UNIVERSIDADE
ESTÁCIO DE SÁ

PÓS-GRADUAÇÃO

Flávio Pernes de Medeiros

METODOLOGIA DA PESQUISA II

Rio de Janeiro
Flávio Pernes de Medeiros

Março de 2011

Agradecimentos

A minha família em especial minha esposa Newli Maura e filhos Bruno e Nicole.

Aos professores em especial ao Oswaldo Borges.

A Deus, pelo dom da vida.

RESUMO

Um padrão de projeto é uma estrutura recorrente no projeto de software orientado a objetos. Pelo fato de ser recorrente, vale a pena que seja documentada e estudada.

Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável.

Palavras-Chave: Java, Padrões, Design, Patterns, Reuso.

ABSTRACT

Design Patterns describe solutions to recurring problems in developing systems of object-oriented software. A design pattern provides a name and define the problem, the solution when this solution and its consequences.

The design patterns to facilitate reuse of design solutions - that is, solutions in the design phase of the software, without considering code reuse. They will also carry a common vocabulary of design, facilitating communication, learning and documentation of software systems.

Padrões de projeto Estrutural

Monografia apresentada à Universidade Estácio de Sá, como requisito final do curso de pós-graduação em Desenvolvimento de sistemas - JAVA.

Orientador: Prof. JOÃO MENDES

Rio de Janeiro
03/2011

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 – Tema

Com a maturidade na engenharia de softwares, novos aspectos e novas abordagens foram surgindo até atingirmos o que chamamos de padrões de projetos.

Seu uso busca o mapeamento de soluções para problemas similares e ou conhecidos, ao invés de se desenvolver soluções a cada novo problema, busca-se primeiro identificar os problemas e aplicar a solução conhecida, diminuindo assim, erros, tempo de construção e praticando reuso.

Neste trabalho abordaremos a aplicabilidade dos padrões estruturais.

O conceito de padrão de projeto foi criado na década de 70 pelo arquiteto Christopher Alexander.[1] Em seus livros, ele estabelece que um padrão deve ter, idealmente, as seguintes características:

No presente trabalho iremos detalhar apenas o padrão estrutural.

Encapsulamento: um padrão encapsula um problema/solução bem definida. Ele deve ser independente, específico e formulado de maneira a ficar claro onde ele se aplica.

Generalidade: todo padrão deve permitir a construção de outras realizações a partir deste padrão.

Equilíbrio: quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto. Uma análise racional que envolva uma abstração de dados empíricos, uma observação da

aplicação de padrões em artefatos tradicionais, uma série convincente de exemplos e uma análise de soluções ruins ou fracassadas pode ser a forma de encontrar este equilíbrio.

Abstração: os padrões representam abstrações da experiência empírica ou do conhecimento cotidiano.

Abertura: um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.

Combinatoriedade: os padrões são relacionados hierarquicamente. Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo. Além da definição das características de um padrão, Alexander definiu o formato que a descrição de um padrão deve ter. Ele estabeleceu que um padrão deve ser descrito em cinco partes:

Nome: uma descrição da solução, mais do que do problema ou do contexto.

Exemplo: uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação.

Contexto: a descrição das situações sob as quais o padrão se aplica.

Problema: uma descrição das forças e restrições envolvidos e como elas interagem.

Solução: relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo

com o padrão, freqüentemente citando variações e formas de ajustar a solução segundo as circunstâncias.

Inclui referências a outras soluções e o relacionamento com outros padrões de nível

mais baixo ou mais alto.

Em 1987, a partir dos conceitos criados por Alexander, os programadores Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área da ciência da computação. Em um trabalho para a conferência OOPSLA, eles apresentaram alguns

padrões para a construção de janelas na linguagem Smalltalk.[2] Nos anos seguintes Beck, Cunningham e outros seguiram com o desenvolvimento destas idéias.

O movimento ao redor de padrões de projeto ganhou popularidade com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, publicado em 1995. Os autores desse livro são Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF". Posteriormente, vários outros livros do estilo foram publicados, como *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, que introduziu um conjunto de padrões conhecidos como **GRASP** (*General Responsibility Assignment Software Patterns*).

1.2 – Delimitação

Neste trabalho iremos abordar os tipos de problemas e a forma correta de uso e aplicabilidade dos padrões de projeto, embora haja 23 padrões de projeto, classificados em 3 grandes grupos: Criacionais, Estruturais e Comportamentais, o foco do nosso trabalho será o padrão Estrutural.

1.3 – Justificativa

É evidente que ao seguir padrões de desenvolvimento, as equipes acabarão trazendo aspectos positivos para seus projetos, além de tornar seus softwares mais fáceis de manter, conseqüentemente reduzindo seu tempo e custo de manutenção.

1.4 – Objetivos

Existem vários objetivos e com a aplicabilidade de padrões de projeto, Os softwares só tendem a ganhar, não importa o tamanho do projeto, pode ser um pequeno projeto ou um grande projeto, seguir padrões sempre acabará trazendo vantagens, as principais que podemos citar são:

Facilidade na manutenção;

Melhoria da documentação;

Melhora a visão geral do sistema;

Melhora a leitura e entendimento do código;

Facilita o desenvolvimento com equipes grandes;

Permite que seus códigos sejam facilmente entendidos por outros, além de você.

1.5 – Metodologia

Este trabalho foi realizado mediante pesquisas e levantamento de material bibliográfico, além de consultas a informações constantes na internet, tendo como enfoque artigos científicos e aplicações atuais que tratassem sobre o tema abordado.

1.6 – Descrição

Além deste primeiro capítulo introdutório, no capítulo dois definimos e contextualizamos os padrões de projeto dentro do processo de desenvolvimento de softwares. No capítulo três descrevemos as técnicas e classificação dos padrões, bem como suas aplicabilidades. No capitulo quatro evidenciamos nossas conclusões.

Capítulo 2

Descobrindo padrões no projeto

2.1 – Principal alvo

O alvo principal do uso dos padrões de projeto no desenvolvimento de software é o da orientação a objetos. Como os objetos são os elementos chaves em projetos OO, a parte mais difícil do projeto é a decomposição de um sistema em objetos. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização e assim por diante. Todos influenciam a decomposição, freqüentemente de formas conflitantes. [3]

Muito dos objetos participantes provêm do método de análise. Porém, projetos orientados a objetos acabam sendo compostos por objetos que não possui uma contrapartida no mundo real.

2.2 – Abstrações

As abstrações que surgem durante um projeto são as chaves para torná-lo flexível. Os padrões de projeto ajudam a identificar abstrações menos óbvias bem como os objetos que podem capturá-las. Por exemplo, objetos que representam processo ou algoritmo não ocorrem na natureza, no entanto, eles são uma parte crucial de projetos flexíveis. Esses objetos são raramente encontrados durante a análise ou mesmo durante os estágios iniciais de um projeto; eles são descobertos mais tarde, durante o processo de tornar um projeto mais flexível e reutilizável.

2.3 – Como escolher o padrão certo

Escolher dentre os padrões existentes aquele que melhor soluciona um problema do projeto, sem cometer o erro de escolher de forma errônea e torná-lo inviável, é uma das tarefas mais difíceis. Em suma, a escolha de um padrão de projeto a ser utilizado, pode ser baseada nos seguintes critérios:

Considerar como os padrões de projeto solucionam problemas de projeto.

Examinar qual a intenção do padrão, ou seja, o que faz de fato o padrão de projeto, quais seus princípios e que tópico ou problema particular de projeto ele trata (soluciona).

Estudar como os padrões se relacionam.

Estudar as semelhanças existentes entre os padrões.

Examinar uma causa de reformulação de projeto

Considerar o que deveria ser variável no seu projeto, ou seja, ao invés de considerar o que pode forçar uma mudança em um projeto, considerar o que você quer ser capaz de mudar sem reprojetá-lo.

Capítulo 3

Padrões de projeto em detalhes

Padrões estruturais

3.1 Adapter

"Objetivo: converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces." [4]

3.1.1 Quando usar

Sempre que for necessário adaptar uma interface para um cliente.

3.1.2 Classe Adapter

Quando houver uma interface que permita a implementação estática.

3.1.3 Objeto Adapter

Quando menor acoplamento for desejado.

Quando o cliente não usa uma interface Java ou classe abstrata que possa ser estendida.

3.1.4 Onde estão os adapters

A API do J2SDK.

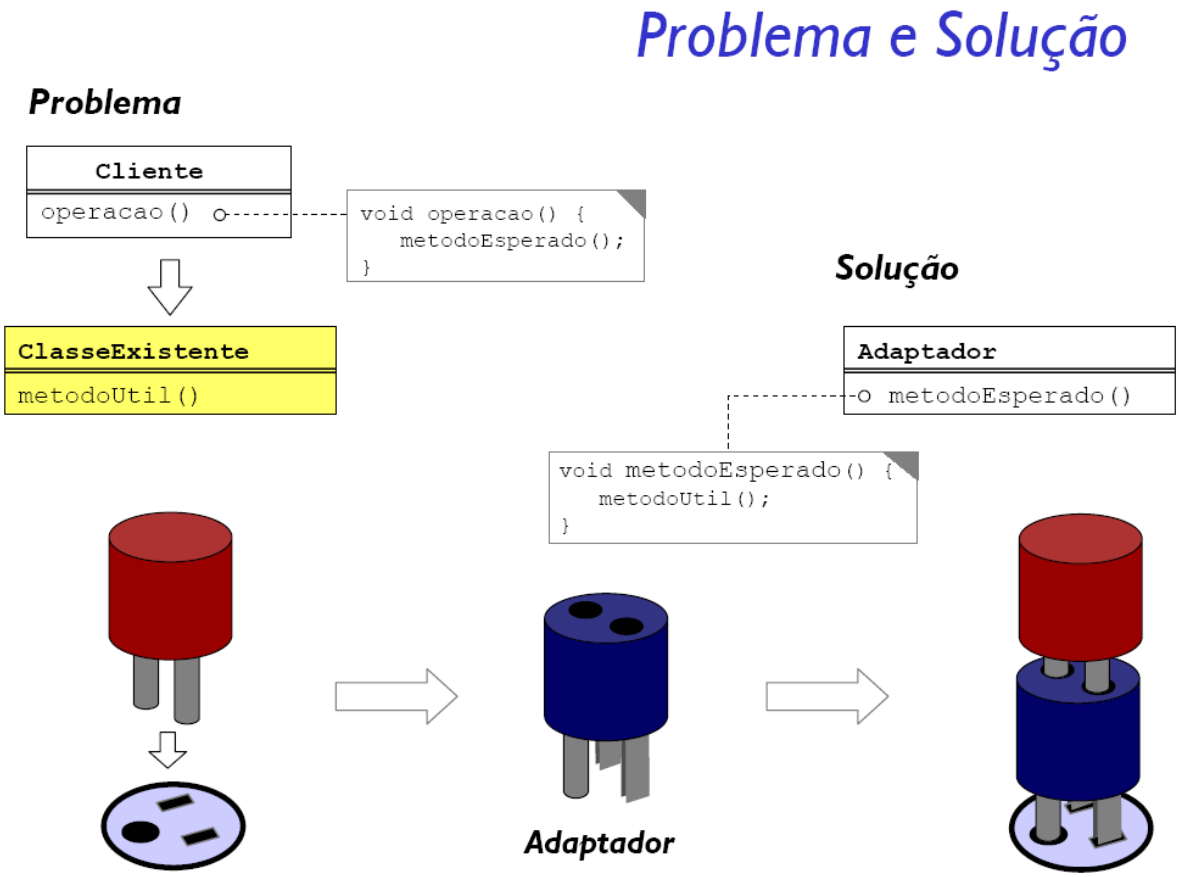


Figura 3.1 – Exemplo da representação do padrão de projeto adapter.

3.1.5 Duas formas de Adapter

Classe Adapter: usa herança múltipla

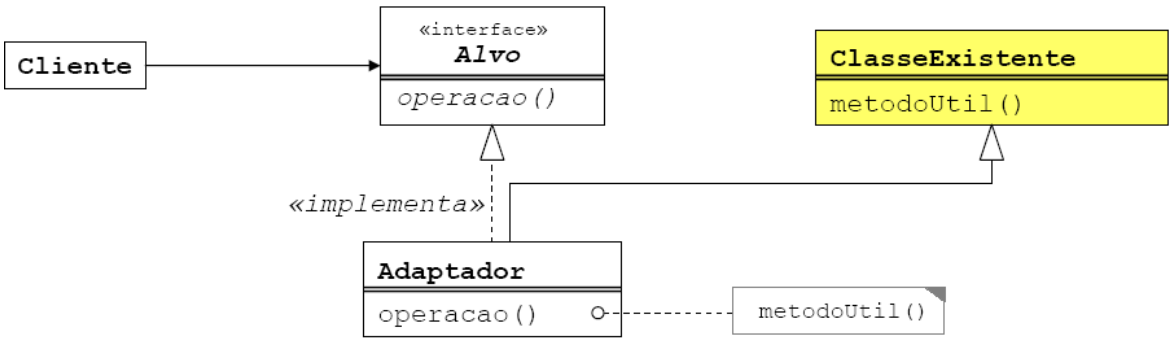


Figura 3.1.5 – Exemplo da representação do padrão de projeto adapter com herança multipla.

Cliente: aplicação que colabora com objetos aderentes à interface Alvo

Alvo: define a interface requerida pelo Cliente

Classe Existente: interface que requer adaptação

Adaptador(Adapter): adapta a interface do Recurso à interface Alvo

3.1.6 Objeto Adapter: usa composição

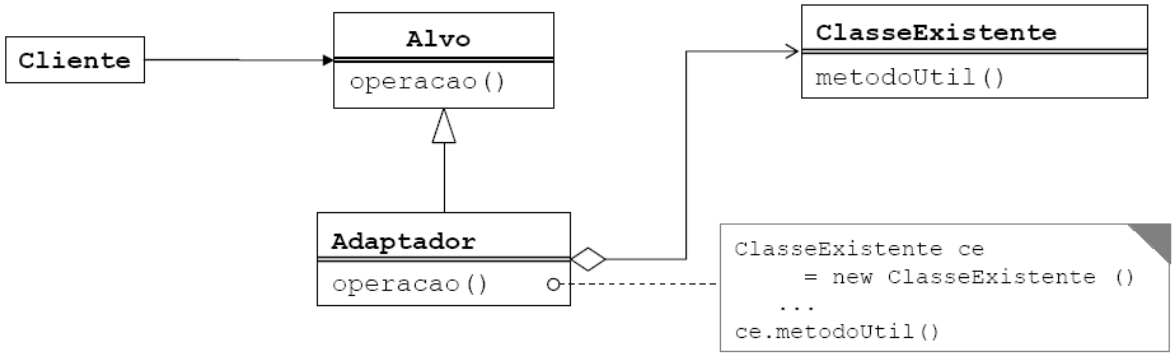


Figura 3.1.6 – Exemplo da representação do padrão de projeto adapter usando composição.

Única solução se Alvo não for uma interface Java

Adaptador possui referência para objeto que tera sua interface adaptada (instância de

Classe Existente).

Cada método de Alvo chama o(s) método(s) correspondente(s) na interface adaptada.

3.1.7 Implemantação em Java do padrão adapter

Classe Adapter:

```
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvo.operacao();
        }
    }
}
```

```
public interface Alvo {
    void operacao();
}
```

```
public class Adaptador extends ClasseExistente implements Alvo {
    public void operacao() {
        String texto = metodoUtilDois("Operação Realizada.");
        metodoUtilUm(texto);
    }
}
```

```
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

Figura 3.1.7 – Exemplo de implementação java do padrão de projeto adapter.

3.2 Bridge

"Desacoplar uma abstração de sua implementação para que os dois possam variar independentemente." [4]

3.2.1 Motivação:

O que motiva a utilização do padrão Bridge é a necessidade de um driver, permitindo implementações específicas para tratar objetos em diferentes meios persistentes.

3.2.2 Aplicabilidade:

Quando for necessário evitar uma ligação permanente entre a interface e a implementação.

Quando alterações na implementação não puderem afetar clientes.

Quando implementações são compartilhadas entre objetos desconhecidos do cliente.

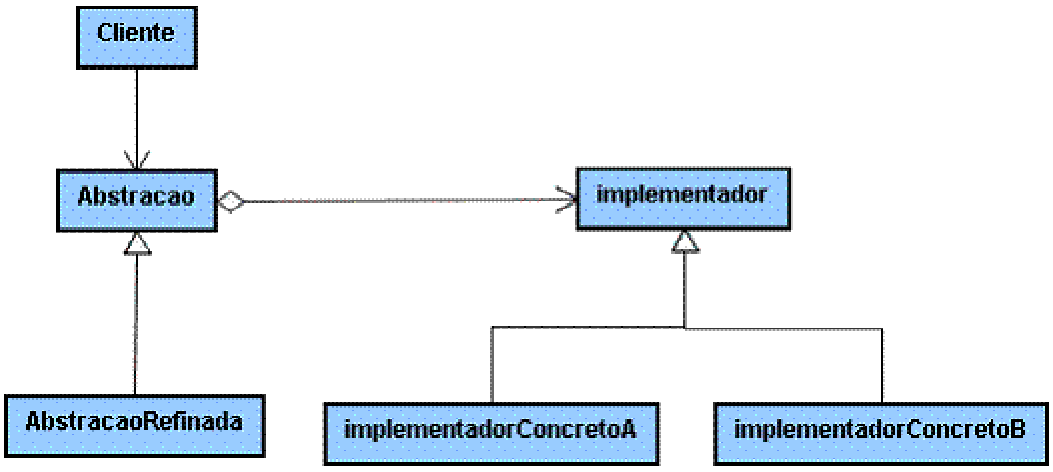


Figura 3.2 – Exemplo estrutural do padrão de projeto bridge.

3.2.3 Vantagens

Detalhes de implementação totalmente inacessíveis aos clientes.

Eliminação de dependências em tempo de compilação das implementações.

Implementação de abstração pode ser configurada em tempo de execução.

3.2.4 Implementação:

A implementação (adaptado [5]) do Padrão *Bridge* ilustra a Ponte entre a classe abstrata ObetoNegocios a ClienteDadosObjeto através de interface DadosObjeto. A interface DadosObjeto não é igual a ObjetoNegocios, já que não existe necessária dependência entre elas.

Este exemplo demonstra desacoplamento de uma abstração de ObjetoNegocio da implementação em DadosObjeto. As implementações de DadosObjeto podem evoluir dinamicamente sem propagar mudanças a nenhum dos objetos cliente, isso pode ser verificado na figura 3.2.1.

Na classe ClienteDadosObjeto encontra-se a implementação que a classe cliente espera através da DadosObjeto, neste caso, ela contém algumas pessoas cadastradas, ela pode adicionar novas pessoas em determinado grupo ou mostrá-las. Para interagir entre os cadastros utiliza-se o Padrão *Iterator,* que será detalhado nos próximos Padrões.

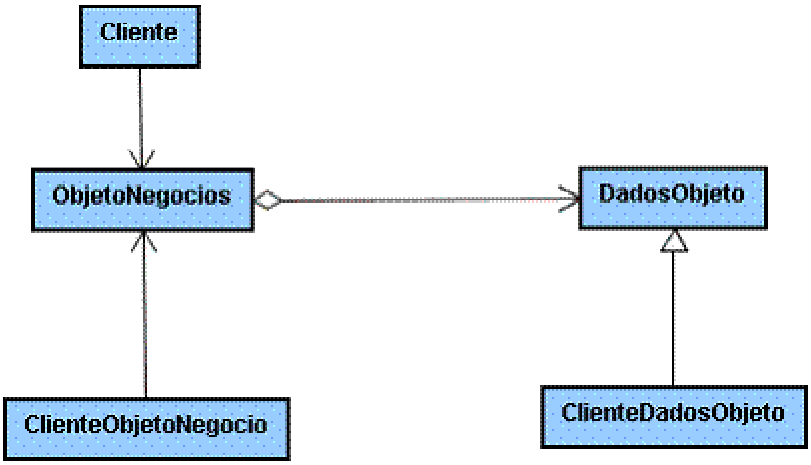


Figura 3.2.1 – Exemplo da Aplicação do Padrão de Projeto Bridge

3.3 Composite

Compõe objetos em estruturas do tipo árvore para representar hierarquias todo-parte.

Faz também com que o tratamento dos objetos individuais e de suas composições seja uniforme.

3.3.1 Motivação

Aplicações gráficas como editores de desenho e sistemas de captura de esquema deixam os usuários construírem diagramas complexos a partir de Componentes simples. O usuário pode agrupar Componentes para formar Componentes maiores, que, por sua vez, podem ser agrupados para formar Componentes maiores ainda. O Padrão *Composite* descreve como usar composição recursiva, de modo que os clientes não tenham que fazer distinção entre componentes simples e composições.

3.3.2 Aplicabilidade

Use o Padrão de Projeto Bridge quando:

Quiser representar hierarquias parte-todo de objetos;

Deseja que os clientes sejam capazes de ignorar as diferenças entre composição de objetos e objetos individuais. Clientes tratarão todos os objetos uniformemente na estrutura Composite.

3.3.3 Vantagens

Objetos complexos podem ser compostos de objetos mais simples recursivamente.

Facilita a adição de novos componentes.

3.3.4 Implementação

Componente:

Declara a interface para objetos na composição;

Implementa comportamento default para interface comum a todas as classes, como apropriado;

Declara uma interface para acessar ou gerenciar seus Componentes filhos;

Folha:

Representa objetos folhas na composição. Uma folha não tem filhos;

Define comportamento para objetos primitivos na composição.

Composição:

Define comportamento para Componentes que têm filhos;

Armazena Componentes filhos;

Implementa operações relacionadas com filhos na interface do Componente.

Cliente:

Manipula objetos na composição através da interface Componente

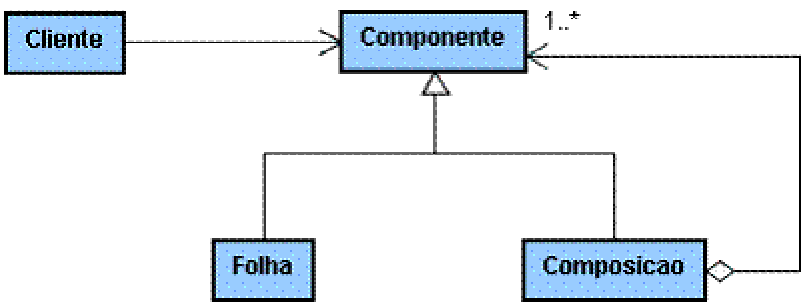


Figura 3.3 – Estrutura do Padrão de Projeto Composite

3.4 Decorator

Agregar responsabilidades adicionais a um objeto dinamicamente. Classes decoradoras oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade [5].

3.4.1 Motivação

Adicionar responsabilidades a um objeto, mas não à sua classe. Acontece, por exemplo, com criação de interfaces gráficas, quando se deseja acrescentar uma borda a um componente qualquer ou uma barra de rolagem a uma área de texto. Uma abordagem mais flexível é inserir o componente em outro objeto que adiciona a borda, um Decorator [5].

3.4.2 Aplicabilidade

Utilizado para adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, isto é, sem afetar outros objetos, da mesma forma, quando se quer retirar responsabilidades.

Quando a utilização de heranças para a implementação do mesmo afetará a flexibilidade do sistema [6].

Um caso de aplicação do Decorator é quando existem muitas variações de um objeto. Imagine por exemplo uma classe Janela com uma subclasse JanelaComBorda. Se houver a necessidade também de janelas com rolagem, tem-se ainda JanelaComRolagem e JanelaComBordaERolagem [6], além de outras possíveis combinações, já que poderia haver menus, botões ou barra de status.

Usando o Decorator, tem-se um número muito menor de subclasses: Janela, DecoratorJanela, DecoratorJanelaBorda, DecoratorJanelaRolagem, DecoratorJanelaMenu, e assim por diante [6].

3.4.3 Estrutura

Esse padrão providencia que cada objeto Decorator contenha outro objeto Decorator. Nesse aspecto, um decorator é como um pequeno composite cujos elementos possuem cada qual um filho único. Diferentemente do padrão Composite, cujo propósito é compor objetos agregados, o propósito do Decorator é compor comportamentos.

3.4.4 Implementação

Componente : define a interface para objetos que podem ter responsabilidades acrescentadas a eles dinamicamente.

ComponenteConcreto : define um objeto para o qual responsabilidades adicionais podem ser atribuídas

Decorator : mantém uma referência para um objeto Componente. Define uma interface que segue a interface de Componente.

DecoratorConcretoA e DecoratorConcretoB: acrescenta responsabilidades ao componente

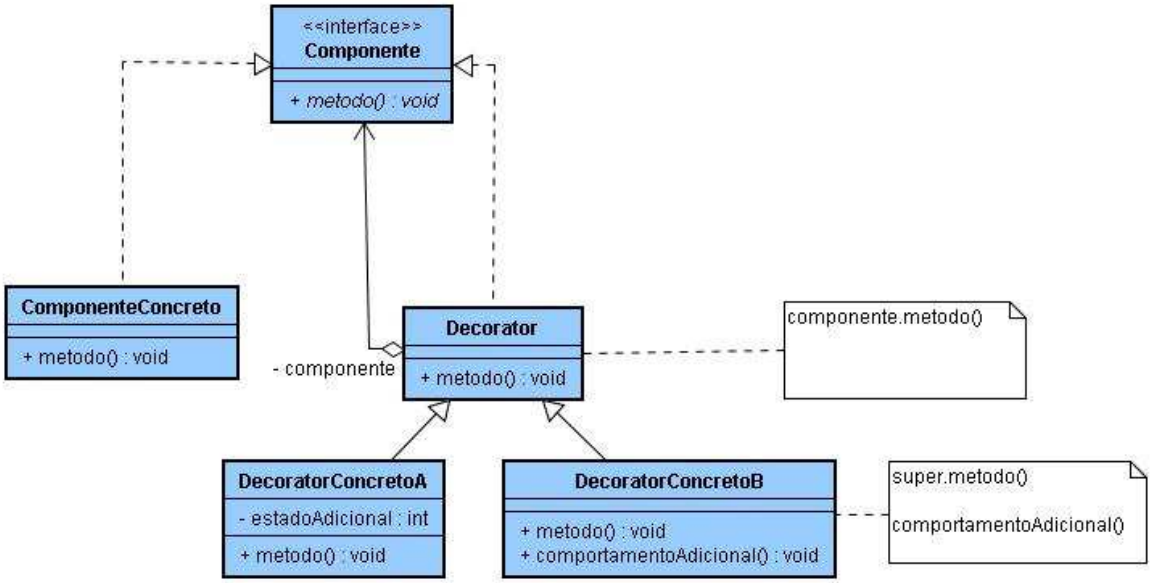


Figura 3.4 – Estrutura do padrão Decorator

3.4.5 Vantagens:

Fornece uma flexibilidade maior do que a herança estática.

Evita a necessidade de colocar classes sobrecarregadas de recursos em uma posição mais alta da hierarquia.

Simplifica a codificação permitindo que você desenvolva uma série de classes com funcionalidades específicas, em vez de codificar todo o comportamento no objeto.

Aprimora a extensibilidade do objeto, pois as alterações são feitas codificando novas classes.

3.4.5 Implementação

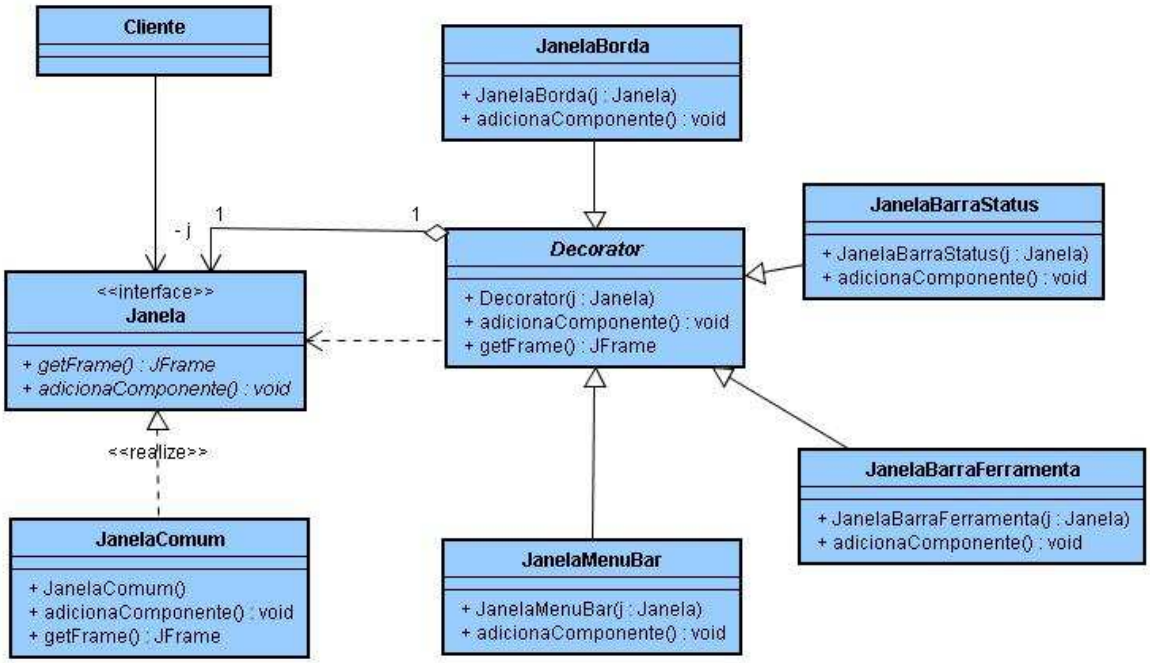


Figura 3.4.1 – exemplo de aplicação

O exemplo ilustrado na Figura 3.4.1, visa gerar janelas com características que podem variar de uma janela para outra. Por exemplo, o cliente pode decidir entre construir uma janela com barra de menus ou não. Desta forma consegue-se aumentar o número de possibilidades de janelas diferentes apresentadas com um pequeno número de classes.

Neste exemplo, a interface *Janela* recebe o papel de ser o *Componente* da estrutura apresentada anteriormente. Por sua vez, *JanelaComum* interpreta o *ComponenteConcreto*, ou seja, é ele que receberá as modificações no decorrer da execução da aplicação. Como o próprio nome diz, o *Decorator* assume a finalidade do *Decorator da estrutura padrão*. *JanelaMenuBar, JanelaBorda, JanelaBarraStatus e JanelaBarraFerramenta* herdam de Decorator, ou seja, são os n decoradores concretos.

3.5 Façade

Oferecer uma interface única para um conjunto de interfaces de um subsistema.

Definir uma interface de nível mais elevado que torna o subsistema mais fácil de usar [5].

Reduzir a complexidade do relacionamento entre uma classe relativa ao cliente e as demais classes utilitárias [5].

3.5.1 Motivação:

Uma grande vantagem da programação orientada a objetos é que ela ajuda a evitar que as aplicações se tornem programas monolíticos, com pedaços incorrigivelmente entrelaçados. No entanto, a aplicabilidade variada das classes em um subsistema orientado a objetos pode oferecer uma variedade expressiva de opções [7].

Existem circunstâncias onde é necessário utilizar diversas classes diferentes para que uma tarefa possa ser completada, caracterizando uma situação onde uma classe cliente necessita utilizar objetos de um conjunto específico de classes utilitárias que, em conjunto, compõem um subsistema particular ou que representam o acesso a diversos subsistemas distintos [7], como ilustrado na Figura abaixo.

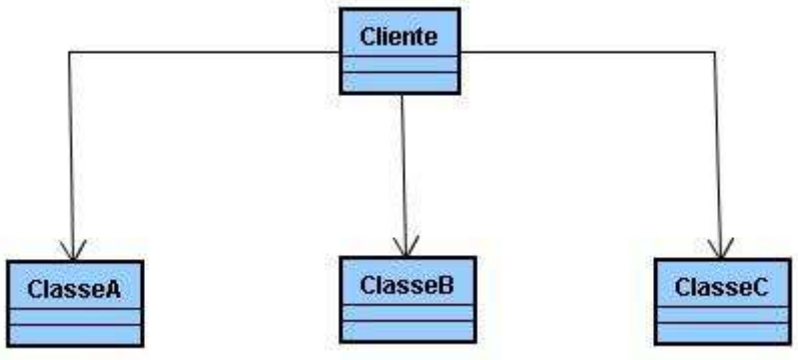


Figura 3.5 – exemplo de herança

3.5.2 Aplicabilidade

Criação de interfaces mais simples para um ou mais subsistemas complexos.

Redução de dependência entre o cliente e as classes existentes nos subsistemas, ocasionando a redução da coesão do sistema

Criação de sistemas em camadas. Este padrão provê o ponto de entrada para cada camada (nível) do subsistema.

3.5.3 Estrutura

A estrutura deste padrão, apresentado nas figuras Facade 3.5.1 e Facade 3.5.2, é bastante simples. A classe que constituirá o Facade deverá oferecer um conjunto de operações que sejam suficientes para que seus clientes possam utilizar adequadamente o subsistema, embora sem conhecer as interfaces de seus componentes [6].

O Facade é um padrão que pode apresentar infinidades de formas de representá-lo. O que importa para este padrão é que o Cliente apenas acesse objetos da classe Facade, esperando algum resultado vindo dela. A classe Facade é que terá a responsabilidade de entrar em contato com as diversas instâncias dentro deste sistema, efetuar possíveis cálculos vindos de classes abaixo dela e retornar as respostas que o cliente pediu.

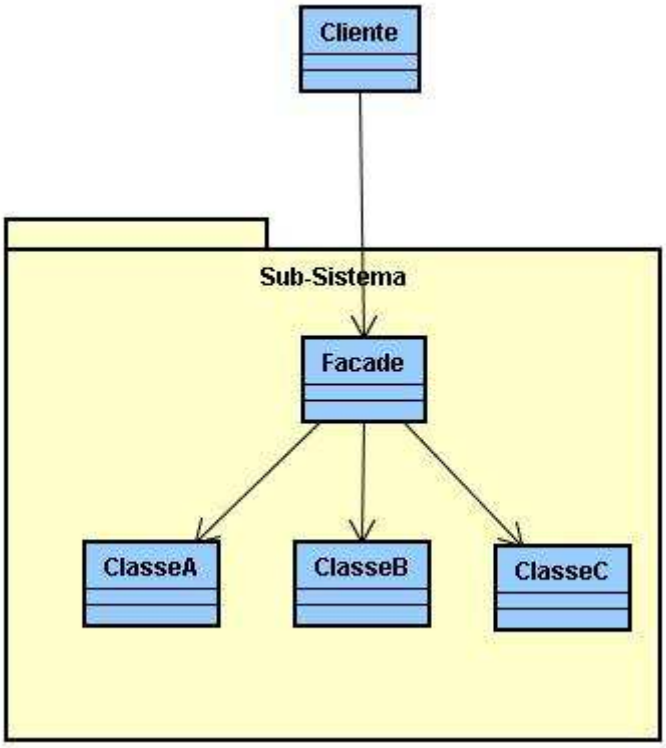


Figura facede 3.5.1 – exemplo de uso com interface

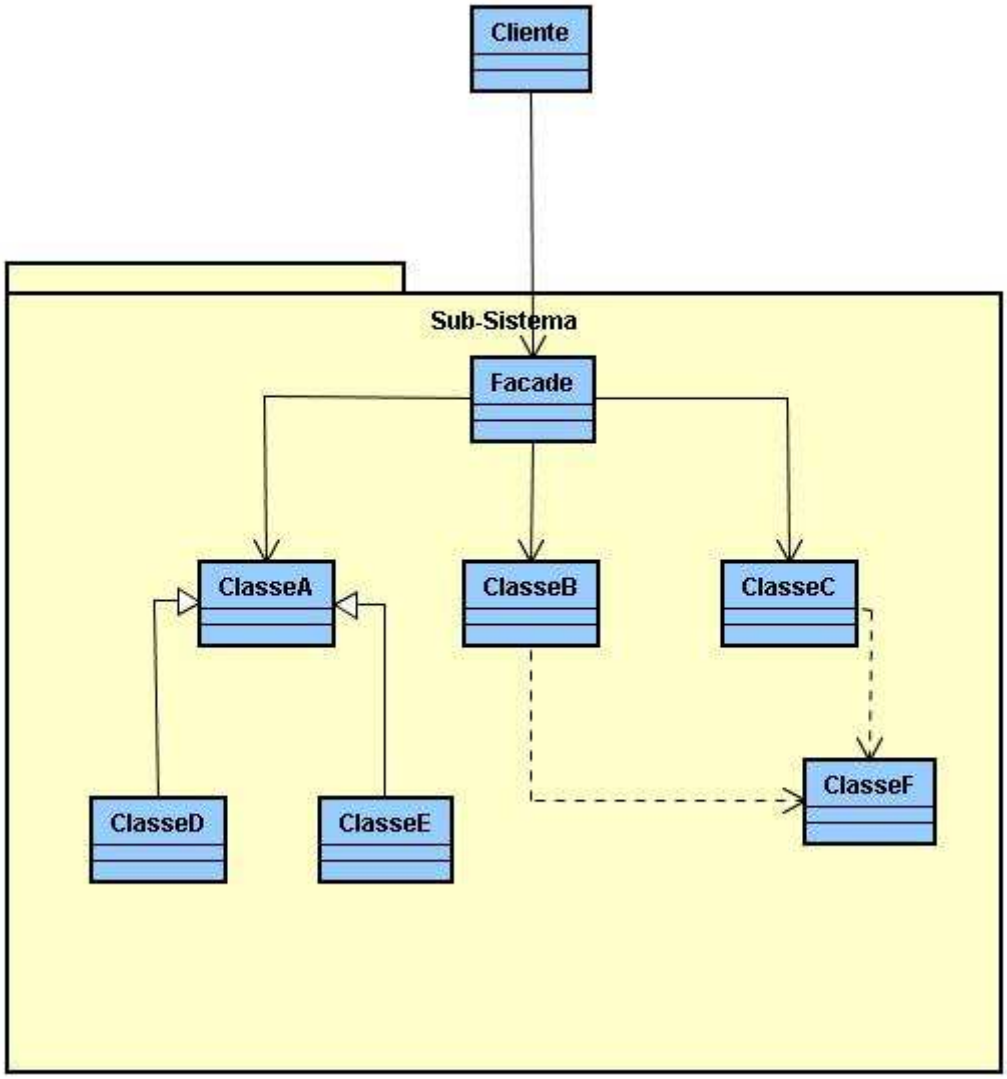


Figura facade 3.5.2 – exemplo de uso de interface com herança

3.5.4 Participantes

Cliente : aguarda respostas da interação do Facade e as Classes do subsistema.

Facade : conhece quais classes do subsistema seriam responsáveis pelo atendimento de uma solicitação e delega solicitações de clientes a objetos apropriados dos subsistemas.

Classes de subsistema : (ClasseA, ..., ClasseB)

Implementam as funcionalidades do subsistema.

Respondem a solicitações de serviços da Facade.

Não têm conhecimento da Facade.

3.5.5 Vantagens:

Protege os clientes da complexidade dos componentes do subsistema;

Promove acoplamento fraco entre o subsistema e seus clientes;

Reduz dependências de compilação, possivelmente complexas ou circulares;

Facilita a portabilidade do sistema [5];

Reduz a união entre subsistemas desde que cada subsistema utilize seu próprio padrão Facade e outras partes do sistema utilizem o padrão Facade para comunicar-se com o subsistema [5];

Não evita que aplicações possam acessar diretamente as subclasses do sistema, se assim o desejarem [5];

3.5.6 Implementação:

Implementar um Facade demanda definir um conjunto de operações reduzidas que permita ocultar a complexidade inerente à utilização de várias classes de um subsistema (adaptado [5]).

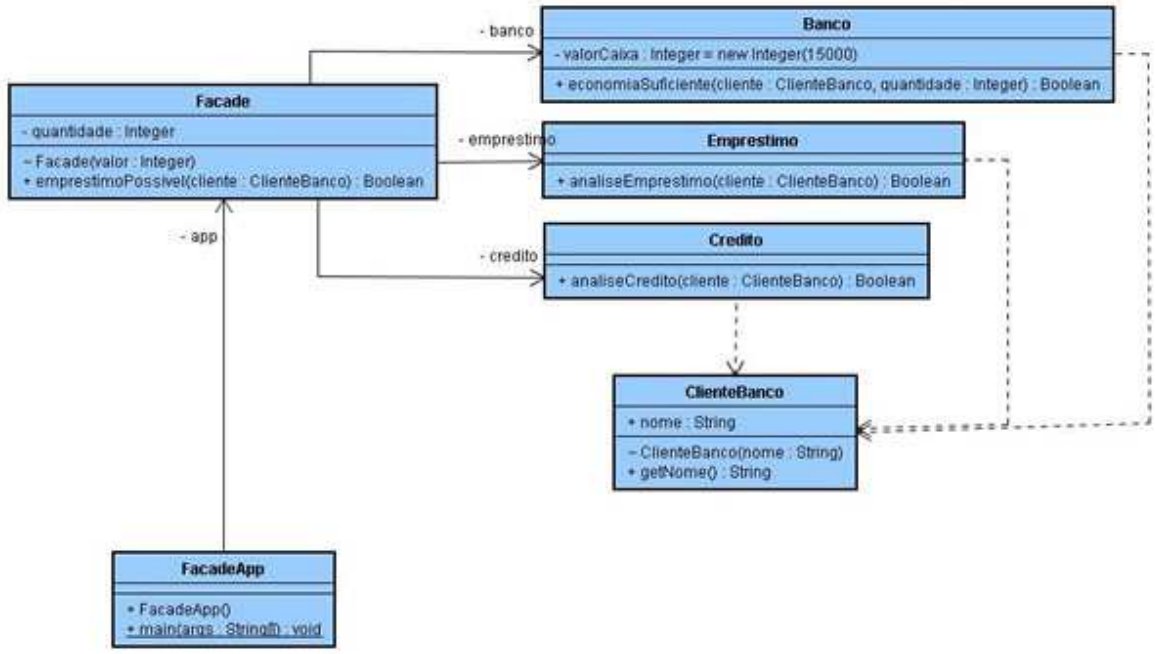


Figura 3.5.3 – exemplo de uso facade para casos complexos.

Neste exemplo, ilustrado na Figura 3.5.3, o cliente necessita consultar várias entidades a fim de saber se têm condições de receber um empréstimo. Para isso, normalmente, o cliente teria que conhecer toda a complexidade envolvida com as regras de concessão de empréstimos e aprende-las, no entanto, nada melhor do que deixar isso a cargo de quem já está no sistema, no caso o *Facade*, que irá se preocupar em colher dados das diferentes classes que podem decidir sobre a possibilidade do empréstimo e retornará esta informação pronta para o cliente.

FacadeApp é o cliente que busca informações que podem vir de vários de subsistemas, *Facade* é a classe responsável por acessar os subsistemas e trazer respostas de forma transparente a quem esteja acessando o *Facade*. *Banco*, *Emprestimo*, *Crédito* e *Consumidor* são classes pertencentes aos subsistemas.

3.6 Flyweight

3.6.1 Motivação:

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" [7].

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.

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.

3.6.2 Aplicabilidade:

A aplicação usa uma grande quantidade de objetos.

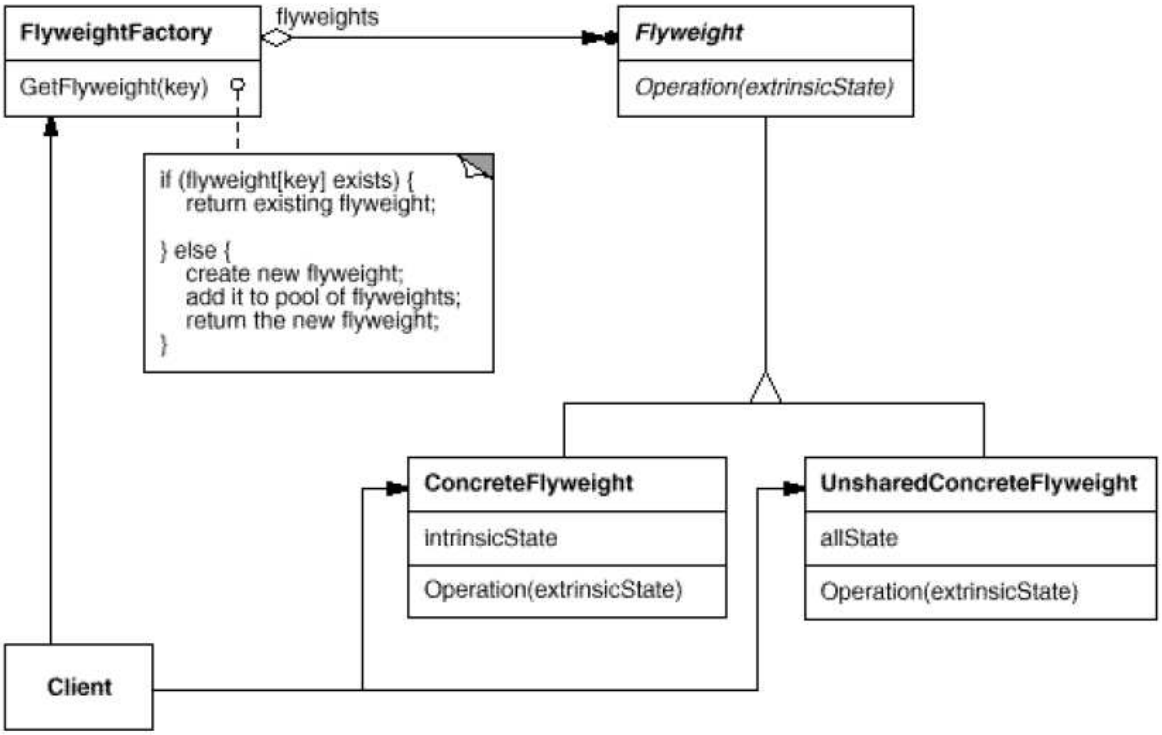
Custos de armazenamento são altos, por causa da imensa quantidade de objetos.

Parte considerável do estado do objeto pode ser tornar extrínseco uma vez que o estado extrínseco é removido, muitos agrupamentos de objetos podem ser substituídos por uma quantidade consideravelmente menor de objetos compartilhados.

A aplicação não depende de identidade de objetos. Visto que flyweights podem ser compartilhados, testes de identidade irão retornar true para objetos conceitualmente distintos.

3.6.3 Vantagens:

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. [8] “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.



3.6 – flyweights.

3.6.3 Implementação:

Participantes :

Flyweight :Declara uma interface através da qual os objetos flyweight podem receber seu estado extrínseco.

ConcreteFlyweight (Caracter) : Implementa a interface Flyweight e adiciona os atributos para guarder o estado extrínseco, se houver algum. Um objeto ConcreteFlyweight precisa ser compartilhavel. Qualquer estado que ele guarde precisa ser intrínseco, ou seja, precisa ser idependente do contexto do objeto ConcreteFlyweight.

UnsharedConcreteFlyweight (Linha, Coluna) : Nem todas as subclasses de Flyweight precisam ser compartilhadas. A interface Flyweight permite o compartilhamento, mas não obriga. É comum para a classe UnsharedConcreteFlyweight ter a classe ConcreteFlyweight como filha em algum nível da estrutura de classes do flyweight.

FlyweightFactory :Cria e gerencia objetos flyweight. Garante que os flyweights serão compartilhados adequadamente. Quando um flyweight é requerido, o objeto FlyweightFactory fornece uma instância existente ao invés de criar uma nova, se nenuma existir.

Client : Mantém referência para o(s) flyweight(s). Calcula ou guarda o estado extrínseco do(s) flyweight(s).

3.6.4 Conseqüências:

Flyweights podem introduzir custos em tempo de execução associados com transferir, buscar e ou computar o estado extrínseco. Apeasr disso, os custos são compensados pela economia de espaço, que cresce conforme mais objetos flyweights são compartilhados. A economia de espaço é função de vários fatores: a redução do número total de instâncias que vêm com o compartilhamento a quantidade de estados intrínsecos por objeto

3.7 Proxy

3.7.1 Motivação:

"O padrão Proxy provê uma referência para um objeto com o objetivo de controlar o acesso a este".[5]

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.

3.7.2 Aplicabilidade:

Remote Proxy - provê um representate local para um objeto em um espaço de endereçamento diferente.

Virtual Proxy - cria objeto sob demanda.

Protection Proxy - controla acesso ao objeto original.

Smart References - executa operações adicionais quando o objeto é acessado

Contagem de referências, carga de objetos persistentes, locks.

Copy-on-write - compartilhar grandes objetos, fazendo uma cópia apenas se necessário.

[9] enumera vantagens e desvantagens de padrão.

3.7.3 Vantagens:

Transparência, pois é utilizada a mesma sintaxe na comunicação entre o cliente e o objeto real;

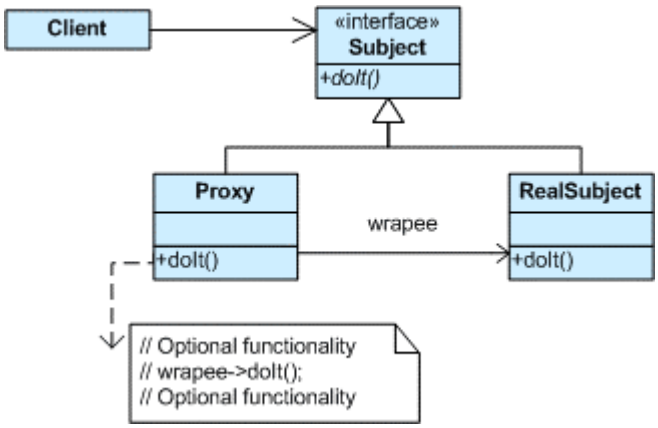
Tratamento inteligente dos dados no cliente e

Maior eficiência com caching no cliente.

3.7.4 Conseqüências:

Possível impacto na performance e Fatores externos como queda da rede pode deixar o Proxy inoperante.

3.7.5 Implementação:



3.7 – padrão proxy

Capítulo 6

Conclusão

Atualmente as metodologias ágeis, os padrões de projetos e orientação a objetos são temas amplamente discutidos e de muita relevância no contexto da tecnologia da informação.

Nunca antes na história da informática se usou e ousou tanto da informática quanto nesta era da web 2.0, empresas particulares, esferas de governo, interoperabilidade de sistemas, tudo isso tem preço e conseqüências, podendo ser o sucesso ou fracasso.

É neste ambiente heterogêneo e nem sempre com fronteiras claras que os padrões de projeto fazem diferença, o seu uso correto e a identificação precoce trazem claras vantagens e diversas melhorias, tanto para o desenvolvimento quanto para a equipe de gestão, uma vez que promove a neguentropia, pois sabemos que sem isto todo sistema tende a morte e quanto mais fácil ela for a manutenibilidade, mais vida útil terá o sistema.

Neste contexto de expansão da tecnologia da informação e dos serviços a que se propõe, fica claro que a maturidade dos arquitetos de sistema é imperativa, posto que são eles que definem os padrões de projeto.

Esta fase deve preceder todo o desenvolvimento de sistemas, o correto mapeamento dos padrões de projeto, são a causa de boa parte não só do sucesso dos projetos mas também de garantias adicionais como manutenções evolutivas de baixo custo, performance e escalabilidade das aplicações.

Bibliografia

[1] WIKIPEDIA, Padrão de projeto de software. Disponível em http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software#cite_note-0 , acessado em 10/11/2010.

[2] WIKIPEDIA, Padrão de projeto de software. Disponível em http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software#cite_note-2 , acessado em 11/02/2011.

[3] GAMMA, Erich, HELM, Richard; JOHNSON, Ralph, VLISSIDES, John. “Padrões de Projeto: soluções reutilizáveis de software orientado a objetos”. 1. ed. Porto Alegre: Bookman, 2000.

[4] Taylor, Lenore. "The Rudd gang of four", The Australian, Sydney, 09 November 2009. Retrieved on 2010-06-18.

[5] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad. Security Patterns: Integrating Security and Systems Engineering, Wiley Series in Software Design Patterns, 2005.

[6] CEFETPR, Padrões de projeto. Disponível em http://www.pg.cefetpr.br/coinf/simone/patterns/decorator.php, acessado em 11/02/2011.

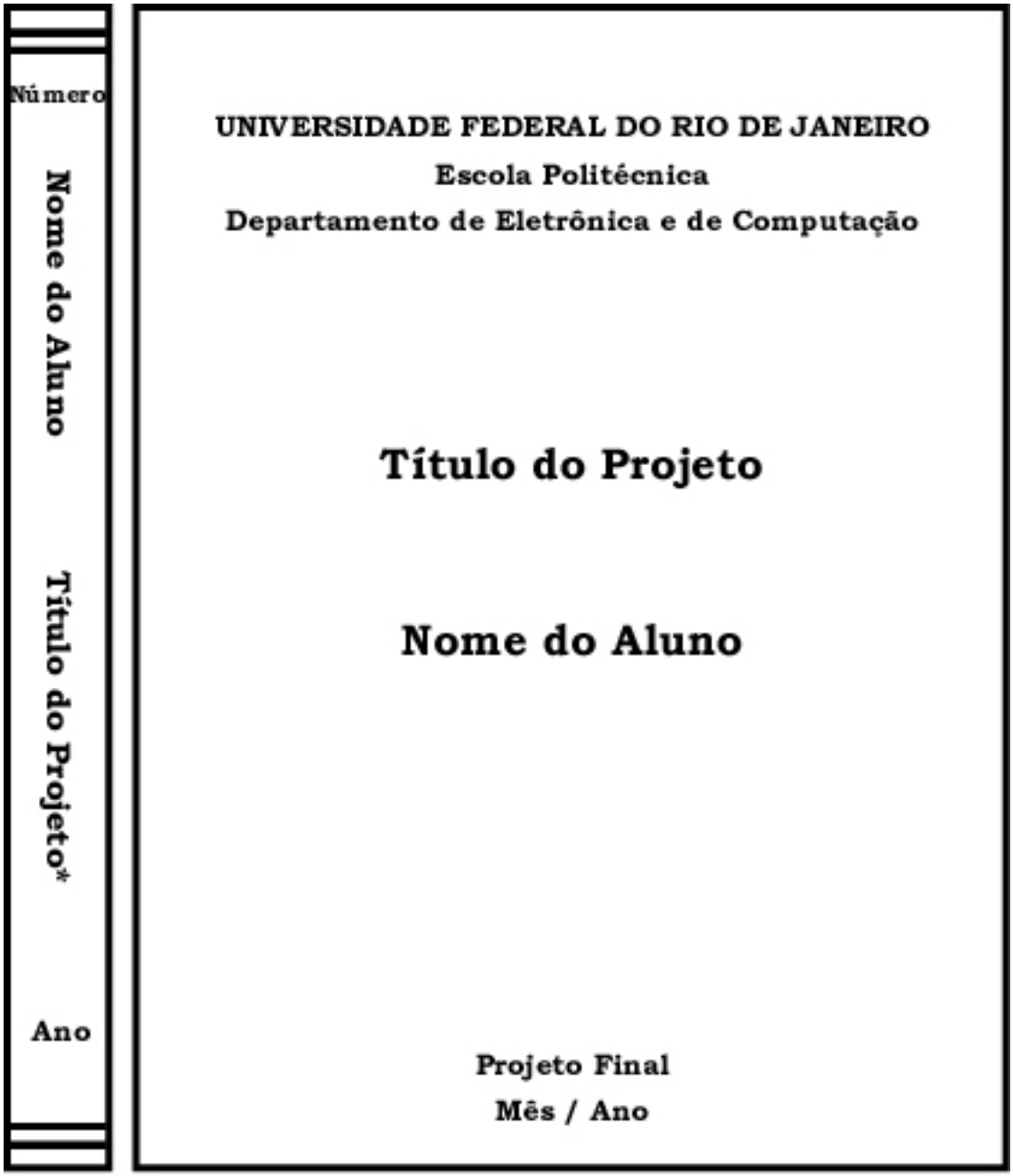
[7] METSKER, S. J. **Padrões de Projeto em Java**. Porto Alegre. Bookman, 2004

[8] Design Patterns Java Workbook, Metsker S., Addison-Wesley, 2002

[9] GFSOLUÇÕES, Padrão de projeto. Disponível em http://www.gfsolucoes.net/trabalhos/padroes_de_projetos.pdf , acessado em 10/03/2011.

B. Apêndice B

Encadernação do Projeto Final



* Título resumido caso necessário
Capa na cor preta, inscrições em dourado