FACULDADE DE CIÊNCIAS APLICADAS "SAGRADO CORAÇÃO"
DIRETORIA DE ENSINO SUPERIOR
COORDENAÇÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO

GUSTAVO ANDRÉ DE FREITAS
RILIANE ALPOIM PARIS
RODRIGO SILVA DE SOUZA

PADRÕES DE PROJETO
FAÇADE, FLYWEIGHT E VISITOR

LINHARES
2007

GUSTAVO ANDRÉ DE FREITAS
RILIANE ALPOIM PARIS
RODRIGO SILVA DE SOUZA

PADRÕES DE PROJETO
FAÇADE, FLYWEIGHT E VISITOR

Trabalho Acadêmico do Curso de Bacharelado em Sistemas de Informação da Faculdade de Ciências Aplicadas "Sagrado Coração" - UNILINHARES, apresentado como requisito para avaliação disciplinar.
Orientador: Prof. Marcos Guimarães.

LINHARES
2007

SUMÁRIO

1 INTRODUÇÃO

A visão de padrões de projetos em software surgiu baseada nos estudos de Christopher Alexander e tomou forma com o livro Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos, lançado pela “Gangue dos Quatro” no início da década de 90.

No presente trabalho explanamos sobre os 23 padrões de projeto de software existentes e classificados quanto à finalidade em criação, estrutural e comportamental. Nosso foco esteve no detalhamento de três padrões de projeto específicos: Façade, Flyweight e Visitor.

2 PADRÕES DE PROJETO

A idéia de padrões de projeto surgiu por intermédio de um arquiteto chamado Christopher Alexander, que começou a se perguntar se era possível criar padrões para sabermos se um projeto arquitetônico era bom ou não, independente do gosto das pessoas. Ele chegou à conclusão de que podemos descrever a beleza por meio de uma base prática e mensurável. A antropologia chegou à mesma conclusão, de que os indivíduos de uma sociedade concordarão em sua grande maioria a respeito do que é considerado um bom projeto, do que é bonito.

Christopher Alexander define um padrão como “uma solução para um problema em um determinado contexto”. (SHALLOWAY, 2004).

Os padrões de projeto podem ser vistos como uma solução que já foi testada repetidas vezes para um mesmo problema. No início da década de 90 alguns desenvolvedores tomaram conhecimento do trabalho de Alexander, e chegaram à conclusão de que esse pensamento poderia ser utilizado perfeitamente no desenvolvimento de softwares.

> “Embora muitas pessoas estivessem trabalhando em padrões de projeto no início da década de 1990, o livro que teve a maior influência sobre essa jovem comunidade foi *Design Patterns: Elements of Reusable Object Oriented software*, escrito por Gamma, Helm, Johnson e Vlissides”. (SHALLOWAY, 2004, p.97).

A partir desse livro esses autores passaram a ser conhecidos como a “Gangue dos Quatro”. Nesse livro foram aplicadas as idéias de padrões de projeto a projeto de software, descrito uma estrutura para catalogar e descrever padrões de projeto[1] e traçadas estratégias orientadas a objetos baseados nesses padrões.

Os autores não criaram padrões de projeto, eles catalogaram os padrões que já existiam na comunidade de desenvolvedores de software, mas não eram ainda descritos como padrões.

[1] Foram catalogados 23 padrões de projeto.

Item	Descrição
Nome	Todos os padrões têm um nome único que os identifica
Intenção	O propósito do padrão
Problema	O problema que o padrão está tentando resolver
Participantes e colaboradores	As entidades envolvidas no padrão
Conseqüências	As conseqüências de utilizar o padrão. Investiga as forças que nele interagem.
Implementação	Como o padrão pode ser implementado
Referência Gangue dos Quatro	O lugar para onde olhar no texto da Gangue dos Quatro a fim de obter mais informações

Tabela 1 – Características principais dos padrões
Fonte: SHALLOWAY, 2003, p. 98.

Os padrões são classificados quanto à sua finalidade, podendo ser de criação, estrutural e comportamental, conforme a tabela 2 demonstra.

- Criação → Auxilia no processo de instanciação do objeto.
- Estrutural → Trata as composições de classes e objetos.
- Comportamental → Caracteriza a maneira pela qual as classes e os objetos interagem e distribuem responsabilidades.

FINALIDADE	ESCOPO	NOME
Criação	Classe	Factory Method
	Objeto	Abstract Factory Builder Phototype Singleton
Estrutural	Classe/Objeto	Adapter
	Objeto	Bridge Composite Decorator Façade Flyweight Proxy
Comportamental	Classe	Interpreter Template Method
	Objeto	Chain of Responsibility Command Interator Mediator Memento Observer State Strategy Visitor

Tabela 2 – classificação dos padrões
Fonte: ADRIANO, 2007.

3 CRIAÇÃO

3.1 PADRÃO FACTORY METHOD

“De acordo com a Gangue dos Quatro, o padrão Factory Method define uma interface para a criação de um objeto, mas deixa que as subclasses decidam qual classe instanciar”. (SHALLOWAY, 2004).

O padrão Factory Method define uma classe comum para a criação de objetos. Isso possibilita que classes sejam criadas sem que o desenvolvedor conheça a classe concreta do objeto a ser criado, sendo necessário apenas conhecer a classe do criador.

Rocha (2003) enumera vantagens e desvantagens desse padrão.

- Vantagens:
    - Criação de objetos desaclopada do conhecimento do tipo concreto do objeto;
    - Conecta hierarquias de classe paralelas e
    - Facilita a extensabilidade.
- Desvantagens:
    - Necessário saber a classe concreta do criador de instancias.

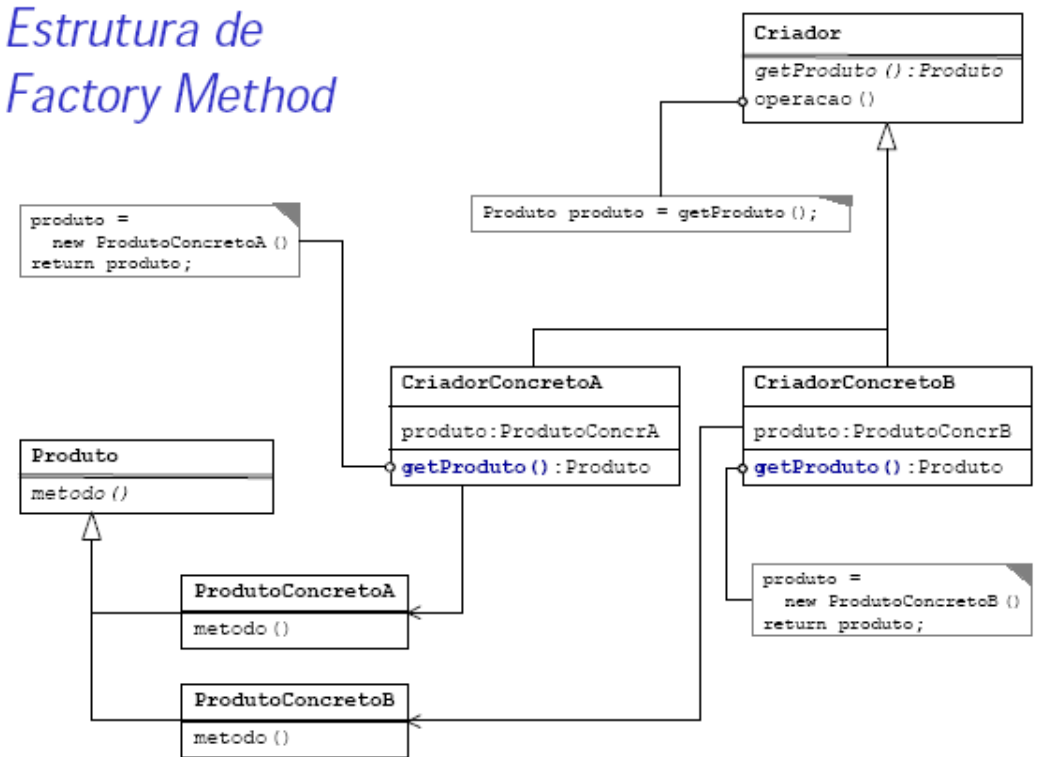

Figura 1 – Factory Method
Fonte: Rocha, 2003.

3.2 ABSTRACT FACTORY

“De acordo com a Gangue dos Quatro, o padrão Abstract Factory fornece uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas”. (SHALLOWAY, 2004).

O padrão Abstract Factory é usado quando se deseja retornar uma das muitas classes de objetos relacionadas. Esse padrão é uma fábrica de objetos que pode retornar uma das várias fábricas disponíveis.

Um exemplo clássico citado por Destro (2006) é o caso onde o sistema necessita de suporte a múltiplas interfaces gráficas de diferentes sistemas operacionais. Você diz ao Abstract Factory que quer seu programa se pareça com o Windows, e ele retorna a fábrica GUI que produz os objetos relacionados ao Windows. Quando for requisitado um objeto específico como um botão, a fábrica de GUI retorna as instancias desses componentes.

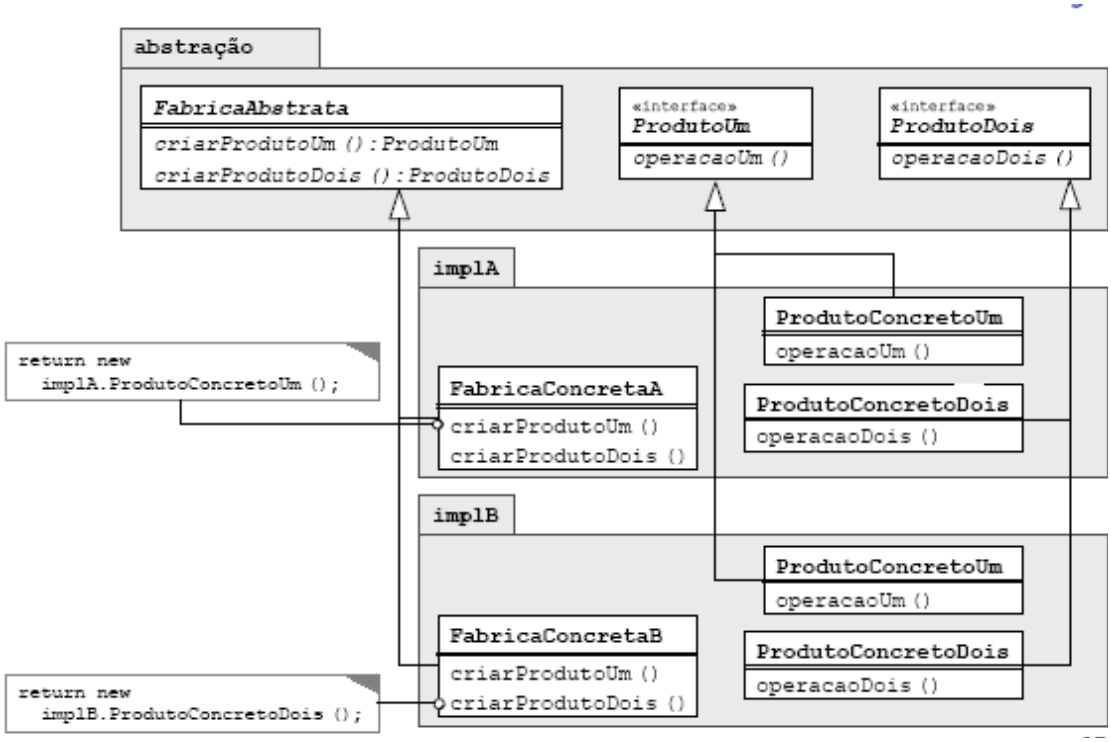

Figura 2 – padrão Abstract Factory
Fonte: Rocha, 2003.

3.3 BUILDER

“O padrão Builder separa a construção de um objeto complexo da sua representação, para que diferentes representações possam ser criadas dependendo das necessidades do programa”. (DESTRO, 2006).

Com o padrão Builder a classe só precisa se preocupar com apenas uma parte da construção de um objeto, muito útil em algoritmos de construção complexos. Veja a figura 9.

Algumas vantagens do padrão Builder são:

- Permite variar a representação interna do objeto que ele cria;
- Oculta os detalhes de como o produto é montado;
- O Builder específico é independente do resto do programa;
- Maior controle sobre o produto final.

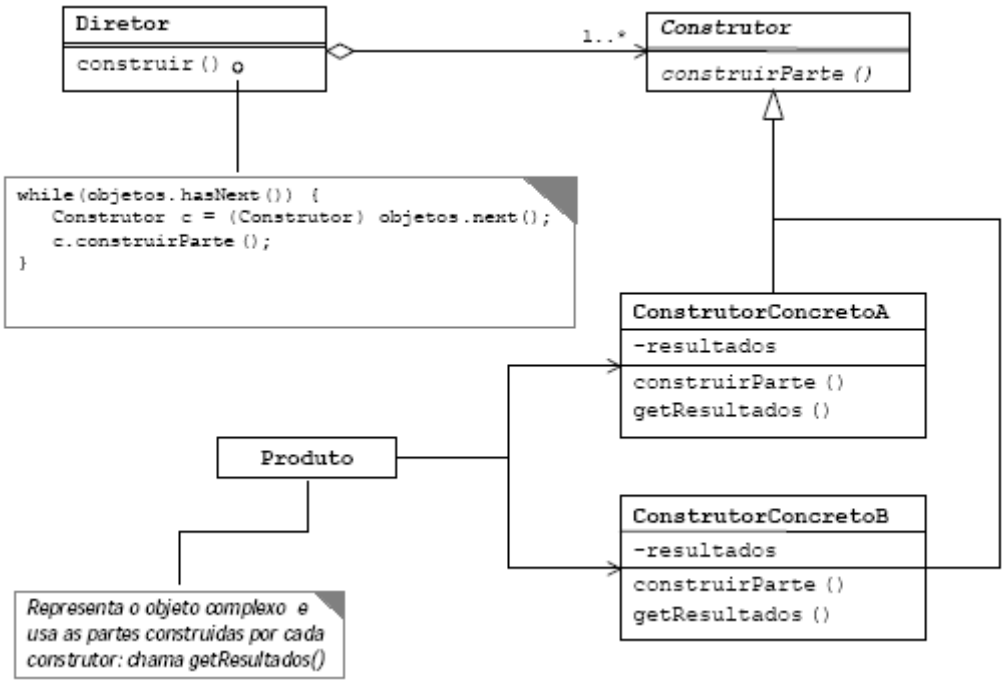

Figura 3 – padrão Builder
Fonte: Rocha, 2003.

3.4 PROTOTYPE

“O padrão Prototype começa com uma classe iniciada e instanciada que copia ou gera um clone dela para se fazer novas instâncias, ao invés de se criar novas instâncias”. (DESTRO, 2006).

Quando é necessário a criação de um novo objeto, mas há interesse em se aproveitar o estado de um objeto já existente nesse novo objeto, torna-se útil a utilização do padrão Prototype, já que ele permite criar um objeto novo aproveitando o estado de um objeto existente, como podemos ver na figura 4.

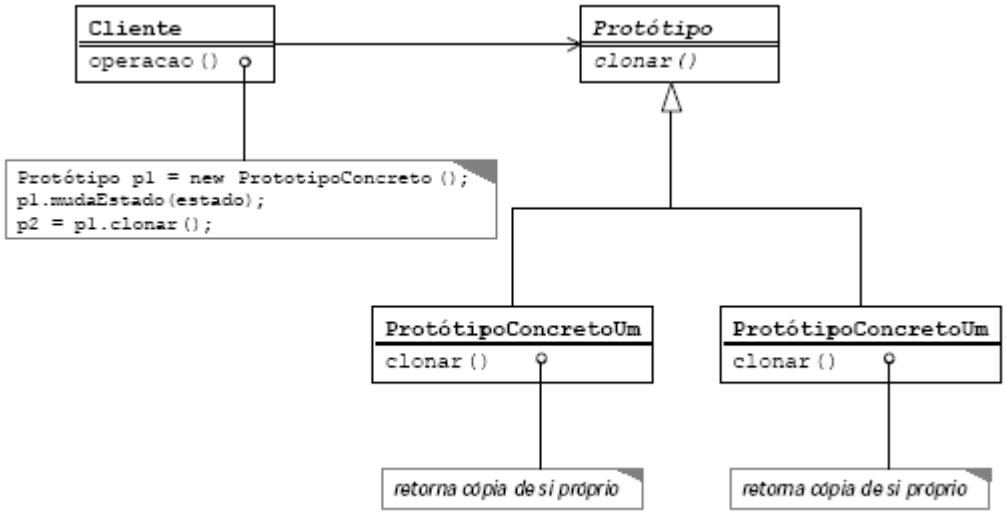

Figura 4 – padrão Prototype
Fonte: Rocha, 2003.

3.5 SINGLETON

“De acordo com a Gangue dos Quatro, o padrão Singleton assegura que uma classe possua somente uma instância e fornece um ponto global de acesso a essa instância”. (SHALLOWAY, 2004).

O padrão Singleton é muito simples e comum, e sua função é assegurar que somente um objeto de uma classe específica será instanciado. O Singleton é usado em aplicações com uma única linha de execução, também conhecida como *Thread*. Ele verifica se o objeto já foi instanciado, se sim, apenas retorna uma referência a ele, senão, instancia o objeto e retorna uma referência para essa instância, conforme mostrado na figura 5.

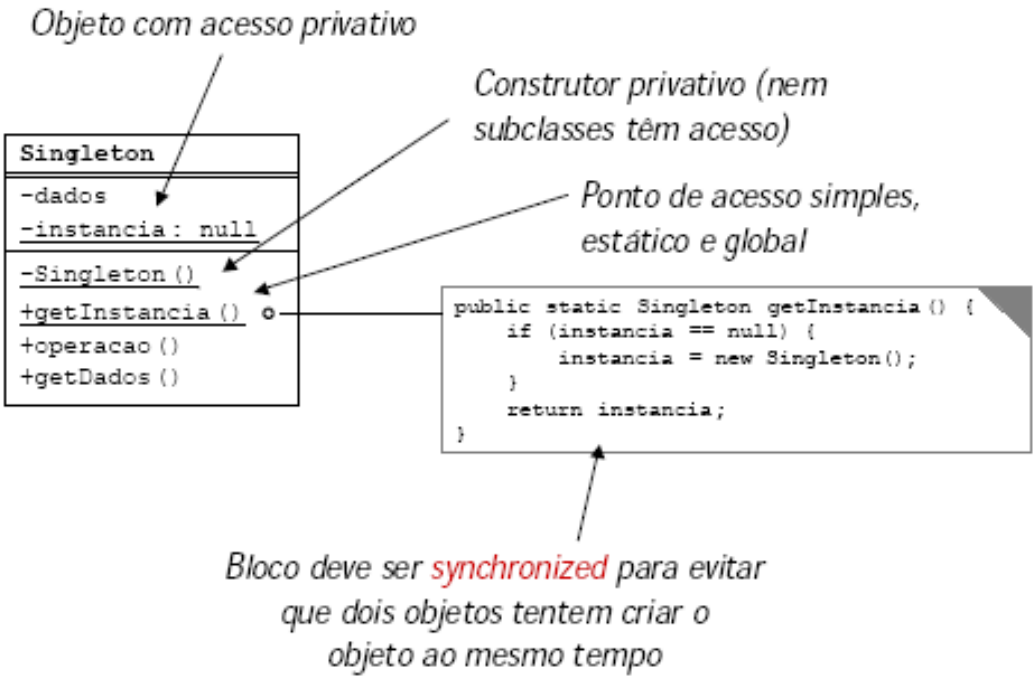

Figura 5 – padrão Singleton
Fonte: Rocha, 2003.

4 ESTRUTURAL

4.1 ADAPTER

Esse padrão funciona como um adaptador entre classes e/ou objetos, permitindo que uma classe e/ou objeto que realize a função desejada, porém não tenha a interface requerida, possa ser utilizada(o) no projeto.

Esse tipo de problema especifica o caso de dois objetos, por exemplo, que necessitam de se comunicar, porém suas interfaces não se comunicam. Um bom exemplo é o caso de ter uma tomada e um plugue diferentes, isto é, a tomada com duas entradas e um plugue de três pinos. O encaixe é impossível. Diante disso utiliza-se de um adaptador que tenha dois pinos, para encaixar na tomada, e que tenha uma entrada para três pinos, para encaixar o plugue.

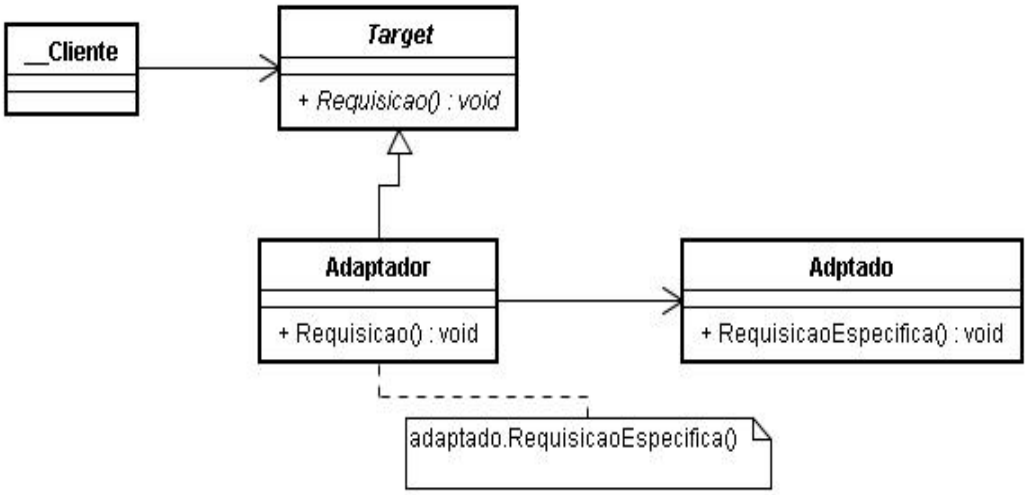

Figura 6: Padrão Adapter
Fonte: CALSAVARA, 2007.

Em um software é possível utilizar esse padrão da mesma forma. Segundo Shalloway (2004, p. 120), “Contando com uma classe que faça o que desejo, ao menos conceitualmente, saberei então que poderei sempre usar o padrão Adapter para fornecer a ela a interface correta.”

A utilização do padrão Adapter traz como conseqüência a facilidade no desenvolvimento do projeto de tal maneira que não é mais necessária a intensa preocupação com a interfaces de classes reutilizadas. Isso devido ao fato de ser possível a utilização de uma classe adaptadora entre as classes e/ou objetos que se deseja trabalhar.

4.2 PADRÃO BRIDGE

“De acordo com a Gangue dos Quatro, a intenção do padrão Bridge é desacoplar uma abstração de sua implementação, de modo que as duas possam variar independentemente”. (SHALLOWAY, 2003, p.137).

A intenção do padrão Bridge é desacoplar um conjunto de implementações do conjunto de objetos que as utilizam. Nesse ponto surge um problema, pois as derivações de uma classe abstrata devem usar múltiplas implementações sem causar um número crescente e incontrolável de classes. Para que isso possa ser controlado é necessário definir uma interface para todas as implementações a serem usadas, de forma que as derivações da classe abstrata as utilizem. O desacoplamento entre as implementações e os objetos que as usam aumenta a capacidade de extensão.

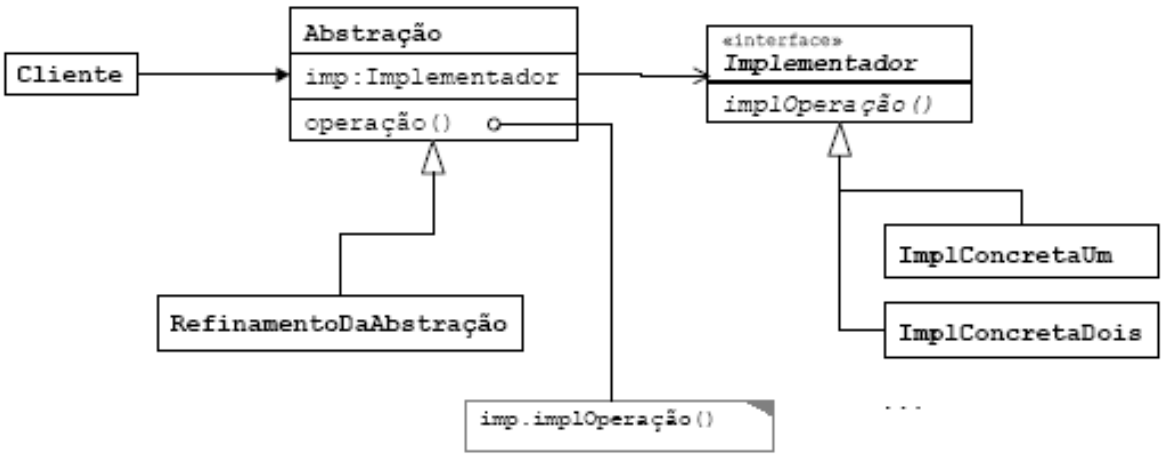

Figura 7 – padrão Bridge
Fonte: Rocha, 2003.

Esse padrão deve ser utilizado quando for necessário evitar ligação permanente entre a interface e a implementação, para que alterações não afetem os clientes. Um bom exemplo de Bridge são os drives JDBC.

4.3 COMPOSITE

“O padrão Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme”. (ROCHA, 2003).

Se um Cliente precisa tratar de maneira uniforme objetos individuais, ou composição desses objetos usa-se o padrão Composite para que ele dê ao Cliente essa possibilidade, conforme a figura 8. Para isso o Composite compõe os objetos em estruturas de árvore representando hierarquias todo-parte.

Rocha (2003) alerta que se o relacionamento possuir ciclos, é preciso tomar precauções adicionais para evitar loops, já que Composite depende de implementações recursivas.

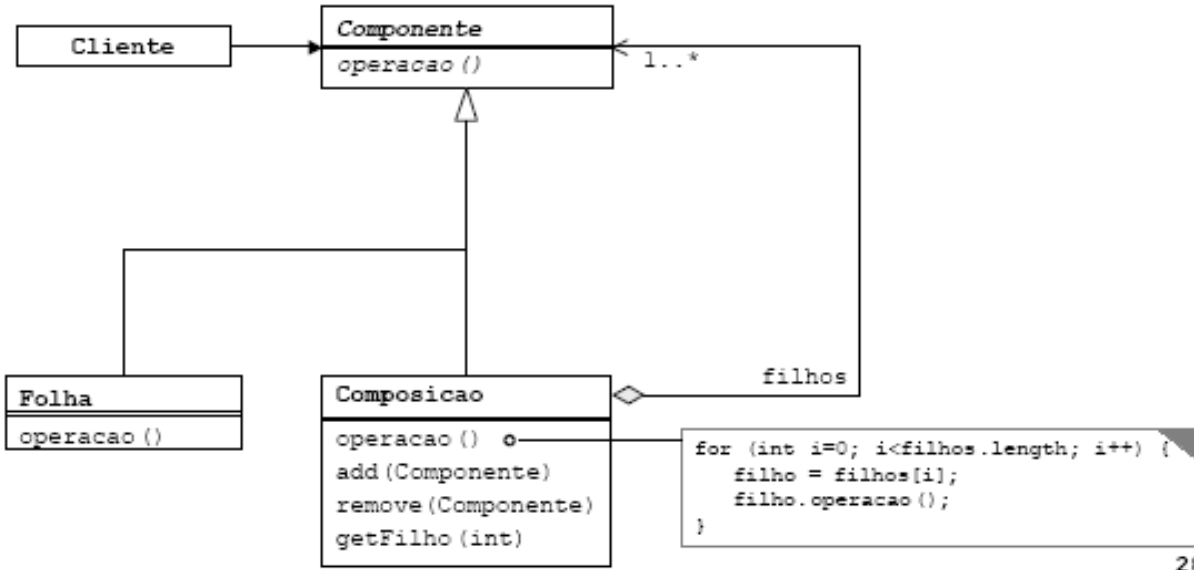

Figura 8 – padrão Composite
Fonte: Rocha, 2003.

4.4 DECORATOR

Esse padrão é utilizado quando há a necessidade de adicionar funcionalidades a objetos, e não a toda a classe, de forma dinâmica, onde este objeto não sabe que está recebendo essa funcionalidade.

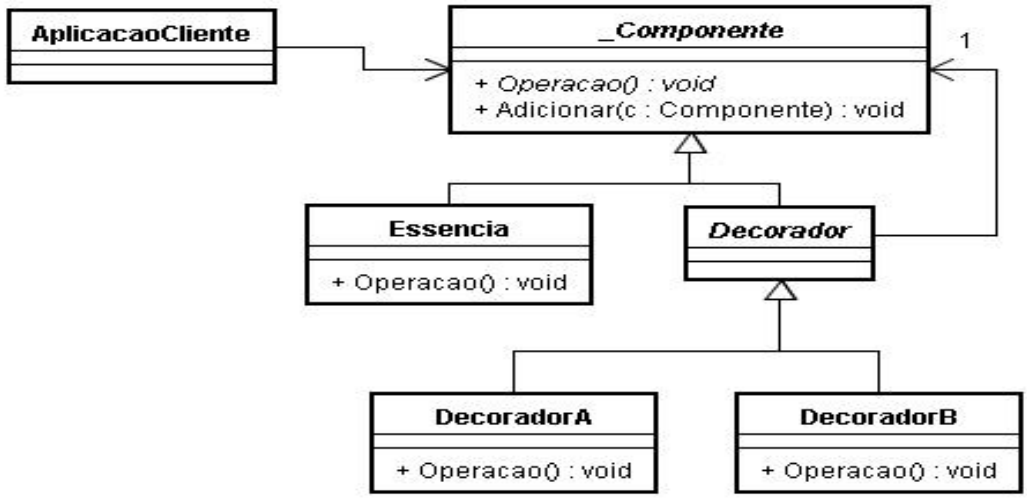

Figura 9: Estrutura do padrão Decorator
Fonte: CALSAVARA,2007.

Quando se deseja, por exemplo, adicionar a funcionalidade de barras de rolagem em uma caixa de texto pode-se utilizar de herança, porém esse recurso pode causar uma grande quantidade de subclasses ou a definição da classe pode não estar disponível para isso.

O padrão Decorator resolve esse problema adicionando a função de barra de rolagem à caixa de texto de maneira dinâmica e sem afetar a classe, isto é, é possível que o cliente utilize da caixa de texto sem as barras de rolagem ou com as barras de rolagem. Outro fato relevante é o fato de uma funcionalidade poder ser adicionada duas vezes, como borda dupla, por exemplo.

As vantagens de utilizar esse padrão é evitar a criação de uma grande quantidade de subclasses, poder definir classes simples, onde somente o que é fundamental será implementado além de poder adicionar funcionalidades de forma dinâmica.

4.5 PADRÃO FAÇADE

“Fornecer uma interface para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado”. (GAMMA, 2000, p.179).

O padrão Façade trata de associações entre classes e objetos, sendo geralmente utilizado em projetos orientados a objetos. Sua função é fornecer uma interface unificada para um conjunto de interfaces em um subsistema, ou seja, é definida uma interface de nível mais alto que torna o subsistema mais fácil de ser utilizado pelo cliente, como mostrado na figura 10.

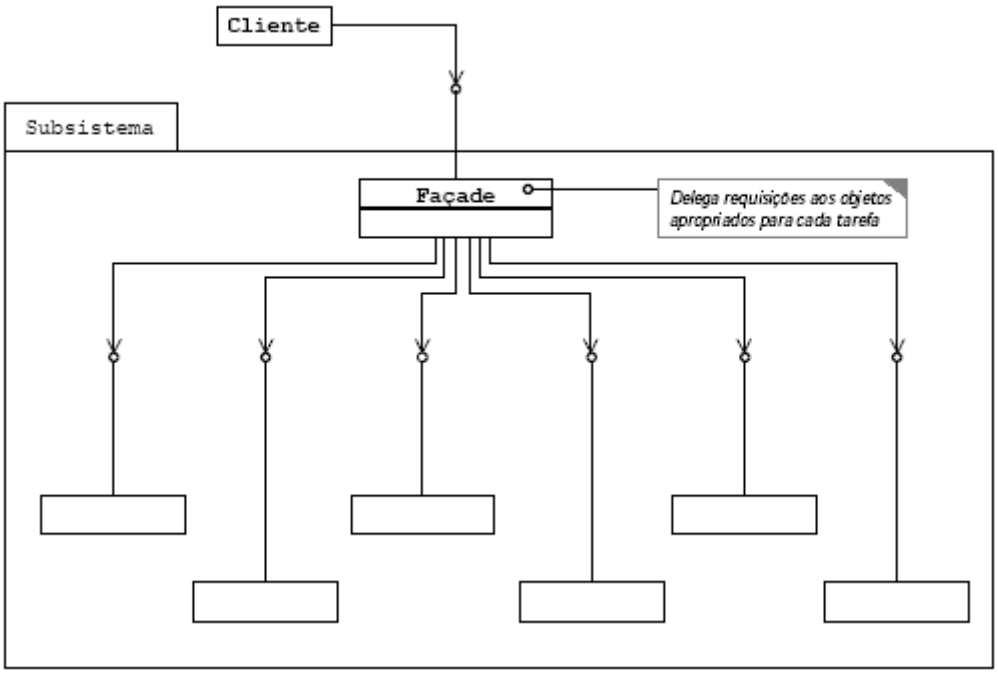

Figura 10 – padrão Façade
Fonte: ROCHA, 2003.

Sem o padrão Façade o Cliente precisa acessar todos os objetos do sistema e conhecê-los, mas com a implementação do Façade, o cliente acessa o Façade e ele se encarrega de acessar os demais objetos.

> “Façades podem ser utilizados não somente para criar uma interface mais simples em termos de chamadas a métodos, mas também para reduzir o número de objetos com os quais o objeto cliente deve lidar”. (SHALLOWAY, 2004, p.109).

No caso de termos um objeto Cliente que deve lidar com BaseDados, Modelos e Elementos, sem o padrão Façade, como na figura 11, o Cliente deve primeiro abrir a BaseDados para obter um Modelo. Só ai ele consulta o Modelo e obtém acesso o Elemento. Só agora ele pode acessar o Elemento. A figura 12 mostra o acesso do Cliente após a implementação do padrão Façade, ele acessa BaseDados Façade e BaseDados Façade se encarrega de encaminhar o Cliente para onde ele precisa naquele momento.

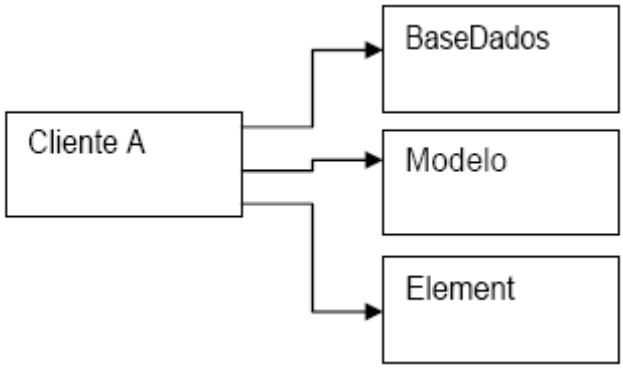

Figura 11 – antes do Façade
Fonte: SHALLOWAY, 2004.

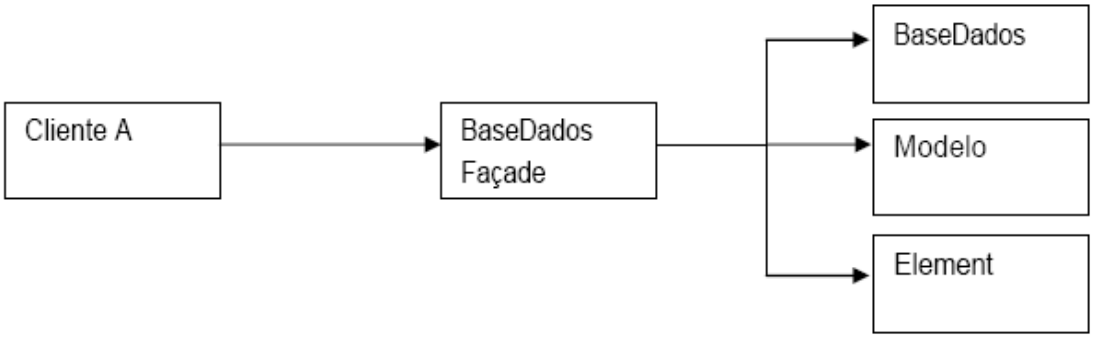

Figura 12 – usando Façade
Fonte: SHALLOWAR, 2004.

Segundo Gamma (2000), os principais benefícios do padrão Façade são:

- Isola os clientes dos componentes do subsistema reduzindo o número de objetos com os quais o cliente tem que lidar.
- Promove um acoplamento fraco entre o subsistema e seus clientes. O acoplamento fraco permite variar os componentes do subsistema sem afetar os seus clientes.

- Não impede as aplicações de utilizarem as classes do subsistema caso necessitem.

O padrão façade possibilita usar um sistema complexo com mais facilidade ou, se necessário, um subconjunto dele. Ele permite ainda que usemos esse sistema de uma maneira particular. Isso torna o sistema mais simples e dentro das reais necessidades de utilização, fornecendo uma coleção de métodos mais fáceis de entender.

Na figura 13, temos um exemplo da implementação em Java desse padrão. Podemos notar que o Façade se coloca entre o objeto Aplicação e os demais objetos que precisam ser acessados para realização de uma compra como, BancodeDados, Cliente, Produto e Carrinho.

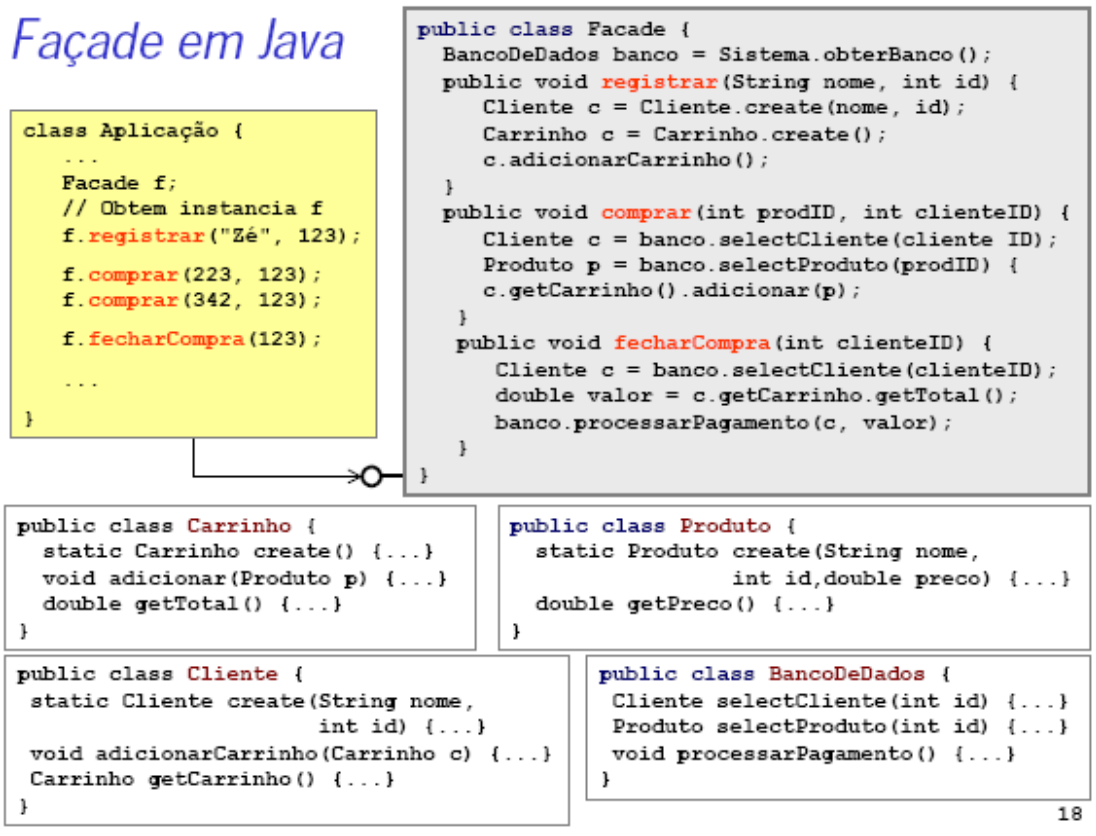

Figura 13 – padrão Façade em Java
Fonte: Rocha, 2003.

Na figura 14, temos a modelagem do exemplo citado na figura 13. Aqui vemos claramente o Façade em ação. A Aplicação acessa o Façade, que se encarrega de acessar os demais objetos necessários a realização da compra.

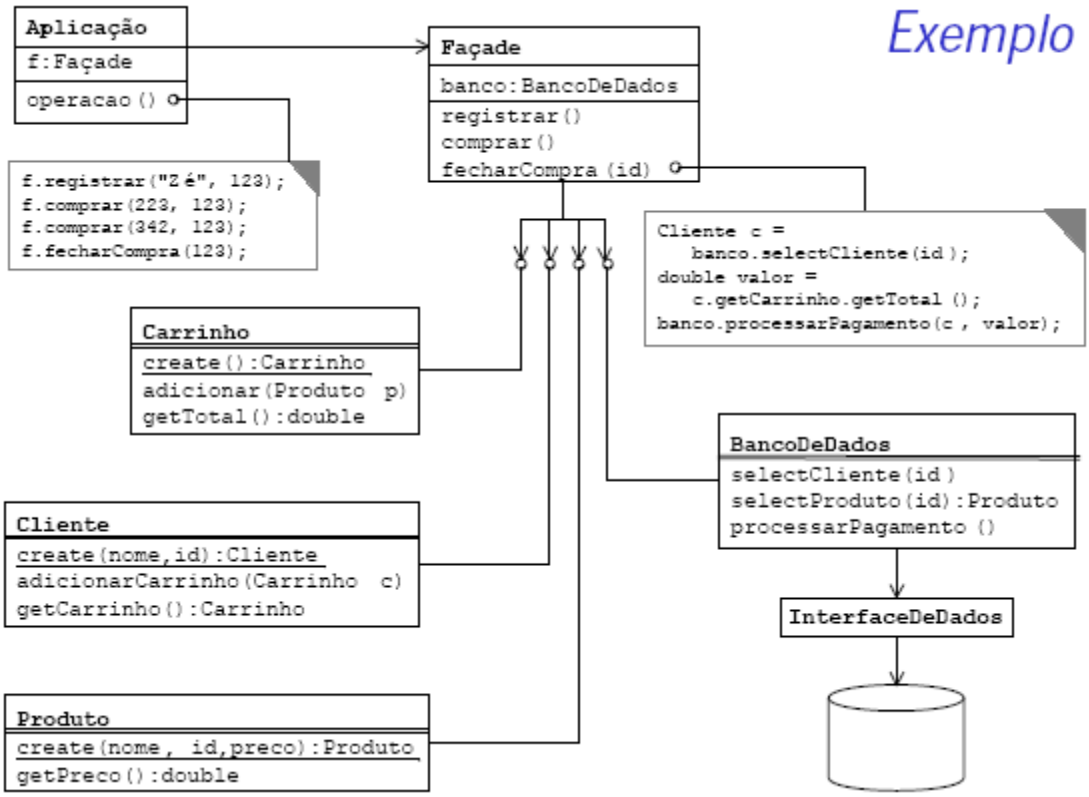

Figura 14 – exemplo de padrão Façade
Fonte: Rocha, 2003.

4.6 FLYWEIGHT

O padrão de projeto Flyweight é um padrão estrutural que tem por finalidade usar de compartilhamento de objetos de granularidade fina, isto é, objetos que são iguais exceto pequenos detalhes.

“Este padrão procura fatorar as informações comuns a diversos objetos em um único componente. Os demais dados, que tornam tais objetos únicos, passam a ser utilizados como parâmetros de métodos”. (PEREIRA, 2004, p. 25).

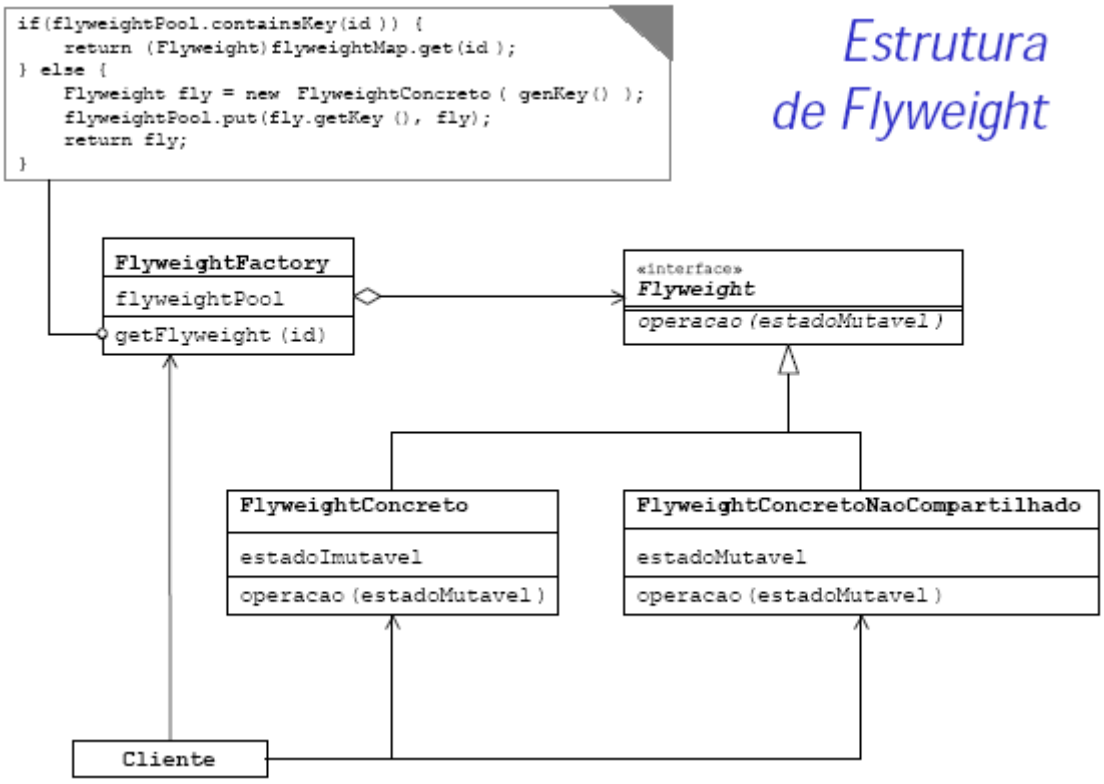

Figura 15 – Estrutura do padrão Flyweight
Fonte: Rocha, 2003.

Utilizando desse padrão é possível reduzir consideravelmente o número de objetos em uma aplicação devido ao fato de que objetos com as mesmas características são unificados em apenas um. Porém é importante ressaltar que esse padrão é indicado quando se tem uma aplicação com um número considerável de objetos que podem ser compartilhados, além de essa aplicação não depender da quantidade de objetos.

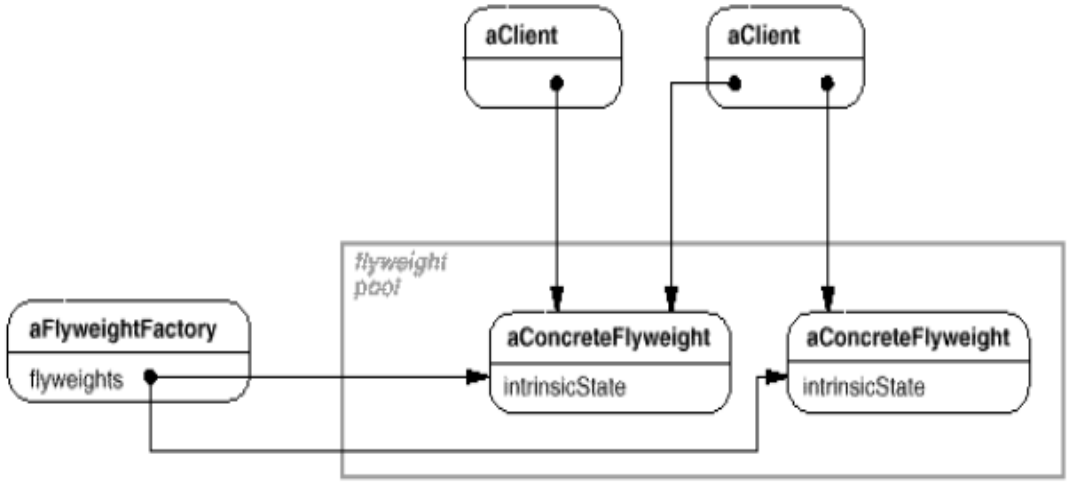

Figura 16: modelo de partilha de um objeto Flyweight
Fonte: Costa, 2007.

Como mostra a figura acima os objetos podem ser utilizados em diversos contextos simultaneamente, sempre de forma independente.

Existem dois estados dos objetos que devem ser levados em conta na adoção do padrão flyweight, os estados intrínseco e extrínseco. O primeiro diz respeito à parte do objeto que será imutável independente do cliente, estará armazenado no objeto Flyweight e é o estado que será compartilhado. O segundo está relacionado à parte do objeto que será mutável e estará armazenado no cliente.

A utilização de objetos flyweights depende muito da possibilidade de esses objetos assumirem um estado extrínseco, ou seja, da possibilidade de mutação que esse objeto pode assumir dependendo do contexto. Estado extrínseco que dizer as partes do objeto que não são dependentes desse. Por exemplo, um caracter tem como estado extrínseco o estilo e o posicionamento, e como estado somente intrínseco o seu código. Quando o estado intrínseco do objeto depende de muitos fatores então torna-se difícil o compartilhamento desse.

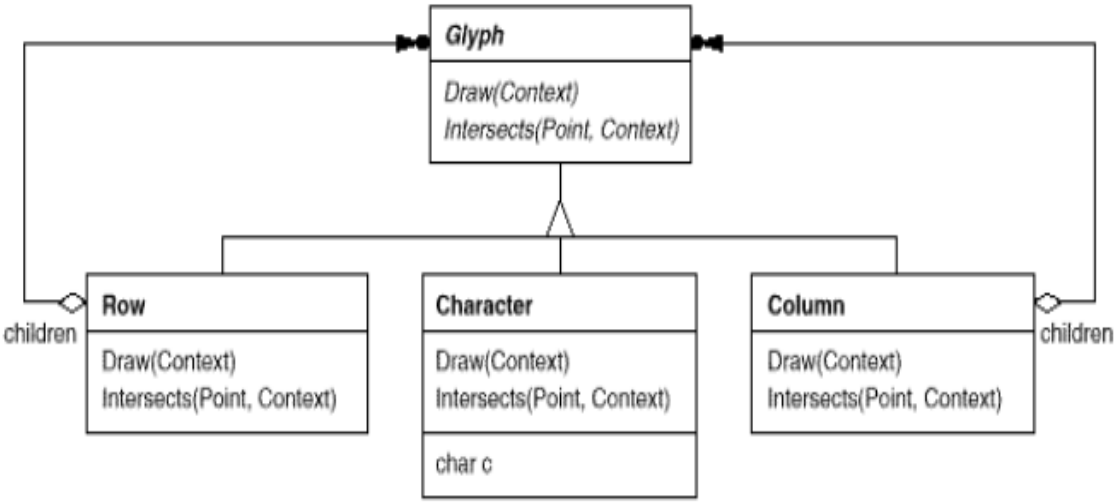

Figura 17: Exemplo de processador de texto
Fonte: Costa.

Na figura acima podemos ver os objetos que representam os objetos no documento, sendo que o objeto Row e o objeto Column são objetos de estado extrínseco, enquanto o objeto Character tem seu estado como intrínseco, isto é, sua posição e forma podem mudar dependendo do cliente, mas seu código continuará o mesmo.

A principal conseqüência do uso do padrão Flyweight é a economia de memória devido à redução do número de instâncias e quantidades de estado intrínseco. Segundo Costa (2007), “Podem ser introduzidos custos em termos de eficiência devido à transferência, procura e/ou computação do estado extrínseco [...]”, devido a isso antes de se adotar esse padrão de projeto é importante avaliar se a quantidade de memória economizada compensa realmente a possível perda de eficiência.

4.7 PADRÃO PROXY

“O padrão Proxy provê uma referência para um objeto com o objetivo de controlar o acesso a este”. (JUNIOR, 2007).

O padrão Proxy é utilizado quando um sistema quer utilizar um objeto real, mas por algum motivo ele não está disponível. A solução então é providenciar um substituto que possa se comunicar com esse objeto quando ele estiver disponível. O cliente passa a usar o intermediário em vez do objeto real. O intermediário possui uma referencia para o objeto real e repassa as informações do cliente, filtrando dados e acrescentando informações.

Rocha (2003) enumera vantagens e desvantagens de padrão.

- Vantagens:
  - Transparência, pois é utilizada a mesma sintaxe na comunicação entre o cliente e o objeto real;
  - Tratamento inteligente dos dados no cliente e
  - Maior eficiência com caching no cliente.
- Desvantagens:
  - Possível impacto na performance e
  - Fatores externos como queda da rede pode deixar o Proxy inoperante.

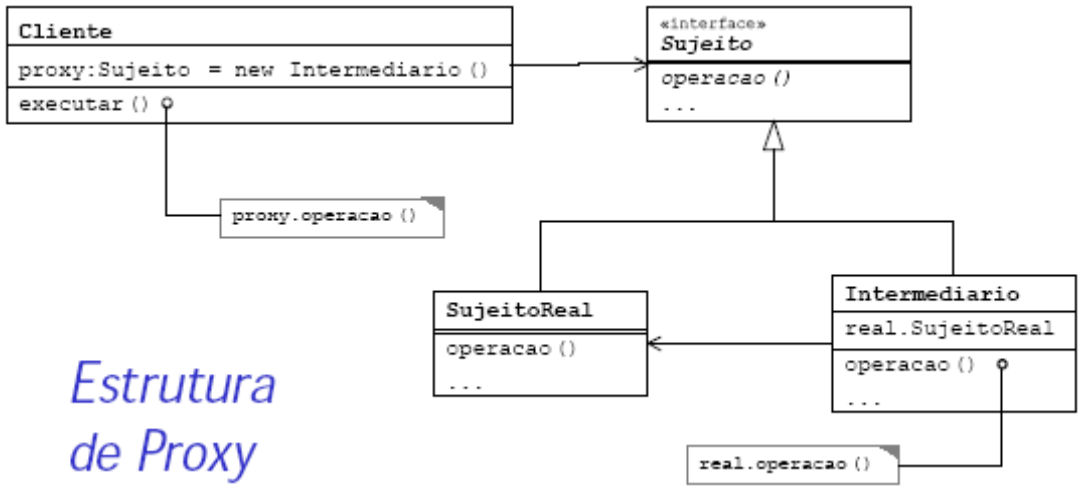

Figura 18 – padrão Proxy
Fonte: Rocha, 2003.

5 COMPORTAMENTAL

5.1 INTERPRETER

“O padrão Interpreter recebe uma linguagem e define uma representação para sua gramática junto com um interpretador que usa a representação para interpretar sentenças na linguagem”. (ROCHA, 2003).

O padrão Interpreter é uma extensão do padrão Command, em que toda uma lógica pode ser implementada com objetos, é pode ser utilizado para realizar uma composição com o command, resultando em uma linguagem de programação usando objetos.

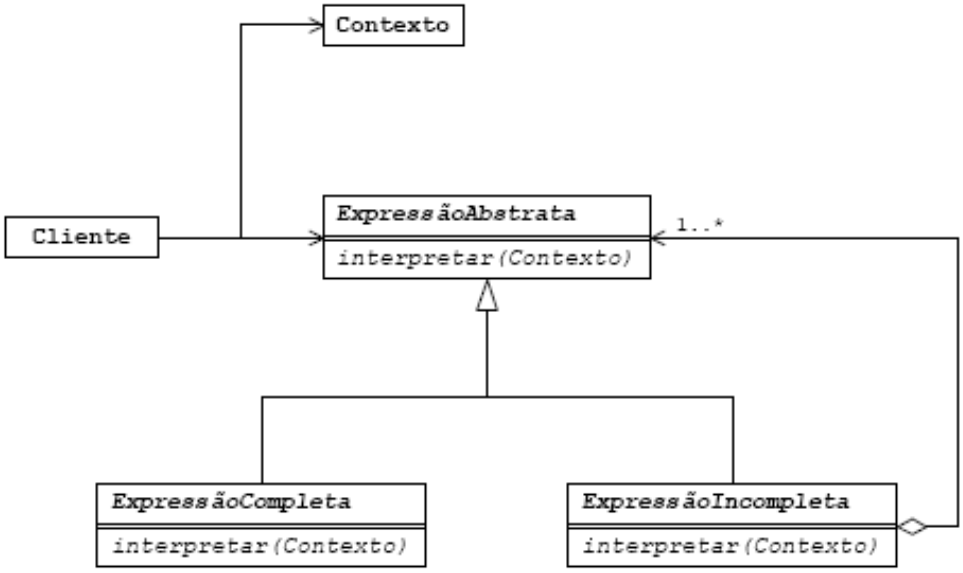

Figura 19 – padrão Interpreter
Fonte: Rocha, 2003.

5.2 TEMPLATE METHOD

O Template Method constitui-se um padrão cuja proposta é ajudar a abstrair um processo comum a partir de procedimentos diferentes.

> “De acordo com a Gangue dos Quatro, o padrão Template Method define um esqueleto de um algoritmo em uma operação, delegando alguns passos para subclasses e os redefinindo em um algoritmo, sem mudar a estrutura do mesmo”. (SHALLOWAY, 2004).

O padrão Template Method nos possibilita uma maneira de captar esse campo comum em uma classe abstrata, enquanto encapsula as diferentes em classes derivadas. Ele serve para controlar a seqüência comum em processos distintos.

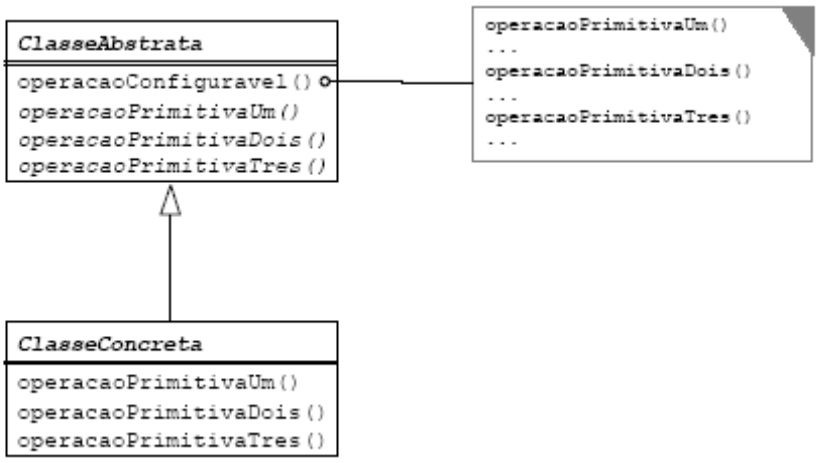

Figura 20 – padrão Template Method
Fonte: ROCHA, 2003.

Esse padrão é utilizado no momento em que a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidas por implementações que podem variar.

5.3 CHAIN OF RESPONSIBILITY

Em um sistema orientado a objetos esses interagem entre si através de mensagens, e o sistema necessita de determinar qual o objeto que irá tratar a requisição. O padrão de projeto Chain of Responsibility permite determinar quem será o objeto que irá tratar a requisição durante a execução. Cada objeto pode tratar ou passar a mensagem para o próximo na cascata.

Em um escritório, por exemplo, onde se tem 4 linha telefônicas, a primeira linha é o primeiro objeto, a segunda linha é o segundo, e assim sucessivamente até a gravação automática que é o quinto objeto. Se a primeira linha estiver disponível ela irá tratar a ligação, se não ela passa a tarefa para o próximo objeto, que é a segunda linha. Se essa estiver ocupada ela passa a tarefa para a próxima e assim sucessivamente até que um objeto possa tratar a tarefa. Nesse caso, se todas as linhas estiverem ocupadas o último objeto, que é a gravação automática, tratará da tarefa.

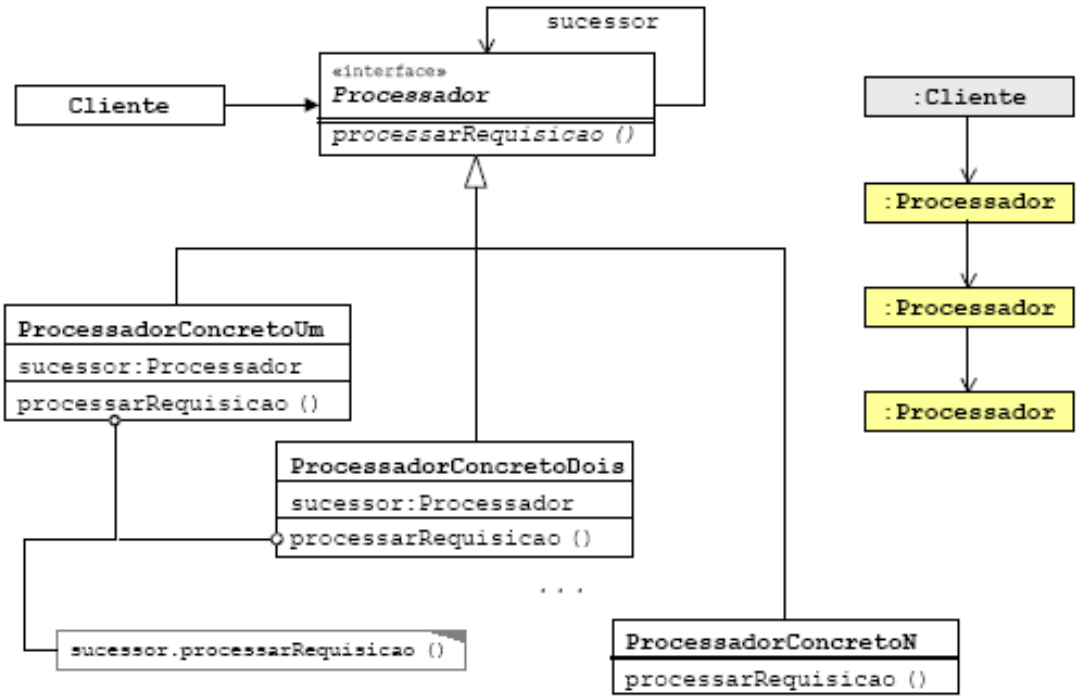

Figura 21: Estrutura de Chain of Responsibility
Fonte: Rocha, 2003

Esse padrão pode ser implementado de várias formas diferentes, sendo essas chamadas de estratégias ou idiom. Essas, por sua vez, podem ser determinadas de maneira a permitir maior ou menor acoplamento entre os participantes utilizando dos padrões mediator e delegation. Com o mediator só o mediator sabe quem é o próximo na cadeia, já com o delegation todos os objetos sabem quem é o seu próximo. (ROCHA, 2003)

Diante disso, pode-se afirmar ainda que é possível que vários objetos possam tratar uma tarefa, permitindo uma divisão de responsabilidades de forma clara.

5.4 COMMAND

O padrão Command é utilizado para gerenciar a relação de uma solicitação com os objetos quando há certa complexidade entre essas, isto é, quando há muitas requisições.

A utilização desse padrão tem por finalidade encapsular um comando de modo que esse comando possa ser utilizado em diversos objetos (DEITEL, 2003).

Quando existe uma interface gráfica onde existe uma grande probabilidade de que o usuário faça várias solicitações é inevitável o aumento da complexidade. Um editor de texto, por exemplo, tem um menu, um menu pop-up e uma barra de ferramentas que tem os mesmos itens, como tamanho da fonte, cor da fonte, etilo da fonte, para facilitar para o usuário. Porém, para o programador é necessário, apesar de os itens serem os mesmos, codificar três vezes a mesma funcionalidade.

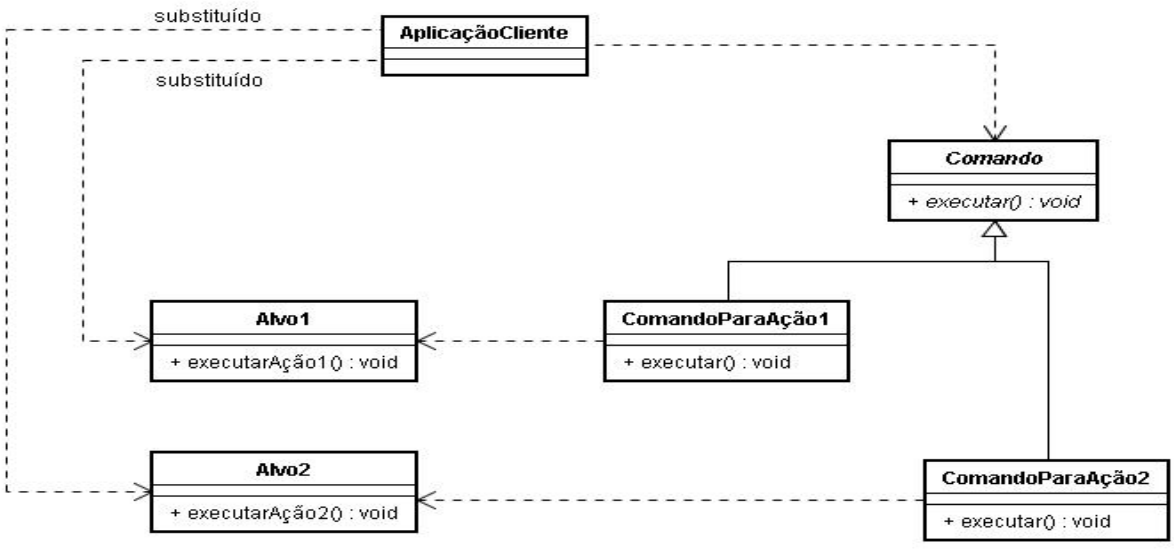

Figura 22: Estrutura do padrão Command
Fonte: CALSAVARA, 2007.

O padrão Command vem a solucionar esse problema permitindo que a funcionalidade desejada seja especificada uma só vez em um objeto reutilizável. Então essa funcionalidade pode ser adicionada a um menu, barra

de ferramentas e outras estruturas sem a necessidade de ser especificada novamente.

Diante disso, temos como conseqüências da utilização desse padrão uma menor dependência do objeto que faz a solicitação em relação ao objeto que executa a operação e o fato de ser mais fácil acrescentar novas funcionalidades a um objeto.

Esse padrão pode ser utilizado com os padrões Composite, Memento, Prototype e Singleton.

5.5 ITERATOR

“Prover uma maneira de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação interna.” (ROCHA, 2003).

Esse padrão é utilizado para acessar o conteúdo de um agregado sem expor sua representação interna, oferecendo uma interface uniforme para atravessar diferentes estruturas agregadas.

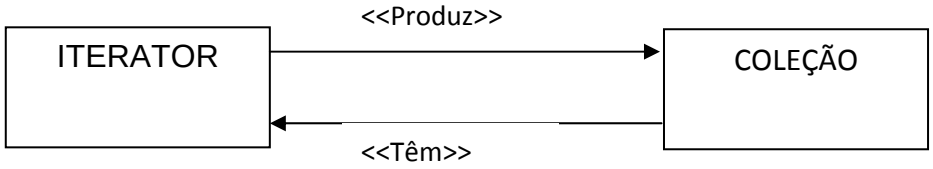

Figura 23 – padrão Iterator
Fonte: ROCHA, 2003.

Em Java, Iterator é implementado através do método iterator() de Collection, devolvendo uma instância de Java.util.iterator, como na figura 24.

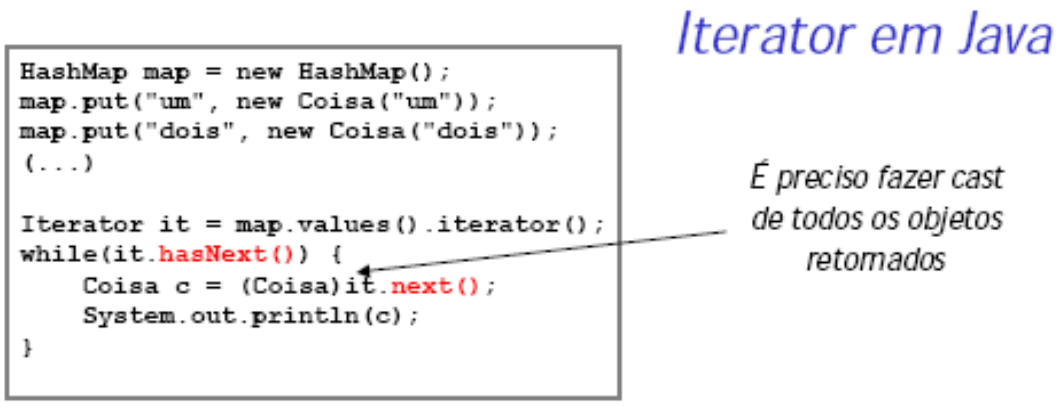

Figura 24 – Iterator em Java
Fonte: ROCHA, 2003.

5.6 MEDIATOR

O padrão Mediator é utilizado como um mediador entre objetos, isto é, quando se tem vários objetos que se comunicam entre si pode-se utilizar um objeto para encapsular a maneira como os outros objetos se comunicam. Os objetos se comunicam diretamente com o objeto mediator e esse se comunica com os outros enviando as mensagens referidas.

De acordo com Besen (2007), o padrão Mediator é utilizado para,

> "[...] particionar um sistema em pequenos pedaços, simplificar o relacionamento entre objetos, diminuir o número de subclasses necessárias para alterar um comportamento, melhorar a reusabilidade de objetos, simplificar o protocolo dos objetos, etc."

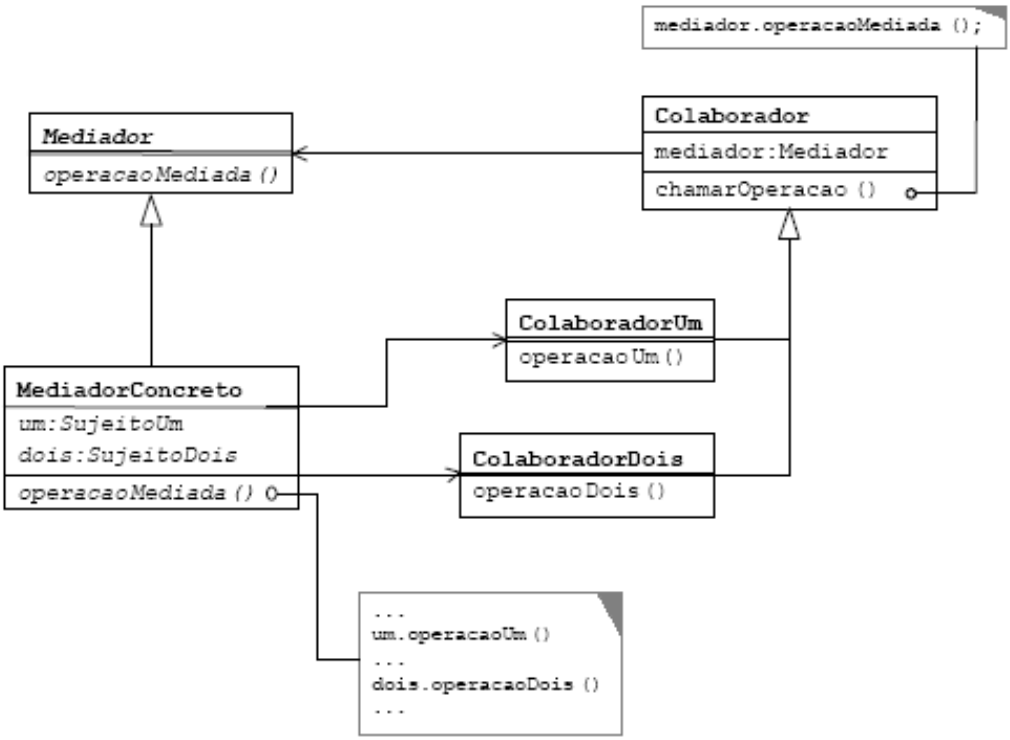

Figura 25: Estrutura do padrão Mediator
Fonte: Rocha, 2003

Como um exemplo de utilização desse padrão podemos citar um interface gráfica de usuário, onde se tem diferentes botões, caixas de seleção, entre outros que interagem entre si. Quando um botão é clicado um caixa de seleção é acionada, por exemplo. Nesse relacionamento é possível colocar um objeto

mediator que recebe os eventos e envia aos outros objetos relacionados para que esses executem suas ações (BESEN, 2007).

5.7 MEMENTO

“O padrão Memento captura e externaliza o estado interno de um objeto, sem violar o encapsulamento, para que o objeto possa ter esse estado restaurado posteriormente”. (ROCHA, 2003).

O padrão Memento é um repositório para guardar o estado dos objetos. Pode ser usado outro objeto, uma string ou um arquivo. É utilizado quando é necessário saber o estado anterior de um objeto, para se desfazer uma operação, mas as informações não podem ser visíveis a todos os objetos envolvidos.

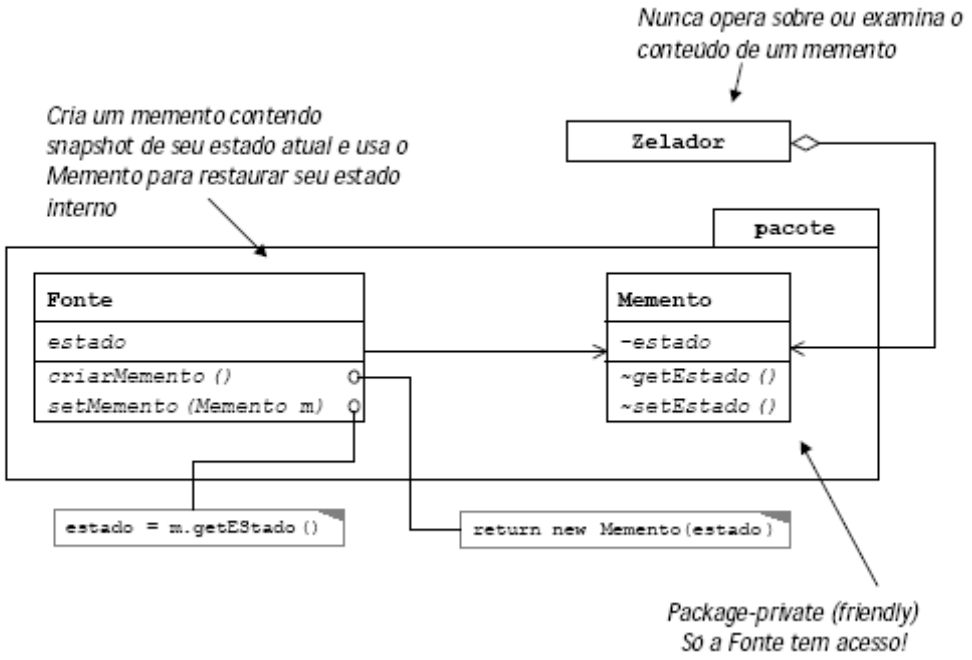

Figura 26 – padrão Memento
Fonte: Rocha, 2003.

5.8 OBSERVER

“O padrão Observer define uma dependência um-para-muito entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente”. (ROCHA, 2003).

Esse é um padrão muito comum, recebendo ainda os nomes de Dependents e Publish-Subscribe. Quando se têm muitos objetos que dependem de outro objeto, é necessário que, quando houver mudanças nesse objeto central, os outros objetos que dele dependem saibam dessas mudanças, para isso é necessário utilizar o padrão Observer.

Segundo Rocha(2003) destaca como desvantagem desse padrão a queda de performance, visto que o abuso do uso desse padrão pode causar sério impacto na performance do sistema.

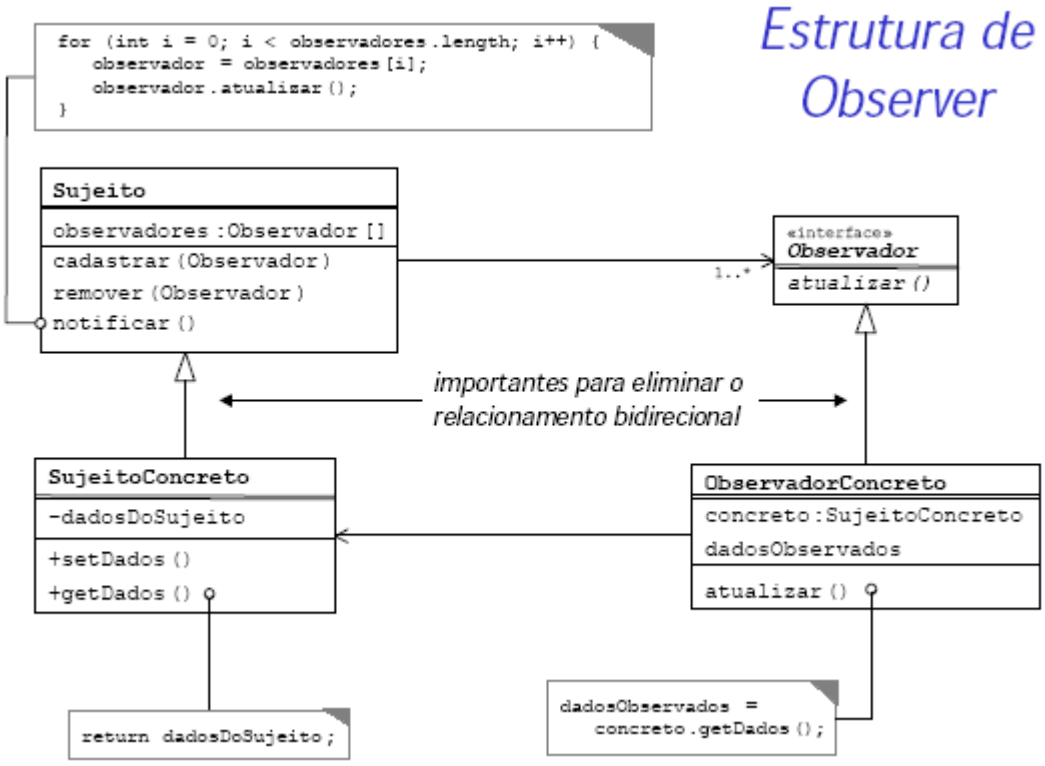

Figura 27 – padrão Observer
Fonte: Rocha, 2003.

5.9 STATE

O padrão State permite resolver problemas relacionados com os estados dos objetos. Quando se tem um objeto que pode assumir vários comportamentos em tempo de execução dependendo do estado em que se encontra, pode-se criar uma classe abstrata que disponibilizará métodos que descreverão possíveis estados que um objeto pode assumir.

Esse recurso é utilizado quando o comportamento do objeto depende do estado em que se encontra.

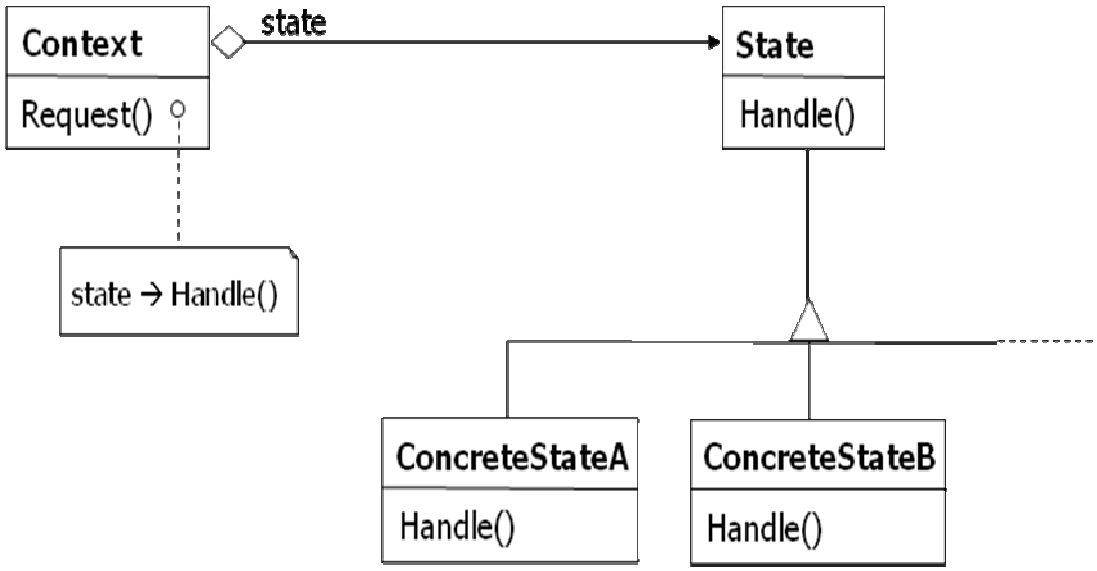

Figura 28: Padrão State
Fonte: Vanini, 2007.

A estrutura acima mostra como esse padrão trabalha com os estados dos objetos. Um objeto Context se relaciona com a superclasse State passando solicitações referentes ao estado. A superclasse State contém os possíveis estados que um objeto pode assumir, nesse caso os estados A e B, representados pelas subclasses ConcreteStateA e ConcreteStateB. Através das referências passadas pelo objeto Context sobre si mesmo é definido o estado que esse assumirá, no caso acima A ou B.

5.10 STRATEGY

“De acordo com a Gangue dos Quatro, o padrão Strategy define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis”. (SHALLOWAY, 2004).

O padrão Strategy é utilizado quando classes relacionadas forem diferentes apenas no seu comportamento. Ele oferece uma maneira de configurar uma classe com várias possibilidades de comportamento. Com esse padrão é possível implementar as classes utilizando polimorfismo.

SHALLOWAY (2004) enumera alguns princípios sobre os quais o padrão Strategy está baseado:

- Os objetos têm responsabilidades;
- As implementações diferentes são realizadas com o uso de polimorfismo;
- Necessidade de gerenciar diferentes implementações do mesmo algoritmo;
- Constitui uma boa prática de projeto desacoplar um comportamento de outro que ocorra no domínio do problema.

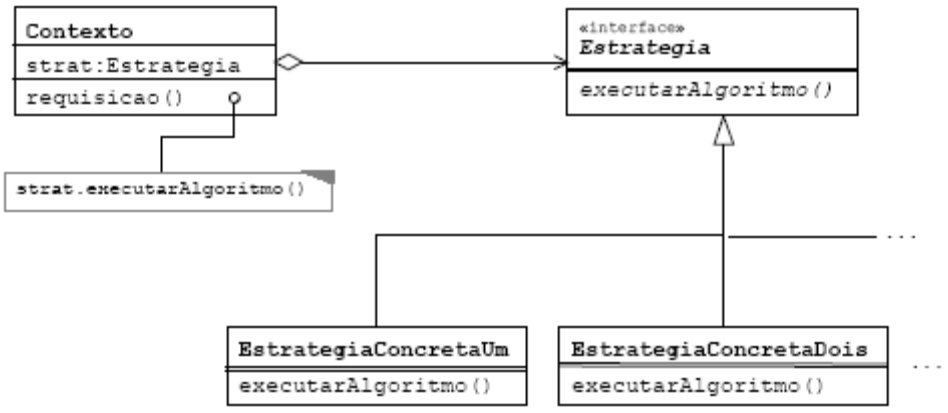

Figura 29 – padrão Strategy
Fonte: Rocha, 2003.

6 CONCLUSÃO

Nas últimas décadas a tecnologia tem exercido papel fundamental para o homem, seja em suas atividades no trabalho, domésticas ou lúdicas. O grande crescimento e avanço de equipamentos que propiciem maior comodidade e otimização nessas atividades têm requerido softwares que colaborem ainda mais para isso.

A atual necessidade de programas e sistemas cada vez mais eficientes para ocuparem o mesmo nível dos hardwares de alta tecnologia faz do trabalho do programador uma tarefa complexa.

Diante disso, os padrões de projeto, introduzidos pioneiramente por Christopher Alexander na arquitetura e identificados na computação pela gangue dos quatro, composta por Gamma, Helm, Johnson e Vlissides, tem considerável importância para a otimização de um software.

> “Cada padrão descreve um problema que ocorre repetidamente no nosso ambiente e, portanto, descreve o cerne da solução desse problema, de tal forma que você pode utilizar essa solução um milhão de vezes repetidas, sem nunca fazê-la duas vezes do mesmo modo.” (Alexander apud Shalloway, 2004, p. 94)

Os padrões de projeto são parte fundamental na elaboração e construção de um software, proporcionando mais eficiência, otimização e diminuição da complexidade do mesmo, atuando no âmbito comportamental, estrutural e de criação do software.

7 REFERÊNCIAS

1 DEITEL, Harvey M.; DEITEL, Paul J. **JAVA - Como programar**. 4 ed. Porto Alegre : Bookman, 2003.

2 GAMMA, Erick; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John; Trad. Luiz A. Meirelles Salgado. **Padrões de Projeto Soluções Reutilizáveis de Software Orientado a Objetos.** Porto Alegre: Bookman, 2000.

3 SHALLOWAY, Alan; TROTT, James R.; Trad. Ana M. de Alencar Price. **Explicando Padrões de Projeto: Uma Nova Perspectiva em Projeto Orientado a Objeto.** Porto Alegre: Bookman, 2004.

4 Sites

ADRIANO, Luis Carlos; ARAÚJO, Everton Coimbra de. **Utilização de Padrões de Projetos em Aplicações para Arquiteturas Multicamadas.** Disponível em: <http://ssooweb01.univali.br/prof/Fernanda%20Pimentel%20Ribeiro/cd/Paineis/21%20out%20-%20Padroes%20de%20Projeto%20em%20Aplicacoes%20Multicamadas%20-%20Lui.pdf>.
Acesso em: 29 Out. 2007.

BESEN, Renato. **Padrão de Projeto Mediator**. Abril 2007.
Disponivel em: <http://www.inf.ufsc.br/~renatob/20071/engenharia_de_software/Design%20Patterns%20-%20Mediator.pdf>.
Acesso em: 30 Out. 2007

CALSAVARA, Alcides. **Padrões de projeto.**
Disponível em: <http://www.ppgia.pucpr.br/~alcides>.
Acesso em: 29 Out. 2007

COSTA, Luís; MARQUES, Marlene. **Flyweight Pattern.**
Disponível em: <http://www.deetc.isel.ipl.pt/Programacao/Programacao_Inv_2004_2005/cp/Turmas/I701/Apresentações/Flyweight%20Pattern.ppt>.
Acesso em: 02 Nov. 2007

DESTRO, Daniel. **Implementando Design Patterns com Java.** 2006. Disponível em: <http://www.guj.com.br/java.tutorial.artigo.137.1.guj>.
Acesso em: 03 Nov. 2007.

JUNIOR, Amadeu Andrade Barbosa. **Padrões de Projeto: Proxy.** 28 de Maio de 2007. Disponível em: <http://twiki.im.ufba.br/bin/viewfile/MAT061/Padroes20071?rev=1;filename=proxy.pdf>.
Acesso em: 03 Nov. 2007.

PEREIRA, Fernando M.Q.; VALENTE, Marco Túlio O.; BIGONHA, Mariza A. S.; BIGONHA, Roberto S..**Chamada Remota de Métodos na Plataforma J2ME/CLDC.**
Disponível em:
<http://compilers.cs.ucla.edu/fernando/publications/journals/Inatel2004.pdf>.
Acesso em: 02 Nov. 2007.

ROCHA, Helder da. **J930: GoF Designer Patterns em Java.** Fevereiro de 2003.
Disponível em: <http://www.argonavis.com.br/cursos/java/j930/index.html>.
Acesso em: 27 Out. 2007.

VANINI, Fernando. **Padrões de Projeto - Design Patterns**
Disponível em:
<http://www.ic.unicamp.br/~vanini/mc747/PadroesdeProjeto.pdf>.
Acesso em: 02 Nov. 2007