

33 design-patterns aplicados com Java

Vinicius Senger e Kleber Xavier



Globalcode

Política de uso do material

Este material foi cedido para a comunidade de desenvolvimento de software brasileira. Não permitimos seu uso por empresas de treinamento para fornecer treinamento, esta é a única restrição! Queremos colaborar com a comunidade e sabemos que se o material for totalmente liberado, empresas sem nenhum domínio do assunto irão tentar vender desonestamente treinamentos associados a este conteúdo e sabemos que a comunidade vai sair prejudicada, além de toda nossa rede de parceiros no Brasil.

Esperamos que vocês aproveitem este conteúdo incrível que vem por mais de duas décadas ajudando diferentes tecnologias de desenvolvimento de software e linguagens orientadas a objetos a melhorarem a qualidade e forma das suas classes e objetos.

Este material foi escrito por Vinicius Senger e Kleber Xavier e está registrado na biblioteca nacional. Sua venda é proibida e ações severas serão tomadas em caso de pirataria ou uso indevido em publicações.

Agradecemos a todos a compreensão e já estamos trabalhando em um projeto para transformar este conteúdo em um wiki, assim poderemos receber diferentes colaborações de todos vocês, principalmente para termos exemplos de código com outros linguagens!



Obrigado novamente,
Vinicius Senger
Fundador Globalcode
Autor principal deste material

1 Introdução.....	9
1.1 Pré-requisitos do curso.....	9
2 Modelagem de software.....	13
2.1 Análise orientada a objetos.....	13
2.1.1 Classes e Objetos.....	13
2.1.2 Abstração.....	14
2.1.3 Encapsulamento.....	15
2.1.4 Construtores.....	16
2.1.5 Herança	16
2.1.6 Classes abstratas e Interfaces.....	16
2.1.7 Polimorfismo.....	17
2.1.8 Coesão.....	17
2.1.9 Acoplamento.....	19
2.1.10 Princípios básicos para projeto e análise orientados a objetos.....	19
2.2 Introdução a UML.....	21
2.2.1 UML e Metodologias de desenvolvimento de software.....	22
2.2.2 Principais diagramas da UML.....	23
2.3 Arquitetura física Vs. Arquitetura lógica.....	31
3 Introdução a design patterns.....	35
3.1 Conceitos fundamentais.....	35
3.2 Descrição de um pattern.....	38
3.3 Anti-pattern.....	39
4 Design patterns GoF.....	43
4.1 Singleton – Criação.....	45
4.1.1 Anti-pattern.....	45
4.1.2 Aplicando o design pattern.....	45
4.1.3 Considerações sobre inicialização.....	47
4.1.4 Herança com Singleton.....	48
4.1.5 Singleton Vs. Static.....	48
4.1.6 Singleton “si pero no mucho”	49
4.1.7 Estrutura e Participantes.....	50
4.2 Command – Comportamento.....	51
4.2.1 Anti-pattern.....	51
4.2.2 Aplicando o design pattern.....	52
4.2.3 Command e Factory.....	52
4.2.4 Command e sistemas de desfazer / refazer (undo / redo).....	53
4.2.5 Estrutura e Participantes.....	53
4.3 Factory Method – Criação.....	54
4.3.1 Anti-pattern.....	55
4.3.2 Aplicando o design pattern.....	56
4.3.3 Factory Method Estático.....	57
4.3.4 Factory Method Dinâmico.....	58

4.3.5 Factory e reuso de objetos	59
4.3.6 Factory Method e outros design patterns	59
4.3.7 Estrutura e Participantes	59
4.4 Abstract Factory – Criação	61
4.4.1 Anti-pattern	61
4.4.2 Aplicando o design pattern	61
4.4.3 Estrutura e Participantes	66
4.5 Laboratório 1	67
4.6 Template Method – Comportamento	68
4.6.1 Anti-pattern	68
4.6.2 Aplicando o design pattern	69
4.6.3 O padrão Template Method nas APIs do Java	71
4.6.4 Estrutura e Participantes	72
4.7 Visitor – Comportamento	73
4.7.1 Anti-pattern	73
4.7.2 Aplicando o design pattern	76
4.7.3 Estrutura e Participantes	80
4.8 Laboratório 2	81
4.9 Adapter - Estrutura	82
4.9.1 Anti-pattern	82
4.9.2 Aplicando o design pattern	85
4.9.3 Estrutura e Participantes	87
4.10 Observer – Comportamento	88
4.10.1 Anti-pattern	89
4.10.2 Aplicando o design pattern	90
4.10.3 Estrutura e Participantes	93
4.11 Laboratório 3	94
4.12 Mediator – Comportamento	95
4.12.1 Anti-pattern	95
4.12.2 Aplicando o design pattern	97
4.12.3 Mediator e user-interfaces	98
4.12.4 Estrutura e Participantes	99
4.13 Façade – Estrutura	100
4.13.1 Anti-pattern	100
4.13.2 Aplicando o design pattern	102
4.13.3 Façade Vs. Mediator	104
4.13.4 Façade na computação distribuída	104
4.13.5 Estrutura e Participantes	105
4.14 Laboratório 4	106
4.15 Composite – Estrutura	107
4.15.1 Anti-pattern	108
4.15.2 Aplicando o design pattern	111
4.15.3 Referência ao objeto pai	116

4.15.4 Estrutura e Participantes	117
4.16 Iterator – Comportamento	118
 4.16.1 Anti-pattern	119
 4.16.2 Aplicando o design pattern	120
 4.16.3 Estrutura e Participantes	123
4.17 State – Comportamento	124
 4.17.1 Anti-pattern	125
 4.17.2 Aplicando o design pattern	129
 4.17.3 Estrutura e Participantes	133
4.18 Laboratório 5	134
4.19 Memento – Comportamento	135
 4.19.1 Anti-pattern	135
 4.19.2 Aplicando o design pattern	136
 4.19.3 Estrutura e Participantes	141
4.20 Laboratório 6	142
4.21 Decorator – Estrutura	143
 4.21.1 Anti-pattern	144
 4.21.2 Aplicando o design pattern	145
 4.21.3 O padrão Decorator nas APIs do Java	148
 4.21.4 Estrutura e Participantes	150
4.22 Chain of responsibility – Comportamento	151
 4.22.1 Anti-pattern	152
 4.22.2 Aplicando o design pattern	155
 4.22.3 Estrutura e Participantes	160
4.23 Laboratório 7	161
4.24 Strategy – Comportamento	162
 4.24.1 Anti-pattern	163
 4.24.2 Aplicando o design-pattern	164
 4.24.3 O padrão Strategy nas APIs do Java	167
 4.24.4 Estrutura e Participantes	169
4.25 Proxy – Estrutura	170
 4.25.1 Anti-pattern	171
 4.25.2 Aplicando o design-pattern	173
 4.25.3 Proxy e Decorator	176
 4.25.4 Estrutura e Participantes	177
4.26 Builder – Criação	178
 4.26.1 Anti-pattern	179
 4.26.2 Aplicando o design pattern	181
 4.26.3 Estrutura e Participantes	185
4.27 Prototype – Criação	186
 4.27.1 Anti-pattern	186
 4.27.2 Aplicando o design pattern	188
 4.27.3 Estrutura e Participantes	189

4.28 Laboratório 8.....	190
4.29 Bridge – Estrutura.....	191
 4.29.1 Anti-pattern.....	192
 4.29.2 Aplicando o design pattern.....	194
 4.29.3 Estrutura e Participantes.....	195
4.30 Interpreter – Comportamento.....	196
 4.30.1 Estrutura e Participantes.....	197
4.31 Flyweight – Estrutura.....	198
 4.31.1 Anti-pattern.....	199
 4.31.2 Aplicando o design-pattern.....	199
 4.31.3 O padrão Flyweight nas APIs do Java.....	200
 4.31.4 Estrutura e Participantes.....	201
5 Design patterns Java EE Blueprints	204
5.1 Service Locator.....	205
 5.1.1 Anti-pattern.....	205
 5.1.2 Aplicando o design-pattern.....	206
5.2 Data Access Object.....	209
 5.2.1 Anti-pattern.....	210
 5.2.2 Aplicando o design pattern.....	211
5.3 Transfer Object / Value Object.....	214
5.4 Laboratório 9.....	218
5.5 Front Controller.....	219
 5.5.1 Anti-pattern.....	220
 5.5.2 Aplicando o design pattern.....	221
5.6 Model-view-controller.....	223
 5.6.1 A divisão da camada intermediária.....	224
 5.6.2 Anti-pattern.....	225
 5.6.3 Aplicando o design pattern.....	226
5.7 Composite View.....	228
5.8 Business Delegate.....	230
5.9 Application Controller.....	235
5.10 Intercepting Filter.....	237
5.11 View Helper.....	240
5.12 Laboratório 10.....	245

CAPÍTULO

1

Introdução

Para uso não comercial

Para uso não comercial

Anotações

1 Introdução

Este curso apresenta 33 diferentes design patterns, de diferentes famílias, com uma abordagem técnica e educacional totalmente diferenciada. Ao término deste treinamento você estará apto a:

- Conhecer todos os fundamentos sobre design pattern, famílias de patterns e padrões de descrição;
- Entender diferentes cenários de desenvolvimento onde podemos aplicar patterns;
- Modelar aplicativos utilizando o máximo da orientação a objetos;
- Aplicar cada um dos 23 patterns GoF com Java;
- Aplicar 10 patterns Java Enterprise;

1.1 Pré-requisitos do curso

- Lógica de programação;
- Orientação a objetos (mínimo 1 ano experiência);
- Conhecer plenamente a linguagem Java;
- Conhecimento UML desejável mas não obrigatório;

Anotações

Para uso não comercial

Anotações

CAPÍTULO

2

Modelagem de Software

Análise Orientada a Objetos

Introdução a UML

Orientação a Aspecto

Arquitetura Física vs. Arquitetura Lógica

Para uso não comercial

Anotações

2 Modelagem de software

Na construção de softwares é comum construir modelos que expliquem as características e o comportamento do sistema. Estes modelos auxiliam na identificação das características e funcionalidades do software e no planejamento de sua construção.

Freqüentemente estes modelos utilizam alguma representação gráfica que facilita o entendimento dos artefatos utilizados e seus relacionamentos. Uma das técnicas mais utilizadas atualmente é a orientação a objetos, para a análise e criação de modelos de software. E a linguagem mais utilizada para a modelagem gráfica orientada a objetos é a UML.

2.1 Análise orientada a objetos

Vamos apresentar a seguir os conceitos básicos utilizados na análise orientada a objetos.

2.1.1 Classes e Objetos

Uma classe é um tipo definido pelo usuário que possui especificações (características e comportamentos) que o identifiquem. De uma maneira mais objetiva, podemos dizer que a classe é um molde que será usado para construir objetos que representam elementos da vida real.

Classe = Características + Comportamentos

Tal molde é gerado através da observação e agrupamento de elementos que possuem as mesmas características e comportamentos sob uma mesma denominação. Exemplificando, em uma loja online (ambiente), identificamos um elemento chamado Produto (classe), que possui um conjunto de características tal como a Identificação e o Preço (atributos).

Produto	
Identificação	Preço

Cada Produto poderia ser materializado com as informações abaixo:

Anotações

Produto	
Identificação	Preço
Core Java	R\$ 500,00

Produto	
Identificação	Preço
Camiseta	R\$ 15,00

Cada materialização da classe constitui um **objeto**.

No entanto, os **Atributos** nem sempre são úteis individualmente. Convém definirmos algumas ações e comportamentos que esse elemento possa executar dentro do ambiente modelado usando os atributos existentes na Classe.

Esses comportamentos são denominados **Métodos** e definem as ações previstas para essa classe. Ainda no exemplo da loja, poderíamos atribuir aos Produtos algumas funcionalidades tais como aumentar/diminuir o preço e alterar a Identificação. Note que normalmente as ações estão associadas aos atributos da classe (Preço, Identificação).

Produto	
Identificação	Preço
Aumentar preço	
Diminuir preço	
Alterar identificação	

Podemos então afirmar que um objeto é uma entidade que possui determinadas responsabilidades. Estas responsabilidades são definidas através da escrita de uma classe, que define os dados (variáveis associadas com os objetos) e métodos (as funções associadas com os objetos).

2.1.2 Abstração

Como uma classe pode representar qualquer coisa em qualquer ambiente, é sugerido que a modelagem foque nos objetivos principais do negócio. Isso evita que o sistema seja muito grande e consequentemente de difícil manutenção e compreensão. A análise focada é denominada abstração.

No mundo real, podemos utilizar um produto que possui informações de garantia, matéria-prima, etc. Porém, conforme o tipo de aplicativo, pode ser ou não interessante colocarmos tais informações na definição do objeto.

Anotações

No mundo real podemos trocar produtos entre pessoas, entretanto, num cenário de e-business, não precisamos implementar este comportamento ao definirmos o Produto.

Exemplo: abstração de Data

Qual é a estrutura básica de uma Data?

- dia
- mês
- ano

Qual o comportamento ou operações relativos a esta entidade?

- Transformar o número do mês em um texto com o nome do mês (Ex: 1 : Janeiro)
- Transformar o número do dia em nome do dia (Ex: 24:Segunda, 25:Terça)..
- Devemos saber se o ano é bissexto.

Exemplo: abstração de Produto

Quais são as características básicas (estrutura) de um produto?

- id
- preço
- nome

Qual o comportamento desta entidade no mundo real?

- Aumentar o preço
- Aplicar desconto
- Alterar o nome

2.1.3 Encapsulamento

Como os objetos são responsáveis por eles mesmos, não é necessário expor todos os seus detalhes e dados para outros objetos. Damos o nome de encapsulamento quando escondemos para acesso externo elementos de interesse exclusivo do próprio objeto.

Tipicamente limitamos o acesso direto aos dados do objeto, mas o encapsulamento pode ser considerado de uma forma mais ampla como qualquer ocultamento.

Algumas vantagens que podem ser obtidas através da utilização de encapsulamento são:

Anotações



- Validar valores a serem atribuídos a cada atributo.
- Definir quais/quando os atributos podem ser alterados.
- Fazer um log das modificações dos atributos
- Permitir que mudanças na estrutura interna dos objetos sejam transparentes para outros objetos.

Quanto mais nossos objetos forem responsáveis por seus próprios comportamentos, menos responsabilidade sobra para programas de controle.

2.1.4 Construtores

Construtores são métodos especiais chamados automaticamente quando um objeto é criado. Seu objetivo é gerenciar a inicialização do objeto. Neste método tipicamente definimos valores iniciais para os dados, configuramos relacionamentos com outros objetos, e todas as demais atividades que são necessárias para a construção de um objeto bem formado.

2.1.5 Herança

Herança é um princípio da orientação a objetos que permite a criação de classes como especializações de classes já existentes. Denominamos a classe base ou mais genérica de super-classe e a especialização de sub-classe. A sub-classe herda os dados e comportamento da super-classe, implementando somente o que for específico. Isso ajuda a diminuir a duplicação de código e permite um reaproveitamento dos dados e comportamento de classes já existentes.

2.1.6 Classes abstratas e Interfaces

Classes abstratas e interfaces permitem a definição de tipos genéricos, nos quais definimos o comportamento esperado para um conjunto de classes derivadas. Esta definição é feita através da criação de métodos sem a implementação do código.

Estes métodos são denominados métodos abstratos e possuem somente a definição dos parâmetros de entrada e valores de retorno esperados. Cabe a implementações específicas ou classes derivadas criarem a implementação do código. Com isso podemos definir um “contrato” esperado para uma família de classes sem nos preocuparmos com os detalhes de implementação.

No caso de classes abstratas podemos também definir dados que serão herdados pelas sub-classes.

Anotações

2.1.7 Polimorfismo

O polimorfismo é um recurso poderoso da orientação a objetos que permite a utilização de uma única referência para diferentes especializações da mesma classe, obtendo dinamicamente o comportamento adequado. Podemos utilizá-lo das seguintes formas:

- Definimos um tipo base (classe ou interface) e criamos classes derivadas, por herança ou por implementação de interface e assim temos várias formas para um tipo base.
- Utilizamos uma declaração de variável de um tipo-base para manipular (via cast up) um objeto de qualquer uma de seus tipos derivados.



Onde uma super-classe é esperada podemos utilizar uma instância de uma sub-classe.

Onde uma interface é esperada podemos utilizar uma instância de uma classe implementadora.

Benefícios

As duas principais formas de se beneficiar do polimorfismo na POO são:

- parâmetros e retornos polimórficos (late-binding)
- coleções heterogêneas

2.1.8 Coesão

Coesão em orientação a objetos é o grau de direcionamento de uma classe para uma única e bem definida responsabilidade, e traz os seguintes benefícios:

- Manutenibilidade.
- Reaproveitamento.

Para exemplificar, imagine os seguintes requisitos para construção de um aplicativo:

- Arquivos de textos são gerados por um servidor de dados com informações cadastrais de clientes, um registro por linha
- Através de uma interface gráfica o usuário seleciona um arquivo fornecido para processamento
- O aplicativo deve ler os registros do arquivo selecionado e apresentá-los na interface gráfica classificando-os por município e por nome de cliente.

Anotações

É possível atender aos requisitos e construir este aplicativo com uma única classe Java. Mas será esta a melhor abordagem a médio e longo prazo? Considere as seguintes solicitações de melhoria, após algum período de utilização do aplicativo:

- O usuário poderá escolher através da interface gráfica outros critérios de classificação.
- O servidor de dados passa a fornecer o código de município ao invés do nome de município nos registros dos arquivos de clientes gerados. Um arquivo à parte é disponibilizado pelo mesmo servidor com a relação de códigos e nomes de municípios.
- Os resultados apresentados na interface gráfica poderão ser encaminhados para um relatório impresso.
- Um novo aplicativo é solicitado, e deve se valer dos mesmos arquivos de texto gerados pelo servidor de dados, e enviar automaticamente e semanalmente relatórios por e-mail para cada gerente de filial com os clientes de sua região classificados por município e por nome.

Na melhor e mais rara das hipóteses essas solicitações de melhoria serão feitas de uma só vez só. No dia a dia podemos esperar que essas solicitações serão feitas individualmente e em datas diferentes ! Qual o esforço de manutenção cada vez que uma nova solicitação é apresentada? E quanto ao novo aplicativo, será que é o caso de uma construção “à partir do zero”?



Valendo-se da coesão podemos minimizar o impacto das solicitações adicionais.

Vamos planejar a versão original do aplicativo de forma coesa:

- Uma classe de interface gráfica responsável apenas pela seleção dos arquivos e pela exibição dos registros já classificados: `Tela.java`
- Uma classe responsável pela leitura dos registros contidos no arquivo texto selecionado: `Leitor.java`
- Uma classe responsável pela classificação dos registros: `Classificador.java`



Graças à coesão cada solicitação de melhoria demandará um esforço de manutenção pontual, ou a criação de novas classes (coesas) se necessário:

Melhoria	Impacto	<code>Tela.java</code>	<code>Leitor.java</code>	<code>Classificador.java</code>	Novas Classes
Outros critérios de classificação	lista com opções de classificação			parametrização do critério de classificação	

Anotações

Código do município ao invés do nome		leitura do arquivo de municípios e transformação de código em nome		
Encaminhamento para impressora	botão para impressão			Impressao.java
Novo aplicativo de envio de relatórios por e-mail		reaproveitamento total	reaproveitamento, com filtro de região	Temporizador.java EnviaEmail.java

Coesão é um conceito **independente** da linguagem de modelagem e da metodologia de desenvolvimento.

2.1.9 Acoplamento

Em orientação a objetos Acoplamento é o grau de conhecimento requerido por uma classe sobre seus membros internos (atributos, métodos, ...) para que possa ser utilizada. O encapsulamento é capaz de diminuir o acoplamento entre classes.

2.1.10 Princípios básicos para projeto e análise orientados a objetos

Algumas técnicas ou diretrizes podem ser utilizadas para escrever código de maneira a torná-lo mais flexível e facilitar a sua manutenção. Estes princípios básicos podem ser observados nos padrões de projeto. Apresentamos alguns destes princípios a seguir:

- Princípio Aberto/Fechado (OCP - Open/Closed Principle): Devemos projetar nossos sistemas de maneira a permitir mudanças (aberto para mudanças) sem que para isso seja necessário alterar o código já existente (fechado para alterações). Ou seja, classes devem ser abertas para extensão e fechadas para modificação.
- Não se repita (DRY - Don't Repeat Yourself): Devemos evitar código duplicado, abstraindo elementos que são comuns e colocando-os em um único lugar.
- Princípio da única responsabilidade (SRP - Single Responsibility Principle): Cada objeto no seu sistema deve ter uma única responsabilidade e todos os serviços deste objeto devem estar relacionados a esta responsabilidade. Este princípio está intimamente ligado a coesão.

Anotações

- Princípio de substituição de Liskov: subtipos devem ser substituíveis por seus tipos base. Está relacionado ao uso correto de herança. Basicamente, define que os métodos herdados pela sub-classe devem fazer sentido para ela.

Para uso não comercial!

Anotações

2.2 Introdução a UML

A UML é uma linguagem de modelagem de software

A UML consiste de um número de elementos gráficos combinados para formar diagramas. Pelo fato de ser uma linguagem, a UML tem regras para a combinação desses elementos.

O propósito dos diagramas é apresentar múltiplas visões de um sistema, o conjunto dessas múltiplas visões é chamado de modelo. É importante deixar claro que um modelo UML diz o que um sistema tem que fazer, mas não como implementá-lo. A seguir temos uma breve descrição dos mais comuns diagramas UML e o conceito que eles representam.

A UML unifica os métodos de Grady Booch, James Rumbaugh e Ivar Jacobson, que trabalhavam em diferentes metodologias.

Nos anos 80 e no começo dos anos 90 estes três trabalhavam em diferentes empresas, cada um desenvolvendo sua própria metodologia para análise e projeto de softwares orientados a objetos. Na metade dos anos 90 eles resolveram juntar seu trabalho, ou seja, decidiram pegar o que havia de bom em cada uma das metodologias e criar uma nova.

A UML é padronizada pela OMG (Object Management Group)

Em 1997 o consórcio criado pelas grandes empresas do mercado, tais como, Oracle, HP, Texas Instruments, Rational, produziram a versão 1.0 da UML e submeteram para aprovação do OMG(Object Management Group). Mais tarde uma outra versão, a 1.1, foi submetida para OMG, que adotou a UML como linguagem de modelagem de software padrão. A partir de então o OMG passou a manter e produzir novas versões da UML.

Anotações

2.2.1 UML e Metodologias de desenvolvimento de software

A UML é uma linguagem para modelagem de sistemas. A UML é a base para algumas metodologias de desenvolvimento de software, como por exemplo RUP (Rational Unified Process).

Uma metodologia define as etapas que devem ser seguidas para desenvolvimento de um sistema, e quando a metodologia utiliza UML geralmente indica em qual fase do desenvolvimento cada diagrama deve ser utilizado. Desta forma, a UML não define quando cada diagrama deve ser utilizado, e sim qual a visão do sistema ele oferece.

Geralmente é inviável desenhar todos os diagramas para todo o projeto, por isto, selecionamos os diagramas mais interessantes para cada caso.

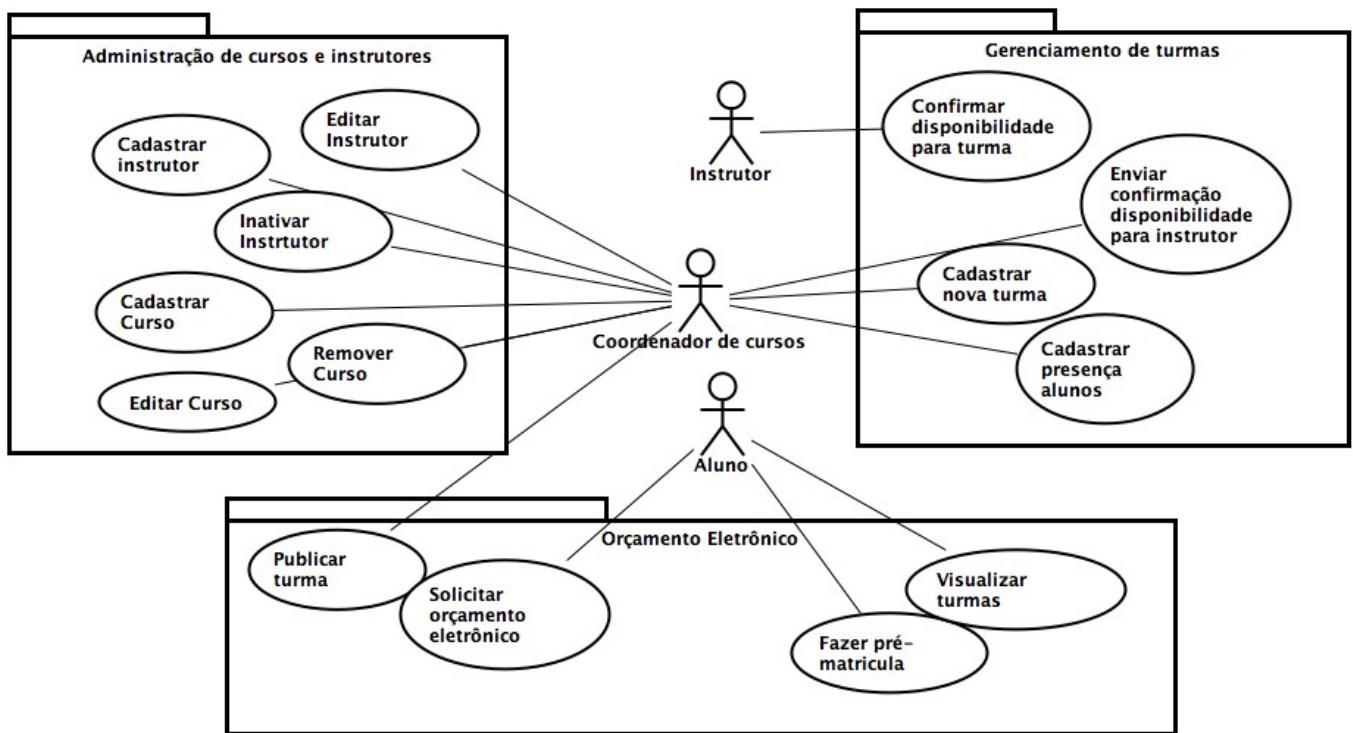
Esta fora do nosso escopo abordar uma metodologia especificamente, no entanto vamos comentar quando os diagramas são frequentemente utilizados independente de metodologia.

Anotações

2.2.2 Principais diagramas da UML

2.2.2.1 Diagrama de casos de uso

Representamos neste diagrama os casos de uso de um sistema, ou seja, as funcionalidades de um sistema, por isto, este diagrama geralmente é utilizado no inicio de um projeto, na fase de análise de requisitos.



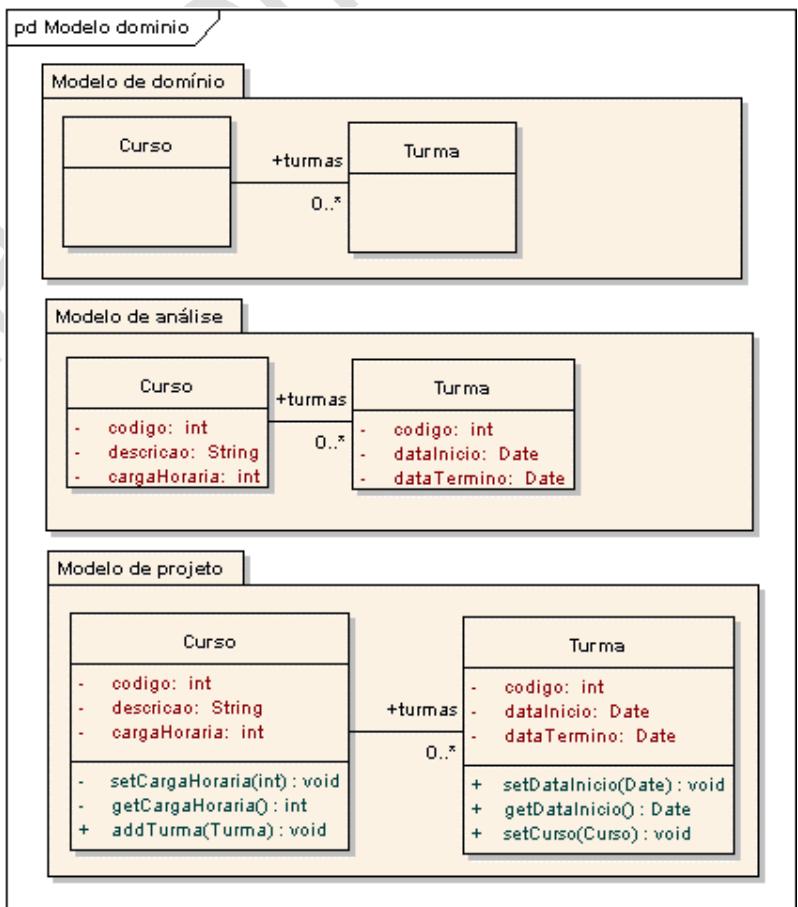
Anotações

2.2.2.2 Diagrama de classes

O diagrama de classes é utilizado para representar a estrutura do sistema, ou seja, as classes do sistema. Geralmente este diagrama é utilizado durante todo o projeto. No inicio, na fase de análise de requisitos utilizamos este diagrama para fazer a modelagem conceitual do sistema, representamos as entidades mais evidentes e o relacionamento entre elas, sem a preocupação com detalhes de implementação.

Na fase de implementação é feito o refinamento do diagrama de classes conceitual criado na análise de requisitos criando o chamado diagrama de classes de implementação. Este diagrama é atualizado enquanto houverem atualizações no sistema.

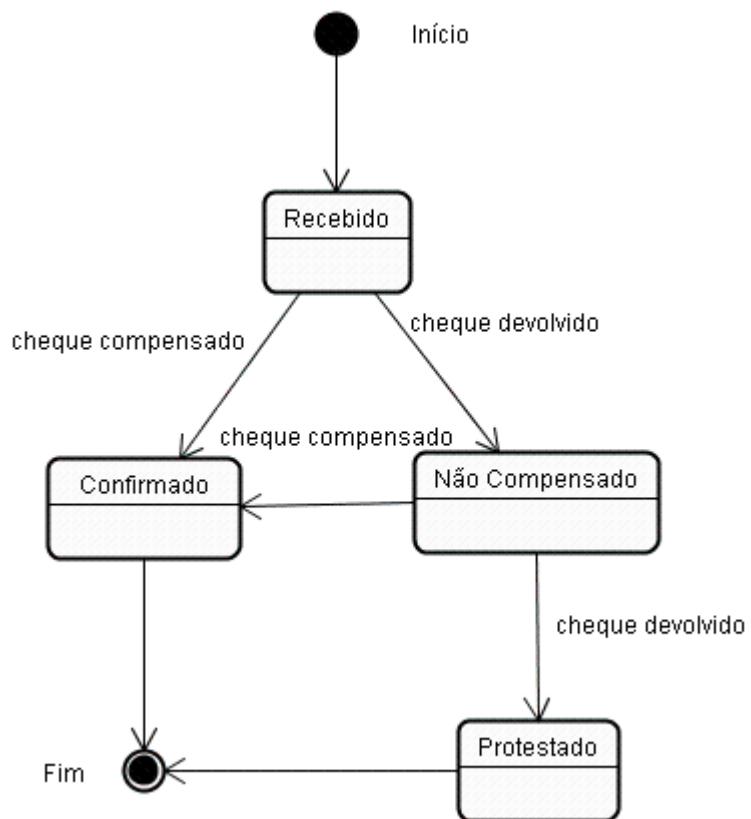
O diagrama de pacotes, que muitas literaturas pregam, é uma maneira de se organizar as classes conceitualmente e pode ser apresentado como se fosse parte do diagrama de classes.



Anotações

2.2.2.3 Diagrama de estados

O diagrama de estados é uma evolução dos diagramas utilizados na programação procedural chamados Máquina de estados, utilizado para representar a mudança de estado de determinada entidade de um sistema. Se imaginarmos a entidade pagamento em qualquer sistema que efetue uma venda, podemos imaginar estados para um pagamento, pagamento recebido, pagamento compensado, pagamento não compensado (Cheque devolvido) entre outros. Para entidades que possuem estados bem definidos é interessante desenharmos o diagrama de estados.

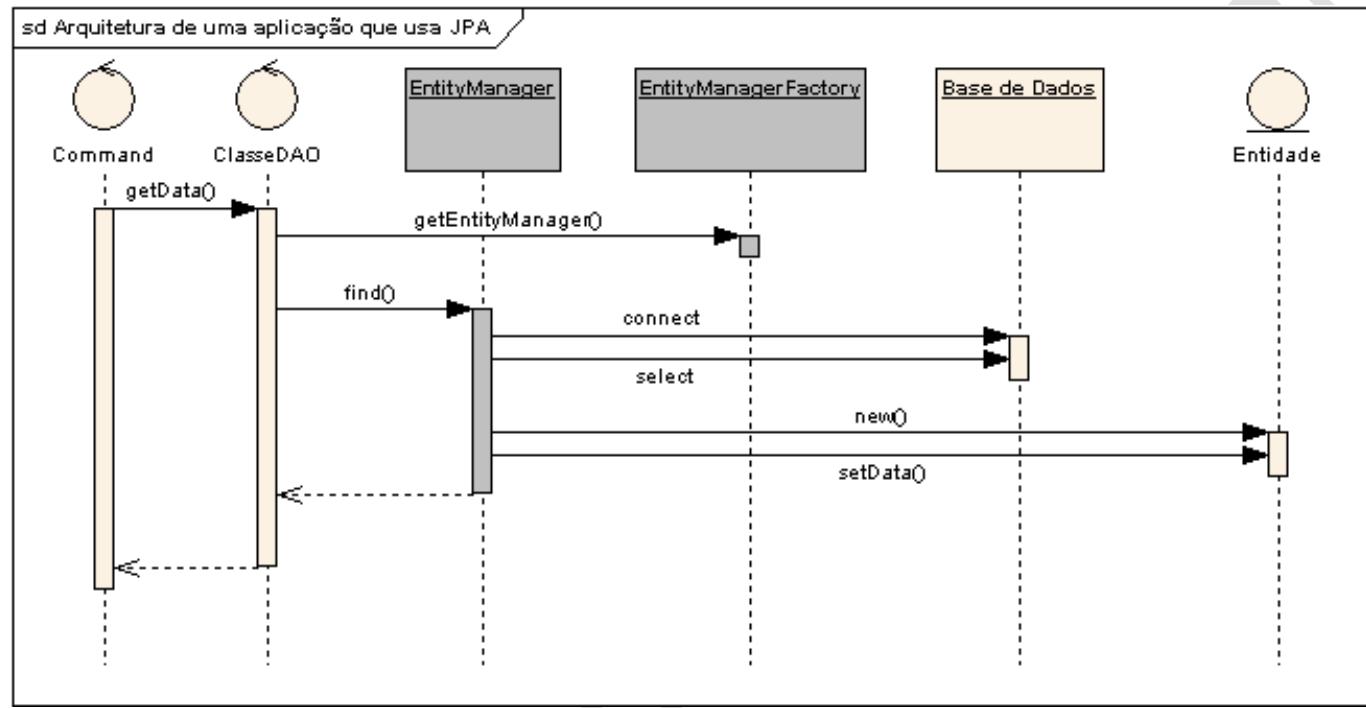


Anotações

2.2.2.4 Diagrama de seqüência

O diagrama de seqüência demonstra quais os passos, em uma linha do tempo, para que determinada tarefa seja executada. Em um diagrama mais detalhado, estes passos podem ser chamadas a métodos.

Utilizamos este diagrama para mostrar a arquitetura de um sistema, design patterns implementados, processos complicados entre outros.

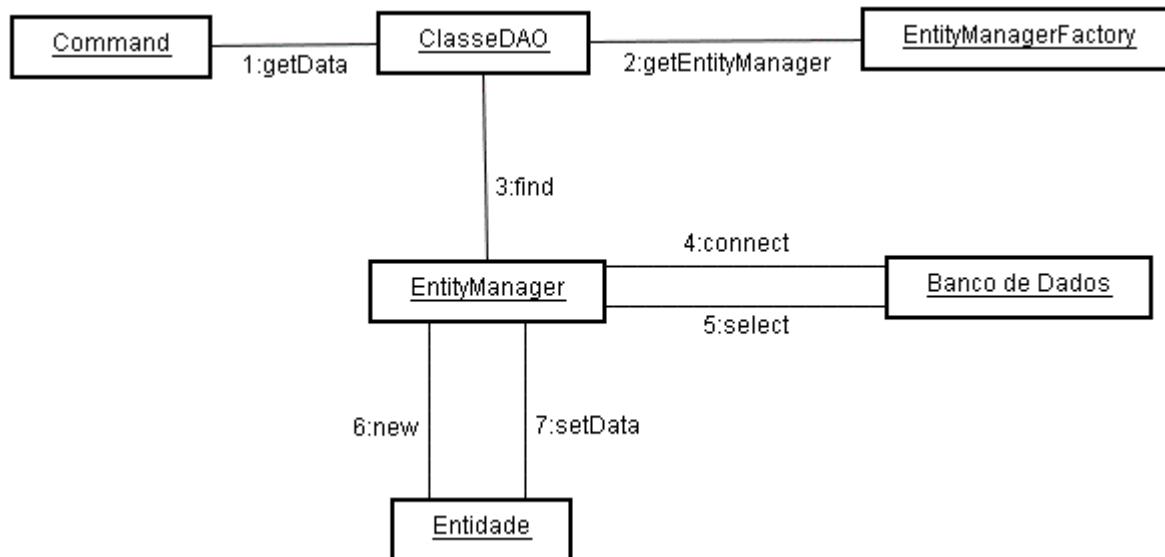


Anotações

2.2.2.5 Diagrama de Colaboração

O diagrama de colaboração é uma visão semelhante daquela apresentada pelo diagrama de seqüência, contudo sem a preocupação do espaço temporal.

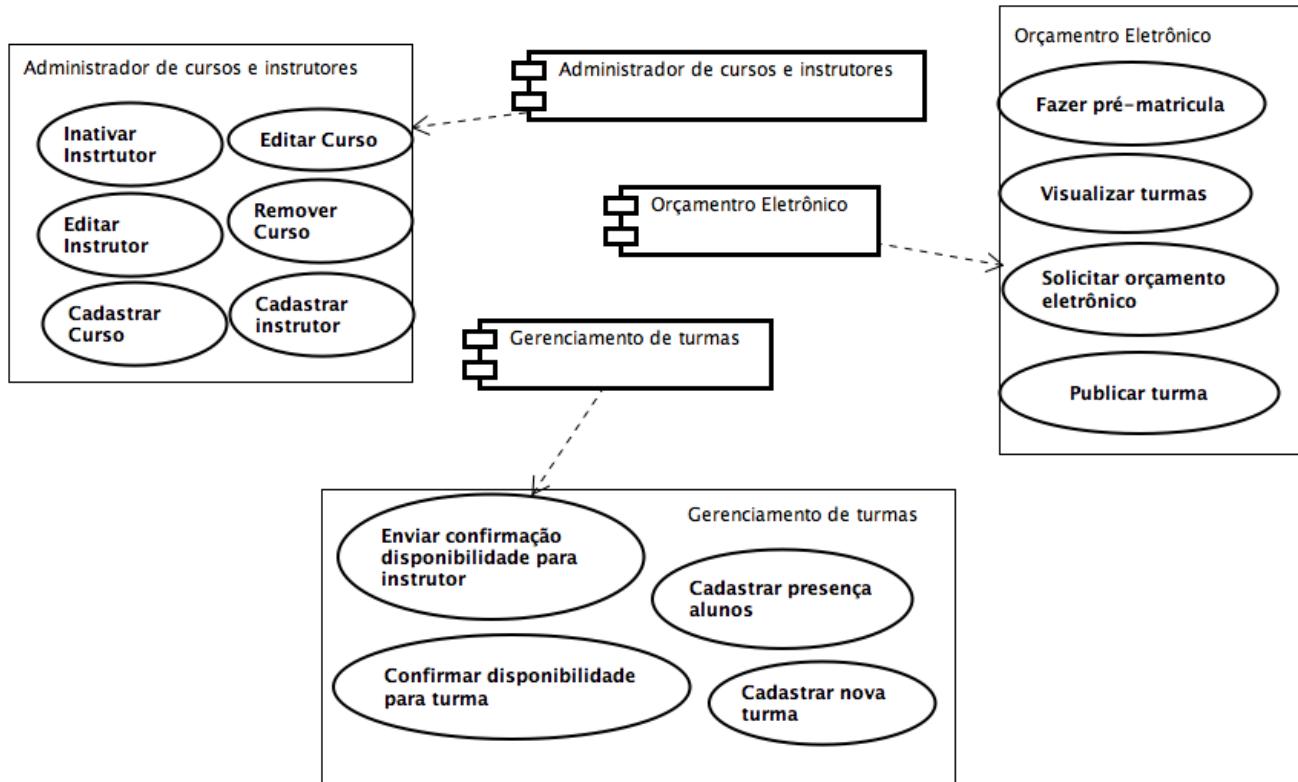
Permite uma visão mais organizada das classes e suas colaborações sem a necessidade de se compreender esse relacionamento ao longo do tempo.



Anotações

2.2.2.6 Diagrama de componentes

O diagrama de componentes é utilizado para mostrar os componentes do sistema. Nele pode-se detalhar as classes que compõe o componente e o relacionamento entre os componentes.

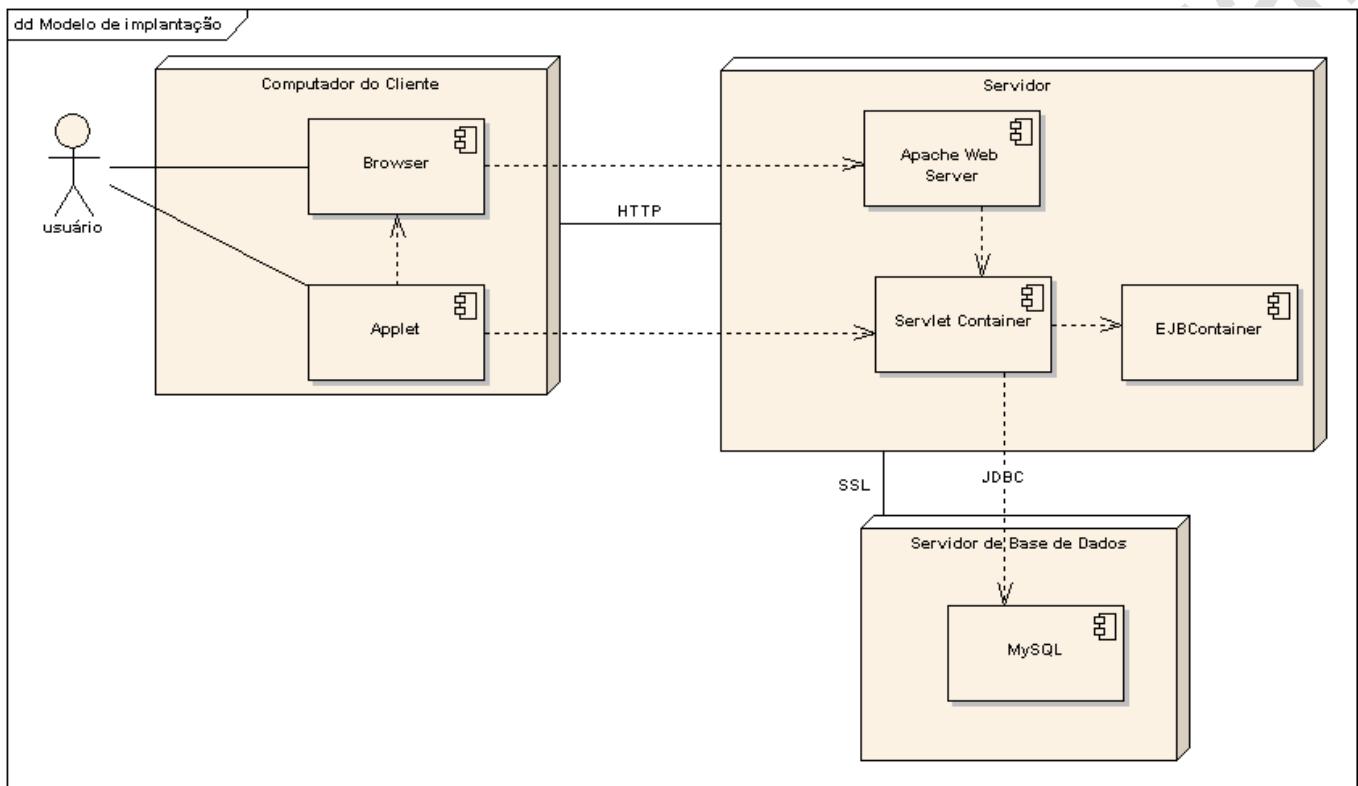


Anotações

2.2.2.7 Diagrama de implantação / deployment

O diagrama de implantação ou deployment é utilizado para detalhar a estrutura utilizada para implantar o sistema dentro de um container.

Este diagrama é útil para que os administradores dos servidores J2EE saibam o que fazer com as classes e componentes produzidos pela equipe de desenvolvimento.

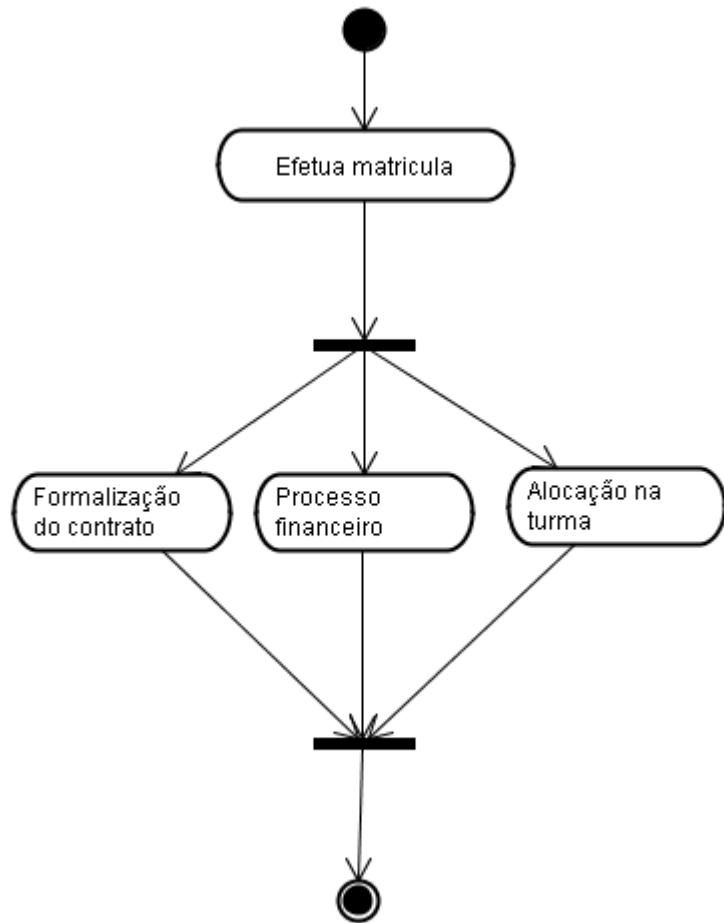


Anotações

2.2.2.8 Diagrama de atividades

O diagrama de atividades é muito parecido com o fluxograma comumente utilizado na programação procedural. Na verdade ele é uma evolução deste diagrama.

Este diagrama é bastante utilizado na fase de análise de requisitos pois é útil para modelar processos complexos como regras de negócio. Este diagrama também pode ser utilizado para modelagem de processos de negócios não relacionados ao software.

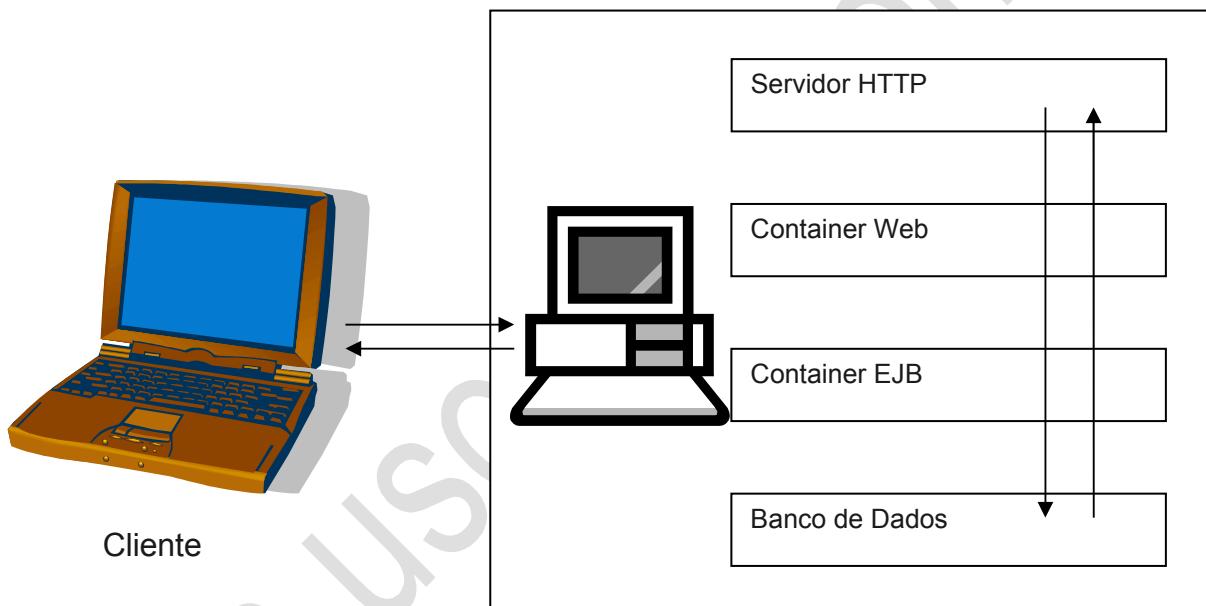


Anotações

2.3 Arquitetura física Vs. Arquitetura lógica

Em sistemas com múltiplos servidores, estes podem estar todos fisicamente na mesma máquina, apesar de existir a divisão conceitual de responsabilidades e processamento, ou podem estar separados fisicamente, cada um em uma máquina separada.

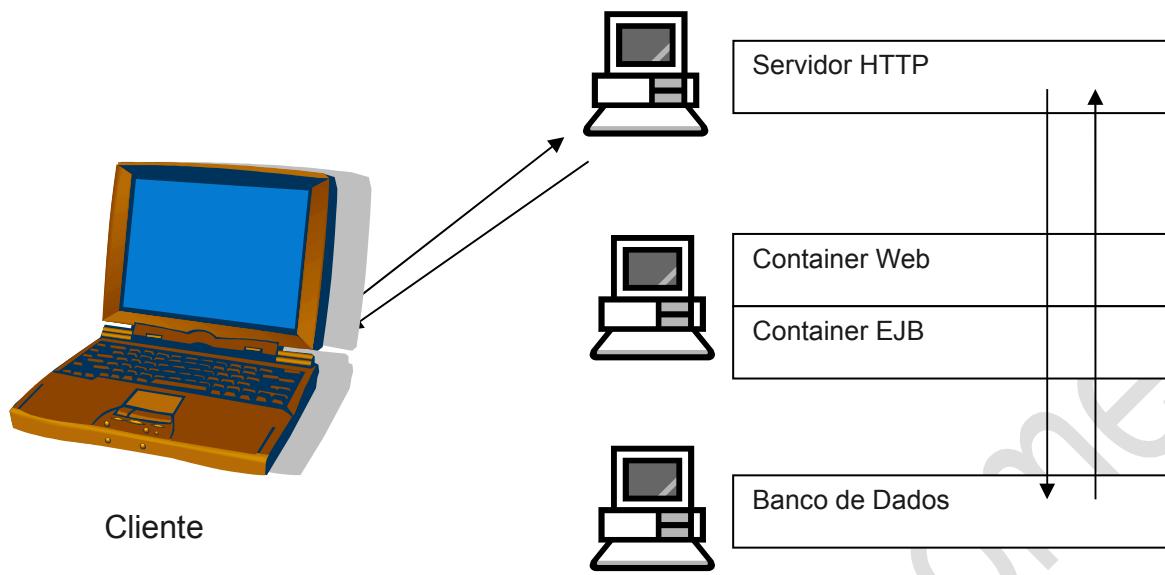
Muitas vezes não é necessário inicialmente separar os servidores fisicamente, mas a divisão conceitual garante a escalabilidade e a flexibilidade da arquitetura, permitindo e facilitando a separação física dos servidores no momento em que seja necessário.



Todos os servidores estão fisicamente na mesma máquina

Anotações

Neste exemplo os servidores HTTP e de banco de dados estão fisicamente separados, do servidor que contém os Containers Java (Web e EJB). Obviamente seria possível separar os containers Java fisicamente.



Os servidores estão separados conforme a necessidade, neste caso o Container Web e Container EJB estão na mesma máquina.

Anotações

CAPÍTULO

3

Introdução a Design Patterns

Conceitos fundamentais
Descrição de um pattern
Anti-pattern

Para uso não comercial

Anotações

3 Introdução a design patterns

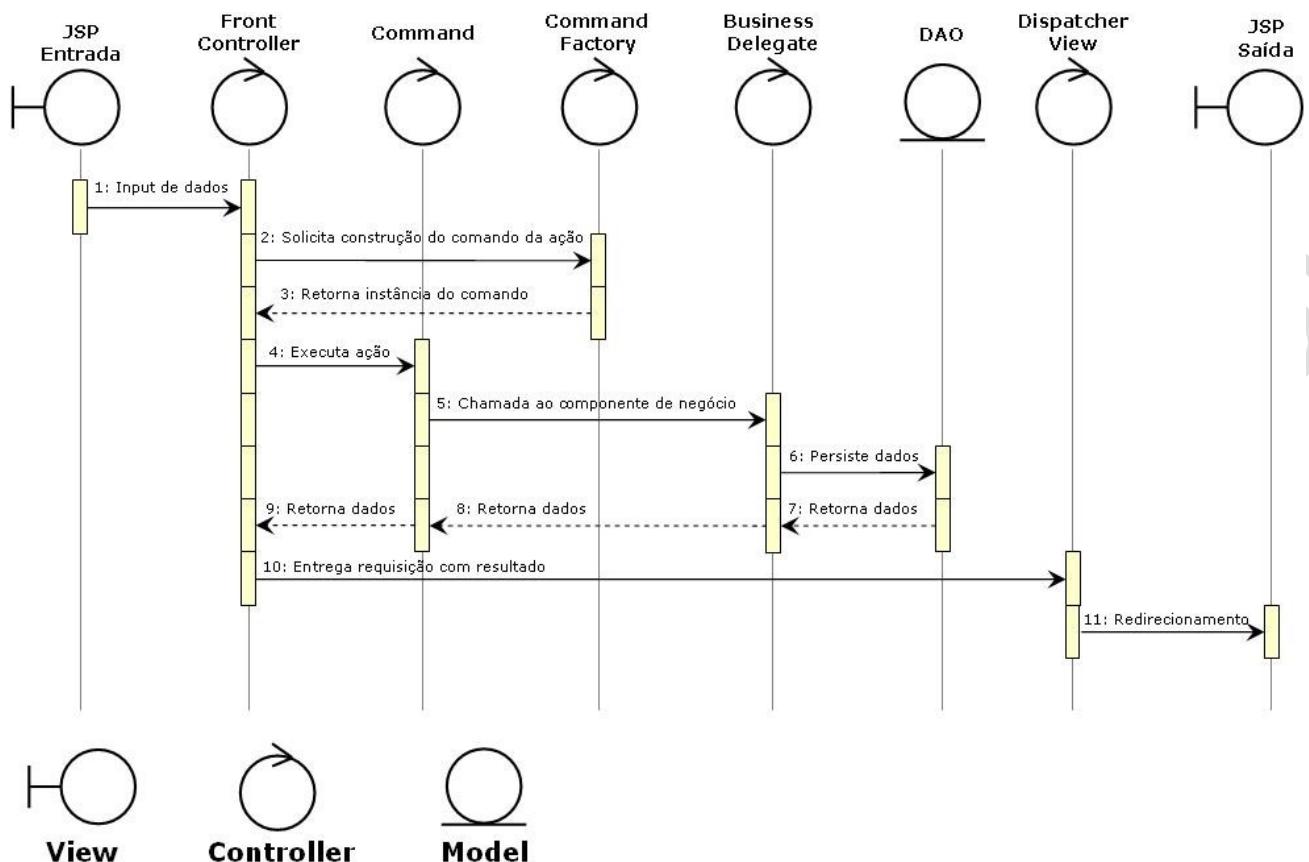
3.1 Conceitos fundamentais

Um design pattern é uma técnica de modelagem de classes e objetos que resolve um problema comum a diversos tipos de aplicativos. Um design pattern representa uma forma de compartilhar conhecimentos sobre programação orientada a objetos além de otimizar a comunicação entre desenvolvedores através de uma terminologia padronizada.

Design patterns não se aplicam exclusivamente ao Java, design patterns são utilizados em diversas linguagens orientadas a objetos, como C++, Smalltalk e até mesmo Microsoft C#. Porém, o que notamos é que, com a forte adoção da linguagem Java, o conhecimento sobre design patterns tem se popularizado cada vez mais.

Podemos de forma simples entender por design patterns, como uma maneira de organizar sua(s) classe(s) diante determinada circunstância. A seguir, demonstramos um diagrama de seqüência exibindo uma arquitetura de três camadas (expansível para outras), utilizando os padrões que formam uma aplicação Web de boa qualidade. Trata-se apenas de uma visão macro com os principais padrões e suas interações.

Anotações



View **Controller** **Model**

Os estereótipos de classes apresentados são criação da **Rational**, empresa pioneira na divulgação e disponibilização da **Unified Modeling Language – UML**, adquirida pela IBM. Esse tipo de diagramação é bastante interessante porque, ao mesmo tempo em que pensamos nos componentes que irão compor a aplicação, também já os dividimos em suas respectivas camadas e responsabilidades.

Conforme observamos no diagrama, as responsabilidades do software estão bem distribuídas entre os diversos objetos (colunas) do diagrama. A idéia de atribuirmos um número finito de responsabilidades por objetos traz um número grande de benefícios:

- Simplicidade e facilidade de entendimento da classe;
- Maior reusabilidade de código;
- Facilidade em estender o aplicativo;
- Facilidade em depurar erros;

Podemos citar duas principais categorias de design patterns popularmente utilizados com Java e J2EE:

1. GoF patterns;
2. J2EE design patterns.

Anotações

Os design patterns GoF (Gang of Four) são mais antigos e utilizados com Smalltalk, C++, Java, entre outras linguagens. Singleton, Factory Method, Memento, Proxy, Adapter, Iterator, Observer e muitos outros fazem parte dos patterns GoF.

Os patterns J2EE são mais focados em problemas encontrados com os padrões de arquitetura e forma de componentização imposta pela plataforma Java 2 Enterprise Edition.

De uma maneira geral podemos afirmar que os diversos design patterns, apesar de servirem para resolver problemas distintos, acabam tendo como objetivo comum, a busca por:

- Código menos extenso possível;
- Código mais legível possível;
- Código mais flexível possível;
- Divisão de responsabilidades entre objetos;
- Aplicação do conceito de reutilização de objetos.

Como consequência, as vantagens em utilizá-los consistem em:

- Possibilitar compartilhamento de arquiteturas entre projetos;
- Padronizar;
- Elaborar documentação implícita;
- Permitir alta manutenibilidade com baixo impacto em alterações de regras de negócio;
- Impor implicitamente melhores práticas (best practices) da Orientação a Objetos;
- Reduzir o custo total de propriedade do software;
- Facilitar o aprendizado da tecnologia e do projeto;
- Possibilitar compartilhamento de conhecimento entre desenvolvedores;
- Facilitar a comunicação entre os desenvolvedores.

Anotações

3.2 Descrição de um pattern

Originalmente o conceito de Design Patterns surgiu com o arquiteto Christopher Alexander, no livro "A Pattern Language: Towns, Buildings, Construction" de 1977. Posteriormente, o mesmo conceito foi reaproveitado para modelagem de software.

Segundo Alexander a descrição de um pattern deve envolver quatro itens:

- o nome do pattern;
- o propósito do pattern, ou seja, o problema que ele resolve;
- a solução utilizada, ou seja, como o pattern resolve o problema;
- as restrições e forças que devem ser consideradas na aplicação do pattern.

O livro de padrões de projeto do GOF segue uma abordagem mais formal, que engloba os conceitos iniciais de Alexander e acrescenta outros elementos. Inicialmente foi criada uma classificação de patterns, de acordo com sua finalidade:

- **padrões de criação:** preocupam-se com o processo de criação de objetos;
- **padrões estruturais:** lidam com a composição de classes ou objetos;
- **padrões comportamentais:** caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades.

A documentação original do GOF envolve os seguintes itens:

Item	Descrição
nome e classificação do pattern	o nome descreve a essência do pattern de forma objetiva e a classificação segue um dos itens apresentados anteriormente (criação, estrutural ou comportamental)
intenção e objetivo também conhecido como motivação	descreve sucintamente o que faz o design pattern outros nomes conhecidos do mesmo pattern, se existirem um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no pattern solucionam o problema
aplicabilidade estrutura	quais são as situações nas quais o design pattern pode ser aplicado uma representação gráfica das classes do projeto
participantes	as classes e/ou objetos que participam do pattern e suas responsabilidades
colaborações	como os participantes colaboram para executar suas responsabilidades
consequências implementação	quais são os custos e benefícios da utilização do pattern que armadilhas, sugestões ou técnicas é necessário conhecer para implementar o padrão
exemplo de código	No GOF são apresentados exemplos de código em Smalltalk e C++
usos conhecidos	exemplo do pattern encontrado em sistemas reais
patterns relacionados	outros design patterns comumente utilizados em conjunto com este.

Anotações

Outros autores utilizam classificações e itens diferentes para documentação de patterns, mas todos acabam englobando os itens originais definidos por Alexander.

Neste curso vamos seguir a classificação do GOF mas utilizaremos uma notação menos formal para descrição de cada um dos patterns.

3.3 Anti-pattern

Da mesma maneira que design patterns descrevem boas práticas recorrentes utilizadas na resolução de problemas, foi criado o conceito de anti-pattern para as práticas pouco recomendáveis. Tipicamente anti-patterns descrevem práticas muito utilizadas, que apesar de aparentemente serem benéficas, na verdade geram vários tipos de problemas de manutenibilidade, flexibilidade, etc.

São duas as características básicas que definem um anti-pattern:

- algum padrão repetitivo de ações, processos ou estruturas que parecem ser úteis inicialmente, mas que acabam produzindo mais efeitos nocivos do que benéficos;
- existe um design pattern ou refactoring conhecido que substitui o padrão nocivo

O termo anti-pattern foi cunhado baseado no livro de design patterns do GOF e foi utilizado pela primeira vez no livro *Antipatterns*, que define suas características básicas e apresenta um catálogo com estas práticas. Uma boa fonte para anti-patterns conhecidos é o site: www.antipatterns.com, mantido pelos autores do livro.

Neste curso apresentaremos os design patterns a partir de anti-patterns e dos problemas decorrentes de seu uso.

Anotações

Para uso não comercial

Anotações

CAPÍTULO

4

Design Patterns GoF

Para uso não comercial

Para uso não comercial

Anotações

4 Design patterns GoF

Sem dúvida representa a família de design patterns mais tradicional na programação orientada a objetos. Os 23 design patterns GoF representam um conjunto de técnicas de modelagem orientada a objetos que reúne conhecimentos de diversos gurus, desenvolvedores, analistas e arquitetos. A seguir seguem as descrições dos patterns extraídas do próprio livro do GoF.

Design-pattern	Descrição
Abstract Factory	Fornece uma interface para a criação de uma família de objetos relacionados ou dependentes sem fornecer os detalhes de implementação das classes concretas.
Adapter	Converte uma interface de uma classe existente em outra interface esperada pelos clientes. Permite que algumas classes com interfaces diferentes trabalhem em conjunto.
Bridge	Separa uma implementação de sua abstração, de forma que ambas possam variar independentemente.
Builder	Separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo possa criar representações diferentes.
Chain of Responsibility	Evita o acoplamento do remetente de uma solicitação ao seu destinatário, permitindo que diversos objetos tenham a chance de tratar a solicitação. Encadeia os objetos receptores e transmite a solicitação através da cadeia até que um objeto a trate.
Command	Encapsula uma solicitação como um objeto, permitindo que clientes sejam parametrizados com diferentes solicitações e suportem operações que possam ser desfeitas.
Composite	Compõe objetos em estruturas de árvore para representar hierarquias do tipo parte-todo. Permite que os clientes tratem objetos individuais e composições de maneira uniforme.
Decorator	Atribui responsabilidades adicionais a um objeto dinamicamente. Fornecem uma alternativa flexível a sub-classes para extensão de funcionalidade.
Façade	Fornece uma interface mais simples para um conjunto de interfaces de um sub-sistema.
Factory Method	Define uma interface para criação de um objeto, mas deixa as sub-classes decidirem qual a classe a ser instanciada.
Design-pattern	Descrição
Flyweight	Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

Anotações

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 sequencialmente os elementos de um objeto agregado sem expor sua representação.
Mediator	Define um objeto que encapsula como um conjunto de objetos interage.
Memento	Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de modo que o mesmo possa ser restaurado posteriormente.
Observer	Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados.
Prototype	Especifica os tipos de objetos a serem criados utilizando uma instância protótipo e criando novos objetos copiando este protótipo.
Proxy	Fornece um objeto representante de outro objeto, de forma a controlar o acesso ao mesmo.
Singleton	Garante que uma classe tenha somente uma instância e fornece um ponto de acesso global a ela.
State	Permite que um objeto altere seu comportamento quando seu estado muda.
Strategy	Define uma família de algoritmos e os encapsula tornando-os intercambiáveis.
Template Method	Define o esqueleto de um algoritmo em uma operação, postergando a implementação de alguns passos para sub-classes.
Visitor	Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. Permite que uma nova operação seja definida sem mudar as classes dos elementos sobre os quais opera.

Anotações

4.1 Singleton - Criação

Este design pattern é bastante popular e tem um objetivo técnico de fácil compreensão. Em muitas situações podemos imaginar classes onde só precisaríamos de uma única instância durante o ciclo de vida do aplicativo. Podemos imaginar os seguintes exemplos de classes:

- ConfigManager: para representar as configurações do aplicativo;
- ServiceLocator: centralizadora de acesso a recursos técnicos externos;
- DriverManager: classe da API JDBC que poderia ser implementada como Singleton;

Portanto, Singleton é um design pattern de criação cujo objetivo é fazer que a classe tecnicamente ofereça apenas uma instância de objeto, que será controlada por ela mesma. Ao aplicarmos o Singleton nas situações corretas, temos como consequência um número menor de objetos de “mortalidade infantil”, pois a classe disponibilizará apenas acesso a um único objeto.

4.1.1 Anti-pattern

Vamos analisar o anti-pattern do Singleton:

```
ConfigManager c = new ConfigManager();
c.getDatabaseServerName();
.....
.....
ConfigManager c = new ConfigManager();
c.getForeColor();
```

Neste caso supostamente duas classes necessitariam de informações encapsulada por ConfigManager e para acessá-las acabam criando uma instância da classe que em curto espaço de tempo será descartada, causando um aumento significativo no processo de construção e destruição / coleta de lixo do objeto.

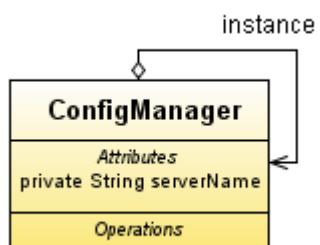
4.1.2 Aplicando o design pattern

Ao aplicar o Singleton devemos:

- Garantir que somente um objeto será criado a partir daquela classe;
- Disponibilizar um método de acesso ao único objeto;

Anotações

4.1.2.1 Representação em UML



4.1.2.2 Código

Exemplo: ConfigManager.java

```

1 package br.com.globalcode.cp.singleton;
2
3 public class ConfigManager {
4     private static ConfigManager instance = new ConfigManager();
5     private String serverName;
6
7     private ConfigManager() {
8     }
9     public static ConfigManager getInstance() {
10         return instance;
11     }
12
13     public String getServerName() {
14         return serverName;
15     }
16
17     public void setServerName(String serverName) {
18         this.serverName = serverName;
19     }
20 }
  
```

Comentários:

- Linha 4: uma instância static da própria classe é declarada e criada;
- Linha 7: o construtor se torna private para garantir tecnicamente que ninguém consiga criar objetos de fora da classe;
- Linha 9 a 11: um método público static que retorna a instancia é declarado;

Anotações

Utilizando o código:

Exemplo: UsaConfigManager.java

```
1 package br.com.globalcode.cp.singleton;
2
3 public class UsaConfigManager {
4     public static void main(String[] args) {
5         String s = ConfigManager.getInstance().getServerName();
6         ConfigManager config = ConfigManager.getInstance();
7         String s1 = config.getServerName();
8     }
9 }
```

Comentários:

- Linha 5: acesso à única instância direto;
- Linha 6 e 7: obtendo uma referência para a única instância e usando posteriormente na linha ;

4.1.3 Considerações sobre inicialização

Na implementação apresentada as classes Singleton serão inicializadas em tempo de carregamento de classe pois temos a construção do objeto na sua declaração:

```
1 public class ConfigManager {
2     private static ConfigManager instance = new ConfigManager();
3     private String serverName;
4
5     private ConfigManager() {
6     }
...
}
```

Muitas vezes você pode querer inicializar o objeto tardiamente / lazy. Isso poderia ser programado dentro do método getInstance:

```
1 public class ConfigManager {
2     private static ConfigManager instance;
3     private String serverName;
4
5     private ConfigManager() {
6     }
7     public static synchronized ConfigManager getInstance() {
8         if(instance==null) instance = new ConfigManager();
9         return instance;
10    }
...
}
```

Anotações

4.1.4 Herança com Singleton

Este tipo de implementação Singleton acaba limitando o uso de herança e polimorfismos em função de trabalharmos com o construtor private. Para obter melhores resultados com objetos Singleton que devem ser criados a partir de uma família de classes, devemos aplicar o design pattern Factory.

4.1.5 Singleton Vs. Static

Em termos de alocação de memória os mesmos resultados poderiam ser obtidos se simplesmente aplicássemos o modificador static em todos os métodos e atributos da nossa classe:

Exemplo: ConfigManagerStatic.java

```

1 package br.com.globalcode.cp.singleton;
2
3 public class ConfigManagerStatic {
4     private static String serverName;
5
6     public static String getServerName() {
7         return serverName;
8     }
9     public static void setServerName(String serverName) {
10        ConfigManagerStatic.serverName = serverName;
11    }
12 }
13
14

```

Neste caso os atributos e métodos são static e para utilizarmos vamos sempre referenciar ConfigManager.getServerName, ou seja, uso menos elegante dos princípios de orientação a objetos. Mas além da elegância, essas classes não costumam ter o cuidado de tornar seu construtor private, portanto nada garantirá que usuários não criem objetos a partir delas.

Mas vamos imaginar o seguinte caso para nos convencermos de vez com relação ao Singleton: você escreveu uma classe ConfigManager com Singleton e modularizou de um jeito seu software, que agora tem a necessidade de ter um ConfigManager por módulo e não mais um ConfigManager para todo o aplicativo. Se você implementou Singleton, poderá simplesmente alterar as instruções do getInstance():

```

1 public class ConfigManager {
2     private static ConfigManager instance = new ConfigManager();
3     private String serverName;
4
5     private ConfigManager() {
6     }
7     public static ConfigManager getInstance() {
8         // programação da regra para se obter nome do módulo atual
9     }
...

```

Anotações

Se nesta situação tivéssemos modelado a classe ConfigManager com seu métodos e atributos static, teríamos um esforço de refatoramento muito maior.

4.1.6 Singleton “si pero no mucho”

Algumas situações técnicas podem levar um Singleton a criar mais de uma instância e por este motivo este design pattern é fruto de inúmeras discussões públicas. Vamos levantar algumas possibilidades, porém cabe a você desenvolvedor aplicar com bom senso e cabe ao arquiteto permitir ou não a aplicação deste recurso em cada cenário específico.

4.1.6.1 Clustering

A primeira situação fácil de se imaginar é quando trabalhamos com mais que uma máquina virtual, seja por motivo de clustering, seja por motivo de particionamento de servidor. Este caso acarretará um objeto Singleton por máquina virtual, o que em grande parte dos casos de uso, não acarretaria em prejuízos, desde que não existam consequências técnicas na classe Singleton diante esta situação. Imagine um Singleton que acessa um recurso extremamente crítico, que efetivamente só um objeto poderia acessá-lo por vez. Diante desta situação o Singleton pode parecer a solução, desde que nunca se implemente o aplicativo em clustering.

4.1.6.2 Class Loader

Uma situação que podemos ter objetos Singleton duplicados dentro da mesma máquina virtual é quando trabalhamos com diferentes class loaders ou quando o servidor de aplicativos trabalha com diferentes class loaders. Um clássico exemplo foi o iPlanet, servidor da Sun, que utilizava um class loader por servlet, sendo assim, se dois diferentes servlets acessarem uma classe Singleton, diferentes instâncias serão criadas, uma para cada class loader.

Anotações

4.1.6.3 Negligenciando synchronized

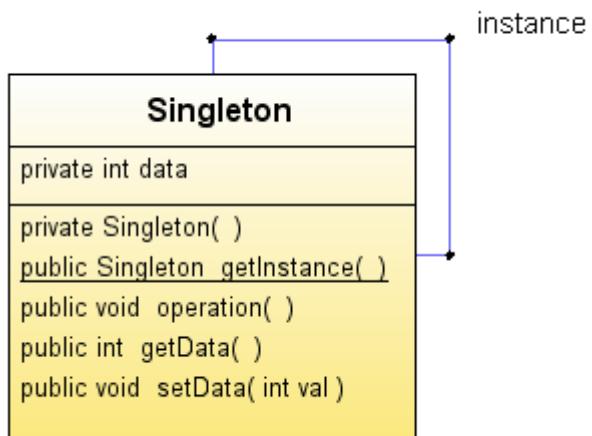
Quando trabalhamos com Singleton e inicialização sob demanda, temos que tomar cuidado com o processo de sincronização de threads, pois mais de uma thread poderá solicitar determinado método ao mesmo tempo.

```

1 public class ConfigManager {
2     private static ConfigManager instance;
3     private String serverName;
4
5     private ConfigManager() {
6     }
7     public static synchronized ConfigManager getInstance() {
8         if(instance==null) instance = new ConfigManager();
9         return instance;
10    }
11 ...

```

4.1.7 Estrutura e Participantes



- **Singleton (ConfigManager)**

Define uma operação `getInstance()` que retorna sua única instância.

Anotações

4.2 Command - Comportamento

*Command é também conhecido como Action ou Transaction.

Este design pattern é bastante aplicado em diversos frameworks de interfaces gráficas e interfaces Web. Aplicamos o Command para encapsular ações que geralmente estão associadas a um objeto de interface gráfica, como por exemplo um botão. O fato é que o construtor da biblioteca do componente botão não tem como conhecer qual ação que aquele botão deverá executar, isso vai variar de acordo com a necessidade da tela, e cabe aos usuários daquela biblioteca programar uma classe que vai encapsular o código que deverá ser executado para cada botão. Command é um design pattern para se desenvolver este tipo classe.

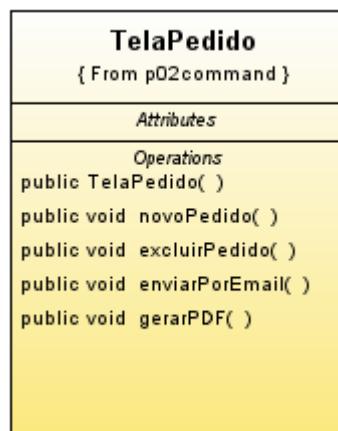
Esta é exatamente a abordagem dos toolkit de interfaces gráficas AWT e Swing do mundo Java e também de vários outros das demais plataformas. Na Web podemos lembrar do framework Jakarta Struts que encapsula cada ação da Web em uma sub-classe denominada Action. Por este motivo se você já utilizou AWT, Swing ou Struts, fatalmente já foi usuário deste pattern.

Vamos comentar algumas técnicas de implementação e dicas para você agora atuar como modelador e não usuário design pattern Command.

4.2.1 Anti-pattern

No diagrama a seguir temos uma classe que supostamente disponibiliza quatro diferentes ações:

1. Novo pedido
2. Excluir o pedido
3. Enviar pedido por e-mail
4. Gerar PDF

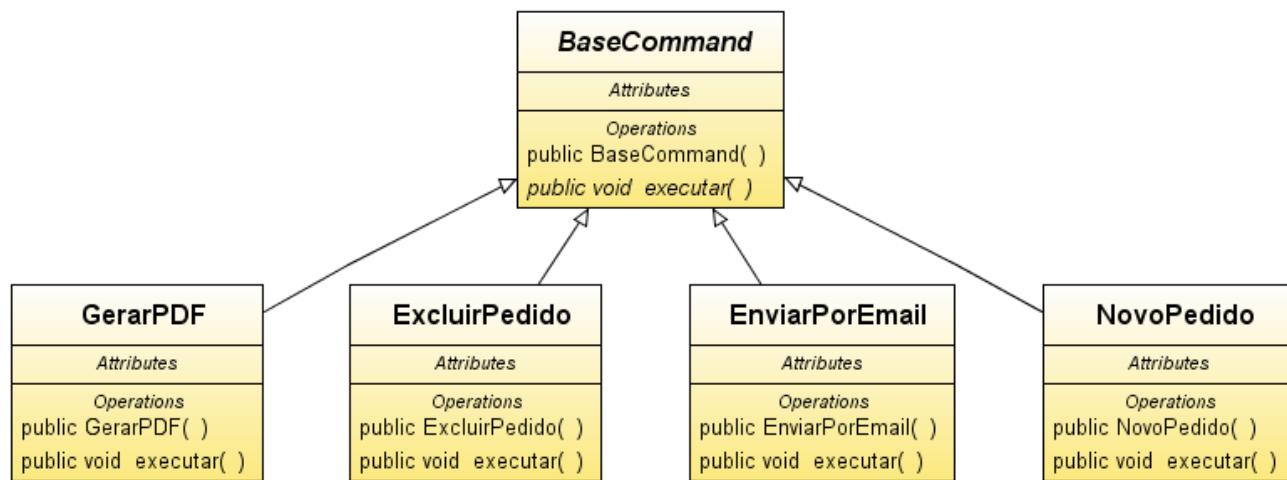


Anotações

Consideramos este o anti-pattern do Command pois no lugar de modelarmos uma classe por comando temos todas elas agrupadas em uma só classe. O crescimento desta classe implicará na construção de vários novos métodos, tornando a modelagem menos flexível em termos de reuso / extensão e manutenção de código.

4.2.2 Aplicando o design pattern

Ao aplicarmos este design pattern tipicamente modelamos uma hierarquia de objetos que representarão as ações do nosso aplicativo. Neste diagrama, criamos uma classe abstrata BaseCommand, com um método chamado executar, tipo void, sem entradas:



Esta hierarquia de classes pode ser implementada através de uma classe base ou então com o uso de uma interface e o seu método de encapsulamento da ação do aplicativo poderá receber parâmetros com informações do evento e retornar valores contendo resultados diversos.

Agora temos que gerar um vínculo entre nosso aplicativo e esta determinada hierarquia de classes. Este vínculo em geral recorre a outros patterns, como factory e dependency injection, que abordaremos posteriormente.

4.2.3 Command e Factory

Um dos design patterns popularmente utilizado em conjunto com o Command é Factory Method. Factories são classes responsáveis por criar objetos de uma determinada hierarquia através de um sistema de vínculo fraco que permitirá a adição de novas classes Commands sem a alteração do aplicativo. Factory Method será o próximo design pattern abordado neste material.

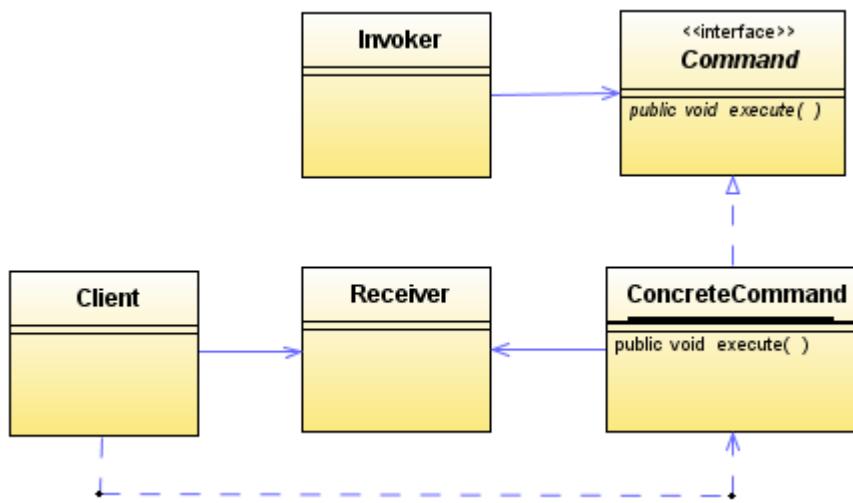
Anotações

4.2.4 Command e sistemas de desfazer / refazer (undo / redo)

Este design pattern deve ser utilizado, em conjunto com outros mais específicos, para a implementação de funcionalidades de undo / redo. Podemos imaginar um sistema que “fotografa” objetos afetados pelo comando antes e depois de sua execução para permitir que o usuário desfaça uma operação.

Sistemas de undo / redo são mais complexos e por vezes, inviáveis quando se trata de ambientes multi-usuário com operação em dados concorrentes.

4.2.5 Estrutura e Participantes



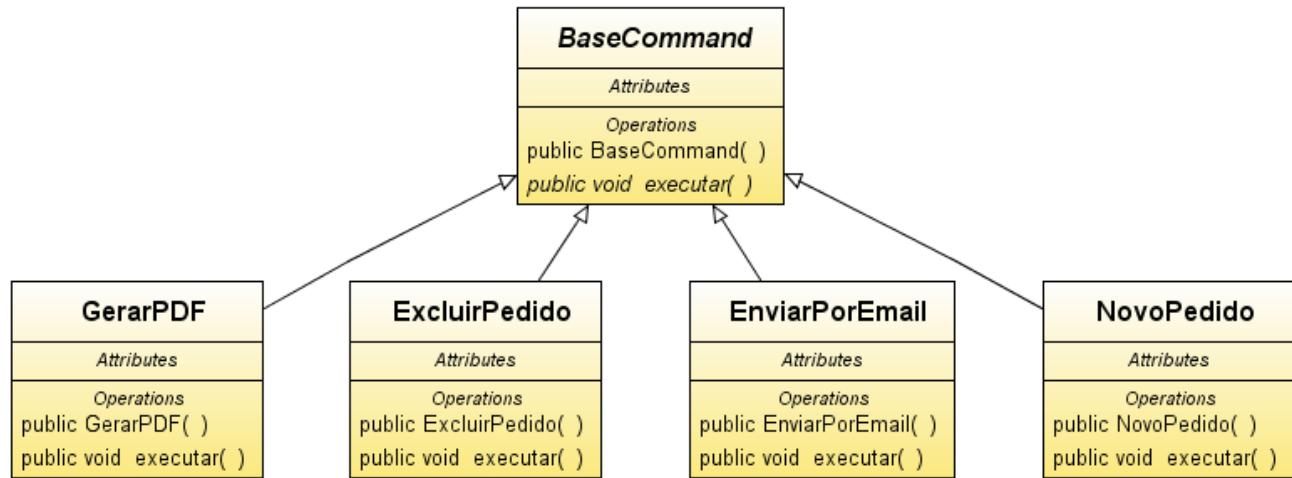
- **Command (BaseCommand)**
Declara uma interface para a execução de uma operação.
- **ConcreteCommand (GerarPDF, ExcluirPedido, EnviarPorEmail, NovoPedido)**
Implementa `execute` através da manipulação do *Receiver* correspondente.
- **Client**
Cria um objeto *ConcreteCommand* e estabelece o seu *Receiver*.
- **Invoker**
Solicita ao *Command* a execução da operação.
- **Receiver**
Sabe como executar as operações associadas a uma solicitação.

Anotações

4.3 Factory Method - Criação

*Factory Method é também conhecido como Virtual Constructor

Quando temos uma hierarquia de classes definidas, como a hierarquia de Commands, fatalmente temos a necessidade de criar as sub-classes conforme um determinado comando foi acionado.



Para o aplicativo usuário desta família de classes, o ideal seria ter vínculo apenas com a classe `BaseCommand` e não com suas demais classes, com isso poderíamos pensar que a criação de um novo comando não acarretaria na alteração do aplicativo em si, facilitando a extensibilidade do modelo e aplicativo.

Factory Method é um método responsável por encapsular esta lógica de criação condicional.

Anotações

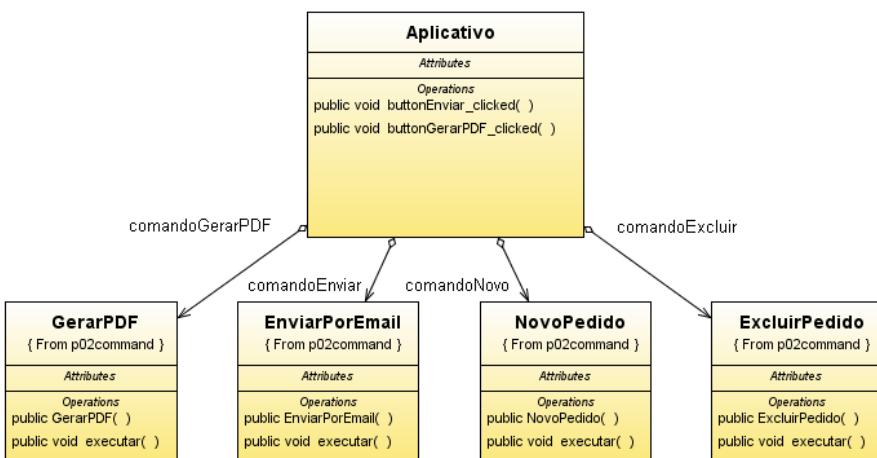
4.3.1 Anti-pattern

Podemos imaginar que o seguinte código representa o anti-pattern do Factory Method:

Exemplo: Aplicativo.java

```
1 package br.com.globalcode.cp.factory;
2
3 import br.com.globalcode.cp.command.*;
4
5 public class Aplicativo {
6     private EnviarPorEmail comandoEnviar;
7     private ExcluirPedido comandoExcluir;
8     private GerarPDF comandoGerarPDF;
9     private NovoPedido comandoNovo;
10
11    public void buttonEnviar_clicked() {
12        comandoEnviar.executar();
13    }
14    public void buttonGerarPDF_clicked() {
15        comandoGerarPDF.executar();
16    }
17    //...
18 }
```

Perceba o forte acoplamento entre a classe aplicativo e cada um dos seus comandos. Tal modelagem conta com a vantagem de encapsular o código de cada comando em classes distintas, o que faz com que a classe Aplicativo não cresça no formato “spaguetti”. Porém, cada novo comando adicionado requer uma alteração em alguma parte do aplicativo, em função do mesmo não contar com um sistema mais dinâmico de fabricação de objetos, uma classe com Factory Method.



Anotações

4.3.2 Aplicando o design pattern

Vamos construir uma classe chamada CommandFactory que implementará o método create(String command):

Exemplo: CommandFactory.java

```
1 package br.com.globalcode.cp.factory;
2
3 import br.com.globalcode.cp.command.BaseCommand;
4
5 public class CommandFactory {
6     public BaseCommand create(String name) {
7         BaseCommand objetoCommand = null;
8         //regra de criação condicional
9         return objetoCommand;
10    }
11 }
```

Vamos implementar posteriormente a lógica de criação que poderá ser estática ou dinâmica. Independente da lógica adotada, o aplicativo seria refatorado para:

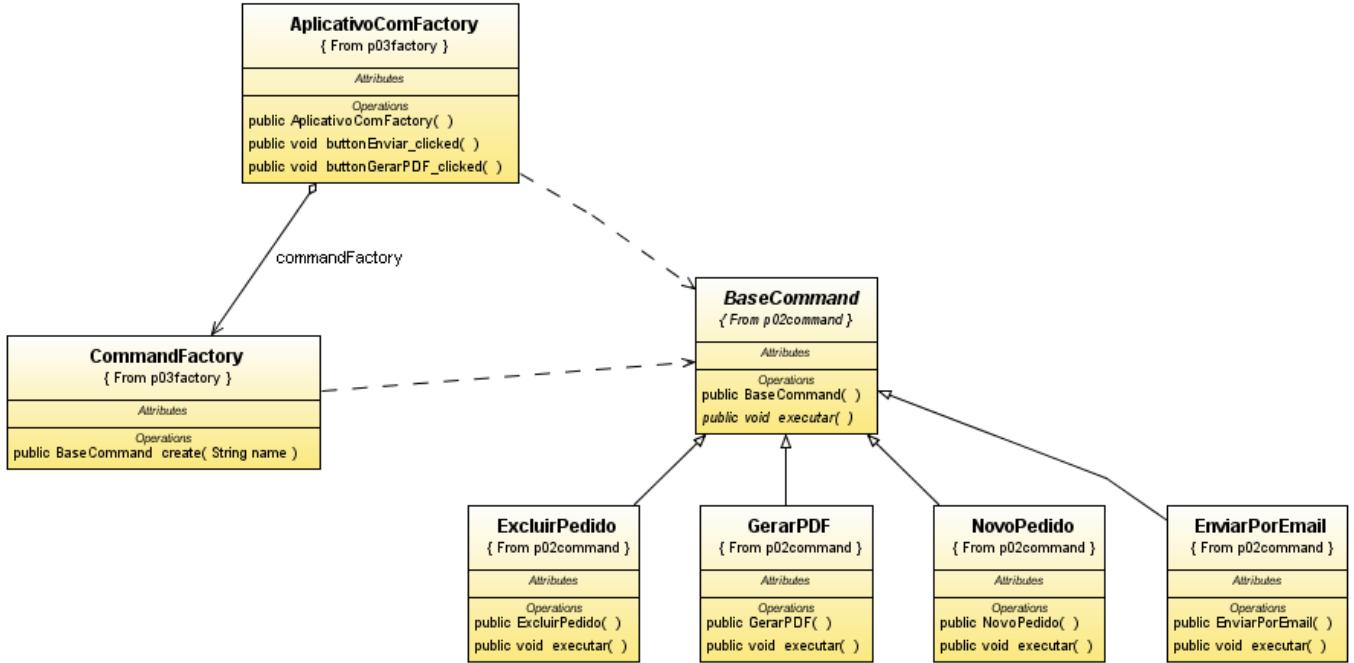
Exemplo: AplicativoComFactory.java

```
1 package br.com.globalcode.cp.factory;
2
3 import br.com.globalcode.cp.command.BaseCommand;
4
5 public class AplicativoComFactory {
6     private CommandFactory commandFactory;
7
8     public AplicativoComFactory() {
9         commandFactory = new CommandFactory();
10    }
11    public void buttonEnviar_clicked() {
12        BaseCommand comandoEnviar = commandFactory.create("EnviarPedido");
13        comandoEnviar.executar();
14    }
15    public void buttonGerarPDF_clicked() {
16        BaseCommand comandoGerarPDF = commandFactory.create("GerarPDF");
17        comandoGerarPDF.executar();
18    }
19    //...
20 }
```

Podemos notar que não existe nenhum vínculo físico entre as classes que representam os comandos e a aplicação, este vínculo esta apenas caracterizado pela String informada para CommandFactory. Normalmente atribuímos este nome, ou melhor ainda, um apelido para ação que será executada.

Anotações

Podemos observar no diagrama a seguir a independência do aplicativo e cada um dos seus comandos:



4.3.3 Factory Method Estático

Um Factory Method estático seria o mais simples de implementarmos e não apresentará grandes vantagens em termos de extensibilidade no modelo:

Exemplo: CommandFactory1.java

```

1 package br.com.globalcode.cp.factory;
2
3 import br.com.globalcode.cp.command.*;
4
5 public class CommandFactory1 {
6     public BaseCommand create(String name) {
7         BaseCommand command = null;
8         if(name.equals("EnviarPedido")) {
9             command = new EnviarPorEmail();
10        }
11        else if(name.equals("GerarPDF")) {
12            command = new GerarPDF();
13        }
14        //...
15        return command;
16    }
17 }
  
```

Nesta implementação teremos a dependência entre a Factory e todos os comandos. Por este motivo não é a implementação mais recomendada. Devemos recorrer a meta-programação nas linguagens, para maior dinâmica.

Anotações

4.3.4 Factory Method Dinâmico

Grande parte das linguagens oferecem algum sistema de meta-programação ou programação reflexiva para podermos criar classes mais independentes e consequentemente frameworks. Java oferece a chamada de Reflection para este fim. Com Reflection temos classes que representam classes, métodos, construtores, entre outros, e podemos chamar métodos ou criar objetos sem vínculo em tempo de compilação. Vejamos a implementação de um Factory Method utilizando Java Reflection:

Exemplo: CommandFactory2.java

```

1 package br.com.globalcode.cp.factory;
2
3 import br.com.globalcode.cp.command.*;
4 import java.io.*;
5 import java.util.Properties;
6
7 public class CommandFactory2 {
8     Properties apelidosComandos = new Properties();
9     public CommandFactory2() {
10         try {
11             apelidosComandos.load(new FileInputStream("commands.properties"));
12         } catch (Exception ex) {
13             //tratar
14         }
15     }
16     public BaseCommand create(String name) {
17         BaseCommand command = null;
18         String stringClasse = apelidosComandos.getProperty(name);
19         try {
20             Class classe = Class.forName(stringClasse);
21             Object object = classe.newInstance();
22             command = (BaseCommand) object;
23         } catch (Exception trateme) {
24             //tratar
25         }
26         return command;
27     }
28 }
```

Utilizamos um arquivo chamado commands.properties que vai mapear o nome lógico do comando com um nome físico de classe Java, completamente qualificado. O conteúdo deste arquivo properties será:

Exemplo: commands.properties

```

1 NovoPedido = br.com.globalcode.cp.command.NovoPedido
2 GerarPDF = br.com.globalcode.cp.command.GerarPDF
3 EnviarPedido = br.com.globalcode.cp.command.EnviarPorEmail
4 # ...outros mapeamentos
```

Pronto! Agora temos um fábrica dinâmica e novos comandos não implicarão em mudança de código Java.

Anotações

4.3.5 Factory e reuso de objetos

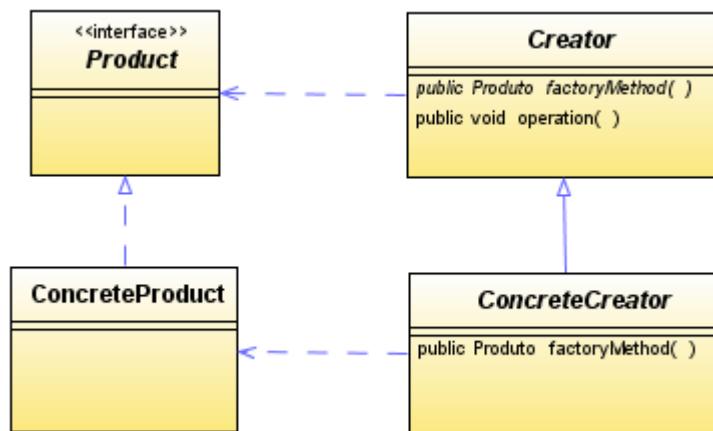
Uma das grandes vantagens de trabalhar com Factory é que podemos controlar o ciclo de vida, quantidade de objetos e também podemos fazer sistemas de cache e pooling. A grande maioria dos frameworks Java para diversas disciplinas trabalham com Factories para garantir reaproveitamento de objetos e evitar a construção e destruição excessiva de objetos, operação que conforme comentamos, apresenta um alto custo para máquina virtual em frente a grandes demandas.

Podemos imaginar classes com Factory Method inteligentes, que reaproveitam através de uma collection os objetos que são criados para as requisições.

4.3.6 Factory Method e outros design patterns

Factory Method é muitas vezes aplicado para construir famílias de objetos de um outro design pattern, como exemplo temos o CommandFactory, mas podemos pensar em factories de Data Access Object, Business Delegate, Façade, Proxy, etc.

4.3.7 Estrutura e Participantes



- **Product** (`BaseCommand`)
Define a interface de objetos que o método fábrica cria.
- **ConcreteProduct** (`GerarPDF`, `ExcluirPedido`, `EnviarPorEmail`, `NovoPedido`)
Implementa a interface de *Product*
- **Creator**

Anotações

Declara o método fábrica o qual retorna um objeto do tipo *Product*. Também pode ser criada uma implementação que retorne um objeto *Product* default. Pode chamar o método fábrica em uma de suas operações.

- **ConcreteCreator**

Redefine o método fábrica para retornar uma instância de *ConcreteProduct*.

Para uso não comercial!

Anotações

4.4 Abstract Factory - Criação

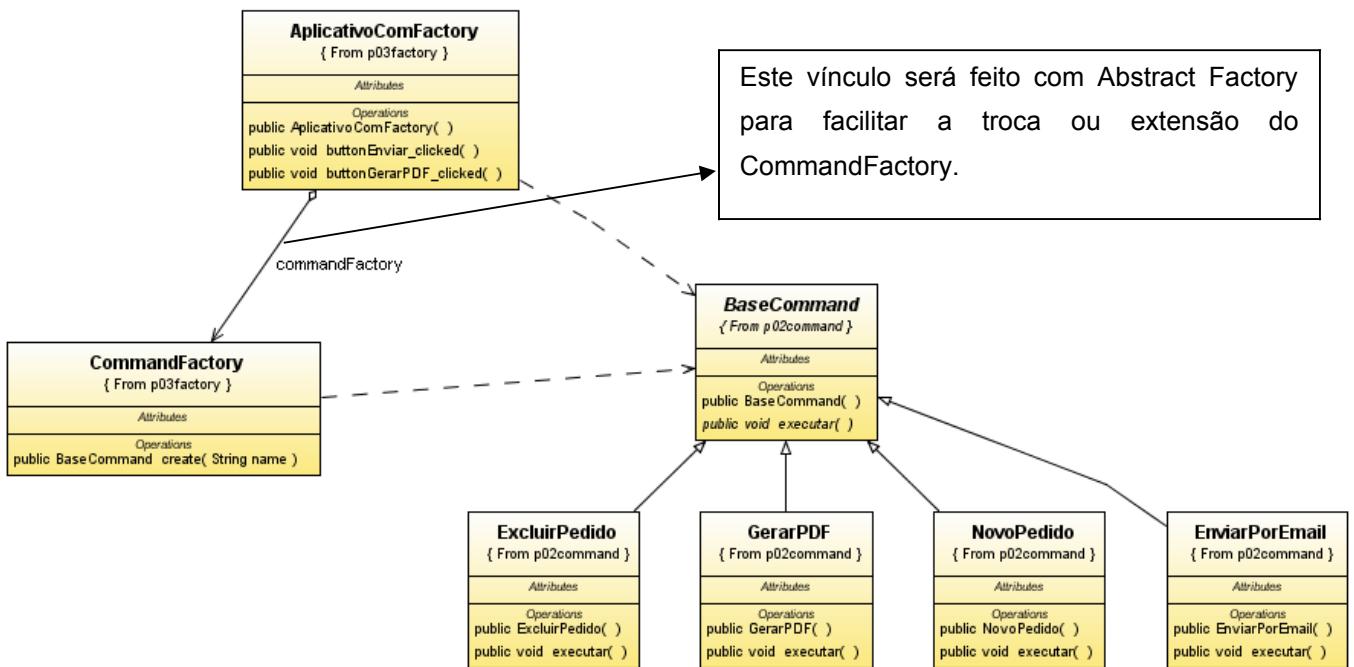
*Também conhecido como Kit

Enquanto design pattern Factory Method se concentra em fornecer um ponto padrão de criação de objetos de uma mesma hierarquia, o Abstract Factory prevê maior flexibilidade no modelo permitindo a troca da classe que implementa Factory Method com dinamismo. Podemos imaginar que o Abstract Factory é um design pattern para criar uma família de classes Factory Method, ou seja, é uma fábrica da fábrica de objetos.

Quando construímos um aplicativo com classes no padrão Factory Method, temos que de alguma forma vincular o aplicativo a classe Factory Method. Abstract Factory é uma solução para este problema.

4.4.1 Anti-pattern

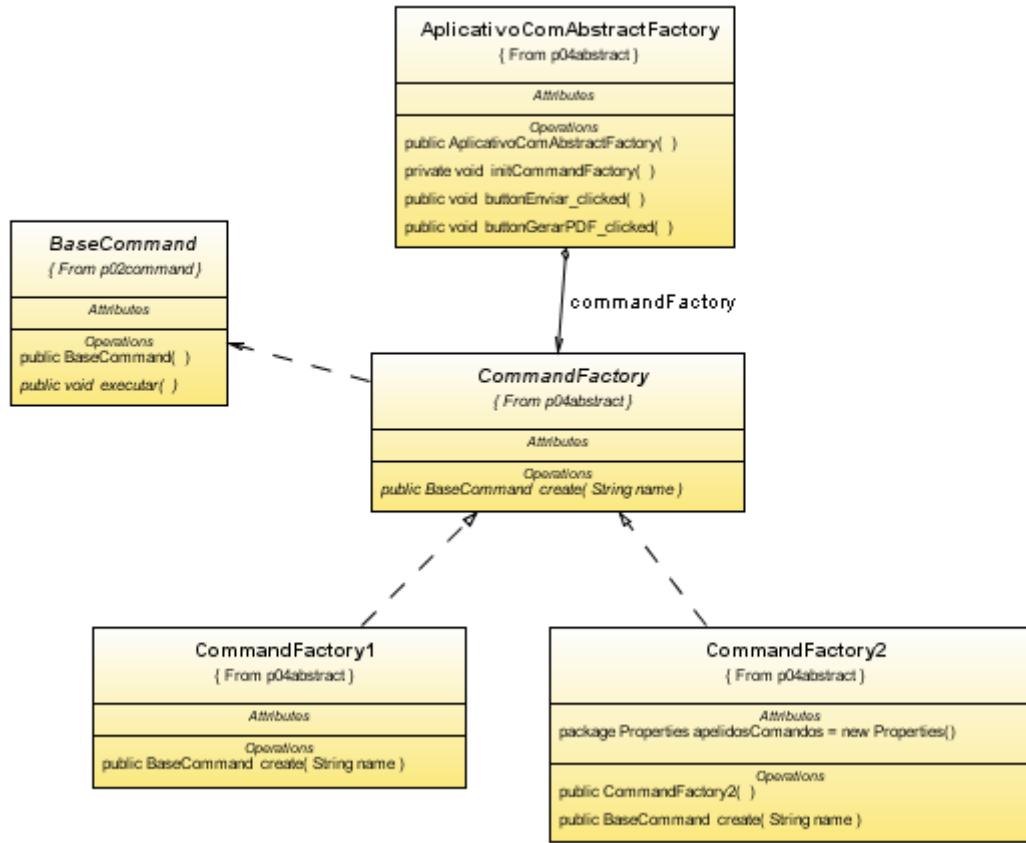
O modelo que foi apresentado anteriormente representa o anti-pattern do Abstract Factory, pois o Aplicativo está totalmente acoplado a uma implementação de Factory. Se mudarmos a implementação da Factory através de uma sub-classe, teremos que mudar o Aplicativo também.



4.4.2 Aplicando o design pattern

Nossa intenção agora é tornar o aplicativo independente de implementação de Command Factory, ou seja, vamos de alguma forma, pode especificar isso em tempo de execução e não em tempo de compilação, facilitando manutenção, extensão e reuso do aplicativo.

Anotações



Agora com este modelo podemos variar os Commands e também podemos variar a classe Factory dele, ou seja, temos dois pontos de extensão e reuso: os comandos individuais, que representam pontos bem granularizados e podemos também reaproveitar, reusar ou alterar suas fábricas com facilidade. Vejamos o código Java com a implementação.

Anotações

A primeira providência foi transformar CommandFactory em interface e depois criarmos as duas implementações que foram apresentadas: Factory Method estático e dinâmico:

Exemplo: CommandFactory.java

```
1 package br.com.globalcode.cp.absfactory;
2
3 import br.com.globalcode.cp.command.BaseCommand;
4
5 public interface CommandFactory {
6     public BaseCommand create(String name);
7 }
```

As implementações de CommandFactory:

Exemplo: CommandFactory1.java

```
1 package br.com.globalcode.cp.absfactory;
2
3 import br.com.globalcode.cp.command.BaseCommand;
4 import br.com.globalcode.cp.command.EnviarPorEmail;
5 import br.com.globalcode.cp.command.GerarPDF;
6
7 public class CommandFactory1 implements CommandFactory{
8     public BaseCommand create(String name) {
9         BaseCommand command = null;
10        if(name.equals("EnviarPedido")) {
11            command = new EnviarPorEmail();
12        }
13        else if(name.equals("GerarPDF")) {
14            command = new GerarPDF();
15        }
16        //...
17        return command;
18    }
19 }
```

Anotações

Segunda implementação com Factory dinâmico:

Exemplo: CommandFactory2.java

```
1 package br.com.globalcode.cp.absfactory;
2
3 import br.com.globalcode.cp.command.*;
4 import java.io.*;
5 import java.util.Properties;
6
7 public class CommandFactory2 implements CommandFactory{
8     Properties apelidosComandos = new Properties();
9
10    public BaseCommand create(String name) {
11        BaseCommand command = null;
12        String stringClasse = apelidosComandos.getProperty(name);
13        try {
14            Class classe = Class.forName(stringClasse);
15            Object object = classe.newInstance();
16            command = (BaseCommand) object;
17        } catch (Exception trateme) {
18            //tratar
19        }
20        return command;
21    }
22    public CommandFactory2() {
23        try {
24            apelidosComandos.load(new FileInputStream("commands.properties"));
25        } catch (Exception ex) {
26            //tratar
27        }
28    }
29 }
```

Anotações

Vejamos agora o código do aplicativo, agora sem vínculo em tempo de compilação com nenhum comando concreto e nenhuma fábrica concreta:

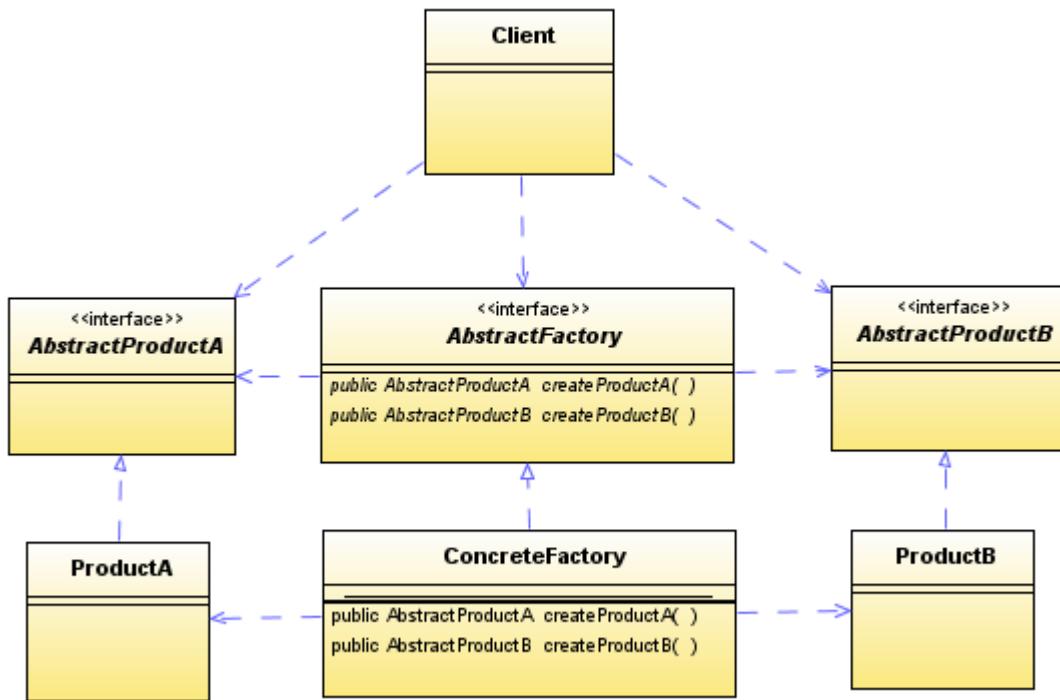
Exemplo: AplicativoComAbstractFactory.java

```
1 package br.com.globalcode.cp.absfactory;
2
3 import br.com.globalcode.cp.command.BaseCommand;
4 import java.io.InputStream;
5 import java.util.Properties;
6
7 public class AplicativoComAbstractFactory {
8     private CommandFactory commandFactory;
9
10    public AplicativoComAbstractFactory() {}
11
12    private void initCommandFactory() throws Exception {
13        Properties propriedades = new Properties();
14        InputStream f = getClass().getResourceAsStream("config.properties");
15        propriedades.load(f);
16        f.close();
17        String nome = propriedades.getProperty("command.factory.class");
18        Class classeFactoryCommandFactory = Class.forName(nome);
19        commandFactory = (CommandFactory) classeFactoryCommandFactory.newInstance();
20    }
21
22    public void buttonEnviar_clicked() {
23        BaseCommand comandoEnviar = commandFactory.create("EnviarPedido");
24        comandoEnviar.executar();
25    }
26    public void buttonGerarPDF_clicked() {
27        BaseCommand comandoGerarPDF = commandFactory.create("GerarPDF");
28        comandoGerarPDF.executar();
29    }
30    //...
31 }
```

Observamos que entre as linhas 17 e 19 a classe Aplicativo inicializa o objeto que representa a fábrica de comandos. Essa operação é feita contando com um arquivo config.properties onde entraremos com o nome físico da classe que representa nossa fábrica de comandos.

Anotações

4.4.3 Estrutura e Participantes



- **AbstractFactory**

Declara uma interface para operações que criam objetos-produto abstratos.

- **ConcreteFactory**

Implementa as operações que criam objetos-produto concretos.

- **AbstractProduct**

Declara uma interface para um tipo de objeto-produto.

- **ConcreteProduct**

Define um objeto-produto a ser criado pela correspondente fábrica concreta.

- **Client**

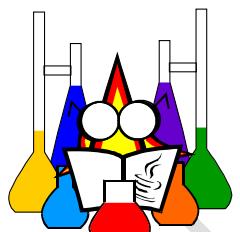
Aplicativo que usa somente as interfaces definidas para produtos e para a fábrica.

Anotações

4.5 Laboratório 1

Objetivo:

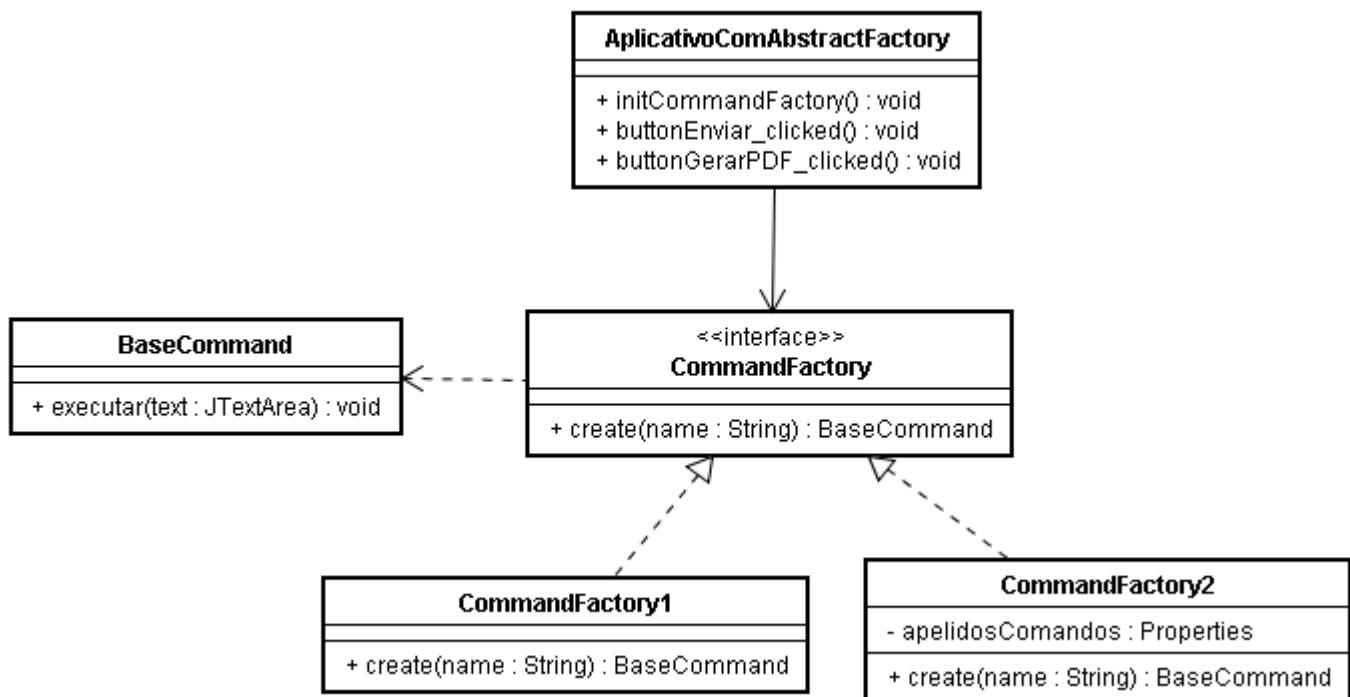
Praticar os patterns Singleton, Command, Abstract Factory e Factory Method.



LABORATÓRIO

Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab01.zip na URL indicada pelo instrutor(a).	
2. Descompacte o arquivo em seu diretório de trabalho.	
3. Analise as classes Aplicativo.java e ConfigManager.java	
4. Aplique o design pattern Singleton na classe ConfigManager.java	
5. Aplique o design pattern Command para refatorar a classe Aplicativo.java	
6. Aplique os design patterns Abstract Factory e Factory Method refatorando novamente a classe Aplicativo.java	
7. Crie 2 implementações de Factory de Command	
8. Você deve utilizar como base o diagrama apresentado a seguir:	



Anotações

4.6 Template Method - Comportamento

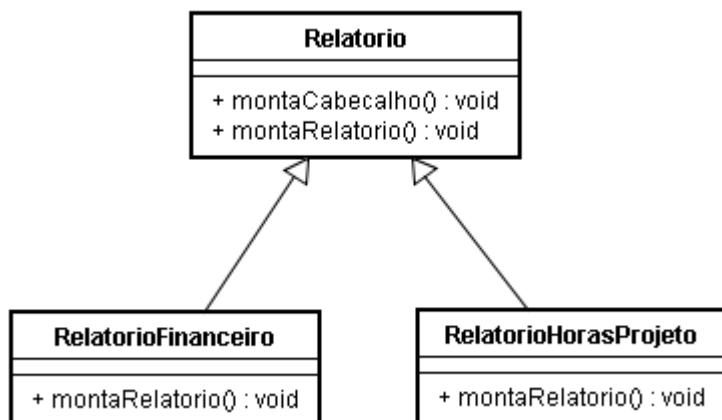
No desenvolvimento de sistemas de software, algumas vezes nos deparamos com problemas onde parte de um determinado algoritmo depende da implementação de classes filhas, mas a estrutura do algoritmo é única. Por exemplo, podemos ter uma classe que monta relatórios denominada Relatorio. O algoritmo para montar relatórios é sempre o mesmo consistindo nas seguintes operações

- montar o cabeçalho
- montar o corpo
- montar o rodapé

Apesar da sequência de operações ser a mesma, os detalhes de implementação podem variar. Relatórios para a mesma empresa podem manter o cabeçalho e o rodapé e variar o conteúdo do corpo. Em outros cenários pode ser necessário variar também o cabeçalho e o rodapé. Como modelar as classes para atender os diversos cenários? Vamos estudar uma solução utilizando herança e sobrescrita de métodos e depois outra utilizando o pattern Template Method, que foi criado para solucionar justamente esse problema.

4.6.1 Anti-pattern

Vamos seguir a solução tradicional OO e utilizar herança para implementar nosso sistema de geração de relatórios, de acordo com o diagrama apresentado a seguir:



Anotações

Exemplo: Relatorio.java

```
1 package br.com.globalcode.cp.templatemethod.antiPattern;
2
3 public class Relatorio {
4
5     package br.com.globalcode.cp.templatemethod.antiPattern;
6
7     public class RelatorioFinanceiro extends Relatorio {
8
9         package br.com.globalcode.cp.templatemethod.antiPattern;
10
11        public class RelatorioHorasProjeto extends Relatorio {
12
13            @Override
14            public void montaRelatorio() {
15                super.montaCabecalho();
16                //regra para montar o corpo do relatorio
17                //regra para montar o rodapé
18            }
19        }
20    }
21 }
```

4.6.2 Aplicando o design pattern

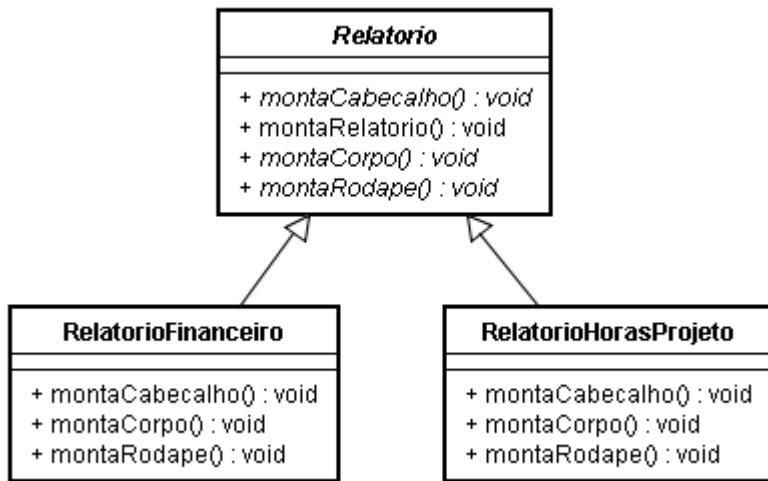
Template Method é uma técnica de modelagem de classes abstratas e sub-classes, que se baseia na seguinte idéia:

1. Definimos uma classe abstrata com métodos abstratos e métodos concretos
2. Nos métodos concretos da classe abstrata, definimos a estrutura dos algoritmos, chamando seus métodos abstratos, mesmo sem saber qual será a implementação
3. Definimos sub-classes que implementam os métodos abstratos.

Dessa forma eliminamos os principais problemas da solução anterior: repetição de códigos e falta de garantia da integridade da execução do algoritmo.

Veja a solução utilizando Template Method.

Anotações



Exemplo: Relatorio.java

```

1 package br.com.globalcode.cp.templatemethod;
2
3 public abstract class Relatorio {
4
5     public void montaRelatorio() {
6         montaCabecalho();
7         montaCorpo();
8         montaRodape();
9     }
10
11    public abstract void montaCabecalho();
12    public abstract void montaCorpo();
13    public abstract void montaRodape();
14 }
  
```

Anotações

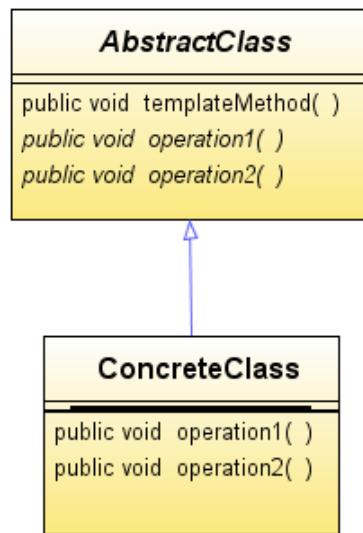
Repare em destaque o Template Method `montaRelatorio` que utiliza os métodos abstratos `montaCabecalho`, `montaCorpo` e `montaRodape`. Com isso, a estrutura do algoritmo é mantida. Outra opção seria criar implementações padrão para a montagem do cabeçalho e rodapé, permitindo que fossem reutilizados pelas subclasses. Nesse caso, somente o método `montaCorpo` seria abstrato.

Agora suas subclasses:

Exemplo: RelatorioFinanceiro.java

```
-->-----
1 package br.com.globalcode.cp.templatemethod;
2
3 public class RelatorioFinanceiro extends Relatorio {
4
5     public void montaCabecalho() {
6         package br.com.globalcode.cp.templatemethod;
7
8         public class RelatorioHorasProjeto extends Relatorio {
9             public void montaCabecalho() {
10                //regra para montar cabeçalho
11            }
12
13            public void montaCorpo() {
14                //regra para montar corpo do relatório
15            }
16
17            public void montaRodape() {
18                //regra para montar rodapé
19            }
20
21        }
22
23        public abstract class InputStream implements Closeable {
24
25            public abstract int read() throws IOException;
26
27            public int read(byte b[], int off, int len) throws IOException {
28                if (b == null) {
29                    throw new NullPointerException();
30                } else if (off < 0 || len < 0 || len > b.length - off) {
31                    throw new IndexOutOfBoundsException();
32                } else if (len == 0) {
33                    return 0;
34                }
35
36                int c = read();
37                if (c == -1) {
38                    return -1;
39                }
40                b[off] = (byte)c;
41
42                int i = 1;
43                try {
44                    for (; i < len ; i++) {
45                        c = read();
46                        if (c == -1) {
47                            break;
48                        }
49                        b[off + i] = (byte)c;
50                    }
51                } catch (IOException ee) {
52                }
53                return i;
54            }
55
56        }
57
58    }
59
60}
```

4.6.4 Estrutura e Participantes



- **AbstractClass (Relatorio)**
Classe que contém a definição do Template Method e o utiliza.
- **ConcreteClass (RelatorioFinanceiro, RelatorioHorasProjeto)**
Classes que customizam o comportamento abstrato.

Anotações

4.7 Visitor - Comportamento

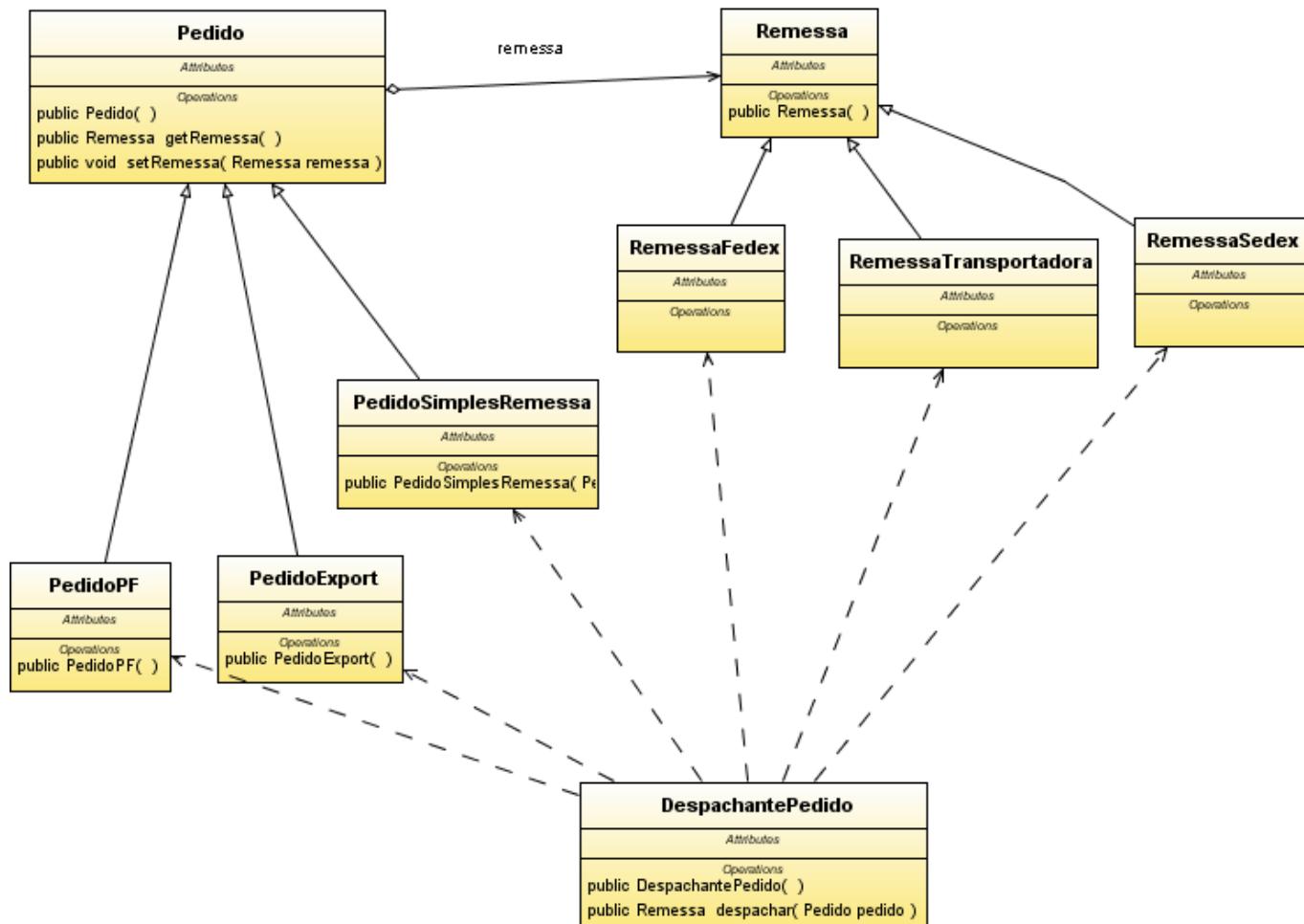
"Geralmente, as novas funcionalidades de um objeto seriam adicionadas nos objetos existentes, se não fosse o uso de design patterns"

O Visitor é um design pattern muito interessante que permite acrescentarmos operações em um objeto sem a necessidade de alterá-lo. Normalmente o Visitor representará uma operação comum que atuará em uma família de objetos de um mesmo tipo, podendo esta operação ser específica para sub-tipos daquela família de objetos. O Visitor é também uma maneira elegante de você reduzir radicalmente o uso de extensos `if(objeto instanceof X), else if(objeto instanceof Y)`, propondo um modelo onde uma classe Visitor poderá representar a operação para todos os seus sub-tipos e esta operação poderá ser acionada pelo objeto sem ele conhecer sua realização concreta.

4.7.1 Anti-pattern

Para podermos trabalhar o conceito deste design pattern, desenvolvemos dois exemplos em paralelo, um baseado em um sistema de pedidos e remessa e outro que dará continuidade no nosso sistema de Commands. Temos o seguinte modelo no sistema de pedidos, exemplo que utilizaremos primeiro:

Anotações



Podemos observar três classes principais:

1. Pedido: com três sub-tipos
2. Remessa: com três sub-tipos
3. DespachantePedido: que tem dependência com todos os pedidos e remessas pois nele concentrarmos a regra de criar diferentes remessas conforme o tipo do pedido.

Anotações

Para executar a operação de criação de remessa a classe DespachantePedido foi programada da seguinte forma:

Exemplo: DespachantePedido.java

```
1 package br.com.globalcode.cp.visitor.antipattern;
2
3 public class DespachantePedido {
4     public void despachar(Pedido pedido) {
5         if(pedido instanceof PedidoSimpleRemessa) {
6             pedido.setRemessa(new RemessaTransportadora());
7         } else if(pedido instanceof PedidoPF) {
8             pedido.setRemessa(new RemessaSedex());
9         } else if(pedido instanceof PedidoExport) {
10            pedido.setRemessa(new RemessaFedex());
11        }
12    }
13 }
```

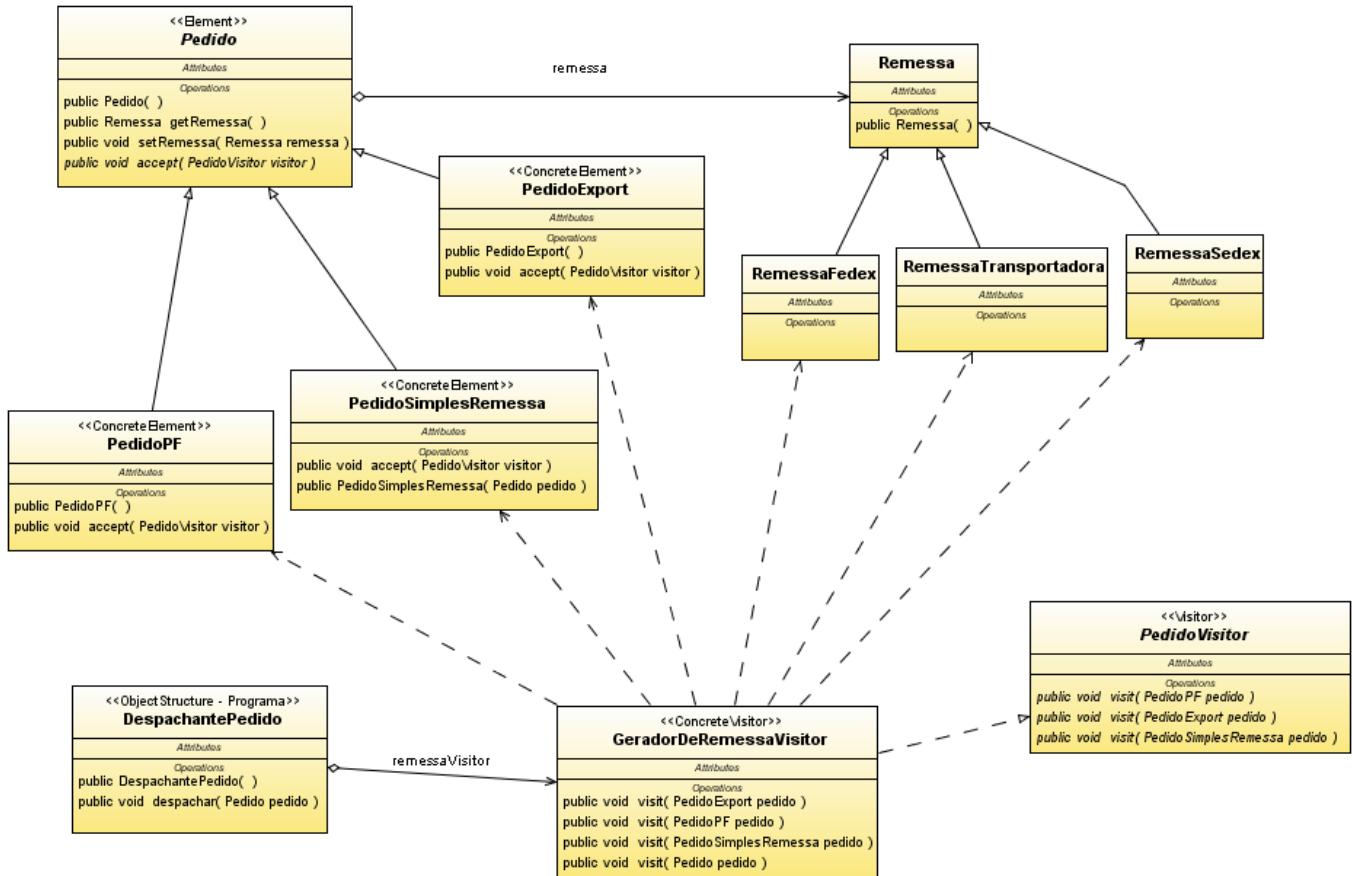
Portanto, este fragmento de código representa o anti-pattern do Visitor pois apresenta as seguintes consequências:

- A classe tende a reunir todas as regras de todas as tarefas associadas ao despacho de pedido;
- A classe tende a ficar confusa a medida que novos tipos de pedido são criados;
- Maior dificuldade entendimento do código;

Anotações

4.7.2 Aplicando o design pattern

Vamos apresentar o cenário de Pedido e Remessa agora refatorado para suportar Visitor. O seguinte diagrama representa o modelo:



Anotações

Repare no diagrama que agora temos o DespachantePedido sem dependências com as remessas e tipos de pedido, esta dependência foi transferida para a classe GeradorDeRemessaVisitor. Para entender melhor como funciona este modelo com Visitor vamos analisar seu código-fonte, começando pela definição da interface Visitor:

Exemplo: PedidoVisitor.java

```
-----  
1 package br.com.globalcode.cp.visitor;  
2  
3 public interface PedidoVisitor {  
4     public void visit(PedidoPF pedido);  
5     public void visit(PedidoExport pedido);  
6     public void visit(PedidoSimplesRemessa pedido);  
7 }
```

Podemos notar que a interface especifica um método para cada sub-tipo de Pedido, uma característica comum do Visitor pois ele tem o propósito de representar uma operação comum a todos os sub-tipos, que terá um comportamento diferente para cada um deles, porém estão reunidas na mesma classe.

Agora a implementação do Visitor na classe de gerenciamento de remessa. Para cada sub-tipo de Pedido estamos criando o tipo correto de Remessa:

Exemplo: GeradorDeRemessaVisitor.java

```
-----  
1 package br.com.globalcode.cp.visitor;  
2  
3 public class GeradorDeRemessaVisitor implements PedidoVisitor {  
4  
5     public void visit(PedidoExport pedido) {  
6         pedido.setRemessa(new RemessaFedex());  
7     }  
8     public void visit(PedidoPF pedido) {  
9         pedido.setRemessa(new RemessaSedex());  
10    }  
11    public void visit(PedidoSimplesRemessa pedido) {  
12        pedido.setRemessa(new RemessaTransportadora());  
13    }  
14    public void visit(Pedido pedido) {  
15        pedido.setRemessa(new Remessa());  
16    }  
17 }
```

Anotações

A classe Pedido, que representa o “alvo” da visita ou então o objeto visitado, deve dar o suporte para o visitante. Este suporte é tipicamente programado através de um método accept:

Exemplo: Pedido.java

```
1 package br.com.globalcode.cp.visitor;
2
3 public abstract class Pedido {
4     private Remessa remessa;
5
6     public Remessa getRemessa() {
7         return remessa;
8     }
9
10    public void setRemessa(Remessa remessa) {
11        this.remessa = remessa;
12    }
13
14    public abstract void accept(PedidoVisitor visitor);
15 }
```

Cada sub-classe de pedido vai implementar o método accept para que o método mais específico do Visitor seja chamado:

Exemplo: PedidoExport.java

```
1 package br.com.globalcode.cp.visitor;
2
3 public class PedidoExport extends Pedido{
4
5     public void accept(PedidoVisitor visitor) {
6         visitor.visit(this);
7     }
8
9 }
```

Exemplo: PedidoPF.java

```
1 package br.com.globalcode.cp.visitor;
2
3 public class PedidoPF extends Pedido{
4     public void accept(PedidoVisitor visitor) {
5         visitor.visit(this);
6     }
7 }
```

Anotações

Agora vamos analisar o código do despachante do pedido, que agora simplesmente configura o(s) Visitor(s) do Pedido e chama seu método accept:

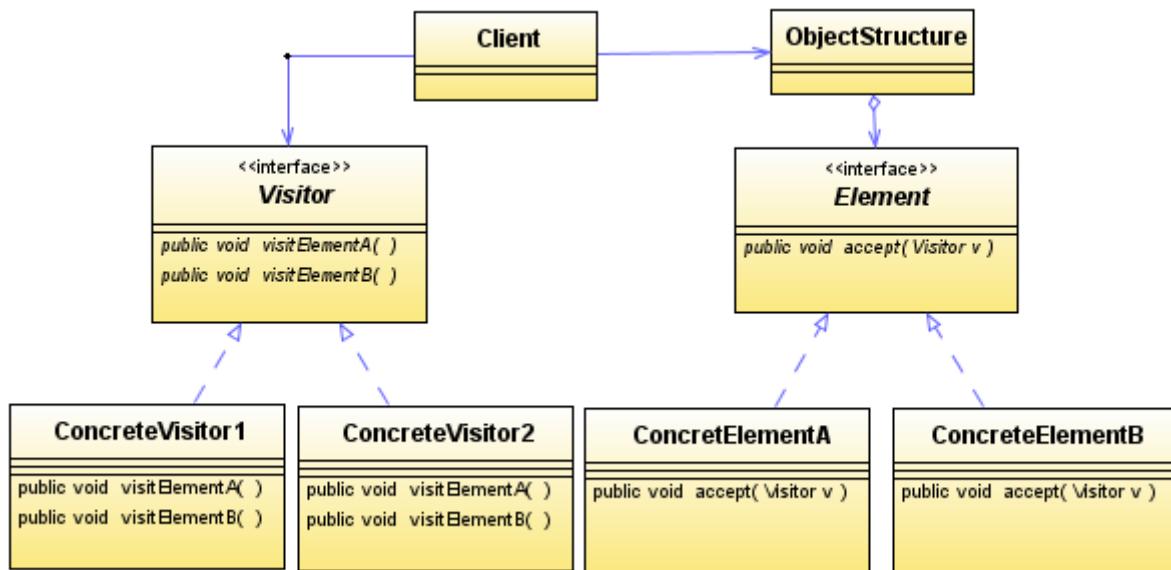
Exemplo: DespachantePedido.java

```
1 package br.com.globalcode.cp.visitor;
2
3 public class DespachantePedido {
4     GeradorDeRemessaVisitor remessaVisitor =
5         new GeradorDeRemessaVisitor();
6
7     public void despachar(Pedido pedido) {
8         pedido.accept(remessaVisitor);
9     }
10 }
```

Portanto podemos notar um grande ganho de extensibilidade na solução, pois facilmente podemos criar novos Visitors com novas regras para toda a família de pedidos sem alterar as classes de pedido. Podemos notar que o Visitor traz uma consequência de dificultar a adição de novas sub-classes de Pedido, pois para cada nova sub-classe teríamos que implementar um método na interface Visitor e consequentemente nas classes Visitor. Sendo assim podemos concluir que o Visitor é um pattern recomendado quando seu modelo tem estabilidade em termos de derivação e novas sub-classes, porém necessidade de flexibilidade de mudanças nos algoritmos programados nas sub-classes existentes.

Anotações

4.7.3 Estrutura e Participantes



- **Visitor (PedidoVisitor)**
Declara uma operação visit para cada classe *ConcreteElement*.
- **ConcreteVisitor (GeradorDeRemessaVisitor)**
São as implementações de Visitor que colecionarão comportamentos para um conjunto de sub-tipos de classes.
- **Element (Pedido)**
Define uma operação accept que aceita um Visitor como parâmetro.
- **ConcreteElement (PedidoPF, PedidoSimplesRemessa, PedidoExport)**
Implementa uma operação accept que aceita um Visitor como parâmetro.
- **ObjectStructure**
Elemento que pode ter seus elementos enumerados ou fornecer uma interface de alto nível para permitir ao visitante visitar seus elementos..

Anotações

4.8 Laboratório 2

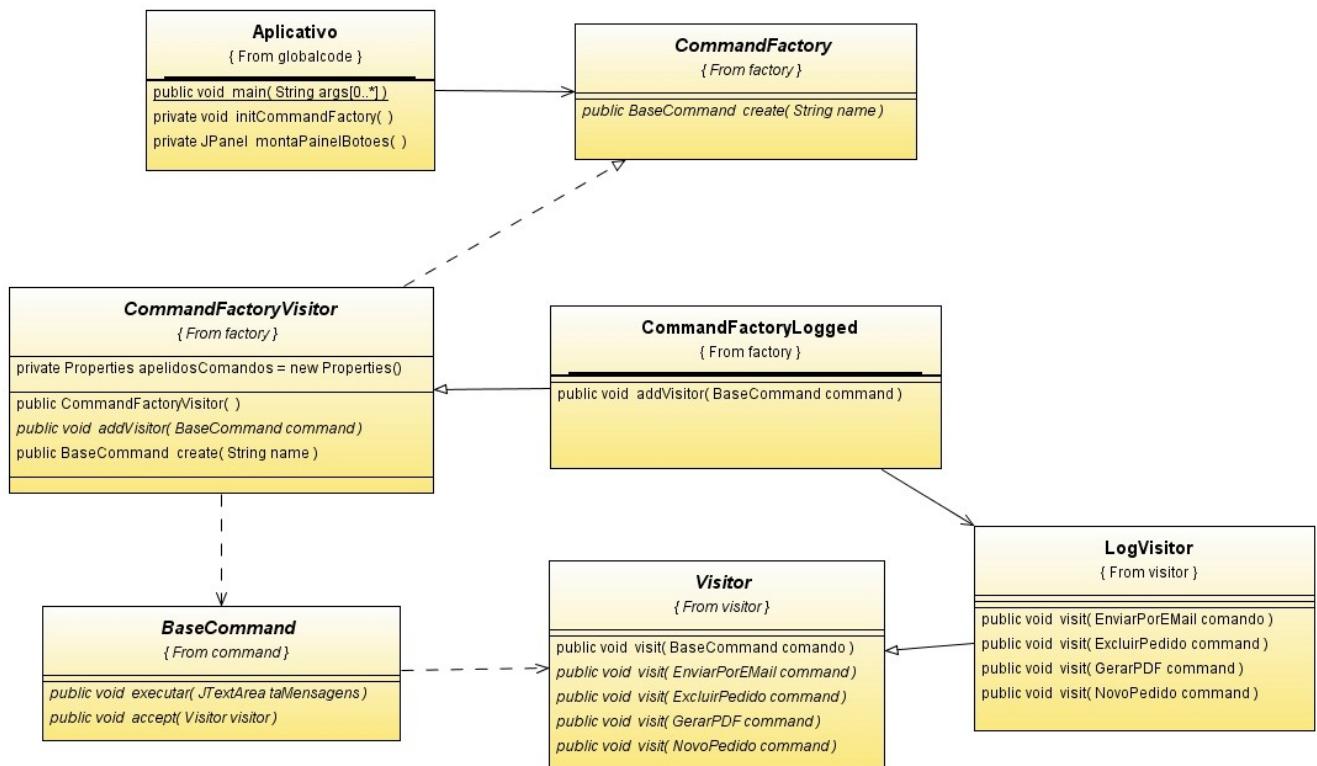


Objetivo:

Refatorar o modelo e aplicar os design-patterns Template Method e Visitor. A idéia é aplicarmos o Visitor permitindo que diferentes comportamentos sejam adicionados aos Commands, sem alterá-los.

Tabela de atividades:

Atividade	OK
1. Caso não tenha finalizado o laboratório anterior, ou deseja seguir com o código-fonte do curso, faça o download do arquivo lab01_sol.zip	
2. O Template Method deverá ser aplicado na classe resultado do refactoring, CommandFactoryVisitor, conforme diagrama a seguir. O método addVisitor será implementado por sub-classes de CommandFactoryVisitor e nela poderemos configurar diferentes Visitors.	

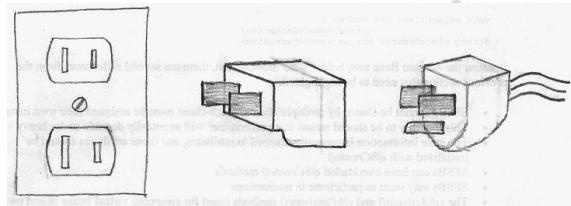


Anotações

4.9 Adapter - Estrutura

*Conhecido também como Wrapper

Com nome bastante sugestivo, este design pattern representa uma técnica para compatibilizar interfaces de objetos distintos porém com abstração semelhante, ou seja, podemos também visualizar como uma técnica de integração de objetos.



Geralmente vamos utilizar o Adapter e modelos de classes e objetos existentes. Podemos dizer que Adapter não é um design pattern de modelagem de novos sistemas, é um design pattern de modelagem reativa e não protótipica.

4.9.1 Anti-pattern

Mais uma vez vamos ilustrar a substituição de if por modelagem com polimorfismo. Imagine o cenário onde você tem uma definição por interface Java do padrão de classes da entidade Cliente. Duas implementações foram criadas com tal interface e existe uma classe de gerenciamento UIManagerClientes, que tem a responsabilidade de transmitir os dados dos clientes para os elementos de uma interface gráfica. Vejamos o diagrama UML:



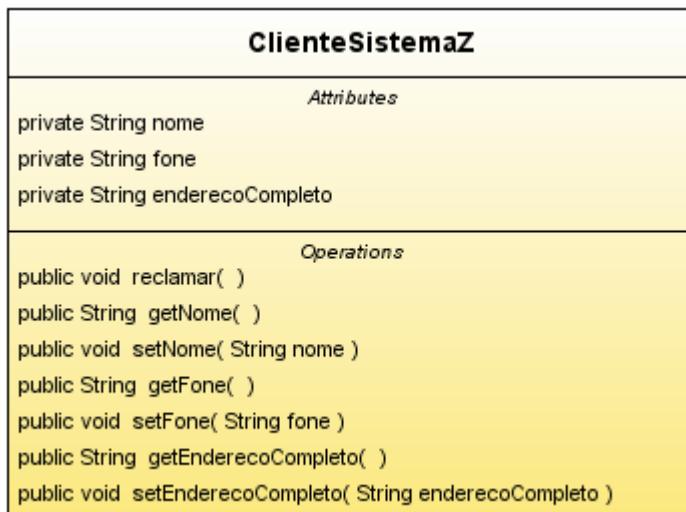
Anotações

Neste modelo temos 3 principais classes: Cliente (modelo domínio de negócio), UIManagerClientes (controle) e ClientePanel (visualização). A classe UIManagerClientes teria a seguinte codificação:

Exemplo: UIManagerClientes.java

```
1 package br.com.globalcode.cp.adapter;
2
3 public class UIManagerClientes {
4     public void apresentarClienteGUI(ClientePanel clientePanel, Cliente cliente) {
5         clientePanel.setRazaoSocial(cliente.getRazaoSocial());
6         clientePanel.setEndereco(cliente.getEndereco());
7         clientePanel.setTelefone(cliente.getTelefone());
8     }
9 }
```

Agora imagine que surgiu um novo sistema Java que deverá ser integrado ao sistema atual sem alterar o original. Neste sistema existe a seguinte implementação de Cliente:



Percebemos que apesar do Cliente apresentar as mesmas características e comportamentos, todos os nomes de métodos estão diferentes: `reclamar()` Vs. `registrarReclamacao`, `nome` Vs. `RazaoSocial`, `Fone` Vs. `Telefone` e `EnderecoCompleto` Vs. `Endereco`. Podemos imaginar que o anti-pattern do Adapter nesta situação seria alterar a classe `UIManagerClientes`, adicionando um novo método: `apresentarClienteGUI(ClientePanel clientePanel, ClienteSistemaZ cliente)`.

Anotações

Para resolver a situação sem o uso de Adapter, o seguinte código seria escrito:

Exemplo: UIManagerClientes.java

```
1 package br.com.globalcode.cp.adapter.antipattern;
2
3 public class UIManagerClientes {
4     public void apresentarClienteGUI(ClientePanel clientePanel, Cliente cliente) {
5         clientePanel.setRazaoSocial(cliente.getRazaoSocial());
6         clientePanel.setEndereco(cliente.getEndereco());
7         clientePanel.setTelefone(cliente.getTelefone());
8     }
9     public void apresentarClienteGUI(ClientePanel clientePanel,
10         ClienteSistemaZ cliente) {
11         clientePanel.setRazaoSocial(cliente.getNome());
12         clientePanel.setEndereco(cliente.getEnderecoCompleto());
13         clientePanel.setTelefone(cliente.getFone());
14     }
15 }
```

A consequência é que a camada de controle tende a crescer a medida que novas adaptações são feitas no sistema causando uma dispersão de lógicas associadas as classes de representação da entidade Cliente. Todo o sistema terá que tratar de uma forma o Cliente padrão, e de outra o ClienteSistemaZ e eventuais novos que surgirem. Este crescimento é bastante desordenado, uma vez que classes como a do Cliente, se comunicam com diversas partes do modelo.

Anotações

4.9.2 Aplicando o design pattern

Para solucionar o problema respeitando o padrão de interface Cliente já existente, vamos criar uma classe adaptadora do ClienteSistemaZ com o seguinte código:

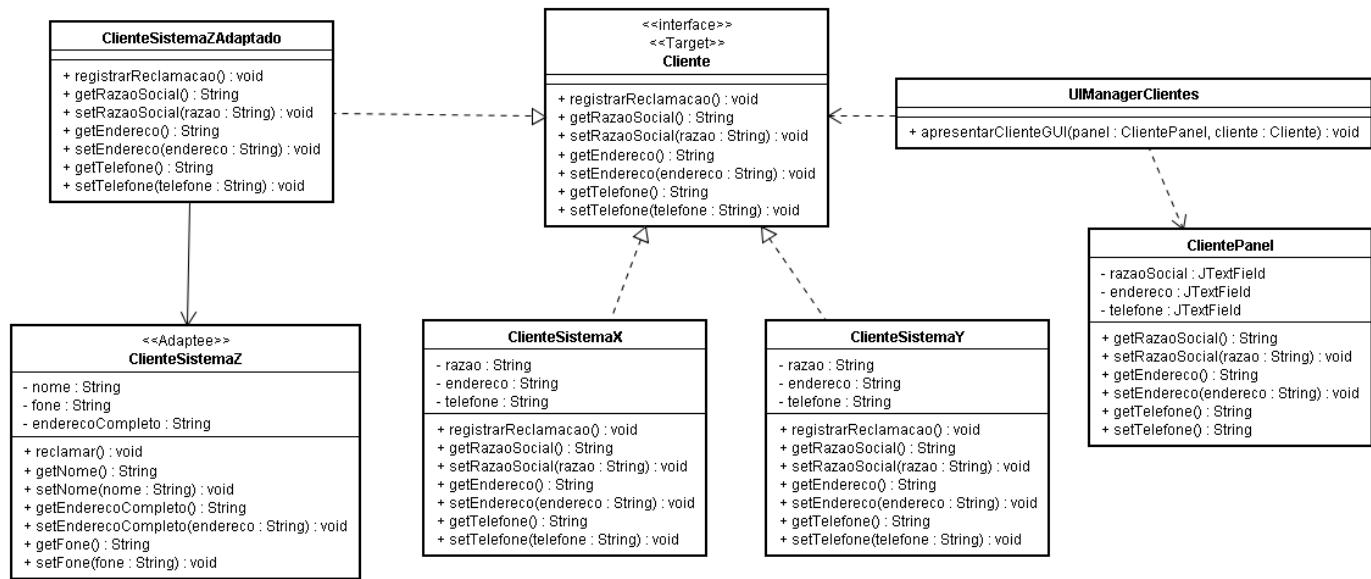
Exemplo: ClienteSistemaZAdaptado.java

```
1 package br.com.globalcode.cp.adapter;
2
3 public class ClienteSistemaZAdaptado implements Cliente {
4
5     private ClienteSistemaZ cliente;
6
7     public ClienteSistemaZAdaptado(ClienteSistemaZ cliente) {
8         this.cliente = cliente;
9     }
10
11    public void setTelefone(String s) {
12        cliente.setFone(s);
13    }
14    public void setRazaoSocial(String s) {
15        cliente.setNome(s);
16    }
17    public void setEndereco(String s) {
18        cliente.setEnderecoCompleto(s);
19    }
20    public void registrarReclamacao() {
21        cliente.reclamar();
22    }
23    public String getTelefone() {
24        return cliente.getFone();
25    }
26    public String getRazaoSocial() {
27        return cliente.getNome();
28    }
29    public String getEndereco() {
30        return cliente.getEnderecoCompleto();
31    }
32 }
```

Note que implementamos a interface Cliente e repassamos as chamadas para objeto ClienteSistemaZ. Esta classe faz o papel de tradutora de interfaces.

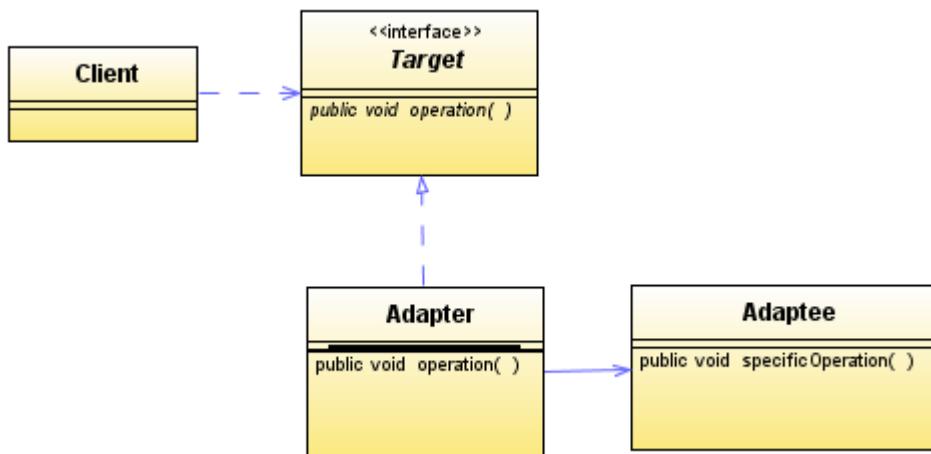
Anotações

Veja no diagrama UML como conseguimos isolar as particularidades da interface ClienteSistemaZ na classe Adapter não comprometendo as demais camadas do modelo:



Anotações

4.9.3 Estrutura e Participantes



- **Adaptee** (`ClienteSistemaZ`)

Classe com interface diferente que deverá ser adaptada para o modelo.

- **Adapter** (`ClienteSistemaZAdaptado`)

Classe que implementa o “de – para” da classe adaptada e a interface alvo.

- **Target** (`Cliente`)

Interface ou super-classe que estabelece a interface que deve ser seguida.

- **Client** (`UIManagerClientes`)

Programa usuário que trabalha com a interface alvo e se não fosse o pattern Adapter sofreria alterações a cada nova classe adaptada.

Anotações

4.10 Observer - Comportamento

*Conhecido também como Dependents, Publish-Subscribe

O design pattern Observer propõe um modelo de notificação de mudança de estado de um determinado objeto. Podemos imaginar que estamos construindo um framework que é configurado por arquivos properties e quando tais arquivos são alterados, gostaríamos que alguns objetos fossem notificados para atualizarem suas configurações. Esta situação poderá ser resolvida através do design pattern Observer.

Sistemas de notificação de mudança de estado são bastante úteis e adotados tanto em soluções corporativas como em ferramentas e diversos outros softwares. Com o desenvolvimento Web clássico (sem uso de AJAX), pouco podemos nos beneficiar deste design pattern para notificar a user-interface sobre mudanças de estado de objetos de negócio, porém agora com o uso crescente de tecnologias do AJAX a Web contará cada vez mais com telas que serão notificadas sobre eventuais mudanças de objetos críticos de negócio. Em contra-partida a tecnologia Java Swing permite a implementação deste design pattern para a construção de aplicativos associados ao design pattern de arquitetura Model-view-controller que será abordado posteriormente.

Anotações

4.10.1 Anti-pattern

Imagine um aplicativo que necessita observar mudanças em um arquivo (será um objeto `java.io.File`) e caso aconteça mudança de timestamp e/ou tamanho, ele deverá recarregar as informações de configuração e em seguida recarregar um suposto contexto de execução. Vejamos em código o anti-pattern de Observer:

Exemplo: Aplicativo.java

```
1 package br.com.globalcode.cp.observer.antipattern;
2
3 import java.io.File;
4
5 public class Aplicativo implements Runnable {
6     File configFile;
7     private long dataAtualizacao = 0;
8     public static void main(String[] args) {
9         new Thread(new Aplicativo()).start();
10    }
11    public Aplicativo() {
12        configFile = new File("config.properties");
13    }
14    public void reloadConfig() {
15        System.out.println("Recarregando configurações...");
16    }
17    public void reloadContext() {
18        System.out.println("Recarregando sessões do framework...");
19    }
20    public void run() {
21        observarArquivo();
22    }
23    public void observarArquivo() {
24        dataAtualizacao = configFile.lastModified();
25        while(true) {
26            if(dataAtualizacao != configFile.lastModified()) {
27                reloadConfig();
28                reloadContext();
29                dataAtualizacao = configFile.lastModified();
30            }
31            try {
32                Thread.sleep(500);
33            } catch(InterruptedException e) {
34                System.out.println("sleep interrompido");
35                break;
36            }
37        }
38    }
39 }
```

Aplicativo
<i>Attributes</i> package File configFile private long tamanhoAtual = 0
<i>Operations</i> <code>public void main(String args[0..1])</code> <code>public Aplicativo()</code> <code>public void reloadConfig()</code> <code>public void reloadContext()</code> <code>public void run()</code> <code>public void observarArquivo()</code>

O código apresentado acumula todas as responsabilidades de observar o arquivo e executar os métodos de reação na classe aplicativo. A principal consequência é que cada nova ação a ser tomada, em função da mudança do arquivo de configuração implica em novo código centralizado na classe aplicativo.

Anotações

Concluímos então que o mecanismo de extensão de novas ações ao evento de mudança de arquivo está bastante precário. Vamos refatorar o código aplicando o design pattern Observer.

4.10.2Aplicando o design pattern

A primeira classe que vamos criar será a classe ObservedFile. Esta classe representará nosso objeto alvo de observação, chamado de Subject, e portanto contará com um método de notificação dos seus observadores que é acionado assim que o arquivo sofre mudanças:

Exemplo: ObservedFile.java

```

1 package br.com.globalcode.cp.observer;
2
3 import java.util.ArrayList;
4
5 public class ObservedFile extends java.io.File implements Runnable {
6     private ArrayList<FileObserver> observers = new ArrayList<FileObserver>();
7     private long dataAtualizacao = 0;
8
9     private long INTERVALO_OBS = 500;
10    public ObservedFile(String name) {
11        super(name);
12        Thread observacao = new Thread(this);
13        observacao.start();
14    }
15    public void run() {
16        dataAtualizacao = this.lastModified();
17        while(true) {
18            if(dataAtualizacao != this.lastModified ()) {
19                notifyObservers();
20                dataAtualizacao = this.lastModified ();
21            }
22            try {
23                Thread.sleep(INTERVALO_OBS);
24            }
25            catch(InterruptedException e) {
26                System.out.println("sleep interrompido");
27                break;
28            }
29        }
30    }
31    public void addObserver(FileObserver f) {
32        this.observers.add(f);
33    }
34    public void notifyObservers() {
35        for(FileObserver observer : observers) {
36            observer.fileChanged(this);
37        }
38    }
39 }
```

Anotações

O código em destaque apresenta estrutura básica do sistema de observação, onde o observado terá uma coleção de observadores (linha 6 e linha 31 a 33) e no momento de mudança significativa de estado (linha 18 a 21), efetua a notificação percorrendo tal coleção de observadores (linha 34 até 37).

Agora vejamos o código que define a interface de um observador, que em geral chamamos de Listeners:

Exemplo: FileObserver.java

```
1 package br.com.globalcode.cp.observer;
2
3 import java.io.File;
4
5 public interface FileObserver {
6     void fileChanged(File f);
7 }
```

Vamos ter duas implementações de observadores sendo uma para recarga de configurações e outra para recarga de contextos:

Exemplo: ReloadConfig.java

```
1 package br.com.globalcode.cp.observer;
2
3 import java.io.File;
4
5 public class ReloadConfig implements FileObserver {
6     public void fileChanged(File f) {
7         System.out.println("Recarregando configurações...");
8     }
9 }
```

Exemplo: ReloadContext.java

```
1 package br.com.globalcode.cp.observer;
2
3 import java.io.File;
4
5 public class ReloadContext implements FileObserver {
6     public void fileChanged(File f) {
7         System.out.println("Recarregando contexto e connections...");
8     }
9 }
```

Agora o Aplicativo vai ficar bem mais enxuto e apenas estabelecerá a comunicação entre observado e observadores:

Anotações

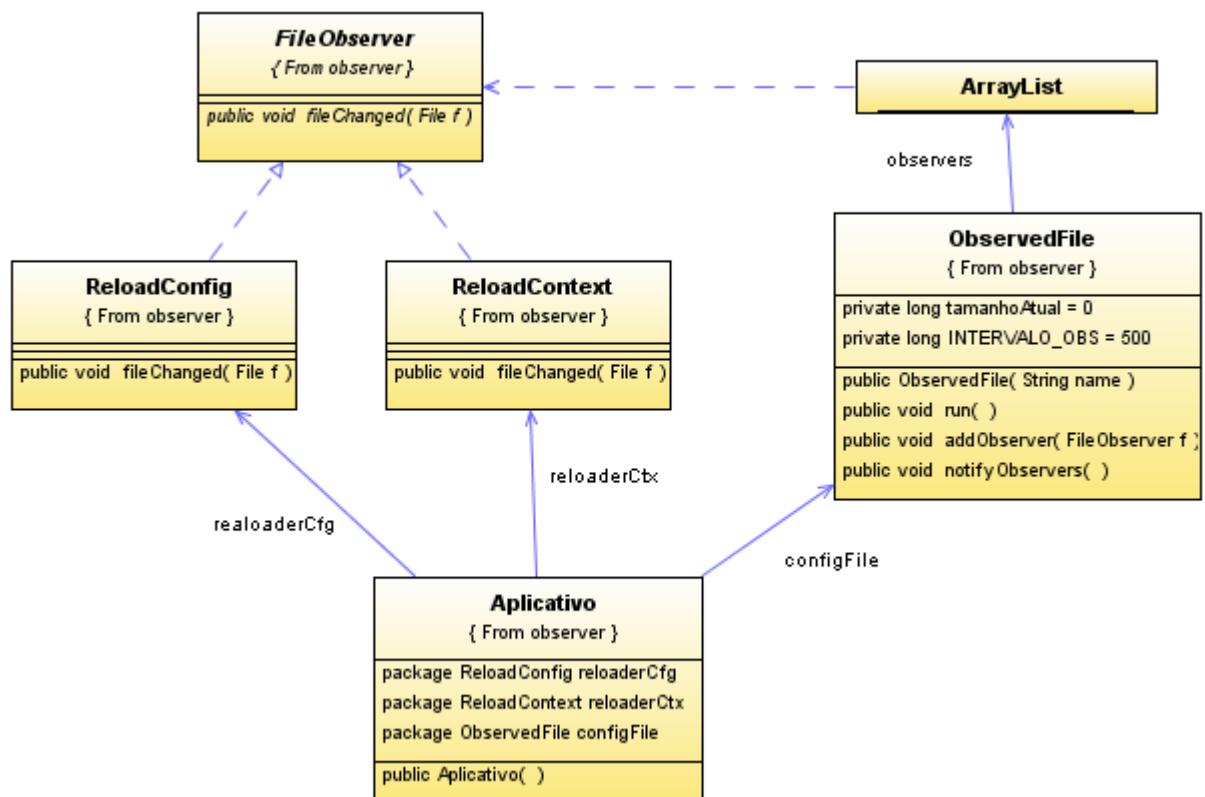
Exemplo: Aplicativo.java

```

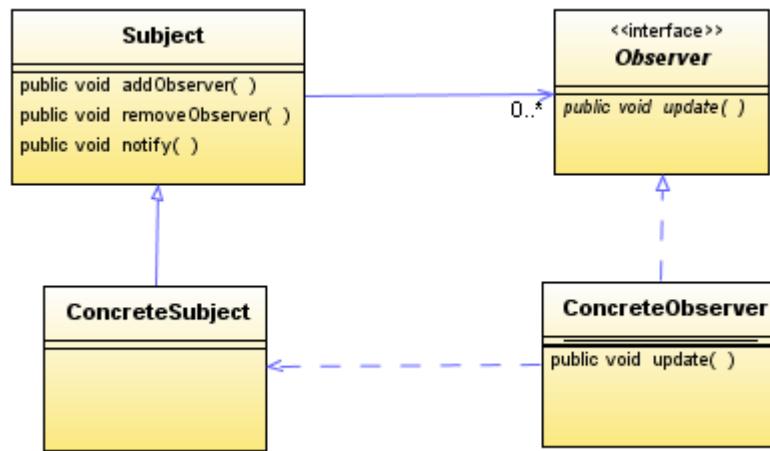
1 package br.com.globalcode.cp.observer;
2
3 public class Aplicativo {
4     ReloadConfig reloaderCfg;
5     ReloadContext reloaderCtx;
6     ObservedFile configFile;
7
8     public Aplicativo() {
9         reloaderCfg = new ReloadConfig();
10        reloaderCtx = new ReloadContext();
11        configFile = new ObservedFile("config.properties");
12        configFile.addObserver(reloaderCfg);
13        configFile.addObserver(reloaderCtx);
14    }
15}

```

Como principal consequência temos agora facilidade de adicionar novos observadores do arquivo de configurações sem acumular tais responsabilidades no Aplicativo. O seguinte diagrama UML representa a implementação apresentada:

**Anotações**

4.10.3 Estrutura e Participantes



- **Subject (ObservedFile)**
Representa a abstração do objeto alvo da observação.
- **Observer (FileObserver)**
Interface dos objetos que serão notificados em caso de mudança do observado.
- **ConcreteSubject (ObservedFile)**
Implementação do objeto alvo da observação.
- **ConcreteObserver (ReloadConfig, ReloadContext)**
Implementação de observadores.

Anotações

4.11 Laboratório 3

Objetivo:

Vamos seguir com a implementação do nosso framework de Commands agregando um sistema de recarregamento de arquivos de configurações, além de prover uma adaptação para um comando criado em outro modelo de classes.



Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab03.zip e descompacte na sua pasta;	
2. Analise a classe <code>AcionarSistemaContabil.java</code> ;	
3. Analise o método <code>buttonIntegrarContabil_clicked()</code> da classe <code>Aplicativo.java</code> ;	
4. Faça um refatoramento no código para aplicar o design pattern Adapter e criar uma classe de adaptação do comando <code>AcionarSistemaContabil</code> . Em seguida altere a classe <code>Aplicativo</code> para que a criação do comando de sistema contábil seja feita através da factory.	
5. Agora você deverá refatorar a classe <code>Aplicativo</code> , de forma que ela seja capaz de identificar mudanças no arquivo <code>config.properties</code> e efetuar o recarregamento caso aconteça.	

Anotações

4.12 Mediator - Comportamento

“Embora o particionamento de um sistema em muitos objetos geralmente melhore a reusabilidade, a proliferação de interconexões tende a reduzi-la novamente.”

O objetivo do Mediator é centralizar comunicações e interfaceamentos complexos de objetos reduzindo o acoplamento e encapsulando o comportamento coletivo dos objetos.

De certa forma Mediator pode se confundir com outros design patterns, em especial o Façade (que será estudado posteriormente), porém algumas características diferenciam bastante o Mediator:

1. Promove comunicação entre objetos multidirecional;
2. Pode-se aplicar Observer para notificações via Mediator;
3. Façade é unidirecional e tem objetivo de abstrair um subsistema para simplificar interfaceamento;
4. Façade é para estrutura enquanto mediator é de comportamento;

4.12.1 Anti-pattern

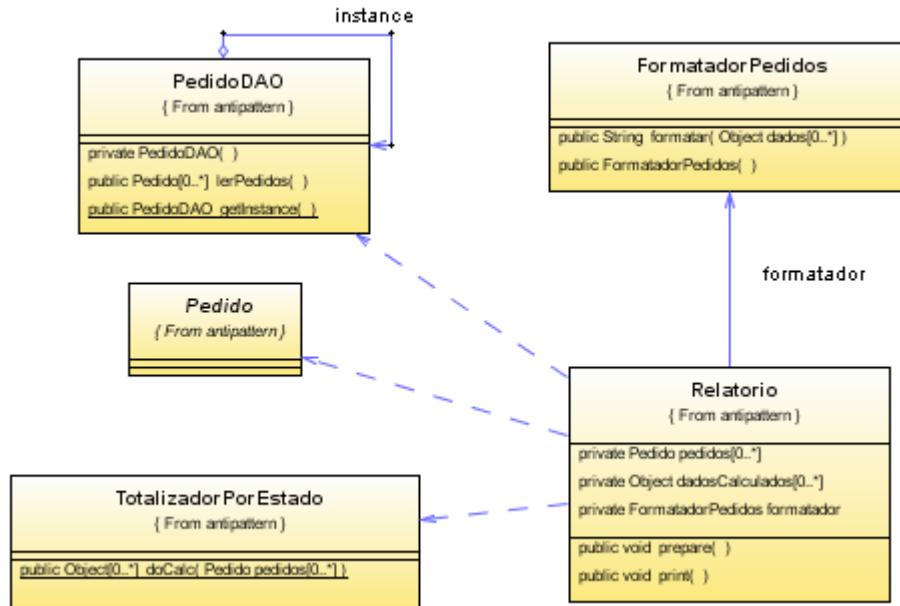
Como cenário para o Mediator, vamos exemplificar uma classe que supostamente confecciona relatórios gerenciais com base em uma coleção de pedidos. Relatórios são componentes que tendem a ser programados em classes que se comunicam com diversos objetos necessários para a confecção do relatório: objetos de negócio, totalizadores, formatadores, conexões, etc. Vejamos o código a seguir:

Exemplo: Relatorio.java

```
-->-----<-----  
| 1 package br.com.globalcode.cp.mediator.antipattern;  
| 2 import java.util.Collection;  
| 3  
| 4 public class Relatorio {  
| 5     private Collection<Pedido> pedidos;  
| 6     private Collection<Object> dadosCalculados;  
| 7     private FormatadorPedidos formatador;  
| 8  
| 9     public void prepare() {  
|10         pedidos = PedidoDAO.getInstance().lerPedidos();  
|11         dadosCalculados = TotalizadorPorEstado.doCalc(pedidos);  
|12     }  
|13     public void print() {  
|14         System.out.println(formatador.formatar(dadosCalculados));  
|15     }  
|16 }
```

Anotações

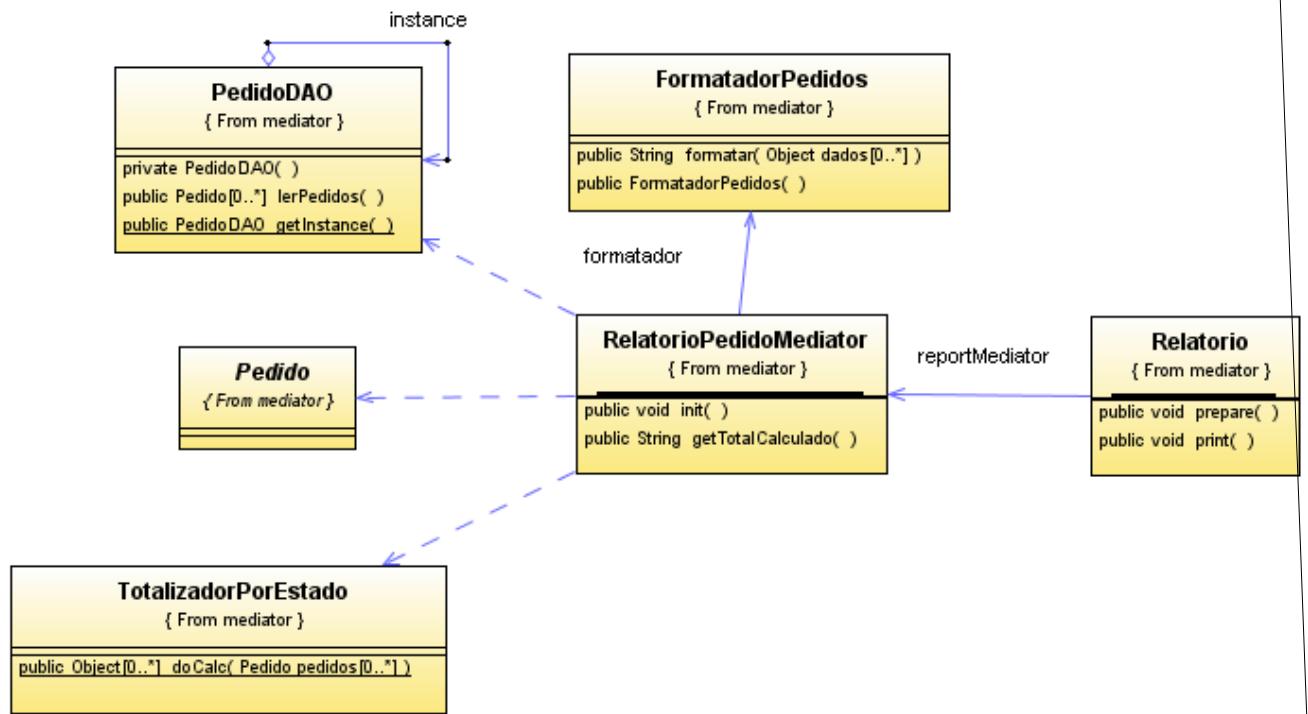
Percebemos que a classe Relatório é dependente de Pedido, TotalizadorPorEstado e FormatadorPedidos, ou seja, mudanças nessas classes podem afetar o relatório diretamente e esta comunicação tende a aumentar a medida que a complexidade do relatório aumenta. Vejamos o diagrama UML representando o modelo do código acima e suas dependências:



Anotações

4.12.2Aplicando o design pattern

Vamos refatorar o código acrescentando então uma classe Mediator entre Relatório e os demais. Como resultado, teremos o seguinte diagrama:



O código da classe Relatorio será:

Exemplo: Relatorio.java

Anotações

A classe Mediator vai negociar com todas as demais necessárias para o relatório:

Exemplo: RelatorioPedidoMediator.java

```
1 package br.com.globalcode.cp.mediator;
2
3 import java.util.Collection;
4
5 public class RelatorioPedidoMediator {
6     private Collection<Pedido> pedidos;
7     private Collection<Object> dadosCalculados;
8     private FormatadorPedidos formatador;
9
10    public void init() {
11        pedidos = PedidoDAO.getInstance().lerPedidos();
12        dadosCalculados = TotalizadorPorEstado.doCalc(pedidos);
13    }
14
15    public String getTotalCalculado() {
16        return formatador.formatar(dadosCalculados);
17    }
18 }
```

Desta forma movemos o acoplamento da comunicação entre esses objetos para o Mediator de forma que nosso relatório poderá ser reutilizado independente de algoritmo de totalização e também independente de tipo de objeto Pedido. Vale lembrar que a comunicação com Mediator pode ser bilateral, ou seja, se não trata-se de um gerador de relatório mas sim um diálogo com o usuário, poderíamos pensar em um sistema onde o Mediator cuida do processo de notificar os componentes da user-interface de eventuais mudanças. Para isso, devemos associar também o design pattern observer.

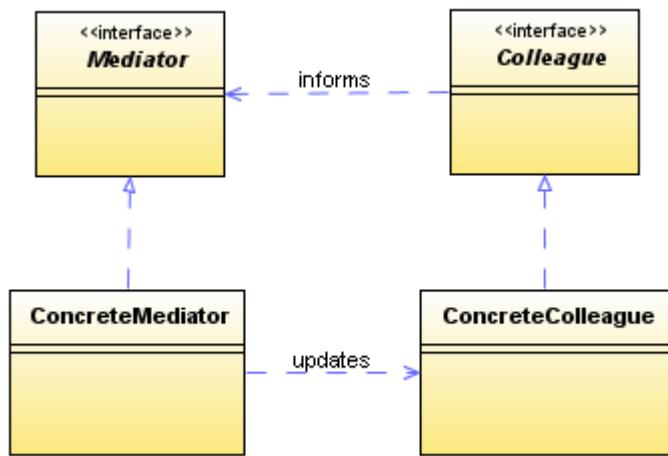
4.12.3 Mediator e user-interfaces

O uso de Mediator para manipulação de user-interface é bastante popular, podemos delegar para um Mediator a responsabilidade de mudar o estado da user-interface, por exemplo de estado de visualização para estado de edição.

Com Swing é comum desenvolvermos classes associadas a componentes de tela como caixas de texto, listagem, checkbox, etc. Tratamos seus eventos via listeners e é comum mudarmos o estado da tela baseado em ações do usuário. Este comportamento poderia ser transferido para um Mediator com o objetivo de tornar a classe que representa a user-interface mais enxuta e desacoplada dos demais objetos intermediados.

Anotações

4.12.4 Estrutura e Participantes



- **Mediator** (`RelatorioPedidoMediator`)

Define a interface de comunicação do Mediator. A interface deve ser separada da implementação quando mais de um tipo de Mediator for implementado.

- **ConcreteMediator** (`RelatorioPedidoMediator`)

Representa uma implementação do Mediator.

- **Colleague classes** (`Pedido`, `FormatadorDePedidos`, `PedidoDAO`)

São as classes intermediadas.

Anotações

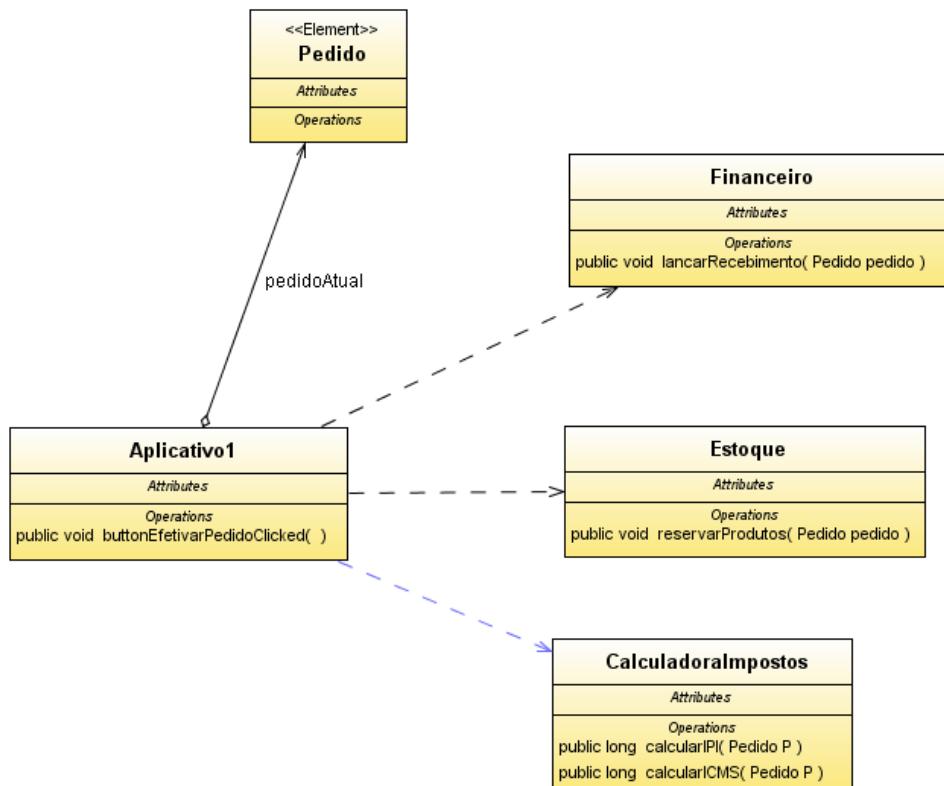
4.13 Façade - Estrutura

Façade encapsula o acesso a uma funcionalidade de um sub-sistema ou módulo, tornando o acesso mais simples e a interface mais amigável. Apesar de contar com uma definição bastante simples, o design pattern Façade pode ser confundido com outros design patterns se não prestarmos atenção em algumas características importantes.

Em diversas ocasiões dentro de cenários corporativos, precisamos executar uma operação / transação que envolve diferentes recursos de um determinado sub-sistema. Podemos imaginar que em determinada ocasião, para efetivar um pedido, é necessário calcular seus impostos, reservar produtos no estoque, agendar entrega e criar um objeto que representa o crédito financeiro da operação. Se pensarmos que diversas classes irão efetivar pedidos, seria interessante criarmos uma classe que tenha a responsabilidade de chamar a seqüência de objetos envolvidos e devolver uma resposta, de forma simples e objetiva, através de um interfaceamento mais simples e com menor acoplamento entre objetos.

4.13.1 Anti-pattern

Vamos colocar em prática o exemplo citado onde temos um processo de efetivar pedidos que envolve um conjunto de objetos de um determinado sub-sistema, conforme diagrama:

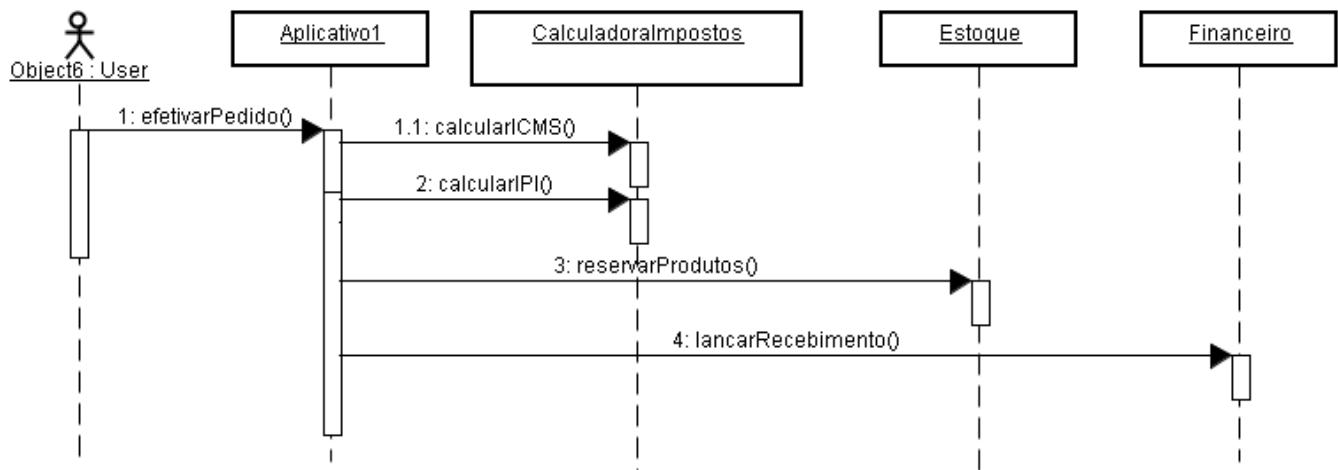


O seguinte código representa o anti-pattern do Façade para este modelo;
Anotações

Exemplo: Aplicativo1.java

```
1 package br.com.globalcode.cp.facade.antipattern;
2
3 public class Aplicativo1 {
4     Pedido pedidoAtual;
5
6     public void buttonEfetivarPedidoClicked() {
7         CalculadoraImpostos calc = new CalculadoraImpostos();
8         Estoque estoque = new Estoque();
9         Financeiro financeiro = new Financeiro();
10        long imposto=0;
11        imposto+=calc.calcularICMS(pedidoAtual);
12        imposto+=calc.calcularIPI(pedidoAtual);
13        estoque.reservarProdutos(pedidoAtual);
14        financeiro.lancarRecebimento(pedidoAtual);
15    }
16 }
```

Neste caso a seqüência para efetivar um pedido seria:



Podemos notar que o aplicativo em questão está acoplado com um conjunto grande de interfaces, no entanto seu único objetivo é efetivar o pedido em questão. Adicionalmente podemos imaginar que o sistema pode dispor de diferentes interfaces para efetivar pedidos, sendo então provável que a chamada a este conjunto de objetos se repita em outra classe.

Anotações

4.13.2 Aplicando o design pattern

Para solucionar este problema vamos criar uma classe que simplificará o acesso à operação e vai orquestrar a chamada de um conjunto de objetos do sub-sistema em questão. Esta classe vai representar uma fachada para este processo, tornando-o mais simples de se acessar, com menor acoplamento entre o cliente e as classes envolvidas no processo e ainda facilitando a gestão de transações e controle de segurança quando necessário.

O seguinte código representa a classe Façade da solução:

Exemplo: PedidoFacade.java

```
1 package br.com.globalcode.cp.facade;
2
3 public class PedidoFacade {
4
5     public void efetivarPedido(Pedido pedidoAtual) {
6         CalculadoraImpostos calc = new CalculadoraImpostos();
7         Estoque estoque = new Estoque();
8         Financeiro financeiro = new Financeiro();
9         long imposto=0;
10        imposto+=calc.calcularICMS(pedidoAtual);
11        imposto+=calc.calcularIPI(pedidoAtual);
12        estoque.reservarProdutos(pedidoAtual);
13        financeiro.lancarRecebimento(pedidoAtual);
14    }
15 }
```

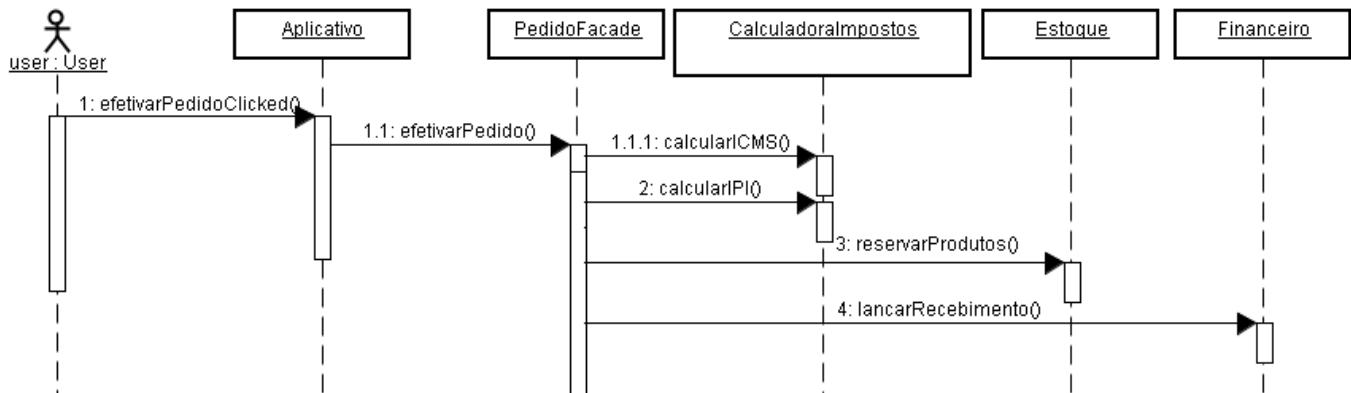
A classe Aplicativo seria reduzida para:

Exemplo: Aplicativo1.java

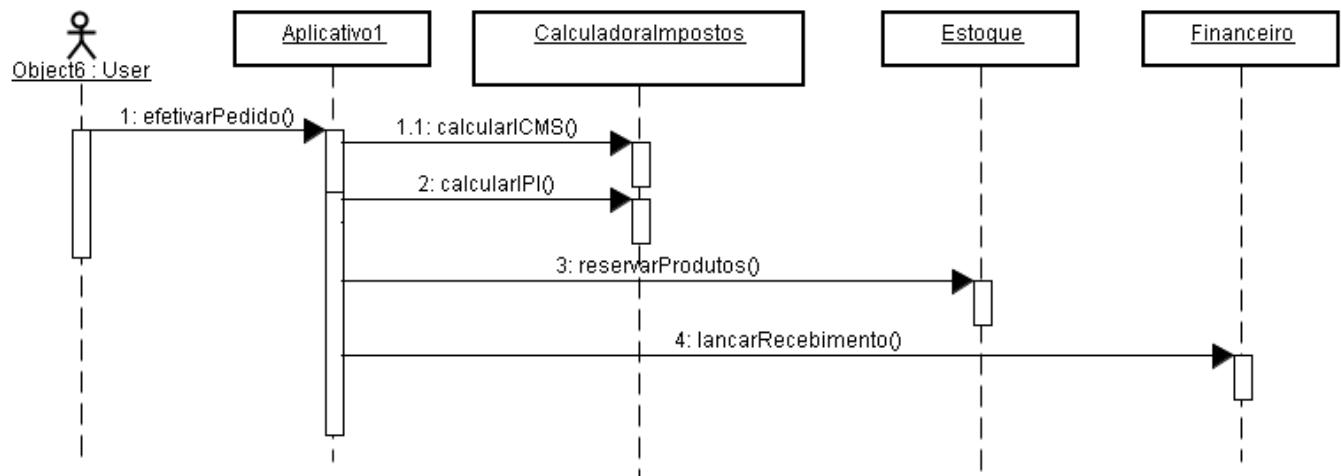
```
1 package br.com.globalcode.cp.facade;
2
3 public class Aplicativo1 {
4     Pedido pedidoAtual;
5
6     public void buttonEfetivarPedidoClicked() {
7         PedidoFacade pedidoFacade = new PedidoFacade();
8         pedidoFacade.efetivarPedido(pedidoAtual);
9     }
10 }
```

Anotações

O seguinte diagrama de seqüência representa o resultado do refatoramento:



Se comparado com o diagrama anterior, percebemos que o Aplicativo utilizando Façade fica desacoplado das classes que fazem parte do processo interno de efetivação do pedido.



Anotações

4.13.3 Façade Vs. Mediator

Vários motivos podem levar você a questionar as diferenças entre os design patterns Façade e Mediator, uma vez que ambos acabam por encapsular a comunicação de um conjunto de objetos. No entanto, existem diferenças que clarificam bastante:

1. Façade promove a idéia de expor processos de um sub-sistema encapsulados por uma fachada;
2. Esta comunicação é unilateral;
3. Façade é um design pattern de estrutura;
4. Mediator promove a idéia de encapsular o comportamento coletivo de objetos, independente de sub-sistema, em um sistema de comunicação que pode ser bilateral, inclusive com notificações através do uso de Observer;
5. Mediator é um design pattern de comportamento;
6. Os benefícios de Mediator são: menor acoplamento, maior probabilidade de reuso em função da intercomunicação de objetos ficar centralizada, possibilidade de comunicação bilateral entre cliente e objetos intermediados;
7. Os benefícios de Façade são: simplificação da interface para sub-sistemas complexos, maior possibilidade controle transacional e de segurança e menor acoplamento.
8. Por suas características, Façade é um design pattern bastante aplicado na computação distribuída, quando clientes se comunicam com sub-sistemas remotos.

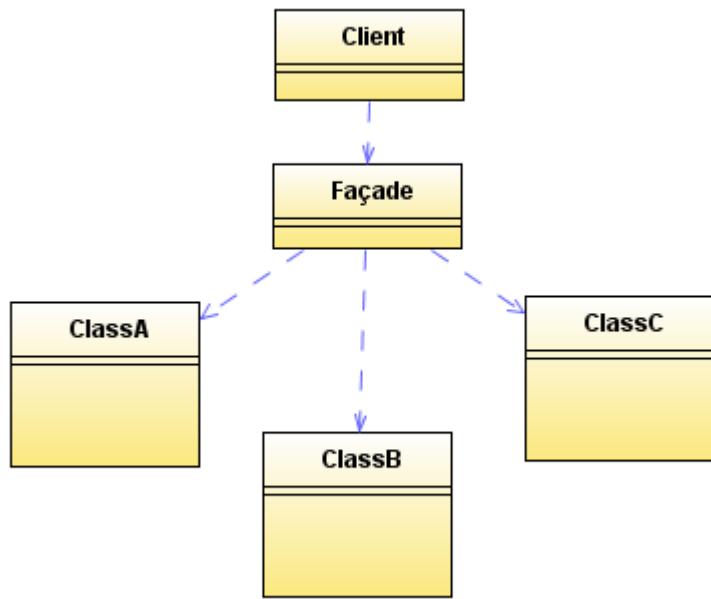
4.13.4 Façade na computação distribuída

Quando temos processos em sub-sistemas distribuídos, como é o caso quando usamos EJB's, o Façade é bastante valioso. Alguns processos podem envolver a chamada a diferentes componentes dentro de um mesmo servidor. Sem o uso de Façade o cliente teria que chamar os diferentes componentes através de múltiplas requisições de rede. Ao usarmos Façade neste cenário, um único componente será exposto como uma fachada remota para o processo e ele localmente chamará os demais componentes envolvidos na transação. Esta é uma derivação do uso de Façade na plataforma Java Enterprise e o design pattern localizado é chamado de Session Façade.

Ao desenvolver um endpoint WebService para um determinado serviço / funcionalidade, muitas vezes acabamos por criar um fachada para um conjunto de componentes complexos de um sub-sistema, portanto, podemos afirmar que algumas vezes quando desenvolvemos WebServices, estamos conscientemente ou inconscientemente usando Façade.

Anotações

4.13.5 Estrutura e Participantes



- **Façade (PedidoFacade)**

Conhece as classes do sub-sistema responsáveis por atender determinada solicitação e delega as chamadas do cliente para as classes adequadas.

- **Classes de sub-sistema (CalculadoraImpostos, Estoque e Financeiro)**

Classes que são chamadas pelo *Façade*.

Anotações

4.14 Laboratório 4

Objetivo:

Praticar os design patterns Mediator e Façade através do refactoring de uma tela de cadastro de produtos.

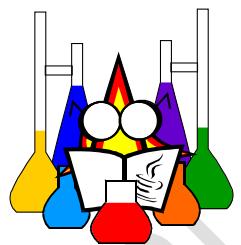


Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab04.zip e descompacte na sua pasta;	
2. Abra o projeto no NetBeans	
3. Execute o projeto pressionando F6 e analise o comportamento da tela de cadastro de produtos.	
4. Esta tela utiliza um listener de eventos para cada botão, e vários dos botões afetam diversos elementos da tela, habilitando ou não sua utilização. Faça o refactoring da classe br.com.globalcode.gui.CatalogoProdutos para que utilize um único Listener que funcione como Mediator da interface gráfica. Para isso crie uma InnerClass chamada CadastroProdutosMediator.	
5. Na própria classe da interface gráfica está sendo feito acesso ao banco de dados através da API JDBC. Imagine que na sua equipe de desenvolvimento existem especialistas em interface gráfica, mas que não conhecem nada de banco de dados. Faça o refactoring do aplicativo para utilizar um Façade para a API de JDBC. Utilize a classe br.com.globalcode.facade.JDBCFacade, preenchendo seus métodos lerProdutos e salvarProduto.	

Anotações

4.15 Composite - Estrutura

Composite é um clássico design pattern de estrutura de objetos onde temos uma árvore que o representa como partes-todo. Temos muitos exemplos de aplicação de Composite em diversos cenários:

- Sistema de diretórios e arquivos;
- Container e componentes Swing;
- Página JSP / JSF;
- Documentos XML;

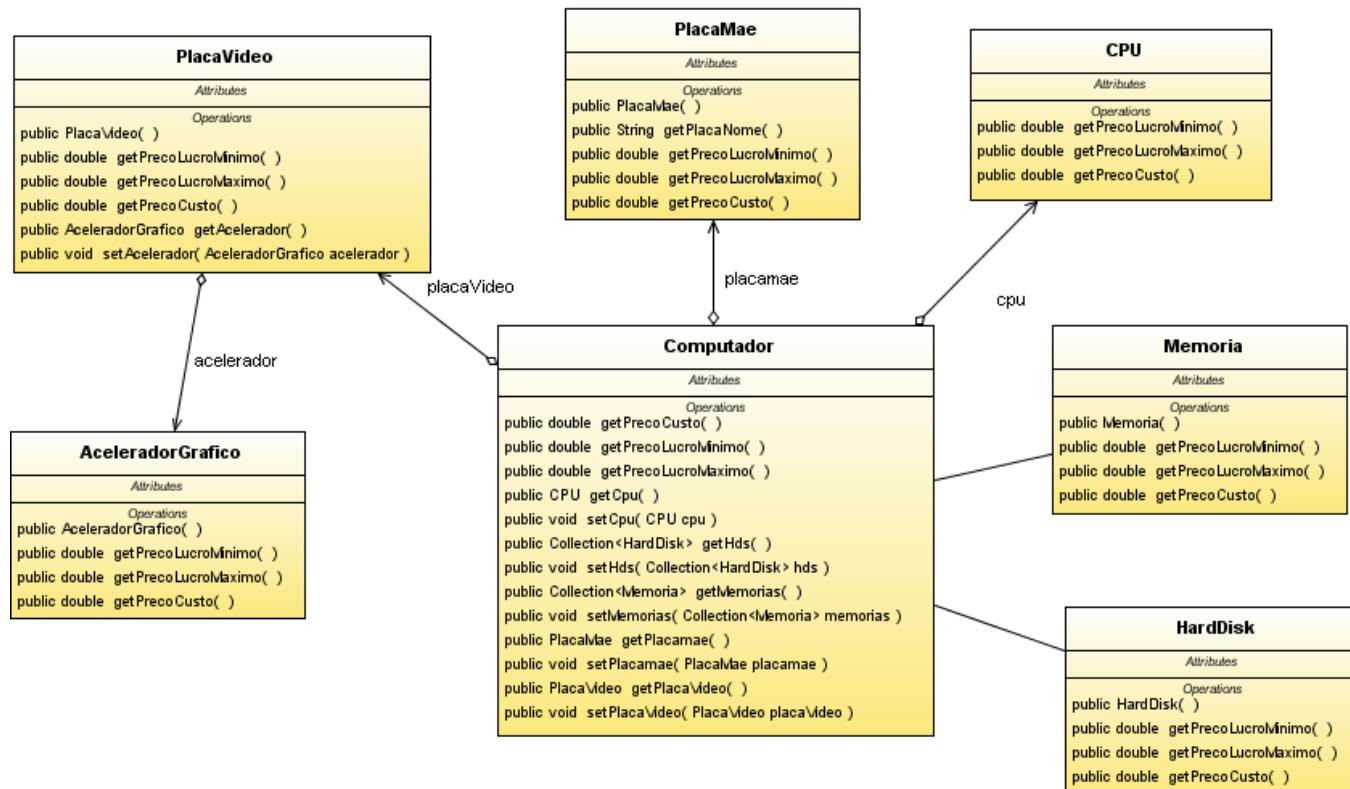
Tipicamente modelos Composite trabalham com sistemas de nós que são chamados de árvore. Em uma árvore de objetos podemos ter nós do tipo galho, que poderão conter outros nós, e nós do tipo folha, que não podem conter outros nós.

Anotações

4.15.1 Anti-pattern

Vamos imaginar que estamos montando um sistema de vendas de computadores que são configurados conforme as necessidades do cliente. Neste caso, teríamos os componentes comuns de um computador: placa-mãe, memória, hard-disk, CPU e placa de vídeo.

Neste caso, sem aplicar o design pattern composite e sem grandes critérios técnicos, podemos imaginar o seguinte modelo como anti-pattern do Composite:



O modelo apresenta uma classe Computador que está diretamente associada com todos seus componentes. Para adicionar um novo componente no computador precisaremos alterar a interface da classe Computador, adicionando a associação e também teremos que mudar a regra de cálculo de preços para que considere este novo componente.

Anotações

Exemplo: Computador.java

```
1 package br.com.globalcode.cp.composite.antipattern;
2
3 import java.util.Collection;
4
5 public class Computador {
6     private CPU cpu;
7     private Collection<HardDisk> hds;
8     private Collection<Memoria> memorias;
9     private PlacaMae placamae;
10    private PlacaVideo placaVideo;
11
12    public double getPrecoCusto() {
13        double preco = 0;
14        preco += getCpu().getPrecoCusto();
15        preco += getPlacamae().getPrecoCusto();
16        preco += getPlacaVideo().getPrecoCusto();
17        for(HardDisk hd: hds) {
18            preco+=hd.getPrecoCusto();
19        }
20        for(Memoria m:memorias) {
21            preco+=m.getPrecoCusto();
22        }
23        return preco;
24    }
25    public double getPrecoLucroMinimo() {
26        double preco = 0;
27        preco += getCpu().getPrecoLucroMinimo();
28        preco += getPlacamae().getPrecoLucroMinimo();
29        preco += getPlacaVideo().getPrecoLucroMinimo();
30        for(HardDisk hd: hds) {
31            preco+=hd.getPrecoLucroMinimo();
32        }
33        for(Memoria m:memorias) {
34            preco+=m.getPrecoLucroMinimo();
35        }
36        return preco;
37    }
38    public double getPrecoLucroMaximo() {
39        double preco = 0;
40        preco += getCpu().getPrecoLucroMaximo();
41        preco += getPlacamae().getPrecoLucroMaximo();
42        preco += getPlacaVideo().getPrecoLucroMaximo();
43        for(HardDisk hd: hds) {
44            preco+=hd.getPrecoLucroMaximo();
45        }
46        for(Memoria m:memorias) {
47            preco+=m.getPrecoLucroMaximo();
48        }
49        return preco;
50    }
51 ... métodos getters e setters omitidos
52    public CPU getCpu() {
53        return cpu;
54    }
```

Anotações

```
55
56    public void setCpu(CPU cpu) {
57        this.cpu = cpu;
58    }
59
60    public Collection<HardDisk> getHds() {
61        return hds;
62    }
63    Copyright 2009 Globalcode – The Developers Company, todos os direitos reservados
64    public void setHds(Collection<HardDisk> hds) {
65        this.hds = hds;
```

```
67  
68     public Collection<Memoria> getMemorias() {  
69         return memorias;  
70     }  
71 }
```

4 Design Patterns GOF

```
72     public void setMemorias(Collection<Memoria> memorias) {  
73         this.memorias = memorias;  
74     }  
75  
76     public PlacaMae getPlacamae() {  
77         return placamae;  
78     }  
79  
80     public void setPlacamae(PlacaMae placamae) {  
81         this.placamae = placamae;  
82     }  
83  
84     public PlacaVideo getPlacaVideo() {  
85         return placaVideo;  
86     }  
87  
88     public void setPlacaVideo(PlacaVideo placaVideo) {  
89         this.placaVideo = placaVideo;  
90     }  
91 }  
92  
93 }
```

Anotações

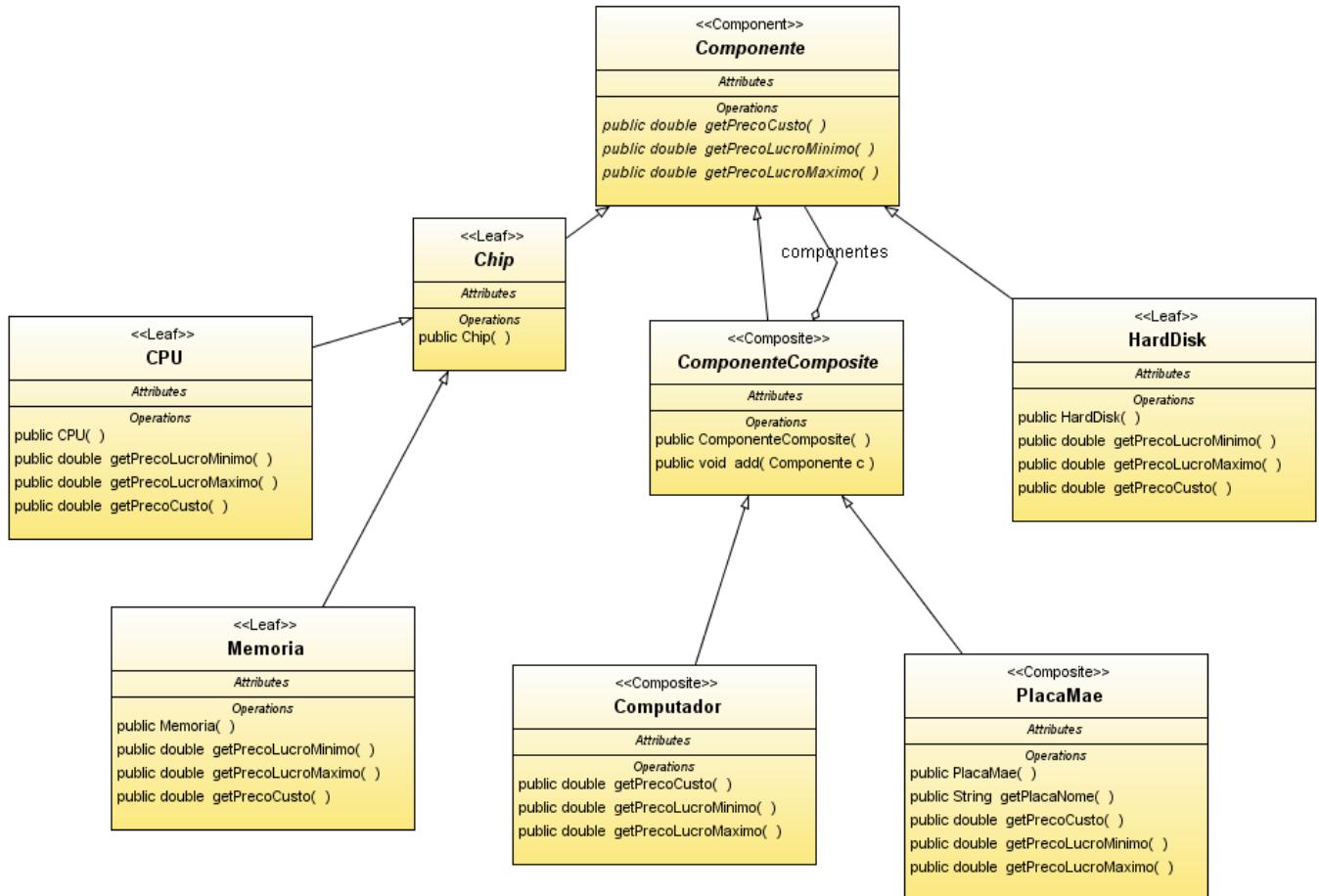
4.15.2Aplicando o design pattern

Quando aplicamos o design pattern Composite, criamos uma estrutura de classes e sub-classes que serão divididas em:

Classes composite: que poderão conter outros componentes;

Classes leaf: que serão componentes finais na hierarquia;

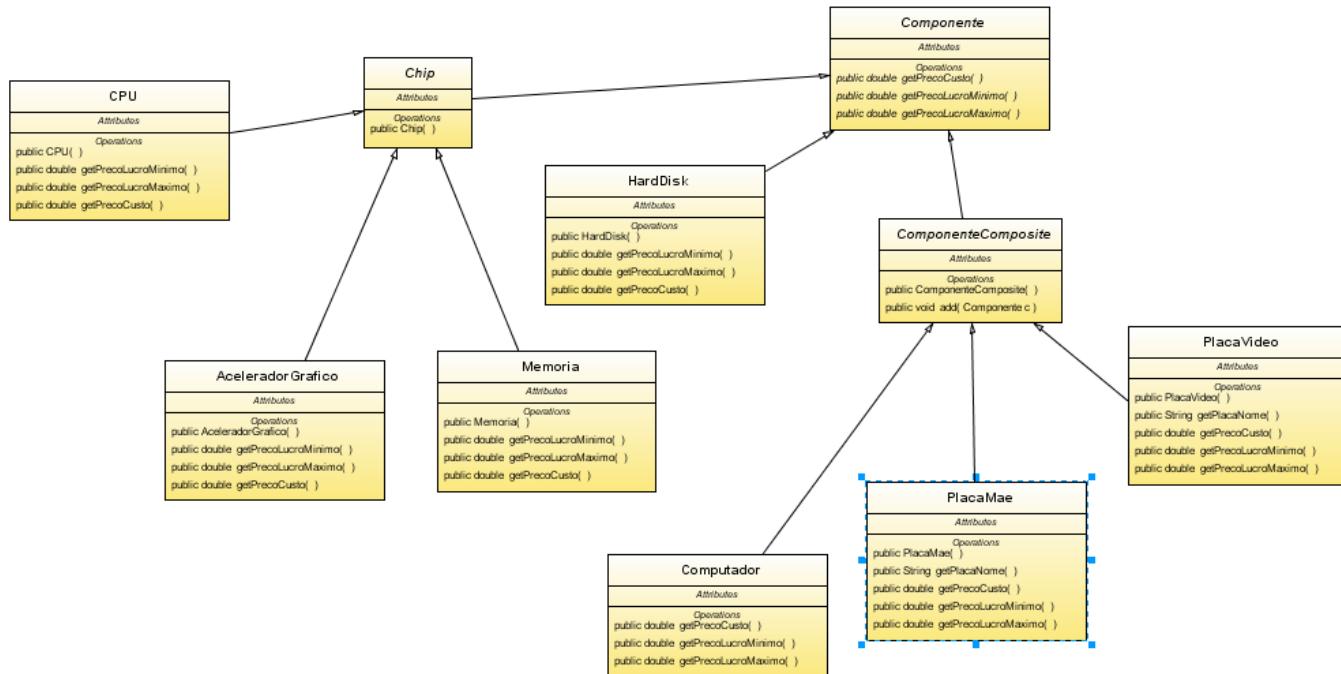
Após o refatoramento do exemplo do anti-pattern, teríamos a seguinte hierarquia de classes:



Observe que criamos uma classe Componente e uma classe ComponenteComposite, desta forma se estendermos Componente, criaremos um objeto Leaf, se estendermos a classe ComponenteComposite, criaremos um componente que será container de outros componentes, compostos ou não. Esse modelo permite maior facilidade de extensão e facilidades para aplicar comportamento dinâmico em combinação com outros design patterns.

Anotações

Com menor quantidade de detalhes podemos verificar o modelo completo, com todas as sub-classes:



Exemplo: Componente.java

```

1 package br.com.globalcode.cpcomposite;
2
3 public abstract class Componente {
4     public abstract double getPrecoCusto();
5     public abstract double getPrecoLucroMinimo();
6     public abstract double getPrecoLucroMaximo();
7
8 }
  
```

Exemplo: ComponenteComposite.java

```

1 package br.com.globalcode.cpcomposite;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5
6 public abstract class ComponenteComposite extends Componente{
7     protected Collection<Componente> componentes= new ArrayList<Componente>();
8
9     public ComponenteComposite() {
10 }
11
12     public void add(Componente c) {
13         componentes.add(c);
14     }
15 }
  
```

Anotações

A sub-classe HardDisk é um exemplo de implementação de objeto leaf:

Exemplo: HardDisk.java

```
1 package br.com.globalcode.cp.composite;
2
3 public class HardDisk extends Componente{
4     public HardDisk() {
5     }
6     public double getPrecoLucroMinimo() {
7         return 70;
8     }
9     public double getPrecoLucroMaximo() {
10        return 100;
11    }
12    public double getPrecoCusto() {
13        return 50;
14    }
15 }
```

A classe PlacaMae um exemplo de Composite:

Exemplo: PlacaMae.java

```
1 package br.com.globalcode.cp.composite;
2
3 public class PlacaMae extends ComponenteComposite{
4     public double getPrecoCusto() {
5         System.out.println("Calculando Preco de custo da composição:");
6         double preco = 100;
7         for(Componente c: componentes) {
8             preco += c.getPrecoCusto();
9         }
10        return preco;
11    }
12    public double getPrecoLucroMinimo() {
13        System.out.println("Calculando Preco com lucro mínimo da composição:");
14        double preco = 150;
15        for(Componente c: componentes) {
16            preco += c.getPrecoLucroMinimo();
17        }
18        return preco;
19    }
20    public double getPrecoLucroMaximo() {
21        System.out.println("Calculando Preco com lucro maximo da composição:");
22        double preco = 200;
23        for(Componente c: componentes) {
24            preco += c.getPrecoLucroMaximo();
25        }
26        return preco;
27    }
28 }
```

Anotações

O código da classe Computador seria refatorado para:

Exemplo: Computador.java

```
1 package br.com.globalcode.cp.composite;
2
3 public class Computador extends ComponenteComposite {
4     public double getPrecoCusto() {
5         System.out.println("Calculando Preço de custo da composição:");
6         double preco = 0;
7         for(Componente c: componentes) {
8             preco += c.getPrecoCusto();
9         }
10        return preco;
11    }
12    public double getPrecoLucroMinimo() {
13        System.out.println("Calculando Preço com lucro mínimo da composição:");
14        double preco = 0;
15        for(Componente c: componentes) {
16            preco += c.getPrecoLucroMinimo();
17        }
18        return preco;
19    }
20    public double getPrecoLucroMaximo() {
21        System.out.println("Calculando Preço com lucro máximo da composição:");
22        double preco = 0;
23        for(Componente c: componentes) {
24            preco += c.getPrecoLucroMaximo();
25        }
26        return preco;
27    }
28 }
```

Anotações

Para finalizar a classe de Exemplo de uso (Client):

Exemplo: Exemplo1.java

```
1 package br.com.globalcode.cp.composite;
2
3 public class Exemplo1 {
4
5     public static void main(String[] args) {
6         Computador c1 = new Computador();
7         PlacaMae placaMae = new PlacaMae();
8         Memoria memoria = new Memoria();
9         CPU cpu = new CPU();
10        AceleradorGrafico aceleradorVideo = new AceleradorGrafico();
11        PlacaVideo placaVideo = new PlacaVideo();
12        placaVideo.add(aceleradorVideo);
13        HardDisk hd1 = new HardDisk();
14        HardDisk hd2 = new HardDisk();
15        placaMae.add(memoria);
16        placaMae.add(cpu);
17        placaMae.add(placaVideo);
18        c1.add(placaMae);
19        c1.add(hd1);
20        c1.add(hd2);
21        System.out.println("Calculando preço de custo da composição: " );
22        System.out.println("Valor: " + c1.getPrecoCusto());
23        System.out.println("Calculando preço com lucro mínimo: " );
24        System.out.println("Valor: " + c1.getPrecoLucroMinimo());
25        System.out.println("Calculando preço com lucro máximo: " );
26        System.out.println("Valor: " + c1.getPrecoLucroMaximo());
27    }
28
29 }
```

Com esse modelo teremos bastante facilidade para criação de componentes e componentes Composite para o computador além de facilidades de programação de comportamento comum para a família de objetos.

Anotações

4.15.3 Referência ao objeto pai

Apesar de não ser uma obrigatoriedade do design pattern Composite, os elementos da composição podem armazenar uma referência para seu objeto pai com a finalidade de facilitar a navegação pelo modelo. Nossa aplicativo de exemplo refatorado para que os componentes da composição mantenham a referência para seu componente pai ficaria da seguinte forma:

Exemplo: Componente.java

```
1 package br.com.globalcode.cpcomposite2;
2
3 public abstract class Componente {
4     private Componente parent;
5     public abstract double getPrecoCusto();
6     public abstract double getPrecoLucroMinimo();
7     public abstract double getPrecoLucroMaximo();
8     public Componente getParent() {
9         return parent;
10    }
11    public void setParent(Componente c) {
12        this.parent = c;
13    }
14 }
```

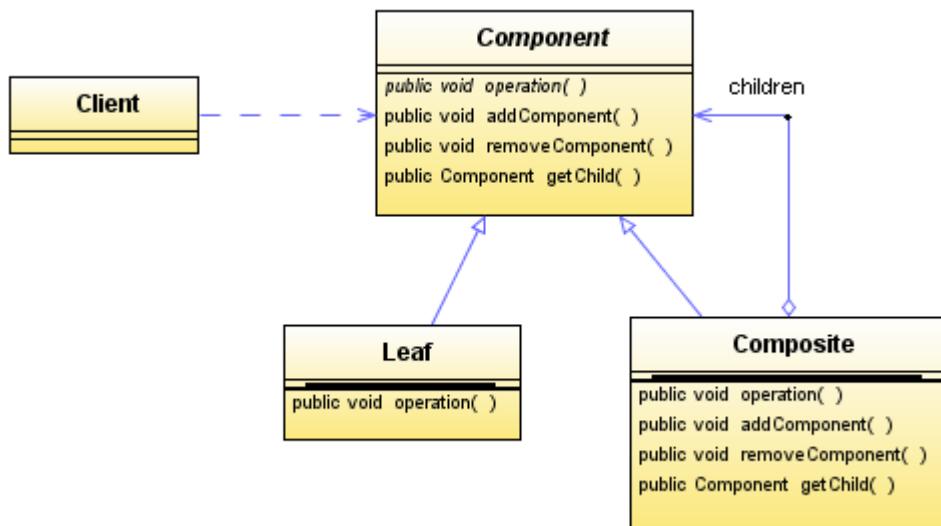
Agora que Componente disponibiliza o set e getParent, vamos alterar o ComponenteComposite para chamar o método setOwner quando receber um componente novo:

Exemplo: ComponenteComposite.java

```
1 package br.com.globalcode.cpcomposite2;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5
6 public abstract class ComponenteComposite extends Componente{
7     protected Collection<Componente> componentes= new ArrayList<Componente>();
8
9     public ComponenteComposite() {
10    }
11
12    public void add(Componente c) {
13        c.setParent(this);
14        componentes.add(c);
15    }
16 }
17
18 }
```

Anotações

4.15.4 Estrutura e Participantes



- **Component (Componente)**
Declara a interface de objetos da composição.
- **Leaf (HardDisk, Memoria, CPU)**
Representa objetos que não são containers de outros componentes.
- **Composite (ComponenteComposite)**
Representa objetos que conterão outros componentes.
- **Client (Exemplo1)**
Aplicativo que manipula os objetos compostos.

Anotações

4.16 Iterator - Comportamento

*Conhecido também como Cursor

Quando temos agregações e composições de objetos, devemos disponibilizar para o cliente uma maneira de percorrer tal estrutura sem necessariamente conhecer o objeto internamente. Iterator é um design pattern que propõe a centralização da regra de navegação em objetos em uma classe, de forma que tal regra possa variar e ele não tenha que expor excessivamente sua estrutura interna de correlacionamento com os objetos que serão percorridos.

O fato é que já estamos bastante acostumados com a idéia de Iterator, uma vez que o Java Collection Framework provê uma interface muito popular, chamada Iterator que utilizamos para percorrer diferentes implementações de coleções de objetos.

Anotações

4.16.1 Anti-pattern

Para ilustrarmos a contra-prova do design pattern Iterator, vamos revisitar a classe Computador.java, desenvolvida no capítulo anterior e vamos imaginar que gostaríamos de imprimir uma lista de todos os componentes do computador, para isso podemos imaginar o seguinte código:

Exemplo: Exemplo1.java

```
1 package br.com.globalcode.cp.iterator.antipattern;
2
3 public class Exemplo1 {
4
5     public static void main(String[] args) {
6         Computador c1 = new Computador();
7         PlacaMae placaMae = new PlacaMae();
8         Memoria memoria = new Memoria();
9         CPU cpu = new CPU();
10        AceleradorGrafico aceleradorVideo = new AceleradorGrafico();
11        PlacaVideo placaVideo = new PlacaVideo();
12        placaVideo.add(aceleradorVideo);
13        HardDisk hd1 = new HardDisk();
14        HardDisk hd2 = new HardDisk();
15        placaMae.add(memoria);
16        placaMae.add(cpu);
17        placaMae.add(placaVideo);
18        c1.add(placaMae);
19        c1.add(hd1);
20        c1.add(hd2);
21        System.out.println("Listando os componentes do computador:");
22        listaComponentes(c1);
23
24    }
25    public static void listaComponentes(ComponenteComposite c1) {
26        for(Componente c: c1.componentes) {
27            if(c instanceof ComponenteComposite) {
28                listaComponente(c);
29                listaComponentes((ComponenteComposite) c);
30            }
31            else {
32                listaComponente(c);
33            }
34        }
35    }
36    public static void listaComponente(Componente c) {
37        System.out.println("Componente " + c.getClass().getName()
38        + " Preco Custo: " + c.getPrecoCusto());
39    }
40 }
```

Esse código de fato exemplifica o pior caso, ou seja, temos a lógica para percorrer o componente dentro do programa cliente. Essa lógica poderia ser transferida para a classe Computador, por exemplo, mas de qualquer forma não estaríamos implementando o design pattern Iterator.

Cabem as seguintes perguntas:

Anotações

- Imagine se você necessitar de um sistema de paginação, onde poderíamos visualizar de N em N componentes por vez?
- E se desejasse trabalhar com diferentes tipos de computadores?
- E se você precisar de diferentes ordenações?

Todo esse código pode ser refatorado em uma classe, ou uma família, Iterator, para atender as diferentes necessidades de navegação por objetos e seus associados.

4.16.2 Aplicando o design pattern

Vamos refatorar as classes para que tenhamos um Iterator que seja capaz de percorrer para o próximo componente do computador ou para o componente anterior. Com a mesma estrutura podemos pensar em paginação, ordenação, métodos para ir para o primeiro, para o último, etc.

O código da classe Computador ficará da seguinte forma:

Exemplo: Computador.java

```
1 package br.com.globalcode.cp.iterator;
2
3 public class Computador extends ComponenteComposite {
4     ... /codigo igual ao exemplo anterior
5     public ComputadorIterator getComputadorIterator() {
6         return new ComputadorIterator(this);
7     }
8 }
```

Anotações

O método `getComputadorIterator` será responsável por criar objetos de interação associados ao computador. A classe `ComputadorIterator` será:

Exemplo: ComputadorIterator.java

```
1 package br.com.globalcode.cp.iterator;
2
3 public class ComputadorIterator {
4     private Computador computador;
5     private int current;
6     private int steps;
7     public ComputadorIterator(Computador c) {
8         this.computador=c;
9     }
10    public Componente proximoComponente() {
11        current++;
12        steps=0;
13        Componente r = getComponente(computador);
14        if(r==null) current--;
15        return r;
16    }
17    public Componente componenteAnterior() {
18        current--;
19        steps=0;
20        Componente r = getComponente(computador);
21        if(r==null) current++;
22        return r;
23    }
24    private Componente getComponente(ComponenteComposite c1) {
25        Componente retorno = null;
26        for(Componente c: c1.componentes) {
27            steps++;
28            if(c instanceof ComponenteComposite) {
29                if(steps==current) {
30                    retorno=c;
31                    break;
32                } else {
33                    Componente c11 = getComponente((ComponenteComposite) c);
34                    if(c11==null && steps==current) return null;
35                    else if(c11!=null && steps==current) return c11;
36                }
37            } else {
38                if(steps==current) {
39                    retorno = c;
40                    break;
41                }
42            }
43        }
44        return retorno;
45    }
46 }
```

Anotações

Por fim, a classe cliente ficará da seguinte forma:

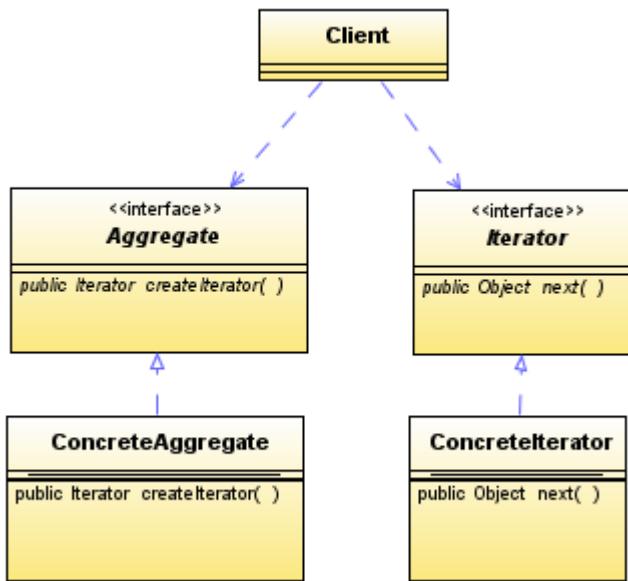
Exemplo: Exemplo1.java

```
1 package br.com.globalcode.cp.iterator;
2
3 public class Exemplo1 {
4
5     public static void main(String[] args) {
6         Computador c1 = new Computador();
7         PlacaMae placaMae = new PlacaMae();
8         Memoria memoria = new Memoria();
9         CPU cpu = new CPU();
10        AceleradorGrafico aceleradorVideo = new AceleradorGrafico();
11        PlacaVideo placaVideo = new PlacaVideo();
12        placaVideo.add(aceleradorVideo);
13        HardDisk hd1 = new HardDisk();
14        HardDisk hd2 = new HardDisk();
15        placaMae.add(memoria);
16        placaMae.add(cpu);
17        placaMae.add(placaVideo);
18        c1.add(placaMae);
19        c1.add(hd1);
20        c1.add(hd2);
21        System.out.println("Listando os componentes do computador:");
22        ComputadorIterator i = c1.getComputadorIterator();
23        Componente clista = null;
24        while((clista = i.proximoComponente())!=null) {
25            System.out.println("Componente " + clista.getClass().getName()
26                + " Preco Custo: " + clista.getPrecoCusto());
27        }
28    }
29 }
```

Veja que agora o cliente simplesmente solicita e manipula um tipo de Iterator. Podemos desenvolver diferentes Iterators para diferentes lógicas e até mesmo diferentes computadores.

Anotações

4.16.3 Estrutura e Participantes



- **Iterator**
Define a interface para percorrer o conjunto de objetos.
- **Concreteliterator (ComputadorIterator)**
Implementação concreta da interface determinada.
- **Aggregate**
Define a interface do modelo de objetos que será percorrido pelo *Iterator*.
- **ConcreteAggregate (Computador)**
Implementação da interface.

Anotações

4.17 State - Comportamento

*Conhecido também como Objects for States

O design pattern State permite que parte do comportamento de um objeto seja alterado conforme o estado do objeto. Sabemos que todo objeto possui atributos, que representam seu estado, e também métodos, que representam seu comportamento.

Podemos facilmente imaginar situações onde programamos o comportamento do objeto conforme seu estado, por exemplo, imagine que em um objeto Pedido, o método cancelar pode variar de acordo com o estado do pedido: se o pedido estiver com estado de **Faturado**, o cancelamento implicará em cancelar faturamento e nota fiscal, devolver para o estoque, cancelar solicitação de entrega, etc. Agora se o pedido estiver com estado de **Entregue**, então o cancelamento implicará em processo de devolução / retirada dos produtos, e mais os processos de cancelamento de fatura.

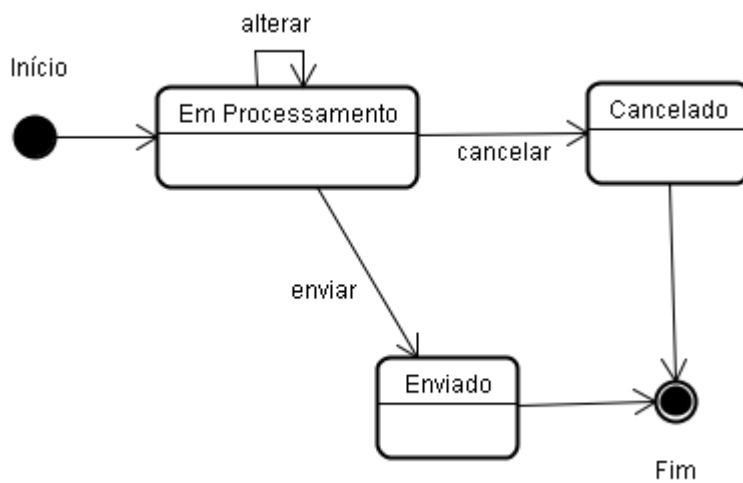
Normalmente sem o uso de State programaríamos o método cancelar com uma porção de if's para tramitar o processo conforme o estado do pedido. State propõe a criação de um modelo de classes que representam polimorficamente, conforme o estado, uma porção do comportamento de um objeto. Ao mudar o estado do objeto, mudamos parte do seu comportamento, ou todo ele se necessário.

Geralmente somente um estado **significativo** de um objeto alterara a forma que o objeto se comporta, por exemplo, alterar a razão social de um objeto empresa, não necessariamente fará com que a empresa passe a se comportar diferente, mas alterar o estado de um suposto objeto Email de “A enviar” para “Enviado” pode fazer com que seu comportamento mude bastante, portanto esta é uma mudança **significativa** de estado de objeto.

Anotações

4.17.1 Anti-pattern

Inicialmente apresentaremos um diagrama que descreve o comportamento esperado para uma entidade Pedido, dependendo do estado no qual ela se encontra.



O estado inicial de um pedido é "Em Processamento". Os eventos que podem ocorrer são solicitações para alterar, cancelar ou enviar o pedido. Dependendo do estado do pedido alguns eventos são inválidos ou provocam alteração de estado. Por exemplo, para um pedido "Em Processamento" a operação alterar é válida e as operações cancelar e enviar provovam alterações de estado para "Cancelado" e "Enviado", respectivamente.

O diagrama representa todos os eventos válidos. Qualquer evento não representado no diagrama representa uma situação de erro. Por exemplo, solicitar o cancelamento de um Pedido enviado não é uma operação válido no sistema.

Uma possível implementação para a classe Pedido é apresentada a seguir:

Exemplo: Pedido.java

```
1 package br.com.globalcode.cp.state.antipattern;
2
3 public class Pedido {
4
5     private enum Status { PROCESSANDO, CANCELADO, ENVIADO };
6     private Status status;
7 }
```

Anotações

Para uso não comercial

Anotações

Exemplo: Pedido.java (cont.)

Para uso não comercial

Anotações

```

8     public void alterar() {
9         switch(status) {
10            case PROCESSANDO:
11                //código para alteração do pedido
12                break;
13            case CANCELADO:
14                String msg1 = "Não é possível realizar alterações" +
15                                "pois o pedido foi cancelado";
16                throw new IllegalStateException(msg1);
17            case ENVIADO:
18                String msg2 = "Não é possível realizar alterações" +
19                                "pois o pedido já foi enviado";
20                throw new IllegalStateException(msg2);
21            }
22        }
23
24    public void cancelar() {
25        switch(status) {
26            case PROCESSANDO:
27                //Código para cancelamento do pedido
28                status = Status.CANCELADO;
29                break;
30            case CANCELADO:
31                String msg1 = "O pedido já foi cancelado";
32                throw new IllegalStateException(msg1);
33            case ENVIADO:
34                String msg2 = "Não é possível cancelar" +
35                                "pois o pedido já foi enviado";
36                throw new IllegalStateException(msg2);
37            }
38        }
39
40    public void enviar() {
41        switch(status) {
42            case PROCESSANDO:
43                //código para envio do pedido
44                status = Status.ENVIADO;
45                break;
46            case CANCELADO:
47                String msg1 = "Não é possível enviar" +
48                                "pois o pedido foi cancelado";
49                throw new IllegalStateException(msg1);
50            case ENVIADO:
51                String msg2 = "O pedido já foi enviado";
52                throw new IllegalStateException(msg2);
53            }
54        }
55    }
56

```

Anotações

Vejamos os passos necessários para criar um novo estado:

1. Acrescentar o valor do novo estado na enumeração Pedido.Status.
2. Atualizar cada um dos métodos introduzindo mais um elemento case na instrução switch.

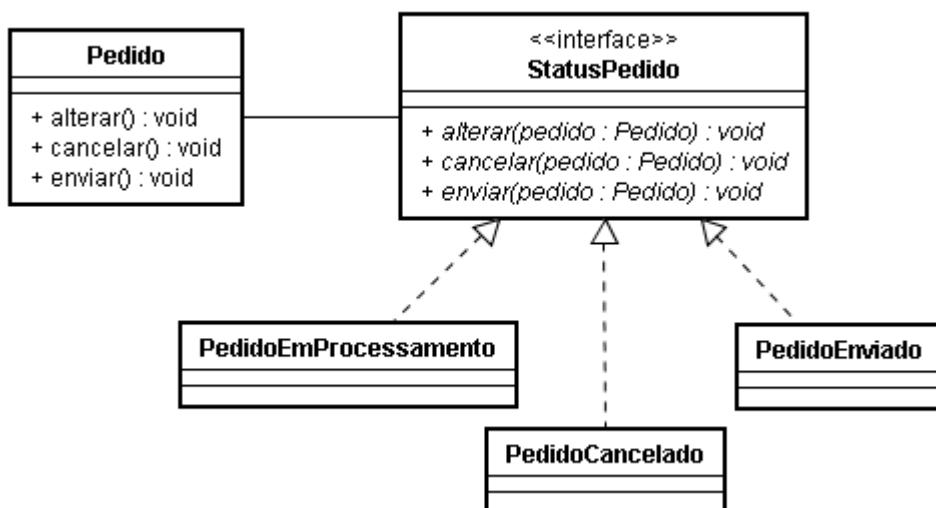
Vejamos os passos necessários para remover um estado:

1. Remover o valor na enumeração Pedido.Status.
2. Atualizar cada um dos métodos removendo um elemento case na instrução switch

Essa é uma abordagem sujeita a erros e que viola o princípio OCP. Para estender o sistema é necessário alterar as classes que já estão prontas.

4.17.2 Aplicando o design pattern

Separamos o algoritmo a ser executado em classes que representam o estado do Pedido.



Exemplo: StatusPedido.java

Anotações

```
| 1 package br.com.globalcode.cp.state;  
| 2  
| 3 public interface StatusPedido {  
| 4     public void alterar(Pedido pedido);  
| 5     public void cancelar(Pedido pedido);  
| 6     public void enviar(Pedido pedido);  
| 7 }
```

Para uso não comercial

Anotações

Exemplo: Pedido.java

```
1 package br.com.globalcode.cp.state;
2
3 public class Pedido {
4
5     private StatusPedido status;
6
7     public void alterar() {
8         status.alterar(this);
9     }
10    public void cancelar() {
11        status.cancelar(this);
12    }
13    public void enviar() {
14        status.enviar(this);
15    }
16    public void setStatus(StatusPedido status) {
17        this.status = status;
18    }
19    public StatusPedido getStatus() {
20        return status;
21    }
22 }
23 }
```

A classe Pedido, além de ficar menor do que sem o uso do pattern, ficou muito mais flexível. Perceba que para acrescentar ou remover estados não é necessária nenhuma alteração nos métodos da classe. Veja como ficam as classes de estado:

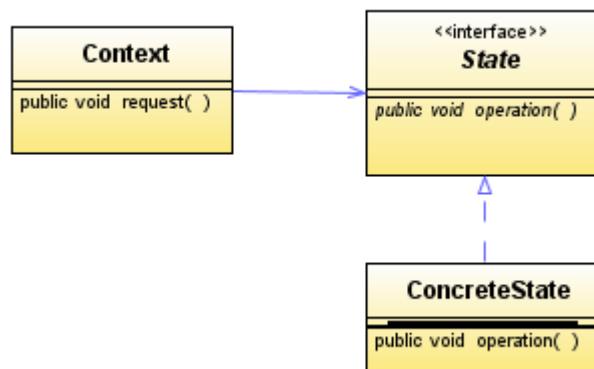
```
1 package br.com.globalcode.cp.state;
2
3 public class PedidoEmProcessamento implements StatusPedido {
4
5     public void alterar(Pedido pedido) {
6         //código para alteração do pedido
7     }
8
9     public void cancelar(Pedido pedido) {
10        //Código para cancelamento do pedido
11        pedido.setStatus(new PedidoCancelado());
12    }
13
14     public void enviar(Pedido pedido) {
15         //Código para envio do pedido
16         pedido.setStatus(new PedidoEnviado());
17     }
18 }
```

Exemplo: PedidoCancelado

```
1 package br.com.globalcode.cp.state;
2
3 public class PedidoCancelado implements StatusPedido {
4
5     public void alterar(Pedido pedido) {
6         String msg = "Não é possível realizar alterações"
7     }
8 }
9
10 package br.com.globalcode.cp.state;
11
12 public class PedidoEnviado implements StatusPedido {
13
14     public void alterar(Pedido pedido) {
15         String msg = "Não é possível realizar alterações" +
16             "pois o pedido já foi enviado";
17         throw new IllegalStateException(msg);
18     }
19
20     public void cancelar(Pedido pedido) {
21         String msg = "Não é possível cancelar" +
22             "pois o pedido já foi enviado";
23         throw new IllegalStateException(msg);
24     }
25
26     public void enviar(Pedido pedido) {
27         String msg = "O pedido já foi enviado";
28         throw new IllegalStateException(msg);
29     }
30 }
```

Anotações

4.17.3 Estrutura e Participantes



- **Context (Pedido)**
Representa o objeto que terá comportamento condicionado ao objeto de estado.
- **State (StatusPedido)**
Define a interface para encapsular os estados.
- **ConcreteState subclasses (PedidoEmProcessamento, PedidoCancelado, PedidoEnviado)**
São as diferentes implementações de comportamento por estado.

Anotações

4.18 Laboratório 5

Objetivo:

Vamos analisar e refatorar uma interface gráfica construída para navegar no modelo Composite de computadores e componentes.

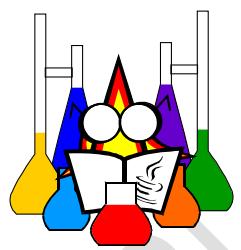


Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab05.zip e descompacte na sua pasta;	
2. Analise todo o pacote br.com.globalcode.lab5.model. Quais classes são Composite e quais são Leaf?	
3. Relacione três outros objetos relacionados com o cenário que você trabalha que poderíamos aplicar Composite.	
4. Execute a interface br.com.globalcode.lab5.MainUI e analise seu funcionamento. Como se relaciona o Iterator com a interface gráfica?	
5. Faça um refatoramento completo da interface gráfica aplicando o design pattern State.	

Anotações

4.19 Memento - Comportamento

*Conhecido também como Token

Memento é o conceito de se capturar o estado de um objeto em um determinado cenário para posteriormente poder restaurar o objeto para tal estado. O uso mais popular deste design pattern vem sendo para sistemas de desfazer, pois esta necessidade vem de encontro com Memento, porém operações de desfazer também são necessárias em transações e orquestração de processos, tornando a aplicação deste design pattern bastante atraente para cenários corporativos em geral.

A aplicação do Memento deve extrair uma cópia do estado do objeto porém sem se expor excessivamente, para evitar a quebra do encapsulamento. Para isso objetos deverão fornecer um método específico para se obter um Memento (recordação) dele mesmo.

4.19.1 Anti-pattern

No caso do Memento o anti-pattern é a simples omissão do recurso de undo / redo em um determinado sistema. Podemos também considerar anti-pattern de Memento qualquer sistema de undo / redo caseiro que não seja implementado com as definições descritas originalmente no pattern, mas dificilmente fugiria muito do Memento, ou seja, fatalmente em alguma parte do modelo, uma classe representaria um memento, pois essa é a informação vital para sistemas de undo / redo.

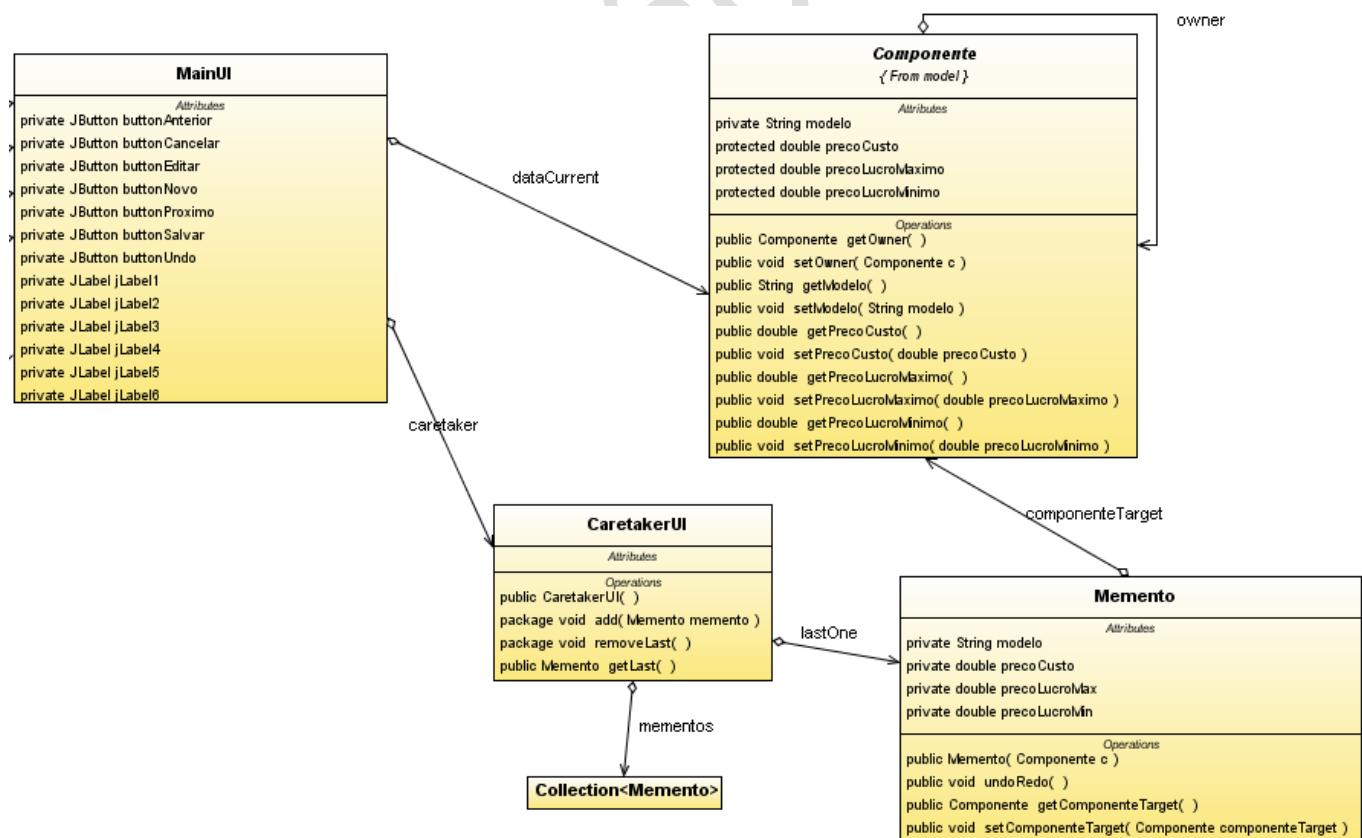
Anotações

4.19.2 Aplicando o design pattern

Ao aplicarmos o design pattern Memento, devemos refatorar todas as classes que terão seus estados armazenados para posteriormente, possivelmente retroceder para este estado armazenado. Devemos planejar bem quem vai participar da funcionalidade de Undo / Redo pois a estratégia de implementação do Memento poderá variar. Por exemplo, imagine que você poderá permitir que o usuário desfaça uma operação de edição, porém a de exclusão de uma informação poderá ser do tipo “undoable”. Caso efetivamente tenha necessidade de desfazer todas as operações, devemos pensar como nosso modelo vai suportar desfazer inclusões e exclusões de objetos além de mudanças de relacionamentos e composições (imagine que um objeto que antes pertencia à placa mãe, passe a fazer parte do computador diretamente..).

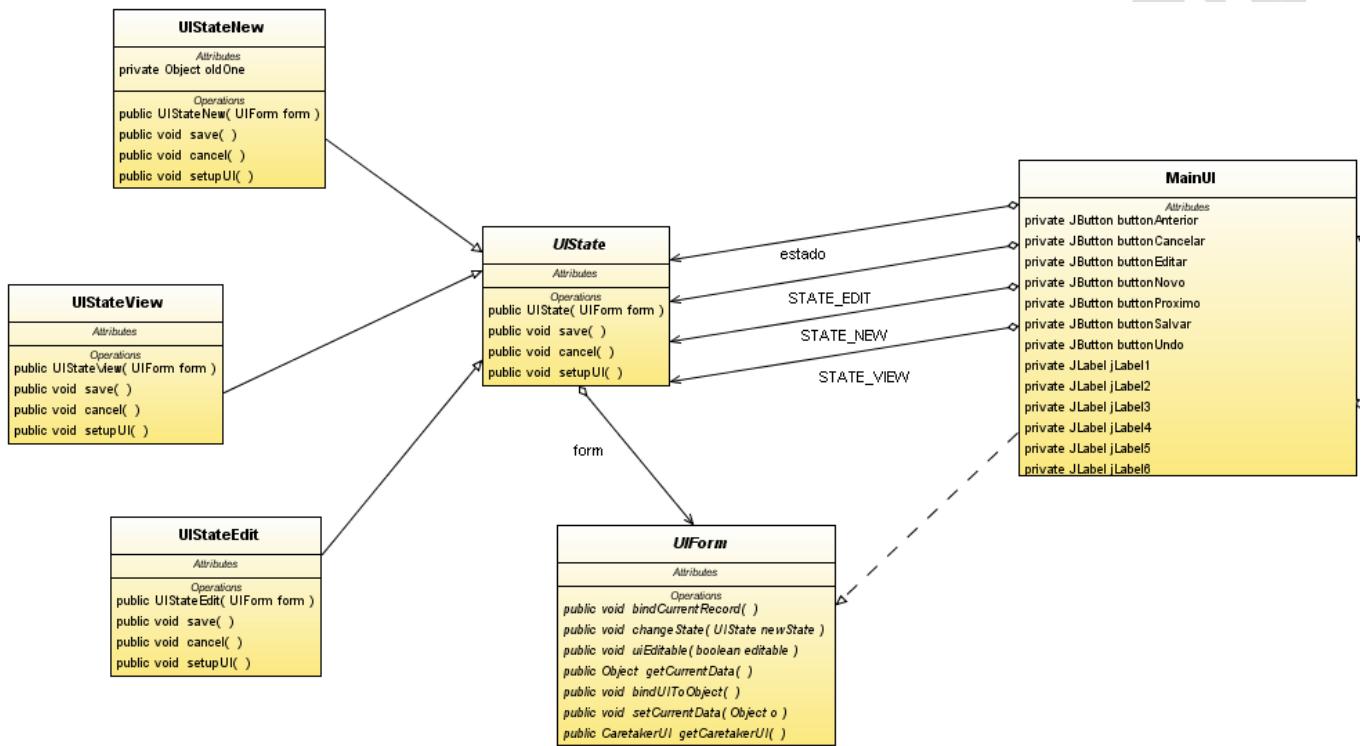
Geralmente as entidades participantes de undo / redo, disponibilizarão um método `saveToMemento` / `getMemento` e `restoreFromMemento`. Tais métodos cuidam de fornecer o Memento para o aplicativo e recuperar seu estado a partir do Memento. No caso da implementação que vamos apresentar, para evitar o refatoramento da família de classes Componente, criamos uma classe Memento que vai cuidar deste processo. Vamos analisar nosso sistema de gerenciamento de componentes de computador e sua interface gráfica agora com suporte a operação de undo.

O seguinte diagrama representa o modelo da interface com usuário, Caretaker, Memento e o Componente, que é o originador do memento:



Anotações

O relacionamento entre a interface gráfica MainUI e seus estados, continua o mesmo exceto novos comportamentos previstos no UIForm:



Anotações

A codificação das classes ficou da seguinte forma:

Exemplo: Memento.java

```

1 package br.com.globalcode.cp.memento;
2
3 import br.com.globalcode.cp.memento.model.Componente;
4
5 public class Memento {
6     private Componente componenteTarget;
7     private String modelo;
8     private double precoCusto;
9     private double precoLucroMax;
10    private double precoLucroMin;
11    public Memento(Componente c) {
12        setComponenteTarget(c);
13        modelo = c.getModelo();
14        precoCusto = c.getPrecoCusto();
15        precoLucroMax = c.getPrecoLucroMaximo();
16        precoLucroMin = c.getPrecoLucroMinimo();
17    }
18    public void undoRedo() {
19        String modeloNovo="";
20        double precoLucroMaxNovo=0, precoCustoNovo=0, precoLucroMinNovo=0;
21        modeloNovo = modelo;
22        precoLucroMaxNovo = precoLucroMax;
23        precoLucroMinNovo = precoLucroMin;
24        precoCustoNovo = precoCusto;
25
26        modelo = componenteTarget.getModelo();
27        precoCusto = componenteTarget.getPrecoCusto();
28        precoLucroMin = componenteTarget.getPrecoLucroMinimo();
29        precoLucroMax = componenteTarget.getPrecoLucroMaximo();
30
31        componenteTarget.setModelo(modeloNovo);
32        componenteTarget.setPrecoCusto(precoCustoNovo);
33        componenteTarget.setPrecoLucroMaximo(precoLucroMaxNovo);
34        componenteTarget.setPrecoLucroMinimo(precoLucroMinNovo);
35    }
36    public Componente getComponenteTarget() {
37        return componenteTarget;
38    }
39    public void setComponenteTarget(Componente componenteTarget) {
40        this.componenteTarget = componenteTarget;
41    }
42 }
```

Nossa implementação de Memento em particular, implementa tanto a operação de desfazer como de refazer para um determinado componente, através do método undoRedo().

Anotações

A implementação de Memento propõe a criação de um participante chamado Caretaker que cuidará da lista de mementos do aplicativo. A nossa interface gráfica vai manter uma instância deste objeto:

Exemplo: CaretakerUI.java

```
1 package br.com.globalcode.cp.memento;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5
6 public class CaretakerUI {
7     private Collection<Memento> mementos;
8     private Memento lastOne;
9
10    public CaretakerUI() {
11        mementos=new ArrayList<Memento>();
12    }
13
14    void add(Memento memento) {
15        mementos.add(memento);
16        lastOne = memento;
17    }
18
19    void removeLast() {
20        if(lastOne!=null) {
21            mementos.remove(lastOne);
22            lastOne=null;
23        }
24    }
25    public Memento getLast() {
26        return lastOne;
27    }
28 }
```

O Caretaker será mais ou menos complexo conforme sua necessidade de navegação pelos memento. Na programação apresentada, só poderemos trabalhar com o último memento. O Caretaker também cuidará do número de níveis de undo que o sistema vai disponibilizar.

Anotações

Adicionalmente alteramos a classe que representa o estado de edição para que seu comportamento agora inclua uma comunicação com Caretaker, adicionando o memento antes da operação de edição dos dados de um componente:

Exemplo: UIStateEdit.java

```
1 package br.com.globalcode.cp.memento;
2
3 import br.com.globalcode.cp.memento.model.Componente;
4
5 public class UIStateEdit extends UIState{
6     public UIStateEdit(UIForm form) {
7         super(form);
8     }
9     public void save() {
10         form.getCaretakerUI().add(
11             new Memento((Componente) form.getCurrentData()));
12         System.out.println("Salvando dados no objeto");
13         form.bindUIToObject();
14     }
15     public void cancel() {
16         form.bindCurrentRecord();
17         form.getCaretakerUI().removeLast();
18     }
19     public void setupUI() {
20         form.uiEditable(true);
21     }
22 }
```

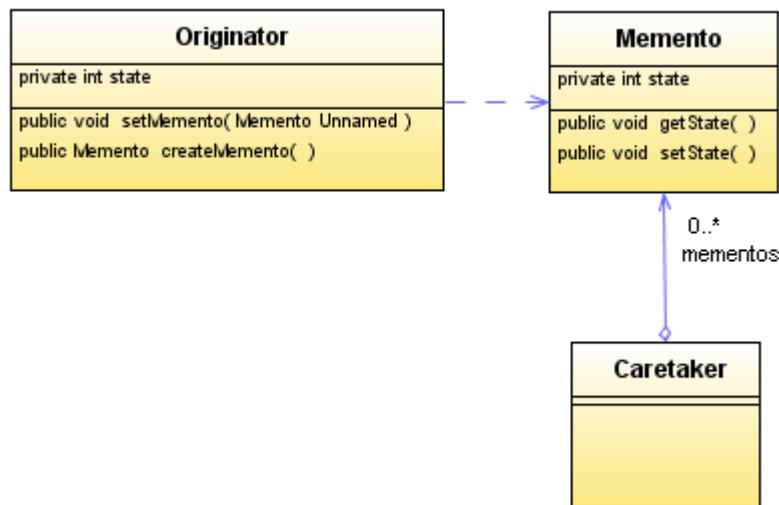
A interface gráfica vai prover um botão que terá o seguinte comportamento:

Exemplo: trecho de MainUI.java

```
307     private void buttonUndoActionPerformed(java.awt.event.ActionEvent evt) {
308
309         this.caretaker.getLast().undoRedo();
310         this.dataCurrent = caretaker.getLast().getComponenteTarget();
311     }
```

Anotações

4.19.3 Estrutura e Participantes



- **Memento (Memento)**
Representa o estado do objeto que vamos armazenar.
- **Originator (Componente)**
Objeto que vai originar os dados para o memento.
- **Caretaker (CaretakerUI)**
Classe responsável pela custódia do memento.

Anotações

4.20 Laboratório 6

Objetivo:

Praticar o uso do design pattern Memento.

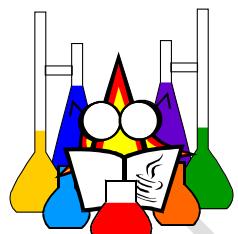


Tabela de atividades:

Atividade	OK
1. Implemente sistema de undo na interface gráfica do laboratório anterior seguindo o padrão Memento.	
2. Para efeito de cadênciā de tempo de laboratório conforme skills de programação da turma, o instrutor poderá optar pela analise do código já pronto, neste caso faça o download do arquivo lab06.zip	

Anotações

4.21 Decorator - Estrutura

*Também conhecido como Wrapper

“Um Decorator oferece uma abordagem do tipo ‘use quando for necessário’ para adição de responsabilidades.”

Decorator é um design pattern muito útil quando queremos adicionar comportamento / responsabilidades sem a necessidade de criar sub-classes de extensão. Os comportamentos comuns a uma família de sub-classes podem ser programados em classes distintas para serem usadas sob demanda, permitindo a adição (e remoção) de tais comportamentos sem alterar uma classe.

A plataforma Java tem uma famosa implementação de Decorator na sua API de Java IO, nas famílias de classes InputStream / OutputStream, Reader / Writer. Graças à aplicação de Decorator, podemos com IO acrescentar a medida que precisamos de buffer, criptografia, compactação, conversão de dados em nossos streamings: teclado, arquivo, rede, etc. No modelo Java IO um tipo de Stream é programado em uma classe, o buffer em outra, compactação em outra, de forma que podemos pegar qualquer Stream e aplicar qualquer comportamento adicional (Decorator) sem necessariamente alterar a classe de Stream.

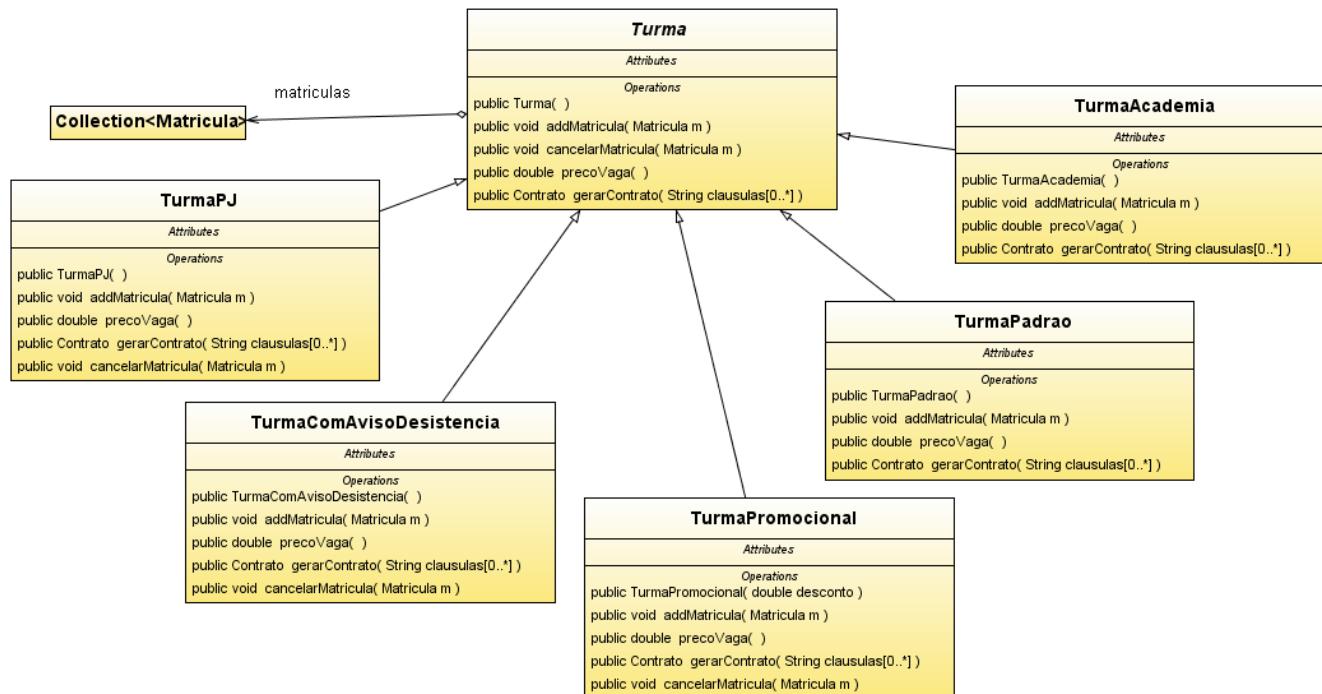
Decorator representa um comportamento comum que poderá ser compartilhado dinamicamente por um conjunto de classes.

“O Decorator segue a interface do componente que decora, de modo que sua presença é transparente para os clientes do componente.”

Anotações

4.21.1 Anti-pattern

Vamos ilustrar um cenário que apresenta oportunidade de refatoramento para Decorator utilizando o modelo de objeto de um sistema de gerenciamento de turmas em escolas. Neste modelo teremos uma classe Turma no topo da hierarquia de diferentes implementações de tipos de turmas em sub-classes:



O modelo apresentado é bastante tradicional e faz uso de sub-classes para definir turmas de diferentes tipos: academia, pessoa jurídica, com aviso de desistência para fila e turma promocional. O grande problema é, e se precisarmos de uma turma que reúna mais do que uma destas características, por exemplo, uma turma promocional e também com aviso de desistência?

Neste caso, sem o uso de Decorator teríamos que escolher herdar ou TurmaPromocional ou TurmaComAvisoDesistencia e modificar seu comportamento por duplicata de código ou por delegação, e este é justamente o problema, imagina que se tivermos um modelo com 10 possíveis tipos de turma, poderemos ter um conjunto enorme de sub-classes para poder representar todas as combinações necessárias.

Este modelo portanto representa o anti-pattern de Decorator.

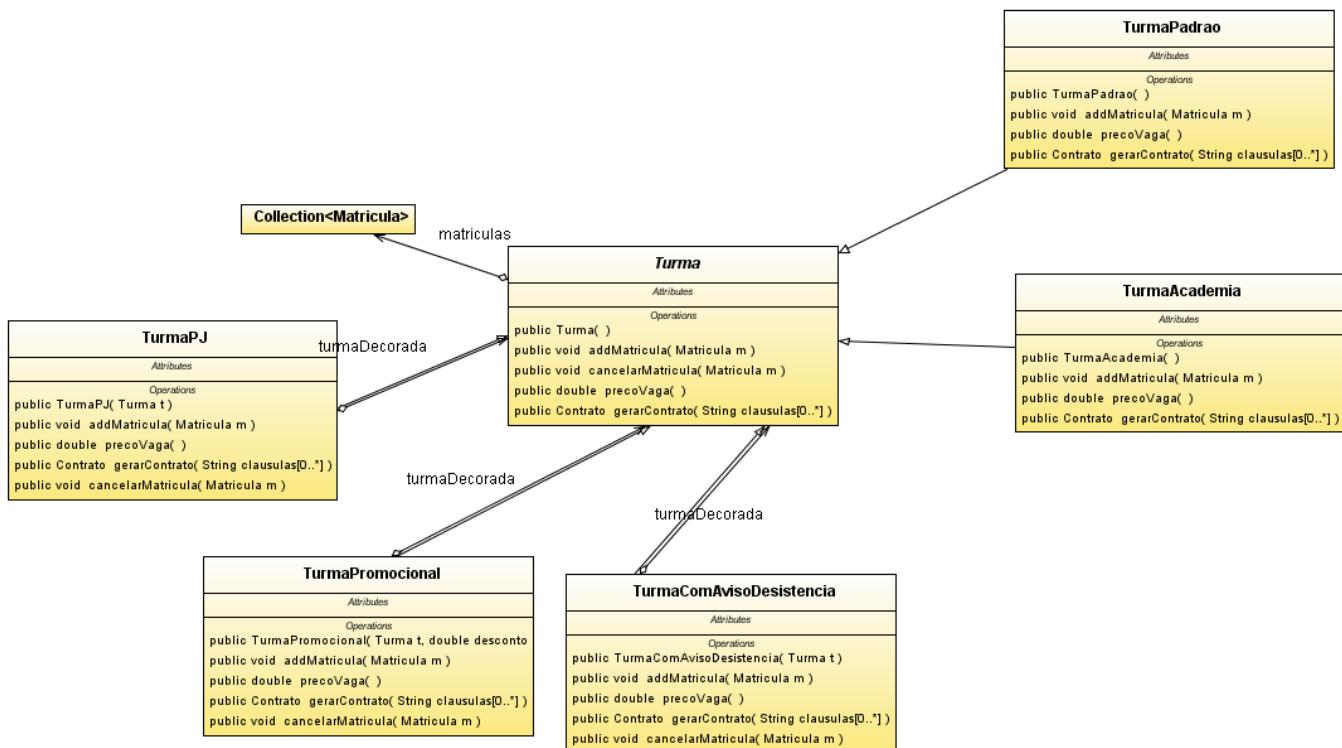
Anotações

4.21.2Aplicando o design pattern

Vamos remodelar as classes da seguinte forma:

- **TurmaPadrao** e **TurmaAcademia** serão sub-classes convencionais de Turma;
- **TurmaPJ**, **TurmaComAvisoDesistencia** e **TurmaPromocional** serão sub-classes de Turma mas vão agir como Decorators;
- Desta forma uma **TurmaPadrao** ou **TurmaAcademia** poderão utilizar as funcionalidades de **TurmaPJ**, **TurmaComAvisoDesistencia** e **TurmaPromocional** conforme necessitarem de um ou mais comportamentos. O interessante do Decorator é justamente a possibilidade de adicionarmos múltiplos decoradores / comportamentos.

O diagrama UML de classes fica da seguinte forma:



TurmaPJ, **TurmaComAvisoDesistencia** e **TurmaPromocional** são sub-classes de turma e adicionalmente mantém uma referência a um objeto decorado. Os construtores das classes foram programados para receber um objeto Turma.

Anotações

Vejamos o código de implementação começando pela classe Turma que define o padrão de comportamento dos objetos:

Exemplo: Turma.java

```
1 package br.com.globalcode.cp.decorator;
2
3 import java.util.Collection;
4
5 public abstract class Turma {
6     private Collection<Matricula> matriculas;
7
8     public abstract void addMatricula(Matricula m);
9     public abstract void cancelarMatricula(Matricula m);
10
11    public double precoVaga() {
12        return 10.0;
13    }
14    public Contrato gerarContrato(String... clausulas) {
15        return null;
16    }
17 }
```

Agora vejamos a classe TurmaPadrao que é uma sub-classe de Turma convencional:

Exemplo: TurmaPadrao.java

```
1 package br.com.globalcode.cp.decorator;
2
3 public class TurmaPadrao extends Turma {
4
5     public void addMatricula(Matricula m) {
6         //lógica para adição de matrícula
7     }
8
9     public double precoVaga() {
10        return 10.0;
11    }
12
13    public Contrato gerarContrato(String... clausulas) {
14        Contrato contrato = null;
15        //lógica para geração de contrato
16        return contrato;
17    }
18
19    public void cancelarMatricula(Matricula m) {
20        //lógica para cancelamento de matrícula
21    }
22 }
```

Anotações

E agora vejamos o código dos Decorators:

Exemplo: TurmaPJ.java

```
1 package br.com.globalcode.cp.decorator;
2
3 public class TurmaPJ extends Turma {
4     Turma turmaDecorada;
5     public TurmaPJ(Turma t) {
6         turmaDecorada=t;
7     }
8     public void addMatricula(Matricula m) {
9         //Regras diferentes para numero de alunos p/ turma
10        turmaDecorada.addMatricula(m);
11    }
12
13    public double precoVaga() {
14        return 0;
15    }
16    public Contrato gerarContrato(String... clausulas) {
17        //Clausulas do Contrato PJ
18        return turmaDecorada.gerarContrato();
19    }
20    public void cancelarMatricula(Matricula m) {
21        turmaDecorada.cancelarMatricula(m);
22    }
23 }
```

Repare no código destacado que o decorator foi programado com um construtor que obriga o usuário a fornecer um objeto a ser decorado. Posteriormente este objeto é utilizado nos métodos do Decorator pois estamos trabalhando com adição de comportamento, no entanto, podemos encontrar situação onde queremos sobrepor por completo o método decorado. As demais classes seguirão a mesma estrutura. Vejamos um código de exemplo uso do modelo:

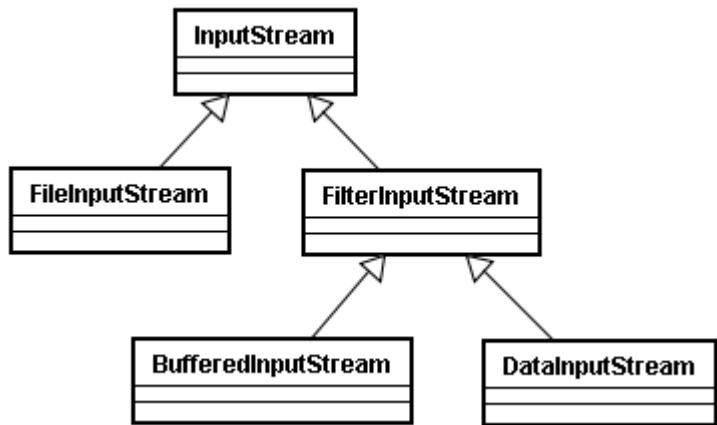
Exemplo: Exemplo1.java

```
1 package br.com.globalcode.cp.decorator;
2
3 public class Exemplo1 {
4
5     public static void main(String[] args) {
6         Turma turmal = new TurmaPadrao();
7         TurmaPromocional turmaPromo =
8             new TurmaPromocional(turmal, 10);
9         TurmaComAvisoDesistencia turmaAviso =
10            new TurmaComAvisoDesistencia(turmaPromo);
11         turmaAviso.addMatricula(new Matricula());
12         //ou...
13         Turma t1 = new TurmaComAvisoDesistencia(
14             new TurmaPromocional(new TurmaPadrao(), 10.0));
15         t1.addMatricula(new Matricula());
16         t1.gerarContrato();
17     }
18 }
```

Anotações

4.21.30 padrão Decorator nas APIs do Java

A API de I/O utiliza o padrão Decorator na criação dos fluxos de leitura e escrita. Por isso é possível combinar diversas funcionalidades em um único fluxo de dados sem a necessidade de subclasses para todas as combinações possíveis.



trecho da classe `java.io.FilterInputStream`

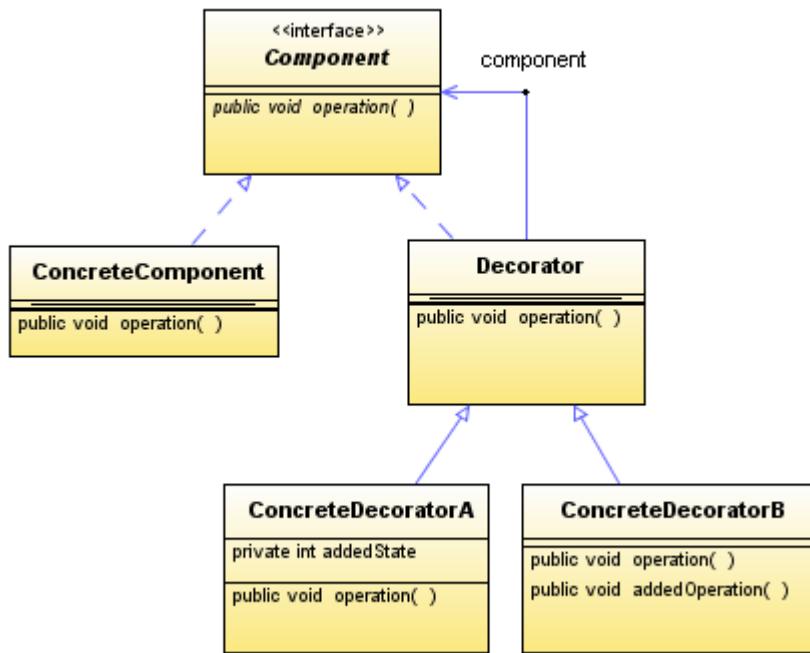
```

1 package java.io;
2
3 public class FilterInputStream extends InputStream {
4
5     protected volatile InputStream in;
6
7     protected FilterInputStream(InputStream in) {
8         this.in = in;
9     }
10
11    public int read() throws IOException {
12        return in.read();
13    }
14
15
16 package java.io;
17
18 public class DataInputStream
19     extends FilterInputStream implements DataInput {
20
21     public DataInputStream(InputStream in) {
22         super(in);
23     }
24
25     public final short readShort() throws IOException {
26         int ch1 = in.read();
27         int ch2 = in.read();
28         if ((ch1 | ch2) < 0)
29             throw new EOFException();
30         return (short)((ch1 << 8) + (ch2 << 0));
31     }
32
33 }
  
```

Para uso não comercial

Anotações

4.21.4 Estrutura e Participantes



- **Component (Turma)**
Define a interface para os componentes onde teremos decorators.
- **ConcreteComponent (TurmaPadrao, TurmaAcademia)**
Representam a implementação concreta do componente que poderá ser decorado posteriormente.
- **Decorator**
Representa a particularidade do componente decorator manter a referência para o componente decorado.
Pode ser omitida, como no exemplo apresentado.
- **ConcreteDecorator (TurmaPJ, TurmaComAvisoDesistencia, TurmaPromocional)**
Representam a implementação dos decorators.

Anotações

4.22 Chain of responsibility - Comportamento

Promove o desacoplamento entre um objeto solicitante e o objeto solicitado, dando a oportunidade da requisição trafegar por uma cadeia de comportamentos que poderão agregar tratamentos. Este design pattern é mais um que permite a adição de comportamento dinâmico porém se difere bastante de Visitor e Decorator pelos seguintes motivos:

1. Decorator atua em uma família de classes e oferece possibilidade de se agregar novos comportamentos por um modelo de agregação, delegação e sub-classes.
2. Decorator segue a interface do componente que decora.
3. Visitor é princípio para modelos estáveis em termos de sub-classes, pois a adição de um novo tipo de classe implicaria em refatorar todos os visitantes já programados para que a considere.
4. Visitor representa um comportamento para um coletivo de implementações de um modelo de sub-classes.
5. Chain of responsibility permite que sejam criados novos pré-processadores de chamadas aos métodos e pode funcionar com diferentes modelos de classes, pois é totalmente baseado na interceptação da chamada de um solicitante para um solicitado, o que pode até dificultar o interfaceamento profundo com tais objetos.
6. Chain of responsibility trabalha com sistema tipo interceptação de requisições permitindo que uma cadeia de objetos atuem na requisição. Um uso bastante conhecido deste pattern esta na implementação de Servlet Filter onde podemos criar diferentes filtros interceptadores de requisições http para agregar tratamentos diversos tais como logging, segurança, filtro de dados, etc.

A medida que analisarmos exemplos concretos em Java as diferenças ficarão cada vez mais claras e você saberá identificar qual pattern é melhor para cada ocasião.

Anotações

4.22.1 Anti-pattern

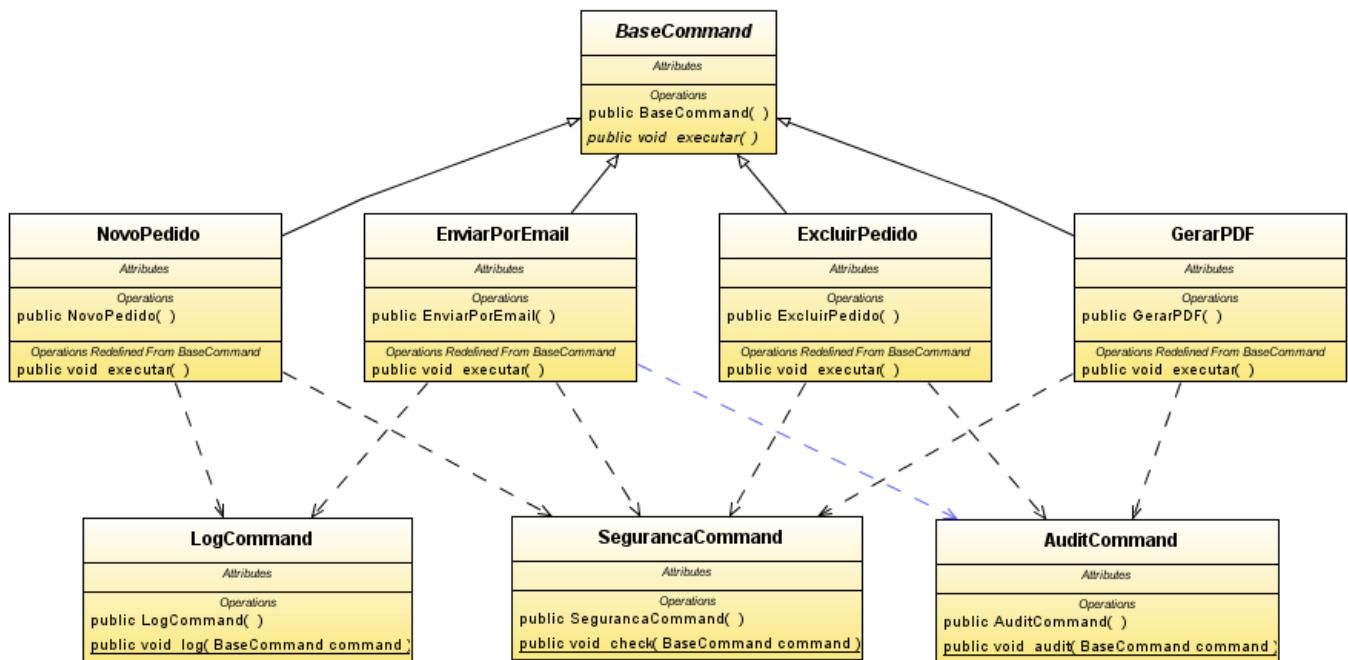
Vamos imaginar a necessidade de efetuarmos algumas operações antes e eventualmente depois da execução dos Commands que desenvolvemos anteriormente com o sistema de fábricas. Existem determinados serviços que são necessários neste sistema para podermos utilizá-lo profissionalmente:

- Segurança
- Log e auditoria
- Validação de dados
- Binding e unbinding de user-interface em JavaBean
- Filtro de dados
- Integração

Para agregarmos tais serviços aos nossos commands sem Chain of Responsibility, podemos programar manualmente cada command com chamadas a essas classes de serviços: log, validação, segurança, etc. Cada command chamaria os serviços necessários e aqueles comuns, seriam chamados por todos eles. Tal modelagem resultaria em um alto acoplamento entre os commands e também dificultaria a operação de simplesmente desativar um dos serviços.

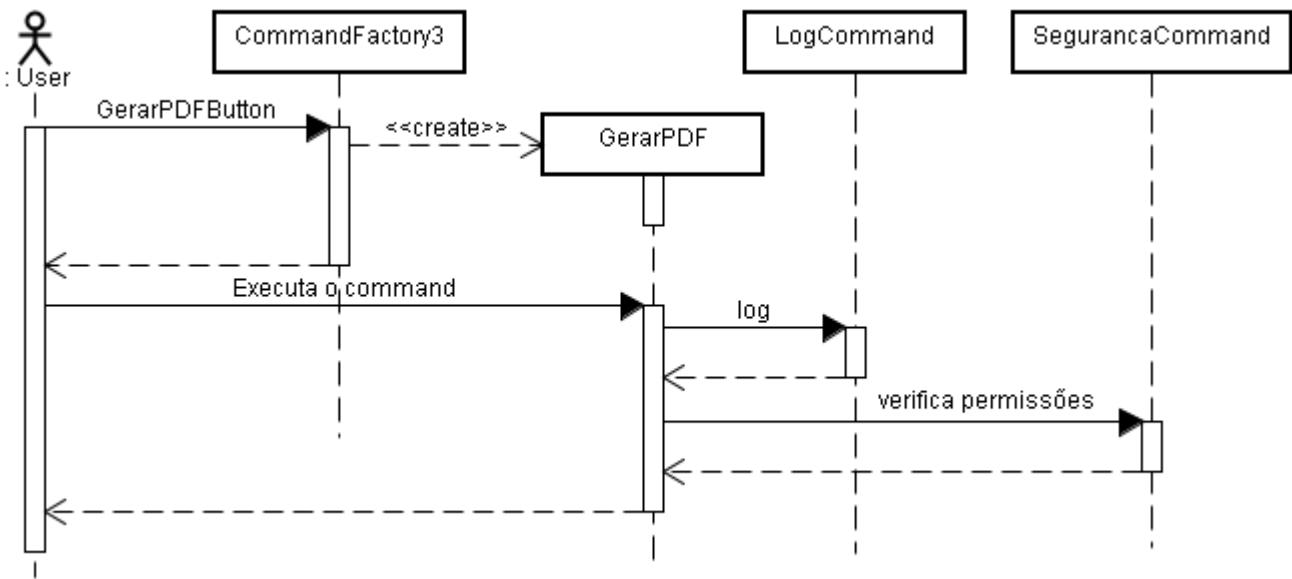
Observe neste diagrama de classes que a contratação dos serviços de auditoria, segurança e logging estão gerando um grande emaranhado de dependências:

Anotações



O seguinte diagrama de seqüência representa a chamada de um command aos serviços:

Anotações



O seguinte código seria desenvolvido representando o anti-pattern do Chain of Responsibility:

Exemplo: GerarPDF.java

```

1 package br.com.globalcode.cp.chain.antipattern;
2
3 public class GerarPDF extends BaseCommand {
4
5     public void executar() {
6         LogCommand.log(this);
7         SegurancaCommand.check(this);
8         //Executa codigo
9     }
10 }
```

Anotações

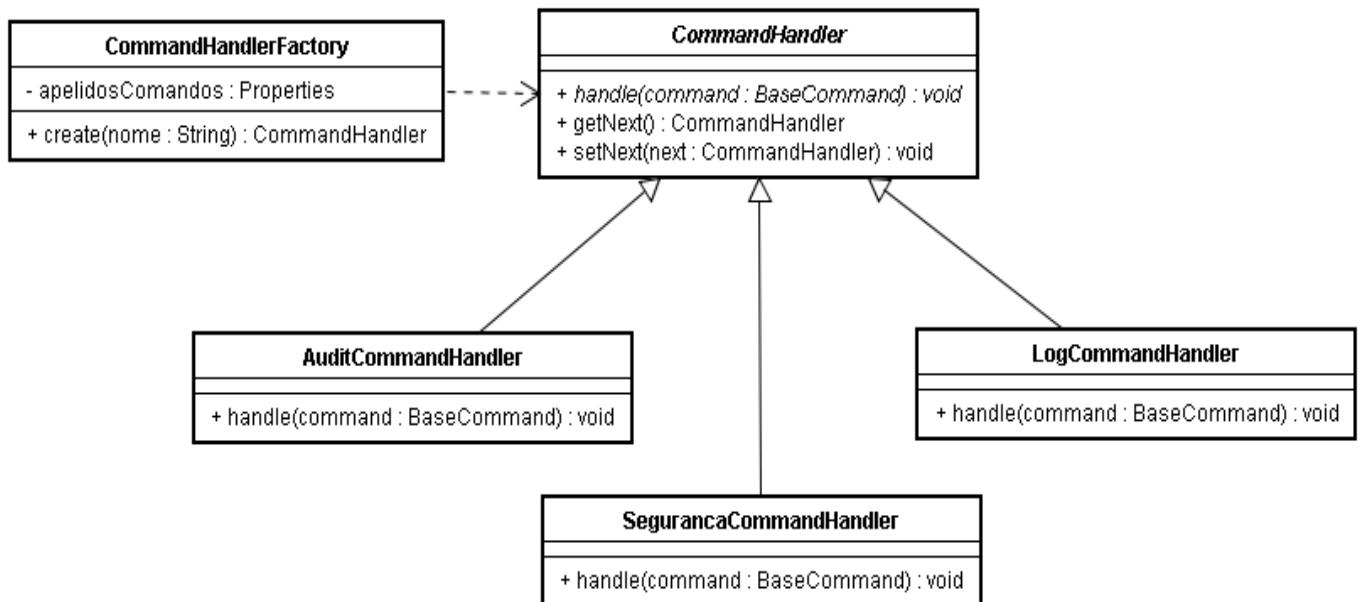
4.22.2Aplicando o design pattern

Vamos refatorar nossa fábrica de commands fazendo com que os serviços comuns aos commands (log, segurança, etc.) possam ser configurados de forma fácil e desacoplada. Vamos apresentar as etapas deste refatoramento começando pela principal idéia, criar uma annotation de configuração de serviços para ser utilizada nos commands. Nossos commands contratarão os serviços da seguinte forma:

Exemplo: GerarPDF.java

```
1 package br.com.globalcode.cp.chain;
2
3 @CommandInterceptor(posProcess=true, preProcess=true,
4 interceptor="log, audit, seguranca")
5 public class GerarPDF extends BaseCommand{
6
7     public void executar() {
8         //lógica de geração de PDF
9     }
10 }
```

No código em destaque observamos a anotação `CommandInterceptor` que permitirá ao framework executar os serviços indicados no parâmetro `interceptor`, no caso “log, audit, segurança”. Teremos nosso framework um interpretador desta annotation que vai indicar qual(is) serviço devem ser executado antes e depois do command. Todos os serviços foram transformados em interceptadores que serão configurado conforme a anotação e serão encadeados com Chain of responsibility pelo framework:



Anotações

O código da annotation encontra-se a seguir:

Exemplo: CommandInterceptor.java

```
1 package br.com.globalcode.cp.chain;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 @Retention(RetentionPolicy.RUNTIME)
9 @Target(ElementType.TYPE)
10 public @interface CommandInterceptor {
11     public boolean preProcess();
12     public boolean posProcess();
13     public String interceptor();
14 }
```

Veja o exemplo da super-classe CommandHandler e um das sub-classes de serviço:

Exemplo: CommandHandler.java

```
1 package br.com.globalcode.cp.chain;
2
3 public abstract class CommandHandler {
4     protected CommandHandler next;
5
6     public CommandHandler() {
7     }
8     public abstract void handle(BaseCommand command);
9     public CommandHandler getNext() {
10         return next;
11     }
12     public void setNext(CommandHandler next) {
13         this.next = next;
14     }
15 }
```

Exemplo: LogCommandHandler.java

```
1 package br.com.globalcode.cp.chain;
2
3 public class LogCommandHandler extends CommandHandler{
4
5     public void handle(BaseCommand command) {
6         System.out.println("Log Command Handle!");
7         getNext().handle(command);
8     }
9 }
```

Repare que temos sempre um objeto Handler encadeado e cabe a nossa codificação chamar o próximo objeto ou interromper o ciclo da cadeia através da omissão do código de chamada “getNext.handle(command)”.

Anotações

Para implementar o interpretador de anotação de interceptação afetando pouco nosso código, criamos um decorador de Command que vai agregar a funcionalidade de analisar a anotação de interceptação. Para facilitar a vida dos aplicativos clientes de nossas fábricas, também criamos uma interface utilizando Generics, denominada GenericFactory. Veja a fábrica de objetos com trabalhando com o decorator:

Exemplo: CommandFactory3.java

```
1 package br.com.globalcode.cp.chain;
2
3 import java.io.*;
4 import java.util.Properties;
5
6 public class CommandFactory3 implements GenericFactory<BaseCommand, String>{
7     Properties apelidosComandos = new Properties();
8
9     public BaseCommand create(String name) {
10         BaseCommand command = null;
11         String stringClasse = apelidosComandos.getProperty(name);
12         try {
13             Class classe = Class.forName(stringClasse);
14             Object object = classe.newInstance();
15             command = new CommandAnnotationDecorator((BaseCommand) object);
16         } catch (Exception trateme) {
17             //tratar
18             trateme.printStackTrace();
19             System.out.println(" Erro: " + trateme.getMessage());
20         }
21         return command;
22     }
23     public CommandFactory3() {
24         try {
25             apelidosComandos.load(getClass().
26                 getResourceAsStream("commands.properties"));
27         } catch (Exception ex) {
28             //tratar
29         }
30     }
31 }
```

Veja o código da interface genérica:

Exemplo: GenericFactory.java

```
1 package br.com.globalcode.cp.chain;
2
3 public interface GenericFactory <Retorno, Entrada>{
4     public Retorno create(Entrada name);
5 }
```

Anotações

Agora vejamos o código do Decorator:

Exemplo: CommandAnnotationDecorator.java

```

1 package br.com.globalcode.cp.chain;
2
3 public class CommandAnnotationDecorator extends BaseCommand {
4
5     protected BaseCommand commandDecorado;
6     protected CommandHandler next;
7     protected boolean pre;
8     protected boolean pos;
9
10    public CommandAnnotationDecorator(BaseCommand c) {
11        this.commandDecorado = c;
12        setupCommandHandlerChain();
13    }
14
15    public void setupCommandHandlerChain() {
16        CommandHandlerFactory handlerFactory = new CommandHandlerFactory();
17        if (commandDecorado.getClass().
18            isAnnotationPresent(CommandInterceptor.class)) {
19            CommandInterceptor aci = commandDecorado.getClass().
20                getAnnotation(CommandInterceptor.class);
21            pre = aci.preProcess();
22            pos = aci.posProcess();
23            String handlers[] = aci.interceptor().split(",");
24            CommandHandler cmd = null;
25            CommandHandler cmdPrevious = null;
26            for (String handler : handlers) {
27                if (cmdPrevious == null) {
28                    cmdPrevious = handlerFactory.create(handler.trim());
29                    next=cmdPrevious;
30                } else {
31                    cmd = handlerFactory.create(handler.trim());
32                    cmdPrevious.setNext(cmd);
33                    cmdPrevious = cmd;
34                }
35            }
36        }
37    }
38
39    public void executar() {
40        //analise de annotations pre
41        //a execucao representa um command no topo da cadeia
42        System.out.println
43            ("Analise de annotation e chamada de chain of responsibility");
44        processPreAnnotation();
45        System.out.println("Vai executar " + commandDecorado.getClass().getName());
46        commandDecorado.executar();
47        processPosAnnotation();
48    }
49

```

Anotações

Exemplo: CommandAnnotationDecorator.java (cont.)

```
50     public void processPreAnnotation() {
51         if (!pre) {
52             return;
53         }
54         next.handle(commandDecorado);
55     }
56
57     public void processPosAnnotation() {
58         if (!pos) {
59             return;
60         }
61         next.handle(commandDecorado);
62     }
63 }
```

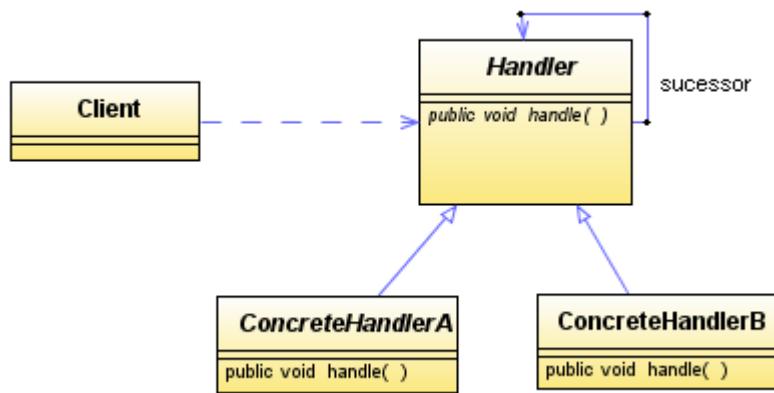
Para podermos configurar os interceptadores por apelidos, criamos um Factory chamada CommandHandlerFactory:

Exemplo: CommandHandlerFactory.java

```
1 package br.com.globalcode.cp.chain;
2
3 import java.io.*;
4 import java.util.Properties;
5
6 public class CommandHandlerFactory
7                 implements GenericFactory<CommandHandler, String>{
8     Properties apelidosComandos = new Properties();
9
10    public CommandHandler create(String name) {
11        CommandHandler command = null;
12        String stringClasse = apelidosComandos.getProperty(name);
13        try {
14            Class classe = Class.forName(stringClasse);
15            Object object = classe.newInstance();
16            command = (CommandHandler) object;
17
18        } catch (Exception trateme) {
19            System.out.println(" Erro: " + trateme.getMessage());
20            trateme.printStackTrace();
21        }
22        return command;
23    }
24    public CommandHandlerFactory() {
25        try {
26            apelidosComandos.load(getClass()
27                                .getResourceAsStream("handlers.properties"));
28        } catch (Exception ex) {
29            System.out.
30                println(" Erro ao obter arquivo de config handlers.properties");
31        }
32    }
33 }
```

Anotações

4.22.3 Estrutura e Participantes



- **Handler (CommandHandler)**
Define a interface de tratamento de solicitações.
- **ConcreteHandler (LogCommandHandler)**
Implementações que tratarão as requisições.
- **Client**
Quem inicia a chamada na cadeia de objetos.

Anotações

4.23 Laboratório 7

Objetivo:

Neste laboratório você vai analisar o framework de commands com a funcionalidade de configuração de interceptadores por anotações.



Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab07.zip a partir da URL indicada pelo instrutor(a).	
2. Descompacte e crie um projeto com NetBeans ou Eclipse;	
3. Analise o código completo executando a classe com main AplicativoComAbstractFactory em modo de debug	
4. Altere configurações dos interceptadores nos Commands e analise o resultado	
5. Crie um novo interceptador chamado ValidarDadosCommand e utilize em algum command.	

Anotações

4.24 Strategy - Comportamento

*Conhecido também como Policy

É um design pattern bastante voltado para encapsulamento de algoritmos que podem variar com facilidade para prover um comportamento mais adequado para um objeto frente a uma situação.

Podemos imaginar uma situação onde um servidor de aplicativos precisa recuperar o estado de um objeto. Este processo de recuperação de estado de objeto poderá variar, pois existem diferentes locais onde estados de objetos são armazenados:

- Memória heap
- Hard-disk
- Database
- Arquivo serializado
- Rede

Portanto o algoritmo de recuperação poderia ser implementado com o padrão Strategy de forma que teríamos uma subclasse de Strategy para cada técnica de recuperação.

Anotações

4.24.1 Anti-pattern

Imagine um cenário de um sistema para lojas que permite diversas regras para descontos nos seus pedidos de compras. Veja uma possível implementação da classe Pedido.

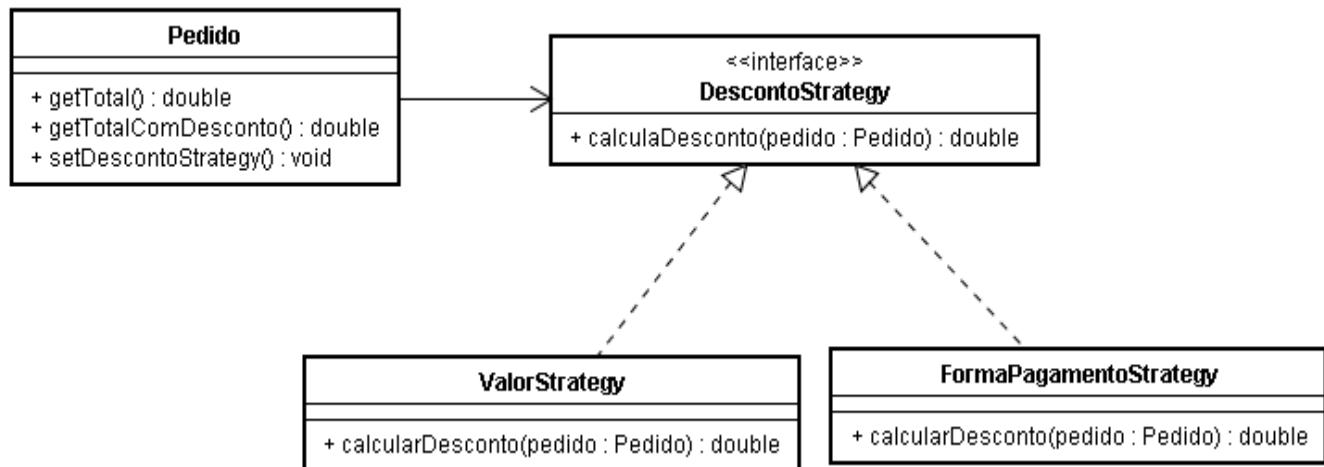
Exemplo: Pedido.java

```
1 package br.com.globalcode.cp.strategy.antipattern;
2
3 import java.util.Collection;
4
5 public class Pedido {
6
7     FormaPagamento pagamento;
8     Cliente cliente;
9     Collection<Item> itens;
10
11    public double getTotal() {
12        double total = 0;
13        //regra que calcula preco total dos itens
14        return total;
15    }
16
17    public double getTotalComDesconto() {
18        if (pagamento.getDescricao().equals("a vista")) {
19            return getTotal() - (0.2 * getTotal());
20        } else if (getTotal() > 500) {
21            return getTotal() - (0.15 * getTotal());
22        } else if (getTotal() > 300) {
23            return getTotal() - (0.1 * getTotal());
24        }
25        return getTotal();
26    }
27 }
```

Anotações

4.24.2 Aplicando o design-pattern

Para melhorar o código anterior, isolamos o elemento da classe Pedido que varia (o algoritmo de cálculo de descontos) em sua própria hierarquia de classes conforme o diagrama apresentado a seguir:



O código fonte de cada classe pode ser visto a seguir:

Exemplo: DescontoStrategy.java

```

1 package br.com.globalcode.cp.strategy;
2
3 public interface DescontoStrategy {
4     public double calculaDesconto(Pedido pedido);
5 }
6
7 package br.com.globalcode.cp.strategy;
8
9 public class ValorStrategy implements DescontoStrategy {
10     public double calculaDesconto(Pedido pedido) {
11         double desconto = 0;
12         if (pedido.getTotal() > 500)
13             desconto += pedido.getTotal() * 0.15;
14         else if (pedido.getTotal() > 300)
15             desconto += pedido.getTotal() * 0.1;
16         return desconto;
17     }
18 }
  
```

Anotações

Exemplo: FormaPagamentoStrategy.java

```
1 package br.com.globalcode.cp.strategy;
2
3 public class FormaPagamentoStrategy implements DescontoStrategy {
4
5     --public double calculaDesconto(Pedido pedido) {
6
1 package br.com.globalcode.cp.strategy;
2
3 import java.util.Collection;
4
5 public class Pedido {
6     FormaPagamento pagamento;
7     Cliente cliente;
8     Collection<Item> itens;
9
10    DescontoStrategy strategy;
11
12    public double getTotal() {
13        double total = 0;
14        //regra que calcula preco total dos itens
15        return total;
16    }
17
18    public double getTotalComDesconto() {
19        return this.getTotal() - strategy.calculaDesconto(this);
20    }
21
22    public void setStrategy(DescontoStrategy st) {
23        strategy = st;
24    }
25
26    public FormaPagamento getPagamento() {
27        return pagamento;
28    }
29    //demais getters e setters omitidos
30 }
```

Anotações

É um exemplo de aplicativo que interage com as classes apresentadas.

Exemplo: ExemploComStrategy.java

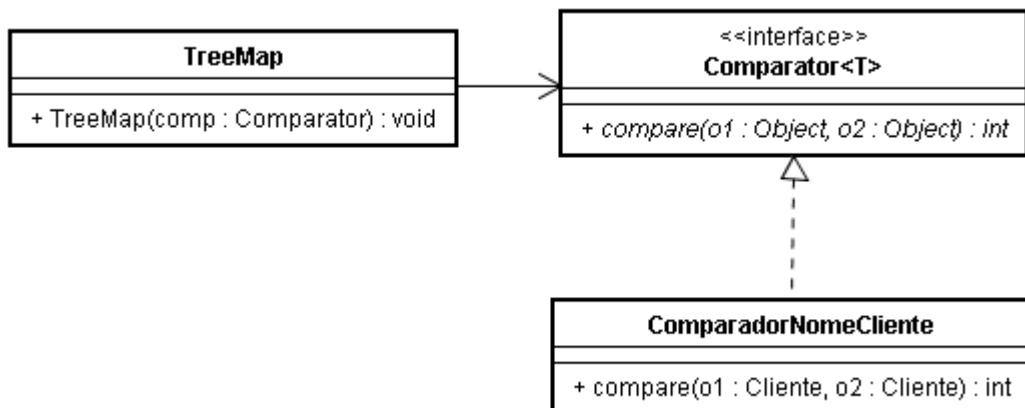
```
1 package br.com.globalcode.cp.strategy;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class ExemploComStrategy {
7
8     public static void main(String[] args) {
9         //inicialização do pedido
10        List<Item> itens = new ArrayList<Item>();
11        Produto p1 = new Produto("Camiseta Casual Class", 40);
12        Produto p2 = new Produto("Caneca Java", 20);
13        itens.add(new Item(p1, 10));
14        itens.add(new Item(p2, 15));
15        Cliente cliente = new Cliente("Carlos da Silva");
16        FormaPagamento fp = new FormaPagamento("a prazo");
17
18        Pedido pedido = new Pedido();
19        pedido.setItems(itens);
20        pedido.setCliente(cliente);
21        pedido.setPagamento(fp);
22
23        //configuração da estratégia para cálculo de desconto
24        DescontoStrategy calculadoraDesconto = new ValorStrategy();
25        pedido.setStrategy(calculadoraDesconto);
26        double totalComDesconto = pedido.getTotalComDesconto();
27        System.out.println("Valor do pedido com desconto:" +
28                           totalComDesconto);
29    }
30 }
```

Anotações

4.24.30 padrão Strategy nas APIs do Java

4.24.3.1 Coleções ordenadas

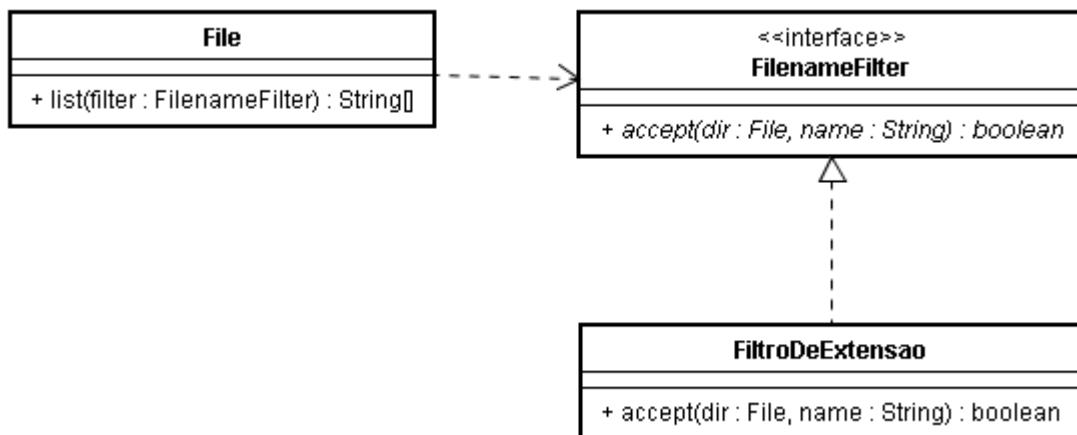
Quando criamos uma implementação de `java.util.Comparator` e passamos no construtor de uma coleção ordenada, como por exemplo `TreeSet` ou `TreeMap`, é utilizado o pattern Strategy. Veja um exemplo de utilização de `Comparator` para um JavaBean Cliente ordenado por nome.



```
1 package java.util;
2
3 public class TreeMap<K, V>
4     extends AbstractMap<K, V>
5     implements NavigableMap<K, V>, Cloneable, java.io.Serializable {
6
7     private final Comparator<? super K> comparator;
8     private transient Entry<K, V> root = null;
9
10    public TreeMap(Comparator<? super K> comparator) {
11        this.comparator = comparator;
12    }
13
14    public V put(K key, V value) {
15        Entry<K, V> t = root;
16        if (t == null) {
17            //inicializacao de root
18        }
19        int cmp;
20        Entry<K, V> parent;
21
22        Comparator<? super K> cpr = comparator;
23        if (cpr != null) {
24            do {
25                parent = t;
26                cmp = cpr.compare(key, t.key);
27                if (cmp < 0) {
28                    t = t.left;
29                } else if (cmp > 0) {
30                    t = t.right;
31                } else {
32                    return t.setValue(value);
33                }
34            } while (t != null);
35        }
36        //codigo omitido
37        return null;
38    }
39}
```

4.24.3.2 Filtros de arquivos

Quando criamos filtros de arquivos e passamos para o método list ou.listFiles da classe java.io.File é utilizado o pattern Strategy. Veja um exemplo com o uso de FilenameFilter.



trecho da classe java.io.File

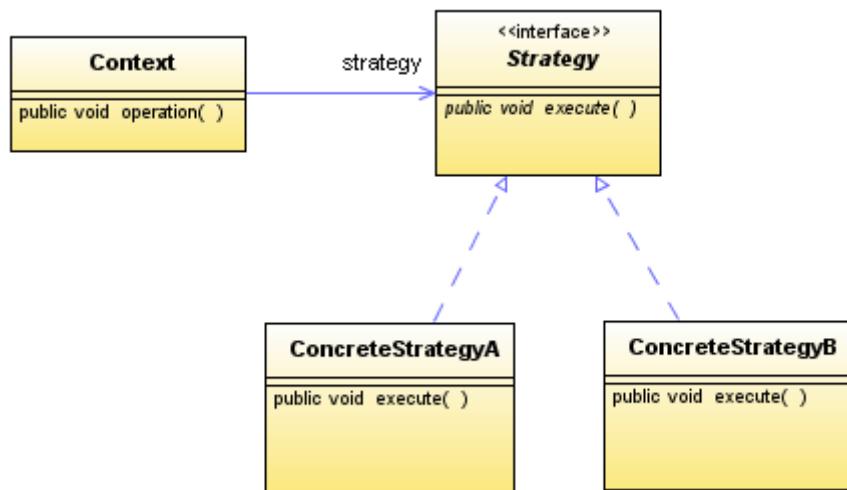
```

5 public class File implements Serializable, Comparable<File> {
6
7     public String[] list(FilenameFilter filter) {
8         String_names[] = list();
9
10        for (int i = 0; i < names.length; i++) {
11            if (filter.accept(names[i])) {
12                v.add(names[i]);
13            }
14        }
15        return (String[]) (v.toArray(new String[v.size()]));
16    }
  
```

```

1 package br.com.globalcode.cp.strategy.java;
2
3 import java.io.File;
4 import java.io.FilenameFilter;
5
6 public class FiltroDeExtensao implements FilenameFilter{
7
8     String extensao;
9
10    public FiltroDeExtensao(String extensao) {
11        this.extensao = extensao;
12    }
13    public boolean accept(File dir, String name) {
14        return name.endsWith(extensao);
15    }
16 }
  
```

4.24.4 Estrutura e Participantes



- **Strategy** (`DescontoStrategy`)

Define uma interface comum para todos os algoritmos suportados.

- **ConcreteStrategy** (`ValorStrategy`, `FormaPagamentoStrategy`)

Classes que implementam os algoritmos.

- **Context** (`Pedido`)

Elemento configurado com uma `ConcreteStrategy` e que utiliza o algoritmo para alguma lógica de negócio.

Anotações

4.25 Proxy - Estrutura

*Conhecido também como Surrogate

Proxy é a idéia de fornecer um objeto substituto ao objeto verdadeiro que será chamado. É popularmente usado em diversas tecnologias para computação distribuída como Java RMI / Enterprise JavaBeans. Quando acessamos um EJB ou objeto RMI não gerenciado por um container Java EE, estamos na verdade acessando um objeto proxy, que tem a mesma interface que o original, porém este objeto vai gerenciar a chamada ao objeto de verdade, atravessando a rede se necessário, cuidando de parâmetros de entrada e saída, exceções, entre outras responsabilidades.

Proxy pode ser facilmente confundido com Decorator, pois ambos trazem a idéia de mantermos a interface do objeto que estamos atuando, porém a estrutura de Proxy não prevê encadeamento de objetos para adição dinâmica de comportamento como o Decorator prevê, Proxy gerencia o acesso a um objeto, se disfarçando dele.

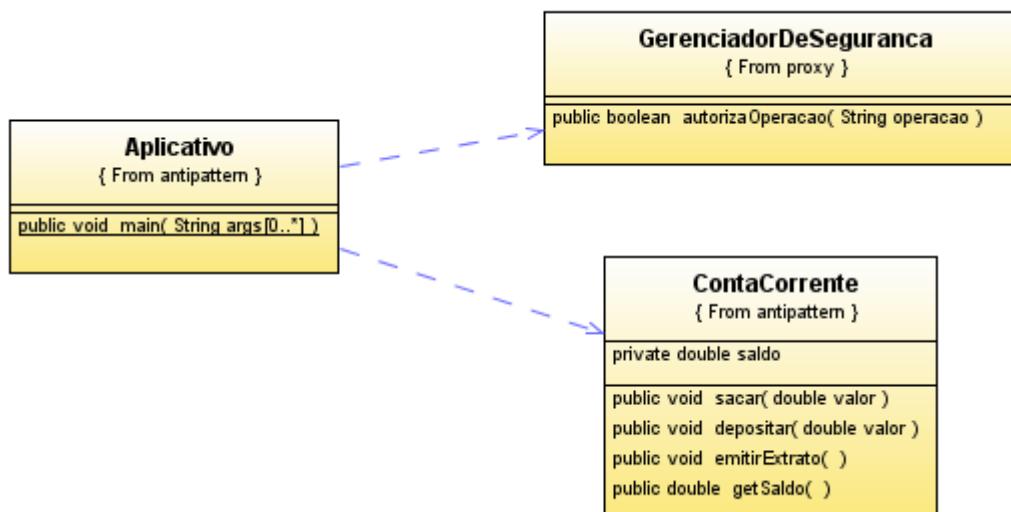
Do GoF:

“Como o Decorator, o padrão Proxy compõe o objeto e fornece uma interface idêntica para os clientes. Diferentemente do Decorator, o padrão Proxy não está preocupado em incluir ou excluir propriedades dinamicamente e não está projetado para composição recursiva.”

Anotações

4.25.1 Anti-pattern

Podemos classificar como anti-pattern de Proxy qualquer modelo em que o aplicativo seja obrigado a gerenciar explicitamente uma característica da manipulação de um objeto que deveria ser transparente. Vejamos no exemplo a seguir esta situação para um cenário aonde é necessário verificar a autorização de um aplicativo no acesso aos métodos de um objeto:



Neste caso queremos acessar os métodos da classe ContaCorrente, porém devemos sempre verificar antes se temos autorização para a operação. A seguir temos o exemplo do código para as classes apresentadas:

Exemplo: Aplicativo.java

```
1 package br.com.globalcode.cp.proxy.antipattern;
2
3 import br.com.globalcode.cp.proxy.GerenciadorDeSeguranca;
4
5 public class Aplicativo {
6
7     public static void main(String[] args) {
8         GerenciadorDeSeguranca gerenciador = new GerenciadorDeSeguranca();
9         ContaCorrente conta = new ContaCorrente();
10        if(gerenciador.autorizaOperacao("deposito")) {
11            conta.depositar(2500);
12        }
13        if(gerenciador.autorizaOperacao("emitirExtrato")) {
14            conta.emitirExtrato();
15        }
16    }
17 }
```

Anotações

Exemplo: GerenciadorDeSeguranca.java

```
1 package br.com.globalcode.cp.proxy;
2
3 public class GerenciadorDeSeguranca {
4
5     public boolean autorizaOperacao(String operacao) {
6         boolean autorizado = false;
7         //Verifica as credenciais de segurança do usuário
8         return autorizado;
9     }
10}
```

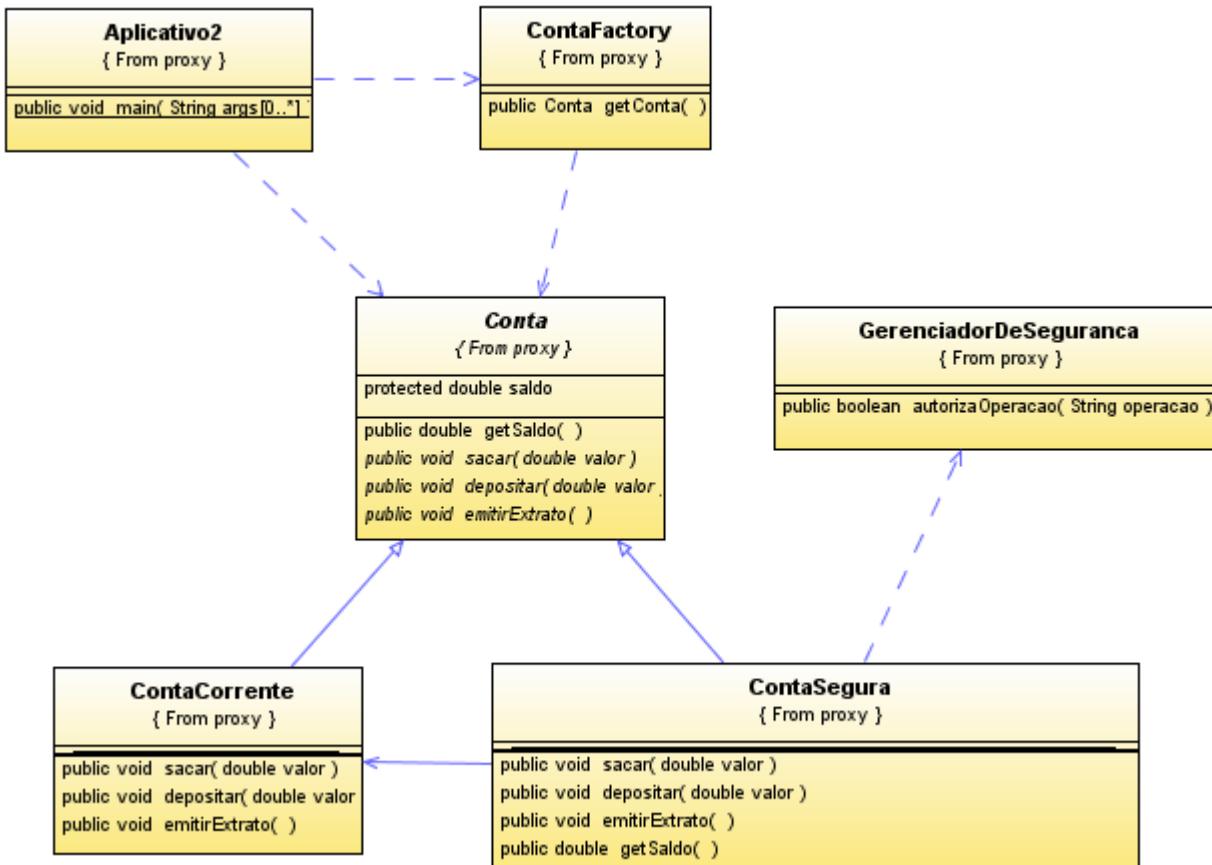
Exemplo: ContaCorrente.java

```
1 package br.com.globalcode.cp.proxy.antipattern;
2
3 public class ContaCorrente {
4
5     private double saldo;
6
7     public void sacar(double valor) {
8         //lógica de saque
9     }
10
11    public void depositar(double valor) {
12        //lógica de depósito
13    }
14
15    public void emitirExtrato() {
16        System.out.println("Saldo atual = " + saldo);
17    }
18
19    public double getSaldo() {
20        return saldo;
21    }
22}
```

Anotações

4.25.2Aplicando o design-pattern

Ao aplicarmos o Proxy, criamos uma classe abstrata Conta, que servirá como superclasse tanto para a classe ContaCorrente como para o proxy criado: ContaSegura. A classe ContaSegura manterá uma referência para um objeto ContaCorrente que será intermediado por ele. Ao mesmo tempo a interação com a classe que gerencia a autorização (GerenciadorDeSegurança) sai do aplicativo e vai para o proxy. Vejamos como fica o novo diagrama de classes:



Apesar de não ser obrigatório, criamos também uma classe ContaFactory, a ser utilizada pelo aplicativo para obter a referência para Conta. Uma implementação alternativa poderia fazer com que o aplicativo instanciasse diretamente a classe ContaSegura.

Vejamos agora como ficam as implementações das classes:
Anotações

Exemplo: Conta.java

```

1 package br.com.globalcode.cp.proxy;
2
3 public abstract class Conta {
4
5     protected double saldo;
6
7     public double getSaldo() {
8         return saldo;
9     }
10
11    public abstract void sacar(double valor);
12    public abstract void depositar(double valor);
13    public abstract void emitirExtrato();
14 }
```

Exemplo: ContaCorrente.java

```

1 package br.com.globalcode.cp.proxy;
2
3 public class ContaCorrente extends Conta{
4
5     public void sacar(double valor) {
6         //lógica de saque
7     }
8
9     public void depositar(double valor) {
10        //lógica de depósito
11    }
12
13    public void emitirExtrato() {
14        System.out.println("Saldo atual = " + saldo);
15    }
16
17 package br.com.globalcode.cp.proxy;
18
19 public class ContaSegura extends Conta{
20
21     ContaCorrente conta = new ContaCorrente();
22     GerenciadorDeSeguranca gerente = new GerenciadorDeSeguranca();
23
24     public void sacar(double valor) {
25         if(gerente.autorizaOperacao("sacar")) {
26             conta.sacar(valor);
27         }
28     }
29
30     public void depositar(double valor) {
31         if(gerente.autorizaOperacao("depositar")) {
32             conta.depositar(valor);
33         }
34     }
35
36     public void emitirExtrato() {
37         if(gerente.autorizaOperacao("emitirExtrato")) {
38             conta.emitirExtrato();
39         }
40     }
41
42     public double getSaldo() {
43         if(gerente.autorizaOperacao("getSaldo")) {
44             return conta.getSaldo();
45         }
46         throw new SecurityException
47             ("Você não tem autorização de consultar o saldo desta conta!!!");
48     }
49 }
```

Exemplo: Aplicativo2.java

```
1 package br.com.globalcode.cp.proxy;
2
3 public class Aplicativo2 {
4
5     public static void main(String[] args) {
6         ContaFactory factory = new ContaFactory();
7         Conta conta = factory.getConta();
8         conta.depositar(2500);
9         conta.emitirExtrato();
10    }
11 }
```

Anotações

4.25.3Proxy e Decorator

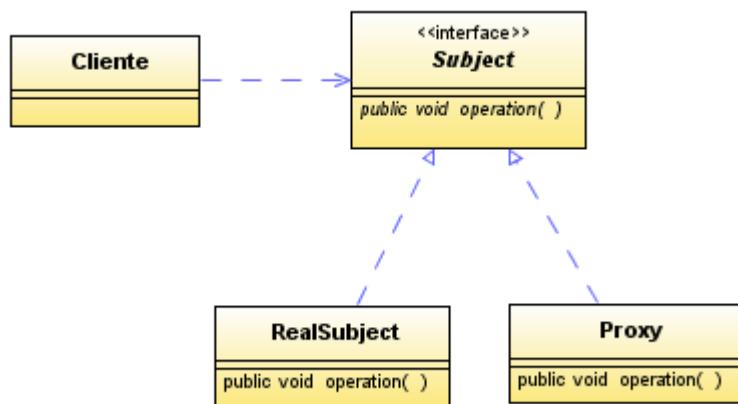
A grande diferença entre a implementação de Proxy e Decorator é que o Proxy não oferece um construtor que aceite um objeto ConsultaCliente, desta forma Proxy não permite comportamentos encadeados recursivamente por agregação.

Agora a questão é que podemos sim implementar serviço de computação distribuída e recursos de acesso remoto via TCP usando um Decorator e com isso você poderá enfrentar alguma dificuldade em decidir quando usar Proxy e quando usar Decorator. A seguinte lista de itens deve esclarecer algumas delas:

- Decorator tem a idéia de se criar comportamentos que poderão ser ligados ou desligados de objetos;
- Decorator oferece comportamento comum a um conjunto de classes evitando combinações de heranças múltiplas de classes para comportamento composto;
- Proxy poderá reunir uma série de serviços (transação, computação distribuída, segurança) em uma só classe Proxy, enquanto com Decorator você desenvolveria cada funcionalidade em uma sub-classe diferente.
- Com Decorator o aplicativo cria o objeto alvo de decoração e depois aplica os Decorators que lhe interessa, enquanto com proxy o aplicativo simplesmente acessa o objeto Proxy pensando ser o objeto de verdade e não tem possibilidades de encadear novos comportamentos ao objeto Proxy.
- Use Decorator quando quiser compartilhar comportamentos granularizados “plugáveis” em uma família de classes.
- Use Proxy quando quer assumir o controle de um objeto de forma transparente para o usuário.

Anotações

4.25.4 Estrutura e Participantes



- **Proxy (ContaSegura)**
Objeto que manterá uma referência para um objeto intermediado.
- **Subject (Conta)**
Representa a definição da interface do objeto que será acessado indiretamente via *Proxy*.
- **RealSubject (ContaCorrente)**
Representa a implementação concreta do objeto gerenciado pelo *Proxy*.

Anotações

4.26 Builder - Criação

Builder propõe que o processo de construção de um objeto, quando complexo, seja separado da definição do objeto, para que possamos ter diferentes algoritmos de construção permitindo diferentes representações para o objeto.

A intenção deste design pattern é bastante simples e tecnicamente na plataforma Java somos usuários de diversas API's que trabalham com Builder:

- DocumentBuilder: para parsing de XML;
- ProcessBuilder: para execução de comandos em sistemas operacionais;

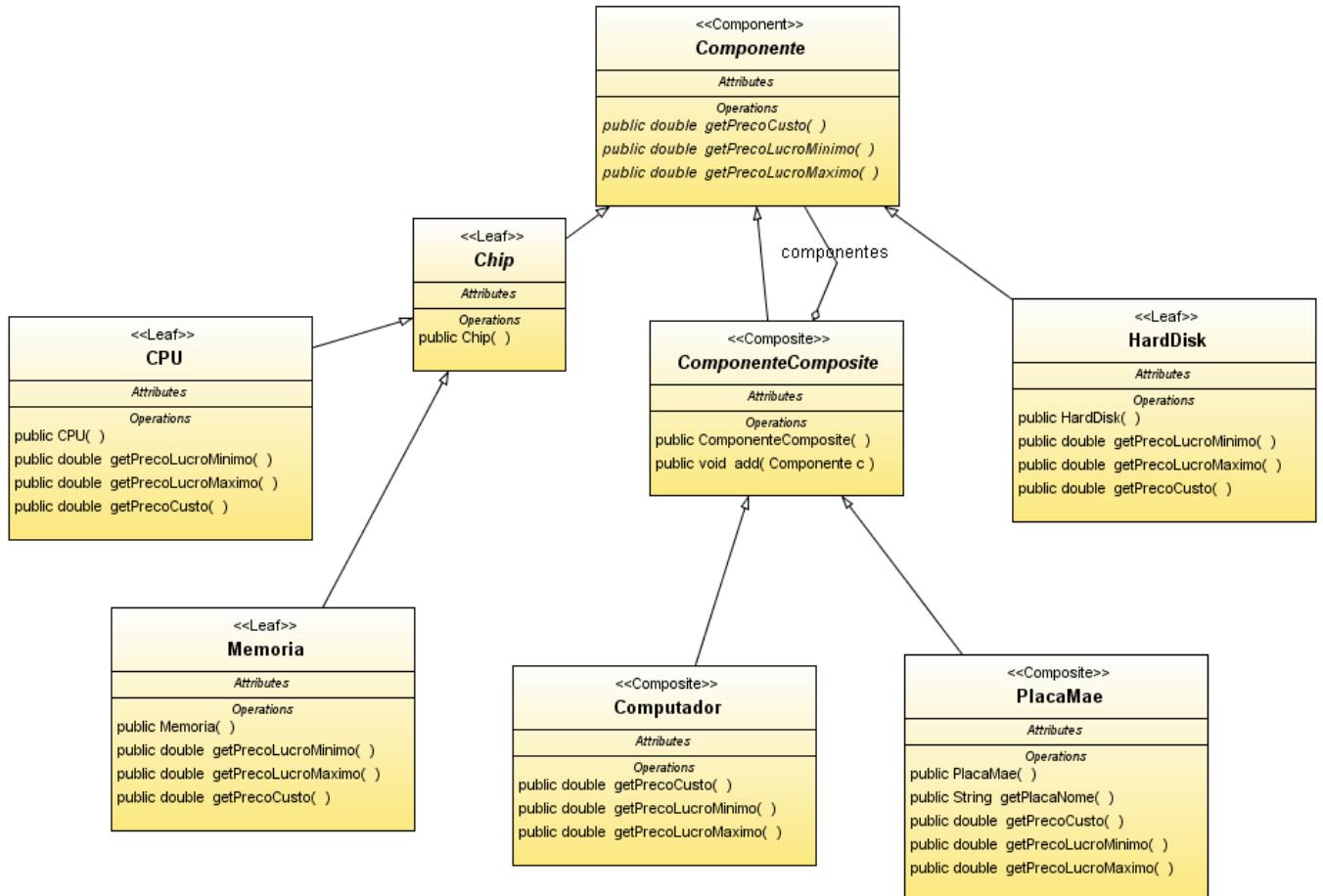
Do GoF: “Um Composite é o que geralmente um Builder faz”

Devemos lembrar desde já que Factory Method e Abstract Factory são padrões que trabalham na criação de objetos de uma família de classes, trabalhando intensamente com polimorfismo, enquanto o Builder tem o foco nos detalhes de construção de uma instância de objeto, que não necessariamente terá uma família de sub-classes.

Anotações

4.26.1 Anti-pattern

Para apresentar o Builder com uma implementação bem próxima das implementações populares, que em geral unem Composite com Builder, vamos voltar para o modelo de objetos Composite que criamos para nosso sistema de orçamento de computadores. Vejamos o diagrama a seguir para relembrar sua estrutura:



Anotações

Agora imagine uma situação onde precisamos construir diferentes tipos de computadores com diferentes componentes: servidor, computador para games e computador básico. Poderíamos criar métodos em classes aleatórias que criam os computadores porém as diferentes lógicas de criação do computador ficariam espalhadas pelo código. O padrão Builder visa organizar justamente os processos de criação de objetos complexos. Vejamos o código que será refatorado:

Exemplo: Exemplo1.java

```

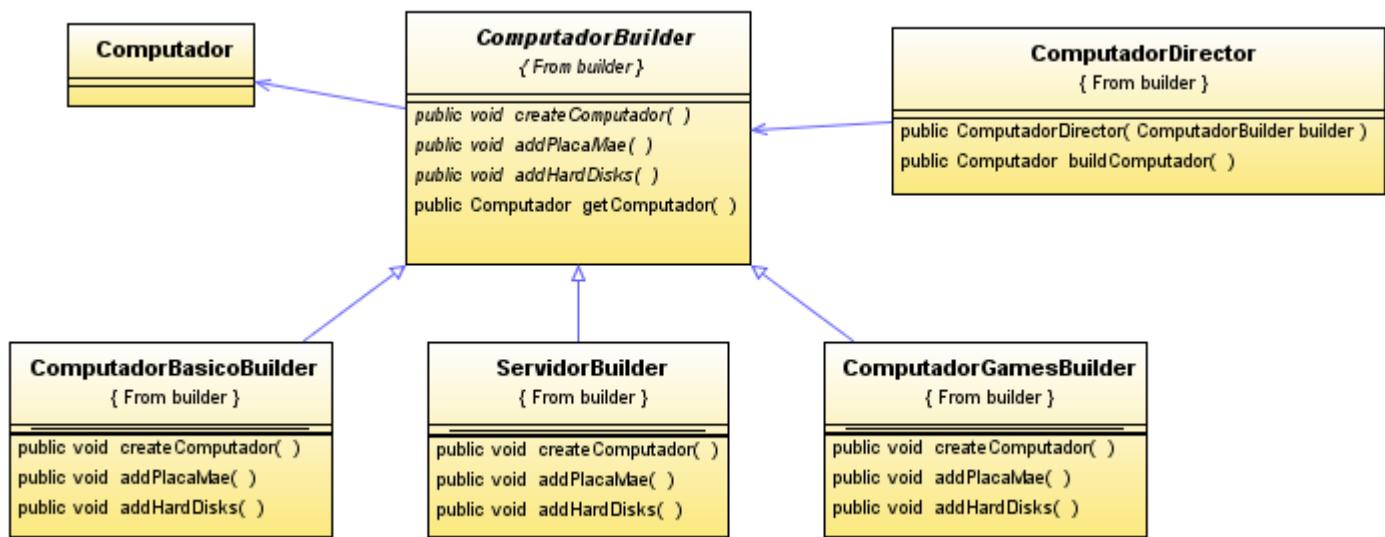
1 package br.com.globalcode.cp.builder.antipattern;
2
3 public class Exemplo1 {
4     public static Computador criarComputadorBasico() {
5         Computador c1 = new Computador();
6         PlacaMae placaMae = new PlacaMae();
7         placaMae.add(new Memoria());
8         placaMae.add(new CPU());
9         placaMae.add(new PlacaVideo());
10        c1.add(placaMae);
11        c1.add(new HardDisk());
12        return c1;
13    }
14    public static Computador criarComputadorGame() {
15        Computador c1 = new Computador();
16        PlacaMae placaMae = new PlacaMae();
17        AceleradorGrafico aceleradorVideo = new AceleradorGrafico();
18        PlacaVideo placaVideo = new PlacaVideo();
19        placaVideo.add(aceleradorVideo);
20        HardDisk hd1 = new HardDisk();
21        placaMae.add(new Memoria());
22        placaMae.add(new CPU());
23        placaMae.add(placaVideo);
24        c1.add(placaMae);
25        c1.add(hd1);
26        return c1;
27    }
28    public static Computador criarServidor() {
29        Computador c1 = new Computador();
30        PlacaMae placaMae = new PlacaMae();
31        PlacaVideo placaVideo = new PlacaVideo();
32        placaMae.add(new Memoria());
33        placaMae.add(new CPU());
34        placaMae.add(placaVideo);
35        c1.add(placaMae);
36        c1.add(new HardDisk());
37        c1.add(new HardDisk());
38        c1.add(new HardDisk());
39        return c1;
40    }
41    public static void main(String[] args) {
42        Computador servidor = criarServidor();
43        Computador basico = criarComputadorBasico();
44        //...
45    }
46 }
```

Anotações

4.26.2Aplicando o design pattern

No fundo a criação do nosso objeto Computador segue uma mesma sequência de passos, variando apenas os componentes específicos que são adicionados dependendo do tipo de computador desejado.

Vamos criar um modelo de classes onde teremos uma classe abstrata que define as operações necessárias para a criação do computador e uma classe que sabe qual a ordem na qual estas operações devem ser executadas. Para cada tipo de computador teremos uma sub-classe da classe abstrata base:



Anotações

Vejamos o exemplo de implementação para cada um dos elementos apresentado no diagrama de classes:

Exemplo: ComputadorBuilder.java

```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.Computador;
4
5 public abstract class ComputadorBuilder {
6
7     protected Computador computador;
8
9     public abstract void createComputador();
10    public abstract void addPlacaMae();
11    public abstract void addHardDisks();
12    public Computador getComputador() {
13        return computador;
14    }
15 }
```

Exemplo: ComputadorDirector.java

```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.Computador;
4
5 public class ComputadorDirector {
6
7     ComputadorBuilder builder;
8
9     public ComputadorDirector(ComputadorBuilder builder) {
10         this.builder = builder;
11     }
12     public Computador buildComputador() {
13         builder.createComputador();
14         builder.addPlacaMae();
15         builder.addHardDisks();
16         return builder.getComputador();
17     }
18 }
```

Anotações

Exemplo: ComputadorBasicoBuilder.java

```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.*;
4
5 public class ComputadorBasicoBuilder extends ComputadorBuilder{
6
7     public void createComputador() {
8         computador = new Computador();
9     }
10
11    public void addPlacaMae() {
12        PlacaMae placaMae = new PlacaMae();
13        placaMae.add(new Memoria());
14        placaMae.add(new CPU());
15        placaMae.add(new PlacaVideo());
16        computador.add(placaMae);
17    }
18
19    public void addHardDisks() {
20        computador.add(new HardDisk());
21    }
22 }
```

Exemplo: ComputadorGamesBuilder.java

```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.*;
4 public class ComputadorGamesBuilder extends ComputadorBuilder{
5
6     public void createComputador() {
7         computador = new Computador();
8     }
9
10    public void addPlacaMae() {
11        PlacaMae placaMae = new PlacaMae();
12        AceleradorGrafico aceleradorVideo = new AceleradorGrafico();
13        PlacaVideo placaVideo = new PlacaVideo();
14        placaVideo.add(aceletorVideo);
15        placaMae.add(new Memoria());
16        placaMae.add(new CPU());
17        placaMae.add(placaVideo);
18        computador.add(placaMae);
19    }
20
21    public void addHardDisks() {
22        HardDisk hd1 = new HardDisk();
23        computador.add(hd1);
24    }
25 }
```

Anotações

Exemplo: ServidorBuilder.java

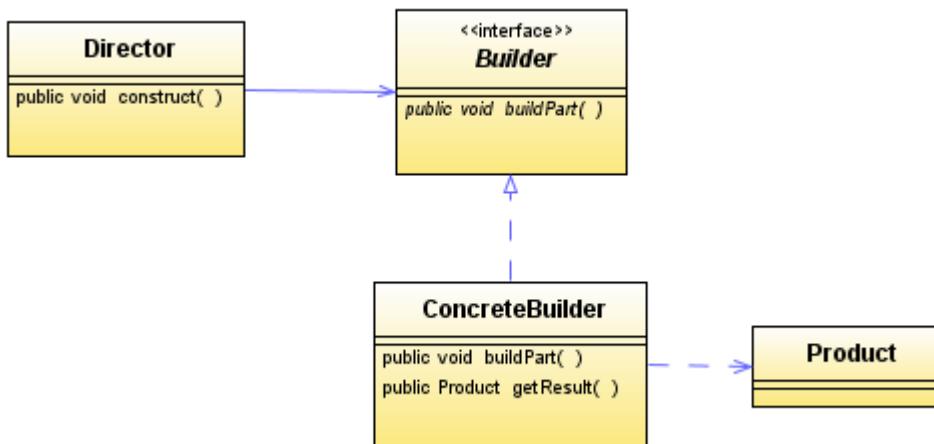
```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.*;
4
5 public class ServidorBuilder extends ComputadorBuilder{
6
7     public void createComputador() {
8         computador = new Computador();
9     }
10
11    public void addPlacaMae() {
12        PlacaMae placaMae = new PlacaMae();
13        PlacaVideo placaVideo = new PlacaVideo();
14        placaMae.add(new Memoria());
15        placaMae.add(new CPU());
16        placaMae.add(placaVideo);
17        computador.add(placaMae);
18    }
19
20    public void addHardDisks() {
21        computador.add(new HardDisk());
22        computador.add(new HardDisk());
23        computador.add(new HardDisk());
24    }
25 }
```

Exemplo: Aplicativo.java

```
1 package br.com.globalcode.cp.builder;
2
3 import br.com.globalcode.cp.iterator.Computador;
4
5 public class Aplicativo {
6
7     public static void main(String[] args) {
8         ServidorBuilder builder = new ServidorBuilder();
9         ComputadorDirector director = new ComputadorDirector(builder);
10        Computador computador = director.buildComputador();
11    }
12 }
```

Anotações

4.26.3 Estrutura e Participantes



- **Builder** (`ComputadorBuilder`)
Especifica a interface para criação de um objeto produto.
- **ConcreteBuilder**
(`ComputadorGameBuilder`, `ComputadorServidorBuilder`, `ComputadorBasicobuilder`)
São as implementações de *Builder*.
- **Director** (`ComputadorDirector`)
Classe que sabe o algoritmo de construção do objeto.
- **Product** (`Computador`)
Representa o objeto-produto final.

Anotações

4.27 Prototype - Criação

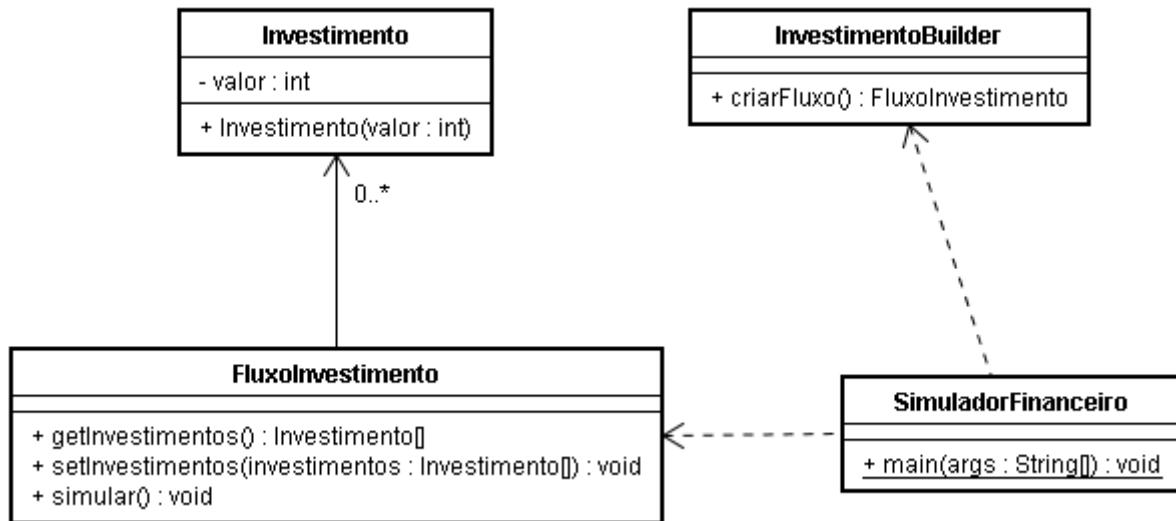
Com intenção objetiva o design pattern Prototype representa a técnica de se criar um objeto utilizando o clone de uma instância-protótipo. Em sistemas com grande volume de dados, podemos imaginar que determinados objetos de grande porte e construção complexa serão utilizados diversas vezes, com diferentes cópias.

Um sistema de mercado financeiro pode ter um conjunto de objetos que representam o fluxo de investimento do banco para 20 anos, e cada simulação financeira vai utilizar um cópia deste fluxo para apresentar a análise de um cenário. Construir este conjunto de objetos que representam o fluxo de investimento a cada simulação pode representar um processo muito caro, pois em geral vai envolver a recuperação dos dados em uma base e depois uma série de cálculos. Portanto, neste caso o padrão Prototype propõe que o objeto tenha algum sistema de clone e saiba fornecer para o cliente uma cópia de si mesmo.

Com Java temos a interface `Cloneable` e facilidades em APIs para implementar o sistema de clone de objetos, aplicando então o padrão Prototype.

4.27.1 Anti-pattern

Vamos imaginar este cenário financeiro de simulações onde temos o seguinte modelo de objetos:



Cada `FluxoInvestimento` terá 20.000 objetos `Investimento` e cada simulação utilizará um cópia de um objeto `FluxoInvestimento`.

Anotações

A classe de construção de objetos FluxoInvestimento sem o padrão Prototype teria o seguinte código:

Exemplo: InvestimentoBuilder.java

```
1 package br.com.globalcode.cp.prototype.antipattern;
2
3 public class InvestimentoBuilder {
4
5     public static FluxoInvestimento criarFluxo() {
6         FluxoInvestimento fluxo = new FluxoInvestimento();
7         int totalInv = 20000;
8         for(int i=0;i<totalInv;i++) {
9             fluxo.getInvestimentos().add(new Investimento(i));
10        }
11    return fluxo;
12  }
13 }
```

E teremos o código de um SimuladorFinanceiro que vai criar diversas cópias de FluxoInvestimento para poder trabalhar as simulações:

Exemplo: SimuladorFinanceiro.java

```
1 package br.com.globalcode.cp.prototype.antipattern;
2
3 import java.util.ArrayList;
4
5 public class SimuladorFinanceiro {
6
7     public static void main(String[] args) {
8         System.out.println("Criando 1000 objetos de Fluxo:");
9         System.out.println("Hora inicio: " + new java.util.Date());
10        ArrayList teste = new ArrayList();
11        for (int i = 0; i < 1000; i++) {
12            teste.add(InvestimentoBuilder.criarFluxo());
13        }
14        System.out.println("Hora termino: " + new java.util.Date());
15    }
16 }
```

Anotações

4.27.2 Aplicando o design pattern

Com Java vamos implementar a interface de flag `Cloneable` e também tornar o método `clone` public (o que herdamos é `protected`) que vai chamar o `clone` da super-classe:

Exemplo: FluxoInvestimento.java

```

1 package br.com.globalcode.cp.prototype;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5
6 public class FluxoInvestimento implements Cloneable {
7     private Collection<Investimento> investimentos =
8         new ArrayList<Investimento>();
9     public Collection<Investimento> getInvestimentos() {
10         return investimentos;
11     }
12     public void setInvestimentos(Collection<Investimento> investimentos) {
13         this.investimentos = investimentos;
14     }
15     public void simular() {
16         System.out.println("Calculando...");
17     }
18     public FluxoInvestimento clone() {
19         try {
20             return (FluxoInvestimento) super.clone();
21         } catch (CloneNotSupportedException ex) {
22             ex.printStackTrace();
23             return null;
24         }
25     }
26 }
```

E agora a classe `builder` vai `clonar` o objeto no lugar de criá-lo manualmente:

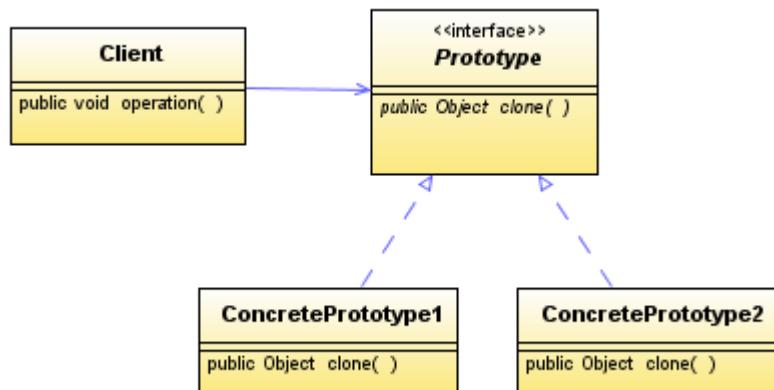
Exemplo: InvestimentoBuilder.java

```

1 package br.com.globalcode.cp.prototype;
2
3 public class InvestimentoBuilder {
4     private static FluxoInvestimento fluxo;
5     private static void criarFluxoInicial() {
6         fluxo = new FluxoInvestimento();
7         int totalInv = 20000;
8         for(int i=0;i<totalInv;i++) {
9             fluxo.getInvestimentos().add(new Investimento(i));
10        }
11    }
12    public static FluxoInvestimento criarFluxo() {
13        if(fluxo==null) criarFluxoInicial();
14        return fluxo.clone();
15    }
16 }
```

Anotações

4.27.3 Estrutura e Participantes



- **Prototype** (`java.lang.Object`)
Especifica a interface que permitirá o clone.
- **ConcretePrototype** (`FluxoInvestimento`)
Representa a implementação concreta da classe que terá uma instancia protótipo.
- **Client** (`SimuladorFinanceiro`)
Quem solicita um novo objeto clone do protótipo.

Anotações

4.28 Laboratório 8

Objetivo:

Você deverá refatorar o framework de comandos para trabalhar com Prototype.

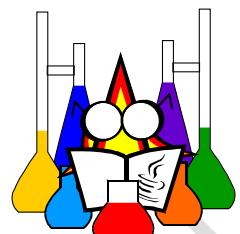


Tabela de atividades:

Atividade	OK
1. Faça o download do arquivo lab07.zip a partir da URL indicada pelo instrutor(a).	
2. Descompacte e crie um projeto com NetBeans ou Eclipse;	
3. Implemente Prototype na fábrica de Commands permitindo que objetos command sejam criados a partir de protótipos	

Anotações

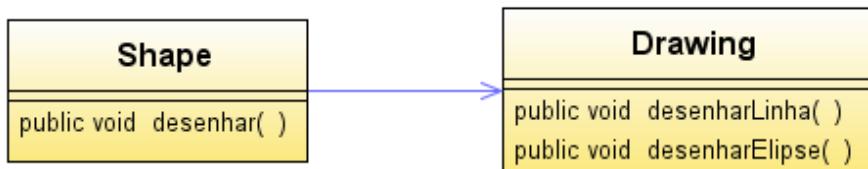
4.29 Bridge - Estrutura

*Conhecido também como Handle/Body

Segundo o GOF, o design pattern Bridge tem como objetivo desacoplar uma abstração de sua implementação, permitindo que ambas variem de forma independente.

Para um correto entendimento deste pattern é fundamental entender o que é uma abstração e o que é uma implementação. Numa análise simplista podemos interpretar a abstração como sendo uma interface ou classe abstrata e as implementações as diversas classes concretas que implementam a interface ou estendem a classe abstrata. Do ponto de vista de análise orientada a objetos esta visão não está errada, mas no pattern bridge temos que ir um passo além.

Consideramos uma abstração a representação de um elemento com o qual o aplicativo cliente tem interesse em interagir e a implementação a representação das principais operações que a abstração deve realizar. Assim, temos por exemplo no diagrama abaixo um exemplo clássico utilizado na descrição deste pattern.



Neste exemplo temos um aplicativo gráfico qualquer que manipula formas geométricas. Assim a classe Shape, que representa uma forma geométrica é a abstração. Neste elemento definimos uma única operação desenhar, que será chamada pelo aplicativo cliente. A implementação é a classe Drawing, que define as operações mais específicas que serão executadas pela abstração Shape.

Assim do ponto de vista do aplicativo cliente, quando for necessário desenhar uma forma geométrica será chamado o método desenhar da cada Shape. Porém, podemos ter diversas formas geométricas, como Círculo, Retângulo ou formas mais complexas que são sub-classes de Shape. Cada sub-classe irá chamar os métodos necessários para ela seja desenhada. Estes métodos estão definidos na classe Drawing, que no nosso exemplo são: desenharLinha e desenharElipse.

Qual a vantagem desta abordagem? Ela permite que a hierarquia de abstrações (sub-classes de Shape) e a hierarquia de implementações (sub-classes de Drawing) variem de forma independente. Assim, podemos ter diversos tipos de formas geométricas desenhadas por diversos elementos de desenho diferentes. Estes elementos de desenho podem corresponder, por exemplo a diversas bibliotecas gráficas distintas. Neste caso,

Anotações

poderíamos inclusive utilizar o pattern Adapter para aproveitar interfaces distintas de cada uma destas bibliotecas gráficas.

Dois princípios básicos normalmente presentes no pattern são:

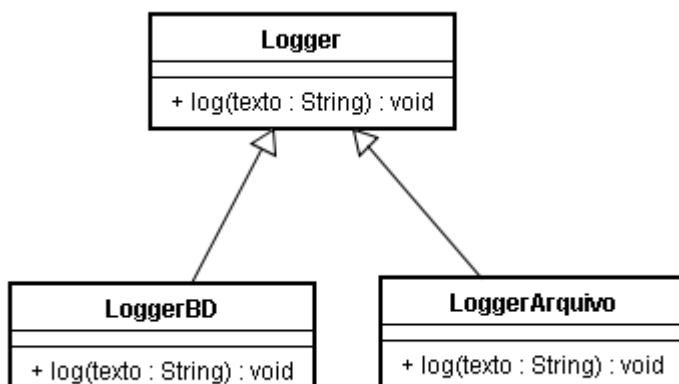
- identificar o que varia e encapsular estes elementos: no exemplo podemos ter diversas formas geométricas distintas e diversas bibliotecas de desenho.
- preferir a composição ao invés da herança: A classe Shape possui um atributo Drawing ao invés de termos diversas sub-classes de Shape para cada biblioteca de desenho distinta.

4.29.1 Anti-pattern

Vamos apresentar o anti-pattern de Bridge simulando diversos requisitos variáveis que foram passados para uma equipe de desenvolvimento ao longo do tempo. É muito comum que os anti-patterns surjam aos poucos durante o desenvolvimento de um sistema, o que faz com que muitas vezes demorem a ser percebidos.

- foi solicitado o desenvolvimento de uma classe utilitária para geração de mensagens de log.
- Inicialmente teremos duas possibilidades de locais de gravação de log: arquivo e banco de dados.
- Os locais de gravação de log são mutuamente exclusivos. Alguns aplicativos que utilizam o logger gravam sempre em bancos de dados, outros gravam sempre em arquivos.

A partir desta solicitação foi criada a seguinte hierarquia de classes:

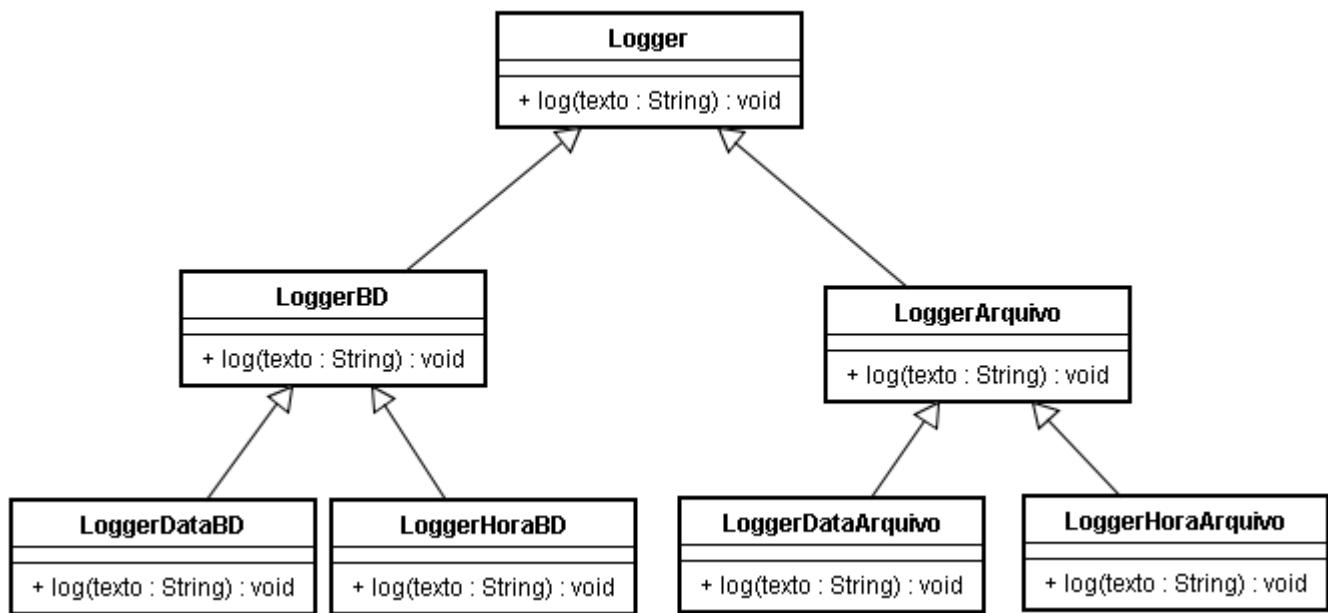


Posteriormente surgiu a seguinte solicitação:

Anotações

- Alguns aplicativos devem ter a mensagem de log formatada de acordo com a data e outros de acordo com a hora.

Um novo diagrama foi feito:



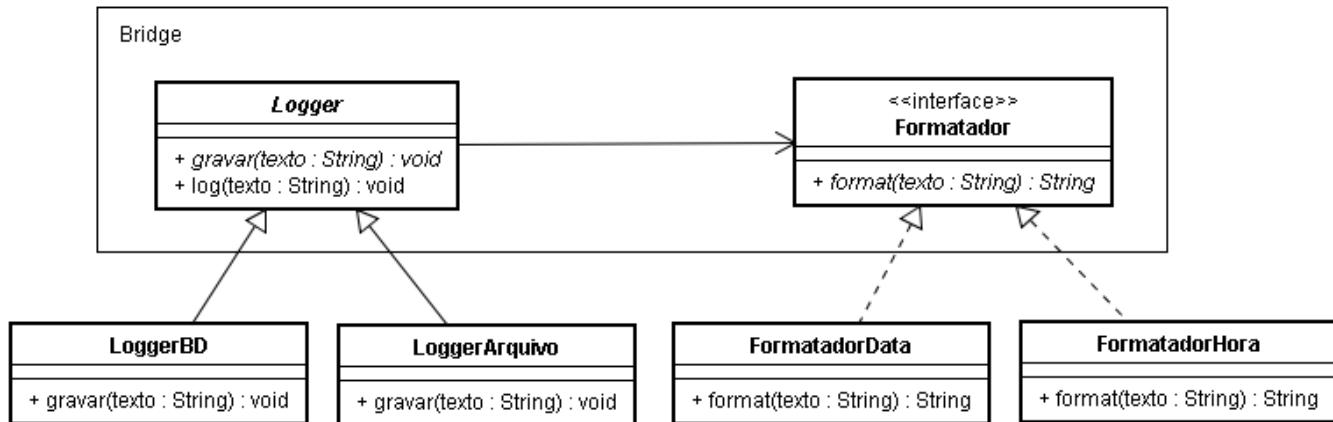
Qual o problema com essa solução?

- provavelmente o código de formatação da mensagem está duplicado. A formatação por data deve ser igual tanto para a classe `LoggerDataBD` quanto para a classe `LoggerDataArquivo`.
- Para cada novo requisito de formatação duas novas classes devem ser geradas: uma para BD e outra para arquivo
- Para cada novo formato de gravação do log duas novas classes devem ser geradas: uma formatada por data e outra por hora.

Anotações

4.29.2Aplicando o design pattern

Se aplicarmos os dois princípios básicos que vimos anteriormente chegamos a uma estrutura de dados correspondente ao pattern Bridge. Identificando o que está variando na nossa estrutura temos: os locais de gravação e o formato das mensagens de log.



Com essa estrutura podemos criar novos formatadores de mensagens e adicionar o suporte a outros tipos de locais de gravação com impacto mínimo na hierarquia de classes.

O código das classes utilizadas encontra-se a seguir:

Exemplo: Logger.java

```

1 package br.com.globalcode.cp.bridge;
2
3 public abstract class Logger {
4
5     Formatador formatador;
6
7     public void setFormatador(Formatador formatador) {
8
9     }
10
11 package br.com.globalcode.cp.bridge;
12
13 public class LoggerDB extends Logger {
14
15     ...
16 }
17
18 package br.com.globalcode.cp.bridge;
19
20 public class LoggerArquivo extends Logger{
21
22     ...
23 }
24
25 package br.com.globalcode.cp.bridge;
26
27 public interface Formatador {
28
29     ...
30 }
31
32 package br.com.globalcode.cp.bridge;
33
34 public class FormatadorData implements Formatador{
35     public String format(String texto) {
36         String novoTexto = null;
37         //regra para formatação do texto por data completa
38         return novoTexto;
39     }
40 }
  
```

```

1 package br.com.globalcode.cp.bridge;
2
3 public class FormatadorHora implements Formatador {
4     public String format(String texto) {
5         String novoTexto = null;
6         //regra para formatação do texto por hora
7         return novoTexto;
8     }
9 }
```



- **Abstraction** (`Logger`)

Define a interface da abstração e mantém uma referência para um objeto do tipo *Implementor*.

- **Implementor** (`Formatador`)

Define a interface para as classes de implementação.

- **RefinedAbstraction** (`LoggerDB`, `LoggerArquivo`)

Implementa a interface definida por *Abstraction*.

- **ConcreteImplementor** (`FormatadorHora`, `FormatadorData`)

Implementa a interface definida por *Implementor*.

Anotações

4.30 Interpreter - Comportamento

Com um foco bastante específico, o design pattern Interpreter propõe um modelo de objetos para definir uma representação para as sentenças gramaticais de uma linguagem, de forma que a linguagem poderá facilmente ampliar sua gramática e respectivos algoritmos de interpretação.

Apesar de não ser comum em cenários corporativos a necessidade de se criar uma linguagem, podemos imaginar cenários menores onde precisamos prover flexibilidade de configuração e customização de um sistema, e este processo pode acabar definindo uma mini-linguagem que deverá ser interpretada.

Um exemplo concreto de implementação do padrão Interpreter ocorre na criação de interpretadores de expressões regulares. Expressões regulares servem para verificar se um determinado texto segue um determinado padrão. Elas possuem uma sintaxe e gramática própria para definição dos padrões que devem ser verificados. Utilizando o padrão Interpreter fica fácil estender futuramente a gramática com novos elementos.

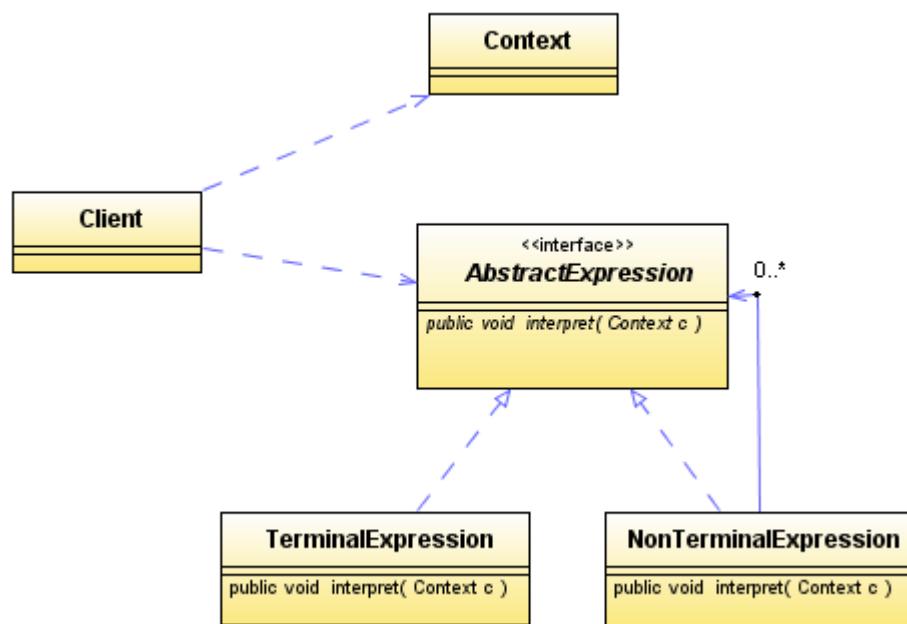
O pattern Interpreter é aplicável em casos como por exemplo:

- linguagens de Query especializadas como EJB-QL e HQL
- linguagens utilizadas para descrever protocolos de comunicação
- linguagens para orquestração de serviços
- qualquer linguagem de domínio especializada

Anotações

4.30.1 Estrutura e Participantes

O seguinte diagrama de classes determina a estrutura geral de utilização do pattern



- **AbstractExpression** (`ExpressaoBooleana`)

Classe abstrata ou interface que declara uma operação comum a todos os elementos da gramática sendo descrita.

- **TerminalExpression** (`LiteralBooleana - true ou false`)

Representa cada um dos elementos literais da gramática.

- **NonTerminalExpression** (`ExpressaoAnd, ExpressaoOr`)

Implementa cada uma das regras da gramática.

- **Context**

Contém informação global para o interpretador (Ex: a String que contém a expressão sendo interpretada).

- **Client** (`Parser`)

Constrói ou recebe o conjunto de elementos que representam uma determinada sentença na linguagem definida pela gramática.

Anotações

4.31 Flyweight - Estrutura

Flyweight é um padrão que propõe a técnica de compartilhamento de objetos para trabalhar com grandes volumes de dados granularizados. Uma forma bastante evidente de entendermos o padrão aplicado na plataforma Java, é analisarmos o seguinte código:

Exemplo: StringJava.java

```
1 package br.com.globalcode.cp.flyweight;
2
3 public class StringJava {
4     public static void main(String[] args) {
5         String x="Globalcode";
6         String y="Globalcode";
7         if(x==y) {
8             System.out.println("Mesmo objeto");
9         }
10        else {
11            System.out.println("Objetos diferentes");
12        }
13    }
14 }
```

Por todas as filosofias já conhecidas, quando comparamos referências a objetos com o operador `==`, não estamos comparando o conteúdo do objeto em si, mas sim se ambas as referências apontam para o mesmo objeto na memória heap. Com isso poderíamos afirmar que o código acima imprimaria na console “Objetos diferentes”, porém ocorre justamente o oposto.

O motivo é que o compilador Java concluiu que a String “Globalcode” está escrita mais do que uma vez no código-fonte (e poderia estar escrita centenas de vezes), com isso ele decidiu por criar um único objeto “Globalcode”, depositar em uma coleção de strings reusáveis (pool) e colocou a mesma referência para x e y. Nada disso aconteceria se tivéssemos tomado o dado do usuário ou então se chamassemos explicitamente o construtor da String: `String x=new String("Globalcode");`.

Flyweight é justamente essa idéia de compartilhar os objetos para trabalhar com um conjunto grande de dados granularizados, afinal um sistema Java pode ter mais de 10.000 classes com Strings em muitas partes.

Anotações

4.31.1 Anti-pattern

Imagine um sistema implementado para uma organização que lida com remessas, como por exemplo os correios, utilizando uma classe que gerencia remessas de acordo com o código de exemplo a seguir:

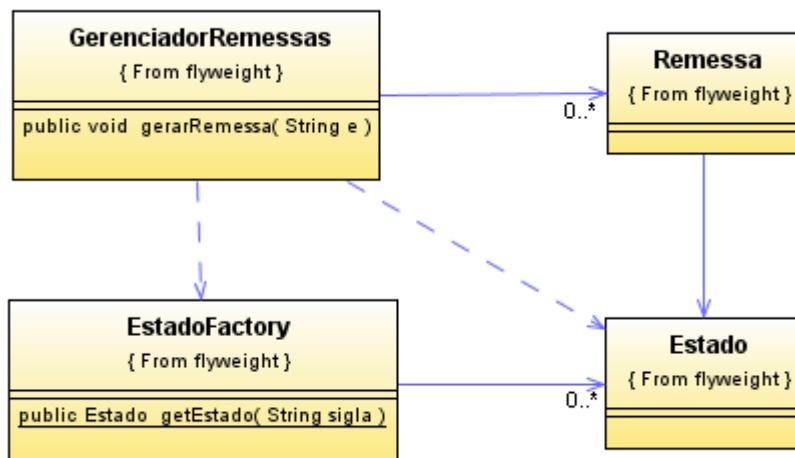
Exemplo: GerenciadorRemessas.java

```
1 package br.com.globalcode.cp.flyweight.antipattern;
2
3 import br.com.globalcode.cp.flyweight.Remessa;
4 import br.com.globalcode.cp.flyweight.Estado;
5 import java.util.*;
6
7 public class GerenciadorRemessas {
8
9     Collection<Remessa> remessas = new ArrayList<Remessa>();
10
11    public void gerarRemessa(String e) {
12        Estado estado = new Estado(e);
13        remessas.add(new Remessa(estado));
14    }
15 }
```

Se forem geradas muitas remessas para o mesmo Estado, estaremos criando inúmeras instâncias desnecessárias que poderiam ser compartilhadas.

4.31.2 Aplicando o design-pattern

Para aplicar o design pattern temos que criar uma Factory que gerencie a criação dos objetos de Estado, reaproveitando instâncias e somente criando novos objetos quando for realmente necessário. O diagrama de classes para a estrutura utilizada se encontra a seguir:



A implementação

Anotações

Exemplo: GerenciadorRemessas.java

```

1 package br.com.globalcode.cp.flyweight;
2
3 import java.util.*;
4
5 public class GerenciadorRemessas {
6
7     Collection<Remessa> remessas = new ArrayList<Remessa>();
8
9     public void gerarRemessa(String e) {
10         Estado estado = EstadoFactory.getEstado(e);
11         remessas.add(new Remessa(estado));
12     }
13 }
```

Exemplo: EstadoFactory.java

```

1 package br.com.globalcode.cp.flyweight;
2
3 import java.util.*;
4
5 public class EstadoFactory {
6
7     private static Map<String,Estado> estados = new HashMap<String,Estado>();
8
9     public static Estado getEstado(String sigla) {
10         Estado estado = estados.get(sigla);
11         if(estado == null) {
12             estado = new Estado(sigla);
13             estados.put(sigla,estado);
14         }
15         return estado;
16     }
17 }
```

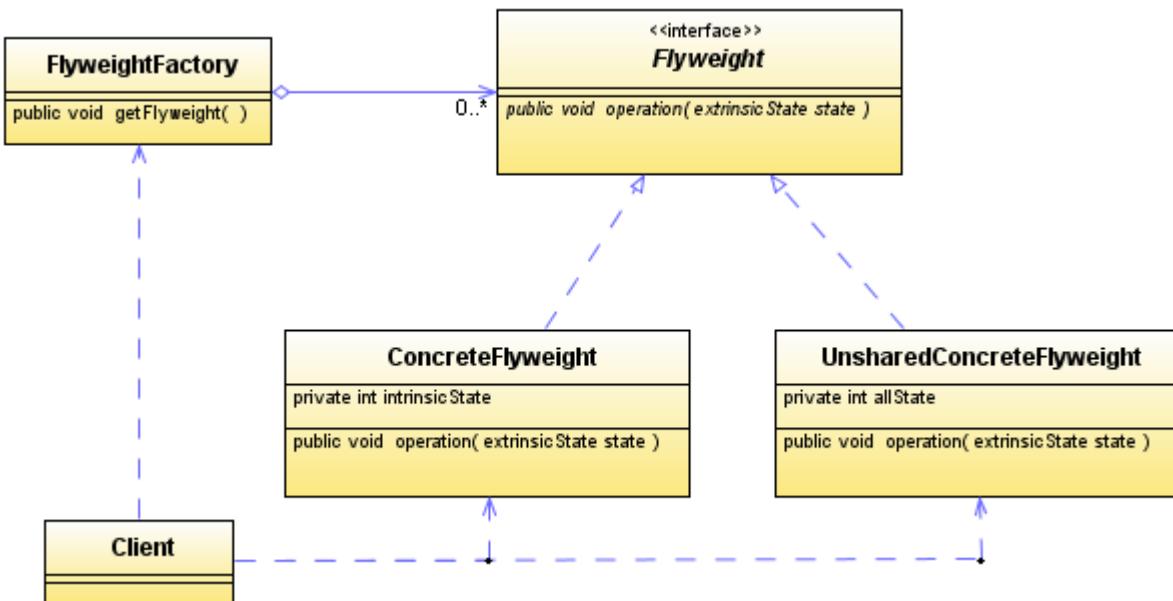
4.31.3 O padrão Flyweight nas APIs do Java

Não é muito usual implementar o padrão Flyweight nos sistemas que ocorrem no dia a dia. Usualmente nos elementos em que podemos nos favorecer mais na economia de criação de instâncias, já foi feita uma implementação nas próprias APIs do Java. Exemplos de classes que utilizam o padrão são:

- String
- Wrapper classes para valores inteiros numéricos até o valor de 127
- Enums
- classes da API de Swing como classes que representam bordas (implementações de javax.swing.border.Border) e nós de JTrees.

Anotações

4.31.4 Estrutura e Participantes



- **Flyweight**

declara uma interface para os flyweights. No nosso caso não foi criada.

- **ConcreteFlyweight (Estado)**

Implementação de *Flyweight*.

- **UnsharedConcreteFlyweight**

Em alguns casos podemos ter implementações da interface que não queremos que sejam compartilhadas.

Por exemplo poderíamos ter remessas internacionais que são pouco freqüentes e qua não precisamos nos preocupar com um número grande de instâncias.

- **FlyweightFactory (EstadoFactory)**

Classe que cria e gerencia os objetos *Flyweight*.

- **Client (GerenciadorRemessas, Remessa)**

Classes que utilizam ou mantém referências para os objetos *Flyweight*.

Anotações

CAPÍTULO

5

Design Patterns Java EE
Blueprints

Para uso não comercial

Anotações

5 Design patterns Java EE Blueprints

Depois da imersão nos padrões GoF a tarefa de se aprender um novo padrão fica bastante simples, pois toda filosofia de design patterns já está bem assimilada além do fato de alguns padrões GoF terem complexidade de contexto muito superior que os demais padrões mais regionalizados em uma tecnologia, como por exemplo os padrões Blueprints. Você poderá notar que os padrões Blueprints mais tratam de dividir responsabilidades do que efetivamente propor um modelo de estrutura de comunicação e relacionamento entre classes e objetos como são grande parte dos padrões GoF. A tabela a seguir apresenta um resumo dos design patterns que serão estudados:

Design-pattern	Descrição
Application Controller	Centraliza e modulariza o gerenciamento de visualizações (views) e ações (actions) do aplicativo.
Business Delegate	Encapsula o acesso a camada de negócios
Composite View	Divide uma “tela” em pequenos pedaços com o objetivo de reaproveitamento.
Data Access Object	Centraliza código de acesso aos dados em repositórios persistentes.
Front Controller	Centraliza as requisições do aplicativo em uma classe ou uma pequena porção de classes.
Intercepting Filter	Intercepta requisições de um usuário, sem que exista um vínculo físico entre o interceptador e o interceptado.
Model-view-controller	Separa os objetos da aplicação em três principais partes: - modelo, controle e visualização. O principal ganho é que podemos reaproveitar melhor cada uma dessas partes.
Service Locator	Centraliza código de acesso a recursos externos.
Transfer Object	Transfere cópia de objetos entre camadas para reduzir tráfego.
View Helper	Encapsula código utilizado na camada de visualização em componentes, visando prover maior reuso e também facilidade de programação.

Anotações

5.1 Service Locator

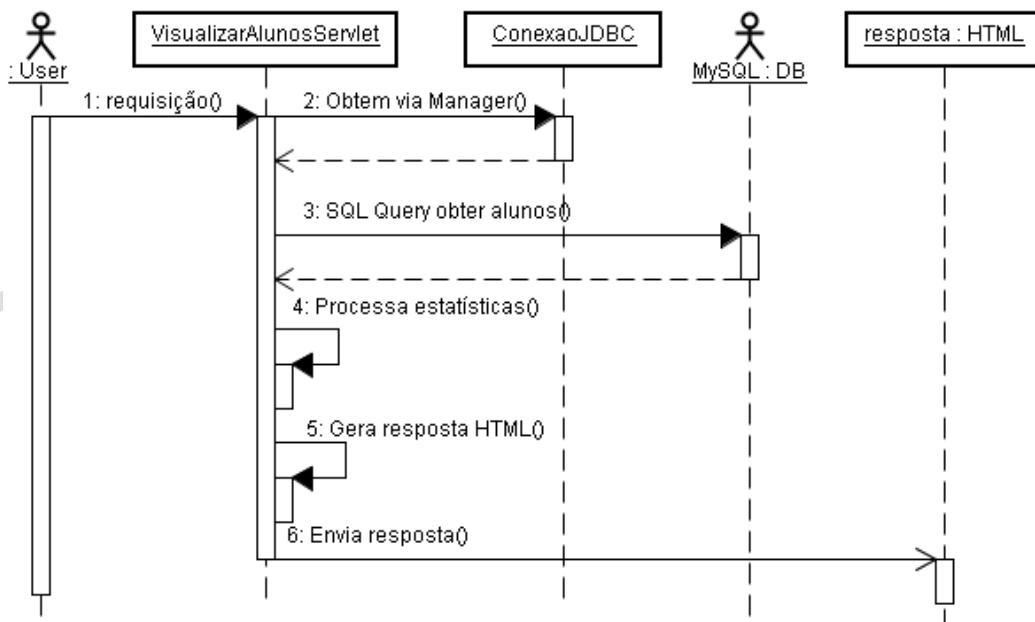
Em aplicações enterprise que utilizam EJB, pool de conexões e outros componentes distribuídos, utilizamos JNDI (Java Naming & Directory Interface) para fazer lookup e obter uma referência para estes objetos. Quando esta operação é repetida muitas vezes em diferentes lugares do código, a manutenção é dificultada e a performance pode ser afetada pela criação desnecessária de JNDI initial contexts e número de lookups.

Service Locator pode centralizar e fazer cache em um pequeno conjunto de classes, as quais terão a responsabilidade de localizar e fornecer referências aos principais recursos utilizados na aplicação, facilitando a manutenção e reduzindo a duplicidade de código. Os seguintes benefícios podem ser obtidos:

- Encapsula a complexidade do lookup;
- Acesso uniforme aos serviços;
- Facilita a adição de novos componentes;
- Aumenta a performance da rede;
- Aumenta a performance do cliente através de cache.

5.1.1 Anti-pattern

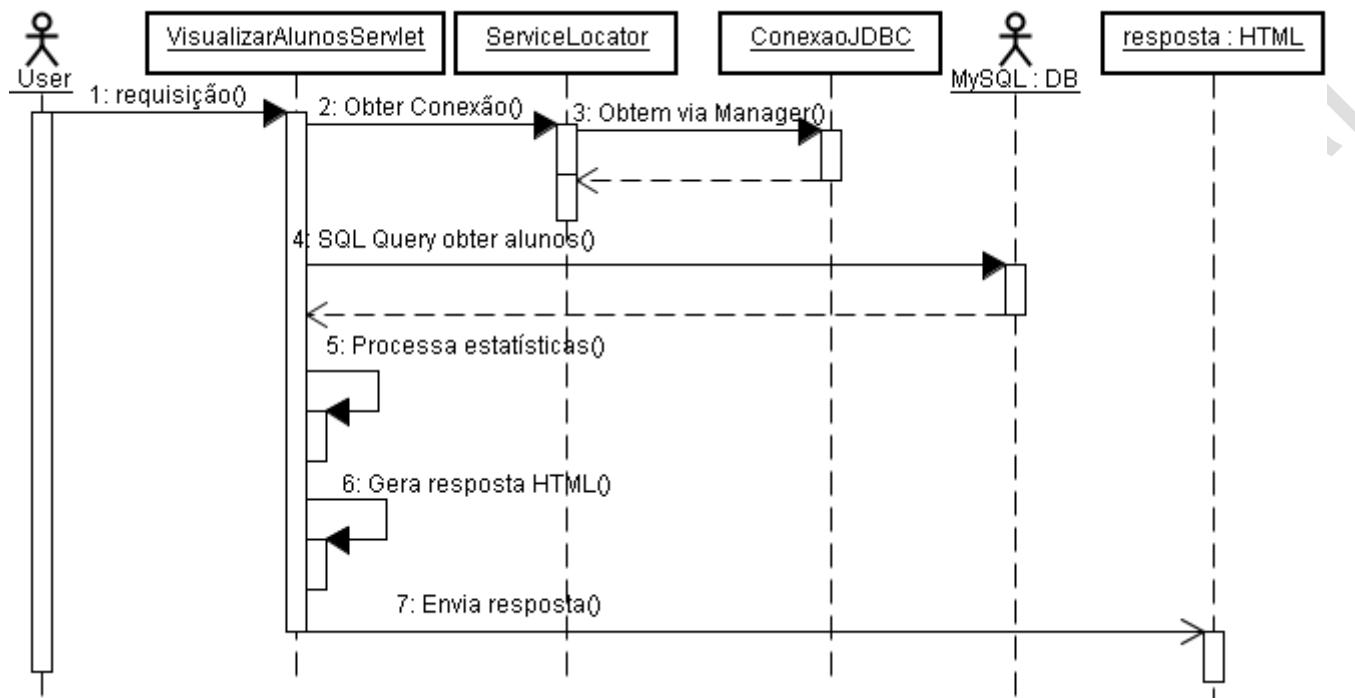
Conforme apresentado no diagrama a seguir, temos um Servlet que obtém uma conexão JDBC diretamente. Se tivermos 10 diferentes Servlets neste modelo, teremos 10 classes com código redundante de obtenção de conexão com banco de dados, que requer diversas configurações e tratamento de exceções.



Anotações

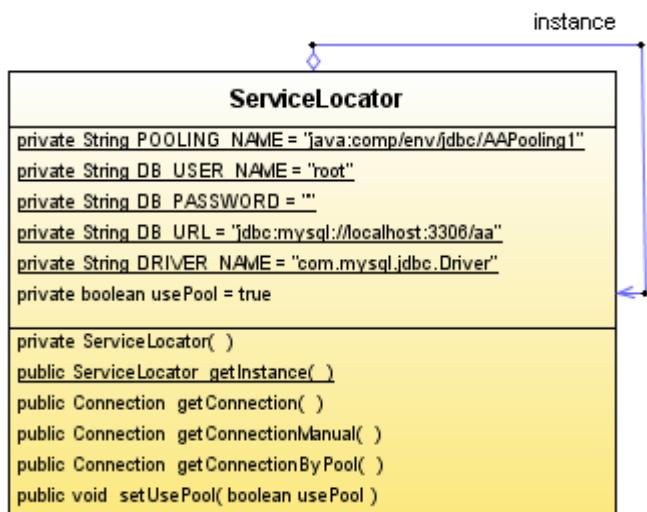
5.1.2 Aplicando o design-pattern

A estratégia será criar uma ou um pequeno conjunto de classes que centralizam as regras de obtenção de recursos, principalmente os recursos externos como conexões:



O Service Locator é freqüentemente desenvolvido como um Singleton.

Exemplo: Diagrama de classe UML br.com.globalcode.idp.locator.ServiceLocator



Anotações

Exemplo: ServiceLocator.java

```
1 package br.com.globalcode.idp.locator;
2
3 import javax.naming.*;
4 import javax.sql.*;
5 import java.sql.*;
6 import br.com.globalcode.idp.exception.GlobalcodeException;
7
8 public class ServiceLocator {
9
10    private final static String POOLING_NAME = "java:comp/env/jdbc/AAPooling1";
11    private final static String DB_USER_NAME = "root";
12    private final static String DB_PASSWORD = "";
13    private final static String DB_URL = "jdbc:mysql://localhost:3306/aa";
14    private final static String DRIVER_NAME = "com.mysql.jdbc.Driver";
15    private static InitialContext initCtx = null;
16    private static ServiceLocator instance = new ServiceLocator();
17    private boolean usePool = true;
18
19    private ServiceLocator() {
20    }
21
22    public static ServiceLocator getInstance() {
23        return instance;
24    }
25
26    public Connection getConnection() throws GlobalcodeException {
27        if (usePool) {
28            return getConnectionByPool();
29        } else {
30            return getConnectionManual();
31        }
32    }
33
34    public Connection getConnectionManual() throws GlobalcodeException {
35        Connection conn = null;
36        try {
37            Class.forName(DRIVER_NAME);
38            conn = DriverManager.getConnection(DB_URL, DB_USER_NAME,
39                                            DB_PASSWORD);
40        } catch (Exception e) {
41            throw new GlobalcodeException(
42                    "Erro ao obter conexao via DriverManager: "
43                    + e.getMessage(), e);
44        }
45        return conn;
46    }
47    public Connection getConnectionByPool() throws GlobalcodeException {
48        Connection conn = null;
49        DataSource ds = null;
50        try {
51            if (initCtx == null) {
52                initCtx = new InitialContext();
53            }
54            -----ds = (DataSource) initCtx.lookup(POOLING_NAME);-----
55            conn = ds.getConnection();
56        } catch (Exception e) {
57            throw new GlobalcodeException("Erro ao obter conexao via JNDI: "
58                                         + POOLING_NAME, e);
59        }
60    }
61    return conn;
62}
63
64    public void setUsePool(boolean usePool) {
65        this.usePool = usePool;
66    }
67}
```

Para uso não comercial

Anotações

5.2 Data Access Object

Em diversos aplicativos encontramos códigos de acesso a dados estão espalhados em várias classes, dificultando a legibilidade, manutenção e mudanças no modelo de dados, bem como, a troca de fornecedor de dados.

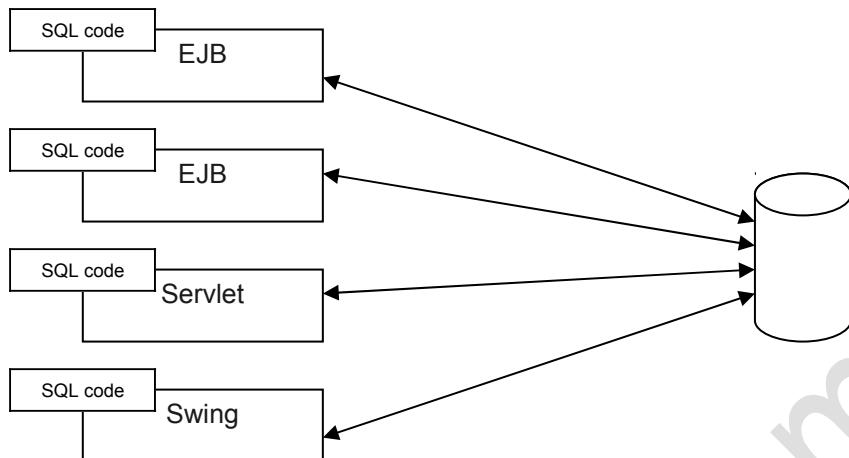
Data Access Object propõe a criação de um ponto central em um pequeno conjunto de classes que são responsáveis por acessar os dados da aplicação.

Aplicações práticas: quando desenvolvemos uma solução com o formato de produto e esta solução será instalada em diversos ambientes, torna-se necessário flexibilizar a aplicação de forma que ela possa ser configurada para trabalhar com múltiplos sistemas de gerenciamento de banco de dados (MySQL, MS-SQL, Sybase, Oracle e outros mais).

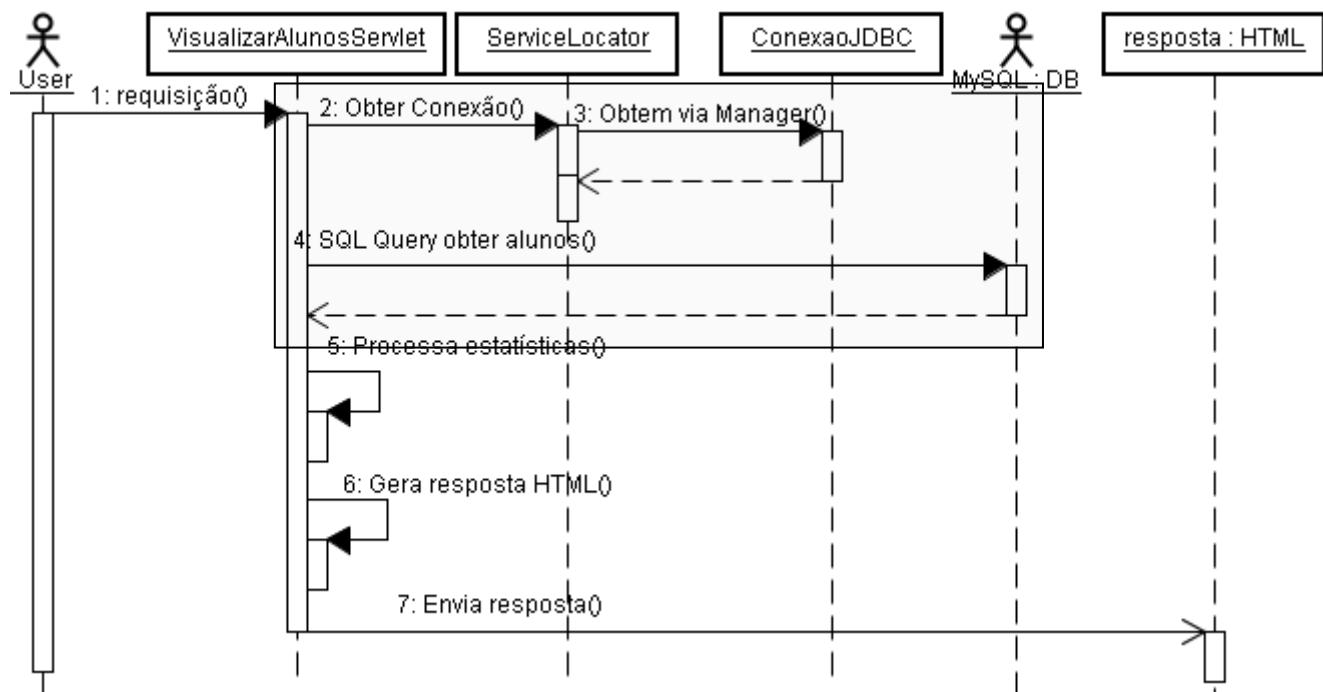
Anotações

5.2.1 Anti-pattern

Nesta imagem temos a idéia de vários componentes acessando uma base de dados:

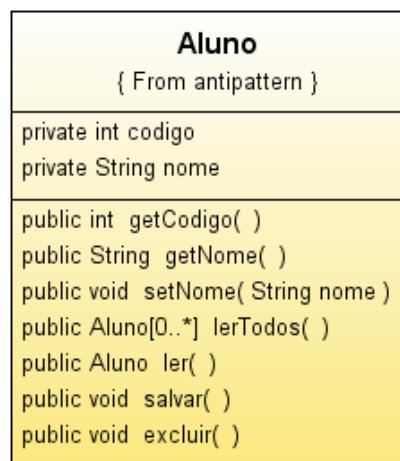


O diagrama de seqüência anterior ilustra o vínculo existente entre um objeto e a comunicação com o banco de dados:



Anotações

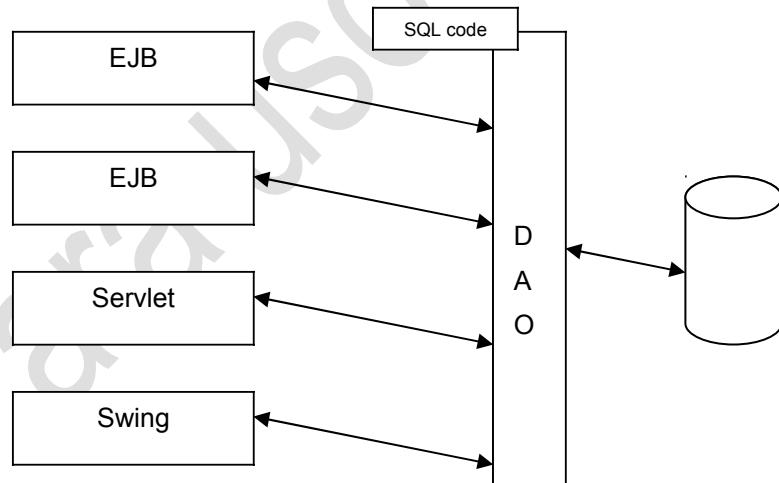
Uma prática errada comum e que contraria a aplicação deste pattern, é vincular código de persistência direto na entidade. Vejamos o diagrama de classes a seguir representando uma suposta entidade aluno:



Esta abstração de uma entidade está incorreta! A responsabilidade da classe Aluno é representar a entidade, seus dados, validações e eventuais processamentos de lógica de negócios. Atribuir a responsabilidade de persistência para a própria entidade prejudica a manutenção e reuso do aplicativo, além de gerar vínculos indesejados entre bibliotecas.

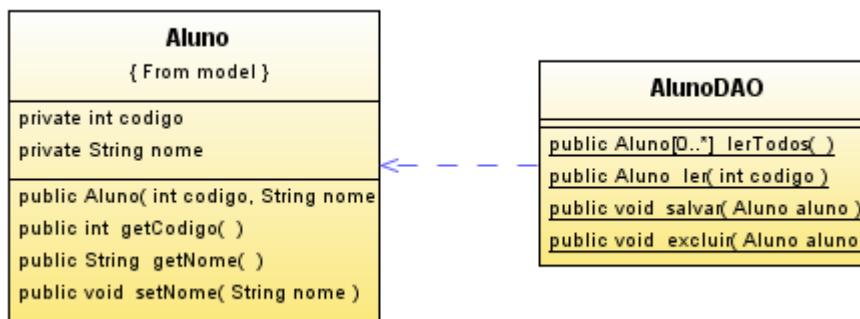
5.2.2 Aplicando o design pattern

A idéia é centralizarmos o acesso ao banco de dados em objetos que terão exclusivamente esta responsabilidade:

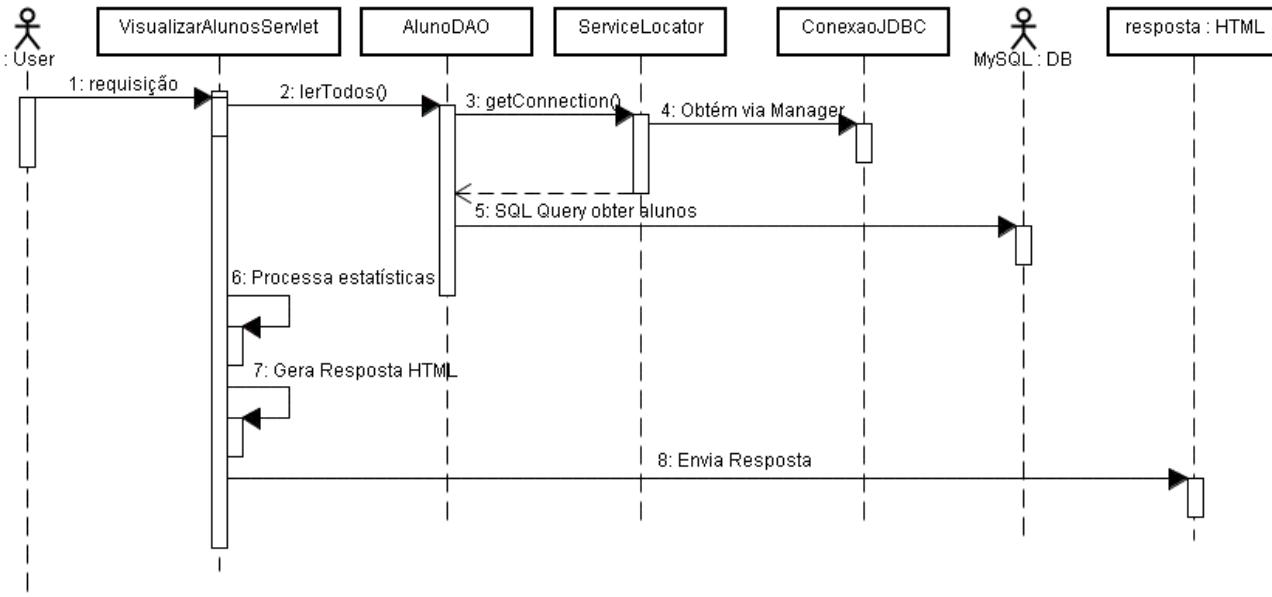


Vejamos a seguir a forma correta de escrever o código para tal situação, aplicando Data Access Object design-pattern:

Anotações



O diagrama de seqüência ficará da seguinte forma:



Quantas classes são necessárias visando comunicação com o sistema de persistência? Algumas abordagens nos levam a crer que a solução vai contar com um DAO para cada entidade; isto é válido, desde que, não haja excessos que ocasionem granularização do que não é necessário. A necessidade de modularização irá responder com precisão à pergunta inicial.

Se, por exemplo, pensarmos em um DAO para armazenar um Pedido de Compra, o banco de dados muito provavelmente vai normalizar os dados e teremos uma tabela para representar o Pedido e outra para representar os itens do Pedido.

Em Java podemos pensar na mesma modelagem: uma classe `ItemDoPedido` e outra `Pedido`, com um array de objetos `ItemDoPedido`. Porém, pelo fato de definirmos que não precisamos do `ItemDoPedido` desacoplado do `Pedido`, torna-se possível desenvolver um DAO para manipular as duas tabelas que trabalham a favor da entidade `Pedido`.

Anotações

Podemos considerar também as seguintes técnicas para o desenvolvimento de DAO's:

- Utilização do pattern Singleton;
- Definir a interface separada da classe DAO;
- Utilização de Factory de DAO's;
- Utilização de arquivos properties para armazenar o código SQL;
- Annotation para código SQL com XDoclet;
- DAO tende a ficar volumoso, portanto, fique atento e, se necessário, combine outros patterns para evitar código extenso;
- Evite utilizar linguagem SQL com instruções complexas e efetue processamento de dados no Java;
- Mesmo utilizando um framework de persistência, utilize DAO centralizando as responsabilidades de comunicação com o framework no DAO.

Anotações

5.3 Transfer Object / Value Object

Originalmente Transfer Object tinha a intenção de reduzir o tráfego na rede quando por exemplo, um servidor Web precisava acessar dados diversos providos por um EJB, ex. Cliente e Pedido. Poderíamos fazer uma chamada para obter os dados do Cliente e outra chamada para se obter os dados do Pedido. Com isso teríamos duas travessias na rede para poder compor os dois objetos que necessitamos na camada Web.

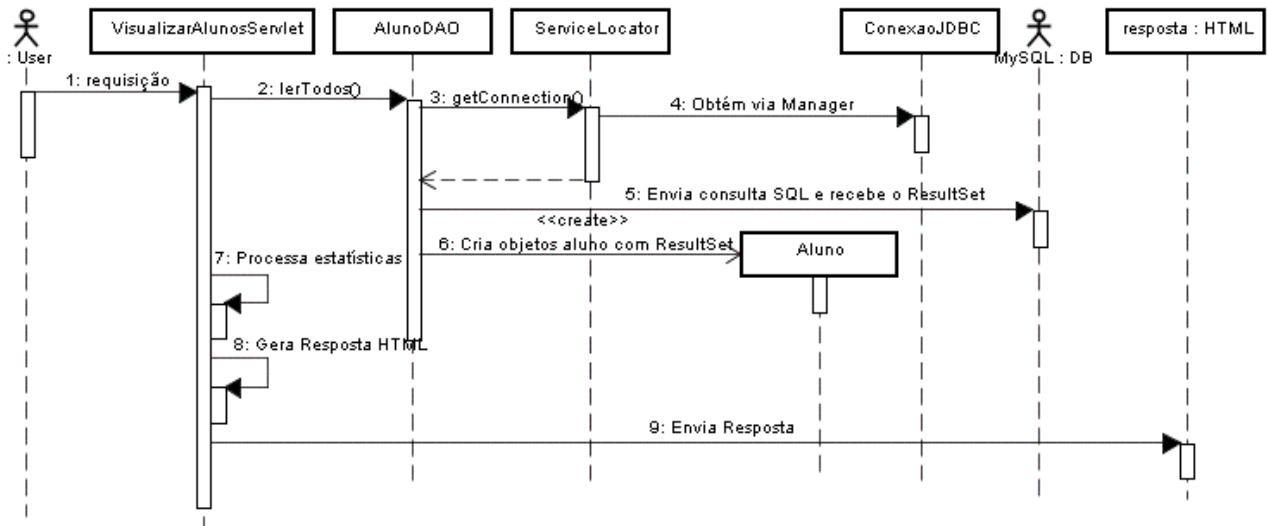
A proposta do Transfer Object é que você crie um único objeto que terá o Cliente e o Pedido e em uma única chamada receba todos os dados necessários, portanto é uma técnica para reduzir a granularidade das chamadas na rede.

Porém, este padrão foi concebido com a idéia que a maioria adotaria EJBs. Como não foi o caso e existe um grande número de aplicativos Web se comunicando com banco de dados diretamente, uma aplicação diferente vem sendo adotada. Em muitas ocasiões quando trabalhamos com banco de dados precisamos ler uma tabela que contém diversas colunas e utilizamos um Resultset que é o Iterator JDBC para percorrer resultados de uma consulta. Em muitas tecnologias é comum encontrar páginas ou componentes que trabalham diretamente com o objeto que representa o conjunto de resultados da consulta (Recordset / Resultset). Esta prática parece ser produtiva no primeiro instante mas gera um altíssimo acoplamento entre user-interface Vs. Banco de dados além de promover o desenvolvimento desorganizado de código SQL embutido no código Java.

O correto portanto é a criação de um objeto que contenha todos os dados que você vai precisar usar do banco de dados e fechar a conexão e o conjunto de resultados imediatamente após a transferência dos dados do Resultset para o objeto. Apesar de não ser a documentação original de Transfer Object, no conhecimento popular esta técnica de transferência de dados da camada do RDBMS para os objetos Web vem sendo chamada de Transfer ou Value Object.

Anotações

Ao aplicarmos o padrão teremos um objeto que representará o modelo de dados:



Agora podemos observar que o DAO irá enviar a consulta no passo 5, em seguida vai percorrer a consulta criando objetos Aluno que representam os dados. Poderá posteriormente fechar a conexão com banco de dados e trabalhar somente com objetos Aluno.

Exemplo: AlunoDAO.java com utilização de Value Object Aluno

```
1 package br.com.globalcode.idp.dao;
2
3 import br.com.globalcode.idp.exception.GlobalcodeException;
4 import br.com.globalcode.idp.locator.ServiceLocator;
5 import br.com.globalcode.idp.model.Aluno;
6 import java.sql.*;
7 import java.util.*;
8
9 public class AlunoDAO {
10
11     private static String SQL_INSERIR_ALUNO =
12         "insert into alunos ( nome ) values ( ? )";
13     private static String SQL_ATUALIZAR_ALUNO =
14         "update alunos set nome = ? where id = ?";
15     private static String SQL_EXCLUIR_ALUNO =
16         "delete from alunos where id = ?";
17     private static String SQL_LISTAR_TODOS =
18         "select * from alunos";
19     private static String SQL_RECUPERAR_ALUNO =
20         "select * from alunos where id = ?";
21 }
```

Anotações

```

22  public static Aluno[] lerTodos() throws DAOException {
23      Connection conn = null;
24      PreparedStatement st = null;
25      ResultSet rs = null;
26      List<Aluno> alunos = new ArrayList<Aluno>();
27      Aluno aluno = null;
28      try {
29          conn = getConnection();
30          st = conn.prepareStatement(SQL_LISTAR_TODOS);
31          rs = st.executeQuery();
32          while (rs.next()) {
33              aluno = new Aluno(rs.getInt("codigo"), rs.getString("nome"));
34              alunos.add(aluno);
35          }
36      } catch (Exception ex) {
37          throw new DAOException(ex);
38      } finally {
39          closeResources(conn, st, rs);
40      }
41      return alunos.toArray(new Aluno[0]);
42  }
43
44  public static Aluno ler(int codigo) throws DAOException {
45      Connection conn = null;
46      PreparedStatement st = null;
47      ResultSet rs = null;
48      Aluno aluno = null;
49      try {
50          conn = getConnection();
51          st = conn.prepareStatement(SQL_RECUPERAR_ALUNO);
52          st.setInt(1, codigo);
53          rs = st.executeQuery();
54          if (rs.next()) {
55              aluno = new Aluno(codigo, rs.getString("nome"));
56          }
57      } catch (Exception ex) {
58          throw new DAOException(ex);
59      } finally {
60          closeResources(conn, st, rs);
61      }
62      return aluno;
63  }
64
65
66  public static void salvar(Aluno aluno) throws DAOException {
67      Connection conn = null;
68      PreparedStatement st = null;
69      try {
70          conn = getConnection();
71          if (aluno.getCodigo() == 0) {
72              st = conn.prepareStatement(SQL_INSERIR_ALUNO);

```

Anotações

```

173     } else {
174         st = conn.prepareStatement(SQL_ATUALIZAR_ALUNO);
175         st.setInt(2, aluno.getCodigo());
176     }
177     st.setString(1, aluno.getNome());
178     st.execute();
179 } catch (Exception ex) {
180     throw new DAOException(ex);
181 } finally {
182     closeResources(conn, st, null);
183 }
184 }
185
186 public static void excluir(Aluno aluno) throws DAOException {
187     Connection conn = null;
188     PreparedStatement st = null;
189     try {
190         conn = getConnection();
191         st = conn.prepareStatement(SQL_EXCLUIR_ALUNO);
192         st.setInt(1, aluno.getCodigo());
193         st.execute();
194     } catch (Exception ex) {
195         throw new DAOException(ex);
196     } finally {
197         closeResources(conn, st, null);
198     }
199 }
200
201 private static Connection getConnection() throws GlobalcodeException {
202     return ServiceLocator.getInstance().getConnection();
203 }
204
205 private static void closeResources(Connection conn, Statement st, ResultSet
rs) {
206     try {
207         if (rs != null) {
208             rs.close();
209         }
210         if (st != null) {
211             st.close();
212         }
213         if (conn != null) {
214             conn.close();
215         }
216     } catch (SQLException e) {
217     }
218 }
219 }
```

```

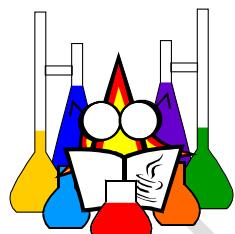
## Anotações

---

## 5.4 Laboratório 9

Objetivo:

Praticar os design patterns Service Locator e DAO.



**LABORATÓRIO**

**Tabela de atividades:**

| Atividade                                                                                                                                                                                         | OK |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1. Faça o download do arquivo lab9.zip na URL indicada pelo instrutor(a).                                                                                                                         |    |
| 2. Descompacte o arquivo em seu diretório de trabalho.                                                                                                                                            |    |
| 3. Abra o projeto no NetBeans e rode o aplicativo pressionando F6                                                                                                                                 |    |
| 4. Este é um aplicativo de gerenciamento de escolas. Navegue no aplicativo e certifique-se que tudo está funcionando a contento.                                                                  |    |
| 5. O aplicativo deverá ser refatorado utilizando os patterns Service Locator e DAO. Não é necessário fazer o refactoring para todas as entidades. Faça apenas para a entidade Aluno (Membership). |    |

Anotações

---



---



---



---

## 5.5 Front Controller

Nos cenários atuais de desenvolvimento, principalmente Web, uma solução representa um conjunto de funcionalidades que podem ser expostas no servidor de diferentes formas. Ao desenvolvermos componentes Servlets e JSPs para atender aos requisitos necessários do projeto, cada Servlet e cada JSP será um potencial ponto de acesso ao seu aplicativo.

Front Controller propõe a centralização das regras de atendimento às requisições dos usuários com a intenção de promover maior facilidade no controle de requisições que poderão ser complexas e contínuas e poderão necessitar de diversos serviços comuns. A abordagem tradicional é a implementação de um Front Controller como um Servlet mapeado para um nome ou coringa, como tipicamente fazemos com Struts configurando seu Front Controller como \*.do.

Quando não trabalhamos com centralização de requisições, enfrentam-se os seguintes problemas:

- Para cada tipo de visualização de dados temos um controlador induzindo a duplicata de código;
- Navegação nas views é de responsabilidade das próprias views;
- Dificuldade de manutenção e compreensão da solução.

Anotações

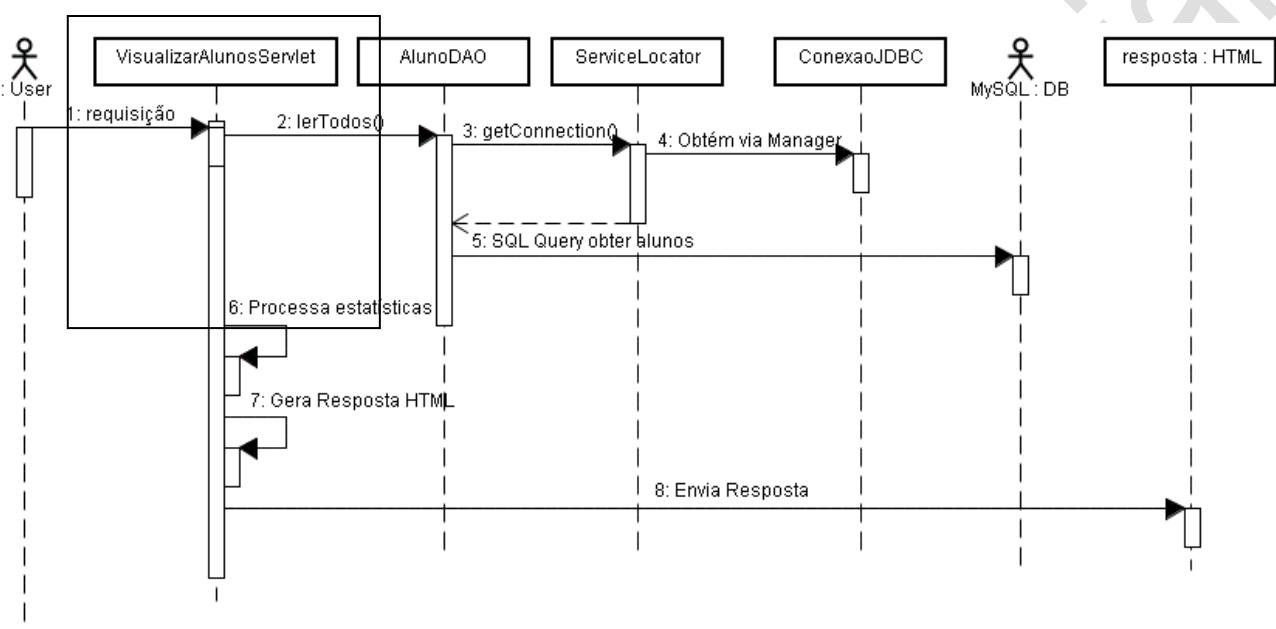
---

---

---

### 5.5.1 Anti-pattern

O anti-pattern de Front Controller é qualquer modelo que permita a user-interface se comunicar com objetos que representam efetivamente a ação do aplicativo. Conforme diagrama de seqüência apresentado, podemos notar que o usuário terá acesso direto ao componente que responde à funcionalidade. Com isso, cada funcionalidade representará um potencial ponto de entrada no sistema.



Ao aplicarmos Front Controller todas as requisições serão centralizadas em uma classe que poderá prover um comportamento comum no tratamento da requisição e posteriormente vai encaminhar para um objeto especializado na solicitação em questão.

Anotações

---



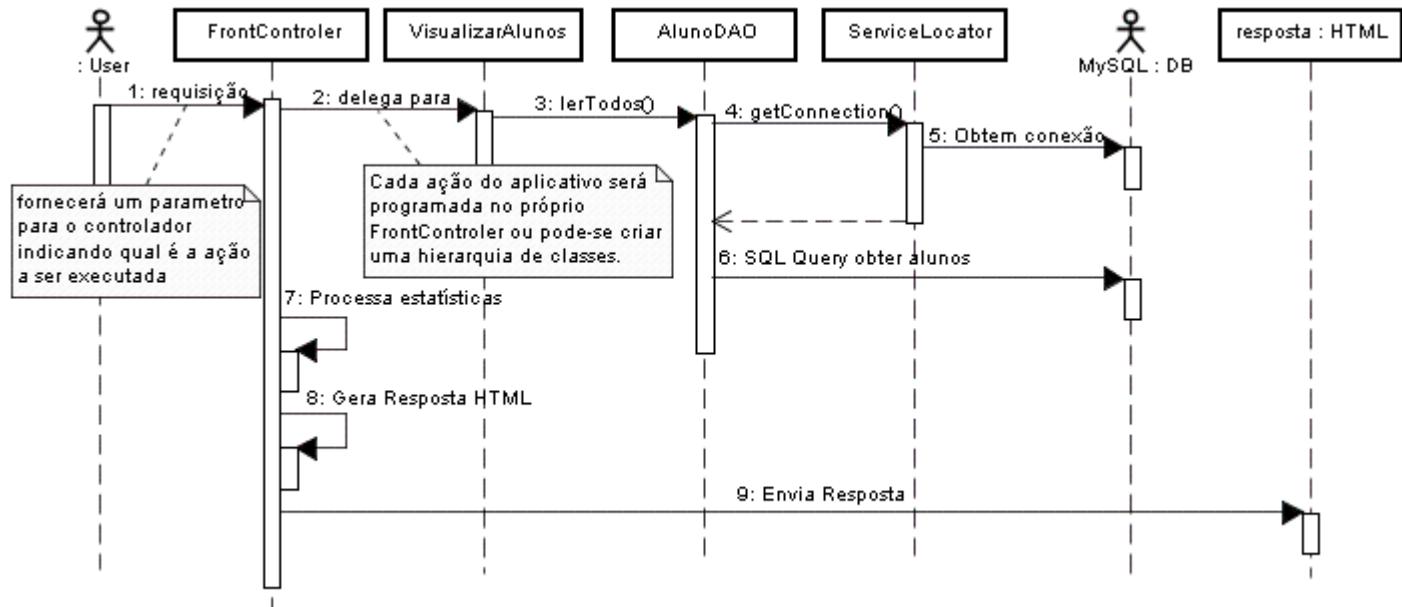
---



---

## 5.5.2 Aplicando o design pattern

Em soluções Web Java EE um Front Controller é geralmente implementado por um Servlet que será de alguma forma parametrizado para saber qual ação ele deverá executar. Uma vez interpretado o parâmetro, o Front Controller vai despachar a requisição para um objeto que representa a funcionalidade:



Anotações

---

---

---

**Exemplo: FrontController.java com utilização de Command e Factory**

```

1 package br.com.globalcode.idp.controller;
2
3 import br.com.globalcode.idp.web.command.CommandFactory;
4 import br.com.globalcode.idp.web.command.WebCommand;
5 import java.io.*;
6 import java.net.*;
7 import javax.servlet.*;
8 import javax.servlet.http.*;
9
10 public class FrontController extends HttpServlet {
11
12 protected void processRequest(HttpServletRequest request,
13 HttpServletResponse response)
14 throws ServletException, IOException {
15 WebCommand aCommand = null;
16 String command = request.getParameter("command");
17 try {
18 aCommand = CommandFactory.createWebCommand(command);
19 aCommand.doAction(request);
20 } catch(Exception e) {
21 throw new ServletException(e);
22 }
23 String destino = (String) request.getAttribute("destino");
24 String navegacao = (String) request.getAttribute("tipoNavegacao");
25 if (destino != null && !destino.equals("")) {
26 if(navegacao.equalsIgnoreCase("forward")) {
27 RequestDispatcher rd = request.getRequestDispatcher(destino);
28 rd.forward(request, response);
29 } else {
30 response.sendRedirect(request.getContextPath()+destino);
31 }
32 } else{
33 response.sendRedirect(request.getContextPath()+"/index.jsp");
34 }
35 }
36
37 @Override
38 protected void doGet(HttpServletRequest request,
39 HttpServletResponse response)
40 throws ServletException, IOException {
41 processRequest(request, response);
42 }
43
44 @Override
45 protected void doPost(HttpServletRequest request,
46 HttpServletResponse response)
47 throws ServletException, IOException {
48 processRequest(request, response);
49 }
50 }
51

```

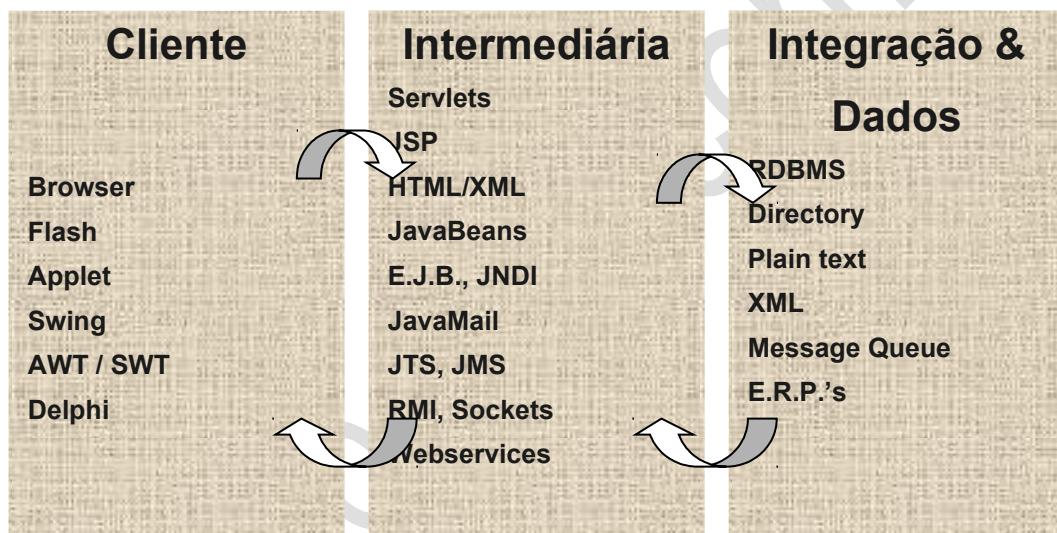
**Anotações**

## 5.6 Model-view-controller

Model View and Controller é um padrão de arquitetura desenvolvido pela comunidade do Smalltalk e amplamente adotado na plataforma Java EE. M.V.C. consta nos Blueprints da Sun, ou seja, vem sendo recomendado pela comunidade Java EE.

Segundo um artigo de Smalltalk, “M.V.C. é um modelo elegante e simples de desenvolver uma solução, mas requer algumas explicações, pois foge do sistema tradicional de construirmos softwares.”.

Sabemos que a melhor maneira de criarmos um software eficientemente modular, reutilizável e flexível é dividindo-o em pequenas partes que, por sua vez, serão divididas em camadas.



Anotações

---

---

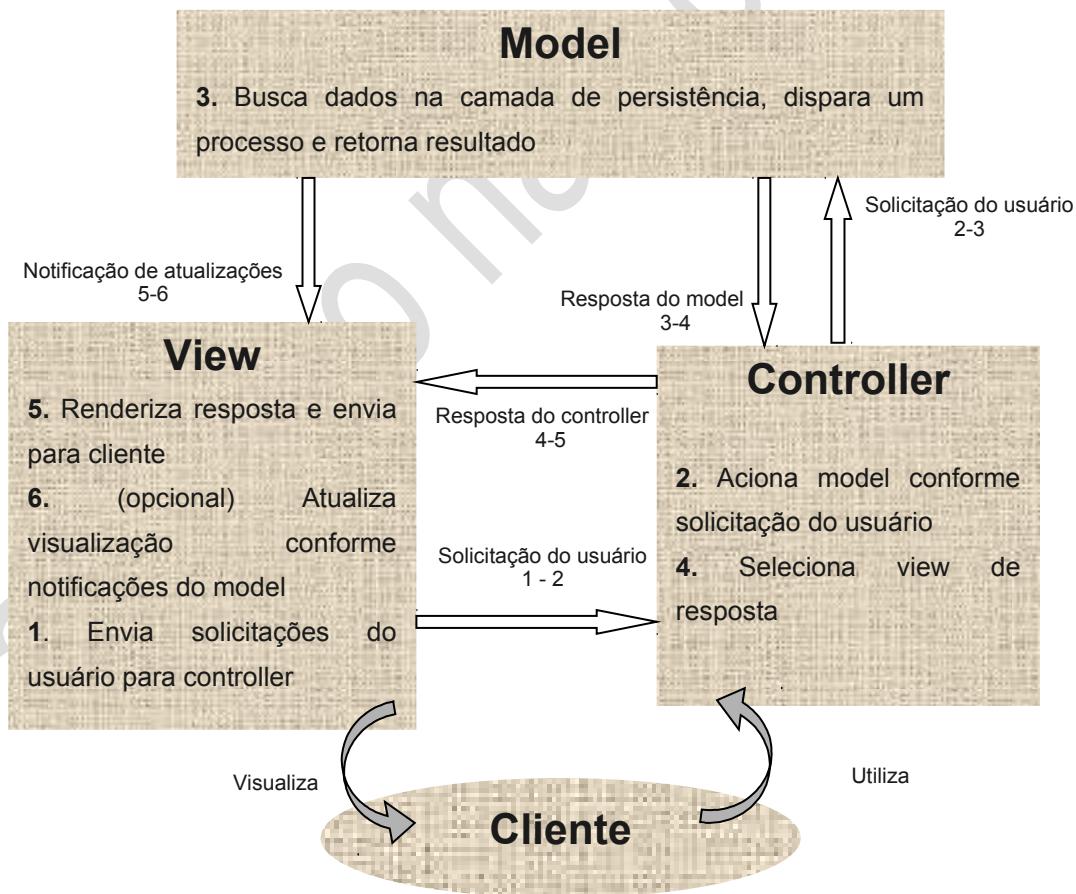
---

## 5.6.1 A divisão da camada intermediária

A tendência atual é a de que a camada intermediária concentre a maior parte do código da solução. O modelo M.V.C. propõe uma maneira de modularizar a camada intermediária e de clientes de forma simples, flexível e inteligente.

Através da utilização deste modelo separamos o software em três principais partes:

- Modelo: representa a lógica da aplicação através de entidades e classes de processos. O modelo pode notificar a visualização quando ele for modificado.
- Controle: responsável por expor a aplicação para o usuário, assim como, controlar, autorizar e efetuar logging de acesso. Age como uma “ponte” entre o usuário e o modelo, isto é, a cada requisição aciona o modelo, recepciona uma resposta e solicita a um view que apresente o resultado.
- Visualização: todo componente de apresentação de resultados e geração de diálogo com o usuário faz parte desta camada. Com o modelo M.V.C. podemos ter muitas visualizações para um mesmo resultado.



Anotações

---

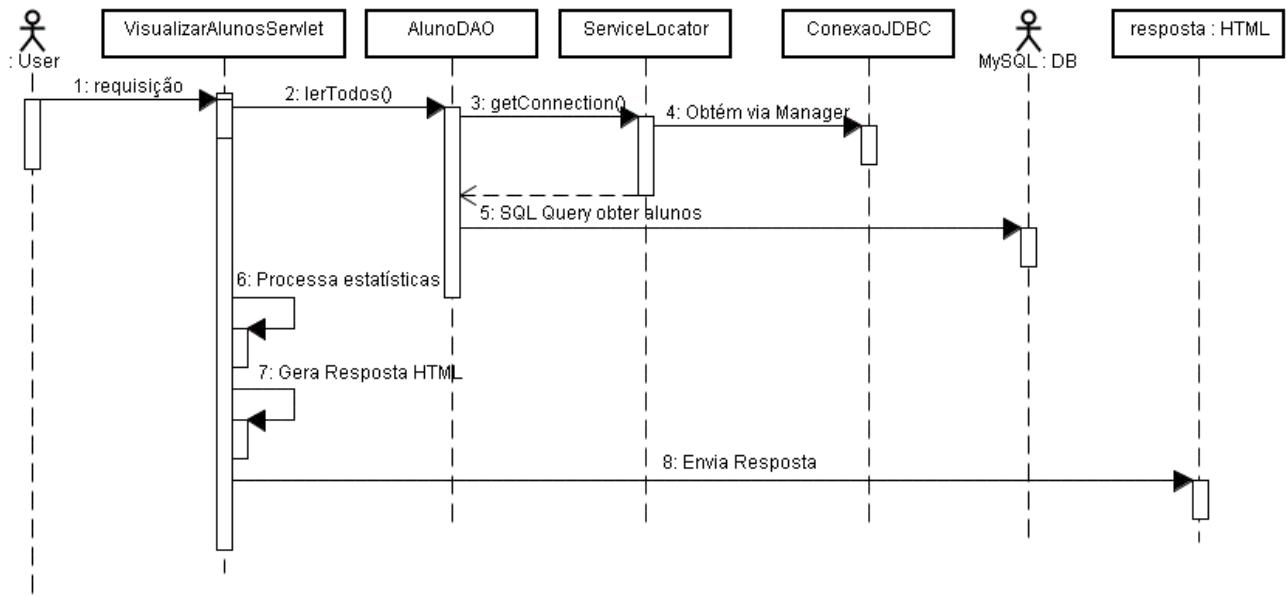


---



---

## 5.6.2 Anti-pattern



Analisando este diagrama de seqüência podemos concluir que **VisualizarAlunosServlet** está sobrecarregado de responsabilidades:

1. Recepção de requisição do usuário (eventualmente faria validações);
2. Obtém dados do banco de dados;
3. Processa estatísticas
4. Renderiza uma página HTML com a resposta para o usuário;
5. Envia resposta;

Portanto este componente está fazendo papel de Model, View e de Controller do aplicativo. Ao refatorar esta arquitetura aplicando MVC todas essas responsabilidades ficarão divididas facilitando reuso, extensão e manutenção do aplicativo.

Anotações

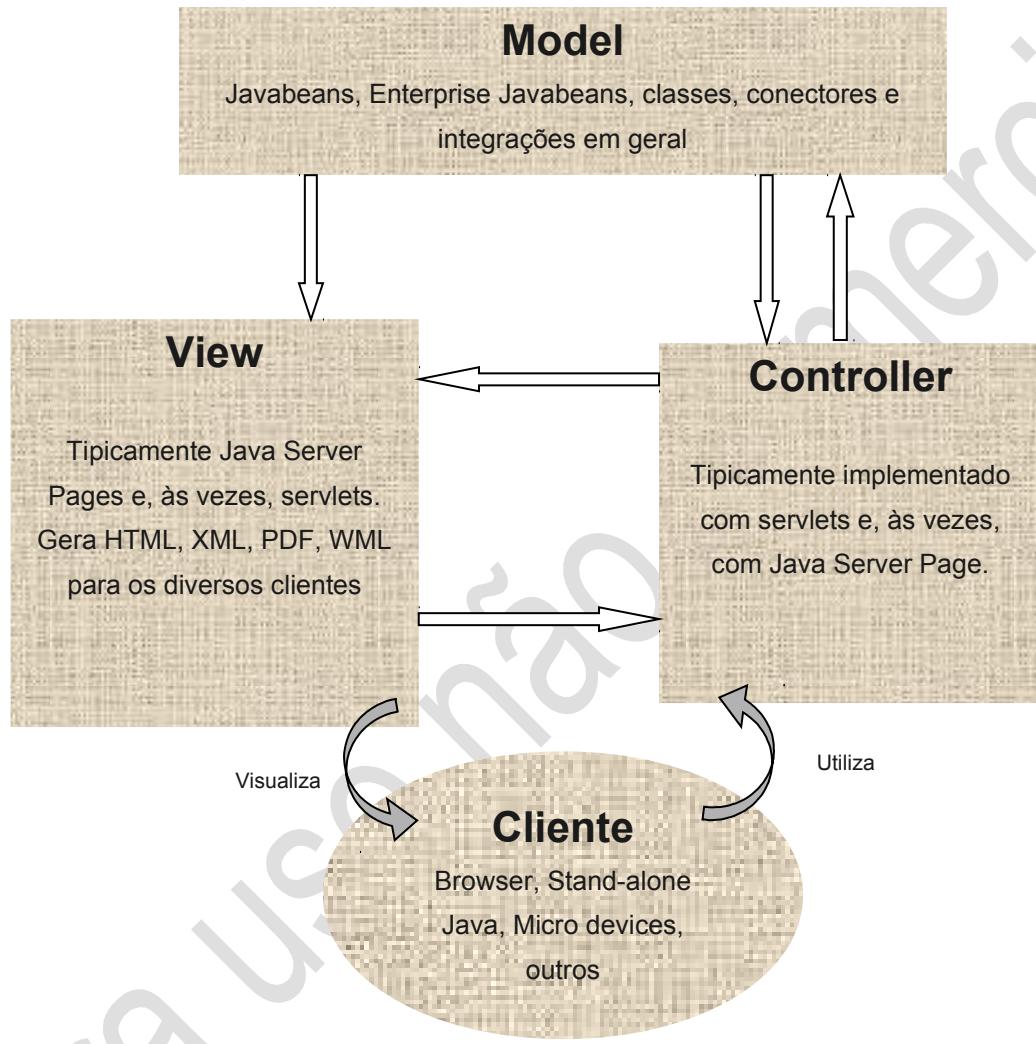
---

---

---

### 5.6.3 Aplicando o design pattern

A implantação do M.V.C. com Java Enterprise Edition geralmente é feita da seguinte forma:



\*Pelo fato do protocolo http ser um protocolo sem conexão, em soluções Web conseguimos receber atualizações no Browser somente com a utilização de um plug-in como Java Applet ou Macromedia Flash

Anotações

---

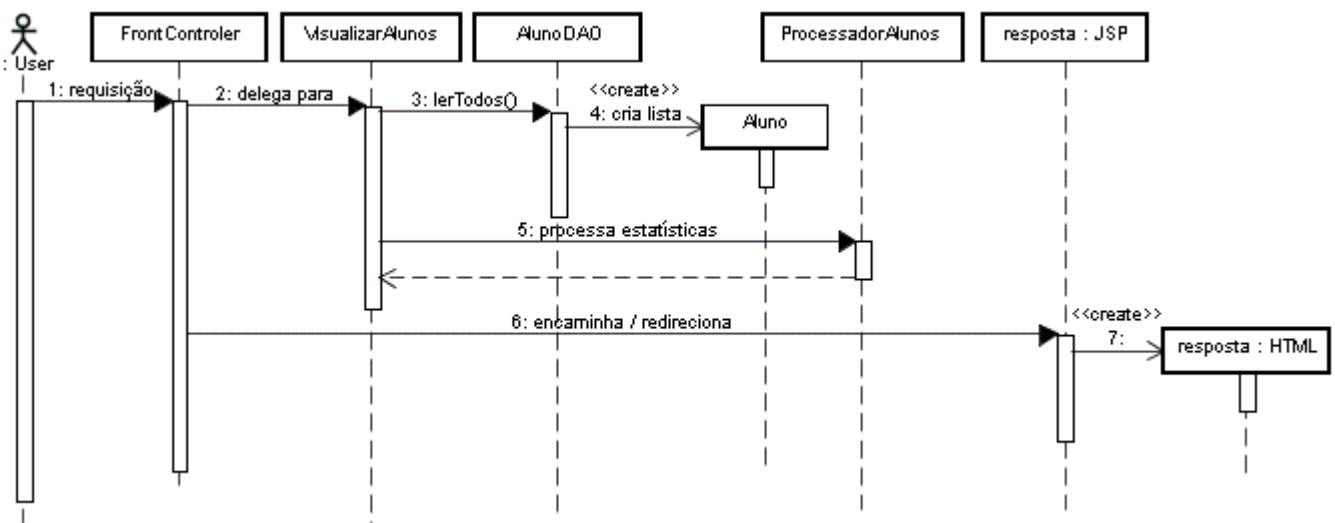


---



---

A sequência da arquitetura refatorada ficará da seguinte forma:



As responsabilidades ficaram divididas da seguinte forma:

- Controller: componente **FrontController** será a recepção do aplicativo e poderá efetuar validações, segurança e controle de navegação nas páginas / telas;
- Model: os objetos **Aluno** e também a rotina de processamento de estatísticas faz parte do model. A classe Data Access Object também faz parte, mas costumamos dizer que ela representa a camada de persistência do model e não o model em si.
- View: agora no lugar de um servlet gerar o HTML, temos o processo de encaminhamento (request forward) para um JSP que faz somente o papel de View do aplicativo. Podemos ter diferentes JSPs processando diferentes visualizações para o mesmo model.

## Anotações

---

## 5.7 Composite View

Telas e páginas de sistemas Web tendem a ter uma complexidade em função dos diferentes tipos de objetos que são normalmente apresentados:

- Menus
- Logotipo da empresa
- Rodapé
- Área de dados
- Árvore de objetos
- Imagens diversas

Alguns destes elementos muitas vezes são apresentados em mais do que uma página, como por exemplo os menus. Composite View tem a intenção de evitar complexidade nas interfaces com usuário promovendo reuso de componentes / fragmentos de tela.

A maneira mais simples e primitiva de se fazer Composite View é através de mecanismos de include de JSPs ou utilizando composições de objetos JPanel Swing. Atualmente no desenvolvimento Web, contamos com alguns frameworks open-source que trabalham com Composite View oferecendo diversos recursos, incluindo um sistema parecido com herança de páginas. Os mais populares frameworks de Composite View Web são:

- Tiles: popularmente utilizado em conjunto com Struts;
- Facelets: semelhante ao Tiles porém focado na tecnologia JavaServer Faces;

A aplicação de Composite View pode variar bastante dependendo do recurso que utilize. Se você já utiliza um dos frameworks acima, então já é usuário deste padrão.

---

Anotações

---

---

---

## Exemplo: index.jsp com utilização de inclusão dinâmica <jsp:include>

---

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@page errorPage="/erros/generico.jsp"%>
3
4 <html>
5 <head>
6 <title>Exemplo de Composite View</title>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 </head>
9 <body>
10 <table>
11 <tr>
12 <td><jsp:include page="top.jsp" flush="true" /></td>
13 </tr>
14 <tr><td>
15 <table>
16 <tr>
17 <td>
18 <jsp:include page="menu.jsp" flush="true"/>
19 </td>
20 <td>
21 <jsp:include page="${param.page}" flush="true"/>
22 </td>
23 </tr>
24 </table>
25 </td></tr>
26 <tr>
27 <td><jsp:include page="bottom.jsp" flush="true"/></td>
28 </tr>
29 </table>
30 </body>
31 </html>
32
```

## Anotações

---

---

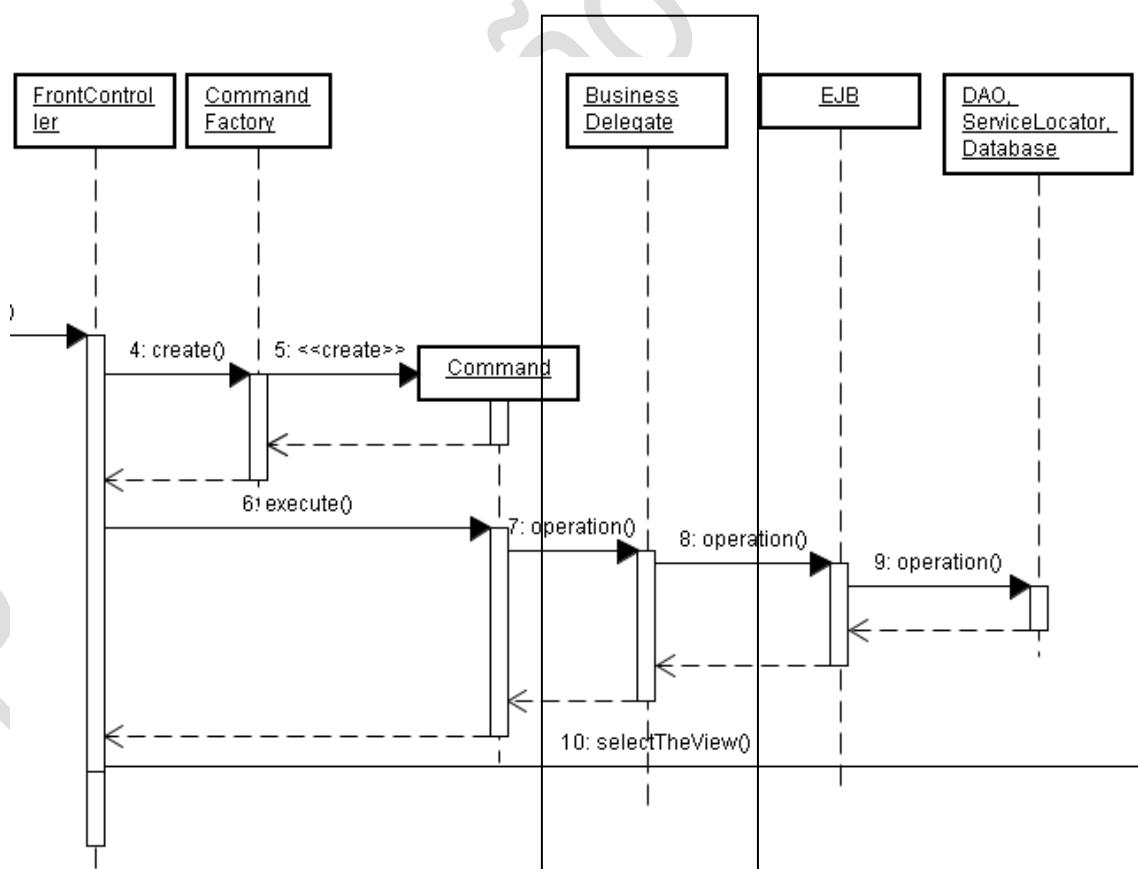
---

## 5.8 Business Delegate

O **Business Delegate** é um padrão J2EE cuja aplicabilidade original era voltada a componentes EJB, onde a chamada à interface Home e o acionamento dos métodos do EJB são encapsulados e tratados pelos métodos de uma classe Java tradicional. Se sua aplicação não faz uso de objetos distribuídos EJB, não tem importância. O objetivo é que a idéia do processamento das regras de negócio seja feita utilizando classes especializadas, também chamadas Helpers.

É muito comum que um processo de negócio seja composto por pequenas partes que formam o todo. Cada uma delas pode ter um nível de complexidade menor ou maior em relação às outras. Escrever uma regra de negócio em apenas uma classe, ou ainda, em um só método, gera problemas de não-reusabilidade de parte do processo (que poderia ser reaproveitado em outro cenário de nossa aplicação), além de dificultar a manutenção, provocando a geração de código redundante.

Pensar em fragmentos de um processo deixa nossa aplicação mais extensível e de fácil entendimento. Então, criamos, a partir daí, uma classe *Helper* que aglutina esses fragmentos formando o todo. Essa é uma excelente prática de desenvolvimento. Geralmente, Helpers são muito bem empregadas em classes que implementam o padrão Command.



Anotações

---



---



---

A camada demarcada representa a implementação do Business Delegate design-pattern, sem o qual, o Command acessaria diretamente a camada de negócio (neste caso, um EJB). Ele representa a forma como o client fará a interface com o EJB.

### **Exemplo: EditarCommand.java utilizando Business Delegate**

---

```
1 package br.com.globalcode.idp.web.command.aluno;
2
3 import br.com.globalcode.idp.business.delegate.AlunoDelegate;
4 import br.com.globalcode.idp.web.command.*;
5 import br.com.globalcode.idp.model.Aluno;
6 import javax.servlet.http.HttpServletRequest;
7
8 public class EditarCommand implements WebCommand{
9
10 public void doAction(HttpServletRequest request)
11 throws WebCommandException {
12
13 String id = request.getParameter("idAluno");
14 Aluno aluno = null;
15 if(id != null && !id.equals("")) {
16 try {
17 aluno = AlunoDelegate.getInstance().getAluno(new Integer(id));
18 } catch (GlobalcodeException ex) {
19 throw new WebCommandException("[Aluno.EditarCommand]"
20 + ex.getMessage(),ex);
21 }
22 } else {
23 aluno = new Aluno();
24 }
25
26 request.setAttribute("Aluno",aluno);
27 request.setAttribute("destino","/aluno/formDados.jsp");
28 request.setAttribute("tipoNavegacao","forward");
29 }
30 }
31
```

Neste exemplo a classe Command faz o acesso à camada de negócio através de um Business Delegate, sem conhecer os detalhes da implementação da lógica. Por exemplo, não sabemos como foi feita a obtenção do aluno, se através de um DAO, um EJB ou outras classes de negócio.

---

### **Anotações**

---

---

---

**Exemplo: AlunoDelegate.java utilizando EJB**

```

1 package br.com.globalcode.idp.business.delegate;
2
3 import br.com.globalcode.idp.locator.EJBLocator;
4 import br.com.globalcode.idp.exception.GlobalcodeException;
5 import br.com.globalcode.idp.model.Aluno;
6 import java.rmi.RemoteException;
7 import java.util.*;
8
9 public class AlunoDelegate {
10
11 private static AlunoDelegate instance = new AlunoDelegate();
12
13 private AlunoDelegate() {}
14
15 public static AlunoDelegate getInstance() {
16 return instance;
17 }
18
19 public List getAllAlunos() throws GlobalcodeException {
20 List<Aluno> alunos = new ArrayList<Aluno>();
21 try {
22 AlunoSessionRemote ejb = EJBLocator.getInstance().getAlunoSession();
23 alunos = ejb.getAllAlunos();
24 } catch(RemoteException e) {
25 throw new GlobalcodeException(e.getMessage(),e);
26 }
27 return alunos;
28 }
29
30 public Aluno getAluno(Integer id) throws GlobalcodeException {
31 Aluno aluno = null;
32 try {
33 AlunoSessionRemote ejb = EJBLocator.getInstance().getAlunoSession();
34 aluno = ejb.getAluno(id);
35 } catch(RemoteException e) {
36 throw new GlobalcodeException(e.getMessage(),e);
37 }
38 return aluno;
39 }
40
41 public void salvarAluno(Aluno aluno) throws GlobalcodeException {
42 try {
43 AlunoSessionRemote ejb = EJBLocator.getInstance().getAlunoSession();
44 ejb.salvarAluno(aluno);
45 } catch(RemoteException e) {
46 throw new GlobalcodeException(e.getMessage(),e);
47 }
48 }
49 }

```

**Anotações**

## Exemplo: AlunoDelegateDAO.java utilizando DAO

---

```
1 package br.com.globalcode.idp.business.delegate;
2
3 import br.com.globalcode.idp.dao.*;
4 import br.com.globalcode.idp.exception.GlobalcodeException;
5 import br.com.globalcode.idp.model.Aluno;
6 import java.util.*;
7
8 public class AlunoDelegateDAO {
9
10 private static AlunoDelegateDAO instance = new AlunoDelegateDAO();
11
12 private AlunoDelegateDAO() {}
13
14 public static AlunoDelegateDAO getInstance() {
15 return instance;
16 }
17
18 public List getAllAlunos() throws GlobalcodeException {
19 Aluno[] alunos = null;
20 try {
21 alunos = AlunoDAO.lerTodos();
22 } catch(DAOException e) {
23 throw new GlobalcodeException(e.getMessage(), e);
24 }
25 return Arrays.asList(alunos);
26 }
27
28 public Aluno getAluno(Integer id) throws GlobalcodeException {
29 Aluno aluno = null;
30 try {
31 aluno = AlunoDAO.ler(id);
32 } catch(DAOException e) {
33 throw new GlobalcodeException(e.getMessage(), e);
34 }
35 return aluno;
36 }
37
38 public void salvarAluno(Aluno aluno) throws GlobalcodeException {
39 try {
40 AlunoDAO.salvar(aluno);
41 } catch(DAOException e) {
42 throw new GlobalcodeException(e.getMessage(), e);
43 }
44 }
45 }
46
```

---

## Anotações

---

---

---

O uso do Business Delegate traz as seguintes consequências.

- Reduz o acoplamento entre a camada de apresentação e a camada de negócio, melhorando a manutenibilidade do sistema. Isto ocorre, pois as alterações na camada de negócio implicam em alterações apenas na classe do Business Delegate e não em diversas classes espalhadas pela camada de apresentação.
- Traduz exceções de infra-estrutura, como RemoteExceptions em exceções de negócio, isolando o cliente dos detalhes da implementação. Assim, é possível trocar uma interface remota para local sem nenhuma alteração no código cliente.
- Melhora a disponibilidade do aplicativo. No caso de falhas no acesso ao serviço o Business Delegate pode executar ações de recuperação como tentar novamente ou reconectar com outro serviço, sem que a falha seja percebida pelo cliente.
- Expõe uma interface uniforme e mais simples de acesso a camada de negócio.
- Melhora a performance através da utilização de cache.
- Introduz uma camada adicional, o que pode ser visto como aumento da complexidade. Porém, os benefícios obtidos facilmente superam este aumento.
- Oculta acessos remotos, permitindo transparência de localização ao cliente.

---

#### Anotações

---

---

---

## 5.9 Application Controller

Application Controller será uma classe responsável por orquestrar as chamadas dos usuários no aplicativo. O Application Controller poderá oferecer em conjunto com classes de apoio:

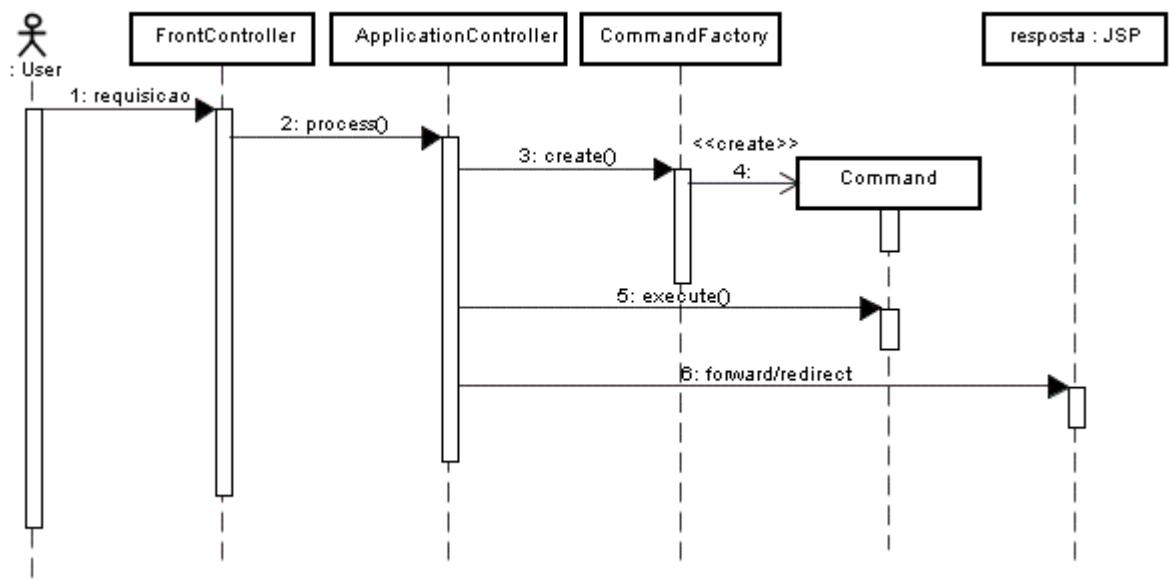
- Execução do comando / ação disparada pelo usuário;
- Controle de navegação e views;

Application Controller não terá tudo isso programado internamente mas será a classe que vai se comunicar com as demais classes (que terão uma porção de outros padrões aplicados) para executar o processo de atendimento de requisições como um todo.

Esta mesma lógica poderia estar programada dentro de um Front Controller. As principais vantagens em separar esta lógica são:

- provê uma maior modularidade ao deixar o gerenciamento das ações e navegação separada dos detalhes de protocolo específicos da recepção das requisições.
- permite o teste desta lógica fora de um container web.

Application Controller é um ponto central do seu aplicativo e quando usamos frameworks como Struts e JSF já temos tudo isso disponível sem necessidade de programar explicitamente.



Anotações

---

---

---

**Exemplo: ApplicationController.java**

```

1 package br.com.globalcode.idp.controller;
2
3 import br.com.globalcode.idp.web.command.CommandFactory;
4 import br.com.globalcode.idp.web.command.WebCommand;
5 import java.io.IOException;
6 import javax.servlet.RequestDispatcher;
7 import javax.servlet.ServletException;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11 public class ApplicationController {
12
13 protected void process(HttpServletRequest request,
14 HttpServletResponse response)
15 throws ServletException, IOException {
16 WebCommand aCommand = null;
17 String command = request.getParameter("command");
18 try {
19 aCommand = CommandFactory.createWebCommand(command);
20 aCommand.doAction(request);
21 } catch (Exception e) {
22 throw new ServletException(e);
23 }
24 String destino = (String) request.getAttribute("destino");
25 String navegacao = (String) request.getAttribute("tipoNavegacao");
26 if (destino != null && !destino.equals("")) {
27 if (navegacao.equalsIgnoreCase("forward")) {
28 RequestDispatcher rd = request.getRequestDispatcher(destino);
29 rd.forward(request, response);
30 } else {
31 response.sendRedirect(request.getContextPath() + destino);
32 }
33 } else {
34 response.sendRedirect(request.getContextPath() + "/index.jsp");
35 }
36 }
37 }
38
39

```

**Anotações**

## 5.10 Intercepting Filter

A maioria das aplicações tem requerimentos, tais como, segurança e log, que serão aplicados a todas as requests. A adição deste serviço, em cada parte da aplicação, demanda muito tempo, além de criar vulnerabilidade a erros (é comum esquecer de implementar segurança ou log em uma página). Trata-se também de uma solução difícil de ser mantida.

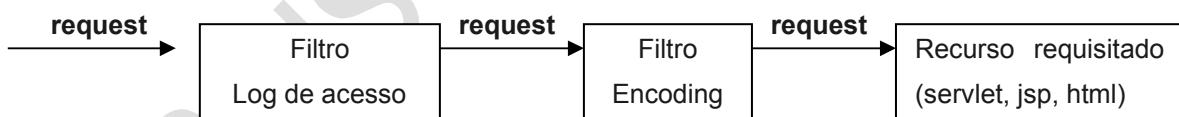
Este pattern propõe a centralização do trabalho comum de infraestrutura em um objeto filtro, interceptando, ao criar uma classe Filter, tanto o recebimento das requisições (Request), como o envio da resposta (Response). Desta maneira o filtro poderá redirecionar, ou ainda, processar dados da request, bem como, os da response, antes que os dados sejam realmente enviados para o cliente.

### Aplicação prática:

- Log;
- Segurança;
- Filtro de conteúdo;
- Debug;
- Processamento de xml e xsl;
- Determinação do encoding adequado.

Também podemos utilizar mais de um filtro, formando o que chamamos chain filter (cadeia de filtros).

Veja um exemplo na figura abaixo:



Neste exemplo existem dois filtros, sendo o primeiro responsável pelo log de acesso a recursos configurados no filtro e o segundo por indicar o encoding adequado na response. Depois que os dois filtros são executados, o recurso requisitado é chamado.

### Benefícios:

- Centralização de lógica utilizada por diversos componentes;
- Maior facilidade em adicionar ou remover serviços, sem afetar o código existente.

### Anotações

---

Código de exemplo de um Servlet Filter.

---

**Exemplo: LogAccessFilter.java**

---

```
1 package br.com.globalcode.idp.web.filter;
2
3 import javax.servlet.*;
4 import javax.servlet.http.*;
5 import java.io.IOException;
6
7 public class LogAccessFilter implements Filter {
8
9 private FilterConfig config = null;
10
11 public void init(FilterConfig config) throws ServletException {
12 this.config = config;
13 String logFile = config.getInitParameter("logfile");
14 System.out.println("logfile: " + logFile);
15 }
16
17 public void destroy() {
18 config = null;
19 }
20
21 public void doFilter(ServletRequest request, ServletResponse response,
22 FilterChain chain) throws IOException, ServletException {
23 HttpServletRequest req = (HttpServletRequest) request;
24 String host = req.getRemoteHost();
25 String requestedUri = req.getRequestURI();
26 //Log em Database
27 // proximo filtro ou entao url solicitada
28 chain.doFilter(request, response);
29 }
30 }
31
```



---

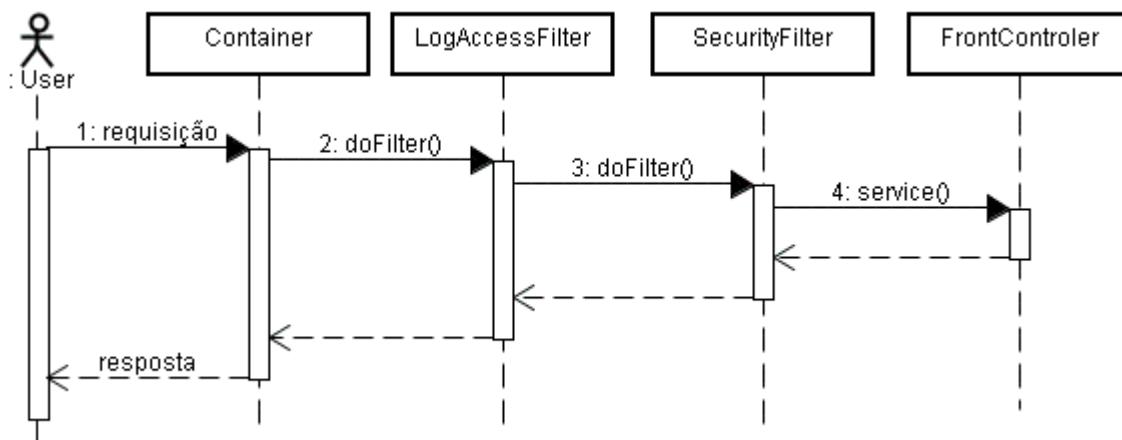
**Anotações**

---

## Deployment Descriptor: web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
5 <filter>
6 <filter-name>LogAccessFilter</filter-name>
7 <filter-class> br.com.globalcode.aa.filter.LogAccessFilter
8 </filter-class>
9 <init-param>
10 <param-name>logfile</param-name>
11 <param-value>/java/logAccessFilter.log</param-value>
12 </init-param>
13 </filter>
14 <filter-mapping>
15 <filter-name>LogAccessFilter</filter-name>
16 <url-pattern>/*</url-pattern>
17 </filter-mapping>
18 </web-app>
```

## Diagrama de sequência



## Anotações

## 5.11 View Helper

O **View Helper** é um padrão J2EE criado para solucionar problemas em JSPs que tendem a ficar extensos e complexos com a inserção de regras para definir o comportamento de exibição de dados dinâmicos entre JavaBeans e HTML.

Um exemplo típico é quando temos que formatar uma data em um JSP; o código que cria a instância de um `SimpleDateFormat` e aplica a máscara de formatação pode ser reaproveitado se estiver em uma classe (um **View Helper**). Outros exemplos: formatador de totais, apresentação de ComboBoxes repleto com dados de um Array ou Collection, etc.

A forma mais comum de solucionar estes problemas de renderização de dados em páginas JSP é a criação de JavaBeans e Custom Tags especializados para esse tipo de finalidade. A Custom Tag é um mecanismo de componentização com forte poder de manipulação de dados e front-ends com markup.

Vantagens de uso de uma Custom Tag:

- Extensível;
- Fácil acoplamento e entendimento;
- Praticamente permite a criação de uma linguagem para sua solução.

No exemplo a seguir apresentamos uma página JSP que utiliza os tags da biblioteca JSTL e um tag customizado do aplicativo para exibir uma página de matrícula para uma turma. O tag `<jaref:ComboMemberships>` monta uma caixa combo HTML com uma lista de alunos obtida a partir de um banco de dados. Esta página JSP é inserida dentro de uma página composta que utiliza o pattern Composite View.

Além disso, apresentamos também o código da classe criada para o tag (`ComboMembershipsHelper.java`) e o seu arquivo de configuração (`helper.tld`).

---

Anotações

---

---

---

## Exemplo: formdata.jsp

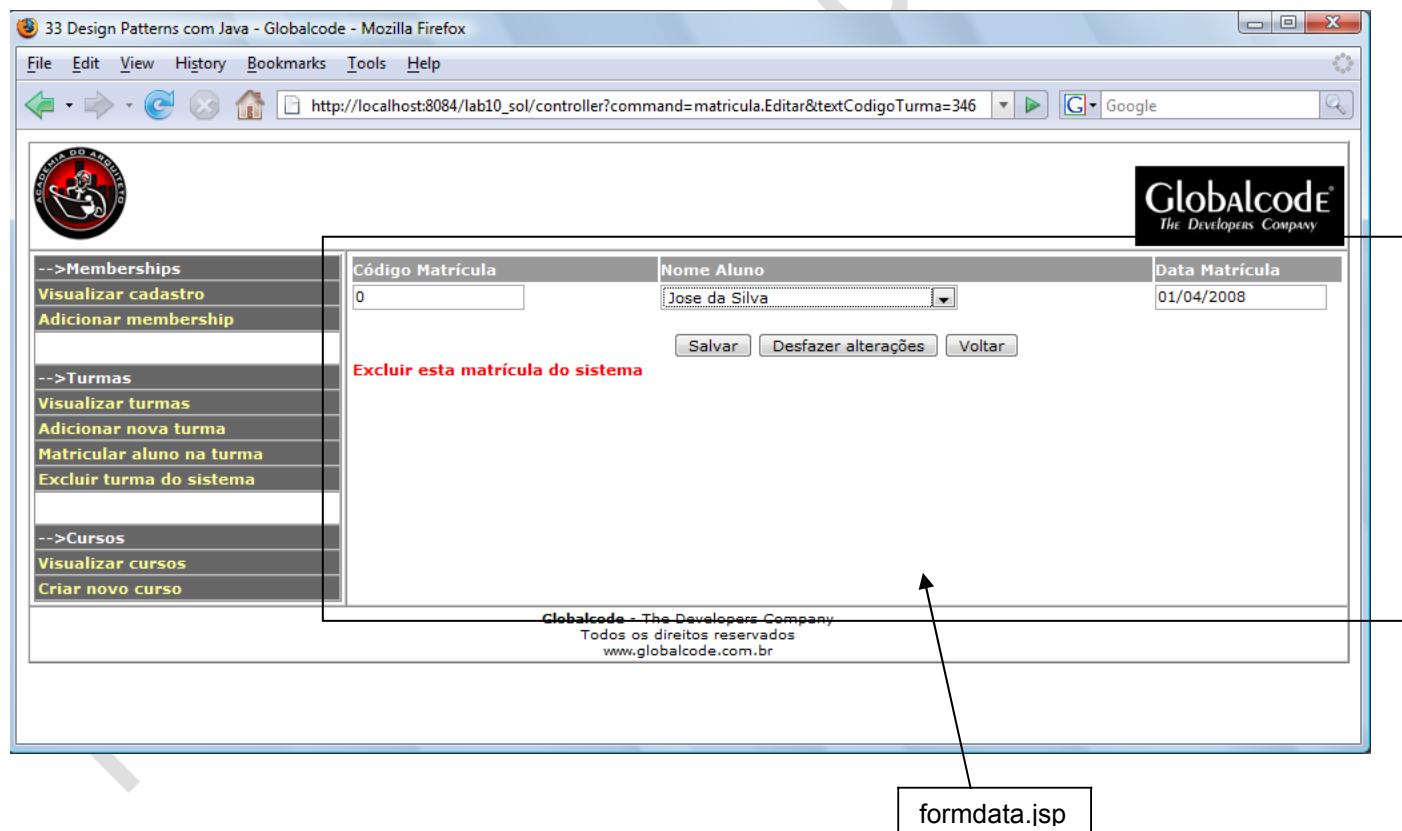
```
1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2 <%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
3 <%@ taglib uri="http://www.globalcode.com.br/jaref" prefix="jaref" %>
4
5 <form name="form1" method="post"
6 action="${pageContext.servletContext.contextPath}
7 /controller?command=matricula.Salvar">
8 <input type="hidden" name="textCodigoTurma"
9 value="${matricula.codigoMatricula==0
10 ? param.textCodigoTurma : matricula.codigoTurma}">
11
12 <table width="100%" border="0" cellspacing="2" cellpadding="0">
13 <tr>
14 <td width="31%" height="19" class="tituloCampo">
15 Código Matrícula</td>
16 <td class="tituloCampo">Nome Aluno</td>
17 <td class="tituloCampo">Data Matrícula</td>
18 </tr>
19 <tr>
20 <td class="Campo">
21 <input name="textCodigoMatricula" type="text"
22 id="textCodigoMatricula"
23 value="${matricula.codigoMatricula}"
24 size="18" maxlength="12" readonly="true"></td>
25 <td width="50%" class="Campo">
26 <c:choose>
27 <c:when test="${matricula.codigoMatricula != 0}">
28 ${matricula.nomeMembership}
29 <input type="hidden" name="comboMemberships"
30 value="${matricula.codigoMembership}">
31 </c:when>
32 <c:otherwise>
33 <jaref:ComboMemberships name="comboMemberships"
34 value="${matricula.codigoMembership}" />
35 </c:otherwise>
36 </c:choose>
37 </td>
38 <td width="19%" class="Campo">
39 <input name="textDataMatricula" type="text"
40 id="textDataMatricula"
41 value='<fmt:formatDate pattern="dd/MM/yyyy"
42 value="${matricula.dataMatricula}" />'
43 size="18" maxlength="12"></td>
44 </tr>
45 <tr>
46 <td colspan="3" class="Campo">&nbsp</td>
47 </tr>
48 <tr>
49 <td colspan="3" class="Campo"> <div align="center">
50 <input type="submit" name="Submit" value="Salvar">
51 </div>
52 </tr>
```

Anotações

```

50 <input type="submit" name="Submit" value="Salvar">
51 <input type="reset" name="Submit2"
52 value="Desfazer alterações">
53 <input name="voltar" type="button"
54 onClick="history.back(1); value='Voltar'" value="Voltar">
55 </div></td>
56 </tr>
57 <tr>
58 <td colspan="3" class="Campo">
59 <a href="${pageContext.servletContext.contextPath}
60 /controller?command=matricula.Excluir
61 &textCodigoMatricula=${matricula.codigoMatricula}">
62
63 Excluir esta matrícula do sistema
64 </td>
65 </tr>
66 </table>
67 </form>
68

```



## Anotações

## Exemplo: ComboMembershipsHelper.java

---

```
|1 package br.com.globalcode.idp.helper.view;
|2
|3 import br.com.globalcode.idp.dao.MembershipDAO;
|4 import br.com.globalcode.idp.exception.GlobalcodeException;
|5 import br.com.globalcode.idp.model.Membership;
|6 import java.io.IOException;
|7 import javax.servlet.jsp.JspContext;
|8 import javax.servlet.jsp.JspTagException;
|9 import javax.servlet.jsp.tagext.SimpleTagSupport;
|10
|11 public class ComboMembershipsHelper extends SimpleTagSupport {
|12
|13 private String cssClass;
|14 private String value;
|15 private String name;
|16
|17 @Override
|18 public void doTag() throws JspTagException {
|19 char aspas = '"';
|20 Membership memberships[] = null;
|21 try {
|22 memberships = MembershipDAO.getInstance().getMemberships(true);
|23 } catch (GlobalcodeException ge) {
|24 throw new JspTagException("Erro Lendo cursos para o ComboTag!");
|25 }
|26
|27 try {
|28 JspContext context = getJspContext();
|29 context.getOut().write("<select name=" + aspas + name + aspas +
|30 " class=" + aspas + cssClass + aspas + ">");
|31 boolean sel = false;
|32 for (int x = 0; x < memberships.length; x++) {
|33 sel = (value != null && value.equals("") +
|34 memberships[x].getCodigoMembership()));
|35 context.getOut().write("<option value=" + aspas +
|36 memberships[x].getCodigoMembership() + aspas +
|37 (sel ? " selected " : "") +
|38 ">" + memberships[x].getNome() + "</option>");
|39 }
|40 context.getOut().write("</select>");
|41 context.getOut().write("");
|42 } catch (IOException ex) {
|43 throw new JspTagException("Erro...");
|44 }
|45 }
|46
|47 public void setCssClass(String cssClass) {
|48 this.cssClass = cssClass;
|49 }
|50 }
```

Anotações

---

---

---

```

51 public void setValue(String value) {
52 this.value = value;
53 }
54
55 public void setName(String name) {
56 this.name = name;
57 }
58 }
59

```

**Deployment Descriptor: helper.tld**

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
5 <tlib-version>1.1</tlib-version>
6 <jsp-version>2.0</jsp-version>
7 <short-name>jaref</short-name>
8 <uri>http://www.globalcode.com.br/jaref</uri>
9
10 <tag>
11 <name>ComboMemberships</name>
12 <tag-class>br.com.globalcode.idp.helper.view.ComboMembershipsHelper
13 </tag-class>
14 <body-content>empty</body-content>
15 <attribute>
16 <name>name</name>
17 <required>true</required>
18 <rtpexprvalue>true</rtpexprvalue>
19 </attribute>
20 <attribute>
21 <name>value</name>
22 <required>false</required>
23 <rtpexprvalue>true</rtpexprvalue>
24 </attribute>
25 <attribute>
26 <name>cssClass</name>
27 <required>false</required>
28 <rtpexprvalue>true</rtpexprvalue>
29 </attribute>
30 </tag>
31 </taglib>
32

```

**Anotações**

## 5.12 Laboratório 10

Objetivo:

Completar o refactoring do aplicativo de gerenciamento de escolas implementando os patterns Front Controller, Command, Factory e MVC.



**Tabela de atividades:**

| Atividade                                                                                                                                                                                                                                                                                           | OK |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1. Faça o download do arquivo lab09_sol.zip na URL indicada pelo instrutor(a).                                                                                                                                                                                                                      |    |
| 2. Descompacte o arquivo em seu diretório de trabalho.                                                                                                                                                                                                                                              |    |
| 3. Abra o projeto no NetBeans e rode o aplicativo pressionando F6                                                                                                                                                                                                                                   |    |
| 4. Este é um aplicativo de gerenciamento de escolas. Navegue no aplicativo e certifique-se que tudo está funcionando a contento.                                                                                                                                                                    |    |
| 5. O aplicativo deverá ser refatorado utilizando os patterns vistos no curso. Devem ser aplicados os patterns Front Controller, Command, Factory e MVC. Não é necessário criar os Commands para todas as funcionalidades. Faça apenas para aquelas relativas a manipulação de alunos (memberships). |    |

Anotações

---

---

---