor ===");
15     minhaPlaylist.setModoDeReproducao(ModoDeReproducao.porAutor);
16     minhaPlaylist.mostrarListaDeReproducao();
17
18     System.out.println("\n=== Lista por Ano ===");
19     minhaPlaylist.setModoDeReproducao(ModoDeReproducao.porAno);
20     minhaPlaylist.mostrarListaDeReproducao();
21
22     System.out.println("\n=== Lista por Estrela ===");
23     minhaPlaylist.setModoDeReproducao(ModoDeReproducao.porEstrela);
24     minhaPlaylist.mostrarListaDeReproducao();
25 }

```

Veja o quão simples foi alterar o modo como a lista é construída. No final, essa é a estrutura do projeto utilizando o Template Method:



(<https://brizenofiles.wordpress.com/2011/09/template-method.png>).

Um pouco de teoria:

Já exemplificamos a principal vantagem do padrão Template Method, a facilidade de alteração do algoritmo principal. No entanto, deve-se tomar cuidado ao utilizar o padrão pois, se for preciso definir muitas operações nas subclasses, talvez seja necessário refatorar o código ou repensar o design.

Outro problema é que, ao definir o método que executa o algoritmo genérico, não é possível proteger este método das subclasses. Ou seja, o cliente do código precisa saber exatamente quais operações substituir para alcançar o efeito desejado. Por exemplo, caso o programador da subclasse OrdenadorPorEstrela redefinissem o método de ordenação para realizar qualquer outra operação, poderíamos ter problemas.

Por isso é tão importante definir os métodos que devem ser sobrescritos como abstratos (abstract em java, ou virtual puro em C++). Dessa maneira garante-se o princípio Open/Closed [2], que diz que uma classe deve ser aberta para extensões (fácil criar novas maneiras de reproduzir músicas) e fechada para alterações.

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 aí! 😊

Referências:

- [1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.
[2] WIKIPEDIA. SOLID. Disponível em: [http://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)).
([http://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design))). Acesso em: 15 set. 2011.

[□ Padrões de Projeto, Template Method](#) [□ Java, Padrões, Projeto, Template Method](#) [□ 5 Comentários](#)

5 comentários sobre “Mão na massa: Template Method”

1. [□ agosto 13, 2013 às 10:18 PM](#)

Angelitah [↗](#)

Muito boas as suas publicações, estão me ajudando muito a entender muitas coisas na faculdade. Muito obrigada! Outra coisa: muito bom seu gosto musical 😊

[□ Responder](#)

[□ agosto 13, 2013 às 10:19 PM](#)

Angelitah [↗](#)

ops, entender*

[□ Responder](#)

2. [□ outubro 28, 2014 às 10:24 AM](#)

vagnerismello [↗](#)

Informações ricas e consistentes. Excelente texto!
Os exemplos são perfeitos! Facilitam a compreensão.
Parabéns!!!

[□ Responder](#)

3. [□ maio 11, 2015 às 10:33 AM](#)

Anônimo

Parabéns pelo conteúdo sobre Design Pattern!
Seus exemplos me auxiliam bastante quando vou ministrar aulas sobre este assunto.

[□ Responder](#)

4. [□ novembro 22, 2017 às 1:09 PM](#)

Felipe Stein Rosa [↗](#)

O site está me ajudando muito e pelo exemplo das musicas do Template Method nós temos gostos musicais em comum rrsrs

[□ Responder](#)

Marcos Brizen

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 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:

```
1 public double calcularSalarioComImposto() {
2     switch (cargo) {
3         case DESENVOLVEDOR:
4             if (salarioBase >= 2000) {
5                 return salarioBase * 0.85;
6             } else {
7                 return salarioBase * 0.9;
8             }
9         case GERENTE:
10            if (salarioBase >= 3500) {
11                return salarioBase * 0.8;
12            } else {
13                return salarioBase * 0.85;
14            }
15        case DBA:
16            if (salarioBase >= 2000) {
17                return salarioBase * 0.85;
18            } else {
19                return salarioBase * 0.9;
20            }
21        default:
22            throw new RuntimeException("Cargo não encontrado :/");
23    }
24 }
```

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 quem 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:

```
1 interface CalculaImposto {
2     double calculaSalarioComImposto(Funcionario umFuncionario);
3 }
```

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

```
1 public class CalculoImpostoQuinzeOuDez implements CalculaImposto {
2     @Override
3     public double calculaSalarioComImposto(Funcionario umFuncionario) {
4         if (umFuncionario.getSalarioBase() > 2000) {
5             return umFuncionario.getSalarioBase() * 0.85;
6         }
7         return umFuncionario.getSalarioBase() * 0.9;
8     }
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:

```
1 public Funcionario(int cargo, double salarioBase) {
2     this.salarioBase = salarioBase;
3     switch (cargo) {
4         case DESENVOLVEDOR:
5             estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
6             cargo = DESENVOLVEDOR;
7             break;
8         case DBA:
9             estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
10            cargo = DBA;
11            break;
12         case GERENTE:
13             estrategiaDeCalculo = new CalculoImpostoVinteOuQuinze();
14             cargo = GERENTE;
15             break;
16         default:
17             throw new RuntimeException("Cargo não encontrado :/");
18     }
19 }
```

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

```
1 public double calcularSalarioComImposto() {
2     return estrategiaDeCalculo.calculaSalarioComImposto(this);
3 }
```

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 aí! 😊

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](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

outubro 17, 2011

Mão na massa: Observer

Problema

Suponha que em um programa é necessário fazer várias representações de um mesmo conjunto de dados. Este conjunto de dados consiste de uma estrutura que contém 3 atributos: valorA, valorB e valorC, como mostra o código a seguir:

```
1 public class Dados {  
2     int valorA, valorB, valorC;  
3  
4     public Dados(int a, int b, int c) {  
5         valorA = a;  
6         valorB = b;  
7         valorC = c;  
8     }  
9 }
```

Como exemplo vamos considerar que é necessário representar dados em uma tabela, que simplesmente exibe os números, uma representação em gráficos de barras, onde os valores são exibidos em barras e outra representação em porcentagem, relativo a soma total dos valores.

A representação deve ser feita de modo que qualquer alteração no conjunto de dados compartilhados provoque alterações em todas as formas de representação, garantindo assim que uma visão nunca tenha dados invalidados.

Também queremos que as representações só sejam redesenhadas somente quando necessário. Ou seja, sempre que um valor for alterado.

Uma primeira solução poderia ser manter uma lista com as possíveis representações e ficar verificando por mudanças no conjunto de dados, assim que fosse feita uma mudança, as visualizações seriam avisadas.

O problema é que precisamos sempre verificar se houve ou não mudança no conjunto de dados, dessa forma o processamento seria muito caro, ou então a atualização seria demorada. Vamos ver então como o padrão Observer pode ajudar.

Observer

Intenção:

“Definir uma dependência um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.” [1]

O padrão Observer parece ser uma boa solução para o problema, pois ele define uma dependência um para muitos, que será necessária para fazer a relação entre um conjunto de dados e várias representações, além de permitir que, quando um objeto mude de estado, todos os dependentes sejam notificados.

Para garantir isto o padrão faz o seguinte: cria uma classe que mantém o conjunto de dados e uma lista de dependentes deste conjunto de dados, assim a cada mudança no conjunto de dados todos os dependentes são notificados. Vejamos então o código desta classe por partes:

```

1 public class DadosSubject {
2
3     protected ArrayList<DadosObserver> observers;
4     protected Dados dados;
5
6     public DadosSubject() {
7         observers = new ArrayList<DadosObserver>();
8     }
9
10    public void attach(DadosObserver observer) {
11        observers.add(observer);
12    }
13
14    public void detach(int indice) {
15        observers.remove(indice);
16    }
17 }

```

Inicialmente definimos a lista de observadores (DadosObserver é uma interface comum aos observadores e será definida a seguir) e o conjunto de dados a ser compartilhado. Também definimos os métodos para adicionar e remover observadores, assim cada novo observador poderá facilmente acompanhar as mudanças.

Dentro da mesma classe, vamos definir as mudanças no estado, ou seja o conjunto de dados:

```

1 public void setState(Dados dados) {
2     this.dados = dados;
3     notifyObservers();
4 }
5
6 private void notifyObservers() {
7     for (DadosObserver observer : observers) {
8         observer.update();
9     }
10 }
11
12 public Dados getState() {
13     return dados;
14 }

```

Sempre que for feita uma mudança no conjunto de dados, utilizando o método “setState()” é chamado o método que vai notificar todos os observadores, executando um update para informar que o conjunto de dados mudou.

Vamos ver então como seria um observador. Vamos definir então a interface comum a todos os observadores, que é utilizada para manter a lista de observadores na classe que controla o conjunto de dados:

```

1 public abstract class DadosObserver {
2
3     protected DadosSubject dados;
4
5     public DadosObserver(DadosSubject dados) {
6         this.dados = dados;
7     }
8
9     public abstract void update();
10 }

```

Definida a interface vamos então construir o observador que mostra os dados em uma tabela:

```

1 public class TabelaObserver extends DadosObserver {
2
3     public TabelaObserver(DadosSubject dados) {
4         super(dados);
5     }
6
7     @Override
8     public void update() {
9         System.out.println("Tabela:\nValor A: " + dados.getState().valorA
10             + "\nValor B: " + dados.getState().valorB + "\nValor C: "
11             + dados.getState().valorC);
12     }
13 }

```

Este observador simplesmente exibe o valor dos dados. Assim, quando o método update for chamado ele irá redesenhar a tabela de dados. Para o exemplo apenas vamos exibir algumas informações no terminal.

Outros observers podem definir outras maneiras de mostrar o conjunto de dados, por exemplo o observer que exibe os valores em porcentagem:

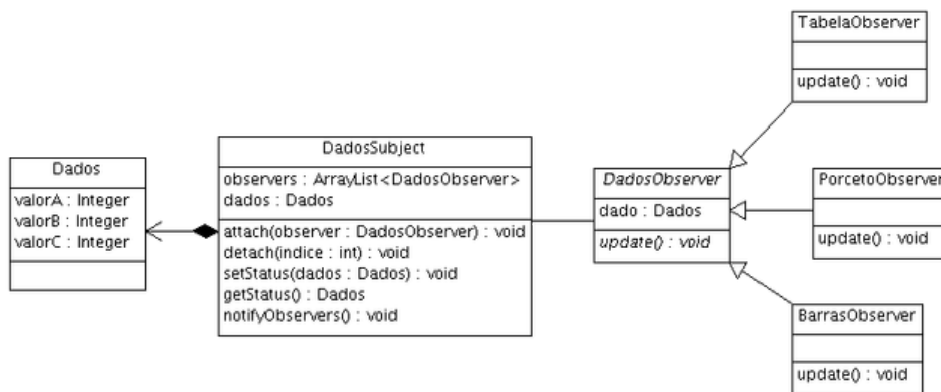

```

1 public class PorcentoObserver extends DadosObserver {
2
3     public PorcentoObserver(DadosSubject dados) {
4         super(dados);
5     }
6
7     @Override
8     public void update() {
9         int somaDosValores = dados.getState().valorA + dados.getState().valorB
10            + dados.getState().valorC;
11         DecimalFormat formatador = new DecimalFormat("#.##");
12         String porcentagemA = formatador.format((double) dados.getState().valorA
13            / somaDosValores);
14         String porcentagemB = formatador.format((double) dados.getState().valorB
15            / somaDosValores);
16         String porcentagemC = formatador.format((double) dados.getState().valorC
17            / somaDosValores);
18         System.out.println("Porcentagem:\nValor A: " + porcentagemA
19            + "\nValor B: " + porcentagemB + "\nValor C: " + porcentagemC
20            + "%");
21     }
22 }
23

```

Para este observer é feito inicialmente o cálculo da soma dos valores e depois é calculado cada valor em relação a este total, exibindo o resultado com duas casas decimais.

A representação UML desta solução é a seguinte:



(<https://brizenofiles.wordpress.com/2011/10/classdiagram1.png>).

Um pouco de teoria

Na nomenclatura do padrão Observer temos as duas classes principais: Subject e Observer. O Subject é o que mantém os dados compartilhados e a lista de observadores que compartilham o dado. O Observer é o que faz utilização dos dados compartilhados e deve ser atualizado a cada modificação.

Como vimos no nosso exemplo o padrão Observer oferece uma excelente maneira de compartilhar um recurso, utilizando uma técnica parecida com o *broadcast*, onde todos os observadores cadastrados em um subject são notificados sobre mudanças.

Ao realizar uma mudança é necessário ter cuidado, pois não se sabe exatamente os efeitos desta mudança nos seus observers ou o custo das atualizações nos observers. Caso as mudanças sejam muito complexas ou muito custosas pode ser interessante implementar uma estrutura intermediária para gerenciar os subjects e observers e suas mudanças.

Outro motivo para se utilizar uma estrutura intermediária entre subject e observer é quando existem muitos subjects e muitos observers interligados. Uma estrutura para mapear subjects e observers pode ser mais eficiente que uma lista de observers em cada subject.

Essa estrutura intermediária muitas vezes pode ser uma instância do padrão Mediator, que vamos abordar no próximo post da série. Também é interessante que esta classe intermediária seja uma instância do padrão Singleton (<http://wp.me/p1Mek8-1Z>), pois é interessante que apenas um objeto centralize o controle de subjects e observers.

Note também que no exemplo acima cometemos um pequeno “erro”, pois não definimos a classe Subject como uma interface, isso dificultaria bastante alterações nesta classe. Geralmente a interface de Subject define apenas os métodos de adição e remoção de Observers e o método de notificação.

As subclasses de Subject vão definir como será a inserção e remoção de observers, e como estes observers serão notificados. Além disso ela deve prover uma maneira dos observers acessarem os dados compartilhados, definindo assim os métodos de “getState()” e “setState()”

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.

□ [Observer, Padrões de Projeto](#) □ [Java, Observer, Padrões, Projeto](#) □ [4 Comentários](#)

4 comentários sobre “Mão na massa: Observer”

1. □ [abril 6, 2015 às 8:32 AM](#)

Will

Olá, tenho uma dúvida. Como seria o funcionamento deste padrão no caso de uma aplicação com banco de dados, onde eu preciso saber quando alguma informação foi alterada em determinadas tabelas e por outro usuário. Nos exemplos que vejo, as interações são feitas sempre no mesmo executável.

□ [Responder](#)

□ [abril 7, 2015 às 8:35 AM](#)

Marcos Brizeno 

A ideia do padrão é ter um local central para notificar mudanças, para adaptar o padrão basta adicionar mais informações quando o mudanças acontecem.

□ [Responder](#)

2. □ [fevereiro 20, 2017 às 2:02 AM](#)

Paulo Roberto Lima da Silva

Olá Marcos

Cara seus blog é de utilidade pública, está me ajudando muito com os padrões...

Para fazer o padrão “observer” funcionar em um projeto web, como seria a atualização dos dados no navegador? Sem aquela opção de fazer o javascript atualizar a pagina de tempo em tempo.

Grande Abraço

Paulo Lima

□ [Responder](#)

□ [fevereiro 20, 2017 às 10:05 AM](#)

Marcos Brizeno 

Obrigado pelo comentário Paulo.

Para sincronizar os dados no navegador é preciso ir no servidor, seja atualizando a página ou via javascript. Você pode implementar seu próprio observer ou utilizar um framework que possua essas facilidades.

□ [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

novembro 21, 2011

Mão na massa: State

Problema:

A troca de estados de um objeto é um problema bastante comum. Tome como exemplo o personagem de um jogo, como o Mario. Durante o jogo acontecem várias trocas de estado com o Mario, por exemplo, ao pegar uma flor de fogo o mario pode crescer, se estiver pequeno, e ficar com a habilidade de soltar bolas de fogo.

Desenvolvendo um pouco mais o pensamento temos um conjunto grande de possíveis estados, e cada transição depende de qual é o estado atual do personagem. Como falado anteriormente, ao pegar uma flor de fogo podem acontecer quatro ações diferentes, dependendo de qual o estado atual do mario:

Se Mario pequeno -> Mario grande e Mario fogo
Se Mario grande -> Mario fogo
Se Mario fogo -> Mario ganha 1000 pontos
Se Mario capa -> Mario fogo

Todas estas condições devem ser checadas para realizar esta única troca de estado. Agora imagine o vários estados e a complexidade para realizar a troca destes estados: Mario pequeno, Mario grande, Mario flor e Mario pena.

Pegar Cogumelo:

Se Mario pequeno -> Mario grande
Se Mario grande -> 1000 pontos
Se Mario fogo -> 1000 pontos
Se Mario capa -> 1000 pontos

Pegar Flor:

Se Mario pequeno -> Mario grande e Mario fogo
Se Mario grande -> Mario fogo
Se Mario fogo -> 1000 pontos
Se Mario capa -> Mario fogo

Pegar Pena:

Se Mario pequeno -> Mario grande e Mario capa
Se Mario grande -> Mario capa
Se Mario fogo -> Mario fogo
Se Mario capa -> 1000 pontos

Levar Dano:

Se Mario pequeno -> Mario morto
Se Mario grande -> Mario pequeno
Se Mario fogo -> Mario grande
Se Mario capa -> Mario grande

Com certeza não vale a pena investir tempo e código numa solução que utilize várias verificações para cada troca de estado. Para não correr o risco de esquecer de tratar algum estado e deixar o código bem mais fácil de manter, vamos analisar como o padrão State pode ajudar.

State

A intenção do padrão:

“Permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.” [1]

Pela intenção podemos ver que o padrão vai alterar o comportamento de um objeto quando houver alguma mudança no seu estado interno, como se ele tivesse mudado de classe.

Para implementar o padrão será necessário criar uma classe que contém a interface básica de todos os estados. Como definimos anteriormente o que pode causar alteração nos estados do objeto Mario, estas serão as operações básicas que vão fazer parte da interface.

```
1 public interface MarioState {
2     MarioState pegarCogumelo();
3
4     MarioState pegarFlor();
5
6     MarioState pegarPena();
7
8     MarioState levarDano();
9 }
```

Agora todos os estados do mario deverão implementar as operações de troca de estado. Note que cada operação retorna um objeto do tipo MarioState, pois como cada operação representa uma troca de estados, será retornado qual o novo estado o mario deve assumir.

Vejamos então uma classe para exemplificar um estado:

```
1 public class MarioPequeno implements MarioState {
2
3     @Override
4     public MarioState pegarCogumelo() {
5         System.out.println("Mario grande");
6         return new MarioGrande();
7     }
8
9     @Override
10    public MarioState pegarFlor() {
11        System.out.println("Mario grande com fogo");
12        return new MarioFogo();
13    }
14
15    @Override
16    public MarioState pegarPena() {
17        System.out.println("Mario grande com capa");
18        return new MarioCapa();
19    }
20
21    @Override
22    public MarioState levarDano() {
23        System.out.println("Mario morto");
24        return new MarioMorto();
25    }
26
27 }
```

Percebemos que a classe que define o estado é bem simples, apenas precisa definir qual estado deve ser trocado quando uma operação de troca for chamada. Vejamos agora outro exemplo de classe de estado:

```

1  public class MarioCapa implements MarioState {
2
3      @Override
4      public MarioState pegarCogumelo() {
5          System.out.println("Mario ganhou 1000 pontos");
6          return this;
7      }
8
9      @Override
10     public MarioState pegarFlor() {
11         System.out.println("Mario com fogo");
12         return new MarioFogo();
13     }
14
15     @Override
16     public MarioState pegarPena() {
17         System.out.println("Mario ganhou 1000 pontos");
18         return this;
19     }
20
21     @Override
22     public MarioState levarDano() {
23         System.out.println("Mario grande");
24         return new MarioGrande();
25     }
26 }
27

```

Bem simples não? Novos estados são adicionados de maneira bem simples. Vejamos então como seria o objeto que vai utilizar o estados, o Mario:

```

1  public class Mario {
2      protected MarioState estado;
3
4      public Mario() {
5          estado = new MarioPequeno();
6      }
7
8      public void pegarCogumelo() {
9          estado = estado.pegarCogumelo();
10     }
11
12     public void pegarFlor() {
13         estado = estado.pegarFlor();
14     }
15
16     public void pegarPena() {
17         estado = estado.pegarPena();
18     }
19
20     public void levarDano() {
21         estado = estado.levarDano();
22     }
23 }

```

A classe mario possui uma referência para um objeto estado, este estado vai ser atualizado de acordo com as operações de troca de estados, definidas logo em seguida. Quando uma operação for invocada, o objeto estado vai executar a operação e se atualizará automaticamente. Como exemplo de utilização, vejamos o seguinte código cliente:

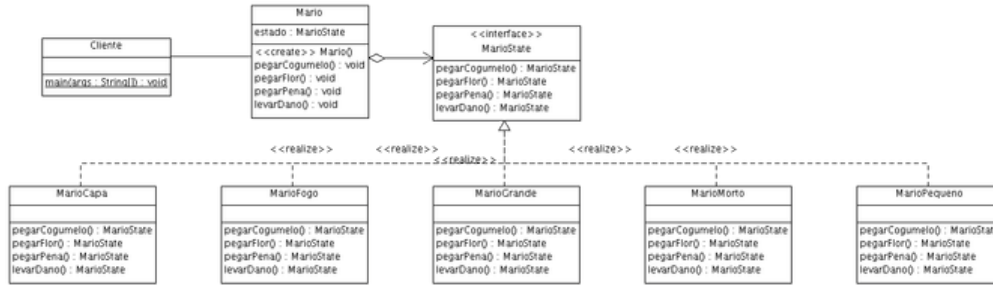
```

1  public static void main(String[] args) {
2      Mario mario = new Mario();
3      mario.pegarCogumelo();
4      mario.pegarPena();
5      mario.levarDano();
6      mario.pegarFlor();
7      mario.pegarFlor();
8      mario.levarDano();
9      mario.levarDano();
10     mario.pegarPena();
11     mario.levarDano();
12     mario.levarDano();
13     mario.levarDano();
14 }

```

Por este código é possível avaliar todas as transições e todos os estados.

O diagrama UML a seguir resume visualmente as relações entre as classes:



(<https://brizeno.files.wordpress.com/2011/11/state.png>).

Um pouco de teoria

Pelo visto no exemplo, o padrão é utilizado quando se precisa isolar o comportamento de um objeto, que depende de seu estado interno. O padrão elimina a necessidade de condicionais complexos e que frequentemente serão repetidos. Com o padrão cada “ramo” do condicional acaba se tornando um objeto, assim você pode tratar cada estado como se fosse um objeto de verdade, distribuindo a complexidade dos condicionais.

Incluir novos estados também é muito simples, basta criar uma nova classe e atualizar as operações de transição de estados. Com a primeira solução seriam necessários vários milhões de ifs novos e a alteração dos já existentes, além do grande risco de esquecer algum estado. Outra grande vantagem é que fica claro, com a estrutura do padrão, quais são os estados e quais são as possíveis transições.

O padrão State não define aonde as transições ocorrem, elas podem ser colocadas dentro das classes de estado ou dentro da classe que armazena o estado. No exemplo vimos que dentro de cada estado são definidos os novos objetos que são retornados. A principal vantagem desta solução é que fica mais simples adicionar os estados, cada novo estado define suas transições. O problema é que assim cada classe de estado precisa ter conhecimento sobre as outras subclasses, e se alguma delas mudar, é provável que a mudança se espalhe.

É comum que objetos estados obedeçam a outros dois padrões: Singleton (<http://wp.me/p1Mek8-1Z>) e Flyweight (<http://wp.me/p1Mek8-45>). Estados singleton são capazes de manter informações, mesmo com as constantes trocas de estados. Estados Flyweight permitem o compartilhamento entre objetos que vão utilizar a mesma máquina de estado.

O padrão State tem semelhanças com outros dois padrões: Strategy (<http://wp.me/s1Mek8-strategy>) e Bridge (<http://wp.me/p1Mek8-2K>). Falando primeiro sobre o Strategy, note que a ideia é muito parecida: eliminar vários ifs complexos espalhados utilizando subclasses. A diferença básica é que o State é mais dinâmico que o Strategy, pois ocorrem várias trocas de objetos estados, os próprios objetos estados realizam as transições.

A semelhança com o padrão Bridge também pode ser notada facilmente pelo diagrama UML, no entanto a diferença está na intenção dos padrões. A intenção do Bridge é permitir que tanto a implementação quanto a interface possam mudar independentemente. No State a ideia é realmente mudar o comportamento de um objeto.

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 aí! 😊

Referências:

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

□ [State](#) □ [Java, Padrões, Projeto, State](#) □ [10 Comentários](#)

10 comentários sobre “Mão na massa: State”

1. [maio 23, 2012 às 11:09 PM](#)

Jackeline Barros [↗](#)

Adorei o código, vai me ajudar muito.. valeu! 😊

[Responder](#)

2. [junho 11, 2013 às 4:43 PM](#)

Heuller César Gomes [↗](#)

Muito bom!

Vlw!

[Responder](#)

3. [dezembro 4, 2014 às 9:31 AM](#)

César

Obrigado pelo exemplo simples...

[Responder](#)

4. [março 31, 2015 às 4:13 PM](#)

André Willik Valenti

Gostei muito cara, parabéns! Vou usar para minhas aulas!

[Responder](#)

[março 31, 2015 às 4:20 PM](#)

Marcos Brizeno [↗](#)

Massa! Fico feliz em ter ajudado!

[Responder](#)

5. [setembro 24, 2015 às 6:31 PM](#)

Ramar Nunes [↗](#)

Primeiramente, parabéns pelos posts, foram feitos com muito cuidado.

Agora um questão. Quando rodei o exemplo, apareceu uma mensagem assim – “Mario Voadr”.

E agora? Tive que olhar todos os estados a procura da mensagem errada.

E se fossem dezenas de estados? E se não fosse uma simples letra omissa?

Alguma sugestão? Devo usar algum padrão de projeto ou usar uma classe pública para strings que são constantes?

Grato pela atenção.

[Responder](#)

[setembro 24, 2015 às 6:36 PM](#)

Marcos Brizeno [↗](#)

Oi Ramar, obrigado pelo apoio.

No exemplo eu deixei as strings “hard-coded” por simplicidade, mas o ideal seria extrair isso para uma classe, ou até mesmo um arquivo de configuração. Assim não precisaria repetir em vários lugares e facilitaria a busca pelo problema, como você relatou.

[Responder](#)

6. [novembro 11, 2015 às 8:39 PM](#)

James

Perfeito, simples e fácil de entender. Preciso fazer uma apresentação explicando o padrão state, e para isso preciso de 2 exemplos, e o seu é perfeito até mesmo para aqueles com mais dificuldades de entender. Parabéns.

[Responder](#)

7. [abril 9, 2017 às 4:03 PM](#)

Rakun

Post bastante interessante, com um exemplo legal.

Só uma observação em relação à implementação. Na especificação do padrão, fica em aberto a opção de onde e como atualizar o estado (se dentro do próprio contexto ou dentro dos estados). Então, acabamos tendo várias formas de fazê-lo.

Uma outra possibilidade é que a interface “StateBase” (MarioState, no exemplo) tenha nos seus métodos handle um parâmetro que é referência do contexto (no exemplo, a classe Mario) e não tem retorno, é void.

Dessa forma, MarioState ficaria com o método: public void handle (Mario mario);

Nas classes de estado concreto (MarioPequeno, por exemplo), a forma de implementação seria algo como:

```
public void pegarPena (Mario mario){  
    System.out.println(“Mario grande com capa”);  
    mario.setEstado(new MarioCapa());  
}
```


E na classe Mario, o método pegarPena ficaria assim:

```
public void pegarPena() {  
    estado.pegarPena(this);  
}
```

Claro que o resultado, no fim das contas, acaba sendo o mesmo. Mas é uma outra abordagem.

 [Responder](#)

 [abril 9, 2017 às 4:59 PM](#)

Marcos Brizeno 

No final das contas acaba ficando a mesma coisa sim, o único comentário com a solução que você sugeriu é que ela altera um objeto que é passado como parâmetro. Recomendo a leitura desse outro post que fala mais sobre o assunto:

<https://brizeno.wordpress.com/2016/07/12/refatorando-tudo-nao-use-parametros-como-retorno/>.

Claro que tudo depende do contexto, sua solução fica mais fácil de usar pois basta chamar o método sem se preocupar com retorno. Mas eu pessoalmente já passei maus bocados lidando com métodos void que alteram os parâmetros :p

Obrigado pelo comentário!

 [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

novembro 19, 2011

Mão na massa: Interpreter

Problema:

Reconhecer padrões é um problema bem complicado, no entanto, quando conseguimos formular uma gramática para o problema a solução fica bem mais fácil. Suponha que é preciso converter uma String representando um número romano em um inteiro que represente seu valor decimal. É fácil perceber que este problema trata-se de reconhecer determinados padrões.

Percorrer a String e procurar cada um dos possíveis casos não é a melhor solução, pois dificultaria bastante a manutenção do código. Então vamos tentar formular o problema como uma gramática. Para simplificar, vamos tratar apenas números de quatro dígitos.

Iniciando a definição da gramática, um número romano é composto por caracteres que representam números de quatro, três, dois ou um dígito:

numero romano ::= {quatro dígitos} {três dígitos} {dois dígitos} {um dígito}

Números de quatro, três, dois e um dígito são formados por caracteres que representam nove, cinco, quatro e um. Com estes caracteres é possível representar qualquer um dos número em romanos:

quatro dígitos ::= um
três dígitos ::= nove | cinco {um} {um} {um} | quatro | um
dois dígitos ::= nove | cinco {um} {um} {um} | quatro | um
um dígito ::= nove | cinco {um} {um} {um} | quatro | um

O “cinco” em romano pode vir seguido por até três números “um”.

E finalmente, os caracteres que representam nove, cinco, quatro e um são os seguinte, considerando apenas números de quatro dígitos:

nove ::= “CM”, “XC”, “IX”
cinco ::= “D”, “L”, “V”
quatro ::= “CD”, “XL”, “IV”
um ::= “M”, “C”, “X”, “I”

Com estas regras podemos criar vários números romanos, por exemplo, CI é igual a 101, pelas regras definidas a construção seria:

número romano -> {três dígitos} {um dígito} -> {um} {um dígito} -> C {um dígito} -> C {um} -> CI

Outro exemplo, CXCIV é 194, pelas regras seria:

número romano -> {três dígitos} {dois dígitos} {um dígito} -> {um} {dois dígitos} {um dígito} -> C {dois dígitos} {um dígito} -> C {nove} {um dígito} -> C XC {um dígito} -> C XC {quatro} -> CXCIV

Uma vez definida a gramática e suas regras, é possível utilizar o padrão Interpreter para montar uma estrutura para interpretar os comandos.

Interpreter

Vejam os a intenção do padrão:

“Dada uma linguagem, definir uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem.” [1]

Ou seja, dada a linguagem, números romanos, construir uma representação para a gramática dela junto com um interpretador para essa gramática.

A estrutura do padrão é muito parecido com a do padrão [Composite](http://wp.me/p1Mek8-M) (<http://wp.me/p1Mek8-M>), é definido inicialmente uma classe abstrata que será a base de todas as classes interpretadoras. Nela é construída a interface básica e o método de interpretar. Este método recebe como atributo um contexto, que é uma classe que vai armazenar as informações de entrada e saída.

Vamos então definir a classe contexto para o nosso problema:

```
1 public class Contexto {
2     protected String input;
3     protected int output;
4
5     public Contexto(String input) {
6         this.input = input;
7     }
8
9     public String getInput() {
10        return input;
11    }
12
13    public void setInput(String input) {
14        this.input = input;
15    }
16
17    public int getOutput() {
18        return output;
19    }
20
21    public void setOutput(int output) {
22        this.output = output;
23    }
24 }
```

Não se preocupe com os getters e setters, eles serão necessários quando formos definir o método de interpretação. O que merece a atenção nesta classe são os atributos de input, que é a String em formato romano, e o output, que é o inteiro que vai armazenar o valor.

Vamos analisar a classe interpretadora em partes. Primeiro vamos dar uma olhada no método de interpretação:

```
1 public abstract class NumeroRomanoInterpreter {
2     public void interpretar(Contexto contexto) {
3         if (contexto.getInput().length() == 0) {
4             return;
5         }
6         // Os valores nove e quatro são os únicos que possuem duas casas
7         // Ex: IV, IX
8         if (contexto.getInput().startsWith(nove())) {
9             adicionarValorOutput(contexto, 9);
10            consumirDuasCasasDoInput(contexto);
11        } else if (contexto.getInput().startsWith(quatro())) {
12            adicionarValorOutput(contexto, 4);
13            consumirDuasCasasDoInput(contexto);
14        } else if (contexto.getInput().startsWith(cinco())) {
15            adicionarValorOutput(contexto, 5);
16            consumirUmaCasaInput(contexto);
17        }
18        // Os valores de um são os únicos que repetem, ex: III, CCC, MMM
19        while (contexto.getInput().startsWith(um())) {
20            adicionarValorOutput(contexto, 1);
21            consumirUmaCasaInput(contexto);
22        }
23    }
24
25    private void consumirUmaCasaInput(Contexto contexto) {
26        contexto.setInput(contexto.getInput().substring(1));
27    }
28
29    private void consumirDuasCasasDoInput(Contexto contexto) {
30        contexto.setInput(contexto.getInput().substring(2));
31    }
32 }
```

Este método recebe o contexto e a ideia é fazer o seguinte: comparar os primeiros caracteres da String com os caracteres que representam nove, quatro, cinco e um. Quando um destes padrões for encontrado é retirado da String, para que não seja repetido, e o seu valor é adicionado ao valor de output do contexto.

Um detalhe que deve ser observado é que os valores que representam nove ou quatro possuem dois caracteres, assim é necessário retirar dois caracteres da string de input. Feito o método de interpretação é necessário definir as strings que vão representar os caracteres nove, cinco, quatro e um.

Além disso, existe outro método não definido, o “multiplicador()”. Este método vai retornar qual o valor relativo do número, por exemplo, se for um número romano de quatro dígitos, o método retornará 1000, se for um de três retornará 100, etc.

```
1 public abstract class NumeroRomanoInterpreter {
2
3     ...
4
5     public abstract String um();
6
7     public abstract String quatro();
8
9     public abstract String cinco();
10
11    public abstract String nove();
12
13    public abstract int multiplicador();
14 }
```

Estes métodos serão definidos nas subclasses. Lembra das definições das regras da gramática? Cada regra daquela vai se tornar uma classe derivada da classe interpretadora. Nela vão ser definidas as strings de um, quatro, cinco e nove. Também será definido o multiplicado relativo de cada uma.

Vejamos então como exemplo a definição da classe que representa números de um dígito:

```
1 public class UmDigitoRomano extends NumeroRomanoInterpreter {
2
3     @Override
4     public String um() {
5         return "I";
6     }
7
8     @Override
9     public String quatro() {
10        return "IV";
11    }
12
13    @Override
14    public String cinco() {
15        return "V";
16    }
17
18    @Override
19    public String nove() {
20        return "IX";
21    }
22
23    @Override
24    public int multiplicador() {
25        return 1;
26    }
27
28 }
```

Nesta classe definimos os números de um dígito, ou uma casa decimal, em caracteres romanos: I, IV, V e IX. O valor do multiplicador será 1. Como outro exemplo, vejamos a classe que representam números de dois dígitos:

```

1  public class DoisDigitosRomano extends NumeroRomanoInterpreter {
2
3      @Override
4      public String um() {
5          return "X";
6      }
7
8      @Override
9      public String quatro() {
10         return "XL";
11     }
12
13     @Override
14     public String cinco() {
15         return "L";
16     }
17
18     @Override
19     public String nove() {
20         return "XC";
21     }
22
23     @Override
24     public int multiplicador() {
25         return 10;
26     }
27 }
28

```

De maneira análoga, nesta classe são definidos os número de duas casas decimais: X, XL, L e XC. O valor do multiplicador será 10. As outras classes seguirão da mesma maneira.

Então vejamos como ficaria o cliente do interpreter deste exemplo:

```

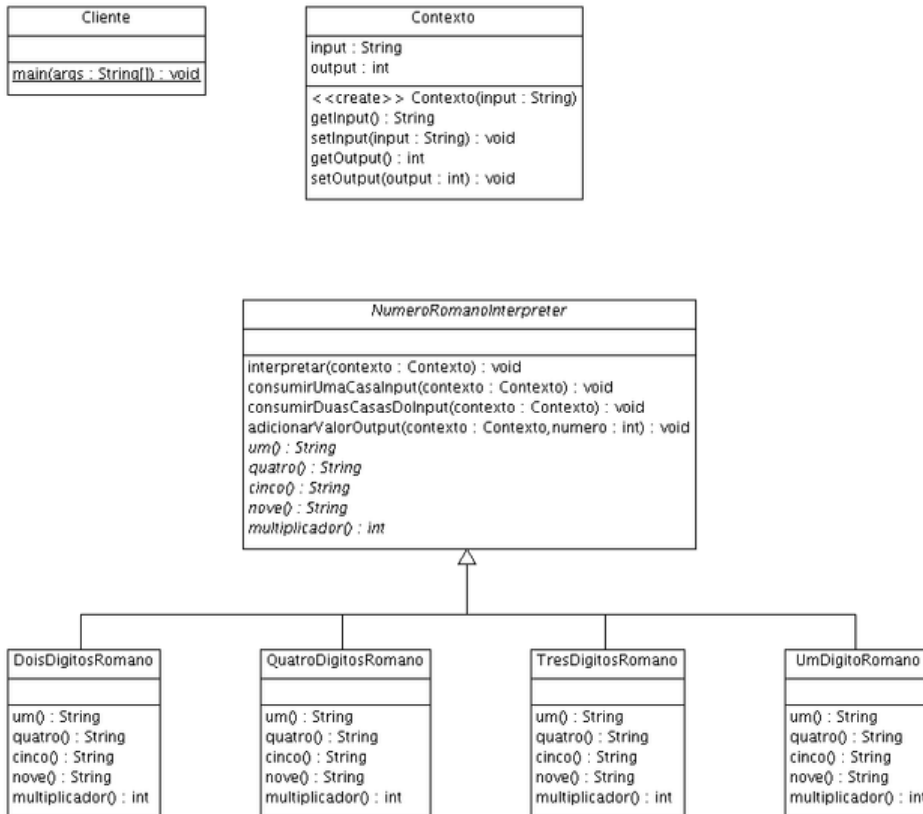
1  public static void main(String[] args) {
2      ArrayList<Interpreter> interpretadores = new ArrayList<>();
3      interpretadores.add(new QuatroDigitosRomano());
4      interpretadores.add(new TresDigitosRomano());
5      interpretadores.add(new DoisDigitosRomano());
6      interpretadores.add(new UmDigitoRomano());
7
8      String numeroRomano = "CXCIV";
9      Contexto contexto = new Contexto(numeroRomano);
10
11     for (NumeroRomanoInterpreter numeroRomanoInterpreter : interpretadores) {
12         numeroRomanoInterpreter.interpretar(contexto);
13     }
14
15     System.out.println(numeroRomano + " = "
16         + Integer.toString(contexto.getOutput()));
17 }

```

Primeiro criamos uma lista onde vamos inserir todos os interpretadores, depois, vamos iterar sobre essa lista chamando os métodos de interpretação e passando um mesmo contexto como parâmetro. No final, podemos verificar o resultado pela saída no terminal.

O exemplo completo pode ser baixado pelo repositório do GitHub, link no final do post. Experimente executar outras instâncias de números romanos e observar o fluxo de chamadas para entender melhor como o padrão foi aplicado.

O diagrama UML para este caso é o seguinte:



(<https://brizeno.files.wordpress.com/2011/11/interpreter.png>).

Um pouco de teoria

Pelo exemplo vimos que a maior dificuldade em utilizar o Interpreter é na verdade conseguir modelar uma gramática para o problema. Feito isso, cada regra da gramática acaba se tornando uma subclasse da expressão abstrata. Desta maneira é fácil implementar a gramática, as classes interpretadoras ficam bem simples.

Outro ponto é a facilidade de alteração e extensão da gramática, pois basta criar novas classes que implementem as novas regras. Suponha que agora a gramática precisa abranger números de cinco dígitos? Basta criar uma nova classe que defina essas regras e alterar o método de interpretação.

Este método de interpretação, quando observado de perto, lembra bastante outro padrão, o Template Method (<http://wp.me/p1Mek8-1C>). Perceba que ele define um algoritmo padrão de interpretação e deixa alguns pontos ganchos que deverão ser implementados nas subclasses.

O padrão Interpreter se assemelha bastante ao Composite. Ambos podem ser entendidos com a ideia de árvores, já que existe uma classe terminal ou folha, e uma classe composta ou nó. Basta analisar os diagramas UML dos dois padrões para perceber a semelhança.

(<http://www.dofactory.com/Patterns/Diagrams/composite.gif>).

(<http://www.dofactory.com/Patterns/Diagrams/interpreter.gif>).

No exemplo citado não houve a necessidade de utilizar uma classe interpreter que tivesse outros objetos interpreter. No entanto pense no seguinte exemplo: a notação musical é composta de notas, que seriam os terminais, e acordes, que seriam agrupamentos de notas. Então um método de interpretação em notas poderia tocar a nota e um método de interpretação em acorde tocaria cada uma das notas do acorde.

A diferença entre os dois padrões é que o Composite, obrigatoriamente define métodos para gerenciamento da estrutura. Na classe composta são encontrados os métodos de adição e remoção de elementos. Já no Interpreter, esses métodos não são necessários, a ideia de árvore é utilizada apenas para facilitar a visualização.

Outra grande diferença é que o padrão Interpreter depende de um contexto externo, enquanto que no Composite esse contexto é desnecessário. No entanto, ambos os padrões se utilizam, pois, geralmente um Composite define alguma operação de interpretação, e a estrutura do Interpreter geralmente é um Composite.

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 aí! 😊

Referências:

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

□ [Interpreter](#) □ [Interpreter, Java, Padrões, Projeto](#) □ [7 Comentários](#)

7 comentários sobre “Mão na massa: Interpreter”

1. □ [agosto 2, 2012 às 12:07 PM](#)

Rejane

Olá!

Gostei muito desse artigo, parabéns.

□ [Responder](#)

2. □ [maio 14, 2013 às 10:08 PM](#)

Paulo

Ajudou demais na direção.

□ [Responder](#)

3. □ [novembro 9, 2013 às 9:55 AM](#)

Franklin G Mendes

Bom dia Amigo, perfeito exemplo, nao consegui encontrar o exemplo em seu repositorio, pode me encaminhar o link ou o codigo do mesmo. email: franklingmendes@gmail.com
Obrigado

□ [Responder](#)


□ [novembro 11, 2013 às 9:32 PM](#)

Marcos Brizeno 

Valeu cara. Vou dar uma olhada no repositório e tentar atualizar com esse código.

□ [Responder](#)

4. □ [dezembro 21, 2015 às 7:13 PM](#)

Ricardo Assis 

Apesar de estar bem legível está faltando alguns métodos no código exposto acima.

□ [Responder](#)

□ [dezembro 22, 2015 às 8:52 AM](#)

Marcos Brizeno 

O código completo pode ser encontrado aqui Ricardo: <https://github.com/MarcosX/Padr-es-de-Projeto>

□ [Responder](#)

5. □ [novembro 4, 2017 às 7:09 PM](#)

Anônimo

Muito boa explicação! Melhor que alguns professores.

□ [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

novembro 5, 2011

Mão na massa: Memento

Problema:

Qualquer bom editor, seja de vídeo, texto, imagens, etc. oferece uma maneira de desfazer ações, recuperando estados anteriores. Como é possível modelar a arquitetura do sistema de maneira que seja possível salvar estados dos elementos?

Para exemplificar vamos pensar em editor de texto que precisa manter o controle apenas do texto que é digitado, um notepad por exemplo. Uma primeira saída poderia ser criar um objeto que representasse o texto com suas informações e guardar estes objetos em uma lista.

O problema desta implementação está em como recuperar os estados sem ferir o encapsulamento da classe? Pois, ao restaurar o objeto precisaríamos de, no mínimo, getters para acessar as informações do objeto e poder copiá-las para o objeto a ser restaurado.

O padrão Memento oferece uma maneira simples de evitar o problema da quebra de encapsulamento e manter o controle das alterações feitas em um objeto. Vejamos então o padrão:

Memento

Intenção:

“Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde.” [1]

Pela intenção do padrão podemos facilmente notar sua aplicabilidade. O estado interno do objeto seria, para o exemplo acima, o texto que está sendo digitado pelo usuário. Assim, o padrão Memento permitiria capturar o estado do texto para que depois ele possa ser reutilizado.

Vamos então ao código do problema. Vamos iniciar com a classe Memento. Ela simplesmente mantém a String que representa o texto e oferece um getter para esta String, permitindo que ela seja recuperada mais tarde.

```
1 public class TextoMemento {
2     protected String estadoTexto;
3
4     public TextoMemento(String texto) {
5         estadoTexto = texto;
6     }
7
8     public String getTextoSalvo() {
9         return estadoTexto;
10    }
11 }
```

Além do Memento, existe outra figura importante, o Caretaker. O Caretaker vai guardar todos os Memento, permitindo que eles sejam restaurados. Como a aplicação é de um editor de texto os Memento devem ser recuperados de maneira LIFO, *Last in First out*, assim o último memento adicionado será o primeiro a ser recuperado. Vejamos o código a seguir:

```

1 public class TextoCaretaker {
2     protected ArrayList<TextoMemento> estados;
3
4     public TextoCaretaker() {
5         estados = new ArrayList<TextoMemento>();
6     }
7
8     public void adicionarMemento(TextoMemento memento) {
9         estados.add(memento);
10    }
11
12    public TextoMemento getUltimoEstadoSalvo() {
13        if (estados.size() <= 0) {
14            return new TextoMemento("");
15        }
16        TextoMemento estadoSalvo = estados.get(estados.size() - 1);
17        estados.remove(estados.size() - 1);
18        return estadoSalvo;
19    }
20 }

```

A partir do Caretaker é possível armazenar e recuperar um estado. No método que retorna o último estado salvo é necessário fazer uma verificação se existe algum estado a ser retornado, caso contrário é retornado um memento vazio.

Uma pequena observação: retornar nulo exigiria uma verificação que poderia facilmente ser esquecida, causando vários problemas na execução do programa. O ideal seria disparar uma exceção, mas como estamos apenas exemplificando, retornar um objeto que não tenha informações é mais simples. Lembre-se: se você precisa retornar nulo, é melhor repensar no seu método.

Agora vamos analisar a classe que representa o Texto:

```

1 public class Texto {
2     protected String texto;
3     TextoCaretaker caretaker;
4
5     public Texto() {
6         caretaker = new TextoCaretaker();
7         texto = new String();
8     }
9
10    public void escreverTexto(String novoTexto) {
11        caretaker.adicionarMemento(new TextoMemento(texto));
12        texto += novoTexto;
13    }
14
15    public void desfazerEscrita() {
16        texto = caretaker.getUltimoEstadoSalvo().getTextoSalvo();
17    }
18
19    public void mostrarTexto() {
20        System.out.println(texto);
21    }
22 }

```

A classe texto possui uma interface que permite escrever um texto, desfazer a operação de escrita e exibir o texto no terminal. Ao escrever um novo texto, primeiro o estado é salvo, então a alteração é feita. Ao desfazer a escrita é solicitado ao Caretaker que pegue o último estado salvo, a partir deste estado é possível pegar o texto e restaurá-lo.

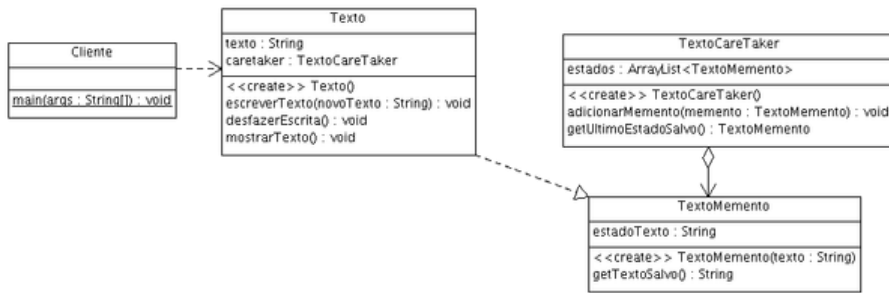
A utilização do padrão seria algo do tipo:

```

1 public static void main(String[] args) {
2     Texto texto = new Texto();
3     texto.escreverTexto("Primeira linha do texto\n");
4     texto.escreverTexto("Segunda linha do texto\n");
5     texto.escreverTexto("Terceira linha do texto\n");
6     texto.mostrarTexto();
7     texto.desfazerEscrita();
8     texto.mostrarTexto();
9     texto.desfazerEscrita();
10    texto.mostrarTexto();
11    texto.desfazerEscrita();
12    texto.mostrarTexto();
13    texto.desfazerEscrita();
14    texto.mostrarTexto();
15 }

```

O diagrama UML para este exemplo seria:



(<https://brizenofiles.wordpress.com/2011/11/memento.png>)

Um pouco de teoria

O padrão Memento oferece uma maneira simples de salvar estados internos de um objeto. Basta salvar todas as informações necessárias em Memento e mais tarde recuperá-las. Ele transfere a responsabilidade de fornecer maneiras de acessar o estado para o objeto Memento, deixando o Originator (no nosso exemplo, a classe Texto) livre destas preocupações.

Uma desvantagem fácil de ser notada é que armazenar a lista de Memento pode ser caro, computacionalmente. Assim, caso o estado seja muito complexo, pode-se utilizar uma classe intermediária que armazena o estado para simplificar a arquitetura, mas não a complexidade. Em muitos editores é comum notar que é possível configurar a quantidade de espaço a ser utilizado para salvar estados do programa. Assim, ao salvar um Memento, o Caretaker pode verificar se o limite foi atingido e eliminar os Memento mais antigos, caso seja necessário.

Outra desvantagem é que é necessário tomar cuidado para que não seja possível ter acesso ao objeto Memento, pois nada impede que apenas o Caretaker, ou o Originator acessem o estado do Memento. Em C++ é possível utilizar o operador **friend** para que os campos privados sejam visíveis somente em algumas classes.

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 aí! 😊

Referências:

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

□ [Memento, Padrões de Projeto](#) □ [Java, Memento, Padrões, Projeto](#) □ [4 Comentários](#)

4 comentários sobre “Mão na massa: Memento”

1. □ [março 16, 2013 às 10:22 PM](#)

Dênis Schmidt

Bem legal a explicação sobre Memento. Porém fiz a implementação um pouco diferente, veja se gosta.

```
package com.schimidsolutions.memento;
```

```

public class Texto {
private final TextoCareTaker textoCareTaker = new TextoCareTaker();
private final StringBuilder textoAtual = new StringBuilder();

public void escrever( final String texto ) {
textoCareTaker.adicionarTextoNoHistorico( new TextoDigitadoMemento( textoAtual.toString() ) );
textoAtual.append( texto );
}

public void desfazer() {
textoAtual.delete( 0, textoAtual.length() - 1 );
textoAtual.append( textoCareTaker.recuperarTextoDoHistorico().getTextoDigitado() );
}

public void mostrarTextoDigitado() {
System.out.println( textoAtual.toString() );
}
}

package com.schmidtsolutions.memento.teste;

import com.schmidtsolutions.memento.Texto;

public class TesteMemento {

public static void main(final String[] args) {
final Texto texto = new Texto();

texto.escrever( "Meu Texto" );
texto.escrever( " são " );
texto.desfazer();
texto.escrever( " é " );
texto.escrever( " ruim " );
texto.desfazer();
texto.escrever( " muito bom!!! " );

texto.mostrarTextoDigitado();
}
}

package com.schmidtsolutions.memento;

import java.util.Stack;

class TextoCareTaker {
private final Stack historicoTexto = new Stack();

void adicionarTextoNoHistorico( final TextoDigitadoMemento textoMemento ) {
historicoTexto.add(textoMemento);
}

TextoDigitadoMemento recuperarTextoDoHistorico() {
return historicoTexto.pop();
}
}

package com.schmidtsolutions.memento;

class TextoDigitadoMemento {
private final String textoDigitado;

TextoDigitadoMemento(final String textoDigitado) {
this.textoDigitado = textoDigitado;
}

final String getTextoDigitado() {
return textoDigitado;
}
}

```

📝 [Responder](#)

📅 [março 17, 2013 às 6:50 PM](#)

marcosbrizeno ↗

Padrões de projeto são ideias e não uma implementação que deve ser seguida e repetida. Muito bom que você implementou sua própria versão!

 [Responder](#)


2.  [junho 24, 2013 às 3:06 AM](#)

Marcela

Adorei a sua explicação, facilitou bastante o entendimento, principalmente por causa dos exemplos.

 [Responder](#)

3.  [novembro 14, 2015 às 3:58 PM](#)

Daniel Lucas 

Muito bom! Me ajudou muito!

 [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

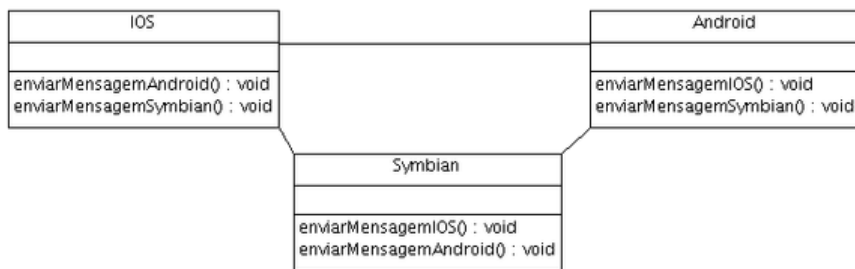
outubro 26, 2011

Mão na massa: Mediator

Problema

Pense na seguinte situação: seria legal ter um aplicativo que trocasse mensagem entre diversas plataformas móveis, um Android enviando mensagem para um iOS, um Symbian trocando mensagens com um Android... O problema é que cada uma destas plataformas implementa maneiras diferentes de receber mensagens.

Obviamente seria uma péssima solução criar vários métodos para cada plataforma. Analise o diagrama abaixo:



(<https://brizen.files.wordpress.com/2011/10/relacionamento-muitos-para-muitos.png>).

Imagine que agora o aplicativo vai incluir a plataforma BlackBerry OS, precisaríamos criar os métodos de comunicação com todas as outras plataformas existentes, além de adicionar métodos em todas as outras plataformas para que elas se comuniquem com o BlackBerry OS.

Esta ideia de relacionamento muitos para muitos pode deixar o design bem complexo, comprometendo a eficiência do sistema, bem como sua manutenibilidade.

Quando uma situação em que um relacionamento muitos para muitos é necessário em Banco de Dados, uma boa prática é criar uma tabela intermediária e deixar que ela relaciona uma entidade com outras várias e vice-versa. Esta é a ideia do padrão Mediator.

Mediator

Intenção:

“Definir um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permitir variar suas interações independentemente.” [1]

Pela intenção podemos perceber que o Mediator atua como um mediador entre relacionamentos muitos para muitos, ao evitar uma referência explícita aos objetos. Outra vantagem que podemos notar é também que ele concentra a maneira como os objetos interagem.

O padrão Mediator consiste de duas figuras principais: o Mediator e o Colleague. O Mediator recebe mensagens de um Colleague, define qual protocolo utilizar e então envia a mensagem. O Colleague define como receberá uma mensagem e envia uma mensagem para um Mediator.

Vamos então implementar o Colleague que servirá como base para todos os outros:

```
1 public abstract class Colleague {
2     protected Mediator mediator;
3
4     public Colleague(Mediator m) {
5         mediator = m;
6     }
7
8     public void enviarMensagem(String mensagem) {
9         mediator.enviar(mensagem, this);
10    }
11
12    public abstract void receberMensagem(String mensagem);
13 }
```

Simples, define apenas a interface comum de qualquer Colleague. Todos possuem um Mediator, que deve ser compartilhado entre os objetos Colleague. Também define a maneira como todos os objetos Colleague enviam mensagens. O método “receberMensagem()” fica a cargo das subclasses.

Como exemplo de Colleague, vejamos as classes a seguir, que representam as plataformas Android e iOS:

```
1 public class IOSColleague extends Colleague {
2
3     public IOSColleague(Mediator m) {
4         super(m);
5     }
6
7     @Override
8     public void receberMensagem(String mensagem) {
9         System.out.println("iOS recebeu: " + mensagem);
10    }
11 }

```

```
1 public class AndroidColleague extends Colleague {
2
3     public AndroidColleague(Mediator m) {
4         super(m);
5     }
6
7     @Override
8     public void receberMensagem(String mensagem) {
9         System.out.println("Android recebeu: " + mensagem);
10    }
11 }
```

As classes Colleague concretas também são bem simples, apenas definem como a mensagem será recebida.

Vejamos então como funciona o Mediator. Vamos primeiro definir a interface comum de qualquer Mediator:

```
1 public interface Mediator {
2
3     void enviar(String mensagem, Colleague colleague);
4
5 }
```

Ou seja, todo Mediator deverá definir uma maneira de enviar mensagens. Vejamos então como o Mediator concreto seria implementado:

```

1  public class MensagemMediator implements Mediator {
2
3      protected ArrayList<Colleague> contatos;
4
5      public MensagemMediator() {
6          contatos = new ArrayList<Colleague>();
7      }
8
9      public void adicionarColleague(Colleague colleague) {
10         contatos.add(colleague);
11     }
12
13     @Override
14     public void enviar(String mensagem, Colleague colleague) {
15         for (Colleague contato : contatos) {
16             if (contato != colleague) {
17                 definirProtocolo(contato);
18                 contato.receberMensagem(mensagem);
19             }
20         }
21     }
22
23     private void definirProtocolo(Colleague contato) {
24         if (contato instanceof IOSColleague) {
25             System.out.println("Protocolo iOS");
26         } else if (contato instanceof AndroidColleague) {
27             System.out.println("Protocolo Android");
28         } else if (contato instanceof SymbianColleague) {
29             System.out.println("Protocolo Symbian");
30         }
31     }
32
33 }

```

O Mediator possui uma lista de objetos Colleague que realizarão a comunicação e um método para adicionar um novo Colleague.

O método “enviar()” percorre toda a lista de contatos e envia mensagens. Note que dentro deste métodos foi feita uma comparação para evitar a mensagem seja enviada para a pessoa que enviou. Para enviar a mensagem primeiro deve ser definido qual protocolo utilizar e em seguida enviar a mensagem.

No nosso exemplo, o método “definirProtocolo()” apenas imprime na tela o tipo do Colleague que enviou a mensagem, utilizar para isso a verificação instanceof.

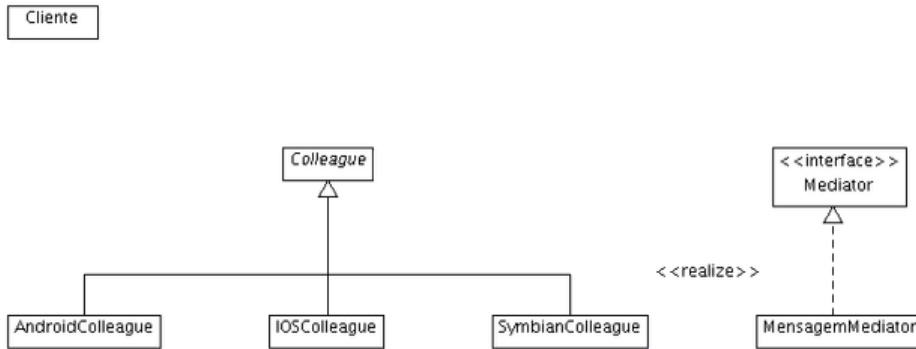
Desta maneira, o cliente poderia ser algo do tipo:

```

1  public static void main(String[] args) {
2      MensagemMediator mediador = new MensagemMediator();
3
4      AndroidColleague android = new AndroidColleague(mediador);
5      IOSColleague ios = new IOSColleague(mediador);
6      SymbianColleague symbian = new SymbianColleague(mediador);
7
8      mediador.adicionarColleague(android);
9      mediador.adicionarColleague(ios);
10     mediador.adicionarColleague(symbian);
11
12     symbian.enviarMensagem("Oi, eu sou um Symbian!");
13     System.out.println("=====");
14     android.enviarMensagem("Oi Symbian! Eu sou um Android!");
15     System.out.println("=====");
16     ios.enviarMensagem("Olá todos, sou um iOS!");
17 }

```

O diagrama UML para este exemplo seria o seguinte:



(<https://brizeno.files.wordpress.com/2011/10/mediator.png>).

Um pouco de teoria

O padrão Mediator tem como principal objetivo diminuir a complexidade de relacionamentos entre objetos, garantindo assim que todos fiquem mais livres para sofrer mudanças, bem como facilitando a introdução de novos tipos de objetos ao relacionamento.

Outro ganho é a centralização da lógica de controle de comunicação entre os objetos, imagine que o protocolo de comunicação com o Android precisasse ser alterado, a mudança seria em um local bem específico da classe Mediator.

Uma vantagem não muito explorada nesse exemplo é que o Mediator centraliza também o controle dos objetos Colleague. Como citamos no post anterior sobre o padrão Observer (<http://wp.me/p1Mek8-2T>), quando o relacionamento entre objetos Observer e Subject fica muito complexo, pode ser necessário utilizar uma classe intermediária que mapeie o relacionamento, facilitando o envio de mensagens aos objetos Observer.

Ao introduzir o Mediator vimos que a complexidade das classes Colleague foi transferida para o Mediator, o que tornou as classes Colleague bem mais simples e fáceis de manter. No entanto isto também pode ser um problema, pois a classe Mediator pode crescer em complexidade e se tornar difícil de manter.

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 aí! 😊

Referências:

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

❑ [Mediator, Padrões de Projeto](#) ❑ [Java, Mediator, Padrões, Projeto](#) ❑ [2 Comentários](#)

2 comentários sobre “Mão na massa: Mediator”

1. [❑ agosto 21, 2013 às 11:52 AM](#)

Ricardo Ferreira

Marcos, parabéns pelo tutorial, muito bom. Eu quero dar uma opinião em relação aos protocolos, não seria melhor cada classe Android, IOS e Symbian terem o método definirProtocolo, porque se precisar mudar do android por exemplo, não ficaria amarrado.

[❑ Responder](#)

📅 agosto 22, 2013 às 9:42 AM

marcosbrizen ↗

Que bom que gostou. Sua pergunta foi muito boa, sim as classes Colleague poderiam definir o protocolo e no método enviar bastaria chamar contato.definirProtocolo.

📅 Responder

Marcos Brizen

Desenvolvimento de Software #showmethecode

novembro 9, 2011

Mão na massa: Chain of Responsibility

Problema:

Uma aplicação de e-commerce precisa se comunicar com vários bancos diferentes para prover aos seus usuários mais possibilidades de pagamentos, atingindo assim um número maior de usuários e facilitando suas vidas.

Ao modelar uma forma de execução do pagamento, dado que precisamos selecionar um entre vários tipos de bancos, a primeira ideia que surge é utilizar uma estrutura de decisão para verificar, dado um parâmetro, qual o banco correto deve ser utilizado.

Já discutimos no post sobre o padrão [Strategy](http://wp.me/s1Mek8-strategy) (<http://wp.me/s1Mek8-strategy>) que utilizar uma estrutura de decisão pode ser muito complexo, e vimos uma boa solução para este problema. Também vimos exemplos de utilização de estruturas de decisão nos padrões [Abstract Factory](http://wp.me/p1Mek8-1h) (<http://wp.me/p1Mek8-1h>) e [Factory Method](http://wp.me/p1Mek8-1c) (<http://wp.me/p1Mek8-1c>).

Poderíamos utilizar os métodos fábricas para gerar o objeto correto para ser utilizado na nossa aplicação. Poderíamos criar estratégias diferentes para cada banco e escolher em tempo de execução.

Em todas estas soluções, nós utilizamos uma forma de “esconder” a estrutura de decisão por trás de uma interface, ou algo similar, para que as alterações fossem menos dolorosas. No entanto, continuamos utilizando as estruturas de decisão.

Vamos analisar então o padrão Chain of Responsibility, que promete acabar com estas estruturas.

Chain of Responsibility

O padrão Chain of Responsibility possui a seguinte intenção:

“Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.” [1]

Pela intenção percebemos como o Chain of Responsibility acaba com as estruturas de decisão, ele cria uma cadeia de objetos e vai passando a responsabilidade entre eles até que alguém possa responder pela chamada.

Vamos então iniciar construindo uma pequena enumeração para identificar os bancos utilizados no nosso sistema:

```
1 | public enum IDBancos {  
2 |     bancoA, bancoB, bancoC, bancoD  
3 | }
```

Agora vamos construir a classe que vai implementar a cadeia de responsabilidades. Vamos exibir partes do código para facilitar o entendimento.

```

1 public abstract class BancoChain {
2
3     protected BancoChain next;
4     protected IDBancos identificadorDoBanco;
5
6     public BancoChain(IDBancos id) {
7         next = null;
8         identificadorDoBanco = id;
9     }
10
11     public void setNext(BancoChain forma) {
12         if (next == null) {
13             next = forma;
14         } else {
15             next.setNext(forma);
16         }
17     }
18 }

```

A nossa classe possui apenas dois atributos, o identificador do banco e uma referência para o próximo objeto da corrente. No construtor inicializamos estes atributos. O método setNext recebe uma nova instância da classe e faz o seguinte:

Se o próximo for nulo, então o próximo na corrente será o parâmetro. Caso contrário, repassa esta responsabilidade para o próximo elemento. Assim, a instância que deve ser adicionada na corrente irá percorrer os elementos até chegar no último elemento.

O próximo passo será criar o método para efetuar o pagamento.

```

1 public void efetuarPagamento(IDBancos id) throws Exception {
2     if (podeEfetuarPagamento(id)) {
3         efetuaPagamento();
4     } else {
5         if (next == null) {
6             throw new Exception("banco não cadastrado");
7         }
8         next.efetuarPagamento(id);
9     }
10 }
11
12 private boolean podeEfetuarPagamento(IDBancos id) {
13     if (identificadorDoBanco == id) {
14         return true;
15     }
16     return false;
17 }
18
19 protected abstract void efetuaPagamento();

```

A primeira parte do algoritmo de pagamento é verificar se o banco atual pode fazer o pagamento. Para isto é utilizado o identificador do banco, que é comparado com o identificador passado por parâmetro. Se o elemento atual puder responder a requisição é chamado o método que vai efetuar o pagamento de fato. Este método é abstrato, e as subclasses devem implementá-lo, com seu próprio mecanismo.

Se o elemento atual não puder responder, ele repassa a chamado ao próximo elemento da lista. Antes disto é feita uma verificação, por questões de segurança, se este próximo elemento realmente existe. Caso nenhum elemento possa responder, é disparada uma exceção.

Agora que definimos a estrutura da cadeia de responsabilidades, vamos implementar um banco concreto, que responde a uma chamada.

```

1 public class BancoA extends BancoChain {
2
3     public BancoA() {
4         super(IDBancos.bancoA);
5     }
6
7     @Override
8     protected void efetuaPagamento() {
9         System.out.println("Pagamento efetuado no banco A");
10    }
11 }

```

O Banco A inicializa seu ID e, no método de efetuar o pagamento, exibe no terminal que o pagamento foi efetuado. A implementação dos outros bancos segue este exemplo. O ID é iniciado e o método de efetuar o pagamento exibe a saída no terminal.

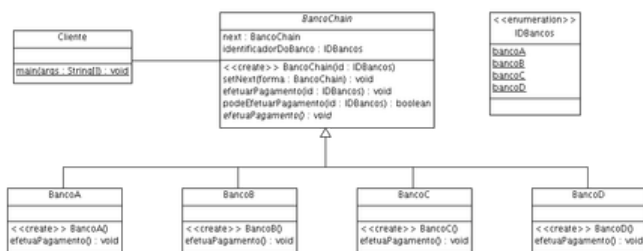
O cliente deste código seria algo do tipo:

```

1 public static void main(String[] args) {
2     BancoChain bancos = new BancoA();
3     bancos.setNext(new BancoB());
4     bancos.setNext(new BancoC());
5     bancos.setNext(new BancoD());
6
7     try {
8         bancos.efetuarPagamento(IDBancos.bancoC);
9         bancos.efetuarPagamento(IDBancos.bancoD);
10        bancos.efetuarPagamento(IDBancos.bancoA);
11        bancos.efetuarPagamento(IDBancos.bancoB);
12    } catch (Exception e) {
13        e.printStackTrace();
14    }
15 }

```

O diagrama UML deste exemplo seria:



(<https://brizenio.files.wordpress.com/2011/11/chain.png>).

Um pouco de teoria

Vimos pelo exemplo que o padrão Chain of Responsibility fornece uma maneira de tomar decisões com um fraco acoplamento. Perceba que a estrutura de cadeia não possui qualquer informação sobre as classes que compõem a cadeia, da mesma forma, uma classe da cadeia não tem nenhuma noção sobre o formato da estrutura ou sobre elementos nela inseridos.

Assim, é possível variar praticamente todos os componentes sem grandes danos ao projeto. Cada elemento implementa sua própria maneira de responder a requisição, e estas podem ser alteradas facilmente.

O problema é que é preciso tomar cuidado para garantir que as chamadas sejam realmente respondidas. No exemplo foi feita uma verificação para saber se o próximo elemento é nulo, para evitar uma acesso ilegal. Mas esta é uma solução para este problema específico. Cada problema exige o seu próprio cuidado.

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.

□ [Chain of Responsibility](#). □ [Chain of Responsibility, Java, Padrões, Projeto](#) □ [9 Comentários](#)

9 comentários sobre “Mão na massa: Chain of Responsibility”

1. □ [novembro 2, 2013 às 10:40 AM](#)

André

Muito bom artigo, Brizeno, parabéns!

Uma observação meio fora do assunto, faça uma revisão ortográfica na seção “about.me”. Dará mais credibilidade ao seu blog. Obrigado!

□ [Responder](#)

2. □ [novembro 8, 2013 às 8:52 AM](#)

Lucas

Muito bom, muito obrigado por compartilhar o conhecimento!

□ [Responder](#)

3. □ [novembro 11, 2013 às 9:21 AM](#)

Charles

Ótimo trabalho me ajudou bastante.

□ [Responder](#)

□ [novembro 11, 2013 às 9:32 PM](#)

Marcos Brizeno 

Valeu!

□ [Responder](#)

4. □ [outubro 4, 2014 às 10:36 PM](#)

Felipe Girotti 

Cara foi uma linguagem simples com exemplos simples que descreve exatamente seu uso, parabéns!

□ [Responder](#)

□ [outubro 6, 2014 às 8:32 AM](#)

Marcos Brizeno 

Obrigado! Fico feliz que tenha ajudado!

□ [Responder](#)

5. □ [fevereiro 24, 2016 às 2:57 PM](#)

Anônimo

Boa tarde Marcos, no seu post sobre Chain of Responsibility não consegui perceber a vantagem em utilizar ele no lugar do Strategy, pois você não executa toda a cadeia de comandos, só executa o comando que atenda ao parâmetro passado!

Se poder por favor esclarece essa minha dúvida.

Desde Obrigado

□ [Responder](#)

6. □ [fevereiro 24, 2016 às 2:58 PM](#)

Renan Aragão 

Boa tarde, não conseguir perceber a vantagem de usar no Chain of Responsibility ao invés do Strategy, pois em ambos você verifica o parâmetro passado, não seria mais apropriado utilizar o Strategy?

Obrigado pelos post's 😊

□ [Responder](#)

□ [fevereiro 25, 2016 às 9:20 AM](#)

Marcos Brizeno 

Oi Renan, a ideia do Chain of Responsibility é encadear várias estratégias. Então não é que um é melhor que o outro, eles tem usos diferentes.

□ [Responder](#)

Marcos Brizen

Desenvolvimento de Software #showmethecode

setembro 15, 2011

Mão na massa: Iterator

O padrão de hoje é o Iterator!

Problema

Imagine que você trabalha em uma empresa de Tv a Cabo. Você recebeu a tarefa de mostrar a lista de canais que a empresa oferece. Ao procurar os desenvolvedores dos canais você descobre que existe uma separação entre os desenvolvedores que cuidam dos canais de esportes e os que cuidam dos canais de filmes. O problema começa quando você percebe que, apesar de ambos utilizarem uma lista de canais, os desenvolvedores dos canais de filmes utilizaram Matriz para representar a lista de canais e os desenvolvedores dos canais de esportes utilizaram ArrayList.

Você não quer padronizar as listas pois todo o resto do código do sistema que cada equipe fez utiliza sua própria implementação (ArrayList ou Matriz). Como construir o programa que vai exibir o nome dos canais?

A solução mais simples é pegar a lista de canais e fazer dois loops, um para percorrer o ArrayList e outro para percorrer a Matriz, e em cada loop exibir o nome dos canais. A impressão dos canais seria algo desse tipo:

```
1  ArrayList<Canal> arrayListDeCanais = new ArrayList<Canal>();
2  Canal[] matrizDeCanais = new Canal[5];
3
4  for (Canal canal : arrayListDeCanais) {
5      System.out.println(canal.nome);
6  }
7
8  for (int i = 0; i < matrizDeCanais.length; i++) {
9      System.out.println(matrizDeCanais[i].nome);
10 }
```

No entanto é fácil perceber os problemas desta implementação sem ao menos ver o código, pois, caso outra equipe utilize outra estrutura para armazenar a lista de canais você deverá utilizar outro loop para imprimir. Pior ainda é se você precisar realizar outra operação com as listas de canais terá que implementar o mesmo método para cada uma das listas.

Ok, então vamos ver agora uma boa solução para esse problema.

Iterator

Vamos analisar qual a intenção do padrão Iterator:

“Fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor sua representação subjacente” [1]

Então utilizando o padrão Iterator nós poderemos acessar os elementos de um conjunto de dados sem conhecer sua implementação, ou seja, sem a necessidade de saber se será utilizado ArrayList ou Matriz. No nosso exemplo os objetos agregados seriam as listas de canais (ArrayList e Matriz).

Inicialmente vamos criar uma interface comum à todos os objetos agregados, ou seja uma lista genérica:


```

1 public interface AgregadoDeCanais {
2     IteradorInterface criarIterator();
3 }

```

Por ser genérica a nossa classe não possui nenhum detalhe da implementação da lista, ou seja, não possui uma ArrayList ou uma Matriz de Canais. A interface define apenas que todas as classes agregadas devem implementar um método de criação de Iterator (será discutido mais adiante).

Na nossa classe concreta nós utilizamos a lista de dados com uma implementação própria e implementamos o método de criação de Iterator, veja a seguir o exemplo da classe de canais que utiliza o ArrayList:

```

1 public class CanaisEsportes implements AgregadoDeCanais {
2
3     protected ArrayList<Canal> canais;
4
5     public CanaisEsportes() {
6         canais = new ArrayList<Canal>();
7         canais.add(new Canal("Esporte ao vivo"));
8         canais.add(new Canal("Basquete 2011"));
9         canais.add(new Canal("Campeonato Italiano"));
10        canais.add(new Canal("Campeonato Espanhol"));
11        canais.add(new Canal("Campeonato Brasileiro"));
12    }
13
14    @Override
15    public IteradorInterface criarIterator() {
16        return new IteradorListaDeCanais(canais);
17    }
18 }

```

O método de criação do Iterator retorna um iterador de Lista, que tem como conjunto de dados o ArrayList canais. Na classe que utiliza uma matriz para guardar os canais, o método de criação do Iterator retorna um iterador de Matriz.

```

1 @Override
2 public IteradorInterface criarIterator() {
3     return new IteradorMatrizDeCanais(canais);
4 }

```

Pronto, conseguimos encapsular os diferentes conjuntos de dados numa interface comum, agora qualquer nova lista que apareça precisa apenas implementar a interface que agrega canais. Vamos ver agora como será a implementação dos iteradores.

Da mesma maneira que criamos uma interface comum aos agregados vamos criar uma interface comum aos iteradores, assim podemos garantir que todo iterador tenha o mínimo de operações necessárias para percorrer o conjunto de dados.

```

1 public interface IteradorInterface {
2     void first();
3
4     void next();
5
6     boolean isDone();
7
8     Canal currentItem();
9 }

```

De maneira bem simples esta interface segue a recomendação do GoF [1]. Ou seja todo iterador possui um método que inicia o iterador (first), avança o iterador (next), verifica se já encerrou o percurso (isDone) e o que retorna o objeto atual (currentItem).

A implementação desses métodos será feita no iterador concreto, levando em consideração o tipo do conjunto de dados. Vamos mostrar primeiro a implementação do iterador do ArrayList. Apesar de já existir o Iterator de um ArrayList nativo do Java, vamos criar o nosso próprio Iterator.

```

1  public class IteradorListaDeCanais implements IteradorInterface {
2
3      protected ArrayList<Canal> lista;
4      protected int contador;
5
6      protected IteradorListaDeCanais(ArrayList<Canal> lista) {
7          this.lista = lista;
8          contador = 0;
9      }
10
11     public void first() {
12         contador = 0;
13     }
14
15     public void next() {
16         contador++;
17     }
18
19     public boolean isDone() {
20         return contador == lista.size();
21     }
22
23     public Canal currentItem() {
24         if (isDone()) {
25             contador = lista.size() - 1;
26         } else if (contador < 0) {
27             contador = 0;
28         }
29         return lista.get(contador);
30     }
31 }

```

A implementação do iterador é bem simples também. Os métodos alteram o contador do iterador, que marca qual o elemento está sendo visitado e, no método que retorna o objeto nós verificamos se o contador está dentro dos limites válidos e retornamos o objeto corrente.

A implementação do iterador de matriz também é bem simples e segue a mesma estrutura. Veja o código a seguir:

```

1  public class IteradorMatrizDeCanais implements IteradorInterface {
2      protected Canal[] lista;
3      protected int contador;
4
5      public IteradorMatrizDeCanais(Canal[] lista) {
6          this.lista = lista;
7      }
8
9      @Override
10     public void first() {
11         contador = 0;
12     }
13
14     @Override
15     public void next() {
16         contador++;
17     }
18
19     @Override
20     public boolean isDone() {
21         return contador == lista.length;
22     }
23
24     @Override
25     public Canal currentItem() {
26         if (isDone()) {
27             contador = lista.length - 1;
28         } else if (contador < 0) {
29             contador = 0;
30         }
31         return lista[contador];
32     }
33 }

```

Agora nós conseguimos encapsular também uma maneira de percorrer a lista de dados. Se um novo conjunto de dados for inseridos nós poderemos reutilizar iteradores (caso a estrutura do conjunto de dados seja a mesma), ou criar um novo iterador que implemente a interface básica dos iteradores.

O código cliente seria bem mais simples, veja:

```

1 public static void main(String[] args) {
2     AgregadoDeCanais canaisDeEsportes = new CanaisEsportes();
3     System.out.println("Canais de Esporte:");
4     for (IteradorInterface it = canaisDeEsportes.criarIterator(); !it
5         .isDone(); it.next()) {
6         System.out.println(it.currentItem().nome);
7     }
8
9     AgregadoDeCanais canaisDeFilmes = new CanaisFilmes();
10    System.out.println("\nCanais de Filmes:");
11    for (IteradorInterface it = canaisDeFilmes.criarIterator(); !it
12        .isDone(); it.next()) {
13        System.out.println(it.currentItem().nome);
14    }
15 }

```

O nosso código utiliza apenas as classes Interfaces, pois assim ficamos independentes de implementações concretas (obedecendo ao princípio de Design Orientado a Objetos Dependency inversion principle [2]).

Um pouco de teoria

A primeira observação a ser feita sobre o Iterator é que ele possui duas formas de implementação. A forma apresentada aqui é chamada de Iterator Externo, pois o cliente (código que utiliza a estrutura do Iterator) é responsável por fazer o percurso. No código mostrado no método main nós utilizamos um for para definir o percurso e as operações no conjunto de dados.

A outra implementação do Iterator é chamada de Interna, pois o código cliente não precisa se preocupar com o ciclo de vida do Iterator, apenas informa qual operação deve ser realizada. Essa implementação utiliza a seguinte ideia: Um classe abstrata implementa o método de percurso e realiza uma operação com cada um dos elementos do conjunto de dados.

```

1 public abstract class IteradorInterno {
2
3     IteradorInterface it;
4
5     public void percorrerLista() {
6         for (it.first(); !it.isDone(); it.next()) {
7             operacao(it.currentItem());
8         }
9     }
10
11     protected abstract void operacao(Canal canal);
12 }

```

Para informar qual operação deve ser executada nós criamos uma subclasse de IteradorInterno e nela implementamos a operação desejada.

```

1 public class IteradorPrint extends IteradorInterno {
2
3     public IteradorPrint(IteradorInterface it) {
4         this.it = it;
5     }
6
7     @Override
8     protected void operacao(Canal canal) {
9         System.out.println(canal.nome);
10    }
11
12 }

```

A execução do iterador interno apenas chama o método percorrerLista() e o iterador fica responsável por executar a operação com todos os objetos do conjunto de dados.

Cada uma das implementações possui efeitos colaterais diferentes, por exemplo, no Iterator Externo o cliente fica responsável por remover o iterador depois que ele for utilizado. No caso da linguagem Java, que possui um garbage collector, este não é um problema tão grande, mas em C++ por exemplo, precisamos tomar o cuidado de excluir o Iterator após seu uso.

Outro problema com o Iterator é que devemos ter uma atenção especial em qual operação o Iterator realizará, pois, caso ele altere, adicione ou remova objetos do conjunto de dados, temos que garantir que essa operação não invalidará os dados do conjunto. Como exemplo imagine que dois iterator são utilizados em paralelo, um deles vai mostrando os dados e o outro procura um elemento específico para removê-lo, o que acontece quando um Iterator acessa um objeto que outro removeu?

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 aí! 😊

Referências:

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

[2] WIKIPEDIA. SOLID. Disponível em: [http://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)). ([http://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design))). Acesso em: 15 set. 2011.

❏ [Iterator, Padrões de Projeto](#) ❏ [Iterator, Java, Padrões, Projeto](#) ❏ [3 Comentários](#)

3 comentários sobre “Mão na massa: Iterator”

1. ❏ [maio 25, 2017 às 12:59 AM](#)

Alexandre Salomão 

Olá Brizenol

Eu gostaria de saber se a assinatura do método criarIterator() na classe CanaisEsportes não deveria ser do tipo IteratorInterface?

Abraço!

❏ [Responder](#)

❏ [maio 25, 2017 às 6:42 AM](#)

Marcos Brizenol 

Oi Alexandre, sim boa observação! Na assinatura das funções sempre é recomendado utilizar uma interface (caso faça sentido) ao invés de uma implementação concreta.

Obrigado pelo comentário!

❏ [Responder](#)

❏ [maio 26, 2017 às 1:20 AM](#)

Alexandre Salomão 

Muito obrigado!

Os seus textos estão me ajudando bastante a compreender Padrões de Projetos!

Marcos Brizen

Desenvolvimento de Software #showmethecode

novembro 4, 2011

Mão na massa: Command

Problema

Suponha uma loja que vende produtos e oferece várias formas de pagamento. Ao executar uma compra o sistema registra o valor total e, dada uma forma de pagamento, por exemplo, cartão de crédito, emite o valor total da compra para o cartão de crédito do cliente.

Para este caso então vamos supor as seguintes classes para simplificar o exemplo: Loja e Compra. A classe Loja representa a loja que está efetuando a venda. Vamos deixar a classe Loja bem simples, como mostra o seguinte código:

```
1 public class Loja {
2     protected String nomeDaLoja;
3
4     public Loja(String nome) {
5         nomeDaLoja = nome;
6     }
7
8     public void executarCompra(double valor) {
9         Compra compra = new Compra(nomeDaLoja);
10        compra.setValor(valor);
11    }
12 }
```

Para focar apenas no que é necessário para entender o padrão, vamos utilizar a classe Compra, que representa o conjunto de produtos que foram vendidos com o seu valor total:

```
1 public class Compra {
2     private static int CONTADOR_ID;
3     protected int idNotaFiscal;
4     protected String nomeDaLoja;
5     protected double valorTotal;
6
7     public Compra(String nomeDaLoja) {
8         this.nomeDaLoja = nomeDaLoja;
9         idNotaFiscal = ++CONTADOR_ID;
10    }
11
12    public void setValor(double valor) {
13        this.valorTotal = valor;
14    }
15
16    public String getInfoNota() {
17        return new String("Nota fiscal nº: " + idNotaFiscal + "\nLoja: "
18            + nomeDaLoja + "\nValor: " + valorTotal);
19    }
20 }
```

Pronto, agora precisamos alterar a classe Loja para que ela, ao executar uma compra saiba qual a forma de pagamento.

Uma maneira interessante de fazer esta implementação seria adicionando um parâmetro a mais no método executar que nos diga qual forma de pagamento deve ser usada. Poderíamos então utilizar um Enum para identificar a forma de pagamento e daí passar a responsabilidade ao objeto específico. Veja o exemplo que mostra o código do método executar compra utilizando uma enumeração:

```

1 public void executarCompra(double valor, FormaDePagamento formaDePagamento) {
2     Compra compra = new Compra(nomeDaLoja);
3     compra.setValor(valor);
4     if(formaDePagamento == FormaDePagamento.CartaoDeCredito){
5         new PagamentoCartaoCredito().processarCompra(compra);
6     } else if(formaDePagamento == FormaDePagamento.CartaoDeDebito){
7         new PagamentoCartaoDebito().processarCompra(compra);
8     } else if(formaDePagamento == FormaDePagamento.Boleto){
9         new PagamentoBoleto().processarCompra(compra);
10    }
11 }

```

O problema desta solução é que, caso seja necessário incluir ou remover uma forma de pagamento precisaremos fazer várias alterações, alterando tanto a enumeração quando o método que processa a compra. Veja também a quantidade de ifs aninhados, isso é um sintoma de um design mal feito.

Poderíamos passar o objeto que faz o pagamento como um dos parametros, ao invés de utilizar a enumeração, assim não teríamos mais problemas com os ifs aninhados e as alterações seriam locais. Ok, está é uma boa solução, mas ainda não está boa, pois precisaríamos de um método diferente pra cada tipo de objeto.

A saída óbvia então é utilizar uma classe comum a todos as formas de pagamento, e no parâmetro passar um objeto genérico! Essa é justamente a ideia do Padrão Command.

Command

Intenção:

“Encapsular uma solicitação como objeto, desta forma permitindo parametrizar cliente com diferentes solicitações, enfileirar ou fazer o registro de solicitações e suportar operações que podem ser desfeitas.” [1]

Pela intenção vemos que o padrão pode ser aplicado em diversas situações. Para resolver o exemplo acima vamos encapsular as solicitações de pagamento de uma compra em objetos para parametrizar os clientes com as diferentes solicitações.

Perceba no código com os vários ifs outra boa oportunidade para refatorar o código. Dentro de cada um dos if, a ação é a mesma: criar um objeto para processar o pagamento e realizar uma chamada ao método de processamento da compra.

Então vamos primeiro definir a interface comum aos objetos que processam um pagamento. Todos eles possuem um mesmo método, processar pagamento, que toma como parâmetro uma compra e faz o processamento dessa compra de várias formas.

A classe interface seria a seguinte:

```

1 public interface PagamentoCommand {
2     void processarCompra(Compra compra);
3 }

```

Uma possível implementação seria a de processar um pagamento via boleto. Vamos apenas emitir uma mensagem no terminal para saber que tudo foi executado como esperado:

```

1 public class PagamentoBoleto implements PagamentoCommand {
2
3     @Override
4     public void processarCompra(Compra compra) {
5         System.out.println("Boleto criado!\n" + compra.getInfoNota());
6     }
7
8 }

```

Agora o método de execução da compra na classe Loja seria assim:

```

1 public void executarCompra(double valor, PagamentoCommand formaDePagamento) {
2     Compra compra = new Compra(nomeDaLoja);
3     compra.setValor(valor);
4     formaDePagamento.processarCompra(compra);
5 }

```

O código cliente que usaria o padrão Command seria algo do tipo:

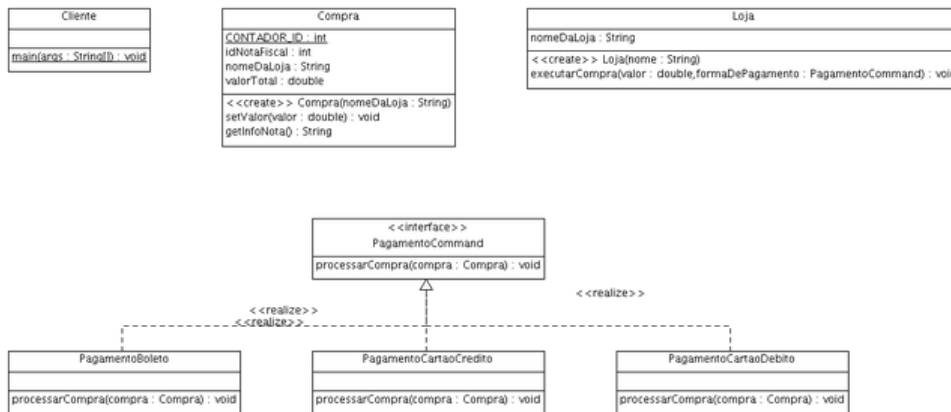
```

1 public static void main(String[] args) {
2     Loja lojasAfricanas = new Loja("Afriacanas");
3     lojasAfricanas.executarCompra(999.00, new PagamentoCartaoCredito());
4     lojasAfricanas.executarCompra(49.00, new PagamentoBoleto());
5     lojasAfricanas.executarCompra(99.00, new PagamentoCartaoDebito());
6
7     Loja exorbitante = new Loja("Exorbitante");
8     exorbitante.executarCompra(19.00, new PagamentoCartaoCredito());
9
10 }

```

Ao executar uma compra nós passamos o comando que deve ser utilizado, neste caso, a forma de pagamento utilizado.

O diagrama UML seria algo do tipo:



(<https://brizenofiles.wordpress.com/2011/11/command.png>)

Um pouco de teoria

O padrão tem um conceito muito simples, utiliza-se apenas da herança para agrupar classes e obrigar que todas tenham uma mesma interface em comum. A primeira vista o padrão pode ser confundido com o padrão [Template Method](http://wp.me/p1Mek8-1C) (<http://wp.me/p1Mek8-1C>), pois ambos utilizam a herança para unificar a interface de várias classes.

A diferença básica é que no padrão Command não existe a ideia de um “algoritmo” que será executado. No padrão Template Method as subclasses definem apenas algumas operações, sem alterar o esqueleto de execução. O padrão Command não oferece uma maneira de execução de suas subclasses, apenas garante que todas executem determinada requisição.

Como visto na intenção do padrão existem várias aplicações do padrão. Um exemplo seria oferecer um maior controle da execução de uma requisição. Por exemplo, caso fosse necessário primeiro armazenar todas as compras, cada uma com formas diferentes de pagamentos, para só então executar todas. O método de processar compra armazenaria os dados e, utilizando um outro método, por exemplo, finalizarCompra, todas as compras seriam processadas.

Ainda seguindo o exemplo, o padrão Command também poderia ser utilizado para desfazer operações, antes que elas sejam executadas de fato. Por exemplo, caso o cliente desista das compras, um método poderia desfazer/cancelar as alterações. Desta mesma forma poderíamos manter o controle sobre as mudanças, criando assim um log das operações.

As vantagens do padrão são: a facilidade de extensão da arquitetura, permitindo adicionar novos commands sem efeitos colaterais; e o bom nível de desacoplamento entre objetos, separando os objetos que possuem os dados dos que manipulam os dados.

Um problema que pode ocorrer ao utilizar o padrão Command é a complexidade dos comandos crescer demais. Por exemplo, se todos os commands precisam realizar várias ações, como: manter a persistência nas alterações, oferecer uma maneira de desfazer alterações, etc. Neste caso pode ser viável a utilização de um agrupamento de comandos, com o padrão [Composite](http://wp.me/p1Mek8-M) (<http://wp.me/p1Mek8-M>).

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 aí! 😊

Referências:

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

□ [Command](#) □ [Command, Java, Padrões, Projeto](#) □ [4 Comentários](#)

4 comentários sobre “Mão na massa: Command”

1. □ [abril 23, 2013 às 1:28 PM](#)

Klevlon Moraes

Muito obrigado por compartilhar o post e o conhecimento Marcos. Só fiquei com dúvida de quem é o receiver e o invoker no exemplo. E no caso a classe main de cliente seria o “Client” do padrão?
Vlw!

□ [Responder](#)

□ [abril 23, 2013 às 9:39 PM](#)

marcosbrizeno ↗

Fico grato em ter ajudado!

Com relação as suas duvidas:

O Invoker seria a classe Loja, pois ela é quem realiza as ações (chama o método executarCompra) passando o objeto Command.

O receiver seria a classe Compra, pois as ações são executadas nessa classe (no exemplo é apenas um sysout).

E sim, a classe Main é o cliente, pois ela é quem utiliza toda a infraestrutura que foi criada no exemplo.

□ [Responder](#)

□ [abril 25, 2013 às 8:15 AM](#)

Klevlon Moraes ↗

Obrigado pela explicação Marcos. Esclareceu bastante. Vou estudar o código para entender melhor o conceito do pattern.
Obrigado por responder e até logo.

2. □ [outubro 20, 2015 às 10:01 AM](#)

Leonardo Valentino

Obrigado por compartilhar explicações tão esclarecedoras sobre Design Patterns, tem me ajudado bastante!

□ [Responder](#)