

Padrões de Projeto

Disciplina: Engenharia de Software - 2009.1

Professora: Rossana Maria de Castro Andrade

Assistente da disciplina: Ricardo Fernandes de Almeida

O que é um Padrão?

- Um padrão descreve uma solução para um problema que ocorre com freqüência durante o desenvolvimento de software, podendo ser considerado como um par “problema/solução” [Buschmann 96]. Design Patterns, ou Padrões de Design*.
 - Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto.
 - Inspirado em "A Pattern Language" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios.

Padrões

- Padrões são um repertório de soluções e princípios que ajudam os desenvolvedores a criar software e que são codificados em um formato estruturado consistindo de
 - Nome.
 - Problema que soluciona.
 - Solução do problema.
- O objetivo dos padrões é codificar conhecimento (*knowing*) existente de uma forma que possa ser reaplicado em contextos diferentes.

Padrões clássicos ou padrões GoF

- O livro "Design Patterns" (1994) de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, descreve 23 padrões de design
 - São soluções genéricas para os problemas mais comuns do desenvolvimento de software orientado a objetos.
 - O livro tornou-se um clássico na literatura orientada a objeto e continua atual.
 - Não são invenções. São documentação de soluções obtidas através da experiência. Foram coletados de experiências de sucesso na indústria de software, principalmente de projetos em C++ e SmallTalk.
 - Os quatro autores, são conhecidos como "The Gang of Four", ou GoF.

Por que aprender padrões?

- Aprender com a experiência dos outros.
 - Identificar problemas comuns em engenharia de software e utilizar soluções testadas e bem documentadas.
 - Utilizar soluções que têm um nome: facilita a comunicação, compreensão e documentação.
- Aprender a programar bem com orientação a objetos
 - Os 23 padrões de projeto "clássicos" utilizam as melhores práticas em OO para atingir os resultados desejados
- Desenvolver software de melhor qualidade.
 - Os padrões utilizam eficientemente polimorfismo, herança, modularidade, composição, abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento.

Por que aprender padrões?(cont)

- Vocabulário comum Faz o sistema ficar menos complexo ao permitir que se fale em um nível mais alto de abstração.
- Ajuda na documentação e na aprendizagem
 - Conhecendo os padrões de projeto torna mais fácil a compreensão de sistemas existentes.

Elementos de um padrão

- Nome
- Problema
 - Quando aplicar o padrão, em que condições?
- Solução
 - Descrição abstrata de um problema e como usar os elementos disponíveis (classes e objetos) para solucioná-lo.
- Conseqüências
 - Custos e benefícios de se aplicar o padrão
 - Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema.

Classificação dos 23 padrões segundo GoF

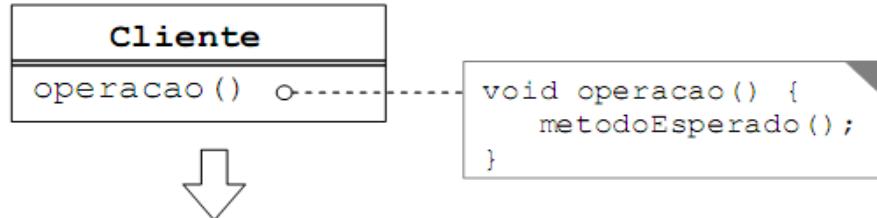
		Propósito		
Escopo	Classe	1. Criação	2. Estrutura	3. Comportamento
	Objeto	Factory Method	Class Adapter	Interpreter Template Method
		Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Adapter

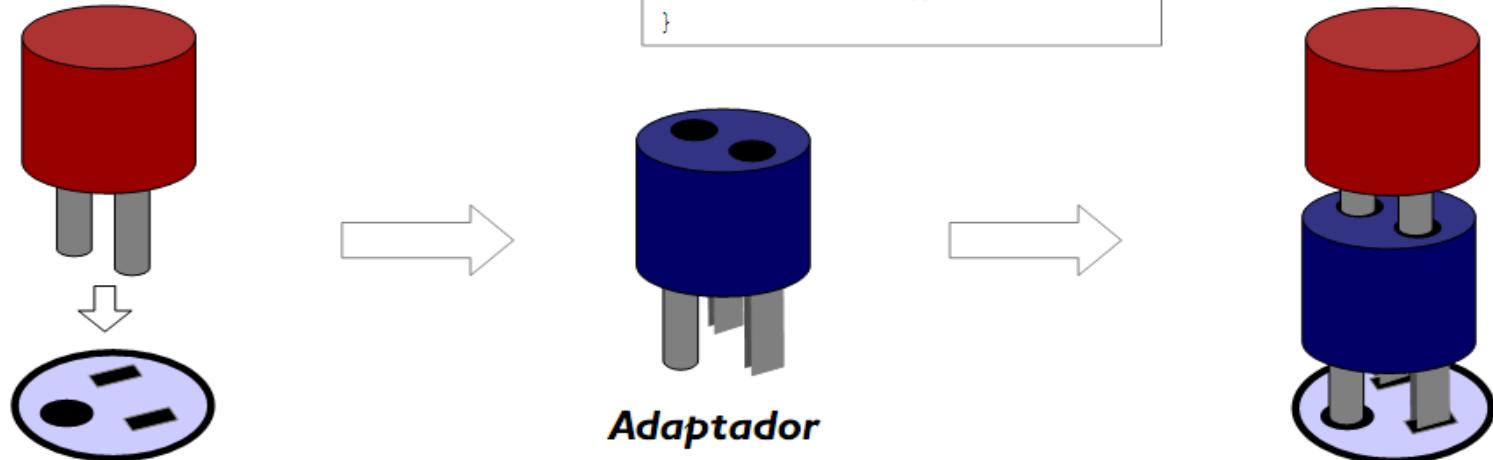
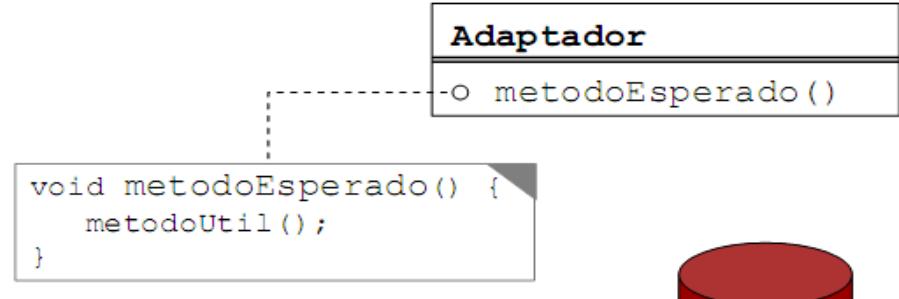
- "Objetivo: converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces." [GoF].

Cenário do Problema

Problema



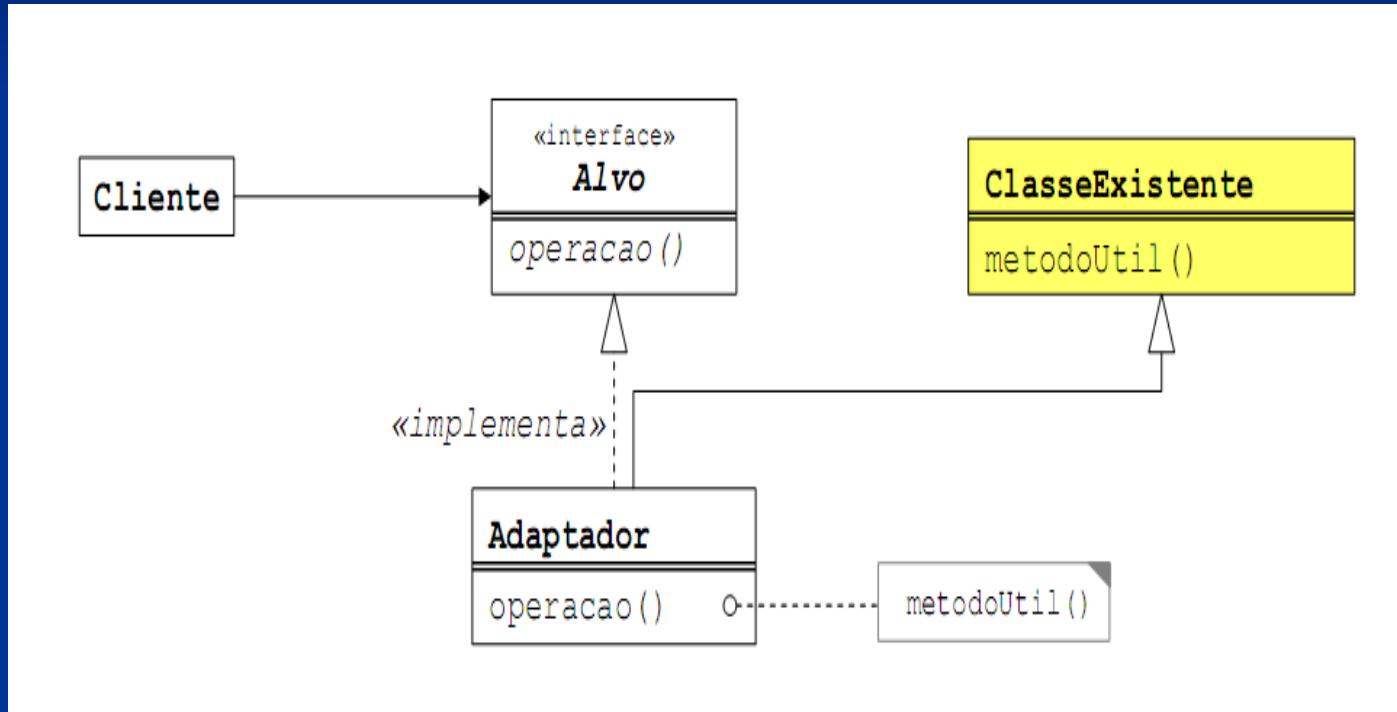
Solução



Adaptador

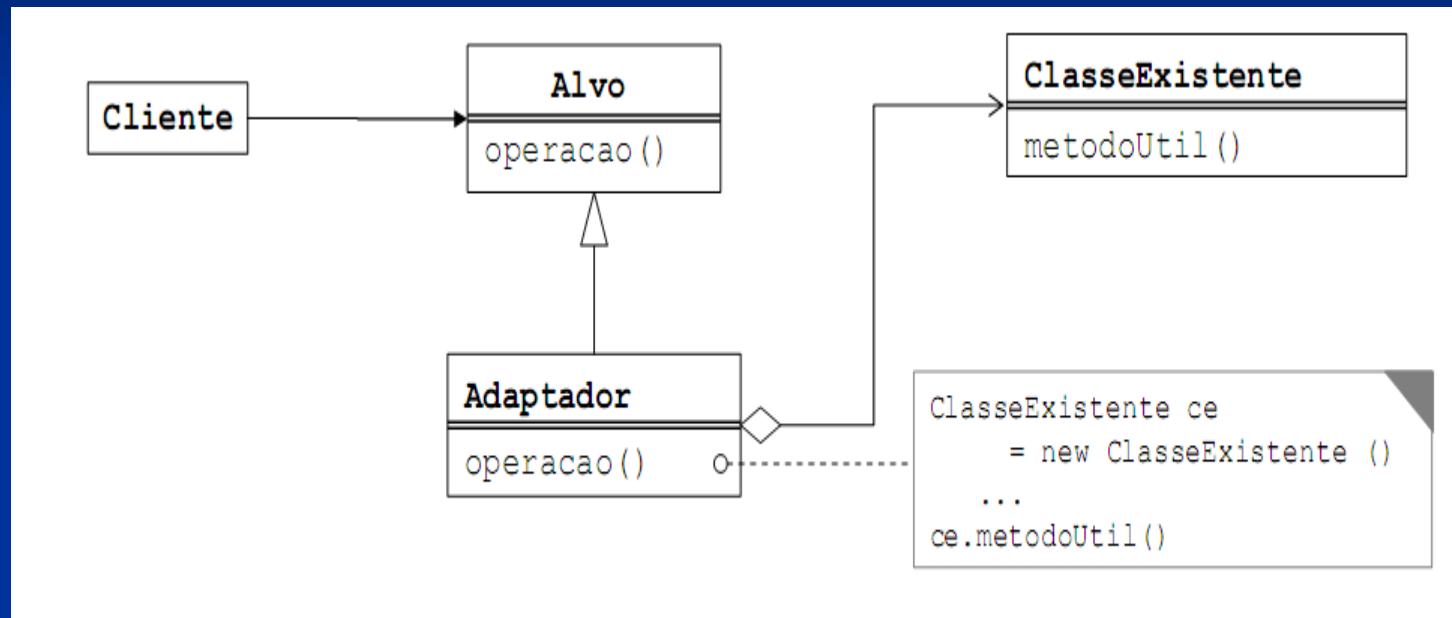
Duas Formas de Usar Adapter

- Class Adapter: usa herança múltipla:



Duas Formas de Usar Adapter(cont..)

- Object Adapter: usa composição



Quando Usar?

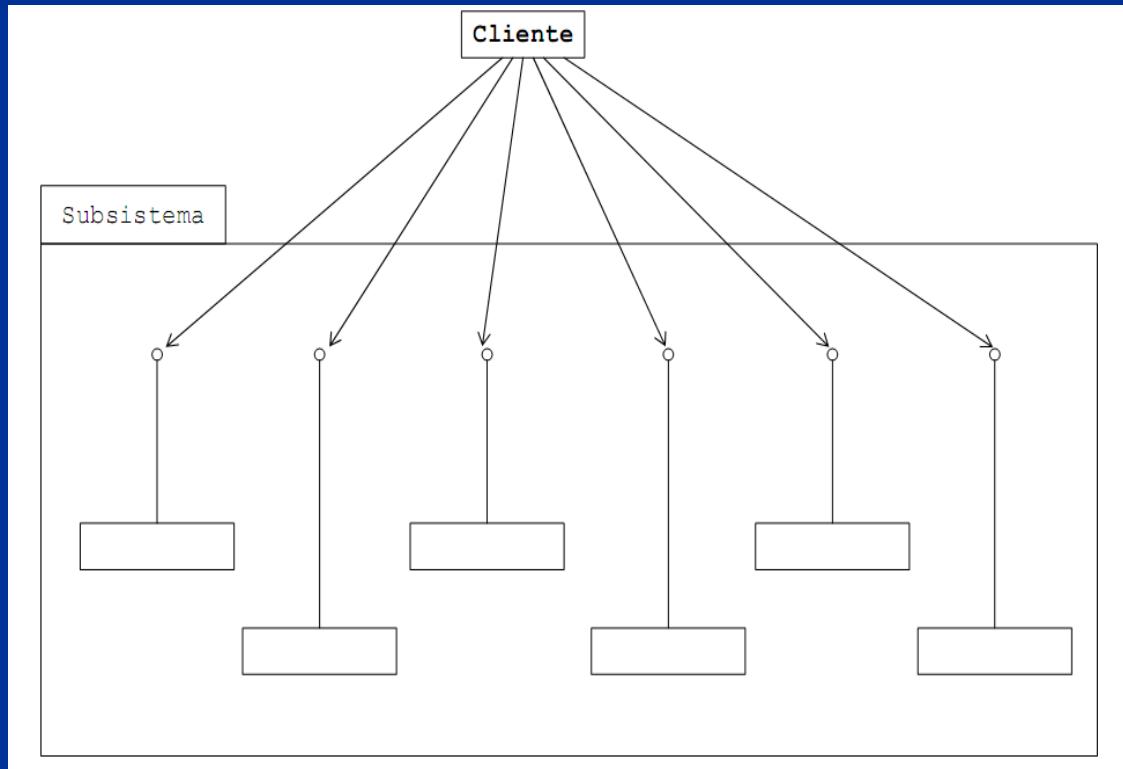
- Sempre que for necessário adaptar uma interface para um cliente Class Adapter.
 - Quando houver uma interface que permita a implementação estática.
- Object Adapter
 - Quando menor acoplamento for desejado.
 - Quando o cliente não usa uma interface Java ou classe abstrata que possa ser estendida.

Facade

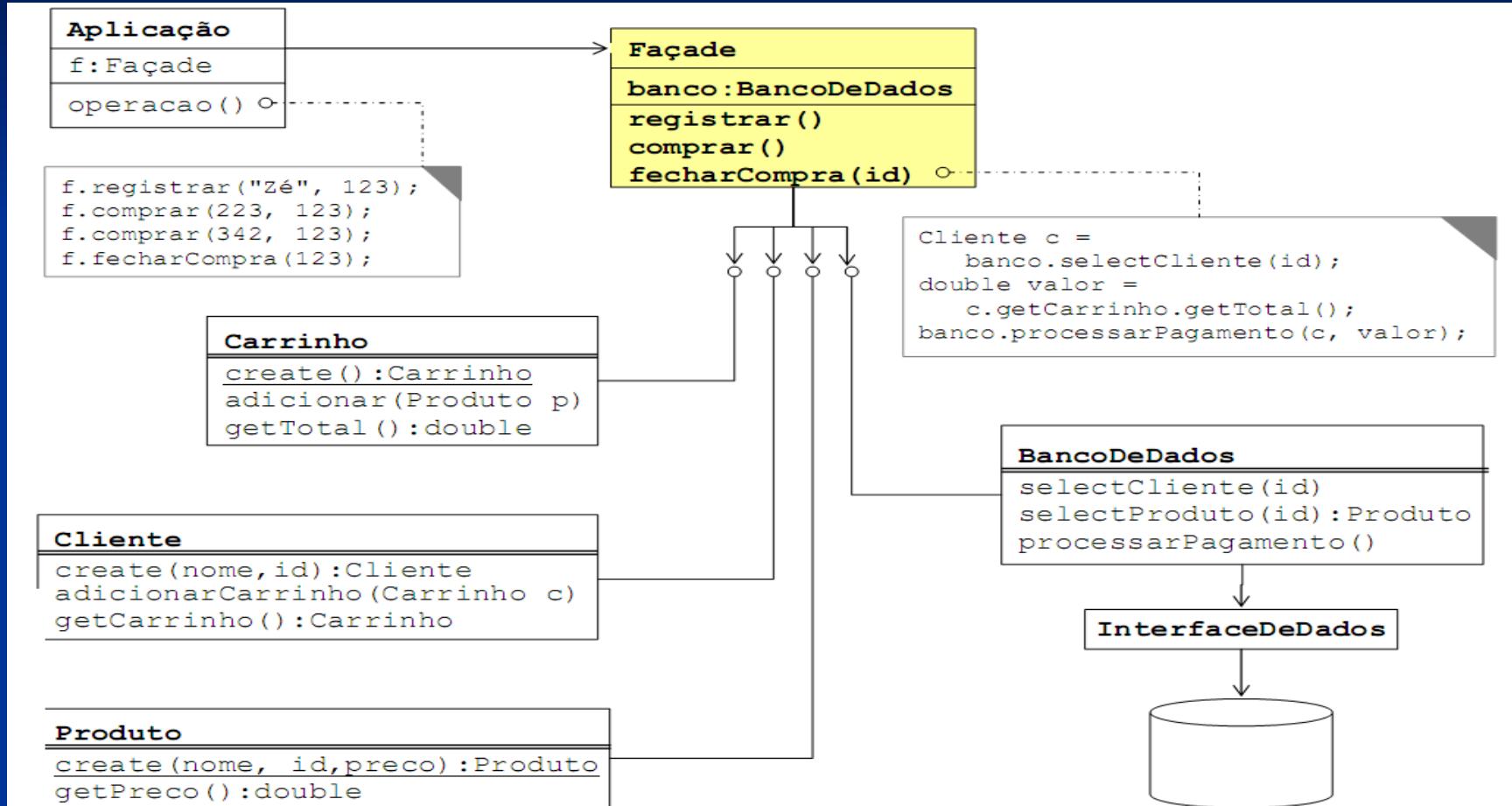
- "Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar." [GoF].

Problema

- Cliente precisa saber muitos detalhes do subsistema para utilizá-lo!



Exemplo



Quando Usar?

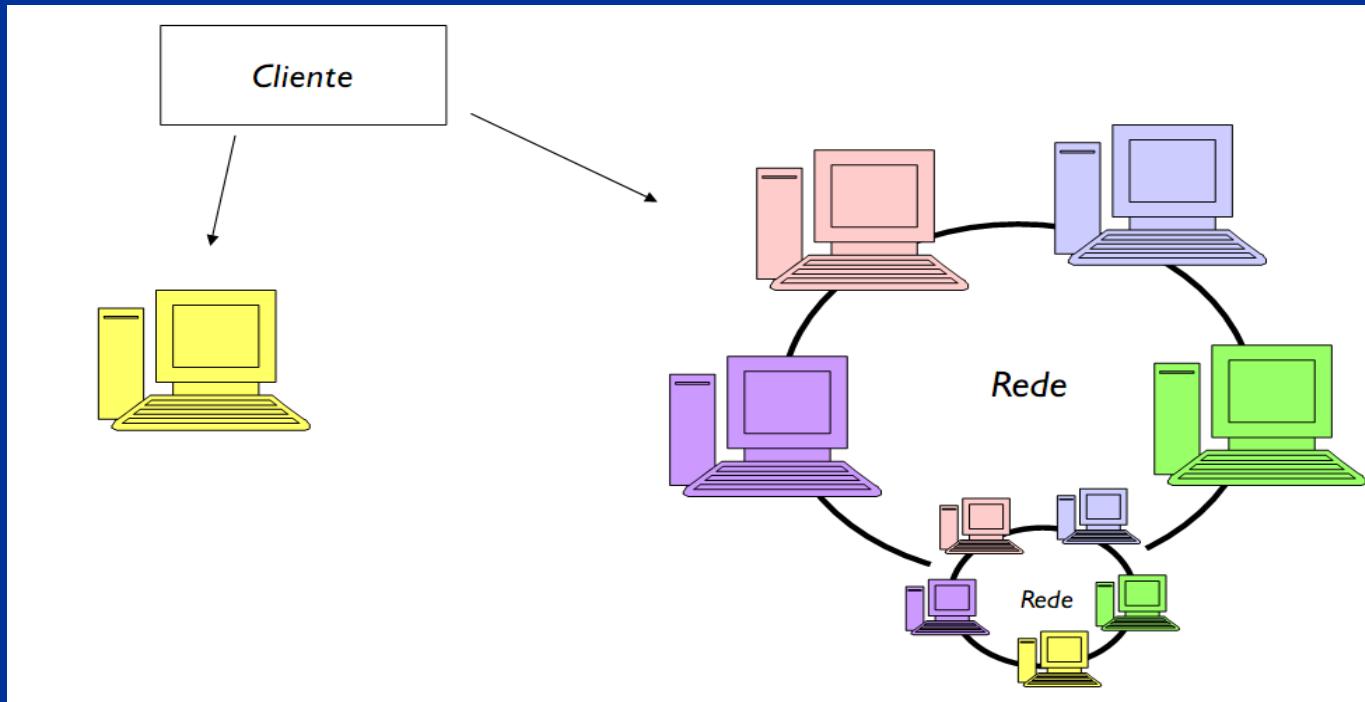
- Sempre que for desejável criar uma interface para um conjunto de objetos com o objetivo de facilitar o uso da aplicação.
 - Permite que objetos individuais cuidem de uma única tarefa, deixando que a fachada se encarregue de divulgar as suas operações.
- Fachadas viabilizam a separação em camadas com alto grau de desacoplamento.
- Existem em várias partes da aplicação
 - Fachada da aplicação para interface do usuário
 - Fachada para sistema de persistência: Data Access Object

Composite

- "Compor objetos em estruturas de árvore para representar hierarquias todo-parte. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme." [GoF].

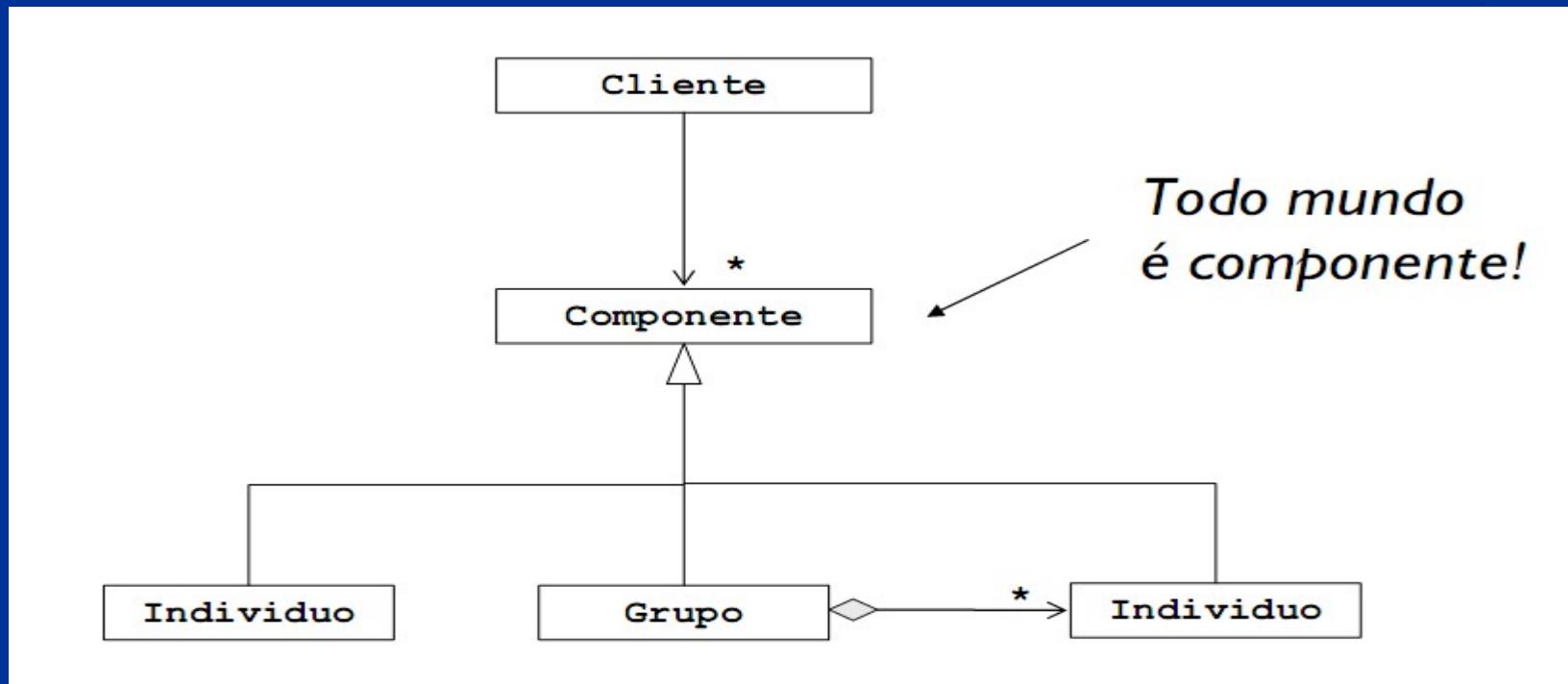
Problema

- Cliente precisa tratar de maneira uniforme objetos individuais e composições desses objetos.

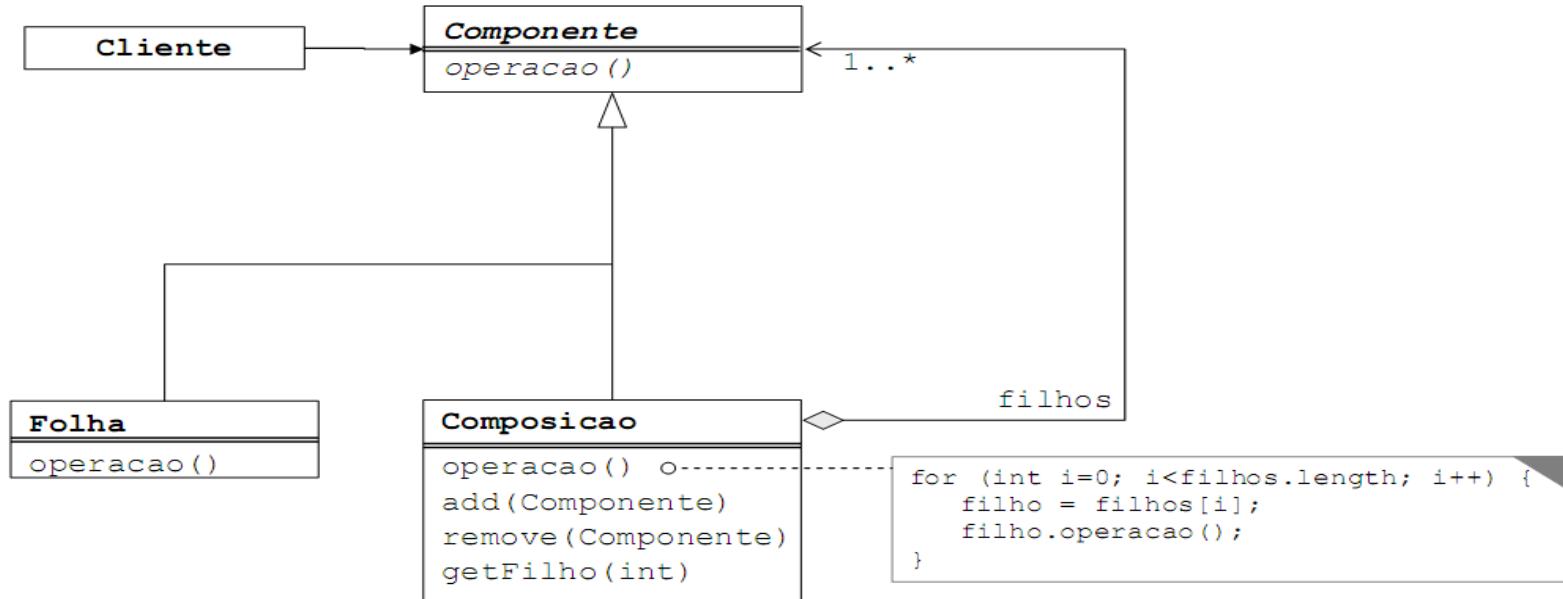


Solução

- Tratar grupos e indivíduos diferentes através de uma única interface.



Estrutura do Composite



Quando Usar?

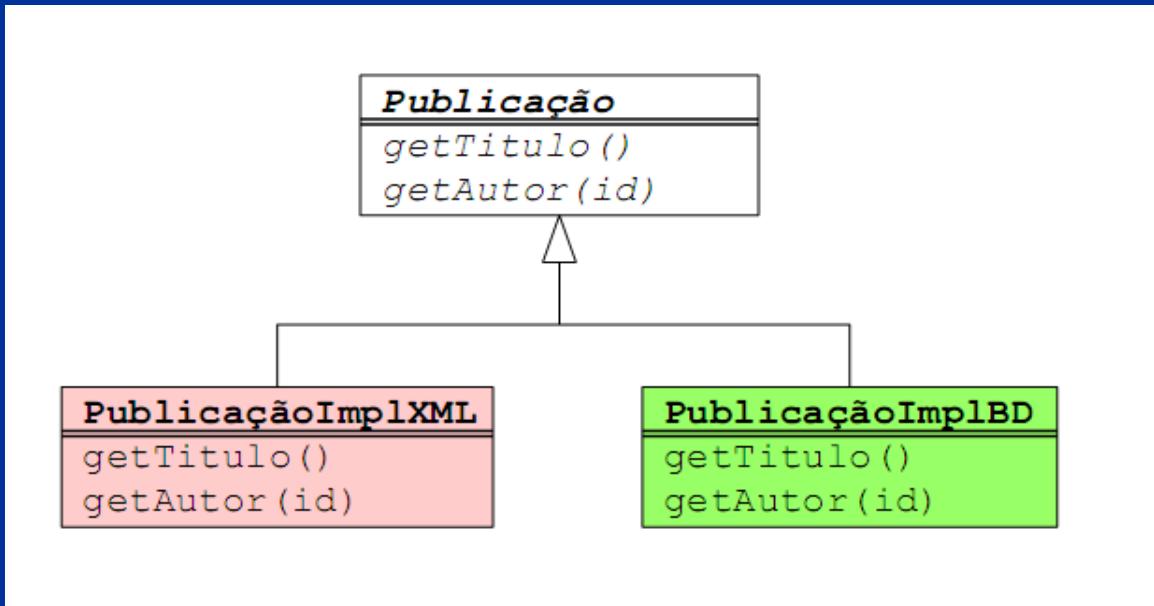
- Sempre que houver necessidade de tratar um conjunto como um indivíduo.
- Funciona melhor se relacionamentos entre os objetos for uma árvore.
 - Caso o relacionamento contenha ciclos, é preciso tomar precauções adicionais para evitar loops infinitos, já que Composite depende de implementações recursivas.
- Há várias estratégias de implementação.

Bridge

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

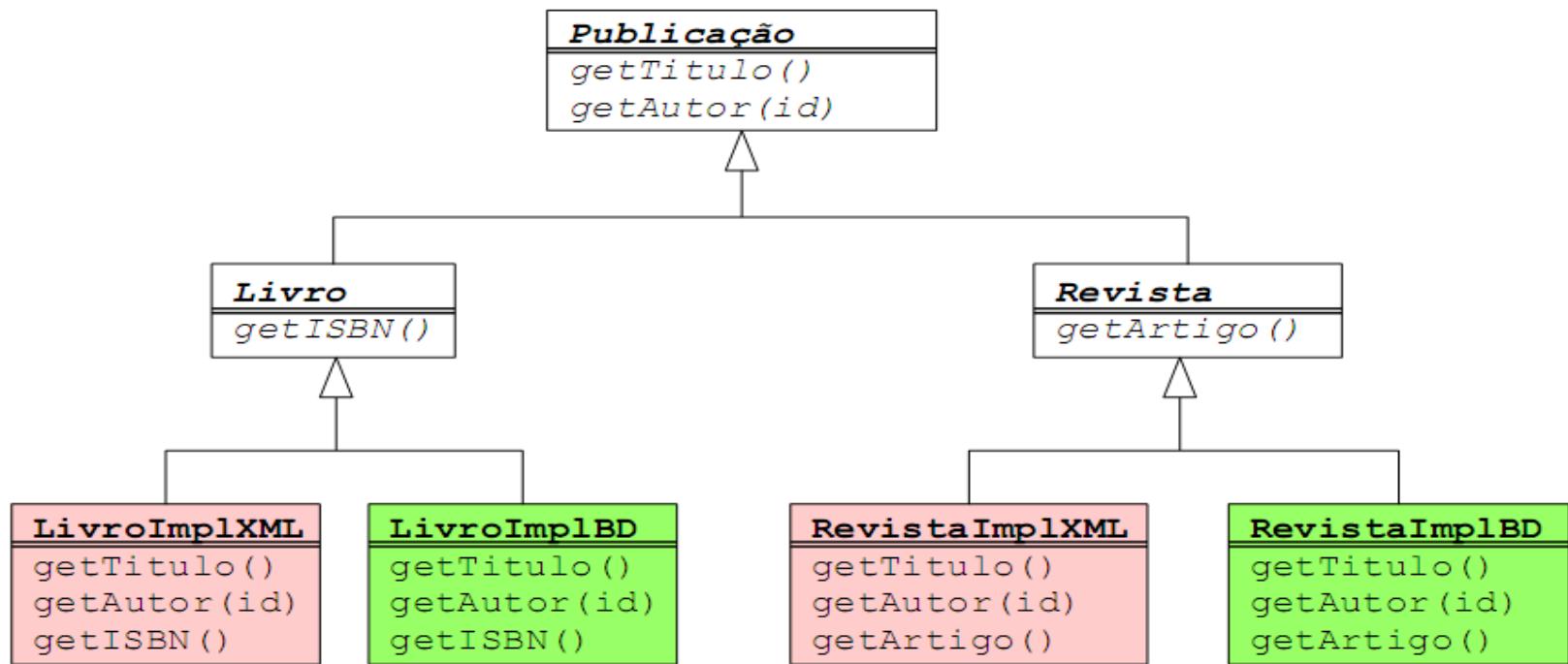
Problema (I)

- Necessidade de um driver.
- Exemplo: implementações específicas para tratar objeto em diferentes meios persistentes.

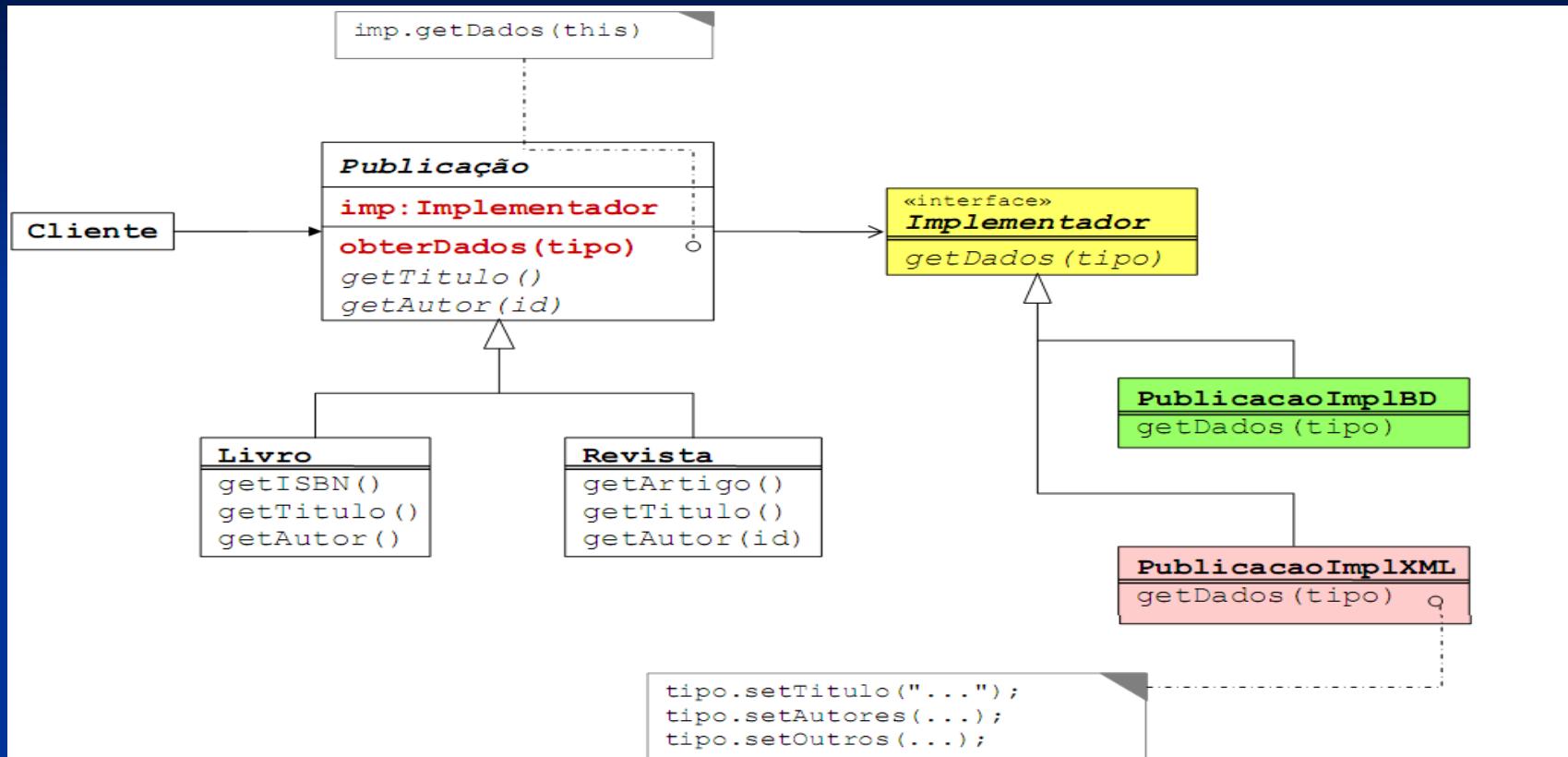


Problema(II)

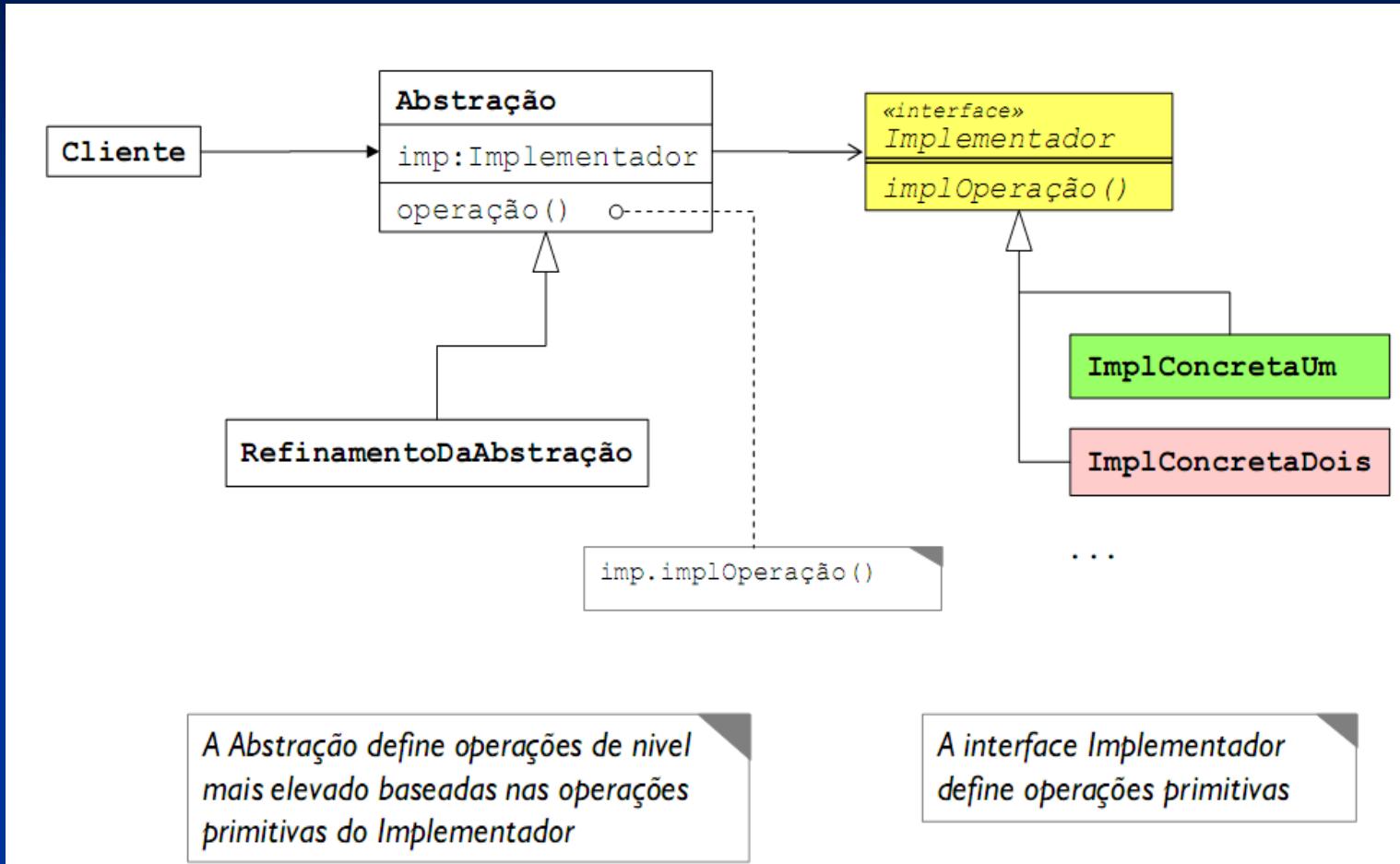
- Mas herança complica a implementação.



Solução: usar Bridge



Estrutura do Bridge



Quando Usar?

- Quando for necessário evitar uma ligação permanente entre a interface e implementação.
- Quando alterações na implementação não puderem afetar clientes.
- Quando tanto abstrações como implementações precisarem ser capazes de suportar extensão através de herança.
- Quando implementações são compartilhadas entre objetos desconhecidos do cliente.

Singleton

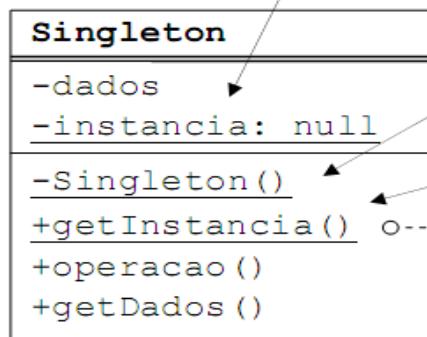
- "Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela." [GoF].

Problema

- Garantir que apenas um objeto exista, independente do número de requisições que receber para criá-lo.
- Aplicações
 - Um único banco de dados.
 - Um único acesso ao arquivo de log.
 - Um único objeto que representa um vídeo.
 - Uma única fachada (Facade pattern).
- Objetivo: garantir que uma classe só tenha uma instância.

Estrutura do Singleton

Objeto com acesso privativo



Construtor privativo (nem subclasses têm acesso)

Ponto de acesso simples, estático e global

```
public static Singleton getInstancia() {  
    if (instancia == null) {  
        instancia = new Singleton();  
    }  
    return instancia;  
}
```

Lazy initialization idiom

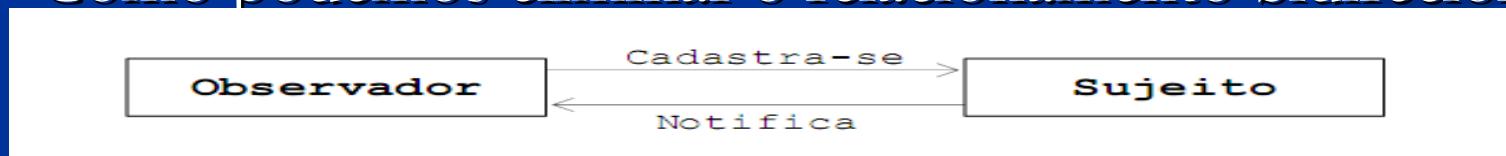
*Bloco deve ser **synchronized*** para evitar que dois objetos tentem criar o objeto ao mesmo tempo*

Observer

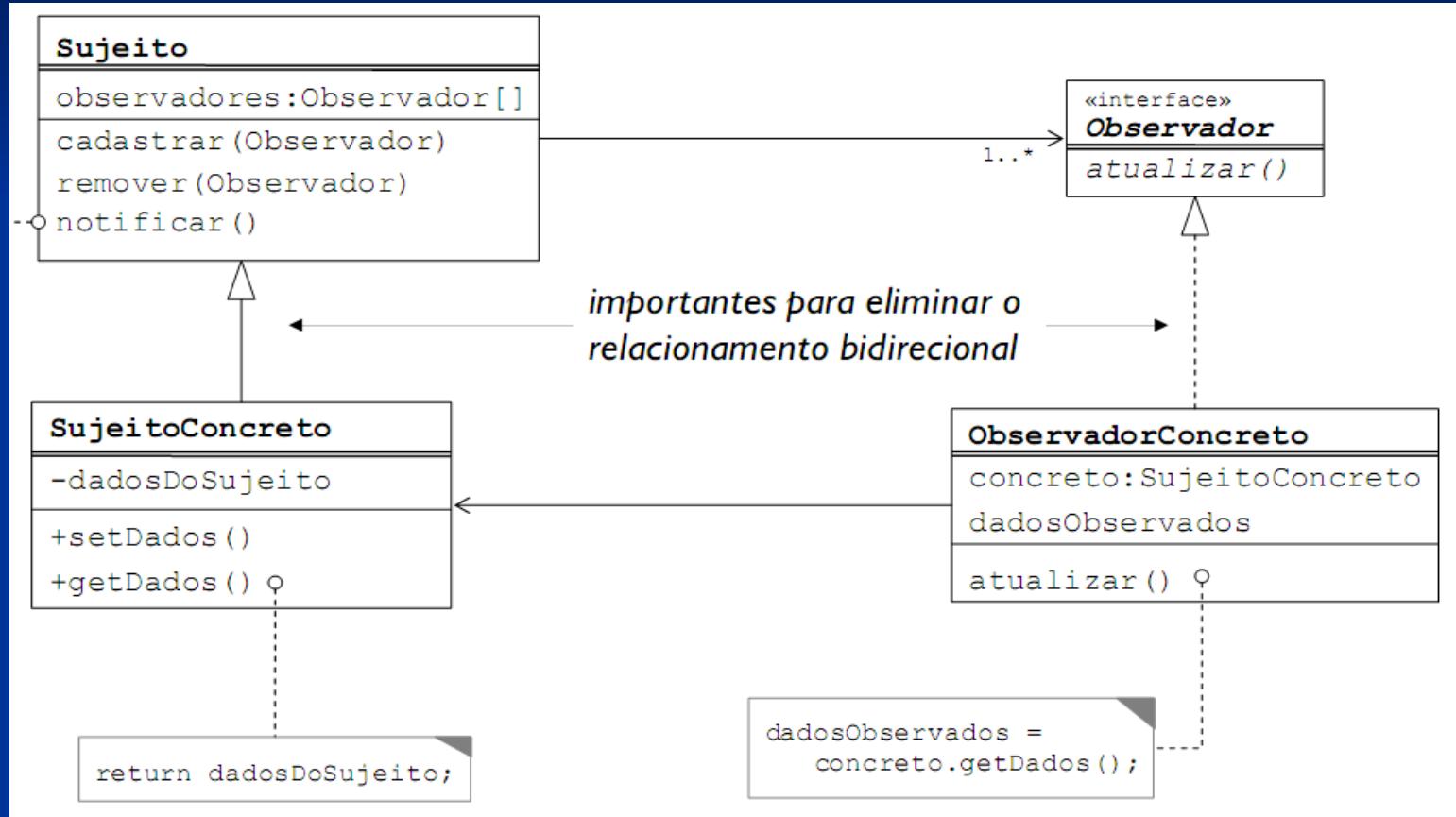
- "Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente." [GoF].

Problema

- Como garantir que objetos que dependem de outro objeto fiquem em dia com mudanças naquele objeto?
- Como fazer com que os observadores tomem conhecimento do objeto de interesse?
- Como fazer com que o objeto de interesse atualize os observadores quando seu estado mudar?
- Possíveis riscos
 - Relacionamento (bidirecional) implica alto acoplamento. Como podemos eliminar o relacionamento bidirecional?



Estrutura do Observer

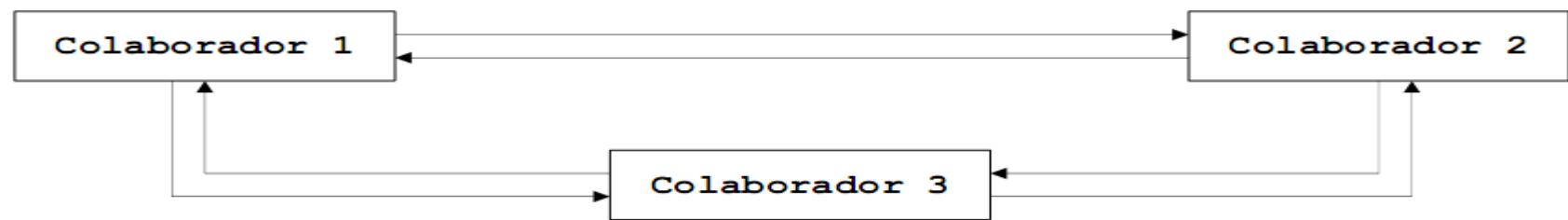


Mediator

- "Definir um objeto que encapsula como um conjunto de objetos interagem. Mediator promove acoplamento fraco ao manter objetos que não se referem um ao outro explicitamente, permitindo variar sua interação independentemente." [GoF].

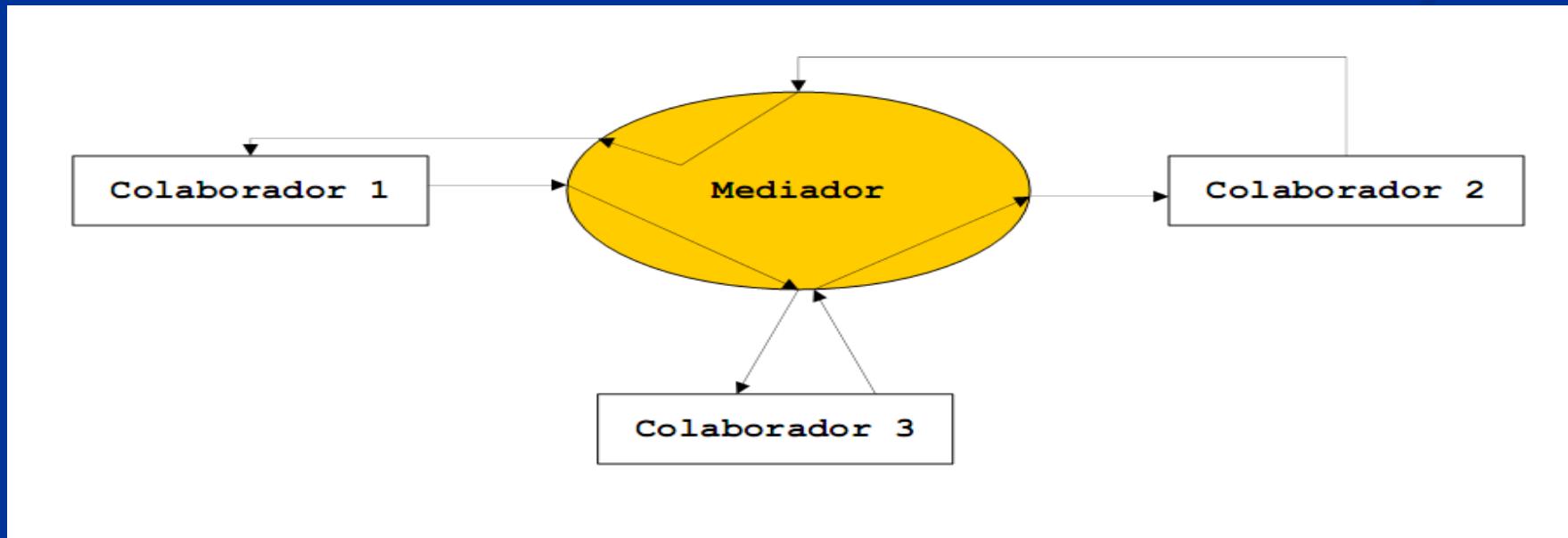
Problema

- Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?
- Como remover o forte acoplamento presente em relacionamentos muitos para muitos?
- Como permitir que novos participantes sejam ligados ao grupo facilmente?

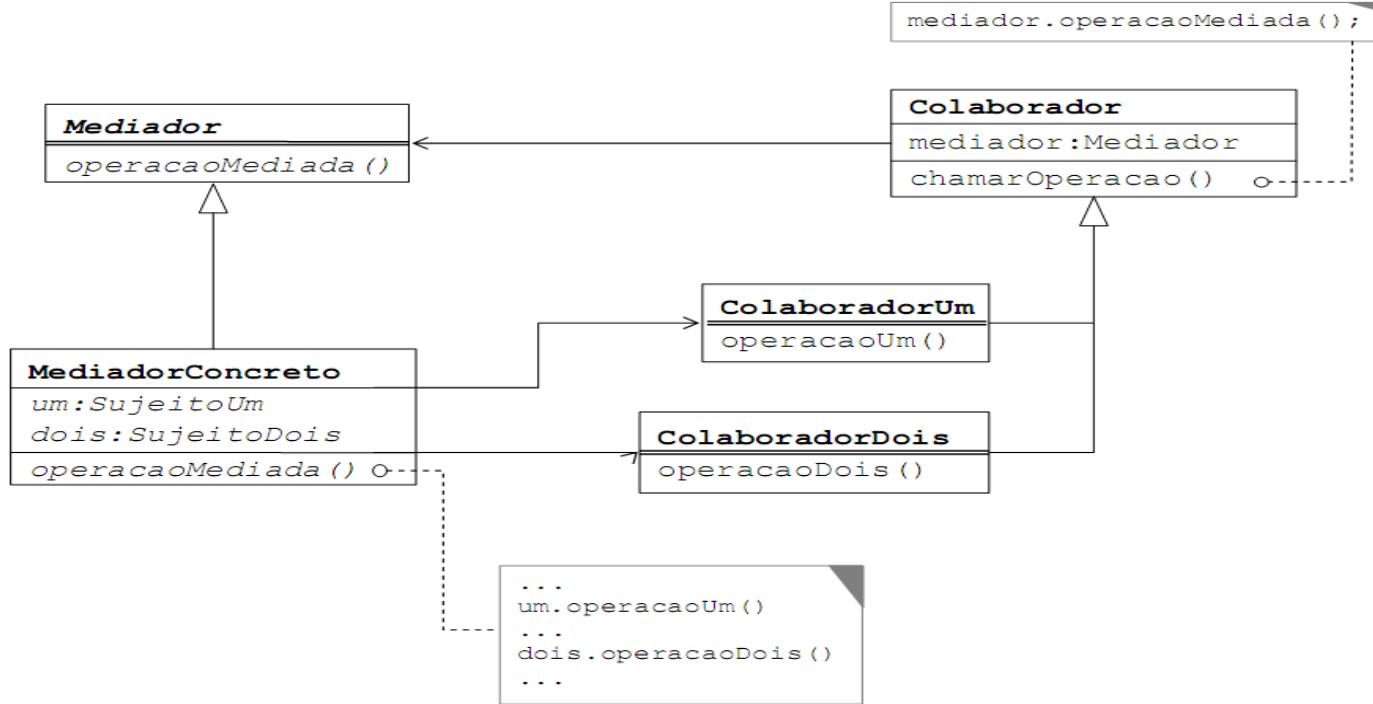


Solução

- Introduzir um mediador.
 - Objetos podem se comunicar sem se conhecer.



Estrutura do Mediator



Descrição da Solução

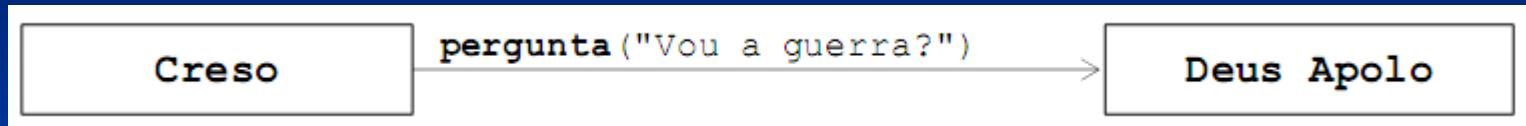
- Um objeto Mediador deve encapsular toda a comunicação entre um grupo de objetos.
 - Cada objeto participante conhece o mediador mas ignora a existência dos outros objetos.
 - O mediador conhece cada um dos objetos participantes.
- A interface do Mediador é usada pelos colaboradores para iniciar a comunicação e receber notificações.
 - O mediador recebe requisições dos remetentes.
 - O mediador repassa as requisições aos destinatários.
 - Toda a política de comunicação é determinada pelo mediador (geralmente através de uma implementação concreta do mediador).

Proxy

- "Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro."
[GoF].

Problema

- Sistema quer utilizar objeto real...

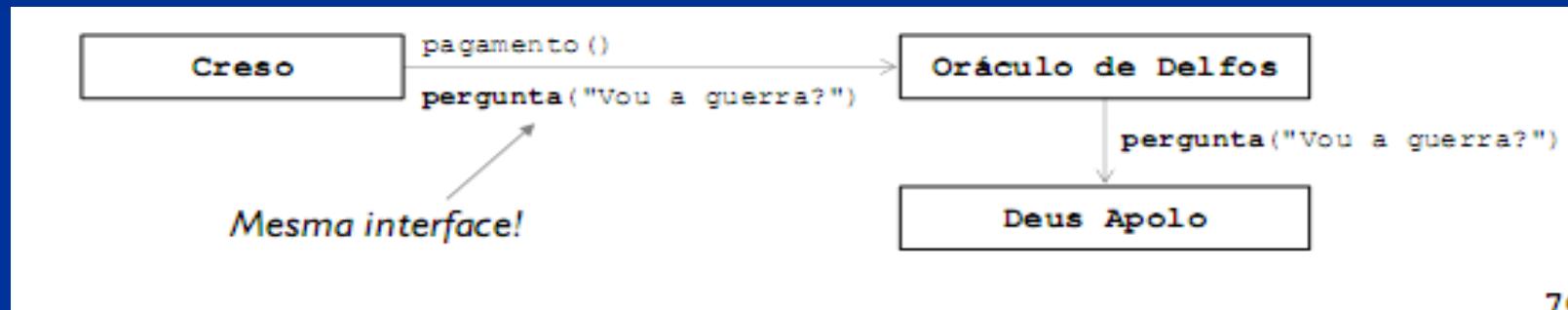


- Mas ele não está disponível (remoto, inacessível, ...)



Problema(cont..)

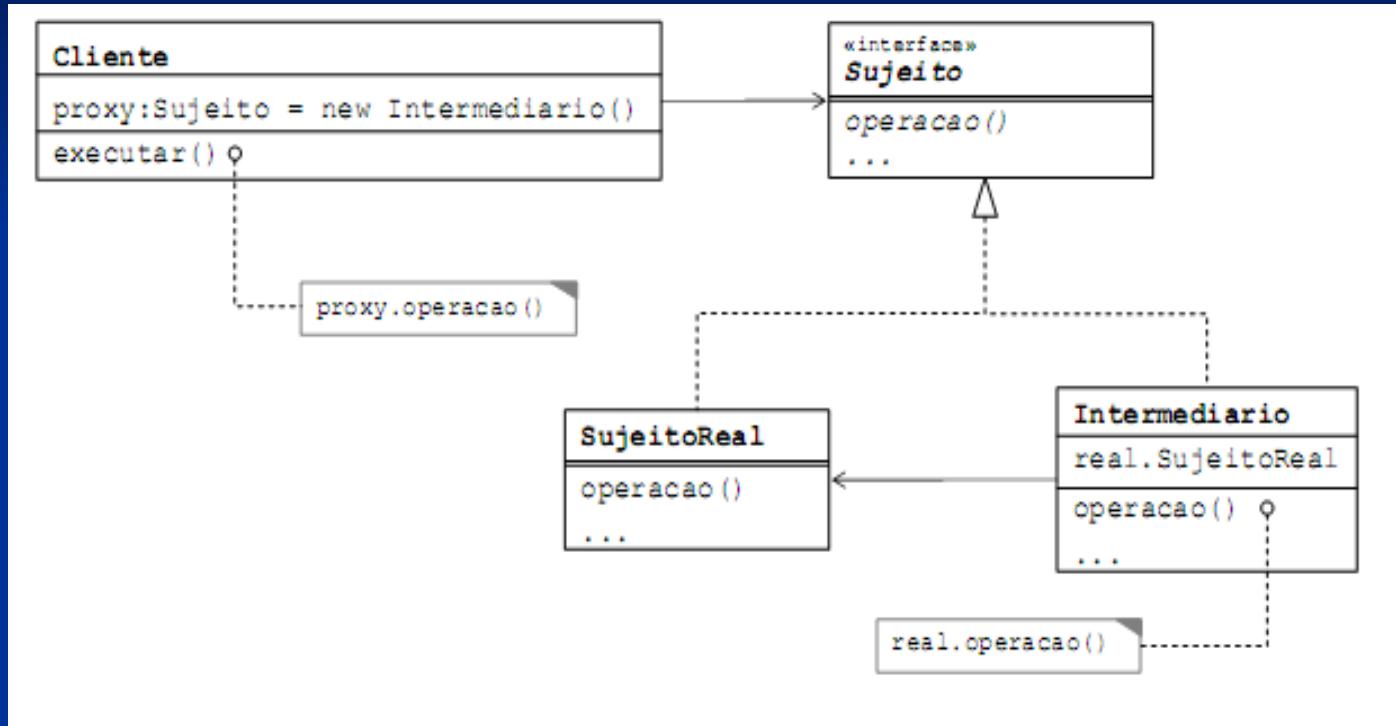
- Solução: arranjar um intermediário que saiba se comunicar com ele eficientemente.



Estrutura do Proxy

- Cliente usa intermediário em vez de sujeito real.
- Intermediário suporta a mesma interface que sujeito real.
- Intermediário contém uma referência para o sujeito real e repassa chamadas, possivelmente, acrescentando informações ou filtrando dados no processo.

Estrutura do Proxy(cont..)



Quando usar??

- A aplicação mais comum é em objetos distribuídos
- Exemplo: RMI (e EJB)
 - O Stub é proxy do cliente para o objeto remoto.
 - O Skeleton é parte do proxy: cliente remoto chamado pelo Stub.
- Outras aplicações típicas:
 - Image proxy: guarda o lugar de imagem sendo carregada.

Chain of Responsibility

- "Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição. Compõe os objetos em cascata e passa a requisição pela corrente até que um objeto a sirva." [GoF].

Problema

- Permitir que vários objetos possam servir a uma requisição ou repassá-la.
- Permitir divisão de responsabilidades de forma transparente.

Estratégias de Chain Of Responsibility

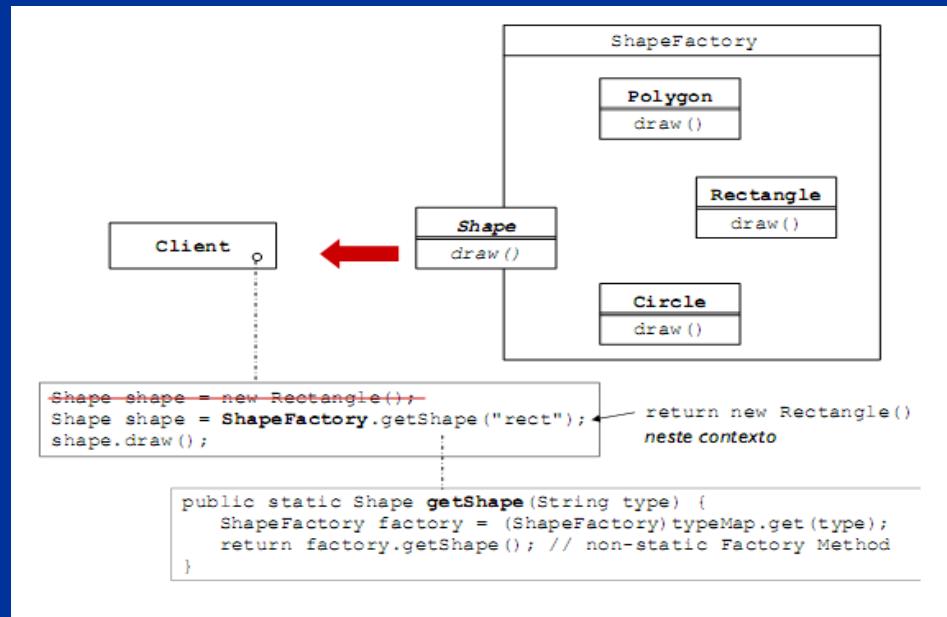
- Pode-se implementar um padrão de várias formas diferentes. Cada forma é chamada de estratégia (ou idiom*)
- Chain of Responsibility pode ser implementada com estratégias que permitem maior ou menor acoplamento entre os participantes.
- Usando um mediador: só o mediador sabe quem é o próximo participante da cadeia
- Usando delegação: cada participante conhece o seu Sucessor.

Factory Method

- "Definir uma interface para criar um objeto, mas deixar que subclasses decidam que classe instanciar. Factory Method permite que uma classe delegue a responsabilidade de instantiation às subclasses." [GoF].

Problema

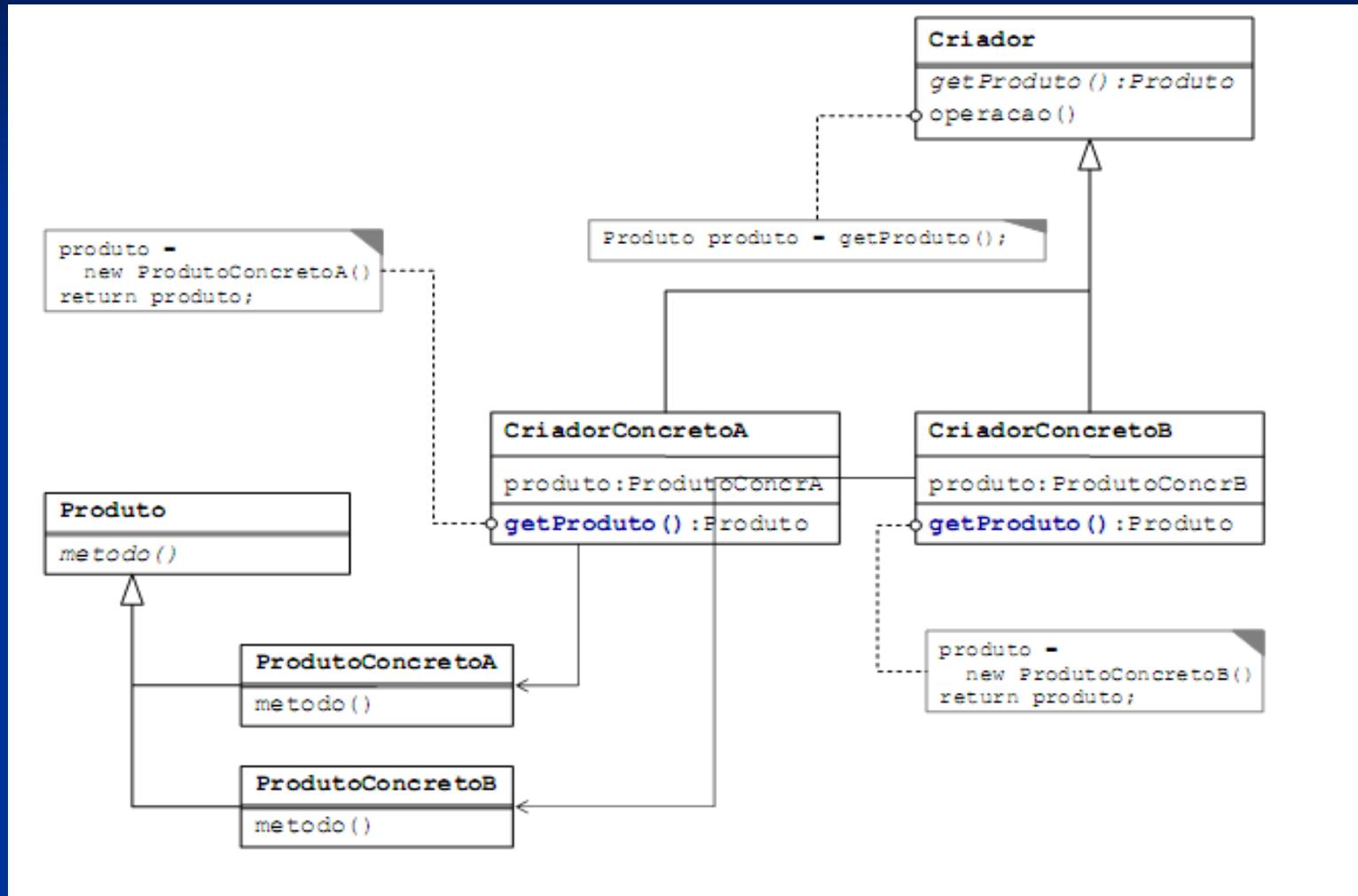
- O acesso a um objeto concreto será através da interface conhecida através de sua superclasse, mas o cliente também não quer (ou não pode) saber qual implementação concreta está usando.



Como Implementar?

- É possível criar um objeto sem ter conhecimento algum de sua classe concreta?
 - Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente.
 - FactoryMethod define uma interface comum para criar objetos.
 - O objeto específico é determinado nas diferentes implementações dessa interface.
 - O cliente do FactoryMethod precisa saber sobre implementações concretas do objeto criador do produto desejado.

Estrutura de Factory Method



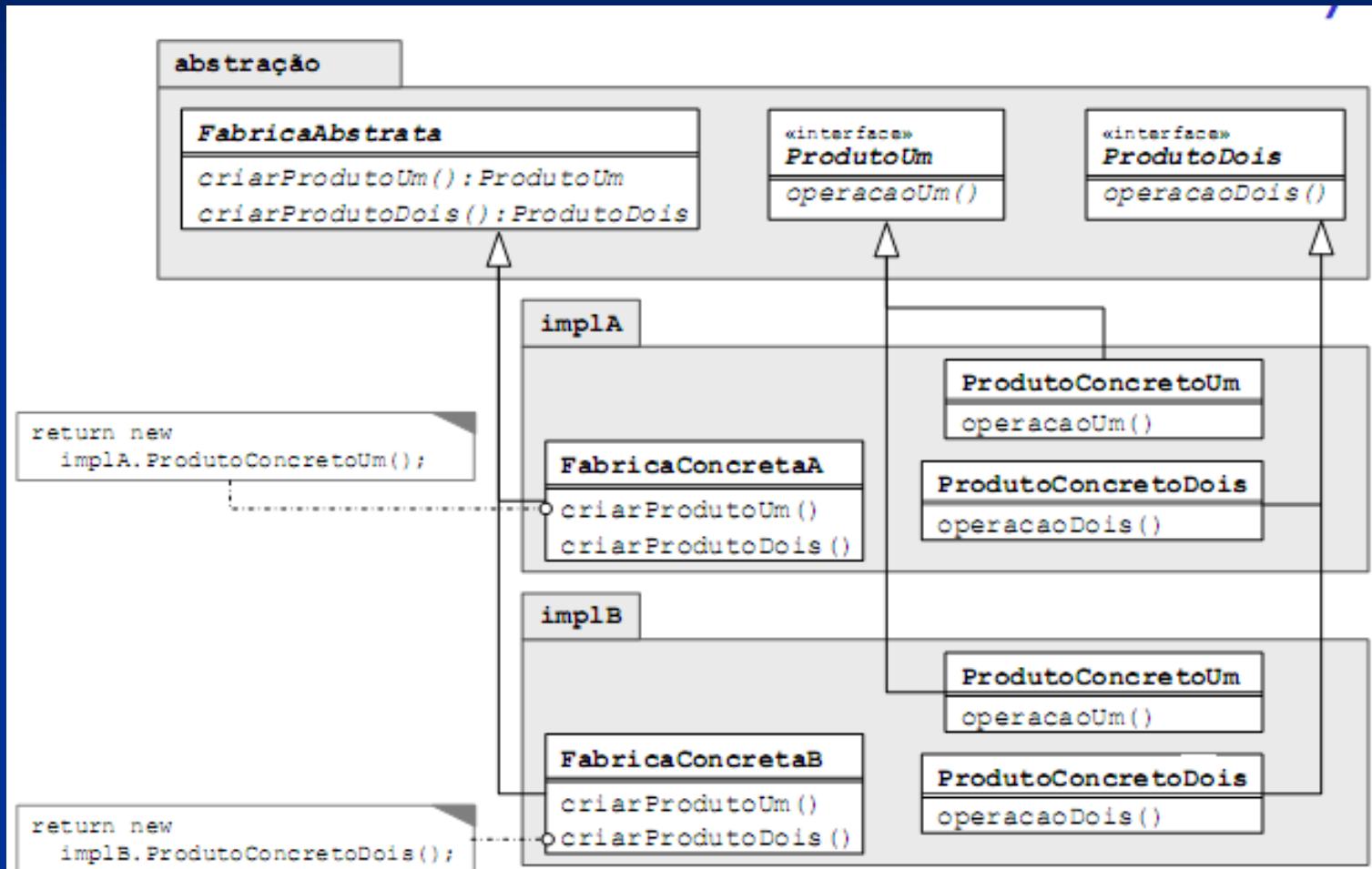
Abstract Factory

- "Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas." [GoF].

Problema

- Criar uma família de objetos relacionados sem conhecer suas classes concretas.

Estrutura do Abstract Factory

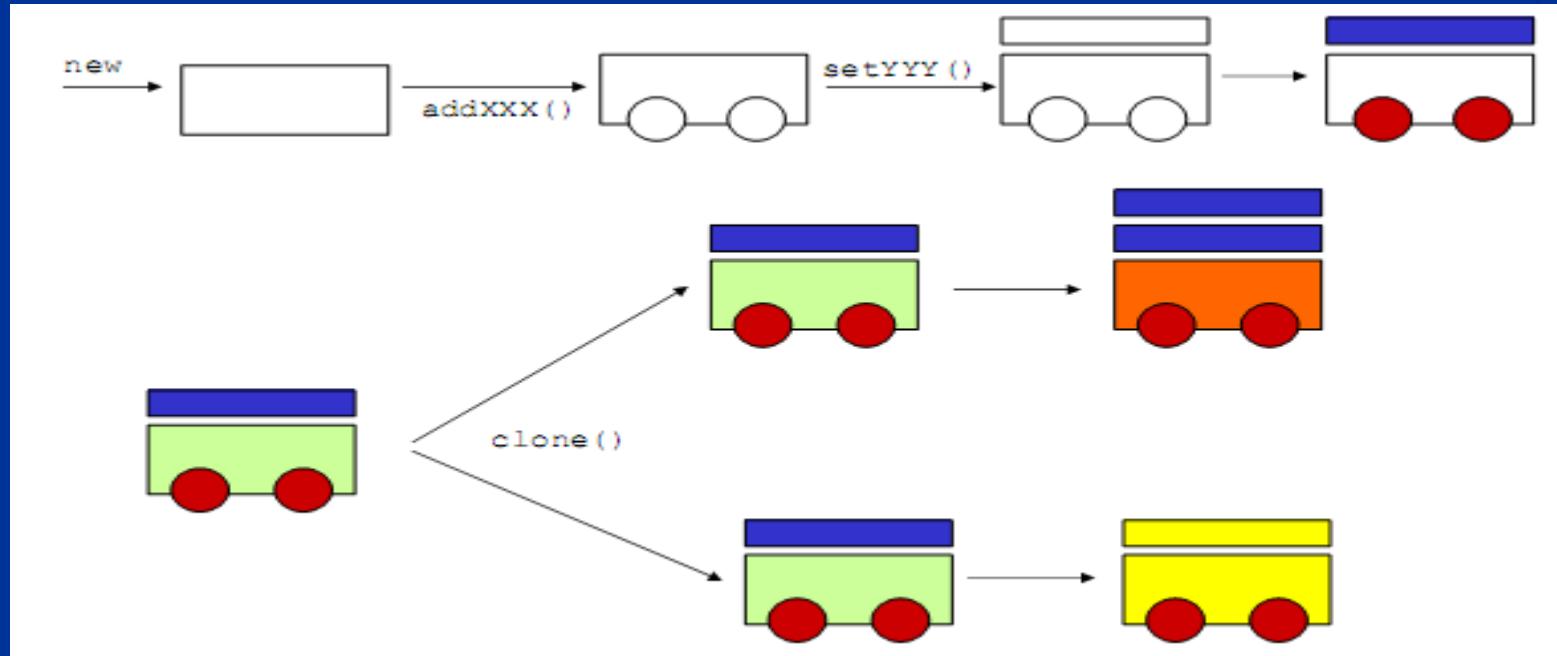


Prototype

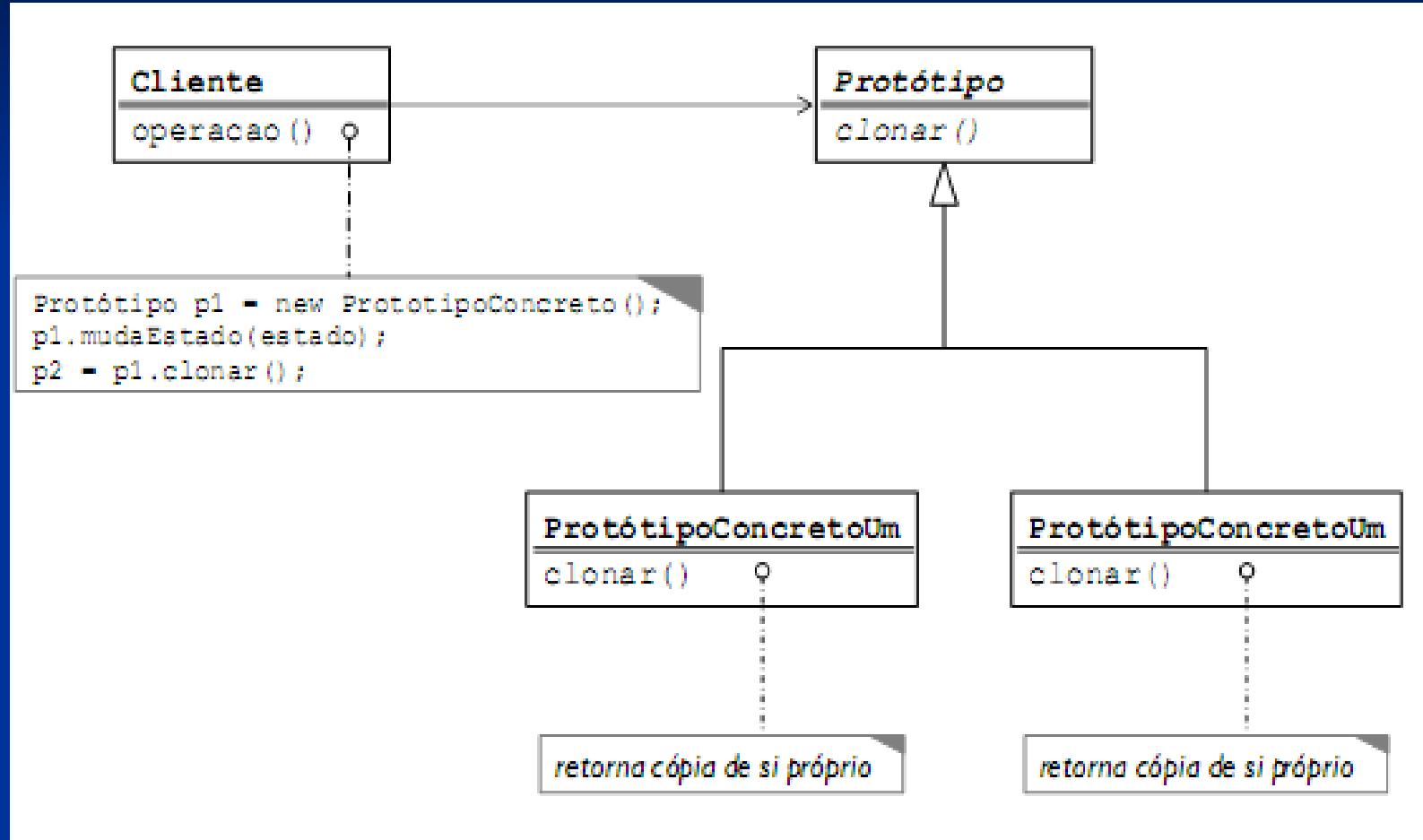
- "Especificar os tipos de objetos a serem criados usando uma instância como protótipo e criar novos objetos ao copiar este protótipo." [GoF]

Problema

- Criar um objeto novo, mas aproveitar o estado previamente existente em outro objeto.



Estrutura Prototype



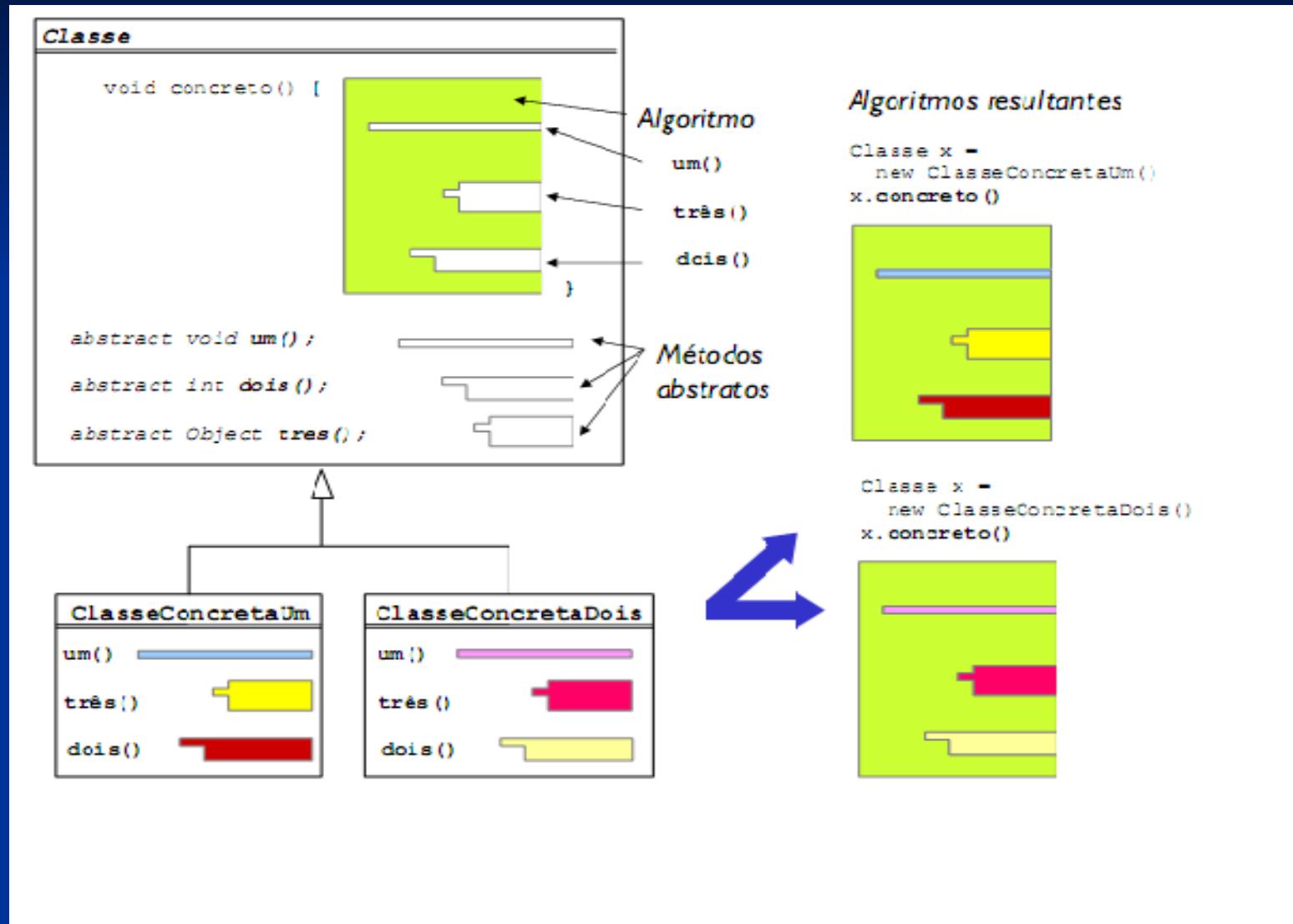
Resumo

- O padrão Prototype permite que um cliente crie novos objetos ao copiar objetos existentes.
- Uma vantagem de criar objetos deste modo é poder aproveitar o estado existente de um objeto.

Template Method

- "Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]

Problema



Solução:Template Method

- • O que é um Template Method
 - Um Template Method define um algoritmo em termos de operações abstratas que subclasses sobreponem para oferecer comportamento concreto
- Quando usar?
 - Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar.

Template Method em Java

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

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

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

State

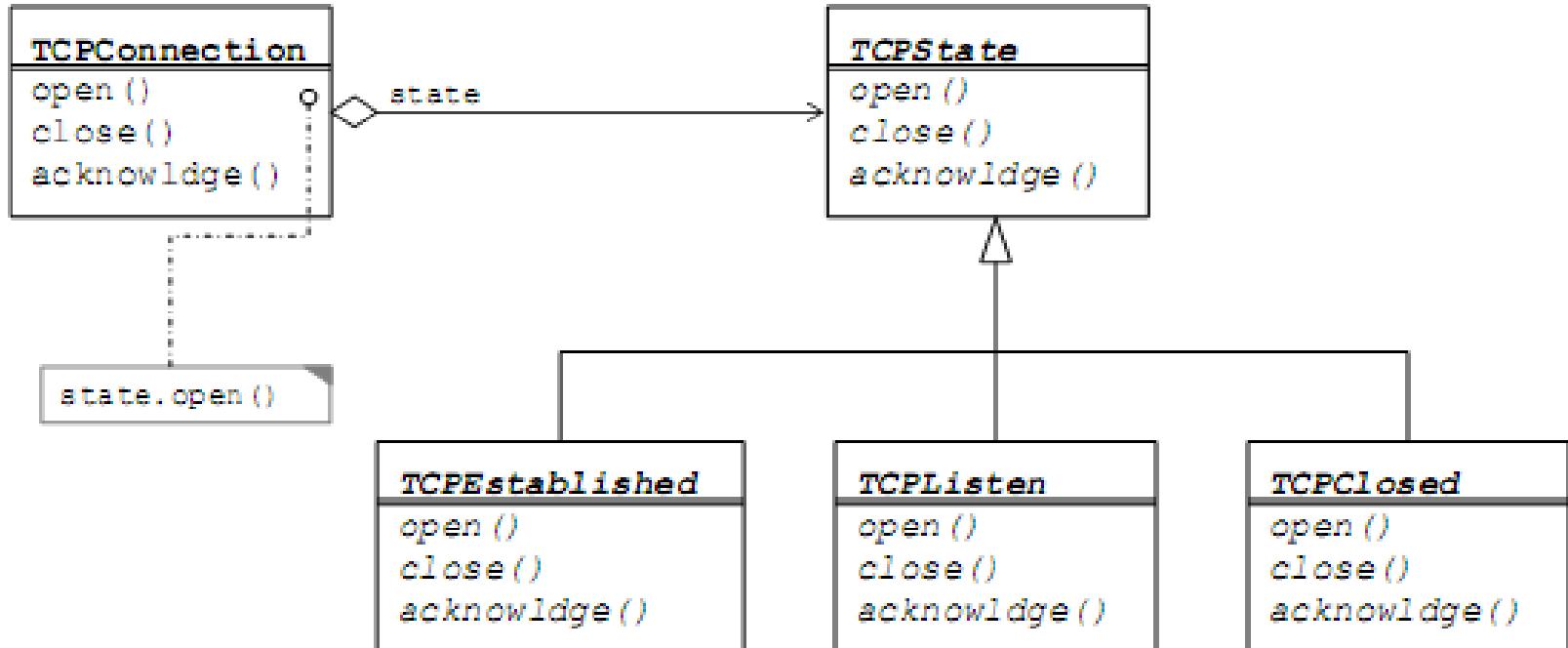
- "Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar. O objeto irá aparentar mudar de classe."
[GoF]

Problema

- Objetivo: usar objetos para representar estados e polimorfismo para tornar a execução de tarefas dependentes de estado transparentes.



Exemplo

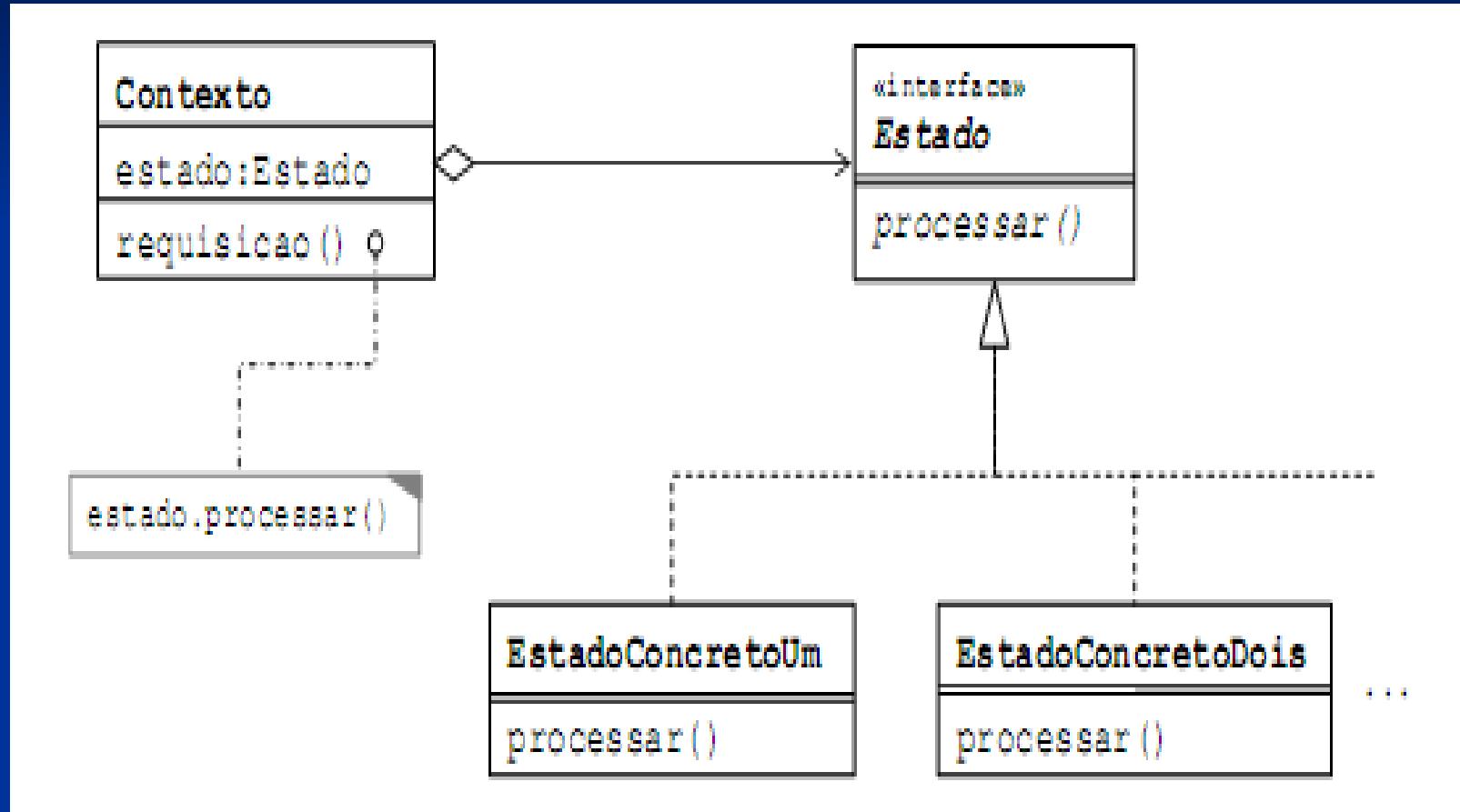


Sempre que a aplicação mudar de estado, o objeto `TCPConnection` muda o objeto `TCPSState` que está usando

Estrutura do State

- Contexto: define a interface de interesse aos clientes mantém uma instância de um EstadoConcreto que define o estado atual.
- Estado:define uma interface para encapsular o comportamento associado com um estado particular do contexto.
- EstadoConcreto: Implementa um comportamento associado ao estado do contexto.

Estrutura State(Cont..)



Strategy

- "Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis. Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam." [GoF]

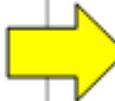
Problema

- Várias estratégias, escolhidas de acordo com opções ou condições.

```
if (guerra && inflação > META) {  
    doPlanoB();  
}  
else if (guerra && recessão) {  
    doPlanoC();  
}  
else {  
    doPlanejado();  
}
```

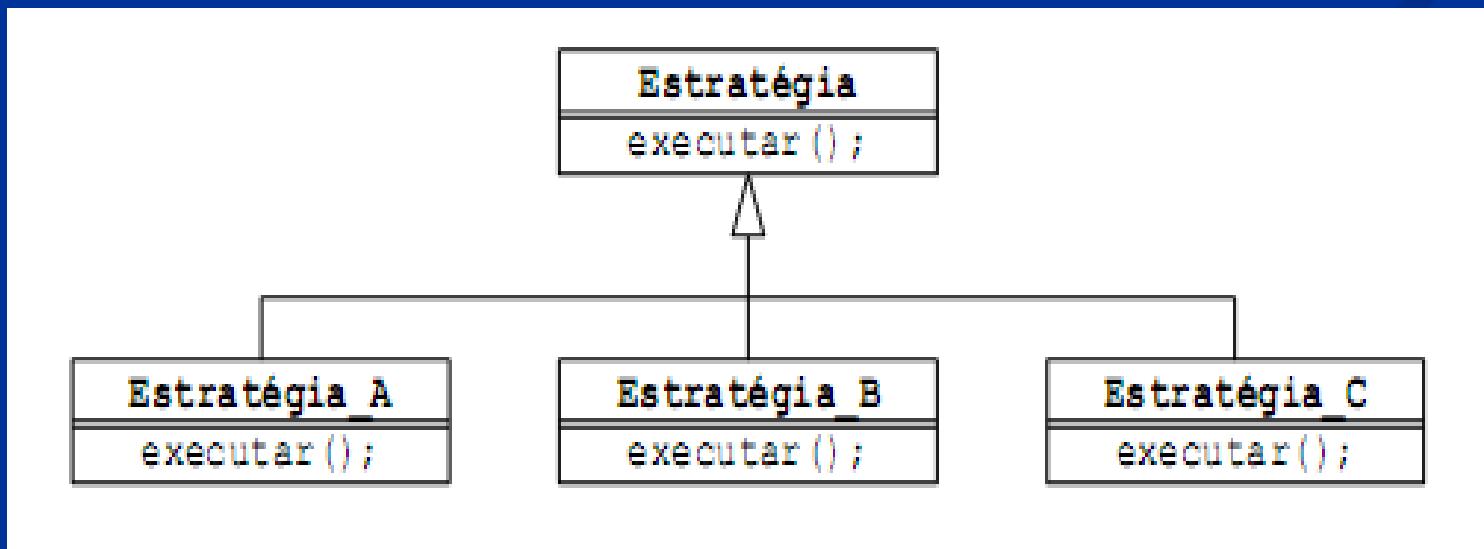
```
if (guerra && inflação > META) {  
    plano = new Estrategia_C();  
}  
else if (guerra && recessão) {  
    plano = new Estrategia_B();  
}  
else {  
    plano = new Estrategia_A();  
}
```

```
plano.executar();
```



Problema(cont..)

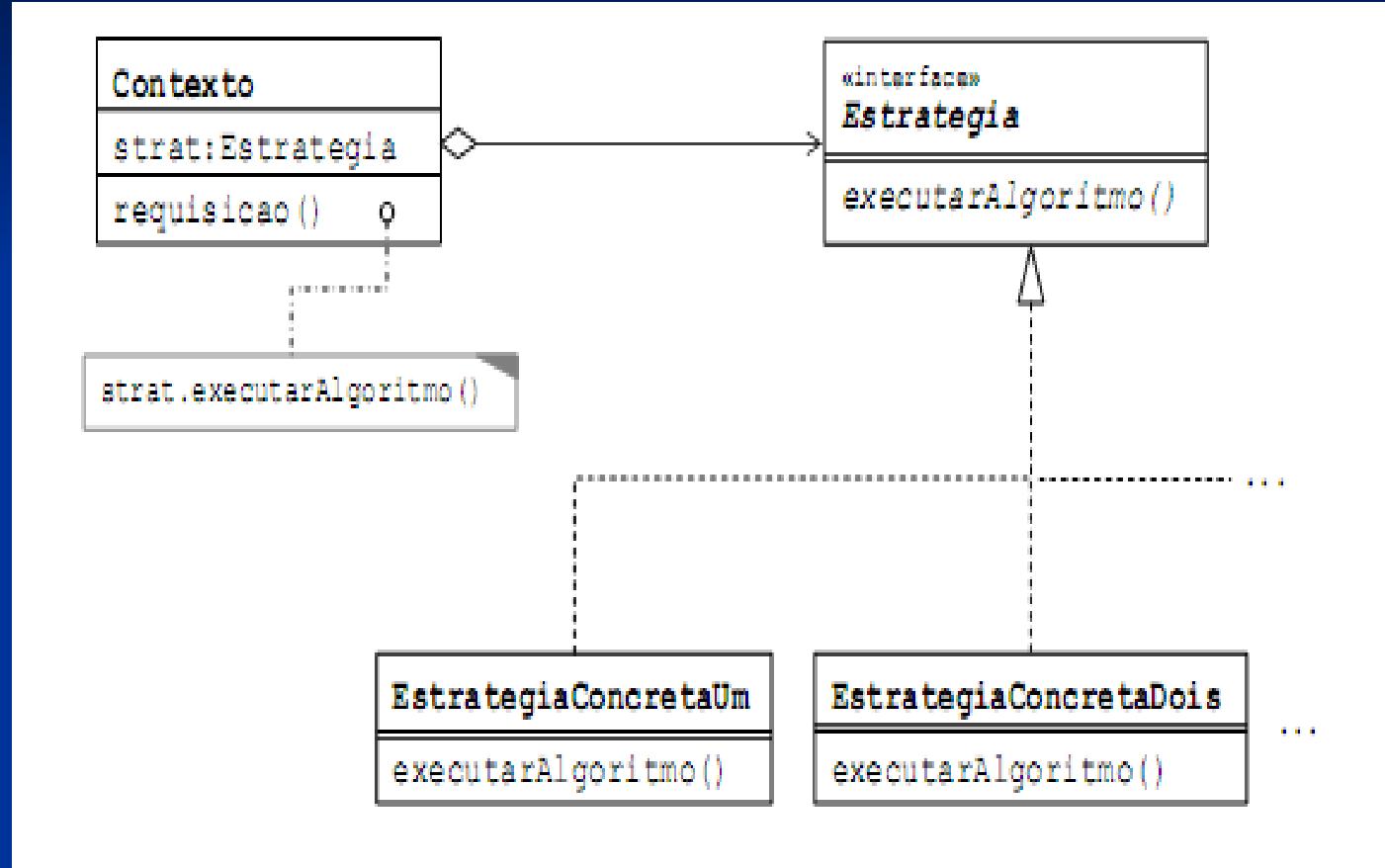
- Idêntico a state na implementação. Diferente na intenção!



Estrutura do Strategy

- Um contexto repassa requisições de seus clientes para sua estratégia. Clientes geralmente criam e passam uma EstrategiaConcreta para o contexto. Depois, clientes interagem apenas com o contexto.
- Estrategia e Contexto interagem para implementar o algoritmo escolhido. Um contexto pode passar todos os dados necessários ou uma cópia de si próprio.

Estrutura Strategy(cont..)



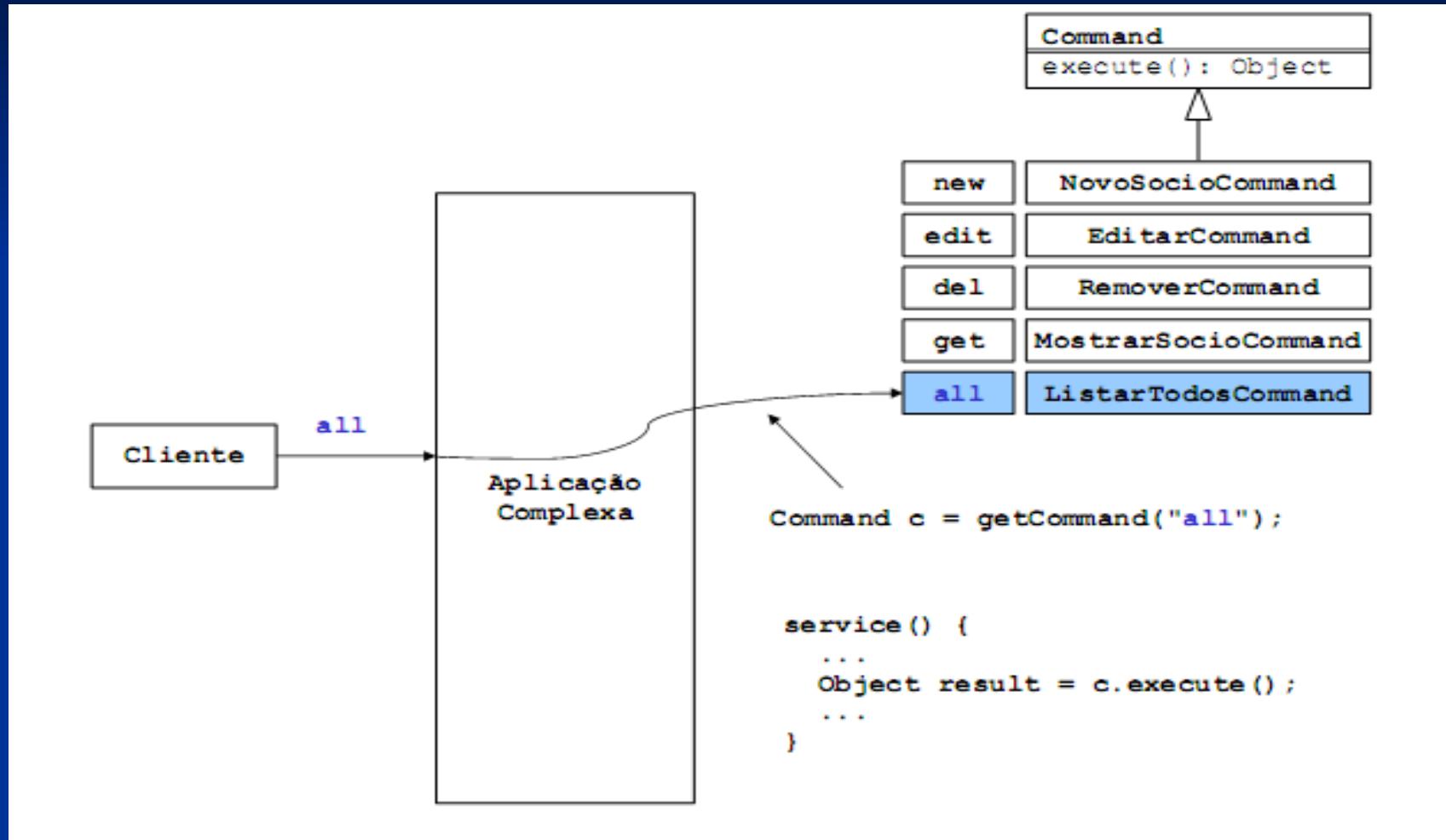
Quando Usar?

- Quando classes relacionadas forem diferentes apenas no seu comportamento.
 - Strategy oferece um meio para configurar a classe com um entre vários comportamentos.
- Quando você precisar de diferentes variações de um mesmo algoritmo.
- Quando um algoritmo usa dados que o cliente não deve conhecer.
- Quando uma classe define muitos comportamentos, e estes aparecem como múltiplas declarações condicionais em suas operações.

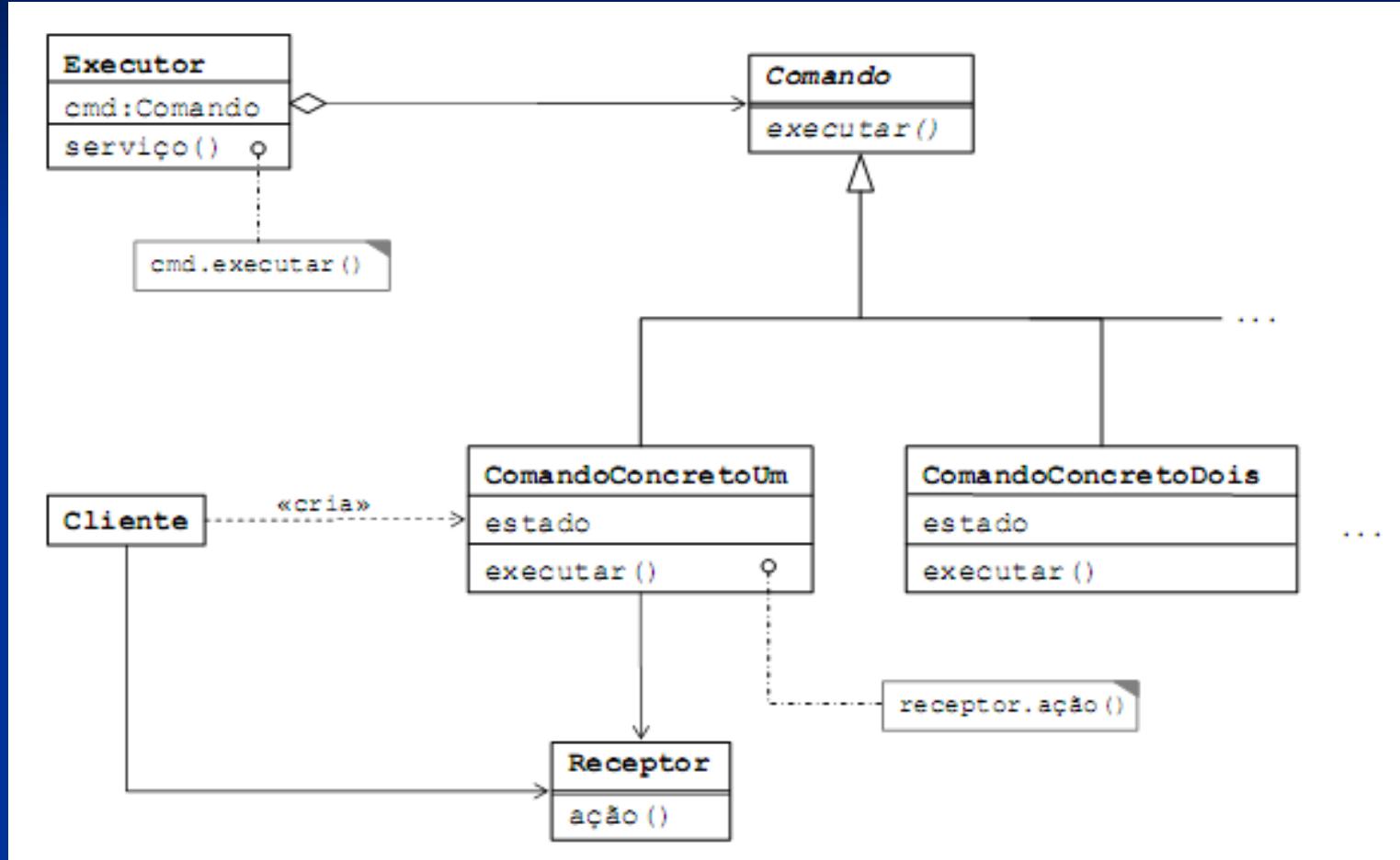
Command

- "Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]

Problema



Estrutura do Command



Iterator

- "Prover uma maneira de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação interna." [GoF]

Problema

Tipo de
referência é
genérico
↓

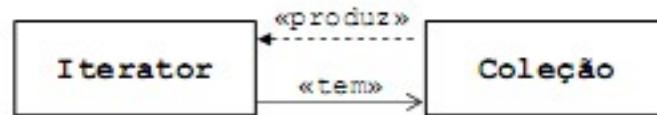
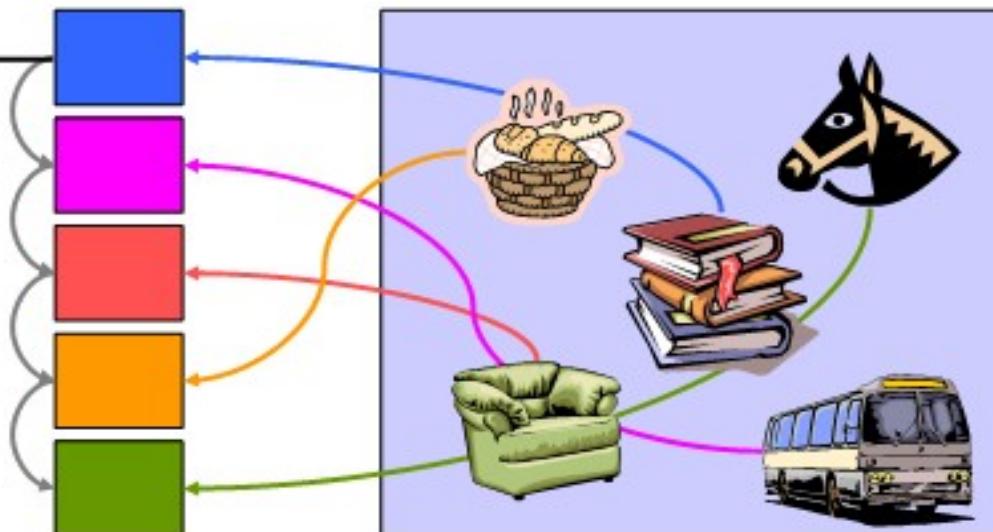
`Object o =
iterator.next()`



`iterator.hasNext() ?`

Iterator

Coleção arbitrária de objetos
(array, hashmap, lista, conjunto,
pilha, tabela, ...)



Para que serve?

- Iterators servem para acessar o conteúdo de um agregado sem expor sua representação interna.
- Oferece uma interface uniforme para atravessar diferentes estruturas agregadas
- Iterators são implementados nas coleções do Java. É obtido através do método iterator() de Collection, que devolve uma instância de java.util.Iterator.
- Interface java.util.Iterator:

```
package java.util;  
public interface Iterator<E> {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

- iterator() é um exemplo de Factory Method

Decorator

- "Anexar responsabilidades adicionais a um objeto dinamicamente. Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade." [GoF]

Problema(1)

- A primeira idéia é criar subclasses que implementem as funcionalidades desejadas
 - ListaSincronizada
 - ListaComEventos
 - ListaNaoModificavel

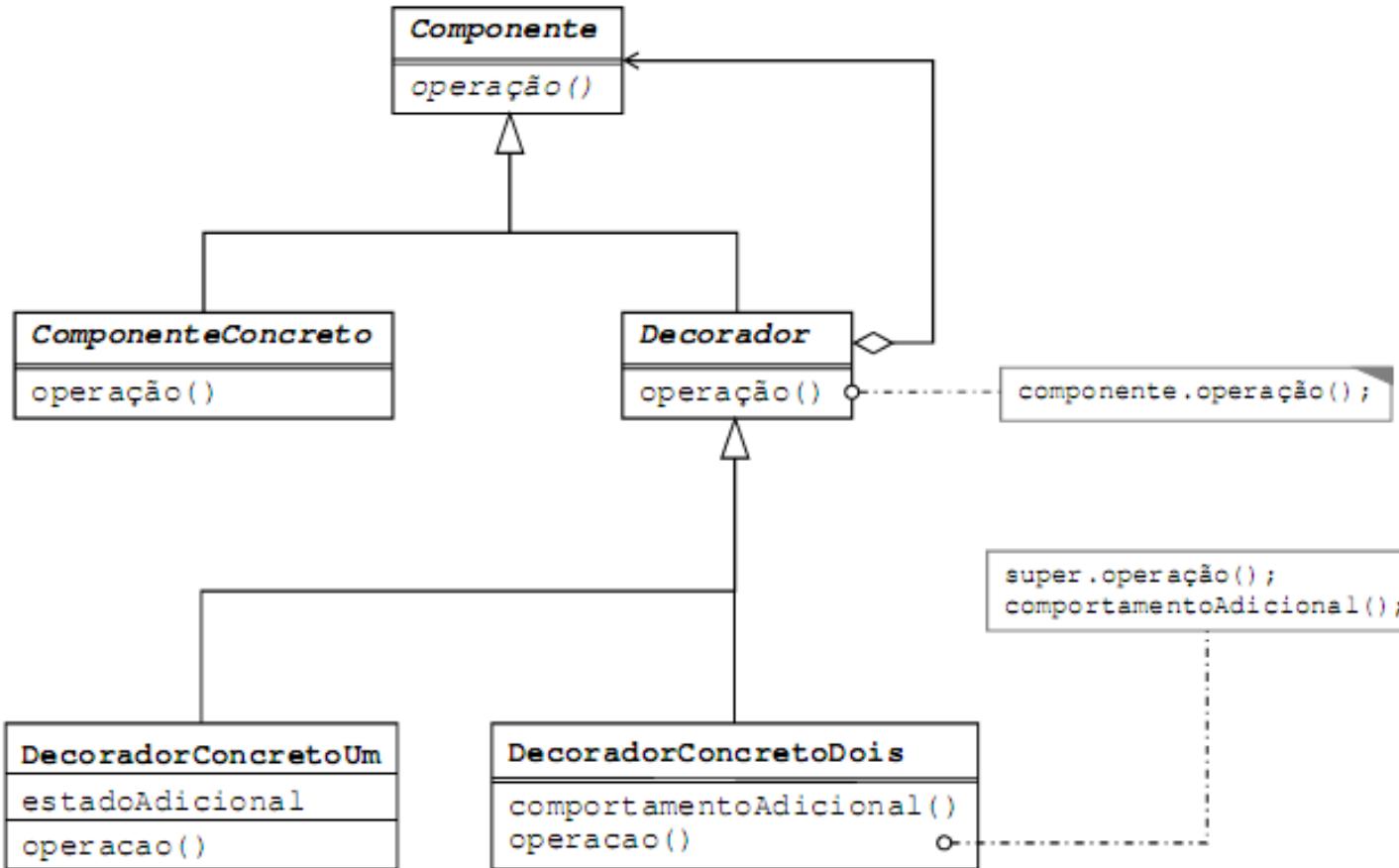
Problema(2)

- Mas e se desejarmos mais de uma funcionalidade ao mesmo tempo?
- Nesse caso teremos uma explosão de classes. As novas classes (além das 3 anteriores) poderiam ser chamadas:
 - ListaNaoModificavelSincronizada
 - ListaComEventosNaoModificavel
 - ListaComEventosSincronizada
 - ListaComEventosNaoModificavelSincronizada

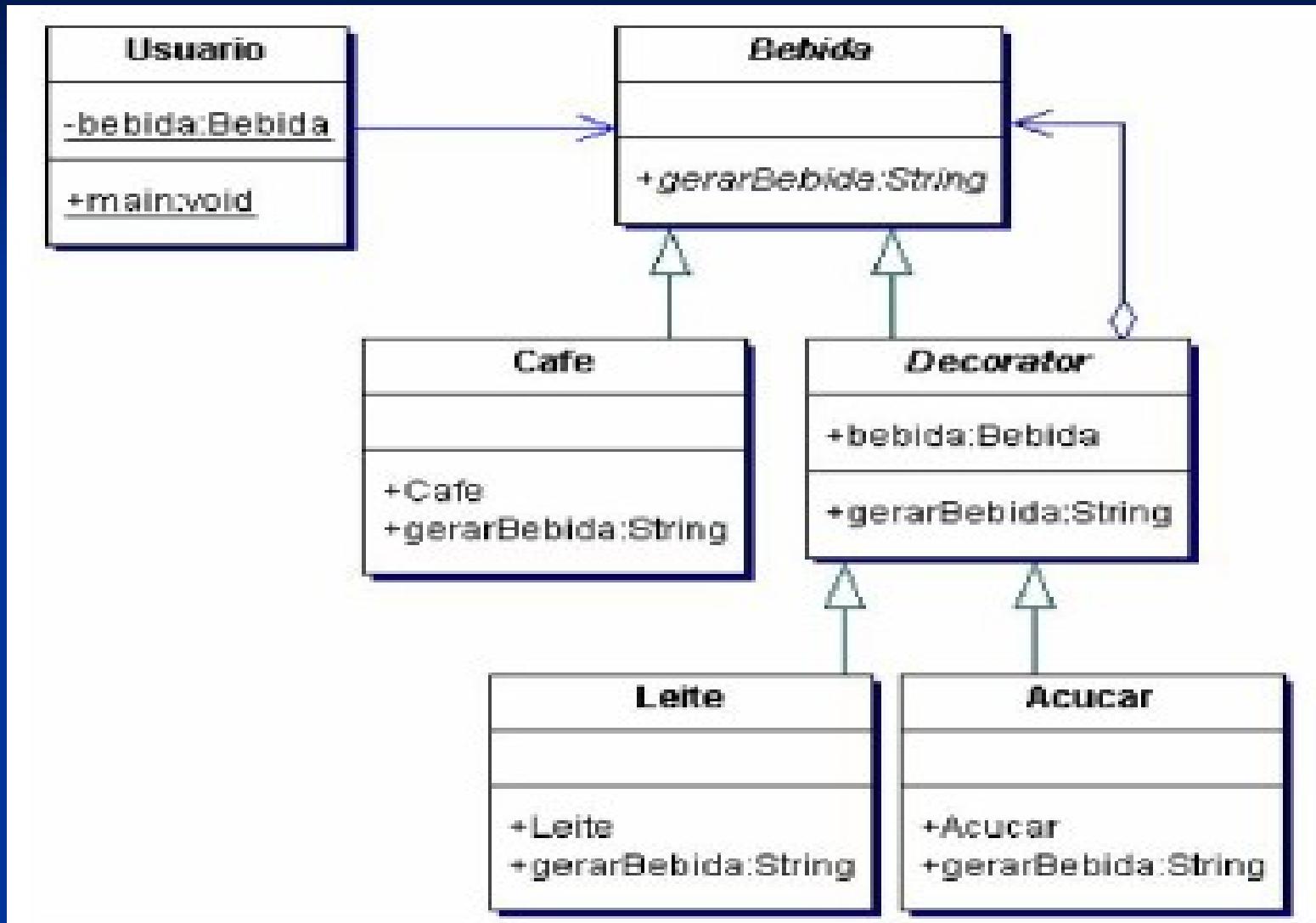
Problema(3)

- Necessidade de adicionar responsabilidades aos objetos, com a impossibilidade de criar extensões das subclasses;
- Às vezes é até possível realizar um grande número de extensões independentes, mas podem causar uma explosão de subclasses para suportar todas as combinações;

Estrutura



Exemplo



Exemplo c/código Fonte

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

```
public class Leite extends Decorator
{
    public Leite(Bebida comp) {
        bebida = comp;
    }
    public String gerarBebida()
    {
        String descricao = null;
        if (bebida != null)
            descricao =
                bebida.gerarBebida();
        descricao = descricao +
                    "adicionando leite,
";
        return descricao;
    }
}
```

Saída

cafe adicionando acucar, adicionando leite, adicionando chocolate,

Quando usar???

- Utilizado para adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, isto é, sem afetar outros objetos, da mesma forma, quando se quer retirar responsabilidades;
- Quando a utilização de heranças para a implementação do mesmo afetará a flexibilidade do sistema;

Padrão Arquitetural: MVC

- Padrões Arquiteturais: expressam o esquema ou organização estrutural fundamental de sistemas de software ou hardware
- Padrões de Projeto: Define soluções de problemas de projetos de software orientado a objetos.

MVC

- Dividir um componente ou uma aplicação interativa (subsistema) em três partes lógicas: o modelo (model) que contém funcionalidades e dados; visões (views) mostram informações para o usuário; controladores (controllers) manipulam os eventos das entradas. Um mecanismo de propagação de mudanças garante a consistência entre a interface do usuário e o modelo.

Problema

- Estender a funcionalidade de uma aplicação através apenas da modificação dos menus, da visão do usuário.

Solução

- O padrão MVC divide a aplicação em três camadas: Entrada (View), Processamento (Controller) e Saída (Model).
- Utilizar o padrão Observer e estendê-lo para permitir o controle das janelas baseado-em-eventos. O Padrão MVC estende o Observer incorporando um elemento controlador (Controller).

Componentes do MVC(1)

- O Modelo diz respeito ao gerenciamento da informação e ao comportamento da aplicação. O Modelo seria uma mera representação do conteúdo do banco de dados ou entidades de domínio e pelas regras de negócio intrínsecas a essas entidades.
- A Visão é responsável por apresentar as entidades de domínio ao usuário, constituindo a parte visível do sistema.

Componentes do MVC(2)

- O Controle garante o total desacoplamento entre essas duas partes principais da aplicação. O Controle interpreta as ações do usuário provenientes da Visão e comanda a execução das regras de negócio contidas no Modelo, além disso, comanda a Visão para que ela apresente adequadamente a informação ao usuário.

Componentes do MVC(3)

- View - Não está preocupada em como a informação foi obtida ou onde ela foi obtida apenas exibe a informação
 - Inclui os elementos de exibição no cliente : HTML , XML , ASP , Applets .
 - É a camada de interface com o usuário.
 - É usada para receber a entrada de dados e apresentar o resultado.

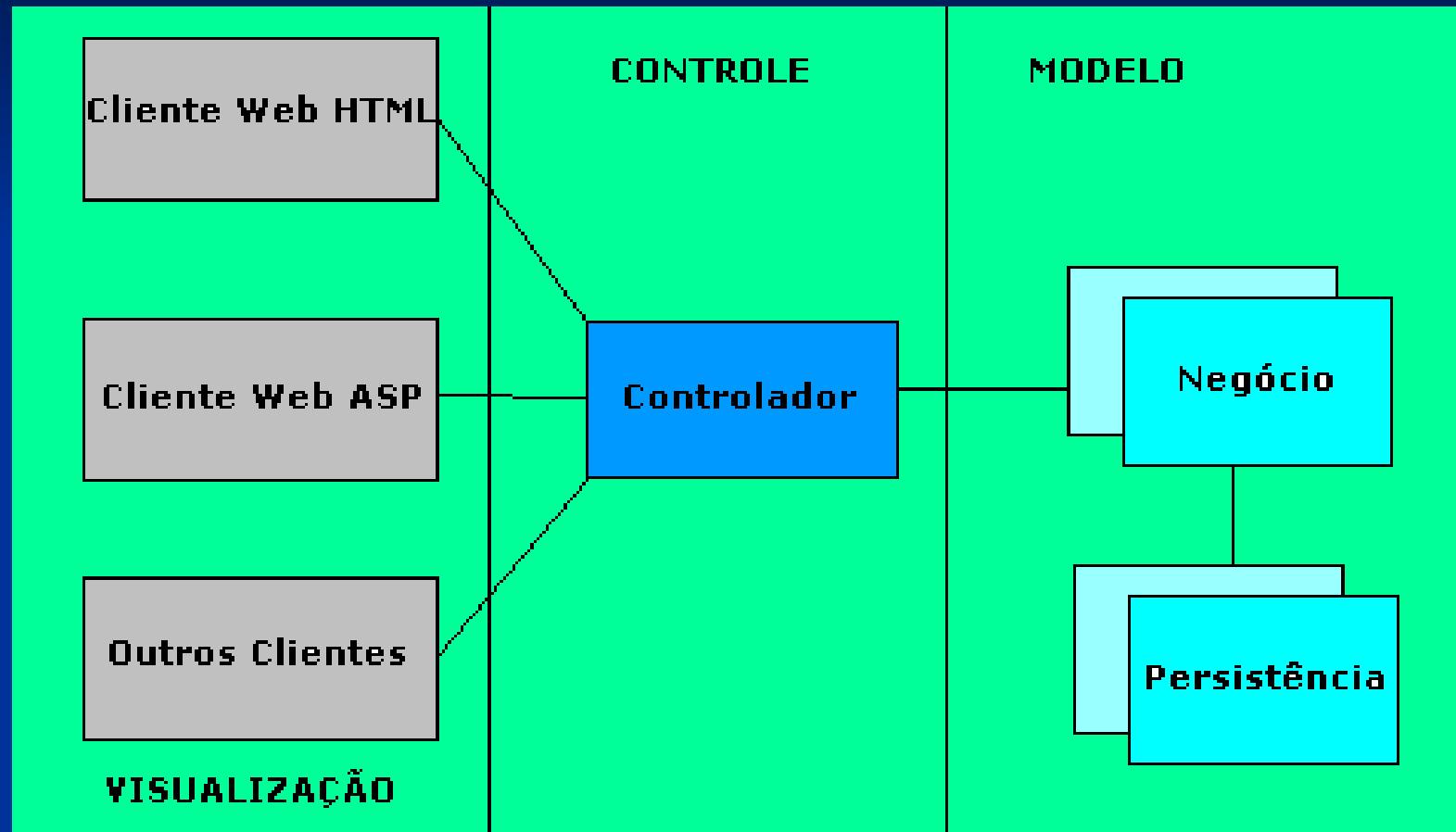
Componentes do MVC(4)

- Model - É o coração da aplicação . Responsável por tudo que a aplicação vai fazer.
 - modela os dados e o comportamento por atrás do processo de negócios.
 - se preocupa apenas com o armazenamento , manipulação e geração de dados.
 - É um encapsulamento de dados e de comportamento independente da apresentação.

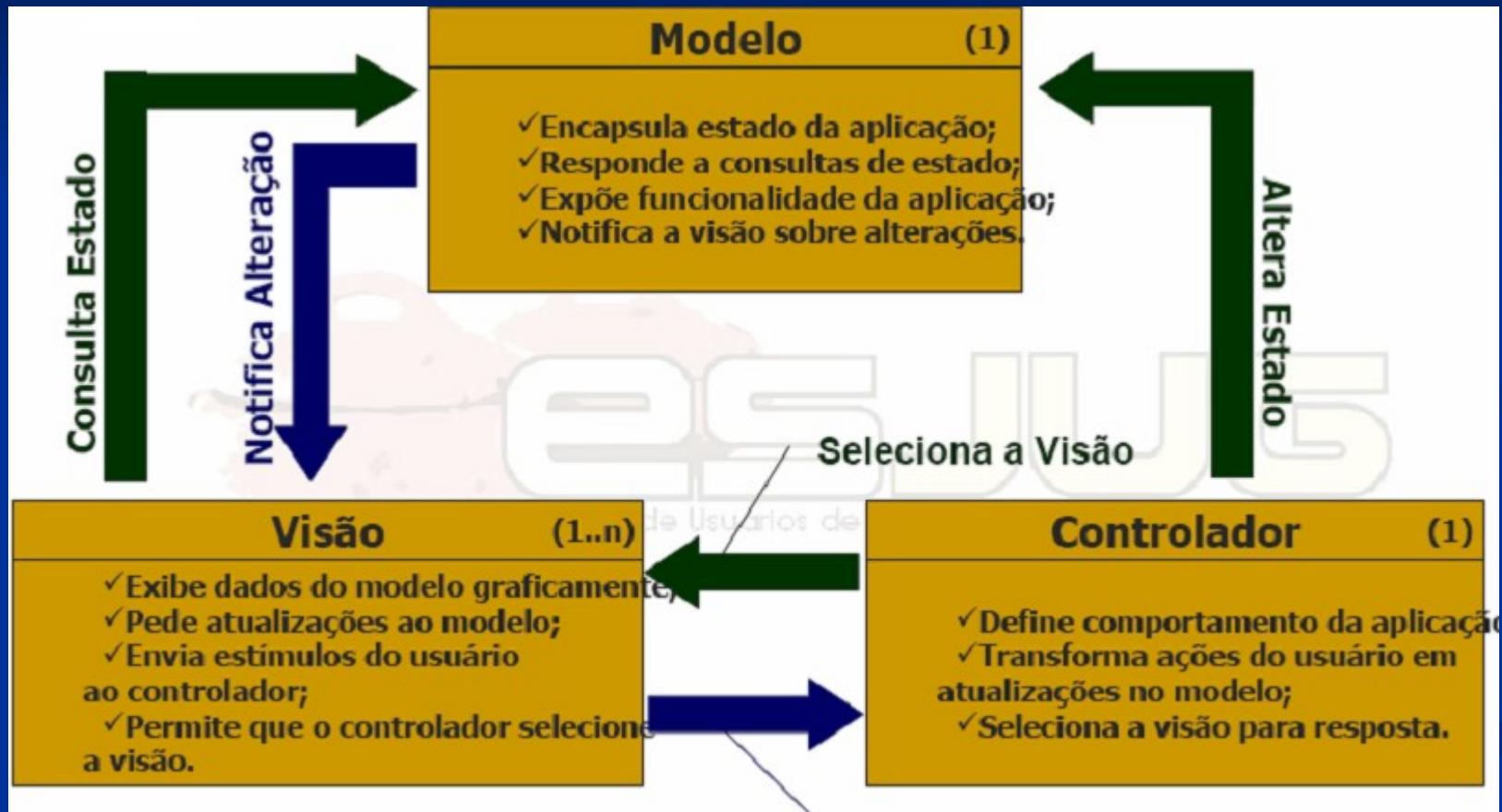
Componentes do MVC(5)

- Camada de Controle - determina o fluxo da apresentação servindo como uma camada intermediária entre a camada de apresentação e a lógica.
 - controla e mapeia as ações.

Estrutura MVC(1)



Estrutura MVC(2)



Obrigado!!!!

E-mail: ricferal@gmail.com