Alguns Padrões de Projeto Gamma e Suas Implementações

Pasteur Ottoni de Miranda Junior

Padrões Gamma de Projeto(ou *Gang-of-Four*, gof)

Os padrões "gof" foram publicados por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides em 1998 [GAM98] e têm se constituído em uma referência universal quando se deseja aplicar a projetos de software estruturas padronizadas que privilegiem principalmente a reutilização. Os itens a seguir apresentam e descrevem estes padrões, com exemplos de implementação em Java (modificados e adaptados a partir de [KIK02]).

1-Definição: Padrões de Projeto

Segundo [GAM98] um padrão de projeto é uma solução padronizada para um problema que se repete muitas vezes dentro de um determinado contexto. Um padrão de projeto abstrai e identifica aspectos comuns de estruturas de projeto, tornando-as reutilizáveis. Portanto, cada padrão trata de um *problema* de projeto em particular, que em última análise determina quando utilizar tal padrão. Cada problema tem uma *solução* proposta que é um gabarito a ser aplicado às mais diversas situações.

2-Padrões de criação

Estes padrões abstraem o processo de instanciação. Ajudam a tornar o sistema independente de como seus objetos são criados, compostos e representados.

2.1-O padrão abstract factory (fábrica abstrata)

***Objetivo**:* Prover uma interface para criação de famílias de objetos relacionados ou dependentes sem que se especifiquem suas classes concretas.

***Problema típico** :* Seja uma classe cliente que utiliza um conjunto de classes através de seus métodos. Suponha que cada uma destas classes deva trabalhar em ambientes distintos,devendo ter, portanto, versões diferentes para cada um destes ambientes. A questão é: como fazer com que o cliente possa utilizar alternativamente estas versões, sem que a forma como as classes são acessadas sofra alterações em função destas versões?

***Solução**:* O padrão *abstract factory* propõe que o cliente acesse diretamente somente classes abstratas. Vejamos na Figura 1:

ProdutoAbstrato: declara uma interface abstrata para os produtos a serem criados pelas respectivas classes concretas e para ser referenciada genericamente pelo cliente.

ProdutoConcreto: classe que especializa as diversas versões de *ProdutoAbstrato.* Obviamente, não é visualizada pelo programa-cliente.

FabricaAbstrata (AbstractFactory): é a classe que provê a interface dos métodos de acesso às diversas versões de *ProdutoAbstrato,* que serão especializados pelas classes derivadas *FabricaConcreta.* É através de seus métodos *CriarProduto* implementados nas subclasses *FabricaConcreta* que o cliente cria as objetos corretos para diversas versões de classes.

FabricaConcreta: cada versão das classes *ProdutoConcreto* exige uma subclasse de *FabricaConcreta* especializada de *FabricaAbstrata.* Ela implementa os métodos *CriarProduto* herdados de *FabricaAbstrata:* por exemplo, em *FabricaAbstrata1 , CriarProdutoA* cria *ProdutoA1* e *CriarProdutoB* cria *ProdutoB1.*

Client: utiliza apenas as interfaces declaradas por *ProdutoAbstrato* e *FabricaAbstrata*. O acesso à versões é feito via métodos *CriarProduto* de *FabricaAbstrata.*

Exemplos de aplicação:

-Criar interfaces comuns para bancos de dados diferentes. Criam-se fábricas abstratas específicas para cada banco de dados.

-Criar interfaces comuns para bibliotecas de interface diferentes, que usam os mesmos objetos.

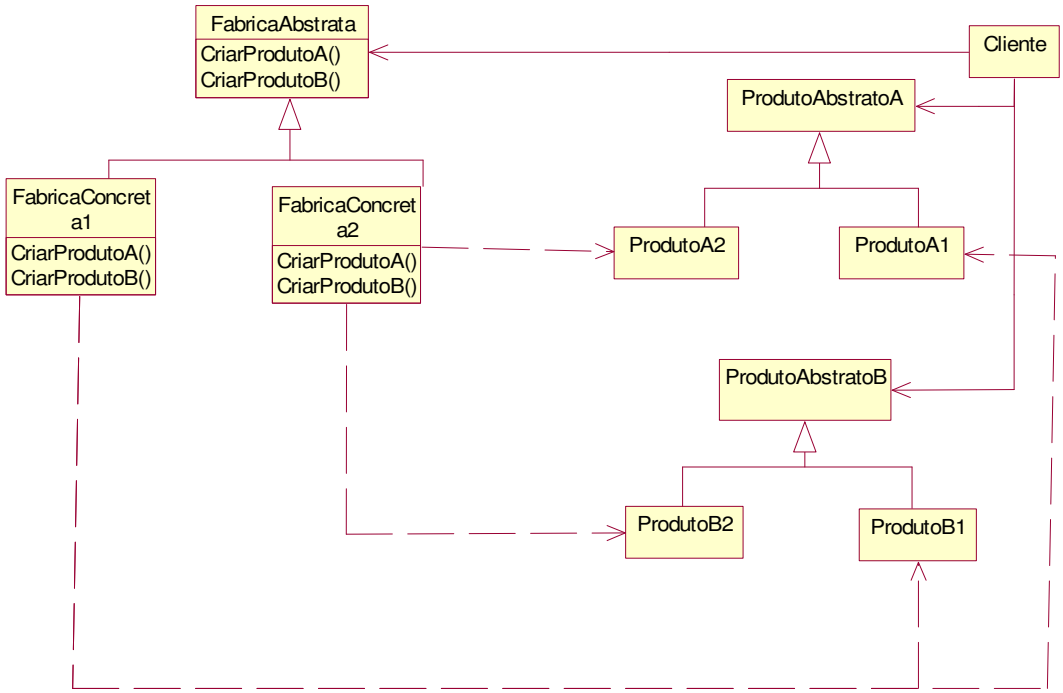

Figura 1-Padrão abstract factory

Uma Implementação em Java:

Neste exemplo vamos criar uma interface simples que utiliza dois objetos: botão e rótulo. Estes dois objetos podem ser criados em duas versões de interface: uma comum e uma "rebuscada", que coloca uma moldura em volta do mesmo. Os comentários elucidam os detalhes de implementação. Apresentaremos aqui somente os detalhes relevantes do código.

```
//Classe que exemplifica ProdutoAbstrato denominada RotuloAbstrato


abstract class RotuloAbstrato extends Jlabel
{
AbstractLabel(String lab){
super(lab);
}
}
```

//Classe que exemplifica ProdutoConcreto denominada RotuloComum

```
class RotuloComum extends RotuloAbstrato
{
    RotuloComum(String lab){
    super(lab);
    }
}
```

//Classe que exemplifica ProdutoConcreto denominada RotuloRebuscado

```
class RotuloRebuscado extends RotuloAbstrato
{
    RotuloRebuscado(String lab){
    super(lab);
```

//Coloca uma borda em volta do rótulo, que fica rebuscado

```
    Border raisedbevel = BorderFactory.createRaisedBevelBorder();
    Border loweredbevel = BorderFactory.createLoweredBevelBorder();
    setBorder(BorderFactory.createCompoundBorder(raisedbevel,
loweredbevel));
    }
}
```

//Classe que exemplifica outro ProdutoAbstrato denominada BotaoAbstrato

```
abstract class BotaoAbstrato extends JButton{
BotaoAbstrato(String lab){
super(lab);
}
}
```

//Classe que exemplifica outro ProdutoConcreto denominada BotaoComum

```
class BotaoComum extends BotaoAbstrato
{
     BotaoComum(String lab){
     super(lab);
     }
}
```

//Classe que exemplifica outro ProdutoConcreto denominada BotaoRebuscado

```
class BotaoRebuscado extends BotaoAbstrato
{
   BotaoRebuscado(String lab){
   super(lab);
   Border raisedbevel = BorderFactory.createRaisedBevelBorder();
   Border loweredbevel =    BorderFactory.createLoweredBevelBorder();
            setBorder(BorderFactory.createCompoundBorder(raisedbevel,
loweredbevel));
   }
}

//Classe FabricaAbstrata (abstract factory) implementada como uma
// interface

public interface FabricaAbstrata {

// Método abstrato que cria o objeto Rotulo, implementado nas classes
// concretas

      public RotuloAbstrato criarRotulo();

//Método abstrato que cria o objeto Botao, implementado nas classes
// concretas
      public BotaoAbstrato criarBotao(String rotulo);



      public String retornaNome();
}







//Implementação da classe abstrata (interface) FabricaAbstrata
//Esta fábrica produz objetos botão e rótulo com uma moldura ao redor

 public class FabricaRebuscada implements FabricaAbstrata {


//Implementação do método abstrato criarRotulo() instancia um
// RotuloRebuscado

public RotuloAbstrato criarRotulo() {
    RotuloRebuscado label = new RotuloRebuscado("Rotulo criado por "
+retornarNome());
    return label;

}


//Implementação do método abstrato criarBotao() que instancia um
//BotaoRebuscado
```

```
     public BotaoAbstrato criarBotao(String Rotulo) {
     BotaoRebuscado button = new BotaoRebuscado(label);
     return button;
      }
```

//Implementação do método retornarNome que retorna o nome da fábrica.

```
      public String retornarNome() {
            return "Fabrica que produz objetos rebuscados";
      }
}
```

//Implementação da classe abstrata (interface) FabricaAbstrata
//Esta fábrica produz objetos botões e rótulos comuns

```
public class FabricaComum implements FabricaAbstrata {


      public String retornarNome() {
            return ("Fábrica comum");
      }
```

//Implementação do método abstrato criarRotulo() instancia um
// RotuloComum

```
public RotuloAbstrato criarRotulo() {
    RotuloComum label = new RotuloComum("Rotulo criado por "
+retornarNome());
            return label;
}
```

//Implementação do método abstrato criarBotao() que instancia um
//BotaoComum

```
public BotaoAbstrato createButton(String label) {
    BotaoComum button = new BotaoComum(label);
    return button;
      }
}
```

//Classe cliente que exibe as duas modalidades de interface, a comum e
//rebuscada,criadas pelas respectivas fábricas concretas. Repare que
seu //código não muda em função da fábrica utilizada.

```
public class Display extends JFrame  {
```

//Construtor parametrizado com a fábrica a ser criada

```
      Display(FabricaAbstrata fabrica) {
```

```
            super("Nova GUI");
            RotuloAbstrato rotulo = fabrica.createLabel();
            BotaoAbstrato botao = fabrica.createButton("OK");
            botao.addActionListener(new myActionListener(this));
            JPanel panel = new JPanel();
            panel.add(rotulo);
            panel.add(botao);
            this.getContentPane().add(panel);
            this.pack();
            this.setVisible(true);
            this.addWindowListener(new myWindowListener(this));
      }

      private class myWindowListener extends WindowAdapter {

            Display display = null;

            protected myWindowListener(Display display) {
                  super();
                  this.display = display;
            }

            public void windowClosing(WindowEvent e) {
                  display.setVisible(false);
            }
      }

      private class myActionListener implements ActionListener {

          Display display;

          protected myActionListener(Display display) {
              super();
              this.display = display;
          }

          public void actionPerformed(ActionEvent e) {
                  display.setVisible(false);
            }
      }
}


//Classe "driver" que executa e testa o padrão

public class Main
{

  //Declarar e instanciar as respectivas Fábricas

      private static FabricaAbstrata factory1 = new FabricaComum();

  private static FabricaAbstrata factory2 = new FabricaRebuscada();

 //Instância factory Recebe a instância ser utilizada. Default: Fábrica
//comum
```

```
      private static FabricaAbstrata factory = factory1;


//Método que cria a interface, detalhes de implementação são irrelevantes
//em nosso contexto

      private static JPanel createGUI()
      {

   //Comando abaixo é um "listener" que ajusta a fábrica a ser utilizada
   //de acordo com a seleção do usuário feita via radio buttons.

            ActionListener radioListener = new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                  if       (e.getActionCommand().equals("fabricacomum"))
factory = factory1;
              else factory = factory2;
    }


            };

            JPanel panel = new JPanel();

    //Instancia os Radio Buttons das respectivas fábricas

            JRadioButton   factoryButton1   =   new   JRadioButton("usar
fábrica comum");
            JRadioButton   factoryButton2   =   new   JRadioButton("usar
fábrica rebuscada");
            factoryButton1.setActionCommand("fabricacomum");
            factoryButton2.setActionCommand("fabricarebuscada");
            factoryButton1.addActionListener(radioListener);
            factoryButton2.addActionListener(radioListener);
            JButton    create    =    new    JButton("Criar    interface
selecionada");

            ButtonGroup choices = new ButtonGroup();

            choices.add(factoryButton1);
            choices.add(factoryButton2);

            create.addActionListener( new ActionListener() {
                  public void actionPerformed(ActionEvent e) {

  //Chama a classe Display, passando como parâmetro a classe
//selecionada via Radio Button correspondente. Repare que para a classe
//Display a interface não muda, o que torna possível a criação de
//qualquer fábrica concreta, sem que ela seja afetada.

                        Display display = new Display(factory);

                  }
            });
```

```
            panel.add(factoryButton1);
            panel.add(factoryButton2);
            panel.add(create);

            return panel;
      }

  //Executa este "driver"

      public static void main(String[] args)
      {
            JFrame frame = new JFrame("Padrão Fábrica Abstrata");
            frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            });

            frame.getContentPane().add(createGUI());

            frame.pack();
            frame.setVisible(true);
      }
}
```

2.2-O padrão builder (construtor)

***Objetivo**:* Fazer com que um processo único de construção de um objeto complexo possa ser utilizado para diferentes formas de representação do mesmo.

***Problema típico** :* Seja um objeto complexo (constituído de várias partes) que tem um processo de construção (junção das partes) específico. Entretanto, este objeto tem muitas versões. A questão é: como fazer com que as diferenças entre as versões não afetem o processo de construção?

***Solução**:* O padrão *builder* propõe uma classe abstrata denominada *builder* (construtora) que encapsula métodos abstratos responsáveis pela construção de partes específicas do objeto complexo. Esta classe é especializada em construtoras concretas responsáveis pelas diversas versões do objeto complexo e os referidos métodos abstratos são por elas implementados. Uma classe denominada *director* (diretora) vai possuir um método que implementa o processo de construção, invocando os métodos de construção, recebendo como parâmetro uma versão do objeto complexo. A classe *director* fica intocada, não modificando o processo de construção em função de novas versões do objeto. Vejamos na Figura 2:

Construtora: é a classe abstrata que provê métodos abstratos responsáveis pela construção das partes do objeto.

ConstrutoraConcreta: é a especialização da classe *Construtora* para cada versão existente do objeto complexo. Implementa os métodos abstratos declarados em *Construtora.*

Diretora: é a classe responsável pela construção do objeto, invocando os métodos definidos na classe *Construtora.*

Produto: constitui as diversas versões do objeto complexo a ser construída.

Exemplos de aplicação:

-Construção de documentos complexos, constituídos de várias partes, que devem ser formatados de diversas maneiras (formato Word, HTML ou RTF, por exemplo).

-Um buffer de impressão deve ter suas partes formatadas para diversos tipos de impressoras. Criam-se classes *ConstrutoraConcreta* especializadas para cada impressora.

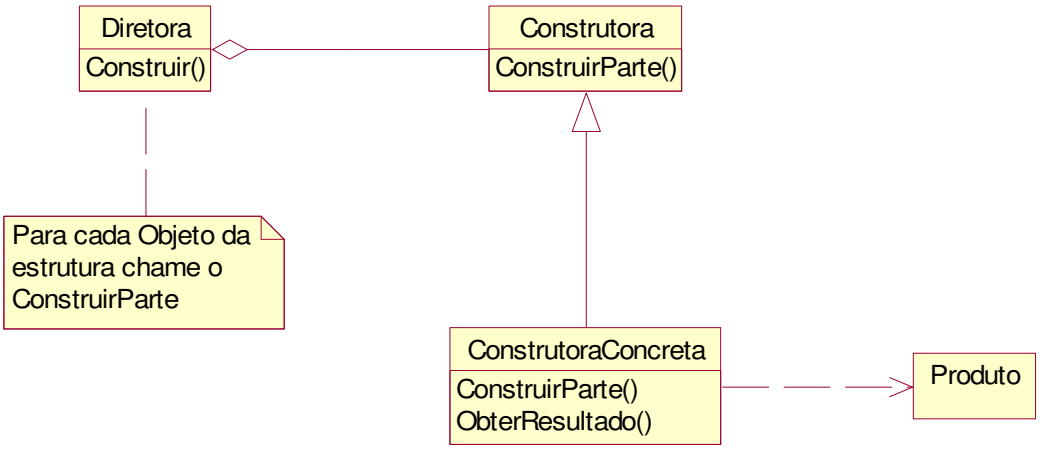

Figura 2-Padrão builder

Uma Implementação em java

O exemplo a seguir, possui uma classe denominada *Construtora,* que possui métodos abstratos para construção das partes de um objeto complexo

denominado *Documento,* que é constituído das partes *titulo, autor e conteúdo.* Os métodos abstratos da classe *Construtora* são especializados para construir o *Documento* em duas versões: uma simples, representada pela classe derivada de *Construtora* denominada *ConstrutoraDocComum,* que não aplica qualquer formatação a *Documento* , e uma versão HTML, que formata *Documento* de forma que possa ser exibido como uma página HTML em um browser. A classe *Main* é a *diretora,* possuindo o método *construir,* que encapsula o processo de criação do objeto complexo, invocando os métodos de construção. Observe que, como os métodos de construção de *Documento* na classe *Construtora* são abstratos, o processo de criação do mesmo independe da versão (simples ou HTML) que está sendo utilizada.

```
 public abstract class Construtora {

//Esta é a classe construtora que possui os métodos de construção do
objeto complexo(objeto Documento).



    protected String resultado;

//Métodos abstratos responsáveis pela construção das partes do objeto
//  complexo Documento. Eles serão particularizados nas classes que
// formatam o documento de maneira comum e HTML

    public abstract void ConstruirTitulo(Documento doc);


    public abstract void ConstruirAutor(Documento doc);



    public abstract void ConstruirConteudo(Documento doc);


//Método que exibe o resultado da construção, não necessitando
// especialização.

    public String obterResultado() {
        return resultado;
    }
}



//Esta classe é derivada de Construtora, especializando seus métodos de
// forma que as partes do objeto complexo Documento sejam construídas
sem // formatação
```

```
public class ConstrutoraDocComum extends Construtora {


    public void ConstruirTitulo(Documento doc) {
        resultado = doc.getTitulo()+"\n";
    }



    public void ConstruirAutor(Documento doc) {
        resultado += doc.getAutor()+"\n";
    }


    public void ConstruirConteudo(Documento doc) {
        resultado += doc.getConteudo()+"\n";
    }
}

public class ConstrutoraDocHTML extends Construtora {
```

//Esta classe é derivada de Construtora, especializando seus métodos de // forma que as partes do objeto complexo Documento sejam construídas com // formatação HTML

```
public void ConstruirTitulo(Documento doc) {
resultado    ="<!DOCTYPE    HTML    PUBLIC    '-//W3C//DTD    HTML    4.01
Transitional//EN'>"+
"<html>"+
"<head>"+
"<title>"+doc.getTitulo()+"</title>"+
"<meta  http-equiv='Content-Type'  content='text/html;  charset=iso-8859-
1'>"+
"</head>"+
"<body>"+
"<div align='center'>"+
"<p><font size='7'><strong>"+doc.getTitulo()+ "<br>"+
"</strong><strong></strong></font><strong><br>"+
"<br>";
}

public void ConstruirAutor(Documento doc) {
resultado += "Autor :"+ doc.getAutor()+"</p> <br>";
}

public void ConstruirConteudo(Documento doc) {
resultado += "<p align='left'>"+doc.getConteudo()+"</p>"+
"</div>"+
"</body>" +
"</html>";


}
```

```
}

class Documento  {

 /*Produto Documento, que é o objeto complexo a ser construído,
constituído de um título, um autor e um conteúdo*/

    private String titulo;
    private String autor;
    private String conteudo;

    public Documento (String tit,  String aut,  String cont)
    {
     titulo = tit;
     autor = aut;
     conteudo = cont;
    }

    public String getTitulo(){
    return titulo;
    }

    public String getAutor(){
    return autor;
    }

    public String getConteudo(){
    return conteudo;
    }

}


//A classe Diretora:

public class Main {


    protected static void construir(Construtora constru, Documento doc)
{

    //Método construir, que constrói as partes do objeto.

        constru.ConstruirTitulo(doc);
        constru.ConstruirAutor(doc);
        constru.ConstruirConteudo(doc);
    }


    public static void main(String[] args) {

        //Instancio os dois tipos de construtora

        Construtora ConstDocComum = new ConstrutoraDocComum();
        Construtora ConstDocHTML = new ConstrutoraDocHTML();
```

```
        //Inicializo o objeto complexo, passando suas partes como
//parâmetro

Documento doc = new Documento("Padrão Builder","Pasteur O.M. Jr.",
"Conteúdo do documento");

        //Construo o documento sem formatação alguma

        construir(ConstDocComum,doc);

        //Construo o documento com formatação HTML

        construir(ConstDocHTML,doc);

        //Exibo os resultados obtidos com as duas formatações

        System.out.println(ConstDocComum.obterResultado());
        System.out.println(ConstDocHTML.obterResultado());
    }
}
```

2.3-O padrão factory method

Objetivo :Permitir a definição de uma interface (o factory method) em uma classe mãe abstrata que, em suas subclasses, especializa-se para criar vários tipos de objetos.

Problema típico: Suponha que uma determinada classe utiliza diversos tipos de objetos. Mas no ponto em que utiliza estes objetos, ela não pode determinar a classe de tais objetos. Como fazer com que esta classe possa utilizar estes objetos independentemente da classe à qual pertençam?

Solução: Criar uma classe abstrata que contenha um *factory method,* que é a interface pela qual classes derivadas criam os objetos específicos. A classe base implementa um método que acessa a criação de tais objetos através da interface do factory method. Na Figura 3 abaixo temos:

Criadora: Classe abstrata que declara um *factory method* abstrato, utilizado por um método (*operação)* que utiliza múltiplos tipos de objetos. Este *factory method* sempre retorna uma instância de um tipo de objeto e é especializado nas classes filhas (*CriadoraConcreto)*

Produto: Constitui uma classe abstrata que provê uma interface comum de objetos a serem criados pelo *factory method.*

ProdutoConcreto: Especializa a interface definida em *Produto.*

CriadoraConcreta: Classe derivada de *Criador* que implementa o *factory method* declarado nesta última capaz de retornar um *ProdutoConcreto* . O método *operacao* implementado em *Criadora* acessa por vínculo tardio (*late binding)* a implementação correta do *factory method* correspondente ao objeto desejado.

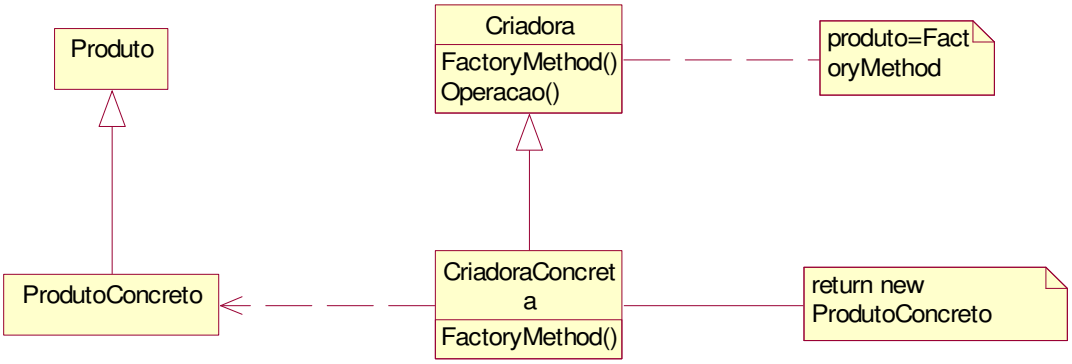

Figura 3-Padrão factory method

Exemplos de aplicação:

-Uma aplicação possui um método que deve abrir e exibir diversos tipos de documento. Este método não sabe qual tipo de documento vai ser exibido e deve abri-los corretamente.
-Uma ferramenta CASE deve mover e redimensionar objetos de tipos diferentes. Estas operações não sabem que tipo de objeto vai ser manipulado.

Uma Implementação em Java:

Neste exemplo temos uma aplicação que deve exibir documentos sem formatação (classe *DocumentoComum)* e documentos formatados em HTML(*DocumentoHTML)*. Estes tipos de documento são derivados da classe *Documento.* Estes documentos são exibidos pela classe *CriadoraDocumento,* que declara um factory method denominado *criarDocumento* e exibe o conteúdo do documento via método *mostrarDocumento,* que invoca *criarDocumento.*Repare que a implementação de *mostrarDocumento* é independente do tipo de documento. As classes *CriadoraDocComum* e *CriadoraDocHTML* implementam o factory method *criarDocumento* que apenas instancia a classe correta de documento (*DocumentoComum* ou *DocumentoHTML).*

```
//Classe abstrata documento, que será especializada em dois tipos:
comum //e HTML

abstract class Documento  {
```

```
    protected String titulo;
    protected String autor;
    protected String conteudo;

    public String getTitulo(){
    return titulo;
    }

    public String getAutor(){
    return autor;
    }

    public String getConteudo(){
    return conteudo;
    }


}
```

//Especialização de Documento: cria documento sem formatação

```
class DocumentoComum extends Documento{

public   DocumentoComum (String tit,  String aut,  String cont){
```

** //Cria um documento comum**

```
  titulo = tit +"\n";
  autor = aut + "\n";
  conteudo = cont +"\n";
}


}
```

//Especialização de Documento: cria documento com formatação HTML

```
class DocumentoHTML extends Documento{

public   DocumentoHTML (String tit,  String aut,  String cont){
```

//Documento é criado com formatação HTML

```
titulo    ="<!DOCTYPE    HTML    PUBLIC    '-//W3C//DTD    HTML    4.01
Transitional//EN'>"+
"<html>"+
"<head>"+
"<title>";
titulo += titulo+tit+"</title>"+
"<meta  http-equiv='Content-Type'  content='text/html;  charset=iso-8859-
1'>"+
"</head>"+
"<body>"+
"<div align='center'>"+
"<p><font size='7'><strong>";
```

```
titulo += titulo+tit+ "<br>"+
"</strong><strong></strong></font><strong><br>"+
"<br>";
autor = "Autor :";
autor += autor+aut+"</p> <br>";
conteudo = "<p align='left'>";
conteudo += conteudo+cont+"</p>"+"</div>"+
"</body>" +
"</html>";
}

}
```

//Classe Produto que declara o factory method

```
public abstract class CriadoraDocumento {
```

//Factory method que cria um Documento e o retorna

```
    public abstract Documento criarDocumento(String tit,  String aut,
String cont);
```

//O método a seguir apenas exibe o documento. Repare que é independente //do tipo de documento.

```
    public final void mostrarDocumento(String tit,  String aut,  String
cont) {
```

//Instancia um documento de qualquer tipo invocando o factory //method

```
      Documento doc = criarDocumento(tit,  aut,  cont);
```

//Exibe o documento de qualquer tipo

```
      System.out.println(doc.getTitulo());
      System.out.println(doc.getAutor());
      System.out.println(doc.getConteudo());

    }
}
```

//Especializa a classe CriadoraDocumento para criar documentos comuns

```
public class CriadoraDocComum extends CriadoraDocumento {
```

//Especializa o factory method criarDocumento para que ele retorne um //DocumentoComum

```
  public Documento criarDocumento(String tit,  String aut,  String
cont) {
        DocumentoComum doc = new DocumentoComum (tit,aut,cont);

        return doc;
    }
}
```

```
//Especializa a classe CriadoraDocumento para criar documentos HTML

public class CriadoraDocHTML extends CriadoraDocumento {

//Especializa o factory method criarDocumento para que ele retorne um
//DocumentoHTML

  public Documento criarDocumento(String tit,   String aut,   String
cont) {
        DocumentoHTML doc = new DocumentoHTML (tit,aut,cont);

        return doc;
    }
}

public class Main {

//Driver para execução do exemplo.Deve exibir qualquer tipo de
documento

    public static void main(String[] args) {

        //Instancio as classes criadoras de documentos comuns e HTML

        CriadoraDocumento criadcomum = new CriadoraDocComum();
        CriadoraDocumento criadhtml = new CriadoraDocHTML();

        CriadoraDocumento cri;

//Se o parâmetro passado for "c", mostra o documento comum,  senão,
//mostra o HTML

        if (args[0].equals("a")){
           cri = criadcomum;
        }
        else {
           cri = criadhtml;
        }
//Uma única chamada para qualquer que seja o documento a exibir:

cri.mostrarDocumento("Padrão Builder","Pasteur O.M. Jr.",
"Conteúdo do documento");

    }
}
```

2.4-O padrão Singleton

Objetivo: Garantir que uma classe possua uma e somente uma instância.

Problema típico:Não faz sentido no contexto de uma aplicação a existência de mais de uma instância de uma determinada classe. O problema é: como fazer

com que o desenvolvedor não tenha acesso a métodos construtores que criem mais de uma instância da classe, mantendo-se fácil acesso à instância?

Solução: A própria classe mantém controle sobre sua instância única, tornando público um método para acesso a esta instância, que impede a criação de outra instância. Na Figura 4 temos o método *Instancia()* que é estático e na verdade é um factory method que retorna *InstanciaUnica* e garante que não será criada mais que uma instância de *Singleton.*

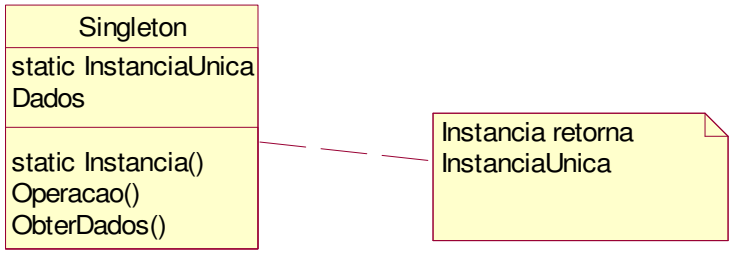

Figura 4-Padrão Singleton

Exemplos típicos:

-Em uma aplicação, deve haver um único spool de impressão para determinada impressora.
-Classes de controle e de gerência (por exemplo, gerenciadoras de janelas e de arquivos) devem ter uma única instância em determinada sessão.

Uma Implementação em Java:

Este exemplo simples e compreensível, implementa um dispositivo que exibe mensagens na tela. Não faz sentido permitir a criação de mais de uma instância dele.

```
// Este singleton é um dispositivo que imprime mensagens para a tela.

public class TelaSingleton {

//Variável instanciaunica, que é estática, de forma a garantir a
criação //de outras instancias
      protected static TelaSingleton instanciaunica = null;

  protected int id = 0;
```

```
//Marcador para indicar no teste que as instâncias são iguais

  public String marc = null;


//Construtor protegido, para impedir que a instancia seja criada de
outra
//forma que não via método Instance()

      protected TelaSingleton() {
       id++;

      }

//Método Instance é um factory method que retorna a instância única
//criada

      public static TelaSingleton Instancia() {

            if(instanciaunica == null) {

//Se instanciaunica não tiver sido criada, criá-la.

                  instanciaunica = new TelaSingleton();

//Ajustar valor do marcador
                  instanciaunica.marc = "INSTANCIA ÚNICA";
            }

//Senão, a retorna, porque já foi criada

            return instanciaunica;
      }



//Método para impressão da mensagem na tela.

      public void imprimir(String mensagem) {
            System.out.println(mensagem);
      }
}







class Main {



      public static void main (String[] args) {

 //Crio o dispositivo para impressão na tela e imprimo mensagem
```

```
    TelaSingleton T1 = TelaSingleton.Instancia();
    T1.imprimir("T1.marc");

//Declaro novo dispositivo e imprimo. A instancia retornada é a mesma:
//T1=T2, o que é verificado pelas mensagens idênticas que são impressas
// via atributo marc do singleton.

  TelaSingleton T2 = TelaSingleton.Instancia();
  T2.imprimir("T2.marc");



      }
}
```

2.5-O padrão Prototype (Protótipo)

Objetivo: Prover uma abordagem diferente para criação de objetos, através da clonagem de uma instância denominada *protótipo.*

Problema típico: Suponha uma classe A que manipula um certo número de tipos de objetos pertencentes a classes especializadas de uma classe base B. A classe A não sabe como criar instâncias de subclasses de B. Uma solução seria criar uma subclasse de A para cada subclasse de B, cada uma delas especializada para lidar com uma subclasse de B. Porém isto poderia gerar um grande número de subclasses de A, algo pouco elegante, principalmente se as subclasses de B diferirem entre si por detalhes. Como fazer, então, com que a classe A consiga manipular as subclasses de B?

Solução: Fazer com que a classe A crie cópias ou "clones" de instâncias já definidas das subclasses de B. Estas instâncias são os *protótipos.* Na Figura 5 temos:
Prototipo: declara uma interface para clonar

PrototipoConcreto: implementa a operação de clonagem

Cliente: cria um novo objeto solicitando a *Prototipo* que clone a si mesmo. O método *Operacao()* acessa objetos *Prototipo,* clonando-os.

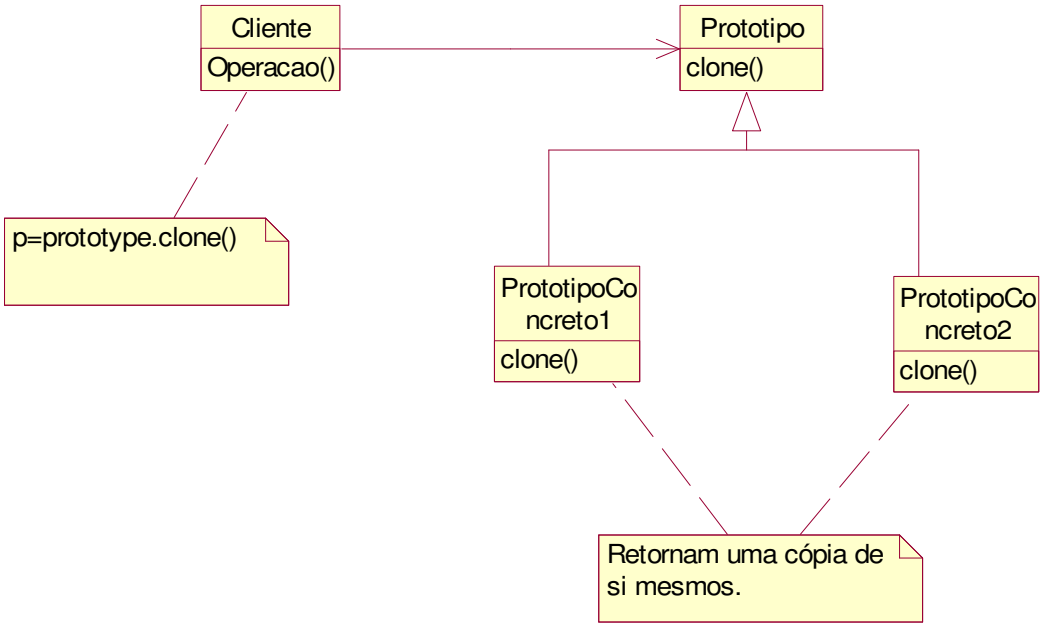

Figura 5- Padrão prototype

Exemplos de utilização:

-Apesar de semelhante ao abstractFactory, este padrão permite a instância de novas classes em tempo de execução. Por exemplo, uma aplicação que permita ao usuário exibir documentos, pode dar ao mesmo o recurso de inserir novos tipos de documentos e exibi-los sem necessidade de recompilação de código.
-Uma ferramenta CASE que trabalhe com frameworks de projeto define protótipos para os mesmos, que serão clonados quando o cliente deles necessitar.

Uma Implementação em Java:

Vamos utilizar aqui uma nova versão da classe *Documento* já apresentada nos exemplos anteriores. Ela é derivada da interface *off the shelf* Java *Cloneable* que possui o método `clone()`, que permite a clonagem de instâncias. Repare que um clone é uma cópia da instância, ou seja, se alterarmos seus atributos , os atributos da instância clonada não serão modificados. Por exemplo, se i1 é a instância a ser clonada e c1 é seu clone temos:
-Se fizermos c1=i1;, c1 não será um clone de i1, será a *própria* i1, ou seja, qualquer alteração efetuada em c1 refletirá em i1 e vice-versa.

-Se fizermos c1=i1.clone();, c1 será um clone (cópia) de i1, ou seja, qualquer alteração efetuada em c1 não refletirá em i1 e vice-versa.
A classe *Documento* possui também as especializações *DocumentoComum* e *DocumentoHTML.* A classe *CriadoraDocumento* é também mantida, mas no caso deste padrão, não precisa ser especializada em *CriadoraDocumentoComum* e *CriadoraDocumentoHTML.* Ao invés disto, ao ser inicializada, ela recebe como parâmetro o *protótipo* da classe a ser clonada. Quando seu método *mostrarDocumento* é invocado o protótipo é clonado e inicializado.

```
abstract class Documento implements Cloneable {

//Classe derivada de Cloneable, que permite clonar suas instâncias.


    protected String titulo;
    protected String autor;
    protected String conteudo;



    public String getTitulo(){
    return titulo;
    }

    public String getAutor(){
    return autor;
    }

    public String getConteudo(){
    return conteudo;
    }

    public void setTitulo(String tit){
      titulo = tit;
    }

    public void setAutor(String aut){
      autor = aut;
    }

    public void setConteudo(String cont){
     conteudo = cont;
    }

    public abstract void Inicializar(String tit,  String aut,  String
cont);

    public Object clone() throws CloneNotSupportedException  {
        return super.clone();
    }
```

```
}
```

//Especialização de Documento: cria documento sem formatação

```
class DocumentoComum extends Documento{

public   DocumentoComum (){
}

//Documento é inicializado sem formatação

public void Inicializar(String tit,  String aut,  String cont){
  titulo = tit +"\n";
  autor = aut + "\n";
  conteudo = cont +"\n";
}


}
```

//Especialização de Documento: cria documento com formatação HTML

```
class DocumentoHTML extends Documento{

public   DocumentoHTML (){
}


//Documento é inicializado com formatação HTML


public void Inicializar(String tit,  String aut,  String cont){
titulo    ="<!DOCTYPE    HTML    PUBLIC    '-//W3C//DTD    HTML    4.01
Transitional//EN'>"+
"<html>"+
"<head>"+
"<title>";
titulo += titulo + tit+"</title>"+
"<meta  http-equiv='Content-Type'  content='text/html;  charset=iso-8859-
1'>"+
"</head>"+
"<body>"+
"<div align='center'>"+
"<p><font size='7'><strong>";
titulo += titulo+tit+ "<br>"+
"</strong><strong></strong></font><strong><br>"+
"<br>";
autor = "Autor :";
autor += autor+aut+"</p> <br>";
conteudo = "<p align='left'>";
conteudo += conteudo+cont+"</p>"+"</div>"+
"</body>" +
"</html>";
}
```

```
}

class CriadoraDocumento {

//Classe cliente responsável pela clonagem dos protótipos através da
// operação mostrarDocumento()

//atributo que vai conter o protótipo a ser clonado

private Documento documentoprototipo;

//Construtor recebe o protótipo e o aloca ao atributo
documentoprototipo


public CriadoraDocumento(Documento doc){
     documentoprototipo = doc;
}

//O método a seguir apenas exibe o documento. Repare que é independente
// do tipo de documento,clonando o protótipo e em seguida incializando
// seus atributos.

    public final void mostrarDocumento(String tit,  String aut,  String
cont) {

      //Instancia um documento de qualquer tipo clonando o protótipo

      try {

      Documento docclone = (Documento)documentoprototipo.clone();

      //Inicializa o clone com os parâmetros corretos

       docclone.Inicializar(tit,aut,cont);

      System.out.println(docclone.getTitulo());
      System.out.println(docclone.getAutor());
      System.out.println(docclone.getConteudo());

      } catch (CloneNotSupportedException ex) {
            System.err.println("Failure! "+ex);
      }

    }
}


//Driver para demonstrar utilização dos clones

public class Main {


    public static void main(String[] args) {

//Criação dos protótipos que serão clonados
```

```
            DocumentoComum doccomum = new DocumentoComum();
            DocumentoHTML dochtml = new DocumentoHTML();

            CriadoraDocumento cri;
```

```
//Se o parâmetro passado for "c", mostra o documento comum,
// senão, mostra o HTML. Passo os protótipos como parâmetros para a
// classe criadora do documento

      if (args[0].equals("a")){
           cri = new CriadoraDocumento(doccomum);
        }
        else {
           cri = new CriadoraDocumento(dochtml);
        }

//Uma única chamada para qualquer que seja o documento a exibir:

cri.mostrarDocumento("Padrão Prototype","Pasteur O.M. Jr.","Conteúdo do
documento");

//Se quisermos mostrar outro documento, basta chamarmos novamente o
//método mostrarDocumento

        cri.mostrarDocumento("Outro doc", "Qualquer","Conteúdo novo");


    }
}
```

3-Padrões estruturais

Os padrões estruturais estão relacionados a como classes e objetos são compostos de forma a gerar estruturas maiores e mais complexas.

3.1- O padrão adaptador (adapter)

Objetivo: Converter a interface de uma classe em outra, permitindo que classes com interfaces incompatíveis possam trabalhar em conjunto.

Problema típico: Suponha que uma aplicação utilize um recurso de uma classe específica A. Este recurso é acessado via métodos da classe A. Entretanto, descobriu-se no mercado uma nova alternativa para realizar as atribuições da classe A, que possui métodos de acesso completamente diferentes dos de A. A questão é: como introduzir esta nova classe na aplicação, sem alterar a interface

de acesso na aplicação, ou ainda, como permitir que novas classes possam ser acessadas pela aplicação sem modificar esta interface de acesso?

Solução: Criar uma classe abstrata comum para a classe A e a nova alternativa, que terá sua interface acessada pelo cliente. Esta classe será particularizada para a classe A e para a classe alternativa e é denominada *adaptadora.* Na Figura 6 temos:

Alvo: define a interface abstrata a ser utilizada pelos clientes (método Requisicao)

Adaptada: interface a ser adaptada. Possui um método RequisicaoEspecifica que precisa ser adaptado para ser utilizado por *Alvo.*

Adaptadora: adapta a interface de *Adaptada* à interface de *Alvo.* Implementa o método *Requisicao* declarado em *Alvo,* invocando o método *RequisicaoEspecifica* de *Adaptada.*

Cliente: acessa a interface comum de Alvo.

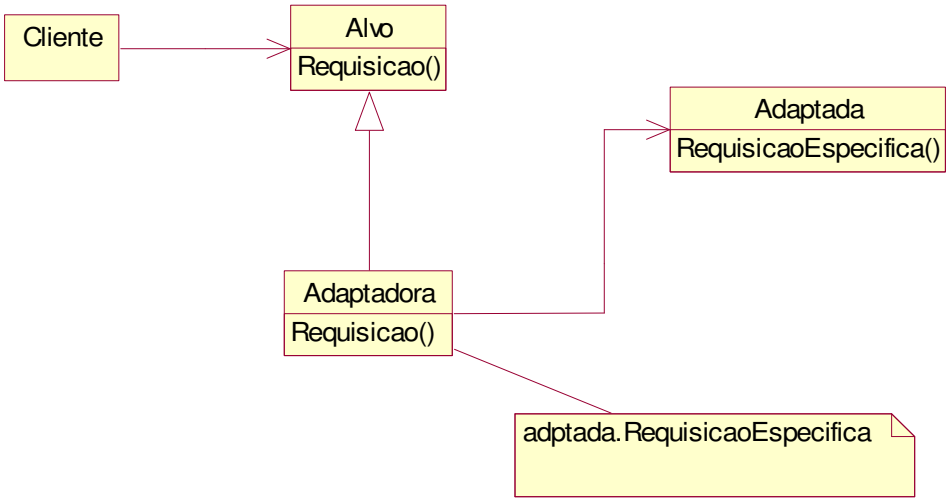

Figura 6-Padrão Adapter

Exemplos de utização:

-Uma aplicação que utiliza diversos dispositivos pode acessar uma interface comum, que será particularizada para cada novo dispositivo inserido, via classe adptadora. Esta classe adaptadora tornar-se-á um *plug-in* para o novo dispositivo na aplicação.

-Uma aplicação utiliza uma interface comum para imprimir, que é adaptada por meio de subclasses adaptadoras a qualquer tipo de impressora.

Uma Implementação em Java:

Este exemplo simples exibe um frame contendo componentes quaisquer via uma interface comum. Qualquer componente é inserido no frame via método `inserircomponente()` definido na classe Alvo denominada `Componente`. A classe cliente `Main` cria o frame e acessa o método `inserircomponente()` da classe `Componente` para inserir um botão e texto. Estes últimos são representados pelas classes Botao e Texto respectivamente e têm seus respectivos métodos `inserirbotao()` e `inserirtextos()` adaptados no método `inserircomponente()` especializado nas classes `BotaoComponenteAdapter` e `TextoComponenteAdapter()`. Repare que, qualquer componente pode ser inserido no painel por uma única interface, desde que seja criada sua classe adptadora correspondente.

```
abstract class Componente {

/* Classe Alvo, que define a interface que o cliente vai usar   */

  protected static JPanel panel=null;


/* Método Requisicao, acessado por cliente e implementado nas
* subclasses adaptadoras                                    */


      public abstract JPanel inserircomponente();


}

class Texto {

/* Interface da classe Texto deve  ser adaptada à interface de
Componente */

  private String txt;

  public Texto(String s){
   txt = s;
  }

/* Método RequisicaoEspecifica, que deve ser adaptado */

      public String inserirtexto() {
```

```
  return txt;

      }
}


class Botao extends JButton{

/* Interface da classe Botao deve  ser adaptada à interface de
Componente */

   public Botao(String s){
     super(s);

   }

/* Método RequisicaoEspecifica, que deve ser adaptado */

   public JButton inserirbotao(){
    return this;

   }




}



class TextoComponenteAdapter extends Componente {

/* Classe adaptadora, que faz a adaptação da interface de
Texto à interface  de Componente */



      private Texto tex;




    public TextoComponenteAdapter(Texto tex) {
        this.tex = tex;
        if (panel == null) {panel = new JPanel();}


    }


/*Implementacao do método Requisicao */

      public JPanel inserircomponente() {
```

```
/*  Chama  RequisicaoEspecifica  (inserirtexto()),  adaptando-o  a
Requisicao
*   (inserircomponente)  */

       panel.add(new JLabel(tex.inserirtexto()));
       return panel;


      }
}


class BotaoComponenteAdapter extends Componente {

/* Classe adaptadora, que faz a adaptação da interface de Botao à
interface  de Componente */

        private Botao bot;


    public BotaoComponenteAdapter(Botao bot) {
        this.bot = bot;
        if (panel == null) {panel = new JPanel();}

    }

/*Implementacao do método Requisicao */

        public JPanel inserircomponente() {

/*  Chama  RequisicaoEspecifica  (inserirbotao()),  adaptando-o  a
Requisicao
*   (inserircomponente)  */


       panel.add(bot.inserirbotao() );
       return panel;


      }
}




/* Classe Cliente, que vai utilizar a interface da classe Componente */

public class Main {


/* Declaração da instância de Componente, onde serão inicializadas as
* classes adaptadoras */

      private static Componente adapter;
```

```
/*Classes Texto e Botao a serem adaptadas */

      private static Texto txt = new Texto("TEXTO");

      private static Botao bot = new Botao("BOTAO");


public static void main(String[] args) {

    JFrame frame = new JFrame("Padrão Adapter");

    frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {System.exit(0);}
            });

    JPanel panel ;


 /*Inicializar a classe que adapta Texto a Componente */

adapter = new TextoComponenteAdapter(txt);


/* Invocar a interface comum inserircomponente(), particularizada para
a  * classe   adaptadora */

    panel = adapter.inserircomponente();

 /*Inicializar a classe que adapta Botao a Componente */

    adapter = new BotaoComponenteAdapter(bot);



/* Invocar a interface comum inserircomponente(), particularizada para
a *  classe  adaptadora */


    panel =adapter.inserircomponente();


            frame.getContentPane().add(panel);

            frame.pack();
            frame.setVisible(true);


      }
}
```

3.2- O padrão bridge (ponte)

Objetivo: Permitir que a abstração de uma classe seja independente de sua implementação, de forma a facilitar e dimuir a necessidade de combinação entre diversos tipos de implementação e diversos tipos de classes.

Problema típico: Suponha uma classe A que possa ser especializada em n tipos de classes. Suponha que esta classe possa ser implementada de m formas. Sendo assim, para suportar estas variedades, necessitaríamos de mXn subclasses para comportar todas as combinações entre tipos de classes e formas de implementação. A questão é: como separar a abstração da implementação, de forma que não seja necessária a criação destas mXn combinações?

Solução: Criar duas classes-bases: uma para abstrair o conceito da classe e outra para abstrair sua implementação. A classe de implementação é especializada nas diversas formas de implementação possíveis e define uma interface para acesso a métodos primitivos. A classe de abstração acessa instâncias da classe de implementação e utiliza os métodos primitivos desta última, para compor os diversos tipos de abstração possíveis. Este acesso é a "ponte" (bridge) que liga a abstração a sua implementação.Na Figura 7 temos:

Abstracao: Define a interface de abstração da classe.
AbstracaoRefinada: Representa as diversas especializações de *Abstração.*
Implementadora: Define a interface das classes de implementação. Provê métodos primitivos para implementação (*ImplOperacao()*), que são utilizados por métodos de nível mais alto (*Operacao()*)na classe *AbstracaoRefinada.*
ImplementadoraConcreta: Implementa a interface da classe *Implementadora,* especializando-a.

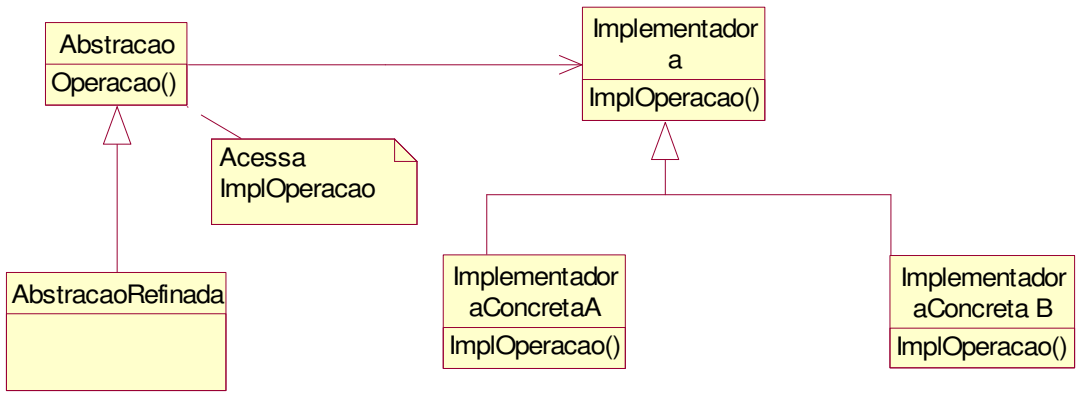

Figura 7 – O padrão Bridge

Exemplos de utilização:

-Suponha um objeto janela que pode ter diversos tipos de estruturação (como caixa, como janela modal, como MDI, etc). Suponha que esta janela tenha implementações particularizadas para ambientes específicos, por exemplo, para ambiente Linux e Windows. Podemos criar uma abstração da classe janela e particularizá-la para os seus diversos tipos possíveis. Criamos também uma classe de implementação de janelas, que será particularizada para as implementações em Linux e Windows.
-Documentos de diversos tipos (por exemplo, artigo, carta, tutorial, etc) têm uma abstração comum e podem ter diversas implementações para formatos diferentes (por exemplo, rtf, doc, HTML, etc).

Uma Implementação em Java:

Este exemplo tem uma classe *Abstracao* chamada `Documento`, particularizada em dois tipos: `DocumentoArtigo` e `DocumentoTutorial`. A classe `Documento` tem 3 métodos *Operacao()*: `imprimirtitulo()`,`imprimirautor()` e `imprimirconteudo()` que acessam as operações básicas *ImplOperacao()* chamadas `imprimircabecalho()`, `imprimirtexto()` e `imprimirfinal()` definidas na interface `ImplementacaoDocumento` e implementadas de duas formas: para documento HTML (`ImplementacaoDocHTML`) e documento comum (`ImplementacaoDocComum`).

```
abstract class Documento  {
/*  Classe Abstracao, base para a criação de documentos de
 *  diversos tipos   */

   protected String titulo;
   protected String autor;
   protected String conteudo;

   private ImplementacaoDocumento impdoc;

    public  Documento(ImplementacaoDocumento  impdoc,String  tit,  String
aut, String cont){
     this.impdoc = impdoc;
     titulo = tit;
     autor = aut;
     conteudo = cont;

    }

/* A seguir temos três operações utilizadas para criação dos tipos de
documento  */

    public String imprimirtitulo(){
```

```
/* Utiliza operação primitiva imprimircabecalho()  da classe de
implementação  ImplementacaoDocumento */
      return impdoc.imprimircabecalho(titulo);



    }

    public String imprimirautor(){

/* Utiliza operação primitiva imprimirtexto()  da classe de
implementação  ImplementacaoDocumento */

      return impdoc.imprimirtexto(autor);


    }

    public String imprimirconteudo(){
/* Utiliza operações primitivas imprimirtexto() e imprimirfinal()  da
interface  de implementação  ImplementacaoDocumento */
      return impdoc.imprimirtexto(conteudo)+impdoc.imprimirfinal();

    }




}



class DocumentoArtigo extends Documento{

/* Especialização de Documento: cria documento sem formatação */


      public  DocumentoArtigo(ImplementacaoDocumento  impdoc,String  tit,
String aut,  String cont){
       super(impdoc,tit,aut,cont);

      }

       public String  imprimir(){

       String                          resumo                          =
imprimirconteudo().substring(1,conteudo.length()/5);
       return    imprimirtitulo()+imprimirautor()    +    resumo    +
imprimirconteudo();


       }
}

class DocumentoTutorial extends Documento {
```

```
/* Especialização de Documento: cria documento com formatação HTML*/


public   DocumentoTutorial(ImplementacaoDocumento   impdoc,String   tit,
String aut, String cont){
       super(impdoc,tit,aut,cont);
}

public String  imprimir(){
       return imprimirtitulo()+imprimirautor() + imprimirconteudo();
}




}



interface ImplementacaoDocumento{

/* Interface para a criação dos diversos tipos de implementação de
documentos  */


/*Operações primitivas de implementação de documentos. Correspondem ao
método   ImplemOperacao() */

  public String imprimircabecalho(String texto);

  public String imprimirtexto(String texto);

  public String imprimirfinal();


}

class ImplementacaoDocComum implements ImplementacaoDocumento{
/* Primeira ImplementadoraConcreta: criação de documentos comuns, sem
formatação */


  public String imprimircabecalho(String texto){

   return texto +  "\n";

  }

  public String imprimirtexto(String texto){

  return texto +  "\n";

  }

  public String imprimirfinal(){
   return " ";
```

```
  }


}

class ImplementacaoDocHTML implements ImplementacaoDocumento{

/* Segunda ImplementadoraConcreta: criação de documentos com formatação
HTML*/

 public String imprimircabecalho(String texto){
return
 "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>"+
"<html>"+
"<head>"+
"<title>"+
texto+"</title>"+
"<meta  http-equiv='Content-Type'  content='text/html;  charset=iso-8859-
1'>"+
"</head>"+
"<body>"+
"<div align='center'>"+
"<p><font size='7'><strong>"+
texto+ "<br>"+
"</strong><strong></strong></font><strong><br>"+
"<br>";


 }

  public String imprimirtexto(String texto){
   return
   "<p align='left'>"+
   texto+"</p>";


  }

  public String imprimirfinal(){
   return
   "</div>"+
   "</body>" +
   "</html>";

  }



}



public class Main {

/* Acessa a interface da classe Documento (Abstracao), para a criação
de 4 Tipos de documentos */
```

```
    public static void main(String[] args) {


/* Cria as instâncias das classes de Implementação */
        ImplementacaoDocumento id1 = new ImplementacaoDocComum();
        ImplementacaoDocumento id2 = new ImplementacaoDocHTML();


/* Cria as instâncias das classes Documento, correspondentes aos 4
tipos de documento. */
        DocumentoArtigo      da1   =   new   DocumentoArtigo(id1,"Padrão
Bridge","Pasteur O.M. Jr.","Documento tipo artigo, comum");
        DocumentoArtigo    da2    =    new    DocumentoArtigo(id2,"Padrão
Bridge","Pasteur O.M. Jr.","Documento tipo artigo, HTML");
        DocumentoTutorial   dt1   =   new   DocumentoTutorial(id1,"Padrão
Bridge","Pasteur O.M. Jr.","Documento tipo tutorial, comum");
        DocumentoTutorial   dt2   =   new   DocumentoTutorial(id2,"Padrão
Bridge","Pasteur O.M. Jr.","Documento tipo tutorial, HTML");

/* Exibe  os 4 tipos de documento criados através do método imprimir */

        System.out.println(da1.imprimir());
        System.out.println(da2.imprimir());
        System.out.println(dt1.imprimir());
        System.out.println(dt2.imprimir());







    }
}
```

3.3- O padrão decorator (decorador)

Objetivo: Estender a responsabilidade de um objeto dinamicamente, em tempo de execução, sem a necessidade de criação de subclasses.

Problema típico: Suponha que um determinado objeto deva possuir novas responsabilidades em determinado momento de seu ciclo de utilização. A questão é: com fazer com que estas responsabilidades não sejam alocadas a todos os objetos da classe à qual pertence, já que elas são específicas dele, ou seja, como adicionar responsabilidades ao objeto quando o mesmo já foi criado?

Solução: Criar uma classe que será a mãe da classe à qual o objeto pertence e de toda e qualquer responsabilidade adicional, que será chamada de classe *decoradora.* As responsabilidades adicionais serão derivadas desta classe. Estas farão referência à classe do objeto, adicionando o comportamento desejado. Na Figura 8 temos:

Componente: define a interface de objetos que devam ter responsabilidades adicionadas dinamicamente.
ComponenteConcreto: define a classe do objeto que deva ter responsabilidades adicionadas dinamicamente.
Decoradora: referencia o *Componente,* e provê a base para as classes que adicionam responsabilidades ao mesmo. Acessa o método *Operacao* de *Componente* que terá sua responsabilidade incrementada.
DecoradoraConcreta: adiciona responsabilidades a *Componente.* Especializa o método *Operacao* da classe *Decoradora,* de forma que ele acrescente uma responsabilidade adicional ao componente executando o método *ComportamentoAdicionado.*

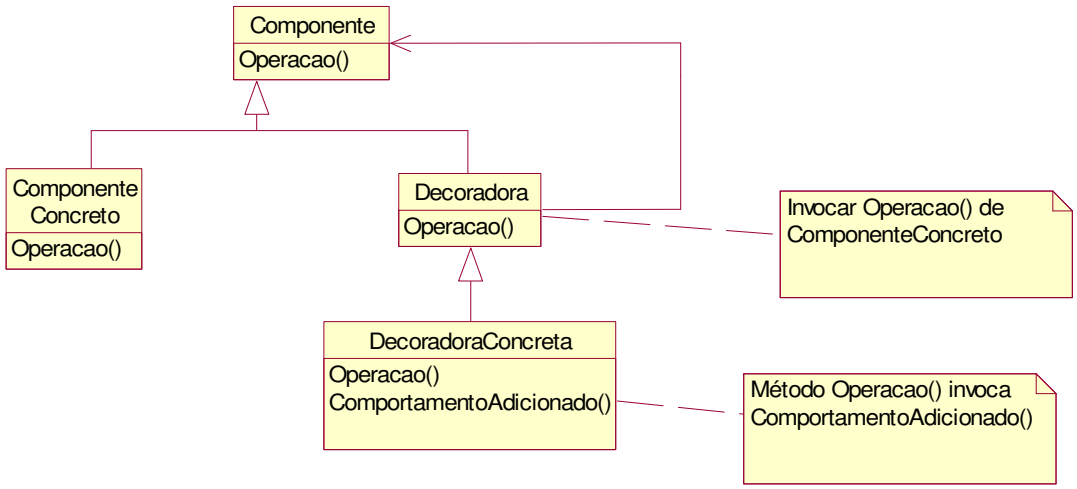

Figura 8 – Padrão Decorator

Exemplos de Utilização:

-Um editor de figuras pode acrescentar características adicionais às mesmas de acordo com o desejo do usuário, em tempo de execução.
-Em um sistema de controle de acesso, responsabilidades podem ser adicionadas ao usuário de acordo com o seu perfil: usuário comum, administrador, desenvolverdor, etc.
-Em um sistema de vendas objetos da classe *pagamento* podem receber responsabilidades adicionais em tempo de execução, de acordo com o tipo de pagamento selecionado, por exemplo, emitir boletas (pagamento por boleta), emitir parcelas (pagamento parcelado), etc.

Uma Implementação em Java:

Neste exemplo, temos dois objetos visuais, um botão e um rótulo.Vamos criar duas classes decoradoras, uma para acrescentar uma borda a eles e outra para

acrescentar um fundo vermelho. Estas características são adicionadas a eles em tempo de execução.

```
abstract class ComponenteVisual {
/** Classe Componente   **/


protected JComponent comp;

/* Operação que será especializada nas subclasses  */

public abstract JComponent desenhar();

}

/*  Classe ComponenteConcreto Botao derivada da ComponenteVisual  */

class Botao extends ComponenteVisual {



/* Retorna a instância de um  botão a ser desenhado na interface */

public JComponent desenhar(){

 return comp;
}


/* Construtor da classe ComponenteConcreto Botao */

public   Botao(String titulo){
   comp = new JButton(titulo);
}

}

/*  Classe ComponenteConcreto Rotulo derivada da ComponenteVisual  */

class Rotulo extends ComponenteVisual {



/* Retorna a instância de um  rotulo a ser desenhado na interface */

public JComponent desenhar(){

 return comp;
}


/* Construtor da classe ComponenteConcreto Rotulo */
```

```
public   Rotulo(String titulo){
   comp = new JLabel(titulo);

}

}



abstract class Decoradora extends ComponenteVisual{

/*Classe Decoradora abstrata da qual serão especializadas todas as
decorações*/


protected ComponenteVisual componente;

public JComponent desenhar(){
    return componente.desenhar();

}

public Decoradora(ComponenteVisual componente){

  this.componente = componente;

}




}

class DecoradoraBorda extends Decoradora {

/* Esta classe acrescenta uma borda ao componente visual */

/*Método desenhar() (operacao()) especializado para colocar a borda */

public JComponent desenhar(){
      componente.comp= componente.desenhar();
      return colocarborda();

}


/*  Método  que  desenha  borda  ao  componente,  (é  o  método
ComportamentoAdicionado) */

private JComponent colocarborda(){

    Border raisedbevel = BorderFactory.createRaisedBevelBorder();
            Border                    loweredbevel                    =
BorderFactory.createLoweredBevelBorder();

      componente.comp.setBorder(BorderFactory.createCompoundBorder(rais
edbevel, loweredbevel));
```

```
return componente.comp;


}

public DecoradoraBorda(ComponenteVisual componente){
     super(componente);
}

}

class DecoradoraFundo extends Decoradora {

/* Esta classe acrescenta cor de fundo ao componente visual */

private Color cor;

/*Método desenhar() (operacao()) especializado para colocar a cor de
fundo */

public JComponent desenhar(){
      componente.comp = componente.desenhar();
      return modificarfundo();

}

/* Método que insere a cor de fundo ao componente (é o método
ComportamentoAdicionado)*/

private JComponent modificarfundo(){

       componente.comp.setBackground(cor);
       return componente.comp;


}

public DecoradoraFundo(ComponenteVisual componente, Color cor){
     super(componente);
     this.cor = cor;
}

}

public class TesteDecorator {


      public static void main(String[] args) {
    /* Criação da instância a ser decorada do objeto botão*/

    Botao botao =  new Botao("TESTE PADRAO DECORATOR-BOTÃO");

    /* Criação da instância onde será inserida a borda */

    DecoradoraBorda botaoborda = new DecoradoraBorda(botao);
```

```
    /*Criação da instância onde será inserido um fundo vermelho */

    DecoradoraFundo             botaofundo             =             new
DecoradoraFundo(botaoborda,Color.red);

    /* Mesma coisa para o objeto rotulo */

    Rotulo rot =  new Rotulo("TESTE PADRAO DECORATOR-ROTULO");
    DecoradoraBorda rotborda = new DecoradoraBorda(rot);
    DecoradoraFundo rotfundo = new DecoradoraFundo(rotborda,Color.red);

    JFrame frame = new JFrame("Padrão Decorator");

    frame.addWindowListener(new WindowAdapter() {
                  public      void      windowClosing(WindowEvent      e)
{System.exit(0);}
            });

    JPanel panel = new JPanel();



    /* Adiciono o botao */

    panel.add(botaofundo.desenhar());

    /* Adiciono o rotulo */

    panel.add(rotfundo.desenhar());


    frame.getContentPane().add(panel);

    frame.pack();
    frame.setVisible(true);

      }
}
```

3.4- O padrão flyweight ("peso-mosca")

Objetivo: Prover uma estrutura para a representação de classes que possuam um grande número de instâncias com características que se repetem com muita frequência.

Problema típico: Uma classe de objetos possui um grande número de instâncias com características comuns. Como conseguir economia de utilização de memória, fazendo com que características comuns não se repitam com freqüência?

Solução proposta: Criar um tipo de objeto, o *flyweight,* que tem a característica de poder ser compartilhado, e utilizado em múltiplos contextos simultaneamente. O flyweight encapsula o chamado estado *intrínseco* dos

objetos que represeta, ou seja, o estado que é comum a todos eles. Dependendo do contexto em que é utilizado, o flyweight assume estados que variam com este contexto, os chamados estados *extrínsecos*. Estes estados não são compartilhados e os clientes que utilizam os flyweights detêem a responsabilidade por passá-los a eles. Esta estrutura proporciona considerável economia de memória. Na Figura 9 temos:

Flyweight: é a classe abstrata através da qual os flyweights são manipulados em função dos estados extrínsecos, passados como parâmetro para o método *Operacao()*.
FlyweightConcreto: é a especialização da classe *Flyweight* que armazena o estado intrínseco.
FabricaFlyweights: classe responsável por criar os *Flyweights.*
Cliente: cria e mantém referências aos *Flywights,* além de computar e/ou armazenar os estados extrínsecos.

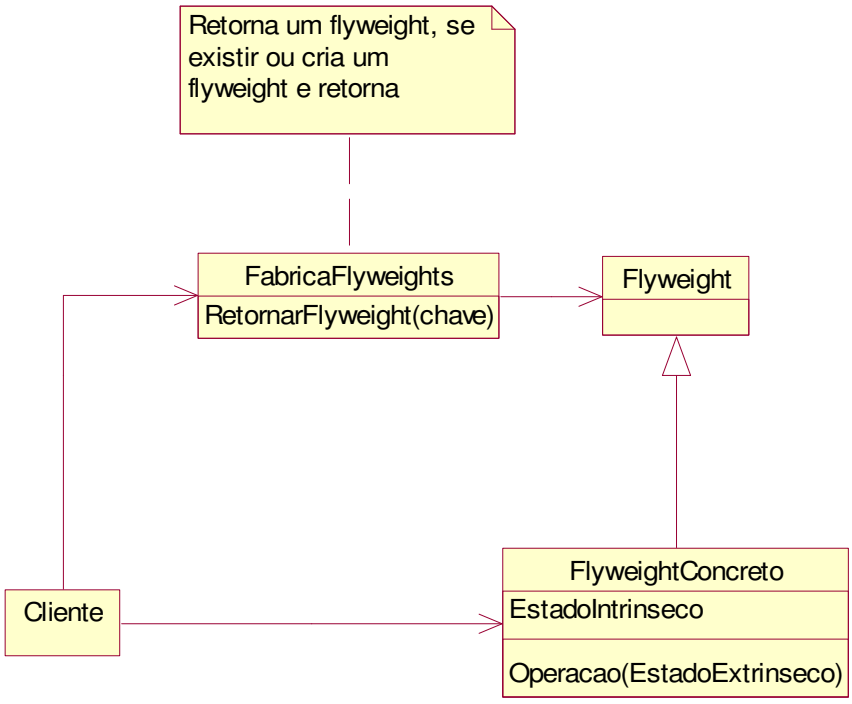

Figura 9-Padrão flyweight

Exemplos de utilização:

-Editores de texto podem manter flyweights contendo estados intrínsecos de caracteres (o código ascii, por exemplo).
-Editores de figuras, podem manter flyweights contendo pontos comuns (por exemplo, o formato).

Uma Implementação em Java:

Neste exemplo, vamos criar flyweights que vão compartilhar caracteres. O estado extrínseco será a tamanho do caracter em uma palavra. No teste, a palavra *paralelepipedo* é utilizada, de forma que sejam criados flyweights compartilháveis para as letras *p, a,r,l,e,i,d* e *o.* O programa cliente mantém um vetor contendo os estados extrínsecos (tamanho) dos caracteres. Repare que ao invés de criarmos 14 instâncias de caracteres da palavra, criamos apenas 9 flyweights que compartilham as ocorrências (flyweights) repetidas.

```
//Classe abstrata Flyweight

abstract class LetraFlyweight {

      public abstract void exibir(int tamanho);
}

//Flyweight Concreto: Letra HTML

class LetraHTMLFlyweight extends LetraFlyweight {

//O estado intrínseco: o caracter

char c;



      public LetraHTMLFlyweight(char c) {
            this.c = c;
      }

//Parâmetro tamanho é o estado extrínseco do flyweight
//Método exibir mostra o estado intrínseco do flyweight, utilizando o
//estado extrínseco tamanho.
      public void exibir(int tamanho) {
            System.out.print("<font
size='"+Integer.toString(tamanho)+"'>"+c);
      }
}

/* Fábrica de flyweights */

class FabricaLetra {


//Armazena o estado intrínseco do flyweight em uma tabela hash

      private Hashtable tabela = new Hashtable();

/* Método RetornaFlywight que retorna  instância do flyweight */

      public LetraFlyweight PegarLetraFlyweight(char c) {
```

```
            Character ch = new Character(c);
/* Se chave de acesso encontra caracter na tabela .... */
            if (tabela.containsKey(ch)) {
/* ... retorna o flyweight do caracter encontrado, */
                  return (LetraFlyweight) tabela.get(ch);
            } else {
/* senão,  cria um novo flyweight e o retorna. */
                  LetraFlyweight      flyweight      =            new
LetraHTMLFlyweight(c);
                  tabela.put(ch, flyweight);
                  return flyweight;
            }
      }
}
```

/*Classe cliente, que cria os flyweights e armazena estados extrínsecos */

```
public class TesteFlyweight {


      public static void main(String[] args) {
/*          Instância do texto a ser exibido */

  LetraHTMLFlyweight[] texto = new LetraHTMLFlyweight[14];

/* Vetor contendo o estado extrínseco de cada caracter, para que possa
ser exibido  posteriormente no tamanho correto*/

  int[] tamanholetra = new int[14];

/* Instanciar a fábrica de letras */

  FabricaLetra fl = new FabricaLetra();

/* Recuperar cada letra da palavra paralelepipedo */

  texto[0]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('p');
  tamanholetra[0]= 7;
  texto[1]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('a');
  tamanholetra[1]= 6;
  texto[2]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('r');
  tamanholetra[2]= 7;
  texto[3]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('a');
  tamanholetra[3]= 5;
  texto[4]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('l');
  tamanholetra[4]= 7;
  texto[5]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('e');
  tamanholetra[5]= 4;
  texto[6]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('l');
  tamanholetra[6]= 6;
  texto[7]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('e');
  tamanholetra[7]= 7;
  texto[8]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('p');
  tamanholetra[8]= 5;
```

```
  texto[9]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('i');
  tamanholetra[9]= 4;
  texto[10]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('p');
  tamanholetra[10]= 7;
  texto[11]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('e');
  tamanholetra[11]= 3;
  texto[12]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('d');
  tamanholetra[12]= 7;
  texto[13]  = (LetraHTMLFlyweight)fl.PegarLetraFlyweight('o');
  tamanholetra[13]= 5;

/* Exibe cada flyweight do texto paralelepipedo passando o estado
extrínseco como parâmetro. */

  for (int i = 0; i<=13 ; i++){

     texto[i].exibir(tamanholetra[i]);

  }

      }
}
```

3.5- O padrão proxy

Objetivo: prover uma forma rápida e leve de acesso a um objeto, que vai substituí-lo enquanto não for necessário acessar sua instância diretamente.

Problema típico: Suponha que um determinado objeto seja grande e por conseguinte ocupe muito espaço de memória. A questão é: como fazer para manter uma referência a este objeto, que não ocupe tanta memória, contendo apenas sua estrutura básica, de tal forma que o objeto somente seja criado e realmente acessado quando realmente for necessário?

Solução: a idéia é criar uma classe, *o Proxy,* que contém exatamente a mesma estrutura do objeto real, porém mais leve, sem a informação que realmente carrega a memória. Este *Proxy* somente acessará o objeto real quando ele for demandado na aplicação. Na verdade, o *Proxy* e o objeto real derivam de uma mesma classe-mãe, e é isto que garante que terão exatamente a mesma estrutura. Na Figura 10 temos:

Assunto: classe-mãe abstrata, da qual *Proxy* e *AssuntoReal* vão ser derivadas. Seus métodos serão implementados por ambas.
AssuntoReal: corresponde à classe do objeto real representado por *Proxy.*
Proxy : representa o *AssuntoReal.* Mantém uma interface idêntica à de *AssuntoReal* de forma que possa representá-lo indiferentemente. O acesso ao *AssuntoReal* é feito através da implementação da operação *Requisicao,* que vai acessar a mesma operação (*Requisicao)* implementada em *AssuntoReal.* O

objeto do tipo *AssuntoReal* somente vai ser instanciado quando for necessário e isto será feito por *Proxy.*

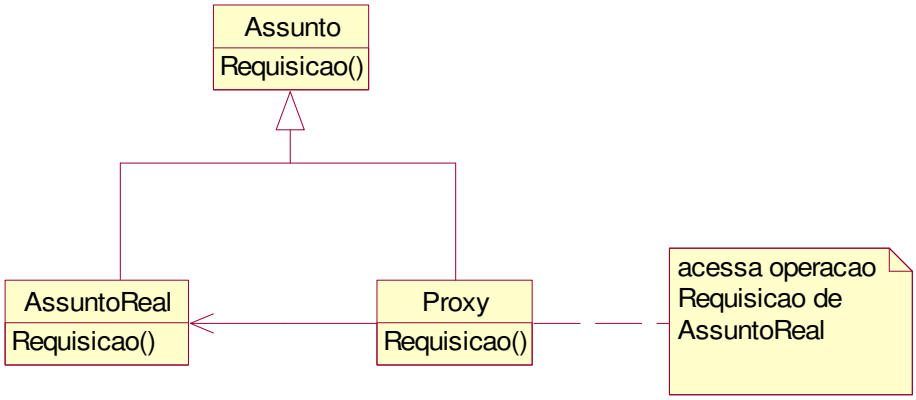

Figura 10-Padrão Proxy

Exemplos de utilização:

-Um editor de imagens pode manipular somente um *Proxy* da imagem enquanto ela não for carregada. Quando isto acontecer, via abertura de um arquivo, por exemplo, o *Proxy* passa a manipular a imagem diretamente, mas enquanto isto não ocorre, manipula apenas informações básicas, com tamanho, nome de arquivo, etc.
-No exemplo do sistema Revista Digital, pode ser criado um Proxy do documento, que vai manipular apenas informações básicas do mesmo (tamanho, nome, data de criação, etc) enquanto não houver necessidade de visualizá-lo. Quando isto for necessário, cria-se a instância real, abrindo seu arquivo e carregando-o na memória.

Uma Implementação em java:

Neste exemplo simples vamos criar um Proxy para manipular um texto simples. Este texto contém um conteúdo (uma string). O Proxy é criado e manipulado apenas com informações de tamanho do texto, enquanto este não for necessário. Quando há necessidade de exibir o texto via operação Imprimir(), a instância do texto é criada (via operação Imprimir()) e partir daí o objeto Proxy passa a manipular o texto real, exibindo seu conteúdo (a string) e tamanho.

```
abstract class Texto
/* Classe abstrata Assunto, que será especializada */

{

  /*Métodos que serão implementados pelo Proxy e pelo AssuntoReal */

  public abstract void Imprimir(String texto);
```

```
  public abstract int retornartamanho();

}

class TextoReal extends Texto {

/*Classe AssuntoReal */

private StringBuffer buffer;

public TextoReal(int tambuf){

/* cria o buffer que conterá o texto */

   buffer = new StringBuffer(tambuf);

}


/* Exibe  um texto na tela*/

public void Imprimir(String texto){


buffer.append(texto);

 System.out.println(buffer);
}

/* Retorna o tamanho do texto que foi alocado ao buffer */

public int retornartamanho(){

return buffer.length();

}

}

class TextoProxy extends Texto {

/* Classe Proxy, que manipula o objeto TextoReal. Ela somente cria a
instância de TextoReal quando precisa imprimi-lo. Enquanto isso, a
instância não existe */

/*Instância de texto real, que será criada somente quando necessário */

      private Texto textoreal=null;

/* tamanhobuffer será a informação que o Proxy terá antes de criar a
instância do  TextoReal */

      private int tamanhobuffer = 0;

 public int retornartamanho(){
    if (textoreal == null){
     return tamanhobuffer;
```

```
    }
    else {return textoreal.retornartamanho();}


 }
/* Construtor do proxy é inicializado apenas com o tamanho do texto a
ser manipulado */
/* Repare que ele não instancia ainda o TextoReal */
      public TextoProxy(int tambuf) {
            tamanhobuffer = tambuf;
      }


      public void Imprimir(String texto) {

  /* Cria a instância de TextoReal, se ela já não existir */

            if     (textoreal     ==     null)     textoreal     =     new
TextoReal(tamanhobuffer);

    /*Imprime o texto */

    textoreal.Imprimir(texto);
}
}


public class TesteProxy {

/*Classe cliente */

      public static void main (String[] args) {
/* Criar a instância do Proxy */

   Texto txtproxy = new TextoProxy(15);

/*Manipulo a instância do Proxy. Neste ponto, nenhuma instância de
texto foi criada */

   System.out.println("Não precisei ainda do texto, cujo tamanho máximo
será "+   txtproxy.retornartamanho());


/*Imprimo o texto. Veja que dentro do método Imprimir, a instância do
texto será criada. */

   txtproxy.Imprimir("TESTE CONCLUIDO");


    }
}
```

3.6- O padrão facade (Fachada)

Objetivo: Produzir uma interface mais simples para que um cliente possa acessar diversos métodos de um grupo de classes.

Problema Típico: Suponha que uma determinada classe necessite da colaboração de diversas outras através do acesso a seus métodos. Chamadas a diversos destes métodos são muito freqüentes e repetitivas, fazendo com que determinadas operações se tornem longas, demandando muita escrita de código. A questão é: como racionalizar e diminuir o acesso direto da classe cliente aos métodos destas classes?

Solução: Criar uma classe, a classe de *Fachada,* que vai possuir operações que chamam os métodos que são utilizados freqüente e repetidamente. O cliente passa a acessar estas operações, ao invés de acessar várias classes e várias chamadas de classes. Na Figura 11, temos:

Fachada: Classe que passa a ser a interface de *Cliente.* É através dela que *Cliente* acessa os métodos das classe *Classe1, Classe2, Classen.* Suas operações instanciam estas classes e invocam as operações das mesmas.
Classe1, Classe2, Classen: são as classes que provêem as operações utilizadas por *Cliente* através da classe *Fachada.*
Cliente: Acessa somente os métodos da classe *Fachada,* ao invés do métodos das classes *Classe1, Classe2, Classen.*

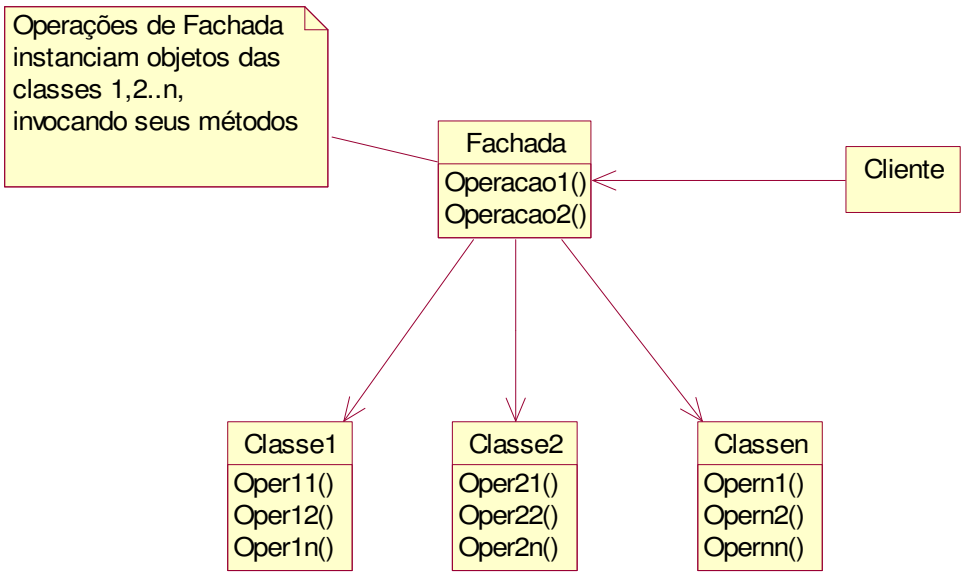

Figura 11-O padrão Facade

Exemplo de Utilização: As classes de controle já vistas anteriormente constituem o exemplo mais típico de classes de fachada.

Uma Implementação em Java:

O exemplo a seguir contém as classes *Formatadora* e *Transformadora,* que respectivamente formata um texto (negrito, itálico, centralizar, edentar e ajustar tamanho) e transforma o texto em maiúsculas ou minúsculas. Documentos acessam freqüente e repetidamente os métodos destas classes, de forma a produzir um documento HTML. Seqüências de acesso a estes métodos se repetem bastante de documento para documento, como criar um título e um subitem. Criou-se então a classe de fachada *TextoHTMLFacade,* que possui métodos capazes de realizar operações freqüentes: `RetornaTextoTitulo(), RetornaTextoItem() e RetornatextoComum()`.Qualquer documento (no exemplo, a classe *ExemploDocumentoHTML*) a ser criado, ao invés de invocar os diversos métodos das classes *Formatadora e Transformadora,* acessa apenas os métodos da classe *TextoHTMLFacade.* Desta forma, a criação é mais rápida e o código fica mais estruturado, facilitando manutenção posterior.

```
class Formatadora {

/* Classe que formata uma determinado texto, adicionando negrito e
itálico, centralizando,adicionando edentação e ajustando tamanho */


    public static String AdicionarNegrito(String texto) {
        return "<strong>"+texto+"</strong>";


    }

    public static String AdicionarItalico(String texto) {
        return "<em>"+texto+"</em>";
    }

    public static String AdicionarCentralizar(String texto) {
    return "<p align='center'>"+texto+"</p>";

    }

    public static String AdicionarEdentacao(String texto) {
    return "<blockquote><p>"+texto+"</p></blockquote>";


    }

    public static String AjustarTamanho(String texto, int tamanho) {

    return "<font size='"+Integer.toString(tamanho)+"'>"+texto+
    "</font>";
```

```
    }



}

class Transformadora {

/*Classe que transforma o texto para maiúsculas e minúsculas ou coloca
somente a primeira letra em maiúscula*/

  public static String TransformarMaiuscula(String s){

     return s.toUpperCase();


  }
  public static String TransformarMinuscula(String s){

     return s.toLowerCase();


  }

  public static String TransformarPrimeiraMaiuscula(String s){
  String st;



   st = s.toLowerCase();

   st= st.substring(0,1).toUpperCase()+ st.substring(1);
   return st;

  }



}

class TextoHTMLFacade {

/*Classe de fachada, que reduz a necessidade de acesso aos métodos das
classes acima */

public String RetornaCabecalho(String titulo){
String cabe =
"<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>"+
"<html>"+
"<head>"+
"<title>"+titulo+"</title>"+
"<meta  http-equiv='Content-Type'  content='text/html;  charset=iso-8859-
1'>"+
"</head>"+"<body>";
```

```
return cabe;

}

public String RetornaFinal(){

return "</body>";


}



public  String RetornaTextoTitulo(String texto){

/* Este método formata um típico título de documento HTML */

 String titulo;
 titulo = Transformadora.TransformarMaiuscula(texto);
 titulo = Formatadora.AdicionarNegrito(titulo);
 titulo = Formatadora.AdicionarCentralizar(titulo);
 return titulo;

}

public  String RetornaTextoItem(String texto){

/* Formata um item dentro de um documento HTML */

 String item;
 item = Transformadora.TransformarPrimeiraMaiuscula(texto);
 item = Formatadora.AjustarTamanho(item,2);
 item = Formatadora.AdicionarNegrito(item);
 item = Formatadora.AdicionarItalico(item);
 item = Formatadora.AdicionarEdentacao(item);
 return item;

}

public String RetornaTextoComum (String texto)  {
/*Formata um texto comum dentro de um documento HTML*/
 String txt;
 txt = Transformadora.TransformarPrimeiraMaiuscula(texto);
 txt = Formatadora.AdicionarEdentacao(txt);
 txt = Formatadora.AdicionarEdentacao(txt);
 txt = Formatadora.AjustarTamanho(txt,1);
 return txt;
}

}


class ExemploDocumentoHTML {

/* Classe cliente, criada para demonstrar a criação de um documento
HTML acessando apenas os métodos da classe facade */
```

```
public String MontarExemplo(){

/* Monta o documento usando somente os métodos da classe facade. Repare
como fica mais limpo o código, sem a necessidade de acessar os métodos
de nível mais baixo das classes Transmormadora e Formatadora */

TextoHTMLFacade  fac = new TextoHTMLFacade();
String doc;
doc = fac.RetornaCabecalho("Teste Padrão Facade");
doc = doc+fac.RetornaTextoTitulo("teste padrão facade");
doc = doc+fac.RetornaTextoItem("item 1");
doc = doc + fac.RetornaTextoComum("conteúdo do item 1");
doc = doc+fac.RetornaTextoItem("item 2");
doc = doc + fac.RetornaTextoComum("conteúdo do item 2");
doc = doc+fac.RetornaTextoItem("item 3");
doc = doc + fac.RetornaTextoComum("conteúdo do item 3");
doc = doc + fac.RetornaFinal();

return doc;

}


}


public class TesteFacade {

    public static void main(String[] args) {

    /* Apenas cria a instância do doc. HTML  e exibe */

    ExemploDocumentoHTML doc = new ExemploDocumentoHTML();
    System.out.println(doc.MontarExemplo());
    }

}
```

3.6- O padrão composite

Objetivo: Prover uma estrutura de árvore para representação eficiente de relações todo-parte.

Problema típico: Suponha uma estrutura de objetos relacionados a outros na forma de relações todo parte. A questão é: como fazer com que todos os objetos, tanto os "todos" como as "partes", possam ser manipulados de maneira idêntica por classes clientes, através de uma interface única?

Solução: O padrão Composite propõe uma classe abstrata comum para objetos "todo" e "parte", contendo operações comuns a eles. Os objetos "todo" contêm

listas de objetos "partes". Classes clientes acessam a mesma interface (a definida pela classe abstrata) tanto para objetos "todo" como para objetos "parte". Na Figura 12 temos:
Componente: declara a interface comum a objetos "todo" e "parte" através de operações comuns. As operações Adicionar(), Remover() e RetornarFilho() permitem manipular objetos filhos (respectivamente, adicionar um filho, remover um filho e retornar um filho) alocados em estruturas de árvore de forma a representar a relação todo-parte.
Folha: representa objetos que não possuem filhos na estrutura de árvore, portanto, as operações de acesso aos mesmos declaradas em *Componente* não são implementadas. Ele implementa apenas operações que não envolvam manipulação de filhos (representadas pelo método *operacao()*).
Composite: classe que representa objetos "todo", constituído por filhos que, como pode ser visto, são também pertencentes à classe *Componente*, podendo ser, portanto, do tipo *Folha* ou *Composite.*
Cliente: manipula objetos na árvore através da interface comum provida por *Componente.*

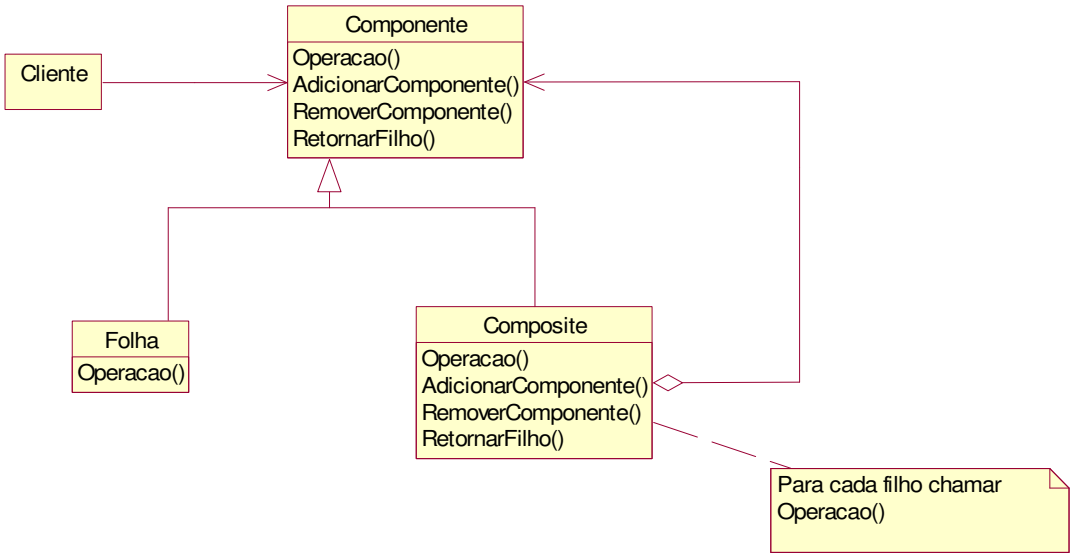

Figura 12-O padrão Composite.

Exemplos de utilização:

-Em uma interface gráfica é comum a composição de objetos mais simples em estruturas complexas. A aplicação cliente manipula tais objetos através de uma interface única.
-Extruturas de hiperdocumentos (como páginas HTML) envolvem documentos-folhas (que não têm link a outros documentos) e documentos compostos (que têm link a outros documentos).

Uma Implementação em Java: O exemplo a seguir simula a estrutura de uma árvore de documentos (hiperdocumentos) constituída de documentos-folhas (*DocumentoSimples)* e documentos compostos (*DocumentoComposto)* ambas derivadas de uma classe *Componente* abstrata *Documento*. A classe cliente *TesteComposite* cria as instâncias dos documentos e monta a hierarquia deles, imprimindo depois toda a estrutura da árvore formada.

```
abstract class Documento {

/* Classe Componente, base de Composite e Folha */


  protected String URL;



/* Operação a ser implementada nas classes filhas */

  public abstract void imprimirURL(String edent);


/* Adiciona um filho ao composite */

  public abstract void adicionarfilho(Documento documento);


/* Remove um filho do composite */

  public abstract void removerfilho(Documento documento);

/* Retorna o filho do composite dado seu índice */

  public abstract Documento retornarfilho(int index);

/*Retorna o númeto de filhos do composite */

  public abstract int numfilhos();
}

class DocumentoComposto extends Documento{

/* Implementação do Composite DocumentoComposto */

/* Lista de documentos filhos */

protected LinkedList docsfilhos = new LinkedList();

public DocumentoComposto(String URL) {
            this.URL = URL;
      }
```

```
public  void imprimirURL(String edent){

/* Operação para imprimir a URL do documento e de seus filhos */

/* Imprime a URL do documento */
   System.out.println(edent+URL);
/* Imprime a URL dos  filhos */
   for (int i = 0; i <= (numfilhos()-1); i++){
       retornarfilho(i).imprimirURL(edent+"-");
   }
}

  public  void adicionarfilho(Documento documento){
  this.docsfilhos.add(documento);
  }
  public  void removerfilho(Documento documento){
  this.docsfilhos.remove(documento);

  }


  public  Documento retornarfilho(int index){
   return (Documento) docsfilhos.get(index);

  }


  public  int numfilhos(){
   return docsfilhos.size();

  }
}


class DocumentoSimples extends Documento  {

/* Implementação da Folha DocumentoSimples */

public DocumentoSimples(String URL) {
            this.URL = URL;
      }


public  void imprimirURL(String edent){
/* Como é filho, imprime somente a URL */
   System.out.println(edent+URL);
}

public  void adicionarfilho(Documento documento){  }
public  void removerfilho(Documento documento){  }


public  Documento retornarfilho(int index){
   return null;  }
```

```
public  int numfilhos(){
return 0;
}

}

public class TesteComposite {

public static void main(String[] args) {
/* Montagem de uma árvore de composites */

 // Documento Raiz:
    DocumentoComposto raiz = new DocumentoComposto("c:/Raiz");
//Dois  filhos compostos do documento raiz:
    DocumentoComposto            filhocomp1            =            new
DocumentoComposto("/FilhoComp1");
    DocumentoComposto            filhocomp2            =            new
DocumentoComposto("/FilhoComp2");
    raiz.adicionarfilho(filhocomp1);
    raiz.adicionarfilho(filhocomp2);
//Dois documentos simples, que serão filhos do documento composto
//filhocomp1:
    DocumentoSimples  ds1 = new DocumentoSimples("/Simples1");
    DocumentoSimples  ds2 = new DocumentoSimples("/Simples2");
    filhocomp1.adicionarfilho(ds1);
    filhocomp1.adicionarfilho(ds2);
//Documento composto que será "neto" do documento raiz, filho de
//filhocomp2:
    DocumentoComposto netocomp = new DocumentoComposto("/netocomp");
    filhocomp2.adicionarfilho(netocomp);
//Documentos simples, que serão filhos de netocomp
    DocumentoSimples  ds3 = new DocumentoSimples("/Simples3");
    DocumentoSimples  ds4 = new DocumentoSimples("/Simples4");
    netocomp.adicionarfilho(ds3);
    netocomp.adicionarfilho(ds4);

 //Imprimir a árvore de documentos a partir do documento raiz:
    raiz.imprimirURL("");
  }
}
```

4-Padrões Comportamentais

Estão relacionados ao comportamento dos objetos, ou seja, com o estabelecimento de padrões de comunicação entre eles.

4.1-Padrão cadeia de responsabilidade (chain of responsibility)

Objetivo: Evitar o acoplamento entre o emissor e o receptor de uma requisição, de forma que mais de um objeto possa manusear esta requisição.

Problema típico: Suponha uma situação em que um objeto envia uma mensagem e esta mensagem não tem um destinatário fixo, ou seja este deve ser definido em determinado contexto. A requisição pode passar por diversos objetos. A questão é: como permitir que esta mensagem seja capturada pelo objeto correto?

Solução: Cada classe de objetos pelos quais a requisição vai passar é derivada de uma classe manipuladora, que possui um método que vai verificar se o objeto em questão é o destinatário. Se não for, a requisição é passada a outro objeto na cadeia, até que se encontre o destinatário. Na Figura 13 temos:

Manipulador: define uma interface para manipular requisições. Suas classes derivadas implementam o método *ManipularRequisicao.*
ManipuladorConcreto: Se puder manipular a requisição, ele o faz, senão, passa a requisição para outro manipulador (seu sucessor, para o qual tem um ponteiro).
Cliente: inicializa qualquer requisição para um *ManipuladorConcreto* na cadeia.

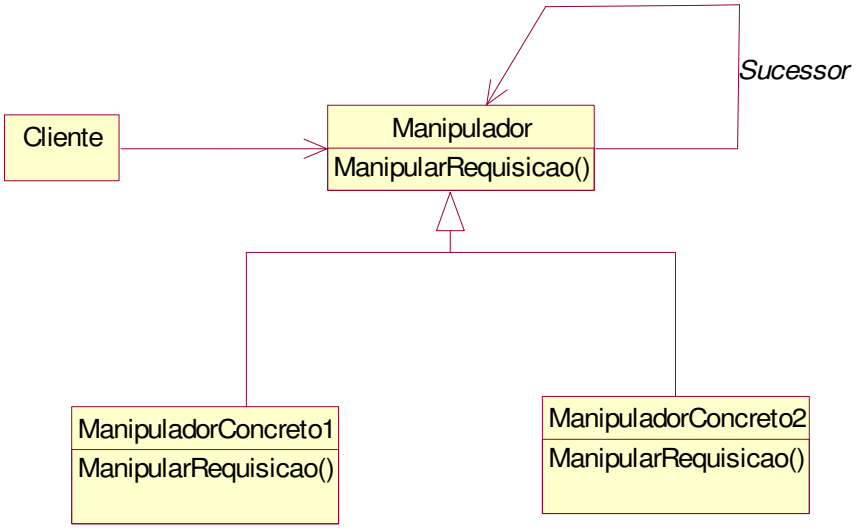

Figura 13-Padrão chain of responsibility

Exemplos de utilização:

-Sistemas de ajuda sensível ao contexto utilizam este padrão para determinar qual objeto está sendo referenciado.
-Sistemas de correio eletrônico podem utilizar este padrão para determinar uma cadeia de comunicação, transmitindo uma mensagem endereço por endereço até um manipulador final da mesma.

Uma Implementação em Java:

O exemplo a seguir apresenta uma interface muito simples constituída de um botão, um painel e um quadro. A cadeia é disparada pelo click no botão, que envia a requisição a seu sucessor (painel), que por sua vez envia a requisição ao quadro, que é o responsável por manipular a requisição.

```
package examples.chain2.java;

import javax.swing.*;
import javax.swing.JFrame;
import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.*;


interface Manipulador {
/* Classe Manipulador  */

      public void manipularRequisicao(String nomemanip);
}

class Botao extends JButton implements Manipulador {

/*Classe ManipuladorConcreto Botao invoca um sucessor caso não seja
o responsável pela requisição */



      protected Manipulador sucessor;


    public Botao(String s, Manipulador sucessor,String nomemanip,String
nome ) {
            super(s);
    this.setName(nome);
            this.sucessor = sucessor;
    final String  nm = nomemanip;
            this.addActionListener( new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        manipularRequisicao(nm);
                  }
            });
      }


      public void manipularRequisicao(String nomemanip) {
/*Método responsável por manipular a requisição vinda de um antecessor
*/


    /*Verifica se é o responsável pela requisição */
```

```
    if (nomemanip.equals(this.getName())){
    /* Se for, finaliza exibindo uma mensagem */
            System.out.println("Requisição recebida e FINALIZADA  por :
"+this.getName() );
    }
    else {
    /* Senão, passa a requisição para seu sucessor */
    System.out.println("Requisição recebida por : "+this.getName()+ " e
enviada para "+sucessor.getClass().getName());
    sucessor.manipularRequisicao(nomemanip);

    }
      }
}


class Painel extends JPanel implements Manipulador {

/*Segunda classe manipuladora, Painel, análoga à Botao */


      protected Manipulador sucessor;


      public Painel(Manipulador sucessor,String nome) {
            super();
    this.setName(nome);
            this.sucessor = sucessor;
      }

  public void manipularRequisicao(String nomemanip) {
    if (nomemanip.equals(this.getName())){
            System.out.println("Requisição recebida e FINALIZADA  por :
"+this.getName());
    }
    else {
    System.out.println("Requisição recebida por : "+this.getName()+ " e
enviada para "+sucessor.getClass().getName());
    sucessor.manipularRequisicao(nomemanip);

    }
      }



}


class Quadro extends JFrame implements Manipulador {

/*Terceira classe manipuladora, Quadro, análoga à Botao e Painel */


      protected Manipulador sucessor;
```

```
      public Quadro(String s,String nome) {

            super(s);
    this.setName(nome);

            this.addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e) {
                        System.exit(0);
                  }
            });
      }

  public void manipularRequisicao(String nomemanip) {
    if (nomemanip.equals(this.getName())){
            System.out.println("Requisição recebida e FINALIZADA  por :
"+this.getName());
    }
    else {
    System.out.println("Requisição recebida por : "+this.getName()+ " e
enviada para "+sucessor.getClass().getName());
    sucessor.manipularRequisicao(nomemanip);

    }
      }



}
```

```
public class Main {

/* Cliente que inicia a cadeia */

      public static void main(String[] args) {

    /*Quadro é o objeto que vai manipular a requisição. É o final da
cadeia */
            Quadro        quadro        =    new    Quadro("Chain    of
Responsibility","frame0");
    /* Painel é o segundo objeto da cadeia, quadro é o sucessor */
            Painel  painel  = new Painel(quadro,"painel0");
    /* Botão é o primeiro objeto da cadeia, seu sucessor é painel */
            Botao botao = new Botao("Padrão chain of responsibility",
painel,"frame0","botao0" );

            quadro.getContentPane().add(painel);
            painel.add(botao);

            quadro.pack();
            quadro.setVisible(true);
      }
}
```

REFERÊNCIAS

[GAM98} GAMMA, E, et al. *Design Patterns-Elements of Reusable Object-Oriented Software* – 1a Edição, Addison Wesley, 1998.

[KIK02] KIKZALES, Gregor. HANNEMAN, Jan *Design Patterns Implementations.* ca.ubc.cs.spl.patterns, 2002.