UNIVAP – UNIVERSIDADE DO VALE DO PARAÍBA
FCC - Faculdade de Ciência da Computação
Curso de Ciência da Computação

PADRÕES DE PROJETO: DESIGN PATTERNS

por
Hugo Bonifácio Ribeiro
Tiago Abilio de Oliveira Gomes Serapião

Prof. Dr. Lineu Fernando Stege Mialaret
orientador

Jacareí - SP, Dezembro de 2004

Relatório do Trabalho de Graduação apresentado à Faculdade de Ciência da Computação da Universidade do Vale do Paraíba como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

PADRÕES DE PROJETO: DESIGN PATTERNS

Hugo Bonifácio Ribeiro
Tiago Abilio de Oliveira Gomes Serapião

Relatório aprovado em versão final pelos abaixo assinados:

_____

Prof. Dr. Lineu Fernando Stege Mialaret
Orientador acadêmico

_____

Prof. Dr. Lineu Fernando Stege Mialaret
Coordenador da Disciplina de TCC

Jacareí - SP, Dezembro de 2004

Padrões de Projeto: Design Patterns

Hugo Bonifácio Ribeiro
Tiago Abilio de Oliveira Gomes Serapião

Banca examinadora:

Prof. Dr. Lineu Fernando Stege Mialaret

Profº Dr. Luiz Alberto Vieira Dias

Profº MSc. José Walmir G. Duque

UNIVAP, Dezembro de 2004.

Sumário

Acrogramas, Abreviações

API	Application Programming Interface
GoF	Gang of Four
J2SE	Java 2 Standard Edition
MVC	Model/View/Controller
OMT	Object Modeling Technique
OOPSLA	Object-Oriented Programming, Systems, Languages, and Applications
SQL	Structured Query Language
UML	Unified Modeling Language

Glossário

acoplamento (coupling): Medida do grau de dependência, uns dos outros, dos componentes de um software.

acoplamento abstrato (abstract coupling): Dada uma classe A que mantém uma referência para uma classe abstrata B, diz-se que a classe A está acoplada abstratamente a B. Chamamos isso de acoplamento abstrato, porque A se refere a um tipo de objeto e não a um objeto concreto.

assinatura (signature): A assinatura de uma operação define o seu nome, seus parâmetros e seu valor de retorno.

classe (class): Uma classe define a interface e a implementação de um objeto. Ela especifica a representação interna do objeto e define as operações que o objeto pode executar.

classe abstrata (abstract class): Classe cuja finalidade primária é definir uma interface. Uma classe abstrata posterga parte ou toda a sua implementação para subclasses. Ela não pode ser instanciada.

classe concreta (concrete class): Classe que não tem operações abstratas. Ela pode ser instanciada.

classe mãe/ancestral (parent class): Classe da qual uma outra classe herda. Sinônimos: superclasse (superclass), classe base (base class) e classe pai ou classe mãe.

classe mixin (mixin class): Classe projetada para ser combinada com outras classes através do uso da herança. As classes mixin são geralmente abstratas.

composição de objetos (object composition): Montagem de objetos para obter um objeto composto com um comportamento mais complexo.

constructor: Uma operação que é automaticamente invocada para inicializar novas instâncias.

delegação (delegation): Mecanismo de implementação pelo qual um objeto repassa (forwards) ou delega uma solicitação para um outro objeto. O delegado executa a solicitação em lugar do objeto original.

destructor: Uma operação que é automaticamente invocada para finalizar um objeto que está para ser deletado.

diagrama de classe (class diagram): Diagrama que ilustra classes, suas estruturas internas e suas operações, e os relacionamentos estáticos entre elas.

diagrama de interação (interaction diagram): Diagrama que mostra o fluxo de solicitações (mensagens) entre objetos.

diagrama de objeto (object diagram): Diagrama que ilustra a estrutura em tempo de execução de um objeto específico.

encapsulamento (encapsulation): Resultado de ocultar uma representação e uma implementação em um objeto. A representação não é visível e não pode ser acessada diretamente a partir do exterior do objeto. As operações são a única forma de acessar e modificar a representação de um objeto.

framework: Conjunto de classes que cooperam entre si e compõem um projeto reutilizável para uma classe específica de software. Um framework fornece uma orientação talvez melhor encapulsante arquitetônica do software, através do particionamento do projeto em classes abstratas e da definição de suas responsabilidades e colaborações. Um desenvolvedor customiza o framework, para uma aplicação particular, através da especialização e da composição de instâncias de classes do mesmo.

friend class: Uma outra classe que tem os mesmos direitos de acesso às operações e dados de uma determinada classe.

herança (inheritance): Relacionamento que define uma entidade em termos de uma outra. A herança de classe define uma nova classe em termos de uma, ou mais, classe(s)mãe(s). A nova classe é chamada de subclasse ou de classe derivada. A herança de classes combina a herança de interface e a herança de implementação. A herança de interface define uma nova interface em termos de uma ou mais interfaces existentes. A

herança de implementação define uma nova implementação em termos de uma, ou mais, implementação(ões) existente(s).

herança privada (private inheritance): Uma classe herdada somente por causa de sua implementação.

interface (interface): Conjunto de todas as assinaturas definidas pelas operações de um objeto. A interface descreve o conjunto de solicitações as quais um objeto pode responder.

ligação dinâmica (dynamic binding): Associação, em tempo de execução, de uma solicitação a um objeto e uma de suas operações.

objeto (object): Entidade existente, em tempo de execução, que empacota tanto os dados como os procedimentos que operam sobre estes dados.

objeto agregado (aggregate Object): Objeto que é composto de sub-objetos. Os sub-objetos são chamados partes do agregado, e o agregado é responsável por eles.

operação (operation): Os dados de um objeto podem ser manipulados somente por suas operações. Um objeto executa uma operação quando ele recebe uma solicitação. Em Java, são chamadas de métodos.

operação abstrata (abstract operation): Operação que declara uma assinatura, mas não a implementa.

operação de classe (class operation): Uma operação dirigida para uma classe e não para um objeto individual.

padrão de projeto (design pattern): Um padrão de projeto sistematicamente nomeia, motiva e explica uma solução de projeto geral, que trata um problema recorrente de projeto em sistemas orientados a objetos. Ele descreve o problema, a solução, quando aplicar a solução e suas conseqüências. Também dá sugestões e exemplos de implementação. A solução é um arranjo genérico de objetos e classes que resolve o problema. A solução é customizada e implementada para resolver o problema em um contexto particular.

polimorfismo (polymorphism): Capacidade de substituir objetos com interfaces coincidentes por um outro objeto em tempo de execução.

receptor (receíver): Objeto alvo, ou destinatário, de uma solicitação.

redefinição (overriding): Redefinição de uma operação (herdada de uma classe ancestral) em uma subclasse.

referência a um objeto (object reference): Valor que identifica um outro objeto.

Relacionamento de um objeto agregado com suas partes. Uma classe define este relacionamento para as suas instâncias (ou seja, objetos agregados).

relacionamento de conhecimento ou associação (acquaintance relationship): Uma classe que se refere a uma outra tem um conhecimento daquela classe.

reutilização de caixa branca (white-box reuse): Estilo de reutilização baseado na herança de classe. Uma subclasse reutiliza a interface e a implementação da sua classe mãe, porém, ao assim fazê-lo, pode ter acesso a aspectos que, de outra maneira, seriam privativos (ou privados) da sua mãe.

reutilização de caixa preta (black-box reuse): Estilo de reutilização, baseado na composição de objetos. Os objetos compostos não revelam detalhes internos uns para os outros e, assim, são análogos a “caixas pretas”.

solicitação (request): Um objeto executa uma operação quando ele recebe uma solicitação correspondente de um outro objeto. Um sinônimo comum para a solicitação é mensagem.

subclasse (subclass): Classe que herda de uma outra classe.

subsistema (subsystem): Grupo independente de classes que colaboram para cumprir um conjunto de responsabilidades.

subtipo (subtype): Um tipo é um subtipo de outro se sua interface contém a interface do outro tipo.

supertipo (supertype): Tipo original do qual um tipo herda.

tipo (type): Nome de uma determinada interface.

tipo parametrizado (parameterized type): Tipo que deixa não-especificados alguns tipos constituintes. Os tipos não especificados são fornecidos como parâmetros no ponto de utilização. Tipos parametrizados são chamados de templates.

toolkit: Coleção de classes que fornece funcionalidades úteis, porém, não define o projeto de uma aplicação;

variável de instância (instance variable): Componente de dados que define parte da representação de um objeto.

Resumo

O mundo corporativo exige a cada dia das empresas produtoras de software que desenvolvam seus produtos de uma forma rápida, a um baixo custo e com muita qualidade. Para atender as necessidades de seus clientes, as empresas desenvolvedoras de sofware têm que utilizar tecnologias que pemitam aprimorar o seu processo de desenvolvimento. *Design Patterns* é uma delas, e por meio deste trabalho são apresentados seus conceitos, características e um estudo sobre seis dos vinte e três padrões mais conhecidos, bem como as vantagens de sua utilização.

Capítulo 1

Introdução

Atualmente não se concebe um processo de desenvolvimento de software sério sem a utilização da orientação a objetos, pois esta permite agregar qualidades importantes aos sistemas desenvolvidos sob seus paradigmas, como a extensibilidade e a reusabilidade [5]. Mas sua aplicação, por si só, não garante a obtenção destas qualidades, pois parece depender um pouco das linguagens e ferramentas empregadas nas fases de desenvolvimento e teste. Parece depender muito das técnicas usadas nas etapas de análise e definição destes sistemas. E parece depender ainda mais das concepções de seus projetistas. Como é usual que a experiência dos projetistas se mostre como fator preponderante no sucesso de qualquer projeto, é desejável compartilhar e transmitir o conhecimento inerente a estas experiências para outros profissionais, iniciantes ou não. Mas como fazer isso? Uma resposta para esta questão se encontra na utilização dos padrões de projeto.

Analisando muitos casos de desenvolvimento, é possível notar que vários dos problemas endereçados são compreendidos apenas superficialmente. Também é comum que a documentação relacionada tanto ao problema como a solução encontrada esteja incompleta ou ausente. Portanto, uma parcela do problema permanece sem análise, enquanto uma parte do conhecimento e experiência adquiridos nestes projetos fica retida apenas por seus participantes, dificultando seu compartilhamento. Desta forma, problemas idênticos que se repetem em outros contextos não são reconhecidos como tal, consumindo tempo e recursos para a definição de soluções que já haviam sido encontradas.

Os padrões de projeto, ou *design patterns*, vêm despertando interesse na comunidade de projetistas de software por proporcionar elementos que conduzem ao reaproveitamento de soluções para projetos, e não apenas à reutilização de código. Os padrões do projeto permitem evidenciar os aspectos essenciais de problemas comuns, levando a sua compreensão mais ampla e orientado a construção de sistemas que exibam efetivamente as qualidades desejada.

Desta forma este trabalho propõe o estudo e implementação de padrões de projeto, permitindo verificar as vantagens de sua utilização. Para implementação é utilizada a plataforma Java, pois esta se beneficia do uso intensivo dos padrões de projeto [4].

Estrutura do Documento

Este documento está organizado conforme descrito abaixo:

- Neste primeiro capítulo consta a história e as origens dos padrões de projeto.

- No capítulo 2 temos uma visão geral sobre o conceito de padrões de projeto e suas características.

- No capítulo 3 é apresentado a organização e classificação dos padrões de projeto segundo GoF, bem como as melhores técnicas utilizadas para selecionar e utilizar um padrão de projeto.

- Nos capítulos 4, 5 e 6 são descritos respectivamente as características dos padrões de criação, estruturais e comportamentais.

- No último capítulo temos a conclusão do trabalho, bem como a discussão sobre o impacto de padrões de projeto no desenvolvimento de sistemas.

História

A idéia de *pattern* surgiu através dos livros escritos pelo arquiteto Christopher Alexander que estuda assuntos relacionados ao planejamento urbano e à arquitetura de prédios. Em seus livros o termo *pattern* referencia estruturas que podem ser utilizadas para solucionar problemas, que ocorrem seguidamente na área onde o escritor atua. No período de 1977 - 1979, Alexander, como era conhecido, escreveu dois dos principais livros que deram origem, mais tarde, ao Design Pattern. São eles: *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977) e *The Timeless Way of Building* (Oxford University Press, 1979) [3]. Apesar destes livros tratarem sobre arquitetura e planejamento urbano as idéias que eles trazem sobre estes assuntos podem ser aplicadas em muitas outras áreas, inclusive no desenvolvimento de *software*.

Mesmo com todas estas idéias sobre *patterns* apresentadas por Alexander, não podemos desconsiderar que outras pessoas também estavam estudando sobre o assunto naquela época. A prática de documentar as melhores maneiras de se resolver um determinado problema era de preocupação, também, de muitos outros autores e profissionais. Don Knuth introduziu na época, por exemplo, idéias como uma programação instruída/dirigida[3].

A aplicação destas idéias apresentadas por Alexander, na área de desenvolvimento de *software,* foi de responsabilidade, principalmente, de duas pessoas: Kent Beck e Ward Cunninghan. No ano de 1987 Kent e Ward[3] estavam trabalhando com *Smalltalk* e projetos de interface para usuários na empresa Textronix quando eles decidiram aplicar algumas idéias apresentadas por Alexander, uma vez que estavam enfrentando problemas para terminar um projeto. Criaram então cinco *'design patterns'* para guiar novos programadores em *Smalltalk* nos projetos de interface gráfica. Isto permitia que os jovens programadores pudessem aproveitar o poder da linguagem e não cometessem falhas já conhecidas por programadores mais experientes. Os cinco *patterns* eram: *WindowPerTask, FewPanes, StandardPanes, NounsAndVerbs e ShortMenus.* Assim como Alexander dizia que os moradores dos prédios é que deveriam projetá-los, Ward e Kent resolveram pedir para que alguns usuários lhes ajudassem no projeto das interfaces. O resultado desta experiência foi muito interessante para Kent e Ward, levando-os a apresentar estes resultados em 1987 no evento *Object-Oriented Programming, Systems, Languages, and Applications* de 87 - OOPSLA'87 - em Orlando através do artigo intitulado *'Using Pattern Languages for Object-Oriented Programs'.*

As idéias sobre *patterns* foram evoluindo e cada vez mais pessoas as estudavam e participavam de discussões sobre o assunto. No final da década de 80, dois grandes

nomes, dentre outros, estavam trabalhando intensamente neste assunto: Jim Coplien e Erich Gamma. Coplien ou Cope, como era conhecido, estava fazendo o trabalho de catalogar *C++ Idioms* de forma que, mais tarde, em 1991 publicou o livro chamado *'Advanced C++ Programming Styles and Idioms'* para projetos orientados a objeto em C++. Nesta mesma época Erich estava ocupado com a sua tese de pós-doutorado refletindo e escrevendo sobre projetos orientados a objeto. Estas idéias de *patterns* para projetos de *software* orientados a objeto estavam se tornando populares entre a comunidade de orientação a objeto, porém a comunidade ainda estava carente de uma boa documentação destes *patterns* para poder utilizar.

Alguns encontros começaram a ser promovidos pela comunidade de orientação a objeto para que estas idéias fossem debatidas e disseminadas. Em 1990, no evento OOPSLA'90, Bruce Anderson apresentou um trabalho intitulado *'Toward an Architecture Handbook'* , o qual fomentou, durante o evento, uma discussão sobre *patterns* entre alguns participantes. Entre estes estavam Erich Gamma e Richard Helm. Foi a primeira vez que eles se encontraram e trocaram idéias sobre o assunto. Mas tarde, logo após este evento, Gamma e Helm se encontraram em Zurich, durante o verão, e iniciaram então um catálogo de *patterns* que mais tarde iriam chamar de *Design Patterns*. Neste encontro eles identificaram e documentaram muitos *patterns*, incluindo alguns familiares como: *Composite* e *Observer*.

No evento OOPSLA'91, após muitos avanços terem acontecido nesta área de *patterns*, estavam presentes, entre outros grandes nomes, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. A partir deste encontro eles começaram a discutir sobre *patterns* para mais tarde lançarem, juntos, o livro que se tornaria um marco nesta área. Foi então, no ano de 1994, no evento OOPSLA'94, que eles apresentaram o seu livro intitulado *'Design Patterns: Elements of Reusable Object-Oriented Software'*. No evento daquele ano eles venderam, aproximadamente, 750 cópias desse livro, surpreendendo até mesmo a editora Addison-Wesley que nunca havia vendido tantas cópias de um livro técnico em uma conferência. Os autores deste livro se intitularam *'Gang Of Four'* sendo muitas vezes conhecidos por este nome. Há muitos outros livros escrito por outros autores que também são bem conhecidos pela comunidade, como o livro *Pattern-Oriented Software Architecture: A System of Patterns* escrito por Frank Buschamann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal, conhecidos também por *'Gang Of Five'*.

Hoje em dia o livro '*Design Patterns: Elements of Reusable Object-Oriented Software*' é muito referenciado por programadores que utilizam linguagens orientadas a objetos, e se constitui em um dos livros fundamentais para uma boa programação, sendo que em 1998 estes autores ganharam o prêmio *Excellence in Programming Award* oferecido pelo *Dr Dobbs Journal*.

Capítulo 2

Padrões de Projeto

2.1 Visão Geral

Um padrão de projeto sistematicamente denomina, motiva e explica uma solução genérica de projeto, aplicável a um problema recorrente no projeto de sistemas orientados a objetos, descrevendo assim o próprio problema, a solução, as aplicações e conseqüências de sua adoção [3]. Esta definição enfatiza o entendimento dos aspectos fundamentais do problema, motivando o projeto na direção de uma solução genérica e reutilizável, pois cada padrão é responsável pela solução de um tipo de problema que ocorre repetidas vezes em nosso ambiente podendo ser aperfeiçoadas e usadas novamente por outros desenvolvedores em situações semelhantes.

O desafio é possibilitar a "transferência" de conhecimento dos projetistas mais experientes para os "novatos", acelerando seu amadurecimento profissional e amplificado as chances de criação de novas soluções. Projetistas experientes evitam a construção do início ao fim, procurando reutilizar as partes semelhantes identificadas em sistemas existentes, de modo que a cada ciclo de uso desta soluções sejam incorporadas novas características que as tornem melhor ou mais genéricas. Mas isso só pode ser feito por profissionais maduros, que já tenham passado por tais experiências, e desde que exista documentação apropriada.

Grande parte da metodologias de projeto enfatizam a solução em si ("o que" e "como" fazer), deixando de lado o "por que" da solução [1], fazendo que a documentação produzida não seja útil para compartilhar o conhecimento adquirido. Mais importante do que a própria solução é a descrição do problema e a aplicabilidade da solução, incluindo-se as sua limitações e conseqüências. Os padrões de projeto pretendem preencher estas lacunas, tornando-se um mecanismo para a comunicação e o compartilhamento de conhecimento entre desenvolvedores. Pretendem também constituir uma linguagem capaz de exprimir mais do que simples estruturas de dados, módulos ou objetos, mas fazer a articulação destes elementos em soluções arquitetônicas para o software [3]. Quando um projetista se familiariza com vários padrões de projeto, adquire uma parcela de experiência de seus desenvolvedores. E assim evita reinventar soluções existentes, permitindo concentrar esforços nos aspectos inéditos do problema e tornando os sistemas assim desenvolvidos provavelmente mais robustos e confiáveis.

2.2 O que é um padrão de projeto?

Christopher Alexander afirma: “cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”[7]. Muito embora Alexander estivesse falando acerca de padrões, em construções e cidades, o que ele diz é verdadeiro em relação aos padrões de projeto orientados a objeto. Nossas soluções são expressas em termos de objetos e interfaces em vez de paredes e portas, mas no cerne de ambos os tipos de padrões está a solução para um problema num contexto. Em geral, um padrão tem quatro elementos essenciais:

1. O **nome do padrão** é uma referência que podemos usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras. Dar nome a um padrão aumenta imediatamente o nosso vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões nos permite conversar sobre eles com nossos colegas, em nossa documentação e até com nós mesmos. O nome torna mais fácil pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas. Encontrar bons nomes é uma das partes mais duras do desenvolvimento de um catálogo de padrões.

2. O **problema** descreve quando aplicar o padrão. Ele explica o problema e seu contexto. Pode descrever problemas de projeto específicos, tais como representar algoritmos como objetos. Pode descrever estruturas de classes ou objeto sintomáticos de um projeto inflexível. Algumas vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão.

3. A **solução** descreve os elementos que compõem o projeto, seu relacionamento suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e de como um arranjo geral de elementos (classes e objetos, no nosso caso) resolve o mesmo.

4. As **conseqüências** são os resultados e análises das vantagens e desvantagens (trades-offs) da aplicação do padrão. Embora as conseqüências sejam

> raramente mencionadas quando descrevemos decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. As conseqüências para o software freqüentemente envolvem compromissos de espaço e tempo. Elas também podem abordar aspectos sobre linguagens e implementação. Uma vez que a reutilização é freqüentemente um fator no projeto orientado a objetos as conseqüências de um padrão incluem o seu impacto sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema. Relacionar estas conseqüências explicitamente ajuda a compreendê-las e avaliá-las.

Padrões de projeto não são projetos tais como listas encadeadas e tabelas de acesso aleatório que podem ser codificadas em classes e ser reutilizadas tais como estão. Nem projetos complexos, de domínio específico, para uma aplicação inteira ou subsistema. Os padrões de projeto são descrições de objetos e classes comunicantes que são customizadas para resolver um problema geral de projeto num contexto particular.

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. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve quando pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as conseqüências, custos e benefícios de sua utilização.

Embora padrões de projeto descrevam projetos orientados a objeto, baseiam-se em soluções reais que foram implementadas em linguagens de programação orientadas a objeto principais, como Smalltalk, C++ e Java, ao invés de implementações em linguagens procedurais (Pascal, C, Ada) ou linguagens orientadas a objetos mais dinâmicas (CLOS, Dylan, Self).

A escolha da linguagem de programação é importante porque influencia o ponto de vista do usuário: nossos padrões assumem recursos de linguagem do nível do Java, e essa escolha determina o que pode ou não pode ser implementada facilmente. Se tivéssemos assumido o uso de linguagens procedurais, nos deveríamos ter incluído padrões de projetos como “Herança”, “Encapsulamento” e “Polimorfismo”. De maneira semelhante, alguns dos nossos padrões são suportados diretamente por linguagens orientadas a objetos menos comuns. Por exemplo, CLOS tem alguns métodos que diminui a necessidade de um padrão como Visitor.

2.3 Características

Embora um padrão de projeto seja a descrição de um problema, de uma solução genérica e sua justificativa, isso não significa que qualquer solução conhecida para um problema constitua um padrão, pois existem características que sempre devem ser atendidas pelos padrões [2][3]:

- descrever e justificar a solução para um problema concreto e bem definido;
- a solução descrita deve ser comprovada previamente;
- descrever relações entre conceitos, mecanismos e estruturas existentes nos sistemas;
- O problema tratado deve ocorrer em diferentes contextos, ou seja, se a solução não tem aplicação em diferentes situações então não constitui um padrão.

Percebemos que os padrões de projeto sustentam uma gama ampla de conceitos: permitem exprimir adequadamente concepções e estruturas que não são necessariamente objetos; capturam a evolução e aprimoramento das soluções, equilibrando seus pontos fortes e fracos; reúnem experiência em torno da solução de um problema comum, e usualmente não constituem soluções triviais. Também deve ser possível sua utilização independente ou em conjunto com vários outros padrões, compondo assim linguagens de padrões.

2.4 Descrevendo Padrões

Os padrões de projeto podem constituir um catálogo de soluções testadas, aprovadas e reutilizáveis, definindo um vocabulário especializado onde cada padrão é associado aos conceitos relativos a um tipo de problema e sua solução. Assim é necessário estruturar sua documentação, sem se prender as linguagens de programação ou contextos particulares, sendo freqüentemente sua descrição textual composta por um roteiro padronizado conhecido como forma. Existem diversas formas que geralmente incluem o nome do padrão de projeto, propósito, problema, contexto, dificuldades, limitações ou restrições, solução, resultados, diagramas e exemplos [4]. O nome do padrão é um elemento importante, pois será adotado pelos desenvolvedores como uma

referência descritiva do problema, sua solução, características e conseqüências. As descrições associadas ao propósito, motivação, aplicabilidade, devem ser valorizadas, pois contêm o problema, racional da solução, recomendações e decorrências de seu uso. A estrutura dos padrões, seus participantes e inter-relacionamentos podem ser descritas por meio de diagramas UML. Também é útil listar outros padrões relacionados e situações reais de aplicação.

Mas as notações gráficas, embora sejam importantes e úteis, não são suficientes. Elas simplesmente capturam o produto final do processo de projeto como relacionamentos entre classes e objetos. Para reutilizar o projeto, nós também devemos registrar decisões, alternativas e análises de custos e benefícios que levaram a ele. Também são importantes exemplos concretos, porque ajudam a ver o projeto em ação.

Nós descrevemos padrões de projeto usando um formato consistente. Cada padrão é dividido em seções conforme descrito a seguir. Desta forma, podemos obter uma estrutura uniforme às informações, tornando os padrões de projeto mais fáceis de aprender, comparar e usar.

Nome e classificação do padrão: O nome do padrão expressa a sua própria essência de forma sucinta. Um bom nome é vital, porque ele se tornará parte do seu vocabulário de projeto. A classificação do padrão reflete o esquema que introduziremos no capítulo “Organizando o catálogo”.

Intenção e objetivo: É uma curta declaração que responde às seguintes questões: o que faz o padrão de projeto? Quais os seus princípios e sua intenção? Que tópico ou problema particular de projeto ele trata?

Também conhecido como: Outros nomes bem conhecidos para o padrão, se existirem.

Motivação: Um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão solucionam o problema. O cenário ajudará a compreender as descrições abstratas que o padrão vem a seguir.

Aplicabilidade: Quais são as situações nas quais o padrão de projeto pode ser aplicado? Que exemplos de maus projeto ele pode tratar? Como você pode reconhecer essas situações?

Estrutura: Uma representação gráfica das classes do padrão usando uma notação baseada na Object Modeling Technique (OMT). Nós também usamos diagramas de interação para ilustrar seqüências de solicitações e colaborações entre objetos.

Participantes: As classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.

Colaborações: Como os participantes colaboram para executar suas responsabilidades.

Conseqüências: Como o padrão suporta a reutilização de seus objetivos? Quais são os seus custos e benefícios e os resultados da sua utilização? Que aspecto da estrutura do sistema ele permite variar independentemente?

Implementação: Que armadilhas, sugestões ou técnicas você precisa conhecer quando da implementação do padrão? Existem considerações específicas de linguagem?

Exemplo de código: Fragmentos ou blocos de código que ilustram como você pode implementar.

Usos conhecidos: Exemplos do padrão encontradas em sistemas reais.

Padrões relacionados: Que padrões de projeto estão intimamente relacionados com este? Quais são as diferenças importantes? Com quais outros padrões este deveria ser usado?

Capítulo 3

Classificação

3.1 Catálogo de Padrões de Projeto

O catálogo contém 23 padrões de projeto. Seus nomes e intenções são listados a seguir para dar uma visão geral.

Abstract Factory: Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Adapter: Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis.

Bridge: Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.

Builder: Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes reapresentações.

Chain of Responsibility: Evita acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.

Command: Encapsula uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre (log) solicitações e suporte operações que podem ser desfeitas.

Composite: Compõe objetos em estrutura de árvore para representar hierarquias do tipo parte-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.

Decorator: Atribui responsabilidades adicionais a um objeto dinamicamente. Os decorators fornece uma alternativa flexível a subclasses para extensão da funcionalidade.

Façade: Fornece uma interface unificada para unir conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar.

Factory Method: Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. O Factory Method permite a uma classe postergar (defere) a instanciação às subclasses.

Flyweight: Usa compartilhamento para suportar grandes quantidades de objetos refinados de maneira eficiente.

Interpreter: Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem.

Iterator: Fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

Mediator: Define um objeto que encapsula como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.

Memento: Sem violar a encapsulação, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.

Observer: Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.

Prototype: Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando este protótipo.

Proxy: Fornece um objeto representante, ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Singleton: Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

State: Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.

Strategy: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O Strategy permite que o algoritmo varie independente dos clientes que o utilizam.

Template Method: Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O Template Method permite que as subclasse redefinam certos passos de um algoritmo sem mudar sua estrutura.

Visitor: Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. Visitor permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

3.2 Organizando o catálogo

Os padrões de projeto variam na sua granularidade e no seu nível de abstração. Por existirem muitos padrões de projeto, necessitamos de uma maneira de organizá-los. Logo, esta seção classifica os padrões de projeto de uma maneira que possamos nos referir a famílias de padrões relacionados. A classificação ajuda a aprender os padrões mais rapidamente, bem como direcionar esforços na descoberta de novos.

Nós classificamos os padrões de projetos por dois critérios, conforme Tabela 3.1. O primeiro critério, chamado finalidade, reflete o que um padrão faz. Os padrões podem ter finalidade de criação, estruturamento ou comportamento. Os padrões de criação se preocupam com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades.

O segundo critério, chamado escopo, especifica se o padrão se aplica primeiramente a classes ou a objetos. Os padrões para classes lidam com relacionamentos entre classes e subclasses. Estes relacionamentos são estabelecidos através do mecanismo de herança, assim eles são estáticos - fixados em tempo de compilação. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são mais dinâmicas. Quase todos utilizam a herança em certa medida. Note que a maioria está no escopo de Objeto.

Os padrões de criação voltados para classes deixam alguma parte da criação de objetos para subclasses, enquanto que o padrão de criação voltado para objetos postergam esse processo para outro objeto. Os padrões estruturais voltados para classes

utilizam a herança para compor classes, enquanto que os padrões estruturais voltados para objetos descrevem maneiras de montar objetos. Os padrões comportamentais voltados para classes usam a herança para descrever algoritmos e fluxo de controle, enquanto que os voltados para objetos descrevem como um grupo de objetos coopera para executar uma tarefa que um único objeto não pode executar sozinho.

		Propósito		
		Criação	**Estrutura**	**Comportamento**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 3.1 – Escopo dos padrões de projeto

Há outras maneiras de organizar os padrões. Alguns padrões são freqüentemente usados em conjunto. Por exemplo, o Composite é freqüentemente usado com o Iterator ou o Visitor. Alguns padrões são alternativos: o Prototype é freqüentemente uma alternativa para o Abstract Factory. Alguns padrões resultam em projetos semelhantes, embora tenham intenções diferentes. Por exemplo, os diagramas de estrutura de Composite e Decorador são semelhantes.

Uma outra maneira, ainda, de organizar padrões de projeto é de acordo com a forma que eles mencionam outros padrões nas seções “Padrões Relacionados”. A figura 3.1 ilustra esse relacionamento graficamente.

Existem, claramente, muitas maneiras de organizar os padrões de projeto. Ter múltiplas maneiras de pensar a respeito deles aprofundará sua percepção sobre o que fazem, como se comparam e quando aplicá-los.

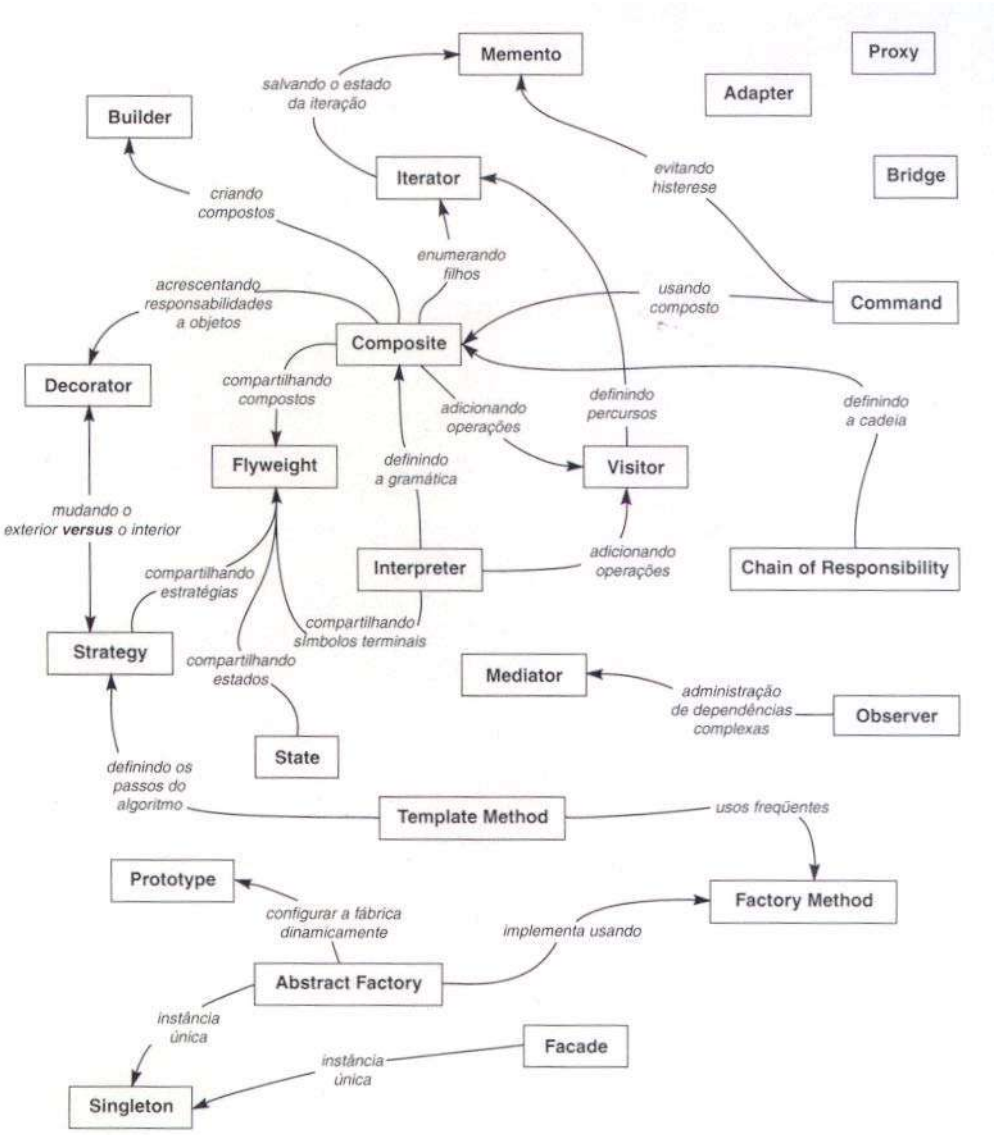


Figura 3.1 – Relacionamentos entre padrões de projeto

3.3 Como selecionar um padrão de projeto

Com mais de 20 padrões de projeto no catálogo para se escolher, pode ser difícil encontrar aquele que trata um problema de projeto particular, especialmente se o catálogo é novo e estranho para você. Aqui apresentamos diversas abordagens para encontrar o padrão de projeto correto para o seu problema:

- Considere como padrões solucionam problemas de projeto. Analise como padrões de projeto ajudam a encontrar objetos apropriados, determinar a granularidade deles, especificar interfaces e várias outras formas pelas quais padrões de projeto solucionam problemas de projeto;

- Verifique a intenção de cada padrão do catálogo. Leia atentamente a intenção de cada padrão para encontrar um ou mais padrões que parecem relevantes para o seu problema;

- Estude como os padrões se interrelacionam. O estudo desses relacionamentos pode ajudar a direcioná-lo para o padrão, ou grupo de padrões adequado;

- Estude padrões de finalidades semelhantes, observando as semelhanças e diferenças de padrões com propósitos similares;

- Examine uma causa de reformulação do projeto;

- Verifique o que deve ser variável no seu projeto. Esta abordagem é o oposto de se focalizar nas causas de reformulação. Ao invés de considerar o que pode forçar uma mudança em um projeto, considere o que você quer ser capaz de mudar sem reprojetá-lo. O foco, aqui, é posto sobre o encapsulamento do conceito que varia, um tema de muitos padrões de projeto. A tabela 3.2 lista o(s) aspecto(s) do projeto que padrões de projeto lhe permitem variar independentemente. Desta forma, eles podem ser mudados sem necessidade de reformulação de projeto.

Propósito	Padrão	Aspecto(s) que pode(m) variar
De Criação	Abstract Factory	famílias de objetos-produto.
	Builder	como um objeto composto é criado.
	Factory Method	subclasse de um objeto que é instanciada.
	Prototype	classe de um objeto que é instanciada.
	Singleton	a única instância de uma classe.
Estruturais	Adapter	Interface para um objeto.
	Bridge	Implementação de um objeto.
	Composite	estrutura e composição de um objeto.
	Decorator	responsabilidade de um objeto sem usar subclasses.
	Façade	interface para um subsistema.
	Flyweight	custos de armazenamento de objetos.
	Proxy	como um objeto é acessado; sua localização.
Comportamentais	Chain of Responsibility	objeto que pode atender a uma solicitação.
	Command	quando e como uma solicitação é atendida.
	Interpreter	gramática e interpretação de uma linguagem.
	Iterator	como os elementos de um agregado são acessados, percorridos.
	Mediator	como e quais objetos interagem uns com os outros.
	Memento	que informação privada é armazenada fora de um objeto e quando.
	Observer	número de objetos que dependem de um outro objeto; como os objetos dependentes se mantêm atualizados.
	State	estados de um objeto.
	Strategy	um algoritmo.
	Template Method	passos de um algoritmo.
	Visitor	operações que podem ser aplicadas a (um) objeto(s) sem mudar sua(s) classe(s).

Tabela 3.2 - Aspectos do projeto que o uso de padrões permite variar

3.4 Como usar um padrão de projeto

Uma vez que escolheu um padrão de projeto, como você o utiliza? Apresentamos aqui uma abordagem passo a passo para aplicar um padrão de projeto efetivamente:

1. Leia o padrão por inteiro uma vez, para obter sua visão geral. Preste atenção na aplicabilidade e conseqüências, para assegurar-se de que o padrão é correto para o seu problema;

2. Volte e estude a estrutura, particularidades e colaborações. Assegure-se de que compreende as classes e objetos no padrão e seus relacionamentos;

3. Veja um exemplo completo do código especificado. O estudo do código ajuda a aprender como implementar o padrão;

4. Escolha nomes para os participantes do padrão que tenham sentido no contexto da aplicação. Os nomes para os participantes dos padrões de projeto são, geralmente, muito abstratos para aparecerem diretamente numa aplicação. No entanto, é útil incorporar o nome do participante no nome que aparecerá na aplicação. Isso ajuda a tornar o padrão mais explícito na implementação. Por exemplo, se você usa o padrão Strategy para um algoritmo de composição de textos, então poderá ter classes como SimpleLayoutStrategy ou TextLayoutStrategy;

5. Defina as classes. Declare suas interfaces, estabeleça os seus relacionamentos de herança e defina as variáveis de instância que representam dados e referências a objetos. Identifique as classes existentes na sua aplicação que serão afetadas pelo padrão e modifique-as de acordo;

6. Defina nomes específicos da aplicação para as operações no padrão. Aqui novamente, os nomes em geral dependem da aplicação. Use as responsabilidades e colaborações associadas com cada operação como guia. Seja consistente, também, nas suas convenções de nomenclatura. Por exemplo, você pode usar consistentemente o prefixo “Criar“ para denotar um método fábrica (FactoryMethod);

7. Implemente as operações para suportar as responsabilidades e colaborações presentes no padrão.

Estas são apenas diretrizes para iniciá-lo nesta técnica. Ao longo do tempo você desenvolverá sua própria maneira de trabalhar com padrões de projeto.

Nenhuma discussão sobre como usar padrões de projeto estaria completa sem algumas palavras sobre como não usá-los. Os padrões de projeto não devem ser aplicados indiscriminadamente. Freqüentemente eles obtêm flexibilidade e variabilidade

pela introdução de níveis adicionais de endereçamento indireto, e isso pode complicar um projeto e/ou custar em termos de desempenho. Um padrão de projeto deverá apenas ser aplicado quando a flexibilidade que ele oferece é realmente necessária. As seções conseqüências são muito úteis quando avaliamos os custos e benefícios de um padrão.

Capítulo 4

Padrões de Criação

Os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto.

Os padrões de criação se tornam importantes à medida que os sistemas evoluem no sentido de depender mais da composição de objetos do que da herança de classes. Quando isso acontece, a ênfase se desloca da codificação de maneira rígida de um conjunto fixo de comportamentos para a definição de um conjunto menor de comportamentos fundamentais que podem ser compostos em qualquer número para definir comportamentos mais complexos. Assim, criar objetos com comportamentos particulares exige mais do que simplesmente instanciar uma classe.

Há dois temas recorrentes nesses padrões. Primeiro, todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo, ocultam o modo como as instâncias destas classes são criadas e juntadas. Tudo o que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Conseqüentemente, os padrões de criação dão muita flexibilidade no que é criado, quem cria, como e quando é criado. Eles permitem configurar um sistema com objetos "produto" que variam amplamente em estrutura e funcionalidade. A configuração pode ser estática (isto é, especificada em tempo de compilação) ou dinâmica (em tempo de execução).

Algumas vezes, os padrões de criação competem entre si. Por exemplo, há casos em que tanto o Prototype como o Abstract Factory podem ser usados proveitosamente. Em outras ocasiões, eles são complementares: Builder pode usar um dos outros padrões para implementar quais componentes são construídos. Prototype pode usar Singleton na sua implementação.

4.1 Singleton

Intenção

Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma.

Motivação

O que motiva sua existência é que muitas aplicações necessitam garantir a ocorrência de uma única instância de classes específicas, pois tais objetos podem fazer uso de recursos cuja utilização deve ser exclusiva, ou porque desejamos que os demais elementos do sistema compartilhem um único objeto particular. Estas situações ocorrem quando vários subsistemas utilizam um único arquivo de configuração, permitindo sua modificação concorrente. Ou quando só devemos estabelecer uma conexão exclusiva com um banco de dados ou um sistema remoto, devendo ser compartilhado por várias tarefas paralelas, entre muitas outras. Ao mesmo tempo não queremos que os usuários destas classes zelem por esta condição de unicidade, mas desejamos oferecer acesso simples à instância única que deverá existir, portanto adicionaremos estas responsabilidades às classes que deverão constituir um Singleton.

Mas, como garantimos que uma classe tenha somente uma instância e que essa instância seja facilmente acessível? Uma variável global torna um objeto acessível, mas não impede você de instanciar múltiplos objetos.

Uma solução melhor seria tornar a própria classe responsável por manter o controle de sua única instância. A classe pode garantir que nenhuma outra instância seja criada (pela interceptação das solicitações para criação de novos objetos), bem como pode fornecer um meio para acessar sua única instância. Este é o padrão Singleton.

Aplicabilidade

Use o padrão Singleton quando:

- deve haver apenas uma instância de uma classe, e essa instância deve dar acesso aos clientes através de um ponto bem conhecido;

- quando a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código.

Estrutura

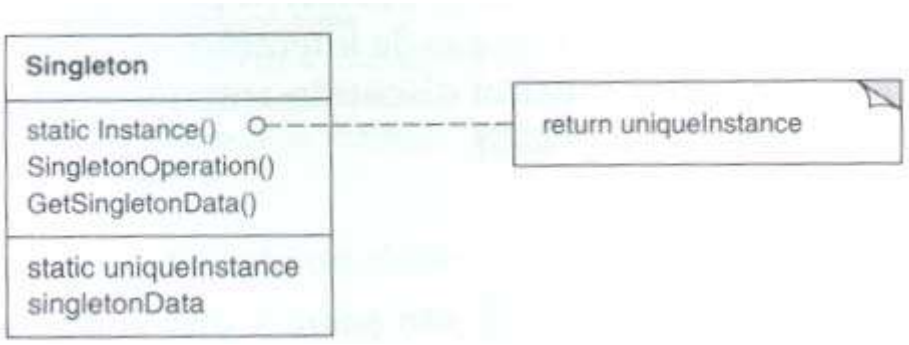


Figura 4.1 – Estrutura do padrão Singleton

Participantes

- definição de uma operação de classe Instance que permite aos clientes acessarem sua única instância.

- Pode ser responsável pela criação da sua própria instância única.

Colaborações

Os clientes acessam uma instância Singleton unicamente pela operação Instance da classe Singleton.

Conseqüências

O padrão Singleton apresenta vários benefícios:

1. *Acesso controlado à instância única.* Porque a classe Singleton encapsula a sua única instância, pode ter um controle total sobre como e quando os clientes a acessam.

2. *Espaço de nomes reduzidos.* O padrão Singleton representa uma melhoria em relação ao uso de variáveis globais. Ele evita a poluição do espaço de nomes com variáveis globais que armazenam instâncias únicas.

3. *Permite um refinamento de operações e da representação.* A classe Singleton pode ter subclasses e é fácil configurar uma aplicação com

uma instância desta classe estendida. Você pode configurar a aplicação com uma instância da classe de que necessita em tempo de execução.

4. *Permite um número variável de instâncias.* O padrão torna fácil mudar de idéia, permitindo mais de uma instância da classe Singleton. Além disso, você pode usar a mesma abordagem para controlar o número de instâncias que a aplicação utiliza. Somente a operação que permite acesso à instância de Singleton necessita ser mudada.

5. *Mais flexível do que operações de classe.* Uma outra maneira de empacotar a funcionalidade de um Singleton é usando operações de classe. Porém, isso torna difícil mudar um projeto para permitir mais que uma instância de uma classe.

Implementação e exemplo

A implementação em Java deste padrão deve utilizar um campo privado e estático do tipo da própria classe para armazenar a referência da única instância permitida, o qual deve ser inicializado de modo a indicar a inexistência de tal instância. Um método estático, cujo tipo de retorno também é a própria classe, provê o ponto único de acesso a tal instância. Se a operação de instanciação ocorrer apenas na primeira solicitação de um objeto da classe, garantimos a instância única ao mesmo tempo em que a criação só ocorrerá quando estritamente necessário (lazy instantiation).

Como o Java suporta a clonagem de objetos, devemos declarar a classe como final, para impedir que suas subclasses possam implementar a interface Cloneable, possibilitando a duplicação de objetos pelo método “clone()”, o que romperia com as obrigações deste padrão. Caso a classe Singleton seja uma subclasse de outra que suporta a clonagem, então o método “clone()” deve ser sobreposto por outro que apenas lance a exceção CloneNotSupportedException, indicando a impossibilidade de realização desta operação.

Na Listagem 4.1 temos a implementação da classe DBConnection destinada fornecer uma conexão única para um banco de dados específico, que obedece a caracterização que fizemos do padrão Singleton. Pelo uso desta classe podemos reduzir a quantidade de recursos utilizada por uma aplicação durante o acesso ao banco de dados, garantindo uma única conexão. Internamente a classe mantém uma referência única para um objeto de tipo Connection, criado pelo construtor privado declarado. Apenas por meio do método “getConnection()” é possível obter a referência para a conexão única. Existe um método adicional “shutdown()” criado para garantir que a conexão seja encerrada adequadamente. Note que são arbitrárias as definições de

campos e métodos adicionais dentro de uma classe que pretende ser um Singleton, devendo atender aos requisitos específicos da aplicação.

```
package singleton;
import java.sql.*;

public final class DBConnection {

   // Referência para instância única
   private static Connection instance = null;
   private String usuario;
   private String senha;
   private String url;
   private String driver;

   // Construtor privado
   private DBConnection(String usuario, String senha, String url, String
driver) throws ClassNotFoundException, SQLException {
      this.usuario = usuario;
      this.senha   = senha;
      this.url     = url;
      this.driver  = driver;
      Class.forName(this.driver);
      System.out.println("[Driver carregado]");
      instance = DriverManager.getConnection(this.url, this.usuario,
this.senha);
      System.out.println("[Conexão efetuada]");
   }

      // Fornece acesso a instância única
      public static Connection getConnection() throws ClassNotFoundException,
SQLException {
         if (instance == null) {
            // Lazy instantiation: só quando preciso
            new DBConnection("sismat", "sismat", "jdbc:odbc:DbSisMat",
"sun.jdbc.odbc.JdbcOdbcDriver");
         }
         System.out.println("[Retorna conexão]");
         return instance; // Retorna instância única da conexão
       }

       // Encerra conexão adequadamente
       public static void shutdown() throws ClassNotFoundException,
SQLException {
         if (instance != null) {
            instance.close();
            instance = null;
            System.out.println("[Conexão fechada]");
         }
      }

   }
```

Listagem 4.1 – Implementação de um Singleton

Na Listagem 4.2 temos um exemplo simples que permite verificar o funcionamento do Singleton. Pelo método “getConnection()” da classe DBConnection obtemos uma conexão para o banco de dados (cujas strings referentes ao driver e url de conexão foram inclusas diretamente no código da classe). Por esta conexão podem ser

estabelecidas sessões com o banco de dados, o que permite a execução de comandos SQL e processamento dos resultados obtidos. Tentativas de obtenção de novas conexões retornarão uma referência para o mesmo objeto, reduzindo a quantidade de recursos tomados do sistema.

```
package singleton;
import java.sql.*;

public class DBConnectionTest {

   public static void main(String[] args) throws Exception{
      Connection con;
       Statement stmt;

       // Obtém conexão
       con = DBConnection.getConnection();
       stmt = con.createStatement();
       int r = stmt.executeUpdate("INSERT INTO TAB_ALUNOS
VALUES(123021,'Teste','111111')");
       System.out.println(r + " row(s) inserted");
       stmt.close();

       // Obtém (a mesma) conexão
       con = DBConnection.getConnection();
       stmt = con.createStatement();
       ResultSet rs = stmt.executeQuery("SELECT nome, senha FROM TAB_ALUNOS");

       while (rs.next()) {
          System.out.println(rs.getString(1) + ":" +
          rs.getString(2));
       }

      rs.close();
       stmt.close();

       // Encerra conexão
       DBConnection.shutdown();
   }

}
```

Listagem 4.2 – Teste do Singleton

Usos conhecidos

A classe Runtime, da API do J2SE, possui um método “getRuntime()” que retorna uma instância da própria classe, o qual permite o acesso ao ambiente de execução por parte da aplicação. Como o ambiente é único, só pode existir uma instância desta classe, assim é claro que Runtime implementa o padrão de projeto Singleton para cumprir com esta restrição. Outros usos deste padrão estão associados ao acesso controlado de filas de impressão, portas de comunicação e outros recursos restritos do sistema. Também é comum que este padrão esteja associado a outros, tais como o Abstract Factory e o Façade.

O padrão Singleton pode ser modificado para que exista um número máximo de instâncias de uma classe, caracterizando assim um pool de objetos, flexibilizando seu

emprego nas situações em que não é necessário garantir uma instância única, mas onde deve ser limitada a quantidade total de objetos e recursos utilizados.

Padrões Relacionados

Muitos padrões podem ser implementados usando Singleton. Entre eles estão o Abstract Factory, Façade, Builder e Prototype.

4.2 Factory Method

Intenção

Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O Factory Method permite que uma classe delegue a responsabilidade de instanciação às subclasses.

Também conhecido como

Virtual Constructor

Motivação

Os frameworks usam classes abstratas para definir e manter relacionamentos entre objetos. Um framework é freqüentemente responsável também pela criação destes objetos.

Considere um framework para aplicações que podem apresentar múltiplos documentos para o usuário. Duas abstrações-chaves neste framework são as classes Application (aplicação) e Document (documento). As duas classes são abstratas e os clientes devem prover subclasses para realizar suas implementações específicas para a aplicação. Por exemplo, para criar uma aplicação de desenho, definimos as classes DrawingApplication e DrawingDocument. A classe Application é responsável pela administração de Documents e irá criá-los conforme exigido - quando o usuário selecionar Open (abrir) ou New (novo), por exemplo, em um menu.

Uma vez que a subclasse Document a ser instanciada é própria da aplicação específica, a classe Application não pode prever a subclasse de Document a ser instanciada - a classe Application somente sabe quando um documento deve ser criado, e não que tipo de Document criar. Isto cria um dilema: o framework deve instanciar classes, mas ele somente tem conhecimento de classes abstratas, as quais não pode instanciar.

O padrão Factory Method oferece uma solução. Ele encapsula o conhecimento sobre a subclasse de Document que deve ser criada e move este conhecimento para fora do framework.

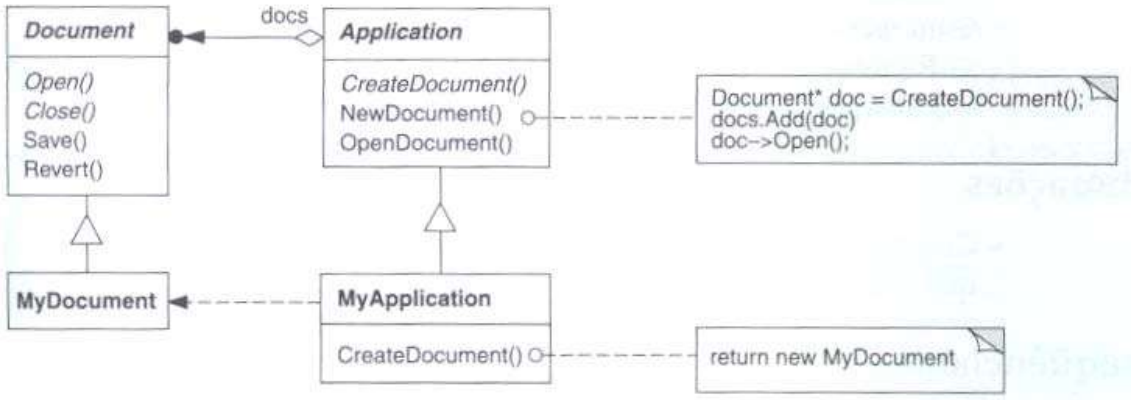


Figura 4.2 – Exemplo de solução oferecida pelo Factory Method

As subclasses de Application redefinem uma operação abstrata CreateDocument em Application para retornar a subclasse apropriada de Document. Uma vez que uma subclasse de Application é instanciada, pode então instanciar Documents específicos da aplicação sem conhecer suas classes. Chamamos de CreateDocument um factory method porque ele é responsável pela "manufatura" de um objeto.

Aplicabilidade

Use o padrão Factory Method quando:

- Uma classe não pode antecipar a classe de objetos a ser criada;
- Uma classe quer que suas subclasses especifiquem os objetos a serem criados;
- Classes delegam responsabilidades para uma dentre outras várias subclasses auxiliares, e você quer localizar qual subclasse auxiliar é delegada.

Estrutura

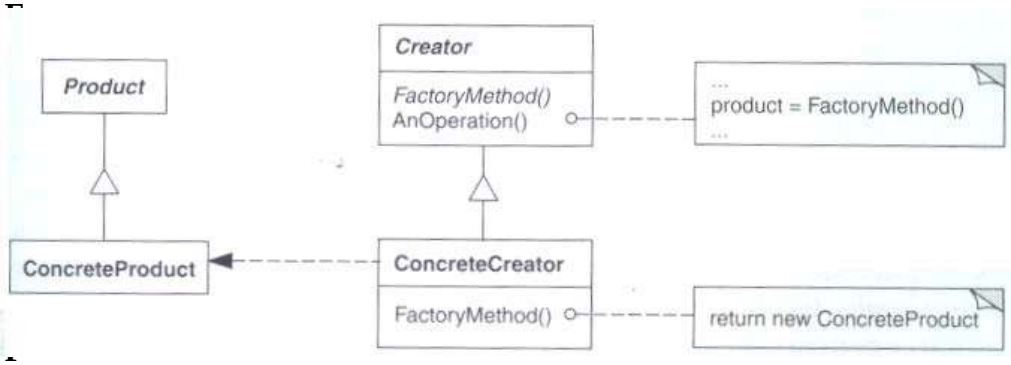


Figura 4.3 – Estrutura do padrão Factory Method

Participantes

- **Product** (Document)
  - Define a interface de objetos que o método fábrica cria.

- **ConcreteProduct** (MyDocument)
  - Implementa a interface de Product.

- **Creator** (Application)
  - Declara o método fábrica o qual retorna um objeto do tipo Product. Creator pode também definir uma implementação por omissão do método factory que retorna por omissão um objeto ConcreteProduct.
  - Pode chamar o método factory para criar um objeto Product.

- **ConcreteCreator** (MyApplication)
  - Redefine (override) o método fábrica para retornar uma instância de um ConcreteProduct.

Colaborações

- Creator depende das suas subclasses para definir o método fábrica de maneira que retorne uma instância do ConcreteProduct apropriado.

Conseqüências

Os padrões FactoryMethods eliminam a necessidade de anexar classes específicas das aplicações no código. O código somente lida com a interface de Product; portanto, ele pode trabalhar com quaisquer classes ConcreteProduct definidas pelo usuário.

Uma desvantagem em potencial dos métodos fábrica é que os clientes podem ter que fornecer subclasses da classe Creator somente para criar um objeto ConcreteProduct em particular. Usar subclasses é bom quando o cliente tem que fornecer subclasses a Creator de qualquer maneira, caso contrário, o cliente deve lidar com outro ponto de evolução.

Apresentamos aqui duas conseqüências adicionais do Factory Method:

1. *Fornece ganchos para subclasses.* Criar objetos dentro de uma classe com um método fábrica é sempre mais flexível do que criar um objeto diretamente. FactoryMethod concede às subclasses um gancho para

fornecer uma versão estendida de um objeto. No exemplo de Documentos, a classe Document poderia definir um método factory chamado CreateFileDialog que cria um objeto *file dialog* por omissão para abrir um documento existente. Uma subclasse de Document pode definir um *file dialog* específico da aplicação redefinindo este método fábrica. Neste caso, o método fábrica não é abstrato, mas fornece uma implementação por omissão razoável.

2. *Concreta hierarquia de classe paralelas.* Nos exemplos que consideramos até aqui o método fábrica é somente chamado por Creators. Mas isto não precisa ser obrigatoriamente assim; os clientes podem achar os métodos fábrica úteis, especialmente no caso de hierarquias de classes paralelas. Hierarquias de classes paralelas ocorrem quando uma classe delega alguma das suas responsabilidades para uma classe separada. Considere, por exemplo, figuras gráficas que podem ser manipuladas interativamente; ou seja, podem ser esticadas, movidas ou giradas usando o *mouse*. Implementar tais interações não é sempre fácil. Isso freqüentemente requer armazenar *e* atualizar informação que registra o estado da manipulação num certo momento. Este estado é necessário somente durante a manipulação; portanto, não necessita ser mantido no objeto-figura. Além do mais, diferentes figuras se comportam de modo diferente quando são manipuladas pelo usuário. Por exemplo, esticar uma figura-linha pode ter o efeito de mover um dos extremos, enquanto que esticar uma figura-texto pode mudar o seu espaçamento de linhas. Com estas restrições, é melhor usar um objeto Manipulator separado que implementa a interação e mantém o registro de qualquer estado específico da manipulação que for necessário. Diferentes figuras utilizarão diferentes subclasses Manipulator para tratar interações específicas. A hierarquia de classes Manipulator resultante é paralela (ao menos parcialmente) à hierarquia de classes de Figure. A classe Figure fornece um método fábrica CreateManipulator que permite aos clientes criar o correspondente Manipulator de uma Figure. As subclasses de Figure substituem este método para retornar uma instância da subclasse Manipulator correta para elas. Como alternativa, a classe Figure pode implementar CreateManipulator para retornar por omissão uma instância de Manipulator, e as subclasses de Figure podem simplesmente herdar essa instância por omissão. As classes Figure que fizerem assim não necessitam de uma subclasse correspondente de Manipulator - por isto dizemos que as hierarquias são somente

parcialmente paralelos. Note como o método fábrica define a conexão entre as duas hierarquias de classes. Nele se localiza o conhecimento de quais classes trabalham juntas.

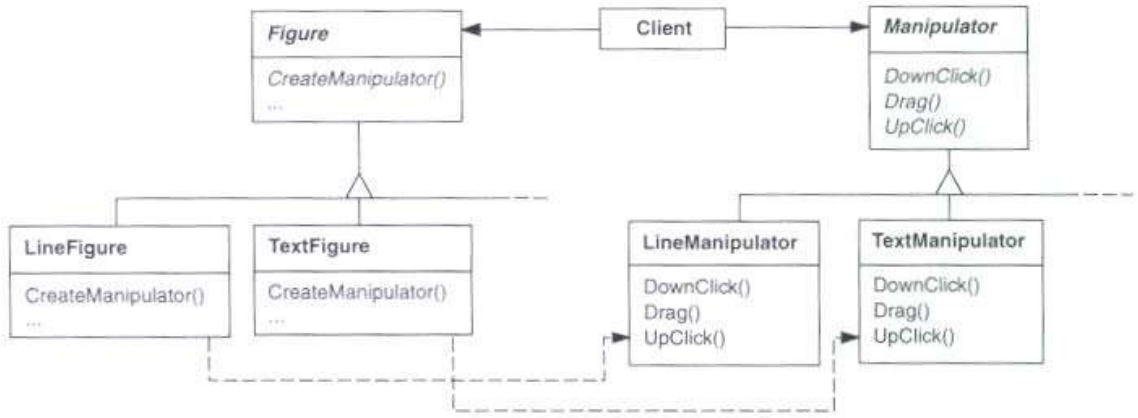


Figura 4.4 – Factory Method com hierarquia paralela de classes

Implementação e exemplo

```
// “Creator”
abstract class Document {^
   //Fields
   protected ArrayList pages = new ArrayList();
   //Constructor
   public Document(){
      this.CreatePages();
   }
   //Properties
   public ArrayList Pages{
      get { return pages; }
   }
   //Factory Method
   abstract public void CreatePages();
}
```

Listagem 4.3.1 – Exemplo do Factory Method – Classe Creator

```
// “ConcreteCreator”
class Resume : Document{
   //implementação Factory Method
   override public void CreatePages(){
      pages.Add ( new SkillPage() );
      pages.Add ( new EducationPage() );
      pages.Add ( new ExperiencePage() );
   }
}
// “ConcreteCreator”
class Report : Document{
   //implementação Factory Method
   override public void CreatePages(){
      pages.Add ( new IntroductionPage() );
      pages.Add ( new ResultsPage() );
      pages.Add ( new ConclusionPage() );
      pages.Add ( new SummaryPage() );
      pages.Add ( new BibliographyPage() );
   }
}
```

Listagem 4.3.2 – Exemplo do Factory Method – Classe ConcreteCreator

Usos conhecidos

Os métodos fábrica permeiam *toolkits e frameworks.* O exemplo precedente de documentos é um uso típico no MacApp. O exemplo do manipulador vem do Unidraw.

ClassView no *framework* Model/View/Controller/Smalltalk-80 tem um método defaultController que cria um controlador, e isso pode parecer ser o método fábrica. Mas subclasses de View especificam a classe no seu controlador por omissão através da definição de defaultControllerClass, que retorna a classe da qual defaultController cria instâncias. Assim, defaultControllerClass é o verdadeiro método fábrica, isto é, o método que as subclasses deveriam redefinir.

Um exemplo mais esotérico em Smalltalk-80 é o método fábrica parserClass definido por Behavior (uma superclasse de todos os objetos que representam classes). Isto permite a uma classe usar um parser (analisador) customizado para seu código-fonte. Por exemplo, um cliente pode definir uma classe SQLParser para analisar o código-fonte de uma classe com comandos SQL embutidos. A Classe Behavior

implementa parserClass retornando a classe Parser padrão da Smalltalk. A classe que inclui comandos SQL embutidos redefine este método (como um método de classe) e retorna a classe SQLParser.

O sistema ORB Orbix da LONA Technologies usa Factory Method para gerar um tipo apropriado de proxy quando um objeto solicita uma referência para um objeto remoto. O Factory Method torna fácil substituir o proxy por omissão por um outro que, por exemplo, use *caching* do lado do cliente.

Padrões relacionados

Abstract Factory é frequentemente implementado utilizado o padrão Factory Method. O exemplo na reação de Motivação no padrão Abstract Factory também ilustra o padrão Factory Method.

Factory Methods são usualmente chamados dentro de Template Methods. No exemplo do documento acima, NewDocument é um método *template*.

Prototypes não exigem subclassificação de Creator. Contudo, freqüentemente necessitam uma operação lnitialize na classe Product. A Creator usa lnitialize para inicializar o objeto. O Factory Method não exige uma operação deste tipo.

Capítulo 5

Padrões de Estrutura

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Dando um exemplo simples, considere como a herança múltipla mistura duas ou mais classes em uma outra. O resultado é uma classe que combina as propriedades das suas classes ancestrais. Este padrão é particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente trabalharem juntas. Um outro exemplo é a forma de classe do padrão Adapter. Em geral, um Adapter faz com que uma interface adaptada (em inglês, adaptee) seja compatível com outra, desta forma fornecendo uma abstração uniforme de diferentes interfaces. A classe adaptadora (adapter) atinge este objetivo herdando, privadamente, de uma classe adaptada. O adapter, então, exprime sua interface em termos da interface da classe adaptada.

Em lugar de compor interfaces ou implementações, os padrões estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes.

O Composite é um exemplo de um padrão estrutural de objetos. Ele descreve como construir uma hierarquia de classes composta para dois tipos de objetos: primitivos e compostos. Os objetos compostos permitem compor objetos primitivos e outros objetos compostos em estruturas arbitrariamente complexas. No padrão Proxy, um procurador funciona como um substituto ou um marcador para outro objeto. Um proxy (procurador) pode ser usado de várias formas. Ele pode atuar como um representante local para um objeto situado num espaço de endereço remoto. Pode representar um grande objeto que deveria ser carregado por demanda. Pode proteger o acesso a um objeto sensível. Proxies fornecem um nível de referência indireta a propriedades específicas de objetos. Daí eles poderem restringir, aumentar ou alterar estas propriedades.

O padrão Flyweight define uma estrutura para o compartilhamento de objetos. Os objetos são compartilhados por pelo menos duas razões: eficiência e consistência. O Flyweight focaliza o compartilhamento para uso eficiente de espaço. As aplicações que usam uma porção de objetos devem prestar atenção no custo de cada objeto. Pode-se obter economia substancial e usando o compartilhamento de objetos, em lugar de

replicá-los. Mas objetos podem ser compartilhados somente se eles não definem estados dependentes do contexto.

Objetos Flyweight não possuem tais estados. Qualquer informação adicional de que necessitem para executar suas tarefas é passada aos mesmos quando necessário. Não tendo estados dependentes de contexto, os objetos Flyweight podem ser compartilhados livremente.

Enquanto o Flyweight mostra o que fazer com muitos objetos pequenos, Façade mostra como fazer um único objeto representar todo um subsistema. Um objeto façade (fachada) é uma representação para um conjunto de objetos. Façade executa suas responsabilidades repassando mensagens para os objetos que ela representa. O padrão Bridge separa a abstração de um objeto da sua implementação, de maneira que elas possam variar independentemente.

O Decorator descreve como acrescentar dinamicamente responsabilidades ao objetos. O Decorator é um padrão estrutural que compõe objetos recursivamente para permitir um número ilimitado de responsabilidades adicionais. Por exemplo, um objeto Decorator que contém um componente de uma interface de usuário pode adicionar uma decoração, como uma borda ou sombra, ao componente, ou pode adicionar urna funcionalidade como rolamento e zoom. Podemos adicionar duas decorações simplesmente encaixando um objeto Decorator dentro do outro, e assim por diante, para outras decorações adicionais. Para conseguir isto, cada objeto Decorator deve oferecer a mesma interface do seu componente e repassar mensagens para ele. O Decorator pode executar o seu trabalho (tal como desenhar uma borda em volta do componente) antes ou depois de repassar uma mensagem.

5.1 Adapter

Intenção

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.

Motivação

Algumas vezes, uma classe de um toolkit, projetada para ser reutilizada não é reutilizável porque sua interface não corresponde à interface específica de domínio requerida por uma aplicação.

Considere, por exemplo, um editor de desenhos que permite aos usuários desenhar e arranjar elementos gráficos (linhas, polígonos, texto, etc.) em figuras e diagramas. A abstração-chave do editor de desenhos é o objeto gráfico, o qual tem uma forma editável e pode desenhar a si próprio. A interface para objetos gráficos é definida por uma classe abstrata chamada Shape. O editor define uma subclasse de Shape para cada tipo de objeto gráfico: uma classe LineShape para linhas, uma classe PolygonShape para polígonos, e assim por diante.

Classes para formas geométricas elementares, como LineShape e PolygonShape, são bastante fáceis de ser implementadas porque as suas capacidades de desenho e edição são inerentemente limitadas. Mas uma subclasse TextShape que pode exibir e editar textos é mais difícil de ser implementada, uma vez que mesmo a edição básica de textos envolve atualizações complicadas de tela e gerência de buffer. Entretanto, pode já existir um toolkit para construção de interfaces de usuários, o qual já oferece uma sofisticada classe TextView para a exibição e edição de textos. Idealmente, gostaríamos de reutilizar TextView para implementar TextShape, porém, o toolkit não foi projetado levando classes Shape em consideração. Assim, não podemos usar de maneira intercambiável objetos TextView e Shape.

Como é possível que classes existentes e não-relacionadas, como TextView, funcionem em uma aplicação que espera classes com uma interface diferente e incompatível? Poderíamos mudar a classe TextView de maneira que ela fosse coerente com a interface de Shape, porém, a menos que tenhamos o código-fonte do toolkit, essa opção não é viável. Mesmo que tivéssemos o código-fonte, não teria sentido mudar TextView; o toolkit não deveria ter que adotar interfaces específicas de domínios somente para fazer com que uma aplicação funcione.

Em vez disso, poderíamos definir TextShape de maneira que ele adapte a interface de TextView àquela de Shape. Podemos fazer isto de duas maneiras: (1) herdando a interface de Shape e a implementação de TextView, ou (2) compondo uma instância de

TextView dentro de uma TextShape e implementando TextShape em termos da interface de TextView.

Estas duas abordagens correspondem às versões do padrão Adapter para classes e para objetos. Chamamos TextShape um adaptador.

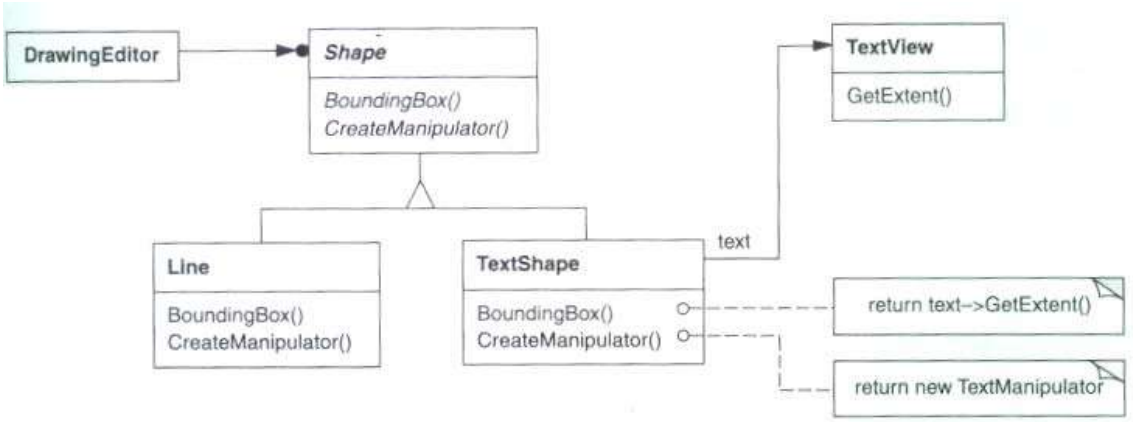


Figura 5.1 – Exemplo de solução oferecida pelo Adapter

A Figura 5.1 ilustra o caso de um adaptador para objetos. Ele mostra como solicitações de BoundingBox, declarada na classe Shape, são convertidas em solicitações para GetExtent, definida em TextView. Uma vez que TextShape adapta TextView à interface de Shape, o editor de desenhos pode reutilizar a classe TextView que seria incompatível de outra forma.

Freqüentemente, um adaptador é responsável por funcionalidades não oferecidas pela classe adaptada. O diagrama mostra como um adaptador pode atender tais responsabilidades. O usuário deveria ser capaz de “arrastar” cada objeto Shape para uma nova posição de forma interativa, porém, TextView não está projetada para fazer isso. TextShape pode acrescentar essa função através da implementação da operação CreateManipulator, de Shape, a qual retorna uma instância da subclasse Manipulater apropriada.

Manipulator é uma classe abstrata para objetos que sabem como animar um Shape em resposta à entrada de usuário, tal como arrastar a forma geométrica para uma nova localização. Existem subclasses de Manipulator para diferentes formas; por exemplo, TextManipulator é a subclasse correspondente para TextShape. Pelo retorno de uma instância de TextManipulator, TextShape acrescenta a funcionalidade que Shape necessita mas que TextView não tem.

Aplicabilidade

Use o padrão Adapter quando:

- você quiser usar uma classe existente, mas sua interface não corresponde a interface de que necessita;

- você quiser criar uma classe reutilizável que coopera com classes não-relacionadas ou não-previstas, ou seja, classes que não necessariamente tenham interfaces compatíveis;

- (somente para adaptadores de objetos) você necessitar usar várias subclasses existentes, porém, é impraticável adaptar estas interfaces criando subclasses para cada uma. Um adaptador de objeto pode adaptar a interface da sua classe-mãe.

Estrutura

Um adaptador de classe usa a herança múltipla para adaptar uma interface à outra, conforme Figura 5.2:

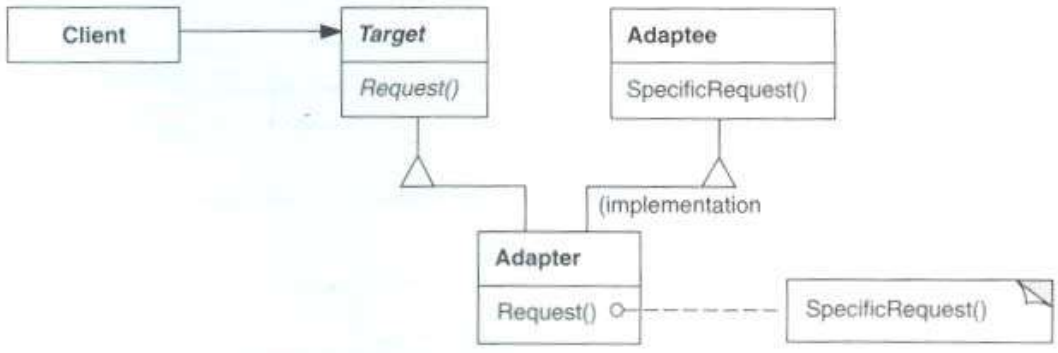


Figura 5.2 – Estrutura do padrão Adapter – Adaptador de classes

Um adaptador de objeto depende da composição de objetos, conforme Figura 5.3:

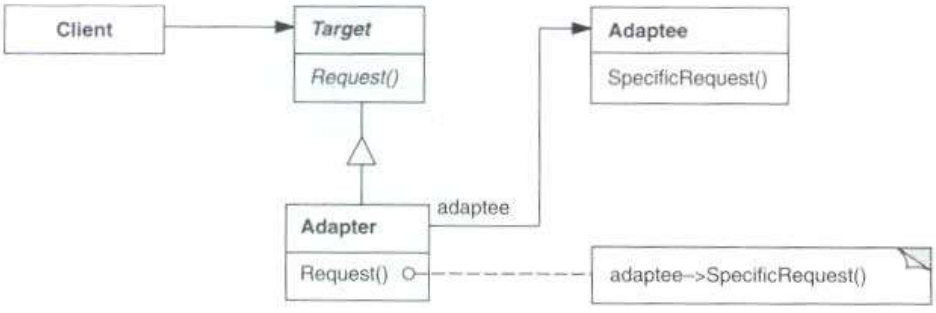


Figura 5.3 – Estrutura do padrão Adapter – Adaptador de objetos

Participantes

- **Target** (Shape)
  - Define a interface específica do domínio que Client usa.

- **Client** (DrawingEditor)
  - colabora com objetos compatíveis com a interface de Target.

- **Adaptee** (TextView)
  - define uma interface existente que necessita ser adaptada.

- **Adaptee** (TextView)
  - define a interface do Adaptee à interface de Target.

Colaborações

- Os clientes chamam operações em uma instância de Adapter. Por sua vez, o adapter chama operações de Adaptee que executam a solicitação.

Conseqüências

Os adaptadores de classes e de objetos têm diferentes soluções de compromisso. Um adaptador de classe:

- adapta Adaptee a Target através do uso efetivo de urna classe Adapter concreta. Em conseqüência, um adaptador de classe não funcionará quando quisermos adaptar uma classe e todas as suas subclasses;
- permite a Adapter substituir algum comportamento do Adaptee, uma vez que Adapter é uma subclasse de Adaptee;

- introduz somente um objeto, e não é necessário endereçamento indireto adicional por ponteiros para chegar até o Adaptee.

Um adaptador de objeto:

- permite a um único Adapter trabalhar com muitos Adaptees - isto é, o Adaptee em si e todas as suas subclasses(se existirem). O Adapter também pode acrescentar funcionalidade a todos os Adaptees de uma só vez;

- torna mais difícil redefinir um comportamento de Adaptee. Ele exigirá a criação de subclasses de Adaptee e fará com que Adapter referencie a subclasse ao invés do Adaptee em si.

Aqui apresentamos outros pontos a serem considerados quando usamos o padrão Adapter:

1. *Quanta adaptação Adapter faz?* Os Adapters variam no volume de trabalho que executam para adaptar o Adaptee à interface de Target. Existe uma variação do trabalho possível, desde a simples conversão de interface - por exemplo, mudar os nomes das operações - até suportar um conjunto de operações inteiramente diferente. O volume de trabalho que o Adapter executa depende de quão similar é a interface de Target em relação aos seus Adaptees.

2. *Adaptadores conectáveis (pluggable).* Uma classe é mais reutilizável quando você minimiza as suposições que outras classes devem fazer para utilizá-la Através da construção da adaptação de interface em uma classe, você elimina a suposição de que outras classes vêm a mesma interface. Dito de outra maneira, a adaptação de interfaces permite incorporar à nossa classe a sistemas existentes que podem estar esperando interfaces diferentes para a classe. ObjectWorks/Smalltalk usa o termo pluggable adapter para descrever classes com adaptação de interfaces incorporadas.

3. *Utilização de adaptadores de dois sentidos para fornecer transparência.* Um problema potencial com adaptadores decorre do fato de que eles não são transparentes para todos os clientes. Um objeto adaptado não oferece a interface do objeto original, por isso ele não pode ser usado onde o original for. Adaptadores de dois sentidos (two-way adapters) podem fornecer essa transparência. Eles são úteis quando dois clientes diferentes necessitam ver um objeto de forma diferente.

Padrões relacionados

O padrão Bridge tem uma estrutura similar a um adaptador de objeto, porém, Bridge tem uma intenção diferente: tem por objetivo separar uma interface da sua implementação, de modo que elas possam variar de maneira fácil e independente. Um adaptador se destina a mudar a interface de um objeto existente.

O padrão Decorator aumenta outro objeto sem mudar sua interface. Desta forma, um Decorator é mais transparente para a aplicação do que um adaptador. Como conseqüência, Decorator suporta a composição recursiva, a qual não é possível com adaptadores puros.

O Proxy define um representante ou “procurador” para outro objeto e não muda a sua interface.

5.2 Decorator

Intenção

Dinamicamente, agregar responsabilidades adicionais a um objeto . Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.

Motivação

Algumas vezes queremos acrescentar responsabilidades a objetos individuais, e não a toda uma classe. Por exemplo, um toolkit para construção de interfaces gráficas de usuário deveria permitir a adição de propriedades, como bordas, ou comportamentos, como rolamento, para qualquer componente da interface do usuário.

Uma forma de adicionar responsabilidades é a herança. Herdar uma borda de uma outra classe coloca uma borda em volta de todas as instâncias de uma subclasse. Contudo, isto é inflexível, porque a escolha da borda é feita estaticamente. Um cliente não pode controlar como e quando decorar o componente com uma borda.

Uma abordagem mais flexível é embutir o componente em outro objeto que acrescenta a borda. O objeto que embute o primeiro é chamado de decorator. O decorator segue a interface do componente que decora, de modo que sua presença é transparente para os clientes do componente. O decorator repassa solicitações para o componente, podendo executar ações adicionais (tais como desenhar uma borda) antes ou depois do repasse. A transparência permite encaixar decoradores recursivamente, desta forma permitindo um número ilimitado de responsabilidades adicionais.

Suponha que tenhamos um objeto TextView que exibe texto numa janela. Por omissão, TextView não tem barras de rolamento porque nem sempre a necessitamos. Quando houver necessidade, poderemos usar um ScrollDecorator para acrescentá-las, conforme Figura 5.4.

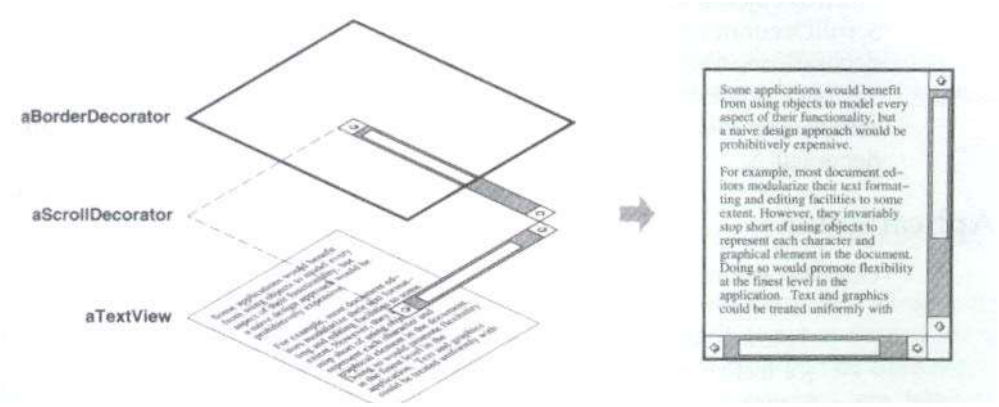


Figura 5.4 – Exemplo de uso do padrão Decorator

Suponha, também, que queiramos acrescentar uma borda preta espessa ao redor do objeto TextView. Também podemos usar um objeto BorderDecorator para esta finalidade. Simplesmente compomos os decoradores com TextView para produzir o resultado desejado.

O diagrama de objetos exibido na Figura 5.5 mostra como compor um objeto TextView com objetos BorderDecorator e ScrollDecorator para produzir uma visão do texto cercada por bordas e rolável:

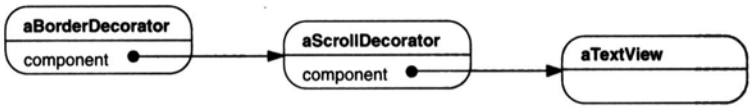


Figura 5.5 – Decorator: Composição de objetos

As classes ScrollDecorator e BorderDecorator são subclasses de Decorator, uma classe abstrata destinada a componentes visuais que decoram outros componentes visuais.

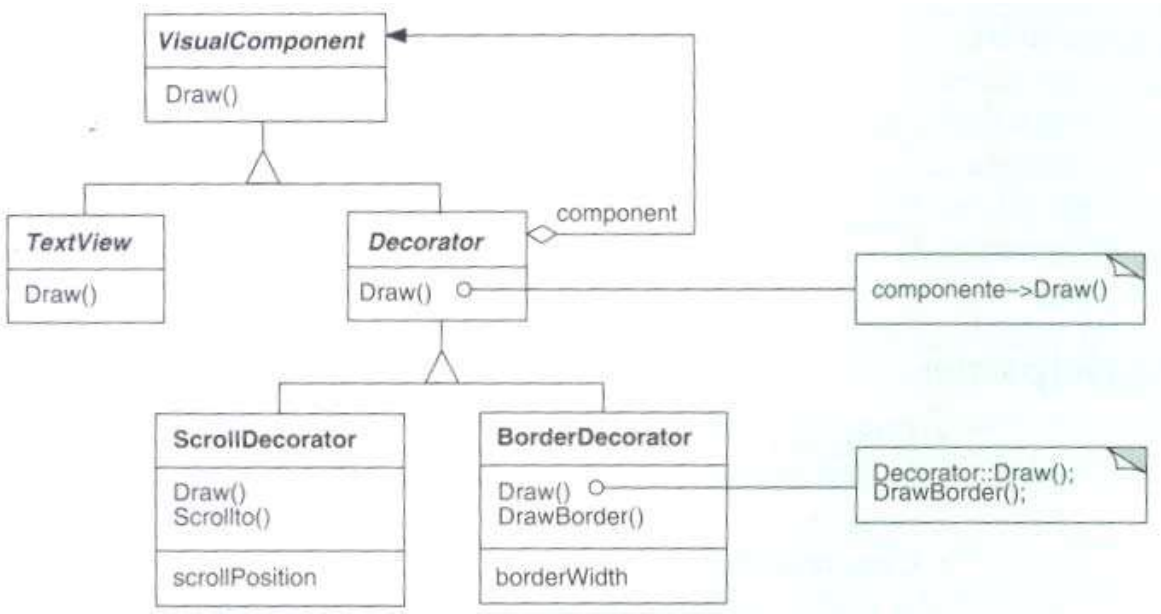


Figura 5.6 – Exemplo de solução do padrão Decorator

VisualComponent é a classe abstrata para objetos visuais. Ela define suas interface de desenho e de tratamento de eventos. Observe como a classe Decorator simplesmente repassa as solicitações de desenho para o seu componente e como as subclasses de Decorator podem estender esta operação.

As subclasses de Decorator são livres para acrescentar operações para funcionalidades específicas. Por exemplo, a operação ScrollTo, de ScrollDecorator, permite a outros objetos fazerem rolamento na interface se eles souberem que existe um objeto ScrollDecorator na interface. O aspecto importante deste padrão é que ele permite que decoradores (decorators) apareçam em qualquer lugar no qual possa aparecer um VisualComponent. Desse modo, os clientes em geral não podem distinguir entre um componente decorado e um não-decorado, e assim são totalmente independentes das decorações.

Aplicabilidade

Use Decorator:

- para acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos;
- para responsabilidades que podem ser removidas;
- quando a extensão através do uso de subclasses não é prática. Às vezes, um grande número de extensões independentes é possível e isso poderia

produzir uma explosão de subclasses para suportar cada combinação. Ou a definição de uma classe pode estar oculta ou não estar disponível para a utilização de subclasses.

Estrutura

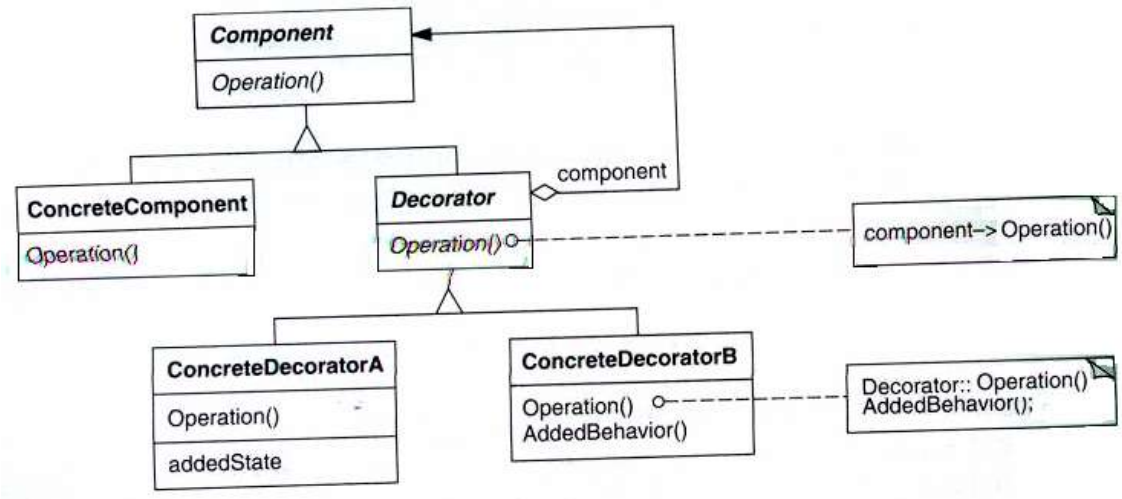


Figura 5.7 – Estrutura do padrão Decorator

Participantes

- **Component** (VisualComponent)
  - Define a interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente.

- **ConcreteComponent** (TextView)
  - Define um objeto para o qual responsabilidades adicionais podem ser atribuídas.

- **Decorator**
  - Mantém uma referência para um objeto Component e define uma interface que segue a interface de Component.

- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  - Acrescenta responsabilidades ao componente.

Colaborações

- Decorator repassa solicitações para o seu objeto Component. Opcionalmente, ele pode executar operações adicionais antes e depois de repassar a solicitação.

Conseqüências

O padrão Decorator tem pelo menos dois benefícios-chaves de duas deficiências:

1. *Maior flexibilidade do que a herança estática.* O padrão Decorator fornece uma maneira mais flexível de acrescentar responsabilidades a objetos do que pode ser feito com herança estática (múltipla). Com o uso de decoradores, responsabilidades podem ser acrescentadas e removidas em tempo de execução simplesmente associando-as e desassociando-as a um objeto. Em comparação, a herança requer a criação de uma nova classe para cada responsabilidade adicional. Isso dá origem a muitas classes e aumenta a complexidade de um sistema. Além do mais, fornecer diferentes classes Decorator para uma específica classe Component permite misturar e associar (match) responsabilidades. Os Decorators também tornam fácil acrescentar uma propriedade duas vezes. Por exemplo, para dar a um TextView uma borda dupla, simplesmente associe dois BorderDecorators. Herdar de uma classe Border duas vezes é um procedimento sujeito a erros, na melhor das hipóteses.

2. *Evita classes sobrecarregadas de características na parte superior da hierarquia.* Um Decorator oferece uma abordagem do tipo “use quando for necessário” para adição de responsabilidades. Em vez de tentar suportar todas as características previsíveis em uma classe complexa e customizada, você pode definir uma classe simples e acrescentar funcionalidade de modo incremental com objetos Decorator. A funcionalidade necessária pode ser composta a partir de peças simples. Como resultado, uma aplicação não necessita incorrer no custo de características e recursos que não usa. Também é fácil definir novas espécies de Decorators independentemente das classes de objetos que eles estendem, mesmo no caso de extensões não-previstas. Estender uma classe

complexa tende a expor detalhes não-relacionados com as responsabilidades que você está adicionando.

3. *Um decorador e o seu componente não são idênticos.* Um decorador funciona como um envoltório transparente. Porém, do ponto de vista da identidade de um objeto, um componente decorado não é idêntico ao próprio componente. Dai não poder depender da identidade de objetos quando você utiliza decoradores.

4. *Grande quantidade de pequenos objetos.* Um projeto que usa o Decorator freqüentemente resulta em sistemas compostos por uma grande quantidade de pequenos objetos parecidos. Os objetos diferem somente na maneira como são interconectados, e não nas suas classes ou no valor de suas variáveis. Embora esses sistemas sejam fáceis de customizar por quem os compreende, podem ser difíceis de aprender e depurar.

Implementação e exemplo

```
public class Leite extends Decorator{
   public Leite(Bebida comp){
      bebida = comp;
   }

   public String gerarBebida(){
      String descricao = null;
      if (bebida != null) descricao = bebida.gerarBebida();
      descricao += “adcionando leite, “;
      return descricao;
   }

}

SAÍDA:
cafe adicionando acucar, adicionando leite, adicionando chocolate,
```

Listagem 5.1 – Exemplo do padrão Decorator

```
public class Usuario{
   private static Bebida bebida;

   public static void main(String[] argv){
      bebida = new Cafe();
      bebida = new Acucar(bebida);
      bebida = new Leite(bebida);
      bebida = new Chocolate(bebida);
      String beber = bebida.gerarBebida();
      System.out.println(beber);
   }
}
```

Listagem 5.2 – Exemplo de uso do padrão Decorator

Padrões relacionados

Adapter: Um padrão Decorator é diferente de um padrão Adapter no sentido de que um Decorator somente muda as responsabilidades de um objeto, não a sua interface; já um Adapter dará a um objeto uma interface completamente nova.

Composite: Um padrão Decorator pode ser visto como um padrão Composite degenerado com somente um componente. Contudo, um Decorator acrescenta responsabilidades adicionais - ele não se destina a agregação de objetos.

Strategy: Um padrão Decorator permite mudar a superfície de um objeto, um padrão Strategy permite mudar o seu interior. Portanto, essas são duas maneiras alternativas de mudar um objeto.

Capítulo 6

Padrões de Comportamento

Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Estes padrões caracterizam fluxos de controle para permitir que você se concentre somente na maneira como os objetos são interconectados.

Os padrões comportamentais de classe utilizam a herança para distribuir o comportamento entre classes. São dois os padrões deste tipo: Template Method e o Interpreter. O Template Method é o mais simples e o mais comum dos dois. Um método template é uma definição abstrata de um algoritmo. Ele define o algoritmo passo a passo. Cada passo invoca uma operação abstrata ou uma operação primitiva. Uma subclasse encarna um algoritmo através da definição das operações abstratas. O outro padrão comportamental de classe é o Interpreter, o qual representa uma gramática como uma hierarquia de classes e implementa um interpretador como uma operação em instâncias destas classes.

Os padrões comportamentais de objetos utilizam a composição de objetos em vez da herança. Alguns descrevem como um grupo de objetos pares (peer objects) cooperam para execução de uma tarefa que nenhum objeto sozinho poderia executar por si mesmo. Um aspecto importante aqui é como os objetos-pares conhecem uns aos outros. Os pares poderiam manter referências explícitas uns para os outros, mas isso aumentaria o seu acoplamento. Levado ao extremo, cada objeto teria conhecimento de cada um dos demais. O padrão Mediator evita essa situação pela introdução de um objeto mediador entre pares. Um mediador fornece o referenciamento indireto necessário para um acoplamento forte.

O padrão Chain of Responsibility fornece um acoplamento ainda mais fraco. Ele permite enviar solicitações implicitamente para um objeto através de uma cadeia de objetos candidatos. Qualquer candidato pode satisfazer a solicitação dependendo de condições em tempo de execução. O número de candidatos é aberto e você pode selecionar quais candidatos participam da cadeia em tempo de execução.

O padrão Observer define e mantém uma dependência entre objetos. O exemplo clássico do Observer está no Model/View/Controller, onde todas as vistas (views) do modelo são notificadas sempre que o estado do modelo muda.

Outros padrões comportamentais de objetos se preocupam com o encapsulamento de comportamento em um objeto e com a delegação de solicitações para ele. O padrão

Strategy encapsula um algoritmo num objeto. Strategy torna fácil especificar e mudar o algoritmo que um objeto usa. O padrão Command encapsula uma solicitação num objeto, de maneira que possa ser passada como um parâmetro, armazenada numa lista histórica ou manipulada de outras formas. O padrão State encapsula os estados de um objeto, de maneira que o objeto possa mudar o seu comportamento quando o seu objeto-estado muda. Visitor encapsula comportamento que de outra forma seria distribuído entre classes, Iterator abstrai a maneira como objetos de um agregado são acessados e percorridos.

6.1 Observer

Intenção

Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

Motivação

Um efeito colateral comum resultante do particionamento de um sistema em uma coleção de classes cooperantes é a necessidade de manter a consistência entre objetos relacionados. Você não deseja obter consistência tornando as classes fortemente acopladas, porque isso reduz a sua reusabilidade.

Por exemplo, muitos toolkits para construção de interfaces gráficas de usuário separam os aspectos de apresentação da interface do usuário dos dados da aplicação subjacente. As classes que definem dados da aplicação e apresentações podem ser reutilizadas independentemente. Elas também podem trabalhar em conjunto. Tanto um objeto planilha como um objeto gráfico de barras pode ilustrar informações no mesmo objeto com dados da aplicação usando diferentes apresentações. A planilha e o gráfico de barras não têm conhecimento um outro, permitindo reutilizar somente aquela que você necessita. Porém, elas se comportam como se conhecessem. Quando o usuário muda a informação na planilha, o gráfico de barras reflete as mudanças imediatamente, e vice-versa.

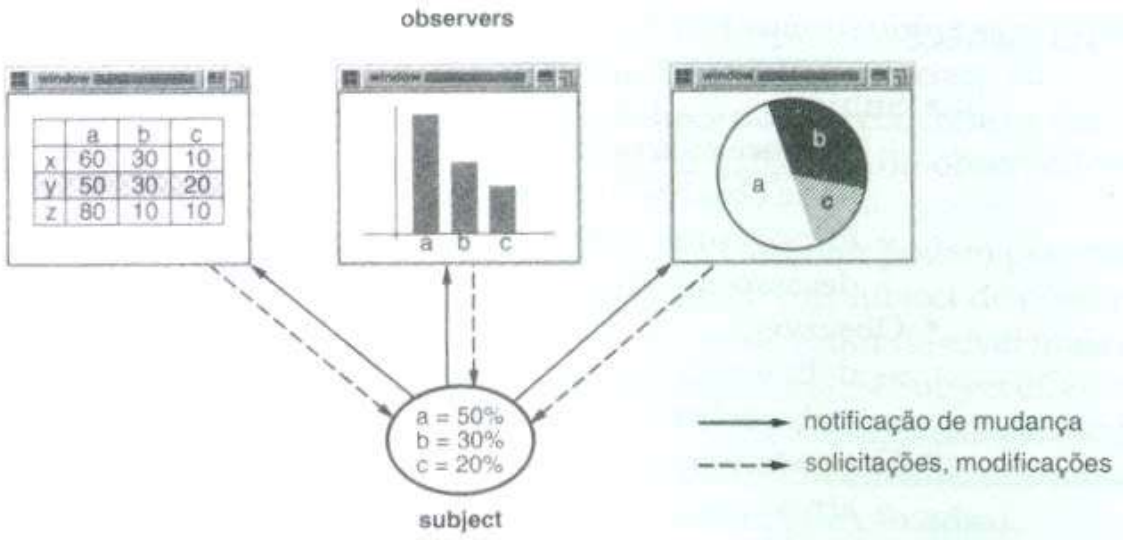


Figura 6.1 – Exemplo de aplicação do padrão Observer

Este comportamento implica que a planilha e o gráfico de barras são dependentes do objeto de dados e, portanto, deveriam ser notificados sobre qualquer mudança no seu estado. E não há razão para limitar o número de objetos dependentes a dois objetos; pode haver um número qualquer de diferentes interfaces do usuário para os mesmos dados.

O padrão Observer descreve como estabelecer estes relacionamentos. Os objetos-chave neste padrão são subject (assunto) e observer (observador). Um subject pode ter um número qualquer de observadores dependentes. Todos os observadores são notificados quando o subject sofre uma mudança de estado. Em resposta a isso, cada observador inquirirá o subject para sincronizar o seu estado com o estado do subject.

Este tipo de interação também é conhecido como publish-subscribe (editor assinante). O subject é o publicador de notificações. Ele envia essas notificações sem ter que saber quem são os seus observadores. Um número qualquer de observadores pode inscrever-se para receber notificações.

Aplicabilidade

Use o padrão Observer em qualquer uma das seguintes situações:

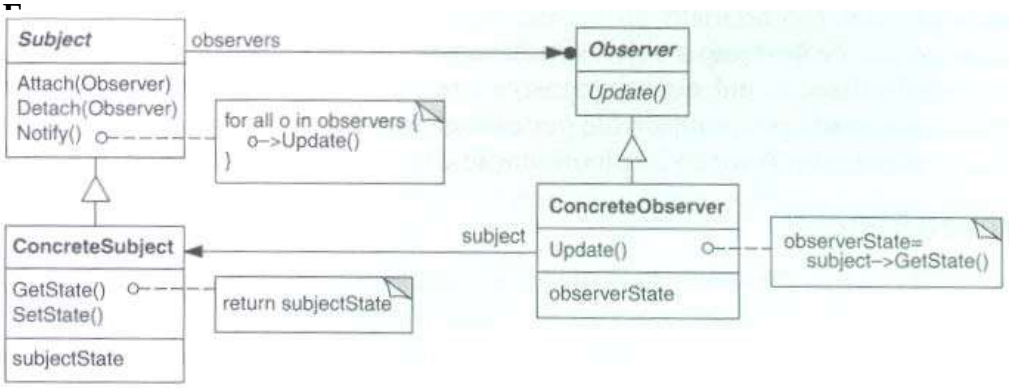
- quando uma abstração tem dois aspectos, um dependente do outro. Encapsulando esses aspectos em objetos separados, fica permitido variá-los e reutilizá-los independentemente;

- quando uma mudança em um objeto exige mudanças em outros, e você não sabe quantos objetos necessitam ser mudados;

- quando um objeto deveria ser capaz de notificar outros objetos sem fazer hipóteses, ou usar informações, de quem são estes objetos. Em outras palavras, você não quer que estes objetos sejam fortemente acoplados.

Estrutura

E



F

Figura 6.2 – Estrutura do padrão Observer

Participantes

- **Subject**
  - conhece os seus observadores. Um número qualquer de objetos Observer pode observar um subject.
  - fornece uma interface para acrescentar e remover objetos para associar e desassociar objetos observer.
- **Observer**
  - define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um Subject.
- **ConcreteSubject**
  - armazena estados de interesse para objetos ConcreteObserver.
  - envia uma notificação para os seus observadores quando seu estado muda.
- **ConcreteObserver**
  - mantém uma referência para um objeto ConcreteSubject.
  - armazena estados que deveriam permanecer consistentes com os do Subject.

- implementa a interface de atualização de Observer, para manter seu estado consistente com o do subject.

Colaborações

- O ConcreteSubject notifica seus observadores sempre que ocorrer uma mudança que poderia tornar inconsistente o estado deles com o seu próprio.

- Após ter sido informado de uma mudança no subject concreto, um objeto ConcreteObserver pode consultar o subject para obter informações. O ConcreteObserver usa esta informação para reconciliar o seu estado com aquele do subject. A figura 6.3 ilustra as colaborações entre um subject e dois observadores:

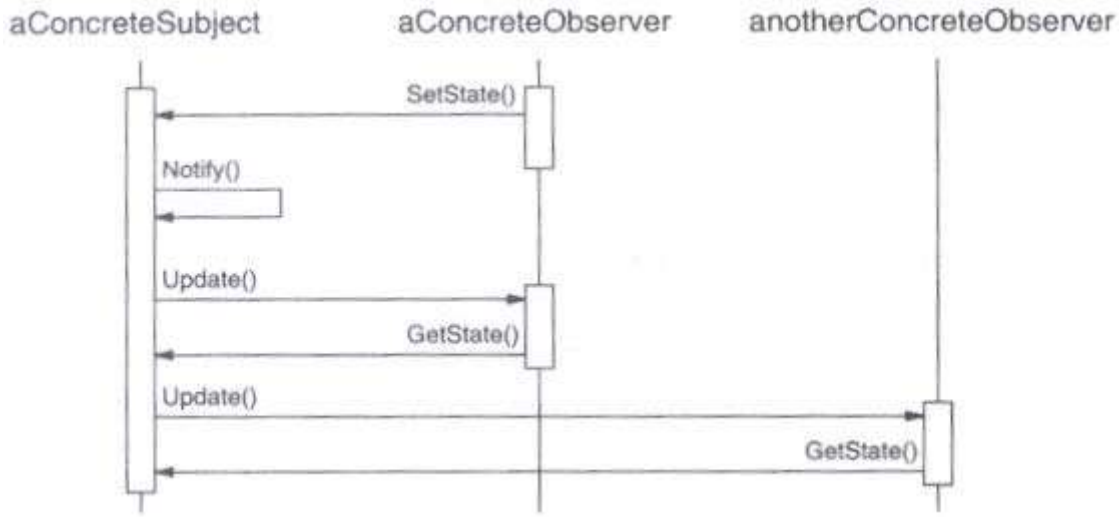


Figura 6.3 – Colaborações entre um subject e dois observadores

Note como o objeto Observer que inicia a solicitação de mudança posterga atualização até que ele consiga uma notificação do subject. Notify não é sempre chamada pelo subject. Ela pode ser chamada por um observador ou por um outro tipo de objeto.

Conseqüências

O padrão Observer permite variar subjects e observadores de forma independente. Você pode reutilizar subjects sem reutilizar seus observadores e vice-versa. Ele permite acrescentar observadores sem modificar o subject ou outros observadores. Benefícios adicionais e deficiências do padrão Observer incluem o seguinte:

1. *Acoplamento abstrato entre Subject e Observer.* Tudo que o subject sabe é que ele tem uma lista de observadores, cada um seguindo a interface simples da classe abstrata Observer. O subject não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre o subject e os observadores é abstrato e mínimo. Por não serem fortemente acoplados, Subject e Observer podem pertencer a diferentes camadas de abstração em um sistema. Um subject de nível mais baixo pode comunicar-se com e informar um observador de nível mais alto, desta maneira mantendo intacta as camadas do sistema. Se Subject e Observer forem agrupados, então o objeto resultante deve cobrir duas camadas (e violar a estrutura de camadas), ou ser forçado a residir em uma das camadas (o que pode comprometer a abstração da estrutura de camadas).

2. *Suporte para comunicações broadcast.* Diferentemente de uma solicitação ordinária, a notificação que um subject envia não necessita especificar seu receptor. A notificação é transmitida automaticamente para todos os objetos interessados que a subscreveram. O subject não se preocupa com quantos objetos interessados existem; sua única responsabilidade é notificar seus observadores. Isto dá a liberdade de acrescentar e remover observadores a qualquer momento. É responsabilidade do observador tratar ou ignorar uma notificação.

3. *Atualizações inesperadas.* Porque um observador não tem conhecimento da presença dos outros, eles podem ser cegos para o custo global de mudança do subject. Uma operação aparentemente inócua no subject pode causar uma cascata de atualizações nos observadores e seus objetos dependentes. Além do mais, critérios de dependência que não estão bem-definidos ou mantidos normalmente conduzem a atualizações espúrias que podem ser difíceis de detectar. Este problema é agravado pelo fato de que o protocolo simples de atualização não fornece detalhes sobre o que mudou no subject. Sem protocolos adicionais para ajudar os observadores a descobrir o que mudou, eles podem ser forçados a trabalhar duro para deduzir as mudanças.

Implementação e exemplo

```
public abstract class Candidato{
   private Vector apuradores;
   private static int votos;
   public void setVotos(int novosVotos){
      votos = votos + novosVotos;
      notifyApurador(this);   }
   public int getVotos(){ return votos; }
   public void addApurador(Apurador adp){ apuradores.add(adp); }
   public void removeApurador(Apurador adp){ apuradores.remove(adp); }
   public void notifyApurador(Candidato candidato){
      Apurador apurador;
      Enumeration enumApuradores = apuradores.elements();
      while( enumApuradores.hasMoreElements() ){
         apurador = (Apurador) enumApuradores.nextElement();
         apurador.update(candidato);
      }   }   }
```

Listagem 6.1 – Implementação do padrão Observer: Classe Candidato

```
public class ApuradorRio extends Apurador{
   public void update(Candidato candidato){
      candidato.getVotos();
   }
}
```

Listagem 6.2 – Implementação do padrão Observer

Usos conhecidos

O primeiro e talvez mais conhecido exemplo do padrão Observer aparece no Model/View/Controller (MVC), da Smalltalk, o framework para a interface do usuário no ambiente Smalltalk. A classe Model, do MVC, exerce o papel do Subject, enquanto View é a classe base para observadores.

Padrões relacionados

Mediator: encapsulando a semântica de atualizações complexas, o ChangeManager atua como um mediador entre subjects e observadores.

Singleton: O ChangeManager pode usar o padrão Singleton para torná-lo único e globalmente acessível.

6.2 Template Method

Intenção

Definir o esqueleto de um algoritmo em uma operação, postergando (deferring) alguns passos para subclasses. Template Method (Gabarito de Método) permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

Motivação

Considere um framework para aplicações que fornece as classes Application e Document. A classe Application é responsável por abrir documentos existentes armazenados num formato externo, tal como um arquivo. Um objeto Document representa a informação num documento, depois que ela foi lida do arquivo.

As aplicações construídas com o framework podem criar subclasses de Application e Document para atender necessidades específicas. Por exemplo, uma aplicação de desenho define as subclasses DrawApplication e DrawDocument; uma aplicação de planilha define as subclasses SpreadsheetApplication e SpreadsheetDocument.

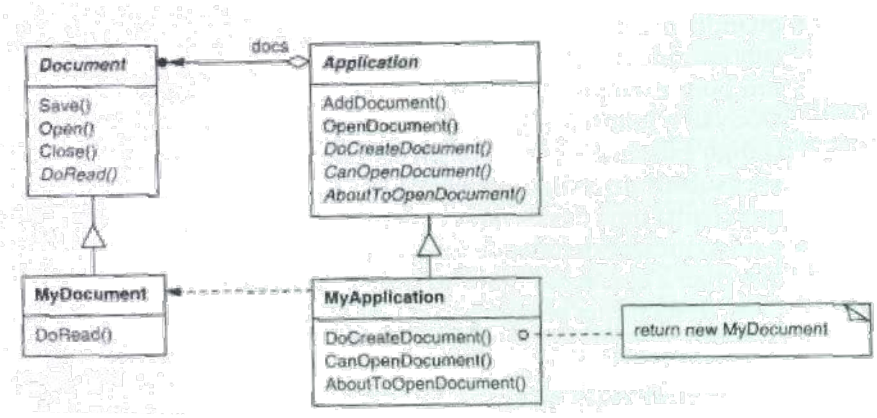


Figura 6.4 – Exemplo do padrão Template Method

A classe abstrata Application define o algoritmo para abrir e ler um documento na sua operação OpenDocument. OpenDocument define cada passo para a abertura de um

documento. Ela verifica se o documento pode ser aberto, cria o objeto Document específico para a aplicação, acrescenta-o ao seu conjunto de documentos e lê Document de um arquivo.

Chamamos OpenDocument um template method. Um método template define um algoritmo em termos da operação abstrata que as subclasses redefinem para fornecer um comportamento concreto. As subclasses da aplicação definem os passos do algoritmo que verifica se o documento pode ser aberto (CanOpenDocument) e cria o Document (DoCreateDocument). As classes Document definem a etapa que lê o documento (DoRead). O método template também define uma operação que permite às subclasses de Application saberem quando o documento está para ser aberto (AboutToOpenDocument), no caso de elas terem esta preocupação.

Pela definição de alguns dos passos de um algoritmo usando operações abstratas, o método template fixa a sua ordem, mas deixa as subclasses de Application e Document variarem aqueles passos necessários para atender suas necessidades.

Aplicabilidade

O padrão Template Method pode ser usado:

- para implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar.

- quando o comportamento comum entre subclasses deve ser fatorado e concentrado numa classe comum para evitar a duplicação de código. Este é um bom exemplo de “refatorar para generalizar”. Primeiramente, você identifica as diferenças no código existente e então separa as diferenças em novas operações. Por fim, você substitui o código que apresentava as diferenças por um método template que chama uma destas novas operações.

- para controlar extensões de subclasses. Você pode definir um método template que chama operações “gancho” em pontos específicos, desta forma permitindo extensões somente nesses pontos.

Estrutura

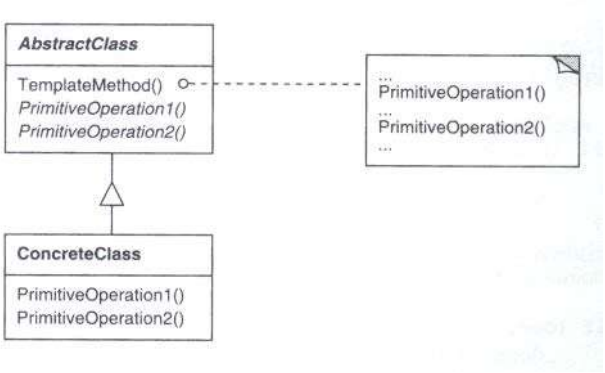


Figura 6.5 – Estrutura do padrão Template Method

Participantes

- **AbstractClass** (Application)
  - define operações primitivas abstratas que as subclasses concretas definem para implementar passos de um algoritmo.
  - implementa um método template que define o esqueleto de um algoritmo. O método template invoca operações primitivas, bem como operações definidas em AbstractClass ou ainda outros objetos.
- **ConcreteClass** (MvApplication)
  - implementa as operações primitivas para executarem os passos específicos do algoritmo da subclasse.

Colaborações

- ConcreteClass depende de AbstractClass para implementar os passos invariantes do algoritmo.

Conseqüências

Os métodos template são uma técnica fundamental para a reutilização de código. Eles são particularmente importantes em bibliotecas de classe porque são os meios para a fatoração dos comportamentos comuns nas bibliotecas de classes.

Os métodos template conduzem a uma estrutura de inversão de controle, algumas vezes chamada de “o princípio de Hollywood”, ou seja: “não nos chame, nós

chamaremos você". Isto se refere a como uma classe-mãe chama as operações de uma subclasse, e não o contrário.

Os métodos template chamam os seguintes tipos de operações:

- operações concretas (ou em ConcreteClass ou em classes clientes);
- operações concretas de AbstractClass (isto é, operações que são úteis para subclasses em geral);
- operações primitivas (isto é, operações abstratas);
- métodos fábrica; e
- operações gancho (hook operations) que fornecem comportamento por falta que subclasses podem estender se necessário. Uma operação gancho freqüentemente não executa nada por falta.

É importante para os métodos template especificarem quais operações são ganchos (podem ser redefinidas) e quais são operações abstratas (devem ser redefinidas). Para reutilizar uma classe abstrata efetivamente, os codificadores de subclasses devem compreender quais as operações projetadas para redefinição.

Uma subclasse pode estender o comportamento de urna operação de urna classe-mãe pela redefinição da operação e chamando a operação mãe explicitamente.

Infelizmente, é fácil esquecer de chamar a operação herdada. Podemos transformar tal operação num método template para dar à (classe) mãe controle sobre a maneira como as subclasses a estendem. A idéia é chamar uma operação gancho a partir de um método template na classe-mãe. Então, as subclasses podem substituir esta operação gancho.

Implementação e exemplo

```
public abstract class Template{
   public abstract String link(String texto, String url);
   public String transform(String texto) { return texto; }
   public String templateMethod(){
      String msg = “Endereço: ” + link(“Empresa”, www.empresa.com);
      Return transform;
   }
}

public class HTMLData extends Template{
   public String link(String texto, String url){
      return “<a href=’” + url + “’>” + texto + “</a>”;
   }
   public String transform(String texto){
      return texto.toLowerCase();
   }
}
```

Listagem 6.3 – Implementação do padrão Template Method

Usos conhecidos

Os métodos template são tão fundamentais que eles podem ser encontrados em quase todas as classes abstratas.

Padrões relacionados

Os Factory Methods são freqüentemente chamados por métodos template.

Strategy: Métodos template usam a herança para variar parte de um algoritmo. Estratégias usam a delegação para variar o algoritmo inteiro.

Capítulo 7

Conclusões

Este trabalho apresentou um estudo das melhores técnicas de padrões de projeto, bem como as caracteríticas dos três grupos de padrões, segundo GoF: de criação, de estrutura e de comportamento.

De uma forma geral, as vantagens encontradas com a utilização de *Design Patterns* no desenvolvimento de grandes sistemas foram:

a. Os padrões de projeto fornecem um vocabulário comum para comunicar, documentar e explorar alternativas de projeto, tornando um sistema menos complexo ao permitir falar sobre ele em um nível de abstração mais alto do que aquele de uma notação de projeto ou uma linguagem de programação.

b. Possibilita compreender os sistemas orientados a objetos existentes, visto que a maioria dos grandes sistemas orientados a objetos usam estes padrões.

c. Desenvolver software de melhor qualidade, visto que os padrões utilizam eficientemente técnicas de orientação a objetos, tais como polimorfismo, herança, composição e abstração para construir código reutilizável e eficiente.

d. O uso de padrões possibilita aos novatos atuarem mais como um especialista.

e. Ajudam a determinar como reorganizar um projeto, quando necessário desenvolvimento de novas funcionalidades de acordo com requisitos especificos.

Sob a ótica do mercado, pode-se dizer que a concorrência exige que as empresas de software sejam competitivas, o que faz com que a melhoria da qualidade, a diminuição de custos e o aumento da produtividade sejam prioridade. Para isso, a utilização de um processo disciplinado de desenvolvimento de software torna-se imprescindível e, *Design Patterns* é uma das tecnologias que permitem agregar qualidade ao processo de desenvolvimento.

Em uma próxima etapa seria interessante que fossem analisados outros padrões, bem como o desenvolvimento de um sistema orientado a objetos empregando *Design Patterns*, ou ainda o estudo de *Anti Patterns*, que diferentemente de *Design Patterns*, esta última busca analisar as soluções inadequadas para um projeto.

Referências Bibliográficas

[1] EUROPEAN CONFERENCE ON OBJECT ORIENTED-PROGRAMMING, Bologna. **Patterns Generate Architectures**, Bologna: 1994. Disponível em < http://www.nku.edu/~hauserj/paper1.pdf >. Acesso em: 11 out. 2004.

[2] COPLIEN, J. O. **Software Patterns**. New York: SIGS Books, 1996. ISBN 18-8484-250-X

[3] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.** Porto Alegre: Bookman, 2000. 367 p., 24 cm. ISBN 85-7307-610-0

[4] JANDL, P. J. **Mais Java.** São Paulo: Futura, 2003. 648 p., 24 cm. ISBN 85-7413-140-7

[5] PAGE-JONES, M. **Fundamentos do Desenho Orientado a Objetos com UML.** São Paulo: Makron Books, 2001. 462 p., 22cm. ISBN: 85-3461-243-9

[6] METSKER, S. J. **Padrões de Projeto em Java**. São Paulo: Bookman, 2004. 409 p., 24 cm. ISBN 85-363-0411-1

[7] Alexander, C. **A Pattern Language: Towns, Buildings, Construction.** New York: Oxford University Press, 1977. 1216p., 21 cm. ISBN 01-9501-919-9