FACULDADE DE MINAS – FAMINAS-BH

CURSO DE SISTEMAS DE INFORMAÇÃO

ALINE DE SOUSA PEREIRA

PADRÕES DE PROJETO:
UMA COMPILAÇÃO DOS MAIS UTILIZADOS EM PROJETOS DE SOFTWARE

BELO HORIZONTE

2008

ALINE DE SOUSA PEREIRA

PADRÕES DE PROJETO:
UMA COMPILAÇÃO DOS MAIS UTILIZADOS EM PROJETOS DE SOFTWARE

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação, da Faminas-BH – Faculdade de Minas, como requisito de avaliação parcial para obtenção do Título de Bacharel em Sistemas de Informação.

Prof. Orientador: Ricardo Terra

BELO HORIZONTE

2008

Aline de Sousa Pereira

Padrões de projetos:
um estudo sobre sua aplicabilidade em projetos de software

Objetivo: demonstrar a funcionalidade e os benefícios da utilização de padrões de projeto no desenvolvimento de sistemas de software orientados a objetos.

FAMINAS-BH – Faculdade de Minas
Curso de Sistemas de Informação

Área de concentração: Desenvolvimento de sistemas de software

Data de aprovação: _____ / _____/ _____

_____

Prof. Ricardo Terra
Orientador

_____

Prof. Paulo Henrique Fernandes de Matos
Coordenador do Curso de Sistemas de Informação

Dedico este trabalho primeiramente a Deus, pois sem Ele nada seria possível. Ao meu noivo, pelo seu apoio e compreensão.

AGRADECIMENTOS

A Deus pela sabedoria e força concedidas ao longo do curso.

Ao meu noivo, Mateus, por toda paciência e companheirismo durante todo este tempo.

Aos professores, pelos ensinamentos oferecidos.

Ao meu orientador, Professor Ricardo Terra, pela firme orientação e compartilhamento de idéias.

A todos que direta ou indiretamente colaboraram na execução deste trabalho.

“Porque a sabedoria serve de defesa, como de defesa serve o dinheiro; mas a excelência do conhecimento é que a sabedoria dá vida ao seu possuidor.”

(Eclesiastes 7,12)

RESUMO

Padrões de projeto refletem soluções para problemas encontrados por desenvolvedores durante a fase de projeto de software. A sua utilização possibilita uma maior coesão e minimização da complexidade e do acoplamento entre os elementos que integram a aplicação, uma vez que utilizam eficientemente os conceitos de orientação a objetos. Isso implica que a correta aplicação de padrões de projeto melhora a qualidade, flexibilidade e manutenibilidade do software. Além disso, facilitam o trabalho dos desenvolvedores de software, pois apresentam soluções eficazes para problemas conhecidos. Os padrões de projeto são classificados em criacionais, estruturais e comportamentais. Os padrões criacionais estão relacionados com a criação de objetos, enquanto que os padrões estruturais referem-se a composição de objetos e classes e, por fim, os padrões comportamentais lidam com interação e compartilhamento de responsabilidade entre objetos e classes. Portanto, o objetivo deste trabalho é estudar os mais utilizados padrões de projeto, destacando a intenção, motivação, aplicabilidade e conseqüências da utilização de cada um deles.

Palavras-chave: padrão de projeto; orientação a objetos; acoplamento; solução.

ABSTRACT

Design patterns are recurring solutions to software design problems found by developers during a phase of software development. These designs provide a higher cohesion and the reducing of the complexity and the coupling between the elements that integrate the application, since they efficiently use the object orientation concepts. Thus, the correct application of design patterns, improves the quality, flexibility and maintainability of the software. Furthermore, facilitating several developers tasks, because they are effective solutions to known problems. The design patterns are classified as creational, structural, and behavioral. Creational design patterns are related to instantiation of objects, while the structural design patterns are related to the composition of classes and objects, and finally the behavioral design patterns are related to the interaction and responsibilities of the classes and objects. Therefore, the objective of this work is to study the most used design patterns, highlighting the intention, motivation, applicability, and consequences of use of each one.

keywords: design patterns; object oriented; coupling; solution.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Desenvolvedores de software, freqüentemente, deparam-se com problemas que já foram resolvidos em projetos anteriores. Porém, a falta da catalogação de soluções de projeto faz com que não saibam onde e como resolveram tal problema.

Os padrões de projeto (*design patterns*, em inglês) aparecem aos projetistas de software orientado a objetos como uma ajuda na resolução de problemas baseados em soluções que já funcionaram em projetos anteriores.

Com a utilização da orientação a objetos, surge a necessidade de seguir um processo detalhado que facilite o entendimento e a manutenção do software, que possibilite a reutilização de códigos e que evite a necessidade de se recriar uma solução existente.

Muitos desenvolvedores de sistemas de software, principalmente os menos experientes, não realizam bons projetos e, muitas vezes, tendem a utilizar técnicas não-orientadas a objetos. Diante disso, os padrões de projeto possibilitam maior qualidade de código, manutenibilidade, flexibilidade, entre outros.

As primeiras idéias sobre padrão de projeto foram propostas no contexto da arquitetura civil, porém os profissionais de software incorporaram esses princípios na criação de padrões de projeto para desenvolvimento de software orientado a objetos.

Gamma *et al*. (2000, p. 20) definem que padrões de projeto “são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular”. De forma semelhante, Alur, Crupi e Malks (2004) afirmam que qualquer definição dada a padrões de projeto relaciona o padrão como uma solução para um determinado problema em um contexto específico.

Na Engenharia de Software, segundo Filho (2003), um processo terá subdivisões que possibilitem avaliar o andamento de um projeto e corrigir seus rumos quando ocorrem problemas se ele for bem definido. Diante disso, uma das medidas a serem adotadas para alcançar tal expectativa é a utilização dos padrões de projeto.

Gamma *et al*. (2000) descrevem os padrões de projeto com as seguintes características:

- **Nome:** é usado para identificar o padrão;
- **Intenção:** representa o propósito do padrão de projeto;
- **Motivação:** descreve um cenário que contém um problema que o padrão irá resolver;
- **Aplicabilidade:** descreve as situações nas quais os padrões podem ser aplicados;
- **Estrutura:** representação gráfica através de diagramas de classe e seqüência para demonstrar as classes do padrão e a relação entre os objetos;
- **Participantes:** entidades que participam do padrão;
- **Colaborações:** descreve como as classes envolvidas colaboram para realizar suas tarefas;
- **Conseqüências:** avaliação dos resultados que serão obtidos com a aplicação do padrão;
- **Implementação:** como o padrão deve ser implementado;
- **Exemplo de código:** códigos que ilustram a implementação do padrão;
- **Usos conhecidos:** demonstração de casos de sistemas reais em que os padrões foram utilizados;
- **Padrões relacionados:** descreve o relacionamento existente entre os padrões.

Conforme propõem Gamma *et a*l. (2000) e demonstrado no Quadro 1, os padrões de projeto são ainda classificados por dois critérios: propósito e escopo. O primeiro divide os padrões de projeto em três categorias: criacional, estrutural e comportamental. Enquanto o segundo especifica se o padrão aplica-se a objetos ou classes.

Quadro 1 A classificação dos padrões de projeto

		Propósito		
		De criação	Estrutural	Comportamental
Escopo	Classe	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: GAMMA *et al*., 2000, p.26.

Este estudo irá abordar a descrição dos padrões de projeto realizada pela GoF[1], na qual foram documentados uma coleção de padrões que servem como referência para muitos autores como Shalloway e Trott, Alur, Crupi e Malks, entre outros.

O objetivo geral deste trabalho é demonstrar a funcionalidade e os benefícios da utilização de padrões de projeto em desenvolvimento de software orientados a objetos. Para isto, este estudo:

- Apresentará os padrões de projeto mais utilizados catalogados pela GoF;
- Descreverá o propósito para utilização de cada padrão abordado, demonstrando os problemas resolvidos por ele, as circunstâncias às quais ele se aplica e os resultados obtidos com a sua utilização.

Por se tratar de um tema pouco abordado no ambiente acadêmico, sendo mais conhecido e utilizado por projetistas de sistemas de software mais experientes, acredita-se que o desenvolvimento deste estudo proporcionará maior conhecimento sobre o assunto, destacando as suas necessidades de utilização mais comuns.

Cada capítulo deste trabalho, apresenta um estudo aprofundado dos padrões de projeto documentados pela GoF, abordando as seguintes características: intenção, motivação, aplicabilidade e conseqüência, bem como uma demonstração, em código Java, da implementação de cada padrão.

O Capítulo 2 apresenta os padrões estruturais mais utilizados. O Capítulo 3 aborda as características dos padrões comportamentais, apresentando os padrões

[1] *GoF - Gang of Four*: refere-se aos quatro autores (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides) responsáveis pela popularização dos padrões de projeto de software através do livro Padrões de Projeto: soluções reutilizáveis de software orientado a objetos.

mais comuns nessa categoria. O Capítulo 4 apresenta os padrões de criação, descrevendo as características dos mais utilizados. Por fim, no Capítulo 5, serão expostas as considerações finais.

2 PADRÕES DE ESTRUTURA

Segundo Gamma *et al*. (2000), os padrões estruturais possuem como objetivo a definição da composição de objetos e classes na formação de estruturas maiores.

Ainda conforme Gamma *et a*l. (2000), os padrões estruturais podem ser aplicados a classes ou objetos. Enquanto os padrões estruturais de objetos demonstram maneiras de complementar objetos para adquirir novas funcionalidades, os padrões estruturais de classe usam herança para definir implementações ou interfaces.

2.1 *Adapte*r

- **Intenção**

A intenção do padrão *Adapter* pode ser definida em:

> Converter a interface de uma classe em outra interface, esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível. (GAMMA *et al*., 2000, p.140).

Diante dessa definição, Shalloway e Trott (2004) concluem que utiliza-se o *Adapter* quando é necessária a criação de uma nova interface para um determinado objeto que funciona de forma semelhante, porém possui uma interface incompatível. A Figura 1 associa a comunicação entre a classe *Cliente* e a *ClasseExistente*, que só é possível através da utilização do padrão *Adapter,* a uma situação em que é necessário conectar um plugue à uma tomada incompatível, nesse caso o encaixe seria impossível se não fosse à utilização de um adaptador.

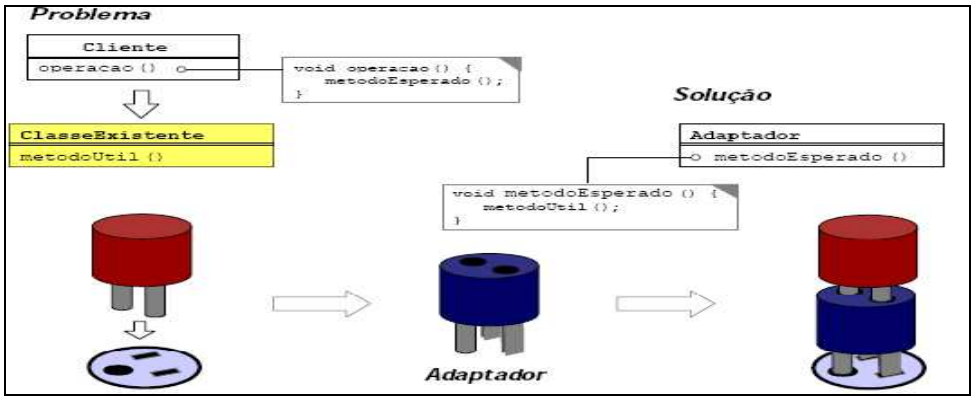

Figura 1 Representação do padrão *Adapter*.
Fonte: ROCHA, 2003.

- **Motivação**

Para exemplificar a motivação da utilização do padrão *Adapter*, a Figura 2 ilustra uma interface denominada *FiguraGeometrica* e suas derivadas que representam triângulos e retângulos.

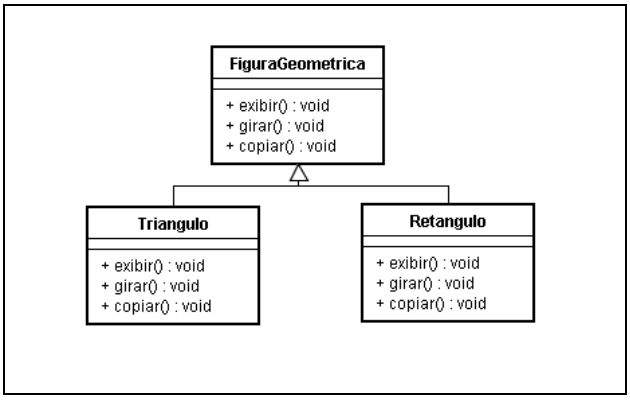

Figura 2 Subclasses de F*iguraGeometrica*.
Fonte: Adaptado de SHALLOWAY e TROTT, 2004, p.116.

Nesse contexto, suponha que foi solicitada a implementação de um losango, ou seja, um novo tipo de *FiguraGeometrica* e que a classe que trabalha com losangos já foi escrita anteriormente e denominada *FiguraLosango*, porém com nomes diferentes para os métodos, conforme ilustrado na Figura 3.

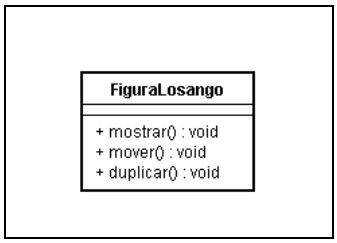

Figura 3 A classe *FiguraLosango*
Fonte: Adaptado de SHALLOWAY e TROTT, 2004, p.117.

A classe *FiguraLosango* não pode ser usada diretamente, porque os nomes dos métodos devem ser iguais aos de *FiguraGeometrica* e aquela deve ser derivada dessa.

Fazer essas alterações seria trabalhoso, pois acarretaria mudanças em todos os objetos criados anteriormente que utilizam a classe *FiguraLosango*. Para solucionar isso, utiliza-se então o padrão *Adapter* que permite a criação de uma nova classe que implemente a interface *FiguraGeometrica*, conforme pode ser observado na Figura 4. O Exemplo 1 contém trechos de código Java demonstrando a implementação do padrão *Adapter* para o contexto apresentado.

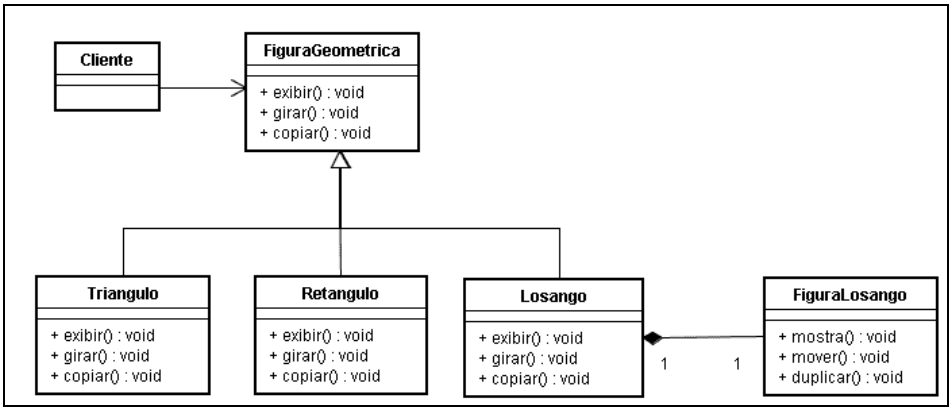

Figura 4 Utilizando o padrão *Adapter*
Fonte: Adaptado de SHALLOWAY e TROTT, 2004, p.118.

```
public class Losango extends FiguraGeometrica {
...
   FiguraLosango figuraLosango = new FiguraLosango();
...
      public void  exibir () {
         figuraLosango.mostra();

      }

  }
}
```

Exemplo 1 Trecho de código Java – implementando o padrão *Adapter*
Fonte: Adaptado de SHALLOWAY e TROTT, 2004, p.124.

Nessa representação, qualquer solicitação feita à um objeto da classe *Losango* será repassada para um objeto da classe *FiguraLosango*, que por sua vez, envia todas as respostas para *Losango*. Então, pode-se deduzir que a nova classe criada, a *Losango*, empacota os métodos da classe *FiguraLosango*, fazendo com que *Losango* contenha um objeto de *FiguraLosango*.

- **Aplicabilidade**

Segundo Gamma *et a*l. (2000), o *Adapter* deve ser usado quando for desejável utilizar uma classe existente, mas sua interface diferir da interface necessária ou quando deseja-se criar classes reutilizáveis que colaborem com classes que não tenham, necessariamente, interfaces compatíveis.

- **Conseqüências**

Segundo Shalloway e Trott (2004), com o *Adapter* pode-se utilizar objetos criados anteriormente em atuais estruturas de classes, eliminando a limitação de suas interfaces.

2.2 *Decorator*

- **Intenção**

Segundo Gamma *et al*. (2000), o padrão *Decorator* agrega dinamicamente novas funções a um objeto. Oferecem uma alternativa flexível ao uso de herança para extensão de funcionalidades.

Nesse sentido, Shalloway e Trott (2004) complementam que o padrão *Decorator* permite a criação de uma cadeia de objetos que inicia com os objetos responsáveis pela nova função, os *Decorators*, e termina com o objeto original, conforme pode ser observado na Figura 5.

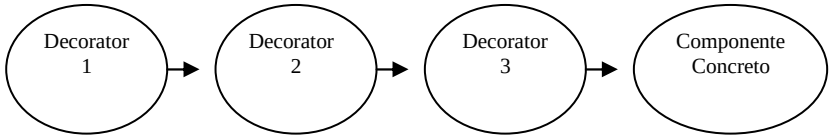

Figura 5 Cadeia de objetos do padrão Decorator
Fonte: SHALLOWAY e TROTT, 2004, p.246.

- **Motivação**

O *Decorator* é usado quando se necessita adicionar novas funcionalidades a um objeto. Em Java, por exemplo**,** utiliza-se muito o padrão *Decorator* nos objetos de entrada e saída de dados (E/S). O Exemplo 2 contém trechos de um código Java que realizada leitura de um arquivo no qual a classe *Reader*, responsável pela leitura *caracter* a *caracter* do arquivo, recebe a funcionalidade de ler o arquivo linha a linha pela classe *LineNumberReader*.

```
public static void main(String[] args) {
   try {
      File file = new File("arquivoTexto.txt");
      Reader in = new FileReader(file);
      LineNumberReader reader = new LineNumberReader(in);
      while (reader.ready()) {
           System.out.print("Linha " + reader.getLineNumber() + " --> ");
           System.out.println(reader.readLine());
      }
      reader.close();
      in.close();
   } catch (FileNotFoundException e) {
           System.out.println("Arquivo não encontrado. Causa: " + e.getMessage());
   } catch (IOException e) {
           System.out.println("Ocorreu um erro de E/S de dados. Causa: " + e.getMessage());
}
```

Exemplo 2 Padrão *Decorator*
Fonte: TERRA, 2008.

- **Aplicabilidade**

Conforme Gamma *et a*l. (2000), o padrão *Decorator* deve ser utilizado para adicionar responsabilidades a objetos sem afetar outros objetos. Ou ainda, em casos em que as responsabilidades podem ser excluídas ou quando a utilização de subclasses não é viável.

- **Conseqüências**

Segundo Gamma *et a*l. (2000), o *Decorator* fornece maior flexibilidade do que a utilização de herança. Porém, é preciso controlar a sua utilização, pois ele resulta na criação de uma quantidade elevada de pequenos objetos, o que pode dificultar a manutenção do código.

2.3 *Proxy*

- **Intenção**

Gamma *et al.* (2000) descrevem que a intenção do padrão *Proxy* é fornecer uma referência (substituto ou ponto de localização) para um objeto de tal forma que o acesso a esse objeto possa ser controlado.

- **Motivação**

De acordo com Gamma *et al*. (2000), o padrão *Proxy* controla o acesso a um objeto a partir de outro, o que é extremamente necessário quando se busca, por exemplo, retardar o custo da criação e inicialização de um objeto até o instante em que sua utilização é realmente necessária.

Para exemplificar a utilização do padrão *Proxy,* a Figura 6 demonstra a implementação em Java de um contexto em que a classe *Intermediario* mantém uma referência à classe *SujeitoReal* e ambas implementam a interface *Sujeito*. Assim, percebe-se, que a classe *Intermediario* passa a ser uma representante da classe *SujeitoReal,* podendo substituí-la caso a mesma esteja indisponível.

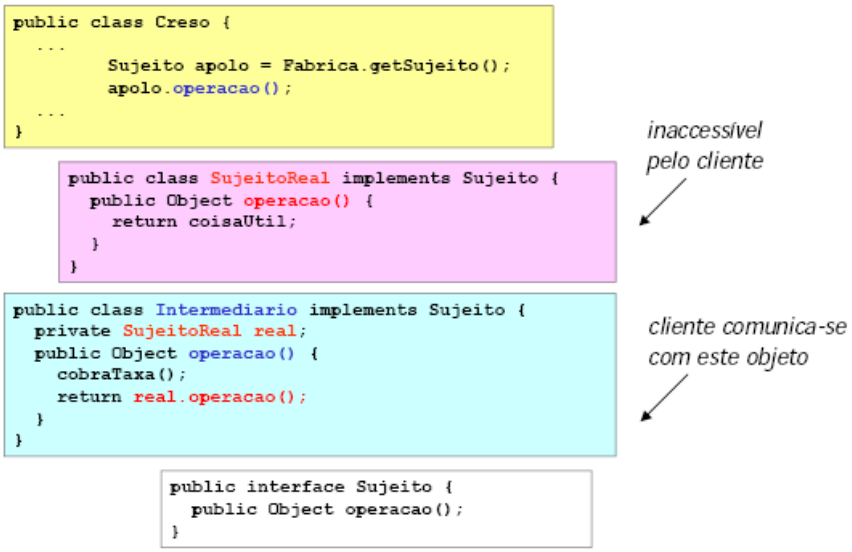

Figura 6 Utilizando padrão Proxy
Fonte: ROCHA, 2003.

A Figura 7 apresenta uma modelagem didática para o contexto representado pela Figura 6.

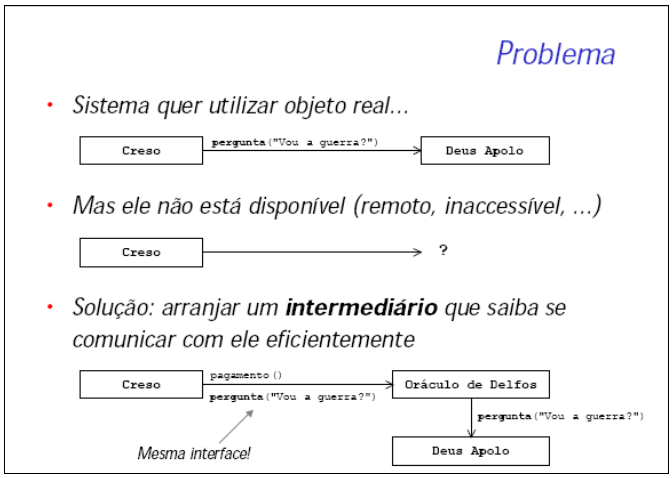

Figura 7 Modelo para representação do padrão *Proxy*
Fonte: ROCHA, 2003.

- **Aplicabilidade**

Segundo Gamma *et al.* (2000), quando um simples apontador de um objeto não é suficiente para a aplicação, utiliza-se o padrão *Proxy*, que fornece uma referência mais sofisticada do objeto. Para Rocha (2003), a utilização mais comum

do padrão *Proxy* é em objetos distribuídos, como *RMI*[2] *(Remote Method Invocation*) e *EJB*[3] *(Enterprise JavaBeans)***.**

- **Conseqüências**

Segundo Gamma *et al.* (2000), o uso do padrão *Proxy* possibilita o acesso a um objeto utilizando uma referência indireta a esse objeto, sendo que a utilização dessa referência varia conforme o tipo de *proxy*:

- *proxy* remoto: oculta o fato de objetos estarem em endereços diferentes.
- *proxy* virtual: possibilita a criação de objetos sob demanda.
- *proxy* de proteção: possibilitam tarefas adicionais de segurança quando um objeto é acessado.

2.4 *Façade*

- **Intenção**

Gamma *et al.* (2000, p. 179) descrevem que a intenção do padrão *Façade* consiste em fornecer uma interface unificada que represente um conjunto de interfaces em um subsistema, facilitando a sua utilização.

- **Motivação**

Conforme Shalloway e Trott (2004), a motivação do padrão *Façade* consiste na necessidade de diminuir a complexidade de interação com um sistema ou mesmo quando é necessário interagir com esse sistema de maneira particular, por exemplo, com um de seus subsistemas. Enfatizando esse aspecto, Gamma *et al.* (2000), destaca que a divisão de um sistema em subsistemas diminui a sua complexidade e a introdução do padrão *Façade* nesse sistema diminui a dependência entre eles.

---

[2] Segundo Bois (2008), RMI permite a comunicação entre objetos que estão em máquinas diferentes em uma rede.

[3] Segundo Santos (2008), EJB é um componente executado do lado do servidor, ou seja, não tem contato com o usuário e que encapsula toda a lógica de negócio da aplicação.

O padrão *Façade* “fornece uma interface única e simplificada para os recursos e facilidades mais gerais de um sistema.” (GAMMA *et al., 2000, p.179).* Fica evidente, portanto, que sua aplicação simplifica a utilização do sistema em que está sendo implementado.

Para exemplificar sua utilização, a Figura 8 contém um diagrama de classe no qual a classe denominada *Facade* se encarrega de divulgar à classe *Aplicacao* todas as operações que o sistema utiliza para realização de uma compra, criando assim, uma fachada do sistema. Desse modo a classe *Facade* conhece as classes responsáveis pela realização de cada operação.

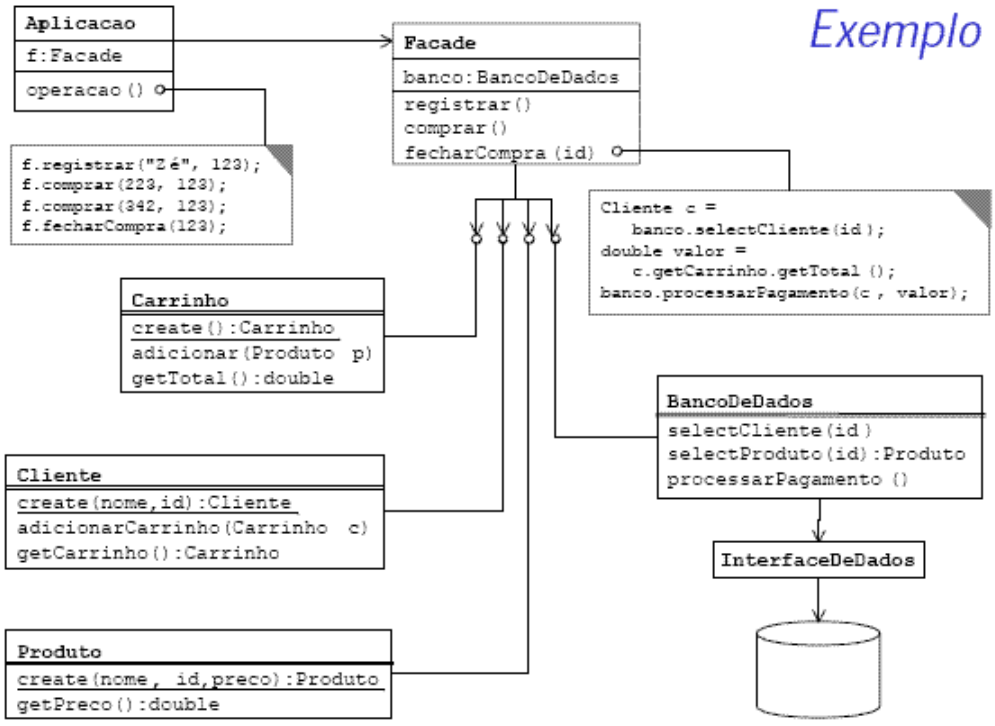

Figura 8 Diagrama de classe utilizando o padrão *Façade*
Fonte: ROCHA, 2003.

Percebe-se que a utilização do padrão *Façade* possibilita representar sistemas inteiros em somente um objeto, no contexto representado pela Figura 8, sem a utilização desse padrão a classe *Aplicacao* teria que conhecer todos os objetos do sistema. A Figura 9 contém a implementação em Java da utilização do padrão *Façade* para o contexto em questão.

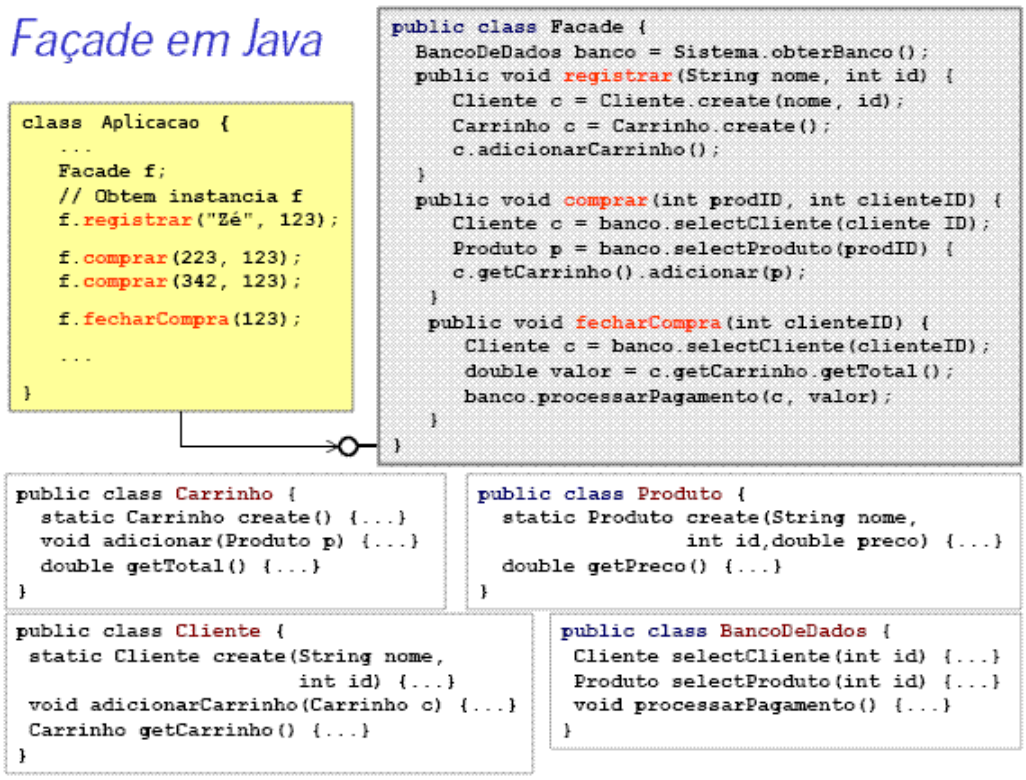

Figura 9 Implementação do padrão *Façade*
Fonte: ROCHA, 2003.

- **Aplicabilidade**

Segundo Rocha (2003), o padrão *Façade* deve ser aplicado quando for desejável facilitar o uso de uma aplicação através da criação de uma interface para um conjunto de objetos, fornecendo assim uma visão mais simples dessa aplicação.

- **Conseqüências**

Segundo Gamma *et al.* (2000), o padrão *Façade* oferece como benefício a redução do número de objetos com que os clientes lidam, uma vez que ele isola os clientes dos componentes do subsistema. Permite variar componentes do subsistema sem afetar os seus clientes, pois promove um acoplamento fraco entre os subsistemas e os clientes. Além de deixar disponível a utilização das classes contidas nos subsistemas para a aplicação.

2.5 *Composite*

- **Intenção**

A intenção do padrão *Composite* é “compor objetos em estruturas de árvores para representarem hierarquias partes-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos”. (GAMMA *et al*., 2000, p.160), conforme demonstra a Figura 10.

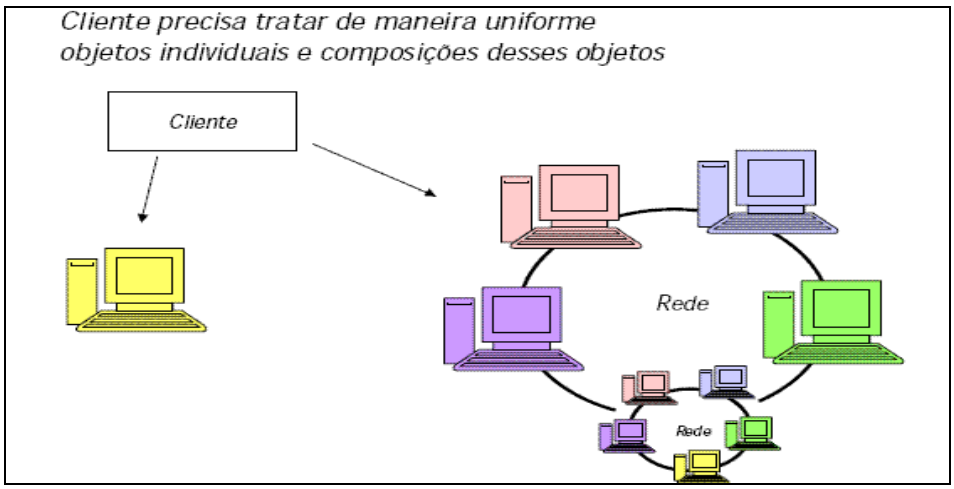

Figura 10 Representação do padrão *Composite*
Fonte: Rocha, 2003.

- **Motivação**

Gamma *et al.* (2000) descrevem que, em aplicações gráficas, um cliente que deseja desenhar alguma figura não precisa distinguir objetos primitivos como linhas e círculos de objetos compostos, que são combinações dos objetos primitivos. Nesse caso, o padrão *Composite* possibilita que o cliente não necessite fazer essa distinção, pois ele estabelece uma classe abstrata responsável por representar tantos os elementos primitivos quanto os compostos.

- **Exemplo de código**

O Exemplo 3 contém a codificação, em Java, de uma aplicação que implementa o padrão *Composite.* Essa aplicação objetiva calcular o preço de livros, discos e coleções de ambos de uma forma uniforme. A classe abstrata *Produto* define uma interface comum a todas as classes. As classes *Livro* e *Disco* são elementos primitivos, estendem de *Produto* e possuem seus respectivos cálculos de preço. Por fim, a classe *Composto* define um comportamento para aceitar um agregado desses objetos primitivos e seu preço é calculado através do somatório do preço de cada um desses objetos.

```
abstract class Produto {
   String referencia;
   String descricao;
   int custo = 0;

   Vector<Produto> componentes = null;

   boolean eComposto() {
      return false;
       }

   final boolean junta(Produto parte) {
      if (eComposto()) {
         if (componentes == null){
            componentes = new Vector< Produto >();
         }
         componentes.add(parte);
       }
       return eComposto();
   }

   abstract double preco();
}
```

```
class Livro extends Produto {
   final double iva=0.12;
   final double margem=0.9;

   String autor;
   String editor;
   int paginas;

   double preco() {
      return custo*(1+margem)*(1+iva);
   }
}
```

```
class Disco extends Produto {
   final double  iva=0.19;
   final double  margem=1.2;

   String autor;
   String editor;
   int duracao;
   int faixas;

   double preco() {
      return  custo*(1+margem)*(1+iva);
   }
}
```

```
class Composto extends Produto {
   boolean eComposto() { return true; }

   double preco() {
      double c=0;
         for (Produto produto: componentes){
            c += produto.preco();
         }
          return c;
   }
}
```

Exemplo 3 Padrão *Composite*
Fonte: LEAL, 2008.

- **Aplicabilidade**

Segundo Rocha (2003), deve-se usar o *Composite* toda vez que for necessário tratar um determinado conjunto de objetos como um objeto individual. Por sua vez, Gamma *et al.* (2000) acrescentam que o *Composite* também deve ser utilizado quando deseja-se determinar hierarquias partes-todo de objetos, ou seja, associar componentes a seus agregados.

- **Conseqüências**

Segundo Gamma *et al.* (2000), *o Composite,* através de objetos primitivos e objetos compostos, define hierarquias de classes. Simplifica o tratamento entre clientes com estruturas compostas e objetos individuais, pois permite que estes sejam tratados uniformemente. Além de facilitar a adição de novos componentes, pois toda estrutura utilizada para tratar os componentes já existentes funciona também para os novos componentes.

3 PADRÕES DE COMPORTAMENTO

Segundo Gamma *et al.* (2000), os padrões comportamentais descrevem a interação e atribuição de responsabilidades entre objetos e classes, sendo que, para determinar o comportamento entre classes, utilizam herança e, para comportamento entre objetos, utilizam composição de objetos.

3.1 *Observer*

- **Intenção**

Segundo Gamma *et al.* (2000), a intenção do padrão *Observer* é notificar e atualizar automaticamente os dependentes de um determinado objeto sempre que ocorrerem alterações neste objeto.

- **Motivação**

Gamma *et al.* (2000) descrevem que, em sistemas em que existam objetos dependentes entre si, é preciso manter a consistência entre esses objetos, ou seja, é necessário notificar qualquer mudança sofrida por um determinado objeto aos seus dependentes, sendo que, para facilitar a reutilização desses objetos, deve-se reduzir o acoplamento entre eles.

Ainda segundo Gamma *et al.* (2000), os *toolkits*[4] para construção de interfaces gráficas de usuário são exemplos da utilização do padrão *Observer,* pois eles mantêm a base de dados da aplicação separada dos componentes de apresentação da interface do usuário, porém, qualquer mudança ocorrida na base de dados deve ser notificada aos componentes de apresentação. A Figura 11 ilustra um contexto no qual os elementos de apresentação são representados por uma planilha e um gráfico de barras que, neste caso, são os observadores. Qualquer mudança ocorrida na base de dados deve ser notificada aos seus observadores.

[4] Segundo Gamma *et al*. (2000) *toolkits* constituem bibliotecas de classes, ou seja, coleções de classes que são reutilizáveis cujo objetivo é fornecer funcionalidades gerais para execução de tarefas em uma aplicação.

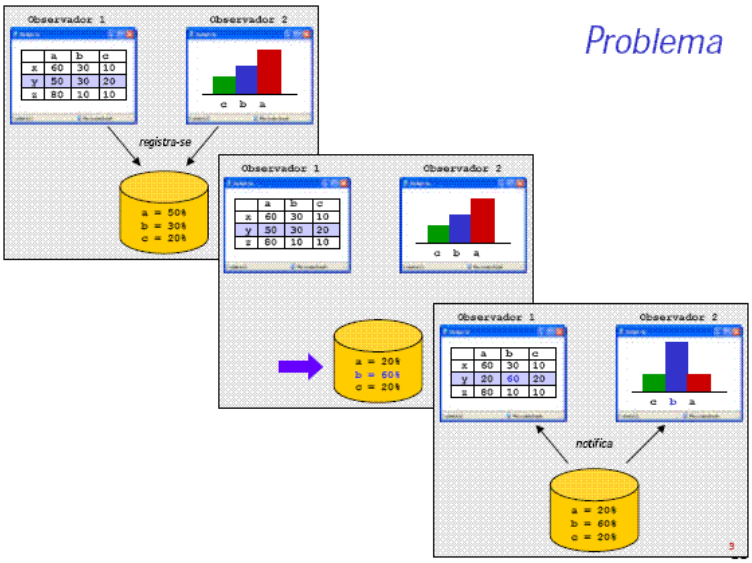

Figura 11 Representação do padrão *Observer*
Fonte: ROCHA, 2003.

- **Aplicabilidade**

Conforme Gamma *et al.* (2000), deve-se utilizar o padrão *Observer* quando mudanças em um objeto acarretam mudanças em outros objetos ou quando for desejável diminuir o acoplamento entre objetos. A aplicação do *Observer* nessas situações permite a reutilização e variação de objetos de forma independente.

- **Exemplo de código**

A Figura 12 contém um exemplo de código, em Java, de implementação do padrão *Observer*. Neste exemplo, a classe *Observable* conhece os seus observadores e fornece métodos para acrescentar e remover os mesmos. A classe *ObservableData* armazena os dados e notifica aos seus observadores quando esses dados mudam. A classe *ConcreteObserver* mantém referência à classe *ObservableData,* fazendo com que seus dados sejam consistentes com os da *ObservableData* e implementa *Observer* que, por sua vez, define uma interface de atualização para os objetos que devem ser notificados caso ocorram mudanças em *Observable*.

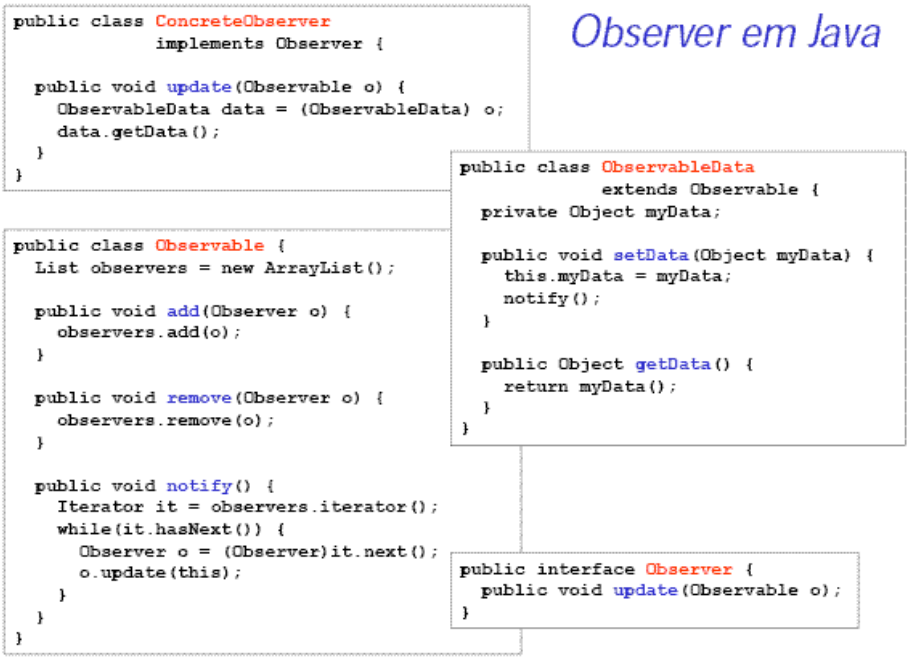

Figura 12 Implementação do padrão *Observer*
Fonte: ROCHA, 2003.

- **Conseqüências**

Conforme Gamma *et al.* (2000), *o padrão Observer* permite tratar objetos observados e observadores de maneira independente. Nessa linha de raciocínio, Rocha (2003) explica que a reutilização e alteração da implementação dos objetos observados e observadores podem ser realizadas sem afetar toda a aplicação.

3.2 *State*

- **Intenção**

Segundo Gamma *et al.* (2000), quando o estado interno de um objeto muda, o padrão *State* permite que ele altere o seu comportamento, fazendo com que ele aparente mudar de classe.

- **Motivação**

Gamma *et al.* (2000) descrevem um cenário motivacional para utilização do padrão *State,* que pode ser observado na Figura 13. Nela, a classe *TCPConnection* representa uma conexão em uma rede de comunicações, sendo que esta pode assumir os seguintes estados: estabelecida, escutando e fechada. As subclasses de *TCPState* (responsável por representar os estados da conexão): *TCPEstablished*, *TCPListen* e *TCPClosed* implementam, respectivamente, esses estados. A classe *TCPConnection* mantém uma instância da subclasse de *TCPState* e, toda vez que a conexão muda de estado, essa instância é substituída por uma instância da subclasse correspondente a esse estado.

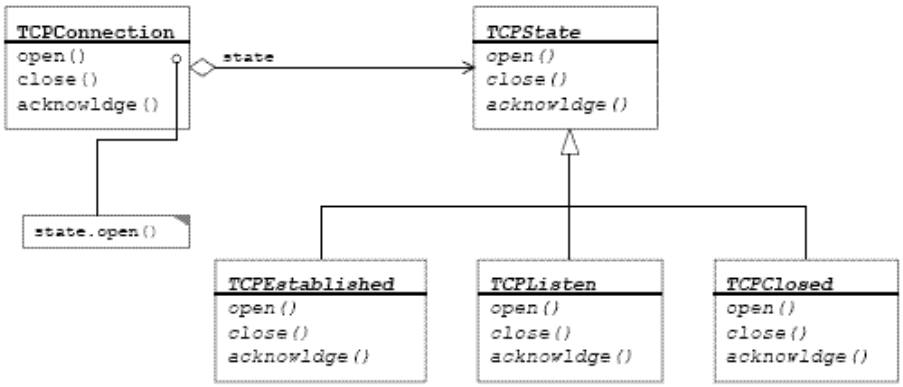

Figura 13 Diagrama de classe utilizando o padrão *State*
Fonte: GAMMA *et al.*, 2000, p.284.

- **Aplicabilidade**

Segundo Gamma *et al.* (2000), deve-se usar o padrão *State* quando o estado de um objeto afeta o comportamento do mesmo em tempo de execução ou quando a aplicação possui várias estruturas condicionais dependentes do estado em que o objeto se encontra.

- **Exemplo de código**

A Figura 14 contém um exemplo de código Java que utiliza o padrão *State*. Nesse exemplo, a classe *Gato* contém um atributo denominado *estado* do tipo *Estado* (interface que encapsula o comportamento associado a um determinado estado de *Gato*). As classes *EstadoVivo*, *EstadoMorto* e *EstadoQuantico* implementam os comportamentos associados aos estados que *Gato* pode assumir.

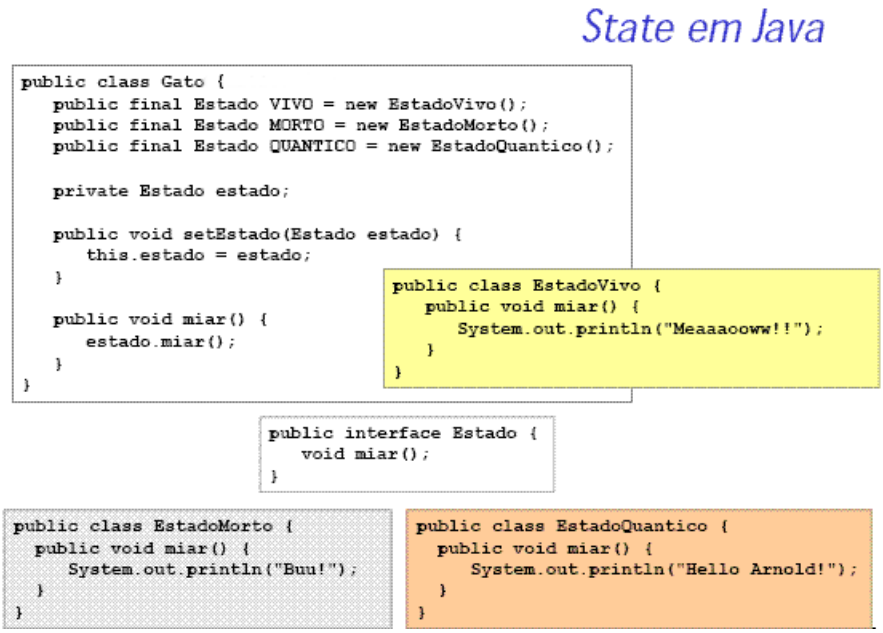

Figura 14 Implementação do padrão *State*
Fonte: ROCHA, 2003.

- **Conseqüências**

Segundo Gamma *et al.* (2000), o padrão *State* coloca os comportamentos relacionados a um determinado estado em um objeto, ou seja, a codificação desse estado é feita em uma classe específica, o que simplifica a adição de novos estados. Apesar de aumentar o número de classes, a utilização do *State* evita o uso excessivo de comandos condicionais. Além disso, as transições de estados são representadas de maneira explícita, já que define objetos separados para estados diferentes.

3.3 *Iterator*

- **Intenção**

Segundo Gamma *et al.* (2000), o padrão *Iterator* possibilita uma maneira de acessar, de forma seqüencial, os elementos de um objeto agregado sem expor detalhes de sua representação interna.

- **Motivação**

Gamma *et al.* (2000) demonstram que, em objetos agregados, como listas, pode ser necessário percorrer os seus elementos sem expor a sua representação interna ou, até mesmo, percorrer a lista de maneiras diferentes. O padrão *Iterator* fornece soluções para questões como essa. Ele delega a função de percurso da lista a um objeto que tem a responsabilidade de manter a posição do elemento que está sendo percorrido na lista, controlando assim, quais elementos já foram percorridos.

- **Exemplo de código**

Segundo Rocha (2003), em Java, o padrão *Iterator* é implementado através do método *iterator()* da interface *Collection*. A Figura 15 contém um fragmento de código Java contendo um objeto de *HashMap* (um tipo de *Collection*) cujos elementos são percorridos através da utilização do método *interator().*

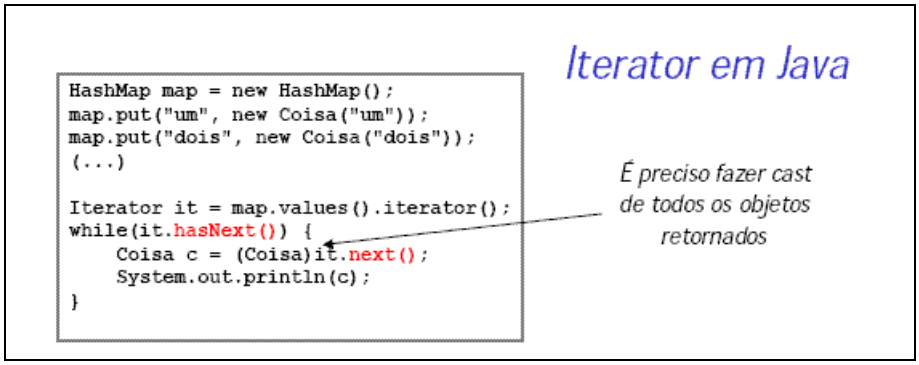

Figura 15 Implementação do padrão *Iterator*
Fonte: ROCHA, 2003.

- **Aplicabilidade**

De acordo com Gamma *et al.* (2000), o padrão *Iterator* deve ser utilizado quando deseja-se percorrer diferentes estruturas agregadas através de uma interface uniforme ou quando não se pretende expor a representação interna de um objeto agregado ao acessar o seu conteúdo. É também utilizado quando for preciso tratar objetos agregados que possuem múltiplos trajetos.

- **Conseqüências**

Conforme Gamma *et al.* (2000), o *Iterator* facilita a mudança de percurso em objetos agregados que possuem múltiplos percursos. Esse padrão permite também atravessar mais de um percurso de um objeto agregado simultaneamente. Além disso, o *Iterator* simplifica a interface do objeto agregado, pois toda interface de percurso do seu conteúdo é definida pelo padrão.

3.4 *Template Method*

- **Intenção**

De acordo com Gamma *et al.* (2000), o propósito do padrão *Template Method* consiste em determinar o esqueleto de um algoritmo, de forma que alguns passos desse algoritmo possam ser redefinidos na subclasse sem ter a sua estrutura alterada.

De forma semelhante, Rocha (2003) explica que o *Template Method* define um algoritmo que utiliza operações abstratas que são sobrepostas nas subclasses, oferecendo um comportamento concreto.

- **Motivação**

Shalloway e Trott (2004) expõem, como exemplo motivacional para utilização desse padrão, casos em que existem procedimentos fixos em um determinado algoritmo, no entanto, alguns passos desse algoritmo podem ter diferentes implementações.

- **Exemplo de código**

A Figura 16 contém um exemplo de código Java que utiliza o padrão *Template Method.* Nesse contexto, a classe abstrata *Template* contém uma operação abstrata cuja implementação pode variar. Assim, as subclasses *XMLData* e *HTMLData* são responsáveis por redefinir essa operação de forma a fornecer um comportamento concreto.

```
public abstract class Template {
   protected abstract String link(String texto, String url);
   protected String transform(String texto) {
      return texto;
   }
   public final String templateMethod() {
      String msg = "Endereço: " + link("Empresa","http://www.empresa.com");
      return transform(msg);
   }
}
```

```
public class HTMLData extends Template {
   protected String link(String texto, String url) {
      return "<a href='" + url + "'>" + texto + "</a>";
   }
}
```

```
public class XMLData extends Template {
   protected String link(String texto, String url) {
      return "<endereco xlink:href='" + url + "'>" + texto + "</endereco>";
   }
}
```

Figura 16 Implementação do padrão *Template Method*
Fonte: ROCHA, 2003.

- **Aplicabilidade**

Segundo Rocha (2003), o padrão *Template Method* deve ser utilizado quando for desejável delegar a implementação dos comportamentos variantes de uma classe às suas subclasses.

Gamma *et al.* (2000) acrescentam ainda que o *Template Method* pode ser utilizado quando subclasses diferentes possuem comportamentos comuns. Nesse caso eles devem ser concentrados em uma única classe evitando a duplicação de código.

- **Conseqüências**

Conforme Gamma *et al.* (2000), o padrão *Template Method* fornece uma estrutura fundamental para reutilização de código, uma vez que permite a fatoração de comportamentos comuns entre classes. Ele também permite que uma superclasse invoque as operações da subclasse e não o contrário. Além disso, esse padrão possibilita que a superclasse controle a maneira como as subclasses a estendem, determinando quais operações podem realmente ser redefinidas.

4 PADRÕES DE CRIAÇÃO

Segundo Gamma *et al.* (2000), os padrões criacionais estão relacionados ao processo de criação de objetos. Padrões criacionais, voltados a classes, repassam alguns passos da criação dos objetos para suas subclasses, enquanto os padrões criacionais, voltados a objetos, delegam esse processo de criação para um outro objeto.

4.1 *Singleton*

- **Intenção**

De acordo com Gamma *et al.* (2000), o padrão *Singleton* assegura que uma classe seja instanciada somente uma vez, além de prover um único ponto de acesso a ela em toda a aplicação.

Nesse sentido, Shalloway e Trott (2004) enfatizam que esse padrão possui um método responsável em limitar o número de instâncias da classe. Além disso, o *Singleton* define o construtor da classe como privado, ou seja, sua invocação é vedada a qualquer outra classe, para garantir que somente a própria classe instancie um objeto de seu tipo.

- **Motivação**

Shalloway e Trott (2004) descrevem um cenário de motivação em que o problema seja assegurar a existência de um único objeto de um tipo específico que é acessado por vários objetos diferentes. Diante desse problema, o padrão Singleton garante que apenas um objeto exista.

- **Exemplo de código**

O Exemplo 4 contém um fragmento de código Java no qual o método *obterInstancia* delimita uma única instância para a classe *ImpostoEUA.* É

demonstrado também que a utilização desse padrão impossibilita outra forma de instanciar essa classe, pois ele torna o seu construtor privativo, impedindo que qualquer objeto possa invocá-lo.

```
public class ImpostoEUA {
	private static ImpostoEUA instancia;

	private ImpostoEUA() {
	}

	public synchronized static ImpostoEUA obterInstancia() {
		if (instancia == null) {
			instancia = new ImpostoEUA();
		}
		return instancia;
	}
}
```

Exemplo 4 Fragmento de código Java com padrão *Singleton*
Fonte: Adaptado de SHALLOWAY e TROTT, 2004, p.259.

- **Aplicabilidade**

Conforme Gamma *et al.* (2000), deve-se utilizar o *Singleton* quando for necessário instanciar uma classe somente uma vez e que essa instancia seja acessada através de um ponto global.

- **Conseqüências**

De acordo com Shalloway e Trott (2004), com a utilização do padrão *Singleton*, não se precisa preocupar com possibilidade ou não da existência de uma instância da classe na qual padrão está sendo aplicado, pois isso é controlado dentro da própria classe.

4.2 *Abstract Factory*

- **Intenção**

A intenção do padrão *Abstract Factory* é “fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.” (GAMMA *et al*., 2000, p.95).

- **Motivação**

Gamma *et al.* (2000) descrevem que *toolkits* que contêm componentes para múltiplas interfaces gráficas de diferentes sistemas operacionais representam um problema resolvido com a utilização do padrão *Abstract Factory,* pois, para que aplicações que utilizam *toolkits* sejam portáveis entre diferentes formas de apresentação gráfica, é necessário que seus componentes não sejam codificados exclusivamente para uma determinada interface gráfica, pois dificultaria a sua mudança para uma outra.

A Figura 17 demonstra a utilização do padrão *Abstracty Factory,* neste caso, a *interface GUIFactory* declara o método para criação de um tipo *Button* e as suas subclasses *WinFactory* e *OSXFactory* realizam as operações para criar o componente *Button* apropriado aos sistemas operacionais Windows e OSX, respectivamente. A aplicação, representada pela classe *Application*, possui conhecimento somente das interfaces *GUIFactory* e *Button*, desconhecendo as classes concretas, ou seja, as classes responsáveis pela criação dos componentes. A implementação, em Java, desse sistema pode ser observada no Exemplo 5.

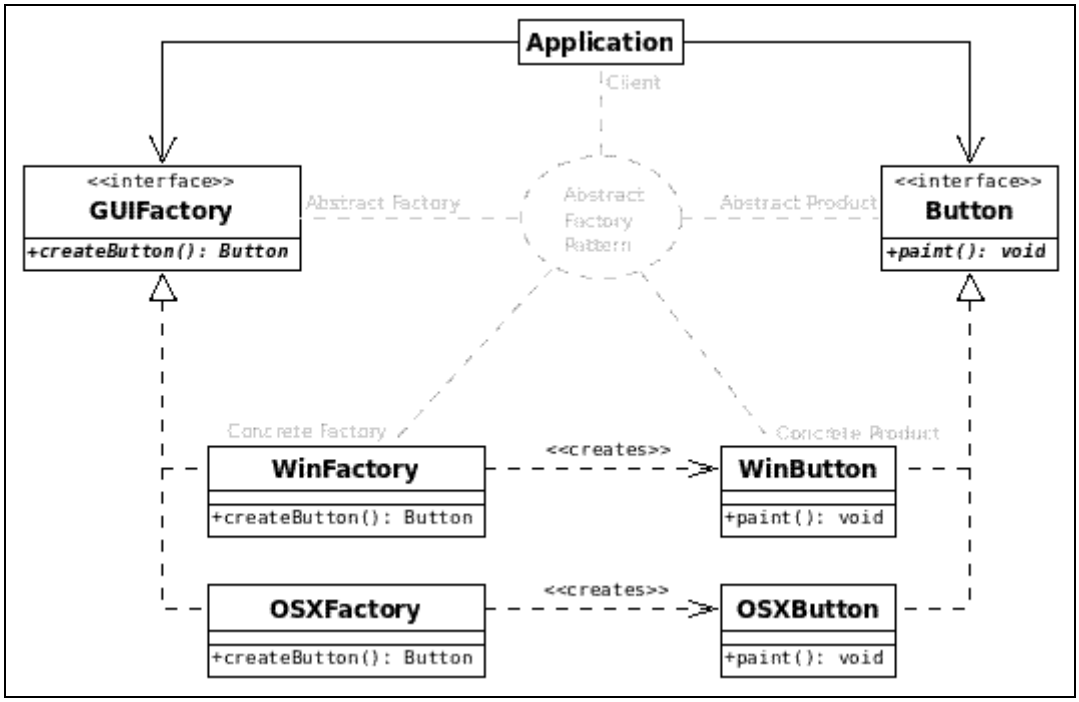

Figura 17 Padrão *Abstract Factory*
Fonte: TERRA, 2008.

```
abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
    public abstract Button createButton();
}


abstract class Button {
   public abstract void paint();
}


class WinFactory extends GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}


class WinButton extends Button {
    public void paint() {
        System.out.println("I'm a WinButton!");
    }
}


class OSXFactory extends GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}


class OSXButton extends Button {
    public void paint() {
        System.out.println("I'm an OSXButton!");
    }
}


public class Application {
    public static void main(String[] args) {
        GUIFactory factory = GUIFactory.getFactory();
        Button button = factory.createButton();
        button.paint();
    }
}
```

Exemplo 5 Padrão *Abstract Factory*
Fonte: TERRA, 2008.

- **Aplicabilidade**

Segundo com Gamma *et al.* (2000), o padrão *Abstract Factory* deve ser utilizado quando for necessário que um sistema funcione independentemente da criação, composição ou representação dos seus produtos ou quando for necessário que uma família de objetos seja utilizada em conjunto.

- **Conseqüências**

Conforme Gamma *et al.* (2000), o padrão *Abstract Factory* permite isolar as classes concretas, pois os clientes manipulam as instâncias dessas classes através de suas interfaces. Ele facilita a troca de famílias de produto, pois as classes concretas aparecem em apenas um lugar da aplicação (quando é instanciada), o que elimina a complexidade ao se realizar mudanças nessas classes, além de assegurar que objetos pertencentes a uma mesma família trabalhem em conjunto.

CONSIDERAÇÕES FINAIS

Este trabalho foi desenvolvido com o propósito de realizar um estudo aprofundado que abordasse os padrões de projeto mais utilizados classificados pela GoF. Demonstrando as necessidades de utilização dos padrões de projeto em desenvolvimento de sistemas de software, apresentando as principais características e vantagens que fazem com que os padrões de projeto sejam uma solução para diversos problemas.

A utilização de padrões de projeto evita a tendência de muitos desenvolvedores em utilizar técnicas não-orientadas a objetos, pois esses padrões utilizam as melhores práticas de orientação a objetos para alcançar os resultados propostos.

Também foi possível notar que a utilização de padrões de projeto possibilita o desenvolvimento de sistemas de software de melhor qualidade, pois, conforme demonstrado em alguns padrões, eles utilizam de forma proveitosa os conceitos de herança, polimorfismo, composição, além de minimizar o acoplamento entre os objetos da aplicação.

É importante ressaltar que este estudo permite observar que os padrões de projetos podem ser utilizados tanto por desenvolvedores de software iniciantes quanto desenvolvedores mais experientes. Para os iniciantes ajudam, dentre outras coisas, a utilizar de forma eficiente os recursos da programação orientada a objetos, bem como utilizar soluções já estabelecidas para os problemas encontrados. Para os mais experientes, que já conhecem muitos problemas enfrentados no desenvolvimento de software, é mais fácil relacioná-los às soluções apresentadas pelos padrões. Diante disso, este trabalho serve como fonte de consulta tanto para iniciantes quanto para os mais experientes.

De acordo com a classificação adotada, os padrões de projeto são divididos em criacionais, comportamentais e estruturais. Dentre os padrões estruturais, foram abordados os padrões *Adapter*, *Decorator*, *Proxy*, *Façade*, *e Composite*. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto. O *Decorator* possibilita a atribuição dinâmica de novas funcionalidades a um objeto. O *Proxy* disponibiliza um ponto de localização através do qual um objeto consegue controlar o acesso a outro. O *Façade* cria uma fachada do sistema de forma a

divulgar todas as operações realizadas por ele. O *Composite* permite que objetos individuais e composições desses objetos sejam tratados da mesma maneira.

Dentre os padrões comportamentais, foram abordados os padrões *Observer*, *State*, *Iterator* e T*emplate Method.* O padrão *Observer* permite que mudanças ocorridas em um objeto sejam notificadas e atualizadas em seus dependentes. O *State* permite a alteração de comportamento de um objeto caso o seu estado interno seja alterado. O *Iterator* fornece uma maneira de percorrer objetos agregados sem expor detalhes de sua estrutura. O *Template Method* permite que alguns passos de um algoritmo sejam definidos pelas subclasses.

Dentre os padrões criacionais, foram abordados os padrões *Singleton* e *Abstract Factory.* O *Singleton* limita que uma classe contenha uma única instância. O *Abstract Factory* trata da criação de famílias de objetos sem especificar suas classes concretas.

O presente estudo, que investigou a utilização de padrões de projeto em desenvolvimento de software, vem facilitar o entendimento sobre esse assunto. Conhecer esses padrões e aplicá-los de forma correta facilita ou, até mesmo, minimiza o trabalho dos desenvolvedores de software. Como trabalho futuro pretende-se desenvolver uma grande aplicação que implemente esses padrões de projeto e outra que não implemente e aplicar diversas métricas de engenharia de software para mensurar e comparar a coesão, o acoplamento, o impacto de modificações, entre outros.

REFERÊNCIAS

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core J2EE patterns**: as melhores práticas e estratégias de design. Rio de Janeiro: Elsevier, 2004.

BOIS, André Rauber Du. RMI: **Remote Method Invocation**. Disponível em: <http://atlas.ucpel.tche.br/~dubois/progavancada/05-RMI.pdf > Acesso em: 28 nov. 2008.

FILHO, Wilson de Pádua Paula. **Engenharia de Software:** fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2003.

GAMMA, Erich et al. **Padrões de Projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

LEAL, José Paulo. **Arquitectura de Software**. Disponível em: <http://www.dcc.fc.up.pt/~zp/aulas/0708/asw/indice?aulas/patterns/catalogo/estruturais/composite/exemplo> Acesso em: 20 nov. 2008.

ROCHA, Helder da. **J930: GoF Design Patterns em Java**. 2003. Disponível em: <http://www.argonavis.com.br/cursos/java/j930/index.html> Acesso em: 10 set. 2008.

SANTOS, Wagner Roberto dos. **EJB 3.1: Conheça as Novidades do Futuro do Java Corporativo**. Revista Mundo Java, Rio de Janeiro: Mundo, 2008. nº31, p.26-35.

SHALLOWAY, Alan; TROTT, James R. **Explicando padrões de projeto**: uma nova perspectiva em projeto orientado a objeto. Porto Alegre: Bookman, 2004.

TERRA, Ricardo (rterrabh@gmail.com). **Padrão de Projeto: Abstract Factory**. [mensagem pessoal]. Mensagem recebida por alinesousap@yahoo.com.br em 22 de outubro de 2008.

TERRA, Ricardo. **JAVA SE: Manipulação de Streams**. 2008. Disponível em: <http://www.ricardoterra.com.br/faminas/files/teiiii/JSE%20-%20Manipulacao%20de%20Streams.pdf> Acesso em: 20 nov. 2008.