Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Bridge

outubro 13, 2011

# Mão na massa: Bridge

O padrão deste post é o Bridge!

# ***Problema***

Como vimos no post anterior sobre o padrão Adapter, utilizamos o padrão Adapter para liberar o nosso código cliente de detalhes de implementação de Frameworks/API/Biblioteca específicas.

Suponha agora que é necessário fazer um programa que vá funcionar em várias plataformas, por exemplo, Windows, Linux, Mac, etc. O programa fará uso de diversas abstrações de janelas gráficas, por exemplo, janela de diálogo, janela de aviso, janela de erro, etc.

Como podemos representar esta situação? A utilização de um Adapter para adaptar as janelas para as diversas plataformas parece ser boa, criaríamos um Adapter para Windows, Linux e Mac e então utilizaríamos de acordo com a necessidade.

No entanto teríamos que utilizar o adaptador de cada uma das plataforma para cada um dos tipos de abstrações de janelas. Por exemplo, para uma janela de diálogo, teríamos um Adapter para Windows, Linux e Mac, da mesma forma para as outras janelas.

Vamos então partir para uma solução bem legal, o padrão Bridge!

# ***Bridge***

Intenção:

“Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente.” [1]

Ou seja, o Bridge fornece um nível de abstração maior que o Adapter, pois são separadas as implementações e as abstrações, permitindo que cada uma varie independentemente.

Para o exemplo as implementações seriam as classes de Janela das plataformas. Vamos iniciar construindo elas, de maneira bem simples. A primeira classe será a interface comum a todas as implementações, chamadas de JanelaImplementada:

```
1  public interface JanelaImplementada {
2
3      void desenharJanela(String titulo);
4
5      void desenharBotao(String titulo);
6
7  }
```

Ou seja, nessa classe definimos que todas as janelas desenham uma janela e um botão. Vamos ver agora a classe concreta que desenha a janela na plataforma Windows:

```
1   public class JanelaWindows implements JanelaImplementada {
2
3       @Override
4       public void desenharJanela(String titulo) {
5           System.out.println(titulo + " - Janela Windows");
6       }
7
8       @Override
9       public void desenharBotao(String titulo) {
10          System.out.println(titulo + " - Botão Windows");
11      }
12
13  }
```

Também uma classe bem simples, apenas vamos exibir uma mensagem no terminal para saber que tudo correu bem.

Agora vamos então para as abstrações. Elas são abstrações pois não definem uma janela específica, como a JanelaWindows, no entanto utilizarão os métodos destas janelas concretas para construir suas janelas.

Vamos então iniciar a construção da classe abstrata que vai fornecer uma interface de acesso comum para as abstrações de janelas:

```
public abstract class JanelaAbstrata {

    protected JanelaImplementada janela;

    public JanelaAbstrata(JanelaImplementada j) {
        janela = j;
    }

    public void desenharJanela(String titulo) {
        janela.desenharJanela(titulo);
    }

    public void desenharBotao(String titulo) {
        janela.desenharBotao(titulo);
    }

    public abstract void desenhar();

}
```

Essa classe possui uma referência para a interface das janelas implementadas, com isso conseguimos variar a implementação de maneira bem simples. Agora veja o exemplo da classe de JanelaDialogo, que abstrai uma janela de diálogo para todas as plataformas:

```
public class JanelaDialogo extends JanelaAbstrata {

    public JanelaDialogo(JanelaImplementada j) {
        super(j);
    }

    @Override
    public void desenhar() {
        desenharJanela("Janela de Diálogo");
        desenharBotao("Botão Sim");
        desenharBotao("Botão Não");
        desenharBotao("Botão Cancelar");
    }

}
```

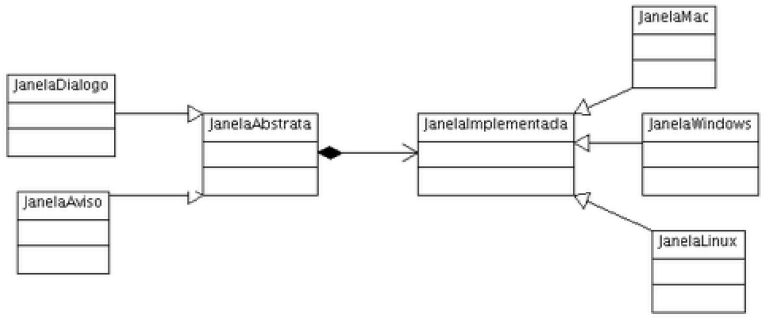
Uma janela de diálogo exibe sempre três botões: Sim, Não e Cancelar. Ou seja, independente de qual plataforma se está utilizando, a abstração é sempre a mesma. Para uma janela de aviso por exemplo, bastaria um botão Ok, então sua implementação seria algo do tipo:

```
public class JanelaAviso extends JanelaAbstrata {

    public JanelaAviso(JanelaImplementada j) {
        super(j);
    }

    @Override
    public void desenhar() {
        desenharJanela("Janela de Aviso");
        desenharBotao("Ok");
    }

}
```

Vamos ver então como seria o cliente da nossa aplicação. Ele fará uso apenas da classe que define uma Janela, assim poderá ficar livre de quaisquer detalhes de abstrações ou de implementações:

```
1  public static void main(String[] args) {
2      JanelaAbstrata janela = new JanelaDialogo(new JanelaLinux());
3      janela.desenhar();
4      janela = new JanelaAviso(new JanelaLinux());
5      janela.desenhar();
6
7      janela = new JanelaDialogo(new JanelaWindows());
8      janela.desenhar();
9  }
```

Note que é bem simples realizar trocas entre as abstrações de janelas e de plataformas. O diagrama UML para este exemplo seria algo do tipo:




# Um pouco de teoria

Como vimos pelo exemplo de codificação o Bridge provê um excelente nível de desacoplação dos componentes. Tanto novas abstrações como novas plataformas podem ser acomodadas pelo sistema sem grandes dificuldades, graças a extensibilidade do padrão.

Outro ponto que é comum a maioria dos padrões é a ocultação dos detalhes de implementação do cliente, que fica independente de qualquer variação ou extensão que precise ser feita.

Um ponto que merece um certo cuidado é sobre a instanciação dos objetos, pois vimos que é necessário especificar a abstração e utilizar uma implementação, assim o cliente precisa conhecer bem as classes, e o que elas realizam para saber exatamente o que, quando e como fazer.

## Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padroes-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.

Bridge, Padrões de Projeto  Bridge, Java, Padrões, Projeto  8 Comentários

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

outubro 3, 2011

# Mão na massa: Adapter

O padrão Adapter:

# ***Problema***

A utilização de frameworks é muito comum atualmente, devido a comodidade e facilidade de utilização. No entanto discute-se muito, principalmente na comunidade ágil, sobre a postura dos programadores frente aos frameworks.

O ponto de discussão é que a utilização de frameworks deve ser considerada algo fora do projeto original, e não um componente interno que torna todo o sistema dependente de um framework específico.

Vamos considerar o seguinte exemplo: é preciso fazer um sistema que manipule imagens, para isto será utilizado uma API que oferece essas funcionalidades. Suponhamos que será necessário ter um método para carregar a imagem de um arquivo e outro para exibir a imagem na tela.

Como podemos então construir o sistema de maneira que ele fique independente de qual API será utilizada?

Para exemplificar considere as duas classes a seguir que representam as APIs utilizadas. Como estas classes fazem parte da API não temos condições de alterá-las.

```
1  public class OpenGLImage {
2
3      public void glCarregarImagem(String arquivo) {
4          System.out.println("Imagem " + arquivo + " carregada.");
5      }
6
7      public void glDesenharImagem(int posicaoX, int posicaoY) {
8          System.out.println("OpenGL Image desenhada");
9      }
10 }
```

```
1  public class SDL_Surface {
2
3      public void SDL_CarregarSurface(String arquivo) {
4          System.out.println("Imagem " + arquivo + " carregada.");
5      }
6
7      public void SDL_DesenharSurface(int largura, int altura, int posicaoX,
8              int posicaoY) {
9          System.out.println("SDL_Surface desenhada");
10     }
11
12 }
```

Como podemos então, além de deixar o sistema independente, unificar uma interface de acesso para qualquer API? Veja que a assinatura dos métodos é bem diferente, desde o nome até quantidade de parâmetros.

Poderíamos utilizar o Template Method para definir uma interface e deixar cada subclasse redefinir algumas operações, no entanto não existe aqui a ideia de um "algoritmo comum".

Vejamos agora o que é o padrão Adapter.

# ***Adapter***

Intenção:

"Converter a interface de uma classe em outra interface, esperada pelo cliente. O Adapter permite que interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível." [1]

Ou seja, dado um conjunto de classes com mesma responsabilidade, mas interfaces diferentes, utilizamos o Adapter para unificar o acesso a qualquer uma delas.

Precisamos então, inicialmente, fornecer uma interface comum para o cliente, oferencendo o comportamento que ele necessita:

```
1  public interface ImagemTarget {
2      void carregarImagem(String nomeDoArquivo);
3
4      void desenharImagem(int posX, int posY, int largura, int altura);
5  }
```

Agora vamos definir os adaptadores:

```java
public class OpenGLImageAdapter extends OpenGLImage implements ImagemTarg

    @Override
    public void carregarImagem(String nomeDoArquivo) {
        glCarregarImagem(nomeDoArquivo);
    }

    @Override
    public void desenharImagem(int posX, int posY, int largura, int altur
        glDesenharImagem(posX, posY);
    }

}
```

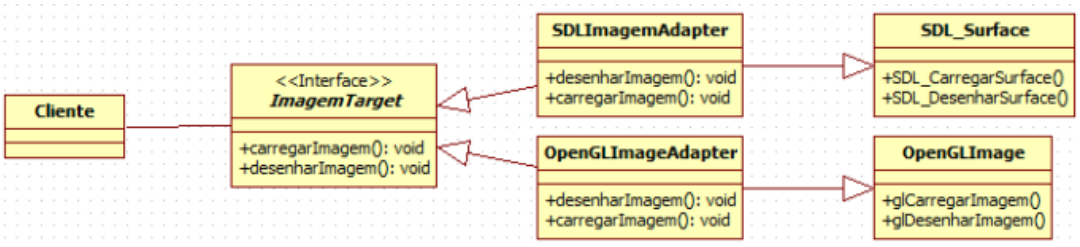
```java
public class SDLImagemAdapter extends SDL_Surface implements ImagemTarget

    @Override
    public void carregarImagem(String nomeDoArquivo) {
        SDL_CarregarSurface(nomeDoArquivo);
    }

    @Override
    public void desenharImagem(int posX, int posY, int largura, int altur
        SDL_DesenharSurface(largura, altura, posX, posY);
    }
}
```

De posse dos adaptadores, nosso cliente fica então independente de qual API será utilizada, utilizando apenas a Interface comum definida no início:

```java
public static void main(String[] args) {
    ImagemTarget imagem = new SDLImagemAdapter();
    imagem.carregarImagem("teste.png");
    imagem.desenharImagem(0, 0, 10, 10);

    imagem = new OpenGLImageAdapter();
    imagem.carregarImagem("teste.png");
    imagem.desenharImagem(0, 0, 10, 10);
}
```

Qualquer mudança em qual API será utilizada é facilmente feita, sem a necessidade de alterar o programa inteiro. Caso fosse necessário utilizar uma nova API também seria simples, bastaria criar o adaptador para a API e utilizá-lo quando fosse necessário.

Veja o diagrama UML abaixo:




(https://brizeno.files.wordpress.com/2011/10/adapter.png)

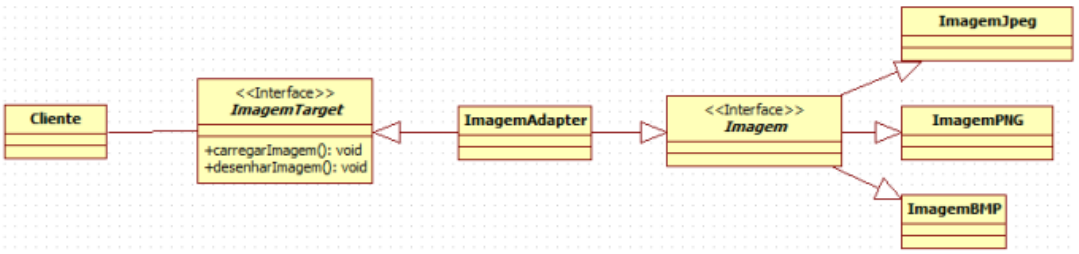
# Um pouco de teoria

O exemplo acima foi bem irreal, pois implementamos as funcionalidades para várias APIs diferentes, o que necessitou um Adapter para cada. Num exemplo real seria necessário apenas um Adapter para a API utilizada, e apenas este Adapter sofreria mudanças quando necessário. A não ser que o projeto realmente precise utilizar várias APIs diferentes.

Os adaptadores podem ser definidos de duas maneiras. Na maneira mostrada no exemplo o adaptador adaptou a Classe, herdando tanto da interface comum quanto da classe da API. Outra maneira seria utilizar um adaptador de Objeto, nele o adaptador instancia um objeto da API e, nas chamadas de métodos repassa a chamada para este objeto.

Cada implementação de adapter possui suas implicações. Um adaptador de classe não vai conseguir trabalhar com as subclasses da classe adaptada, já ao utiliza um adaptador de objeto é possível trabalhar com objetos de subclasses da classe adaptada.

Por exemplo, considere que uma API de imagem possui uma classe para cada tipo de imagem (JPEG, PNG, BMP, etc..) e que estas classes possuem uma interface comum:




(https://brizeno.files.wordpress.com/2011/10/adapter-invic3a1vel3.png)

Utilizar um adaptador de classe não seria viável pois precisaríamos de um adaptador para cada subclasse, já que herdando de uma classe genérica não adicionaria nada.

Já utilizando um adaptador de objeto bastaria tem uma referência para um objeto da interface e instanciar qualquer uma das subclasses.

Quão mais próximas forem as responsabilidades das classes adaptadas, mais simples será o adaptador. No exemplo citado as duas classes (OpenGLImage e SDL_Surface) possuiam a mesma responsabilidade, assim cada adaptador precisou apenas realizar a chamada necessária. Daí surge um problema com o uso do adapter: a diferença de interface.

Caso a diferença de responsabilidades seja muito grande, um adaptador pode acabar por subutilizar uma determinada classe, limitando sua interface ao que é comum as outras classes adaptadas.

Outro aspecto foi que, utilizando o padrão Adapter, deixamos a API como uma espécie de "plug-in" do sistema. O nosso cliente trata apenas com a interface, ficando independente de qual API é utilizada. Esse foi justamente o ponto tratado no início deste post. Utilizando este padrão as regras de nogócio de um sistema ficam independentes de detalhes de implementação.

## 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.

 Adapter, Padrões de Projeto  Adapter, Java, Padrões, Projeto  4 Comentários

# 4 comentários sobre “Mão na massa: Adapter”

1.  outubro 3, 2011 às 7:02 PM
   **Yuri Adams**
   É isso ai Marcos. Já está nos meus favoritos seu blog! Parabéns

    Responder
2.  novembro 4, 2015 às 3:10 PM
   **neycandidoribeiro**
   Caramba, que super blog! Cheguei quatro anos atrasado, mas antes tarde do que nunca.

    Responder
3.  abril 6, 2016 às 10:22 PM
   **Anônimo**
   Concordo com o colega acima, super blog mesmo!!! dicas valiosas e muito conhecimento existe aqui.

    Responder
4.  maio 31, 2016 às 11:48 AM
   **PAULO CESAR**
   valeu

   também agradeço

    Responder

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Composite

setembro 12, 2011

# Mão na massa: Composite

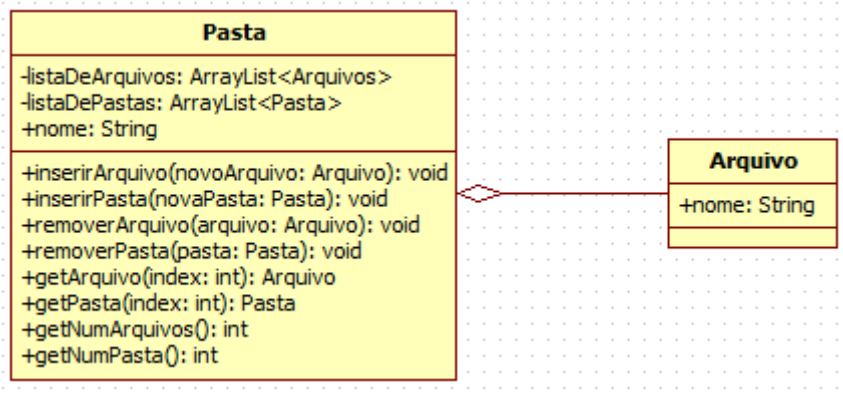
Desta vez vamos falar sobre o padrão Composite!

## *Problema*

Imagine que você está fazendo um sistema de gerenciamento de arquivos. Como você já sabe é possível criar arquivos concretos (vídeos, textos, imagens, etc.) e arquivos pastas, que armazenam outros arquivos. O problema é o mesmo, como fazer um design que atenda estes requerimentos?

## *Uma solução?*

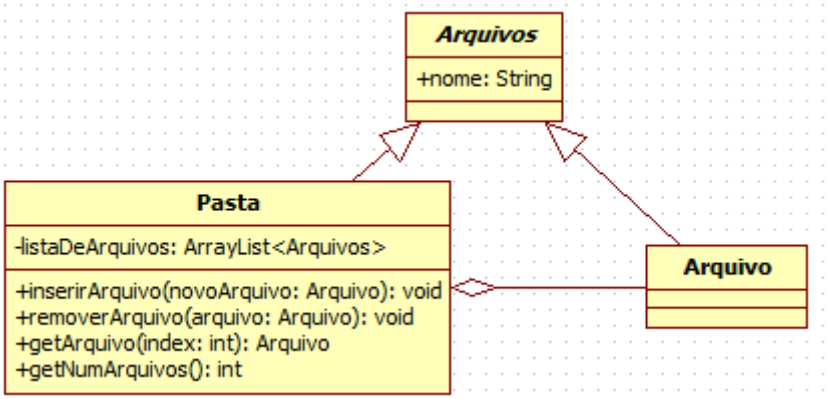
Podemos criar uma classe que representa arquivos que são Pastas, estas pastas teriam uma lista de arquivos concretos e uma lista de arquivos de pastas. Então poderíamos adicionar pastas e arquivos em uma pasta e, a partir dela navegar pelas suas pastas e seus arquivos.




O problema com este design é a interface que esta classe deverá ter. Como são duas listas diferentes precisaríamos de métodos específicos para tratar cada uma dela, ou seja, um método para inserir arquivos e outro para inserir pastas, um método para excluir pastas e outro para excluir arquivos, e assim vai.

Sempre que quisermos inserir uma nova funcionalidade no gerenciador precisaremos criar a mesma funcionalidade para arquivos e pastas. Além disso, sempre que quisermos percorrer uma pasta será necessário percorrer as duas listas, mesmo que vazias.

Bom, vamos pensar em outra solução. E se utilizássemos uma classe base Arquivo para todos os arquivos, assim precisaríamos apenas de uma lista e de um conjunto de funções. Vejamos abaixo:




Pronto, resolvido o problema dos métodos duplicados. E agora, será que está tudo bem? Como faríamos a diferenciação entre um Arquivo e uma Pasta? Poderíamos utilizar o "instace of" e verificar qual o tipo do objeto, o problema é que seria necessário fazer isso SEMPRE, pois não poderíamos confiar que, dado um objeto qualquer, ele é um arquivo ou uma pasta! Sempre teríamos que fazer isso:

```
1  if(arquivo instanceof Arquivo){
2      // Código para tratar arquivos de video
3  } else if(arquivo instanceof Pasta){
4      // Código para tratar arquivos de audio
5  }
```

Ok, então vamos ver uma boa solução para o problema: o padrão Composite!

# Composite

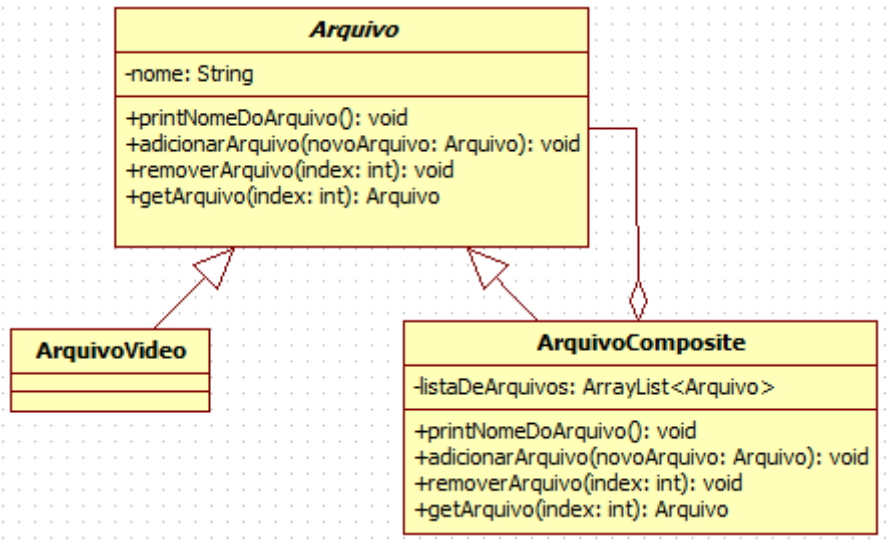
Vamos ver então qual a intenção do padrão Composite:

"Compor objetos em estruturas de árvore para representar hierarquia partes-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos." [1]

Bom, desta vez a intenção não está tão bem clara. A estrutura de árvore será explicada mais adiante, no momento o que interessa é a segunda parte da inteção: tratar de maneira uniforme objetos individuais.

Como o nosso problema era uniformizar o acesso aos arquivos e pastas, provavelmente o Composite seja uma boa solução.

A ideia do Composite é criar uma classe base que contém toda a interface necessária para todos os elementos e criar um elemento especial que agrega outros elementos. Vamos trazer para o nosso exemplo inicial para tentar esclarecer:




A classe base Arquivo implementa todos os métodos necessários para arquivos e pastas, no entanto considera como implementação padrão a do arquivo, ou seja, caso o usuário tente inserir um arquivo em outro arquivo uma exceção será disparada. Veja o código da classe abaixo:

```
public abstract class ArquivoComponent {

    String nomeDoArquivo;

    public void printNomeDoArquivo() {
        System.out.println(this.nomeDoArquivo);
    }

    public String getNomeDoArquivo() {
        return this.nomeDoArquivo;
    }

    public void adicionar(ArquivoComponent novoArquivo) throws Exception {
        throw new Exception("Não pode inserir arquivos em: "
                + this.nomeDoArquivo + " - Não é uma pasta");
    }

    public void remover(String nomeDoArquivo) throws Exception {
        throw new Exception("Não pode remover arquivos em: "
                + this.nomeDoArquivo + " -Não é uma pasta");
    }

    public ArquivoComponent getArquivo(String nomeDoArquivo) throws Exception {
        throw new Exception("Não pode pesquisar arquivos em: "
                + this.nomeDoArquivo + " - Não é uma pasta");
    }
}
```

Uma vez que tudo foi definido nesta classe, para criar um arquivo de vídeo por exemplo, basta implementar o construtor:

```
1  public class ArquivoVideo extends ArquivoComponent {
2
3      public ArquivoVideo(String nomeDoArquivo) {
4          this.nomeDoArquivo = nomeDoArquivo;
5      }
6  }
```

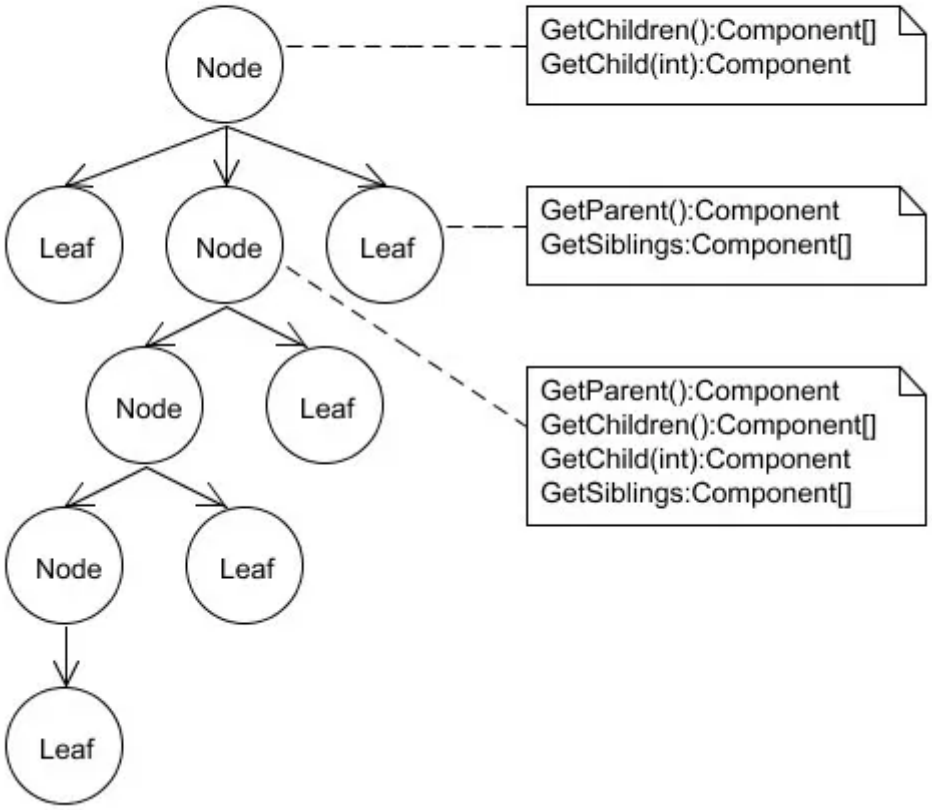
Já na classe que representa a Pasta nós sobrescrevemos o comportamento padrão e repassamos a chamada para todos os arquivos, sejam arquivos ou pastas, como podemos ver a seguir:

```java
public class ArquivoComposite extends ArquivoComponent {

    ArrayList<ArquivoComponent> arquivos = new ArrayList<ArquivoComponent

    public ArquivoComposite(String nomeDoArquivo) {
        this.nomeDoArquivo = nomeDoArquivo;
    }

    @Override
    public void printNomeDoArquivo() {
        System.out.println(this.nomeDoArquivo);
        for (ArquivoComponent arquivoTmp : arquivos) {
            arquivoTmp.printNomeDoArquivo();
        }
    }

    @Override
    public void adicionar(ArquivoComponent novoArquivo) {
        this.arquivos.add(novoArquivo);
    }

    @Override
    public void remover(String nomeDoArquivo) throws Exception {
        for (ArquivoComponent arquivoTmp : arquivos) {
            if (arquivoTmp.getNomeDoArquivo() == nomeDoArquivo) {
                this.arquivos.remove(arquivoTmp);
                return;
            }
        }
        throw new Exception("Não existe este arquivo");
    }

    @Override
    public ArquivoComponent getArquivo(String nomeDoArquivo) throws Excep
        for (ArquivoComponent arquivoTmp : arquivos) {
            if (arquivoTmp.getNomeDoArquivo() == nomeDoArquivo) {
                return arquivoTmp;
            }
        }
        throw new Exception("Não existe este arquivo");
    }

}
```

Com isto não é necessário conhecer a implementação dos objetos concretos, muito menos fazer cast. Veja como poderíamos utilizar o código do Composite:

```
1  public static void main(String[] args) {
2      ArquivoComponent minhaPasta = new ArquivoComposite("Minha Pasta/");
3      ArquivoComponent meuVideo = new ArquivoVideo("meu video.avi");
4      ArquivoComponent meuOutroVideo = new ArquivoVideo("serieS01E01.mkv");
5
6      try {
7          meuVideo.adicionar(meuOutroVideo);
8      } catch (Exception e) {
9          System.out.println(e.getMessage());
10     }
11
12     try {
13         minhaPasta.adicionar(meuVideo);
14         minhaPasta.adicionar(meuOutroVideo);
15         minhaPasta.printNomeDoArquivo();
16     } catch (Exception e) {
17         System.out.println(e.getMessage());
18     }
19
20     try {
21         System.out.println("\nPesquisando arquivos:");
22         minhaPasta.getArquivo(meuVideo.getNomeDoArquivo())
23                 .printNomeDoArquivo();
24         System.out.println("\nRemover arquivos");
25         minhaPasta.remover(meuVideo.getNomeDoArquivo());
26         minhaPasta.printNomeDoArquivo();
27     } catch (Exception e) {
28         e.printStackTrace();
29     }
30 }
```

Agora podemos visualizar a tal estrutura de árvore, suponha que temos pastas dentro de pastas com arquivos, a estrutura seria parecida com a de uma árvore, veja a seguir:




Como uma estrutura de árvore temoas Nós e Folhas. No padrão Composite os arquivos concretos do nosso exemplo são chamados de Folhas, pois não possuem filhos e os arquivos pasta são chamados de Nós, pois possuem filhos e fornecem operações sobre esses filhos.

# *Um pouco de teoria*

Bom, vimos como o padrão pode ser utilizado, vamos agora explorar mais um pouco sua teoria.

A primeira vantagem, e talvez a mais forte, seja o fato de os clientes do código Composite serem bem simplificados, pois podem tratar todos os objetos da mesma maneira. No nosso exemplo utilizei exceções para que ficasse mais evidente quando um método de uma Pasta é chamado em um Arquivo, mas suponha que os métodos não fizessem nada, a utilização seria mais simplificada ainda, pois não precisaríamos de blocos try e catch.

No entanto, o mal tratamento destas exceções podem gerar problemas de segurança e ai surge uma outra forma de implementar o padrão, restringindo a interface comum dos objetos. Para isto basta remover os métodos de gerenciamento de arquivos (adicionar, remover, etc) da classe base, assim apenas os arquivos pastas teriam estes métodos.

Em contrapartida o usuário do código precisa ter certeza se um dado objeto é Pasta para realizar um cast e chamar os métodos da pasta. Veja o método main que utiliza esta implementação:

```
1  public static void main(String[] args) {
2          ArquivoComponent meuVideo = new ArquivoVideo("meu video.rmvb");
3          ArquivoComponent meuOutroVideo = new ArquivoVideo("novo video.rmvb
4          ArquivoComponent minhaPasta = new ArquivoComposite("minha pasta/")
5
6          ((ArquivoComposite) minhaPasta).adicionar(meuVideo);
7          ((ArquivoComposite) minhaPasta).adicionar(meuOutroVideo);
8          minhaPasta.printNomeDoArquivo();
9      }
```

Perceba que precisamos utilizar um cast para fazer a chamada aos métodos da pasta. Caso o objeto não fosse uma pasta de fato teríamos problemas. De acordo com suas necessidades você deve optar qual implementação utilizar.

No repositório do Git (link abaixo) você encontrará as duas implementações, em package separados.

## *Código fonte completo*

O código completo pode ser baixado no seguinte repositório Git: 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.

 Composite, Padrões de Projeto  Composite, Java, Padrões, Projeto  7 Comentários

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Decorator

agosto 31, 2011

# Mão na massa: Decorator

Continuando com a série de post sobre padrões de projeto na prática, vamos falar agora sobre o padrão Decorator.

# ***Problema***

Imagine que você está desenvolvendo um sistema para um bar especializado em coquetéis, onde existem vários tipos de coquetéis que devem ser cadastrados para controlar a venda. Os coquetéis são feitos da combinação de uma bebida base e vários outros adicionais que compõe a bebida. Por exemplo:

Conjunto de bebidas:

- Cachaça
- Rum
- Vodka
- Tequila

Conjunto de adicionais:

- Limão
- Refrigerante
- Suco
- Leite condensado
- Gelo
- Açúcar

Então, como possíveis coquetéis temos:

- Vodka + Suco + Gelo + Açúcar
- Tequila + Limão + Sal
- Cachaça + Leite Condensado + Açúcar + Gelo

E então, como representar isto em um sistema computacional?

# Uma solução?

Bom, poderíamos utilizar como uma solução simples uma classe abstrata Coquetel extremamente genérica e, para cada tipo de coquetel construir uma classe concreta.

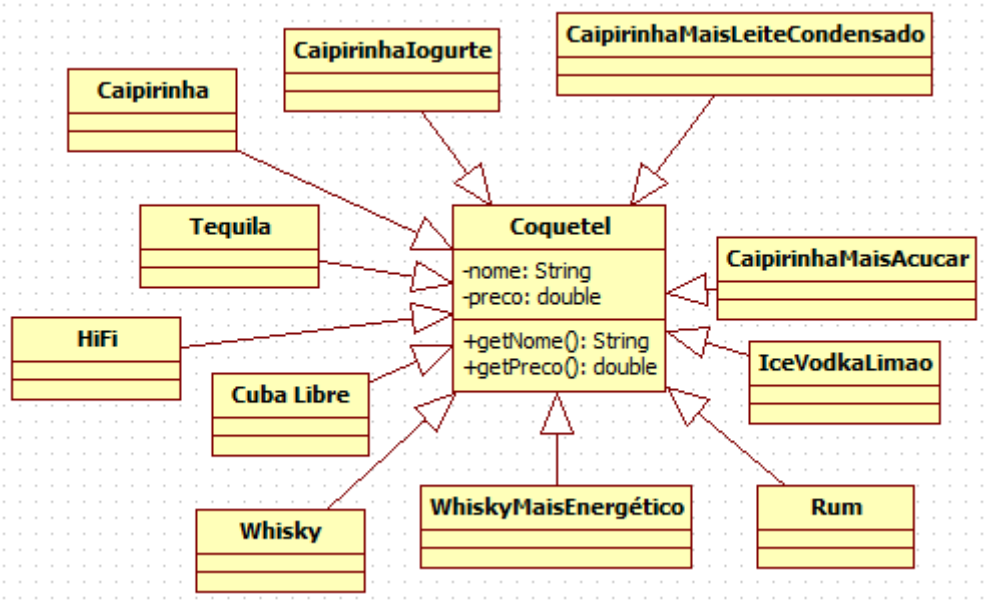
Então teríamos a classe base Coquetel:

```
1  public abstract class Coquetel {
2      String nome;
3      double preco;
4
5      public String getNome() {
6          return nome;
7      }
8
9      public double getPreco() {
10         return preco;
11     }
12 }
```

A nossa classe define apenas o nome e o preço da bebida para facilitar a exemplificação. Uma classe coquetel concreta seria, por exemplo, a Caipirinha:

```
1  public class Caipirinha extends Coquetel {
2      public Caipirinha() {
3          nome = "Caipirinha";
4          preco = 3.5;
5      }
6  }
```

No entanto, como a especialidade do bar são coquetéis, o cliente pode escolher montar seu próprio coquetel com os adicionais que ele quiser. De acordo com nosso modelo teríamos então que criar várias classes para prever o que um possível cliente solicitaria! Imagine agora a quantidade de combinações possíveis? Veja o diagrama UML abaixo para visualizar o tamanho do problema:



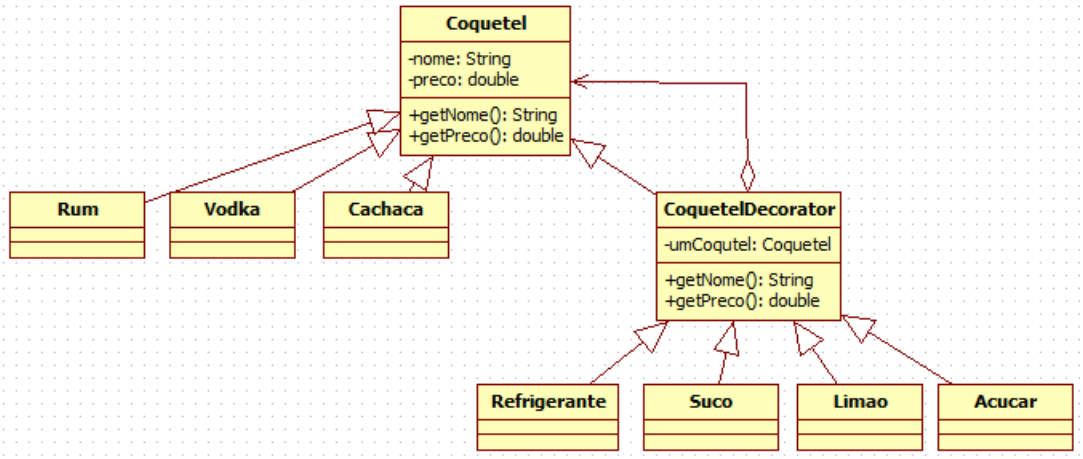

Além disso, pode ser que o cliente deseje adicionar doses extras de determinados adicionais, desse modo não seria possível modelar o sistema para prever todas as possibilidades! Então, como resolver o problema?

# ***Decorator***

Vamos ver qual a Intenção do padrão Decorator:

"Dinamicamente, agregar responsabilidades adicionais a objetos. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades." [1]

Perfeito, exatamente a solução do nosso problema! Queremos que, dado um objeto Coquetel, seja possível adicionar funcionalidades a ele, e somente a ele, em tempo de execução. Vamos ver a arquitetura sugerida pelo padrão:




Certo, então todos os objetos possuem o mesmo tipo Coquetel, esta classe define o que todos os objeto possuem e é igual a classe já feita antes. As classes de bebidas concretas definem apenas os dados relativos a ela. Como exemplo vejamos o código da bebida Cachaça:

```
public class Cachaca extends Coquetel {
    public Cachaca() {
        nome = "Cachaça";
        preco = 1.5;
    }
}
```

Todas as classes de bebidas possuirão a mesma estrutura, apenas definem os seus atributos. A classe Decorator abstrata define que todos os decoradores possuem um objeto Coquetel, ao qual decoram, e um método que é aplicado a este objeto. Vejamos o código para exemplificar:

```
public abstract class CoquetelDecorator extends Coquetel {
    Coquetel coquetel;

    public CoquetelDecorator(Coquetel umCoquetel) {
        coquetel = umCoquetel;
    }

    @Override
    public String getNome() {
        return coquetel.getNome()  + " + " + nome;
    }

    public double getPreco() {
        return coquetel.getPreco() + preco;
    }
}
```

Lembre-se de que como o decorador também é um Coquetel ele herda os atributos nome e preço. Nas classes concretas apenas definimos os modificadores que serão aplicados, de maneira semelhante as classes de bebidas concretas, vejamos o exemplo do adicional Refrigerante:

```
public class Refrigerante extends CoquetelDecorator {

    public Refrigerante(Coquetel umCoquetel) {
        super(umCoquetel);
        nome = "Refrigerante";
        preco = 1.0;
    }

}
```

Perceba que no construtor do decorador é necessário passar um objeto Coquetel qualquer, este objeto pode ser tanto uma bebida quanto outro decorador. Ai está o conceito chave para o padrão Decorator. Vamos acrescentando vários decoradores em qualquer ordem em uma bebida. Vamos ver agora como o padrão seria utilizado, veja o seguinte código do método main:

```
public static void main(String[] args) {
        Coquetel meuCoquetel = new Cachaca();
        System.out.println(meuCoquetel.getNome() + " = "
                + meuCoquetel.getPreco());

        meuCoquetel = new Refrigerante(meuCoquetel);
        System.out.println(meuCoquetel.getNome() + " = "
                + meuCoquetel.getPreco());
    }
```

Perceba que o tipo do coquetel varia de acordo com o decorador aplicado. Então, quando chamamos o método getNome ou getPreco o primeiro método chamado é o método do último decorador aplicado.

O método do decorador por sua vez chama o método da classe mãe, este método então chama o método do Coquetel ao qual ele decora. Se esse coquetel for outro decorador o pedido é repassado até chegar a um coquetel que é uma bebida de fato e finalmente responde a requisição sem repassar a nenhum outro objeto.

De maneira semelhante a recursão, os valores calculados vão sendo retornados até chegar no último decorador aplicado e então são repassados ao objeto. É como se os decoradores englobassem tanto outros decoradores quanto o componente em si.

# Um pouco de teoria

Como já dito o padrão Decorator adiciona funcionalidades ao objeto em tempo de execução. Note bem que, ao contrário da herança que aplica funcionalidades a todos os objetos dela, o padrão decorator permite aplicar funcionalidades apenas a um objeto específico.

Justamente devido a essa propriedade é que o padrão Decorator possui uma flexibilidade maior que a herança estática. Além disso, como o Decorator aplica apenas as funcionalidades necessárias ao objeto nós evitamos o problema de classes sobrecarregadas, que possuem funcionalidade que nunca são utilizadas.

# Problemas com o Decorator

No entanto este comportamento altamente dinâmico do Decorator traz alguns problemas, como por exemplo, dado um objeto Coquetel não é possível verificar se ele possui um decorador Limão, Refrigerante ou qualquer outro. Assim, caso cada um dos decoradores implementasse outros métodos específicos, um desenvolvedor que utilizasse um coquetel qualquer não possui nenhuma garantia sobre o tipo do coquetel. Por exemplo, se o decorador Limão implementa um método arder(), não é possível afirmar que um coquetel qualquer possui este método.

O problema é pior ainda pois não é possível sequer verificar o tipo do coquetel, por exemplo, adicione este código no final do método main:

```
1  System.out.println(meuCoquetel instanceof Cachaca);
```

Será exibido no console "false" pois, como aplicamos o decorador Refrigerante modificamos o tipo do coquetel.

Além disso é necessário criar vários pequenos objetos, que possuem o mesmo comportamento, para criar os coquetéis necessários. No primeiro modelo apresentado teríamos várias classes, mas apenas um objeto para representar um Coquetel. Utilizando a estrutura do Decorator precisamos criar um objeto para cada novo decorador, além do objeto bebida.

## Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: 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.

 Decorator, Padrões de Projeto  Decorator, Java, Padrões, Projeto  10 Comentários

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Facade

novembro 17, 2011

# Mão na massa: Facade

## ***Problema:***

No desenvolvimento de jogos é comum a utilização de subsistemas de um Framework/API. Por exemplo, para reproduzir um determinado som é utilizado o subsistema de Audio, que normalmente provê funcionalidades desde a configuração da reprodução de audio, até a reprodução de um determinado arquivo.

Antes de iniciar o jogo de fato é necessário realizar ajustes em todos os subsistemas que serão utilizados, por exemplo, é necessário configurar a resolução do subsistema de Video para que este possa renderizar imagens corretamente.

Para exemplificar veja as interfaces da seguintes classes, que representa o subsistema de Audio:

```java
public class SistemaDeAudio {

    public void configurarFrequencia() {
        System.out.println("Frequencia configurada");
    }

    public void configurarVolume() {
        System.out.println("Volume configurado");
    }

    public void configurarCanais() {
        System.out.println("Canais configurados");
    }

    public void reproduzirAudio(String arquivo) {
        System.out.println("Reproduzindo: " + arquivo);
    }
}
```

Como falado ela fornece os métodos para configuração e reprodução de arquivos de audio. Para reproduzir um arquivo de audio, por exemplo, seria necessário realizar as seguintes operações:

```java
public static void main(String[] args) {
    System.out.println("##### Configurando subsistemas #####");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();

    System.out.println("##### Utilizando subsistemas #####");
    audio.reproduzirAudio("teste.mp3");
}
```

Neste exemplo de código cliente, o próprio cliente deve instanciar e configurar o subsistema para que só depois seja possível a utilização dos mesmos. Além disso, existe um comportamento padrão que é executado antes de reproduzir um som: sempre deve ser configurado o canal, a frequência e o volume.

Agora pense como seria caso fosse necessário utilizar vários subsistemas? O código cliente ficaria muito sobrecarregado com responsabilidades que não são dele:

```java
public static void main(String[] args) {
    System.out.println("##### Configurando subsistemas #####");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();

    SistemaDeInput input = new SistemaDeInput();
    input.configurarTeclado();
    input.configurarJoystick();

    SistemaDeVideo video = new SistemaDeVideo();
    video.configurarCores();
    video.configurarResolucao();

    System.out.println("##### Utilizando subsistemas #####");
    audio.reproduzirAudio("teste.mp3");
    input.lerInput();
    video.renderizarImagem("imagem.png");
}
```

Vamos ver então como o padrão Facade pode resolver este pequeno problema.

# ***Facade***

A intenção do padrão:

“Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Facade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.” [1]

Pela intenção é possível notar que o padrão pode ajudar bastante na resolução do nosso problema. O conjunto de interfaces seria exatamente o conjunto de subsistemas. Como falado em [1] um subsistema é análogo a uma classe, uma classe encapsula estados e operações, enquanto um subsistema encapsula classes.

Nesse sentido o Facade vai definir operações a serem realizadas com estes subsistemas. Assim, é possível definir uma operação padrão para configurar o subsistema de audio, evitando a necessidade de chamar os métodos de configuração de audio a cada novo arquivo de audio que precise ser reproduzido.

A utilização do padrão Facade é bem simples. apenas é necessário criar a classe fachada que irá se comunicar com os subsistemas no lugar no cliente:

```
1  public class SistemasFacade {
2      protected SistemaDeAudio audio;
3      protected SistemaDeInput input;
4      protected SistemaDeVideo video;
5
6      public void inicializarSubsistemas() {
7          video = new SistemaDeVideo();
8          video.configurarCores();
9          video.configurarResolucao();
10
11         input = new SistemaDeInput();
12         input.configurarJoystick();
13         input.configurarTeclado();
14
15         audio = new SistemaDeAudio();
16         audio.configurarCanais();
17         audio.configurarFrequencia();
18         audio.configurarVolume();
19     }
20
21     public void reproduzirAudio(String arquivo) {
22         audio.reproduzirAudio(arquivo);
23     }
24
25     public void renderizarImagem(String imagem) {
26         video.renderizarImagem(imagem);
27     }
28
29     public void lerInput() {
30         input.lerInput();
31     }
32
33 }
```

A classe fachada realiza a inicialização de todos os subsistemas e oferece acesso aos métodos necessários, por exemplo o método de renderização de uma imagem, a reprodução de um audio.

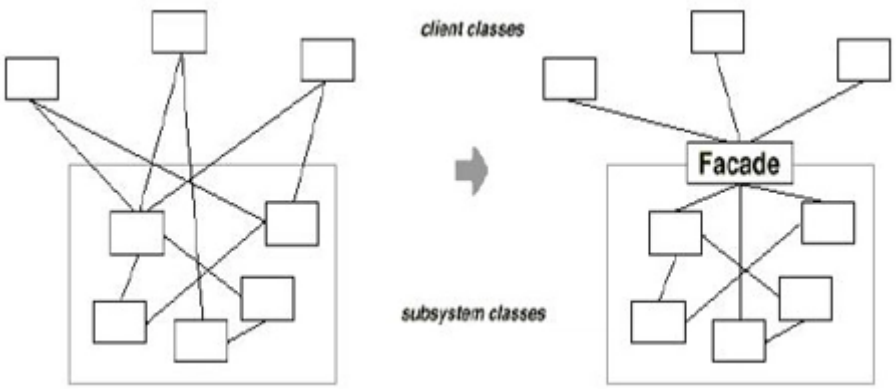
Com esta mudança, tiramos toda a responsabilidade do cliente, que agora precisa se preocupar apenas em utilizar os subsistemas que desejar.

```
1  public static void main(String[] args) {
2      System.out.println("##### Configurando subsistemas #####");
3      SistemasFacade fachada = new SistemasFacade();
4      fachada.inicializarSubsistemas();
5
6      System.out.println("##### Utilizando subsistemas #####");
7      fachada.renderizarImagem("imagem.png");
8      fachada.reproduzirAudio("teste.mp3");
9      fachada.lerInput();
10 }
```

A utilização do padrão é bem simples e poder ser aplicado em várias situações.

# Um pouco de teoria

A ideia básica do padrão, como visto, é remover a complexidade das classes clientes. Visualmente falando, ele faz o seguinte:




Note que as classes do subsistema continuam sendo visíveis em todo o projeto. Portanto, caso seja necessário, o cliente pode definir suas configurações sem sequer utilizar a classe fachada. Ainda mais, o cliente pode criar uma fachada própria, que define suas operações customizadas. Por exemplo, se não serão utilizados joysticks no projeto, não há necessidade de inicializá-los.

O problema com essa centralização da complexidade é que a classe fachada pode crescer descontroladamente para abrigar uma conjunto grande de possibilidades. Nestes casos pode ser mais viável procurar outros padrões, como Abstract Factory para dividir as responsabilidades entre subclasses.

Existem algumas semelhanças, fáceis de serem notadas, deste padrão com outros já discutidos aqui. Por exemplo, já que o Facade define uma interface, qual a sua diferença em relação ao padrão Adapter, já que ambos definem uma nova interface? A diferença básica é que o Adapter adapta uma interface antiga para uma outra interface enquanto que o Facade cria uma interface completamente nova, que é mais simples.

Outra semelhança que também pode ser notada é com o padrão Mediator, já que ambos reduzem a complexidade entre um grande conjunto de objetos. A diferença é que como o padrão Mediator centraliza a comunicação entre os objetos colegas, normalmente adicionando novas funcionalidades e sendo referenciado por todos os colegas. No padrão Facade, a classe fachada apenas utiliza os subsistemas, sem adicionar nada, além disso as classes do subsistema não sabem nada sobre a fachada.

## *Código fonte completo*

O código completo pode ser baixado no seguinte repositório Git: 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.

 Facade  Facade, Java, Padrões, Projeto  5 Comentários

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Flyweight

novembro 13, 2011

# Mão na massa: Flyweight

## *Problema:*

No desenvolvimento de jogos são utilizadas várias imagens. Elas representam as entidades que compõe o jogo, por exemplo, cenários, jogadores, inimigos, entre outros. Ao criar classes que representam estas entidades, é necessário vincular a elas um conjunto de imagens, que representam as animações.

Quem desenvolve jogos pode ter pensado na duplicação de informação quando as imagens são criadas pelos objetos que representam estas entidades, por exemplo, a classe que representa um inimigo carrega suas imagens. Quando são exibidos vários inimigos do mesmo tipo na tela, o mesmo conjunto de imagens é criado repetidamente.

A solução para esta situação de duplicação de informações pelos objetos é a utilização do padrão Flyweight.

## *Flyweight*

A intenção do padrão:

"Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina." [1]

Pela intenção percebemos que o padrão Flyweight cria uma estrutura de compartilhamento de objetos pequenos. Para o exemplo citado, o padrão será utilizado no compartilhamento de imagens entre as entidades.

Antes de exemplificar vamos entende um pouco sobre a estrutura do padrão. A classe Flyweight fornece uma interface com uma operação que deve ser realizado sobre um estado interno. No exemplo esta classe irá fornecer uma operação para desenhar a imagem em um determinado ponto. Desta forma a imagem é o estado intrínseco, que consiste de uma informação que não depende de um contexto externo. O ponto passado como parâmetro é o estado extrínseco, que varia de acordo com o contexto.

Vamos então ao código da imagem e do ponto:

```
1  public class Imagem {
2      protected String nomeDaImagem;
3
4      public Imagem(String imagem) {
5          nomeDaImagem = imagem;
6      }
7
8      public void desenharImagem() {
9          System.out.println(nomeDaImagem + " desenhada!");
10     }
11 }
```

Para simplificar o exemplo, será apenas exibida uma mensagem no terminal, indicando que a imagem foi desenhada.

```
1  public class Ponto {
2      public int x, y;
3
4      public Ponto(int x, int y) {
5          this.x = x;
6          this.y = y;
7      }
8  }
```

A classe Flyweight vai apenas fornecer a interface para desenho da imagem em um ponto.

```
1  public abstract class SpriteFlyweight {
2      public abstract void desenharImagem(Ponto ponto);
3  }
```

Outro componente da estrutura do Flyweight é a classe Flyweight concreta, que implementa a operação de faot:

```
1  public class Sprite extends SpriteFlyweight {
2      protected Imagem imagem;
3
4      public Sprite(String nomeDaImagem) {
5          imagem = new Imagem(nomeDaImagem);
6      }
7
8      @Override
9      public void desenharImagem(Ponto ponto) {
10         imagem.desenharImagem();
11         System.out.println("No ponto (" + ponto.x + "," + ponto.y + ")!")
12     }
13 }
```

Nesta classe também será apenas exibida uma mensagem no terminal para dizer que a imagem foi desenhada no ponto dado. O próximo componente da estrutura do Flyweight consiste em uma classe fábrica, que vai criar os vários objetos flyweight que serão compartilhados.

```
1  public class FlyweightFactory {
2
3      protected ArrayList<SpriteFlyweight> flyweights;
4
5      public enum Sprites {
6          JOGADOR, INIMIGO_1, INIMIGO_2, INIMIGO_3, CENARIO_1, CENARIO_2
7      }
8
9      public FlyweightFactory() {
10         flyweights = new ArrayList<SpriteFlyweight>();
11         flyweights.add(new Sprite("jogador.png"));
12         flyweights.add(new Sprite("inimigo1.png"));
13         flyweights.add(new Sprite("inimigo2.png"));
14         flyweights.add(new Sprite("inimigo3.png"));
15         flyweights.add(new Sprite("cenario1.png"));
16         flyweights.add(new Sprite("cenario2.png"));
17     }
18
19     public SpriteFlyweight getFlyweight(Sprites jogador) {
20         switch (jogador) {
21         case JOGADOR:
22             return flyweights.get(0);
23         case INIMIGO_1:
24             return flyweights.get(1);
25         case INIMIGO_2:
26             return flyweights.get(2);
27         case INIMIGO_3:
28             return flyweights.get(3);
29         case CENARIO_1:
30             return flyweights.get(4);
31         default:
32             return flyweights.get(5);
33         }
34     }
35 }
```

Além de criar os vários objetos a serem compartilhados, a classe fábrica oferece um método para obter o objeto, assim, o acesso a estes objetos fica centralizado e unificado a partir desta classe.

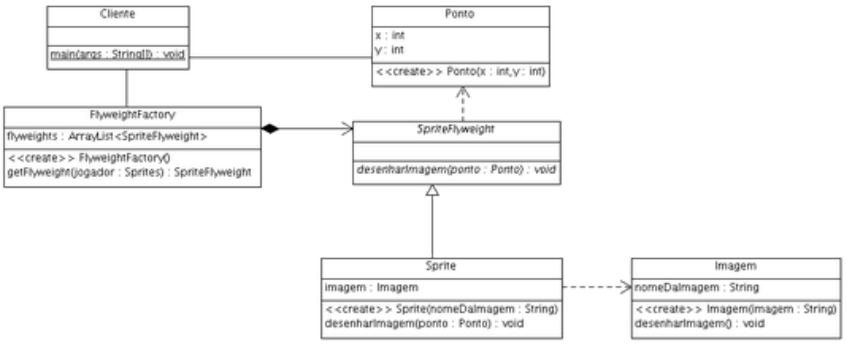
Para exemplificar a utilização do padrão, vejamos o seguinte código cliente:

```
1  public static void main(String[] args) {
2      FlyweightFactory factory = new FlyweightFactory();
3
4      factory.getFlyweight(Sprites.CENARIO_1).desenharImagem(new Ponto(0, 0
5
6      factory.getFlyweight(Sprites.JOGADOR).desenharImagem(new Ponto(10, 10
7
8      factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
9              new Ponto(100, 10));
10     factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
11             new Ponto(120, 10));
12     factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
13             new Ponto(140, 10));
14
15     factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(
16             new Ponto(60, 10));
17     factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(
18             new Ponto(50, 10));
19
20     factory.getFlyweight(Sprites.INIMIGO_3).desenharImagem(
21             new Ponto(170, 10));
22 }
```

É exibido um conjunto de imagens para exemplificar o uso em um jogo. São desenhados inimigos de vários tipos, o cenário do jogo e o jogador. Note que o acesso aos objetos fica centralizado apenas na classe fábrica.

No desenvolvimento de jogos real, as referências dos objetos seriam espalhadas pelas entidades, garantindo a não duplicação de conteúdo.

O diagrama UML que representa esta implementação é o seguinte:




# Um pouco de teoria

A solução implementada pelo padrão Flyweight é bem intuitiva. No entanto vale a pena comentar alguns detalhes. Percebeu que, na classe fábrica fica centralizado o acesso a todos os objetos compartilhados? O aconteceria se houvessem duas ou mais instâncias desta classe? Seriam criados vários objetos, sem nenhuma necessidade.

Para evitar este problema vale a pena dar uma olhada em outro padrão, o Singleton. Aplicando este padrão na classe fábrica, garantimos que apenas uma instância dela será utilizada em todo o projeto.

O ponto fraco do padrão é que, dependendo da quantidade e da organização dos objetos a serem compartilhados, pode haver um grande custo para procura dos objetos compartilhados. Então ao utilizar o padrão deve ser analisado qual a prioridade no projeto, espaço ou tempo de execução.

Imagine que existe um grupo de objetos que serão compartilhados juntos, por exemplo, uma sequência de objetos do cenário. Nesta situação, existe uma combinação com outro padrão, o Composite. Com ele é possível agrupar um conjuntos de objetos Flyweight que serão compartilhados juntos.

Outro ponto de interesse é a instanciação de todos os objetos flyweight na classe fábrica. Suponha que algum objeto é instanciado, mas nunca é utilizado? Pode ser implementado uma estratégia de garbage collection, que controla o número de instâncias de um determinado objeto. Ao não ser mais utilizado, o objeto é liberado da memória, reduzindo mais ainda o espaço.

## *Código fonte completo*

O código completo pode ser baixado no seguinte repositório Git: 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.

 Flyweight  Flyweight, Java, Padrões, Projeto  2 Comentários

__PRESENT

Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Proxy

outubro 1, 2011

# Mão na massa: Proxy

## ***Problema***

Suponha que um determinado programa faz uma conexão com o banco de dados para pegar algumas informações relativas aos usuários do sistema. Para simplificar o exemplo, considere a seguinte classe:

```java
public class BancoUsuarios{
    private int quantidadeDeUsuarios;
    private int usuariosConectados;

    public BancoUsuarios() {
        quantidadeDeUsuarios = (int) (Math.random() * 100);
        usuariosConectados = (int) (Math.random() * 10);
    }

    public String getNumeroDeUsuarios() {
        return new String("Total de usuários: " + quantidadeDeUsuarios);
    }

    public String getUsuariosConectados() {
        return new String("Usuários conectados: " + usuariosConectados);
    }
}
```

Nesta classe nós consultamos o número de usuários que estão conectados e o total de usuários do sistema. Poderiam existir diversas outras operações de consulta ao banco de dados, apenas simplificamos o exemplo.

O que queremos é implementar uma nova funcionalidade que permite verificar se um usuário possui ou não permissão para visualizar as informações do banco. A classe nos fornece uma maneira de acessar o banco de dados do sistema, no entanto ela não possui nenhuma proteção sobre quem está tentando acessá-la. Como podemos implementar um mecanismo de proteção?

Uma primeira solução seria adicionar os campos de usuário e senha e verificar se quem está tentando acessá-la possui as devidas permissões. O problema com essa abordagem é que, como precisaríamos alterar a própria classe, todo o resto do programa que usa essa classe teria que ser alterado também.

Uma solução melhor seria utilizar outra classe para verificar se o usuário possui permissão de acesso e só então exibir as informações do banco. Vamos mostrar esta solução utilizando o padrão Proxy.

# ***Proxy***

Vejamos a intenção do padrão Proxy:

“Fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto.” [1]

Extamente a solução falada antes. Vamos utilizar uma classe substituta à classe BancoUsuarios para controlar o acesso.

Vamos então criar a classe Proxy. Para garantir que ela possa realmente substituir a classe original precisamos fazer com que a classe proxy extenda o comportamento da classe original:

```
1  public class BancoProxy extends BancoUsuarios {
2
3      protected String usuario, senha;
4
5      public BancoProxy(String usuario, String senha) {
6          this.usuario = usuario;
7          this.senha = senha;
8      }
9  }
```

Pronto, como a classe BancoProxy extende o comportamento de BancoUsuarios nós podemos utilizar um BancoProxy em qualquer lugar onde um BancoUsuarios era esperado. Além disso adicionamos aqui os campos de usuário e senha, que serão verificados nas operações.

Vamos então sobreescrever os métodos que buscam informação no banco. Agora, como temos o usuário e a senha, podemos realizar a verificação. Caso o usuário tenha permissão de acesso nós realizamos a chamado ao método da classe BancoUsuarios, caso contrário, retornamos nulo:

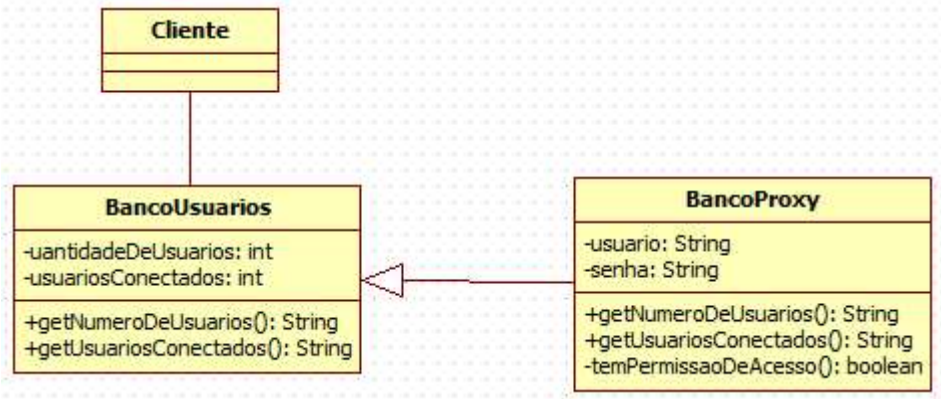
```java
public class BancoProxy extends BancoUsuarios {

    protected String usuario, senha;

    public BancoProxy(String usuario, String senha) {
        this.usuario = usuario;
        this.senha = senha;
    }

    @Override
    public String getNumeroDeUsuarios() {
        if (temPermissaoDeAcesso()) {
            return super.getNumeroDeUsuarios();
        }
        return null;
    }

    @Override
    public String getUsuariosConectados() {
        if (temPermissaoDeAcesso()) {
            return super.getUsuariosConectados();
        }
        return null;
    }

    private boolean temPermissaoDeAcesso() {
        return usuario == "admin" && senha == "admin";
    }
}
```

Pronto, com esta classe nós implementamos a segurança necessária sem precisar alterar o programa. Quando for necessário implementar um acesso seguro, utilizamos a classe Proxy, quando não, podemos utilizar a classe original sem nenhum problema. Vejamos por exemplo o seguinte código cliente:

```
1  public static void main(String[] args) {
2  System.out.println("Hacker acessando");
3  BancoUsuarios banco = new BancoProxy("Hacker", "1234");
4  System.out.println(banco.getNumeroDeUsuarios());
5  System.out.println(banco.getUsuariosConectados());
6
7  System.out.println("\nAdministrador acessando");
8  banco = new BancoProxy("admin", "admin");
9  System.out.println(banco.getNumeroDeUsuarios());
10 System.out.println(banco.getUsuariosConectados());
11 }
```

Veja que utilizamos uma referência a um objeto do tipo BancoUsuarios, mas instanciamos um objeto do tipo BancoProxy. Isto mostra o que falamos antes sobre utilizar um proxy em qualquer lugar que um objeto original seria esperado. Caso houvesse um método que tivesse como entrada um objeto BancoUsuarios poderíamos sem nenhum problema utilizar um objeto BancoProxy para manter a segurança.

O diagrama UML para este caso de uso do Proxy seria bem simples:




# Um pouco de teoria

Antes de falar sobre o padrão proxy é preciso comentar um pouco sobre os tipos de proxy que podem ser utilizados:

- Protection Proxy: esse é o tipo de proxy que utilizamos no exemplo. Eles controlam o acesso aos objetos, por exemplo, verificando se quem chama possui a devida permissão.
- Virtual Proxy: mantem informações sobre o objeto real, adiando o acesso/criação do objeto em si. Como exemplo podemos citar o caso mostrado em [1], onde é utilizado um Proxy que guarda algumas informações sobre uma imagem, não necessitando criar a imagem em si para acessar parte de suas informações.
- Remote Proxy: fornece um representante local para um objeto em outro espaço de endereçamento. Por exemplo, considere que precisamos codificar todas as solicitações enviadas ao banco do exemplo anterior, utilizaríamos um Remote Proxy que codificaria a solicitação e só então faria o envio.
- Smart Reference: este proxy é apenas um susbtituto simples para executar ações adicionais quando o objeto é acessado, por exemplo para implementar mecanismos de sincronização de acesso ao objeto original.

Cada proxy implicaria em um design diferente. No exemplo que citamos, consideramos que a classe original não poderia ser modificada, por isso foi necessário extender seu comportamento. No entanto, caso essa não fosse uma restrição poderíamos criar uma interface comum de acesso ao banco e utilizar ela em nosso programa, assim a implementação do programa ficaria independente da implementação do banco em si (segura ou não).

A principal vantagem de utilizar o Proxy é que, ao utilizar um substituto, podemos fazer desde operações otimizadas até proteção do acesso ao objeto. No entanto isto também pode ser visto como um problema, pois, como a responsabilidade de um proxy não é bem definida é necessário conhecer bem seu comportamento para decidir quando utilizá-lo ou não.

## *Código fonte completo*

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

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

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

## *Referências:*

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

Padrões de Projeto, Proxy  Java, Padrões, Projeto, Proxy  2 Comentários

__PRESENT