Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Vítor E. Silva Souza
vitorsouza@gmail.com

http://www.javablogs.com.br/page/engenho
http://esjug.dev.java.net

Sobre o Instrutor

- Formação:
  - Graduação em Ciência da Computação, com ênfase em Engenharia de Software, pela Universidade Federal do Espírito Santo (UFES);
  - Mestrado em Informática (em andamento) na mesma instituição.
- Java:
  - Desenvolvedor Java desde 1999;
  - Especialista em desenvolvimento Web;
  - Autor do blog Engenho – www.javablogs.com.br/page/engenho.
- Profissional:
  - Consultor em Desenvolvimento de Software Orientado a Objetos – Engenho de Software Consultoria e Desenvolvimento Ltda.

Estrutura do Curso

Módulo 1	Introdução

Módulo 2	Padrões de Criação

Módulo 3	Padrões de Estrutura

Módulo 4	Padrões de Comportamento
Módulo 5	O Padrão Model-View-Controller

Conteúdo deste módulo

- Introdução;
- Chain of Responsibility;
- Command;
- Interpreter;
- Iterator;
- Mediator;
- Memento;
- Observer;
- State;
- Strategy;
- Template Method;
- Visitor;
- Conclusões.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Introdução

Introdução

- Descrevem padrões de comunicação entre objetos;
  - Fluxos de comunicação complexos;
  - Foco na interconexão entre objetos.
- Escopo de classe: herança;
- Escopo de objeto: composição.

Padrões de Comportamento

"Preocupam-se com algorítmos e a delegação de responsabilidades entre objetos."

Escopo de Classe	Interpreter
	Template Method
Escopo de Objeto	Chain of Responsibility
	Command
	Iterator
	Mediator
	Memento
	Observer
	State
	Strategy
	Visitor

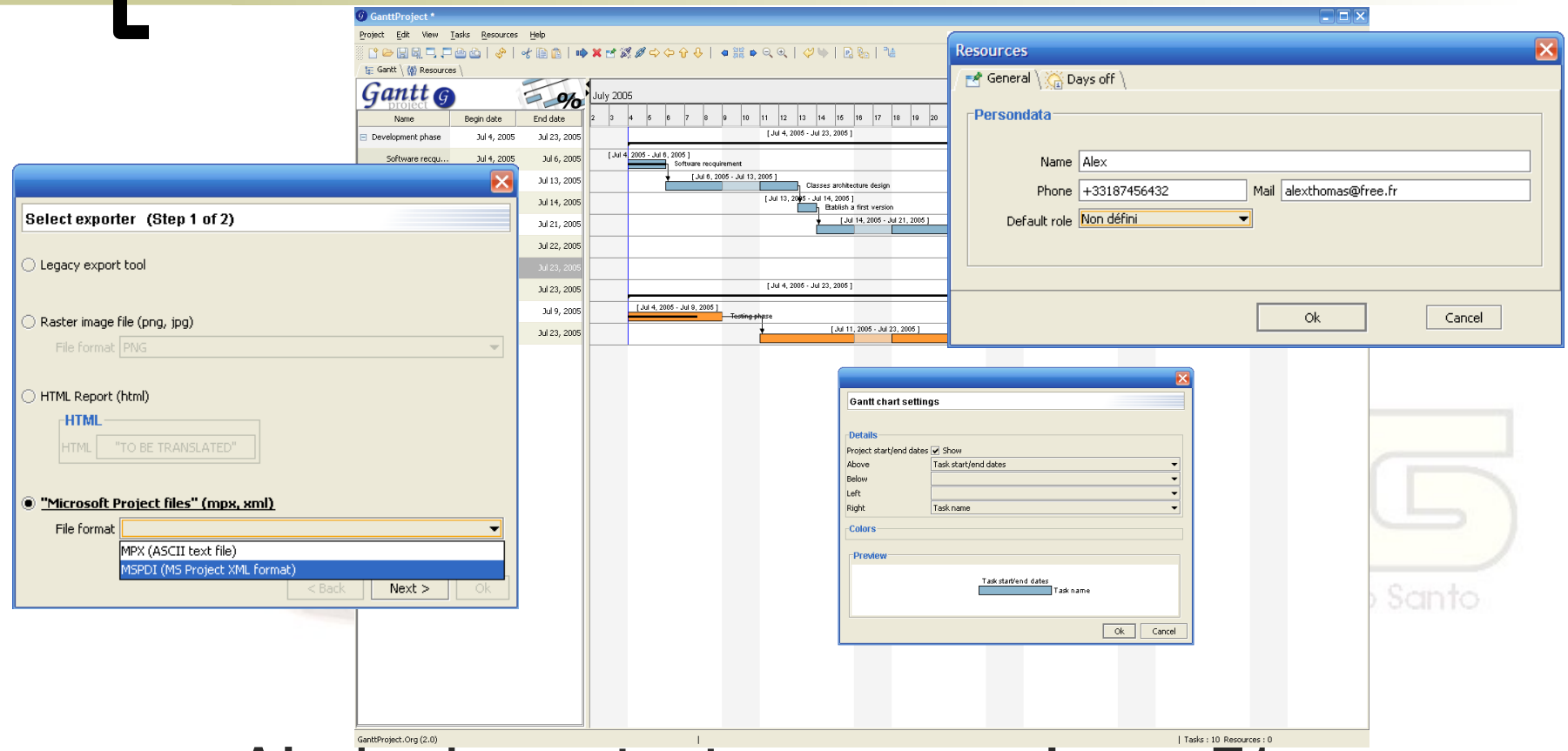
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Chain of Resposibility
(Cadeia de Responsabilidades)
Comportamento / Objeto

Descrição

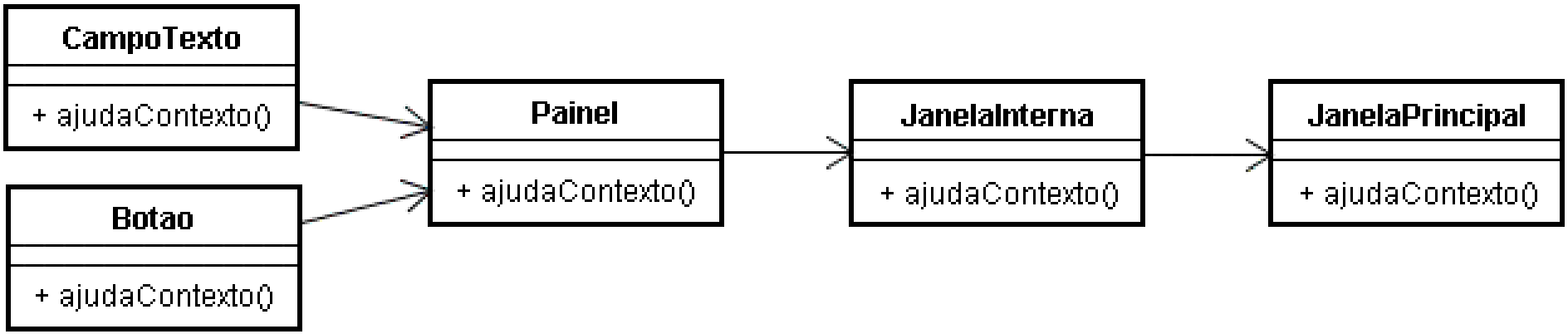
- Intenção:
  - Formar uma cadeia de objetos receptores e passar uma requisição pela mesma, dando a chance a mais de um objeto a responder a requisição ou colaborar de alguma forma na resposta.

O problema



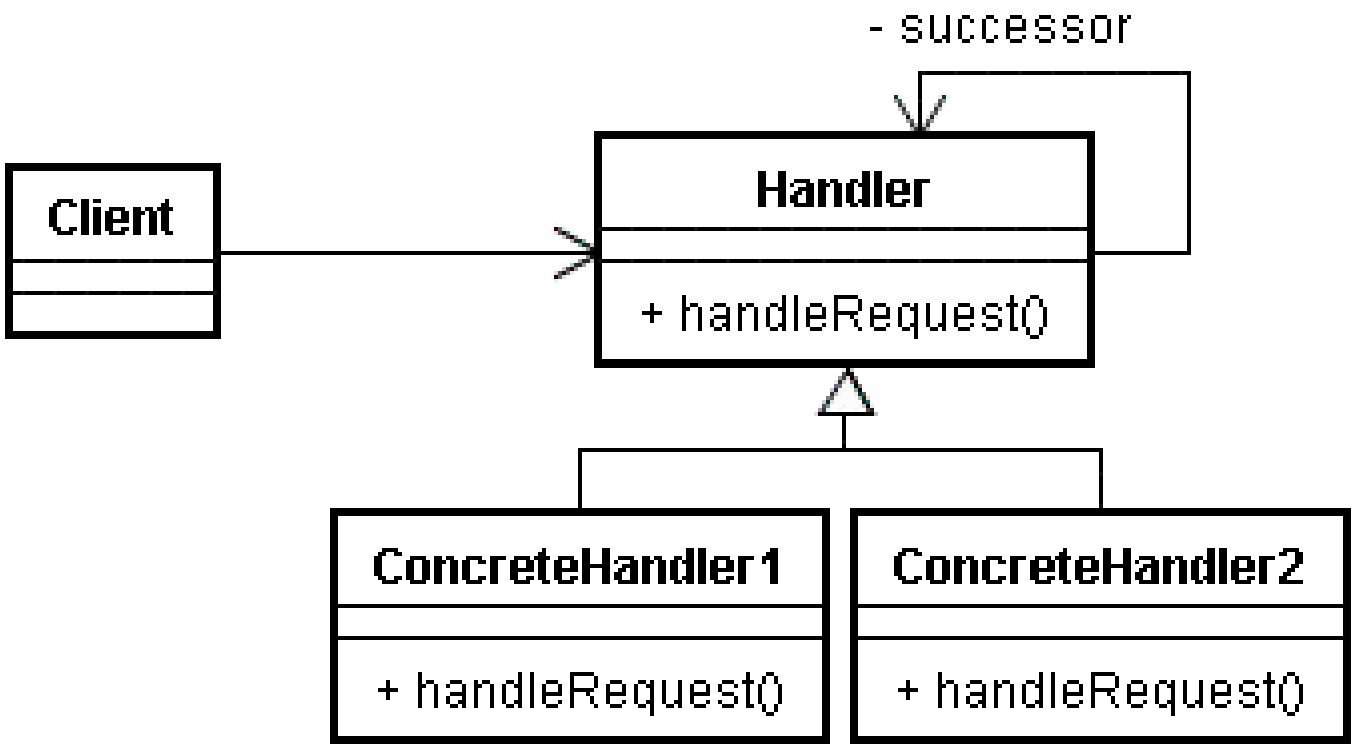
- Ajuda de contexto: ao pressionar F1 com o foco em um componente gráfico, qual componente acionará a ajuda?

A solução

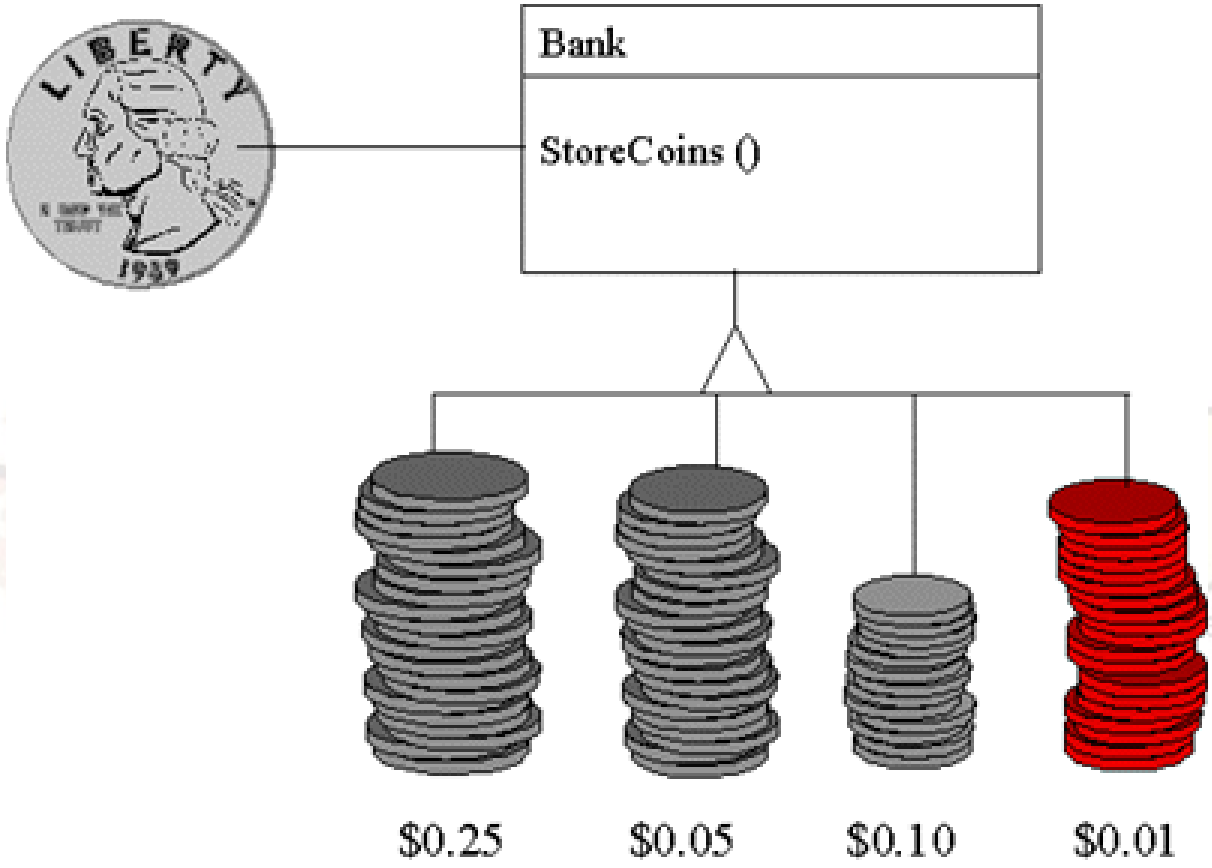


- Componentes colocados em cadeia na ordem filho -> pai;
- Se não há ajuda de contexto para o filho, ele delega ao pai e assim sucessivamente.

Estrutura


- successor
Client
Handler
+ handleRequest()
ConcreteHandler1
+ handleRequest()
ConcreteHandler2
+ handleRequest()

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- mais de um objeto pode responder a uma requisição e:
  - não se sabe qual a priori;
  - não se quer especificar o receptor explicitamente;
  - estes objetos são especificados dinamicamente.

Vantagens e desvantagens

- Acoplamento reduzido:
  - Não se sabe a classe ou estrutura interna dos participantes. Pode usar Mediator para desacoplar ainda mais.
- Delegação de responsabilidade:
  - Flexível, em tempo de execução.
- Garantia de resposta:
  - Deve ser uma preocupação do desenvolvedor!
- Não utilizar identidade de objetos.

Chain vs. Decorator

Chain of Responsibility	Decorator
Analogia com uma fila de filtros.	Analogia com camadas de um mesmo objeto.
Os objetos na cadeia são do "mesmo nível".	O objeto decorado é "mais importante". Os demais são opcionais.
O normal é quando um objeto atender, interromper a cadeia.	O normal é ir até o final.

Exemplos

- Os interceptadores (módulo 3: padrão Decorator) são encadeados num Chain of Responsibility;
  - Interceptors são um "meio termo" entre estes dois padrões.
- O framework WebWork utiliza a idéia dos interceptadores em cadeia.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Command
(Comando)
Comportamento / Objeto

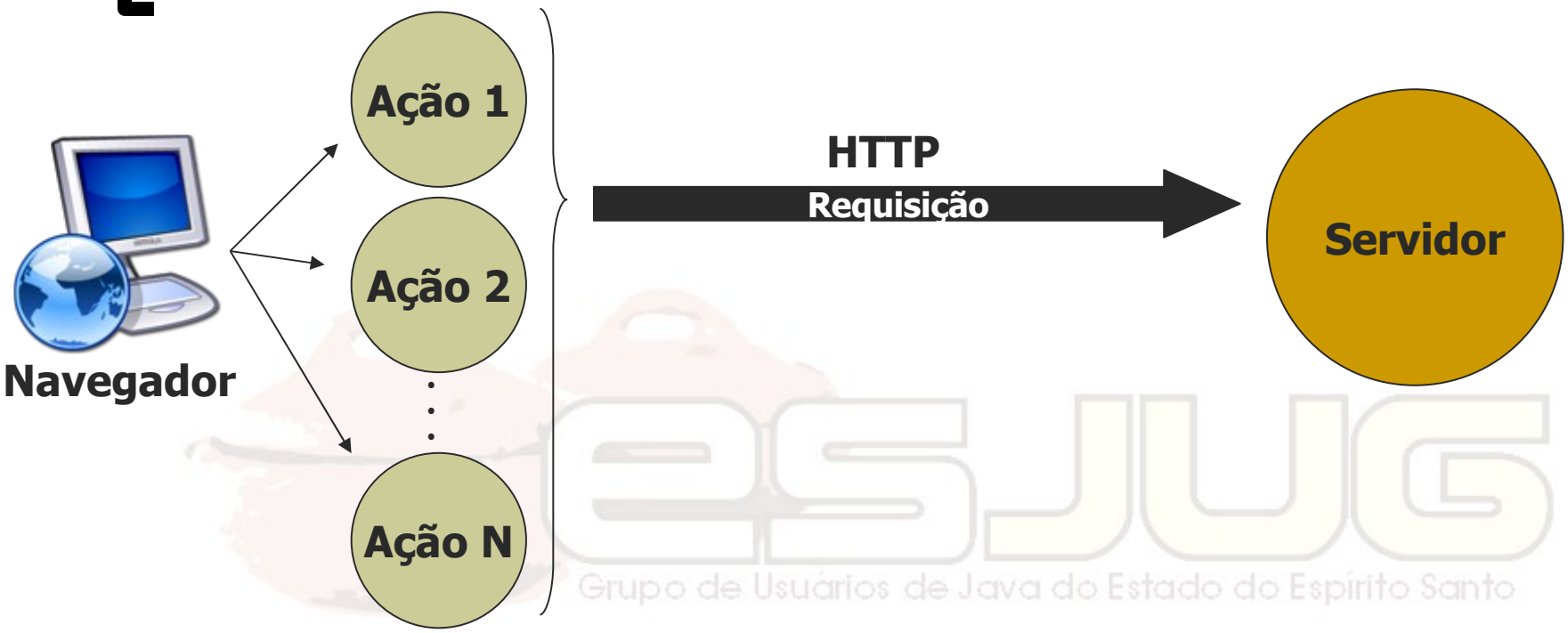
Descrição

- Intenção:
  - Encapsular uma requisição como um objeto, permitindo parametrização, enfileiramento, suporte a histórico, etc.
- Também conhecido como:
  - Action, Transaction.

O problema

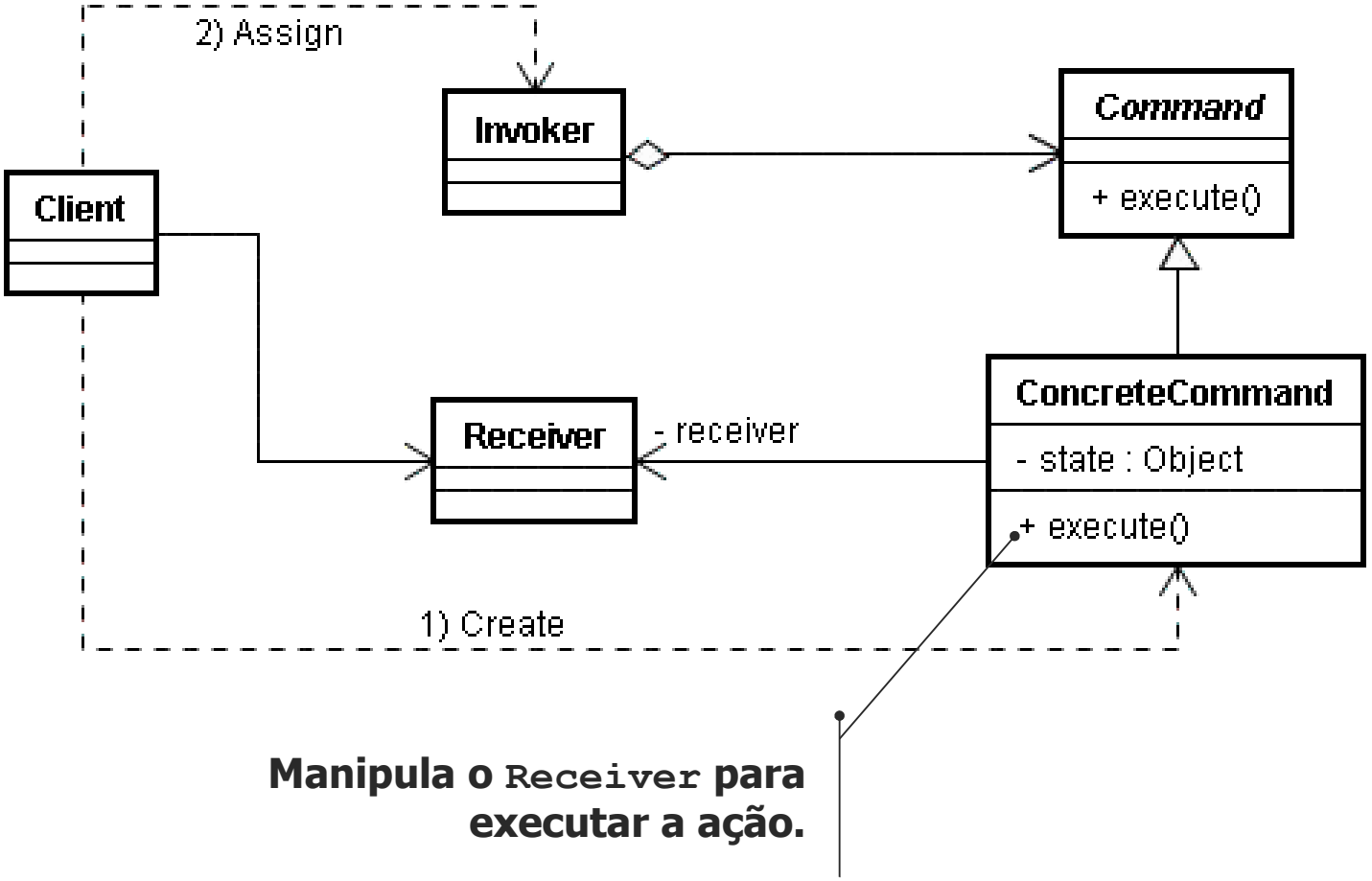
- A cada requisição:
  - Uma funcionalidade diferente deve ser executada;
  - Conjuntos de dados diferentes são enviados.

A solução

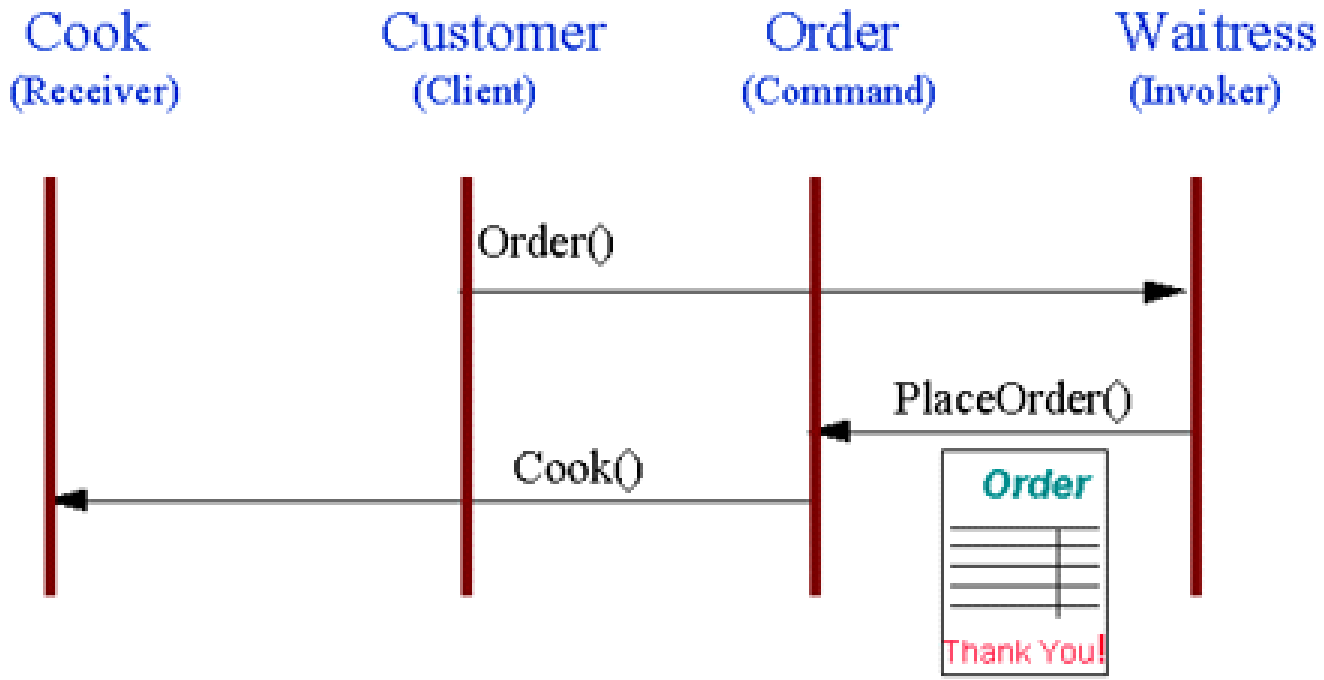


- Cliente escolhe uma ação que encapsula os dados e o que deve ser executado;
- Servidor executa esta ação.

Estrutura


2) Assign
Invoker
Command
+ execute()
Client
Receiver
- receiver
ConcreteCommand
- state : Object
+ execute()
1) Create
Manipula o Receiver para executar a ação.

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- quiser parametrizar ações genéricas;
- quiser enfileirar e executar comandos de forma assíncrona, em outro momento;
- quiser permitir o *undo* de operações, dando suporte a históricos;
- quiser fazer log dos comandos para refazê-los em caso de falha de sistema;
- quiser estruturar um sistema em torno de operações genéricas, como transações.

Vantagens e desvantagens

- Desacoplamento:
  - Objeto que evoca a operação e o que executa são desacoplados.
- Extensibilidade:
  - Comandos são objetos, passíveis de extensão, composição, etc.;
  - Pode ser usado junto com Composite para formar comandos complexos;
  - É possível definir novos comandos sem alterar nada existente.

Exemplos em Java

- O framework WebWork utiliza este padrão para atender requisições web, como mostrado na motivação;
- Eventos em Swing utilizam uma abordagem parecida, só não encapsulam a execução do comando.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Interpreter
(Interpretador)
Comportamento / Classe

Descrição

- Intenção:
  - Definir a gramática de uma linguagem e criar um interpretador que leia instruções nesta linguagem e interprete-as para realizar tarefas.

O problema

```
// Verificar se o e-mail é válido.
int pos = email.indexOf('@');
int length = email.length();
if ((pos > 0) && (length > pos)) {
    // Ainda verificar se tem ".com", etc.
}
else { /* E-mail inválido. */ }
```

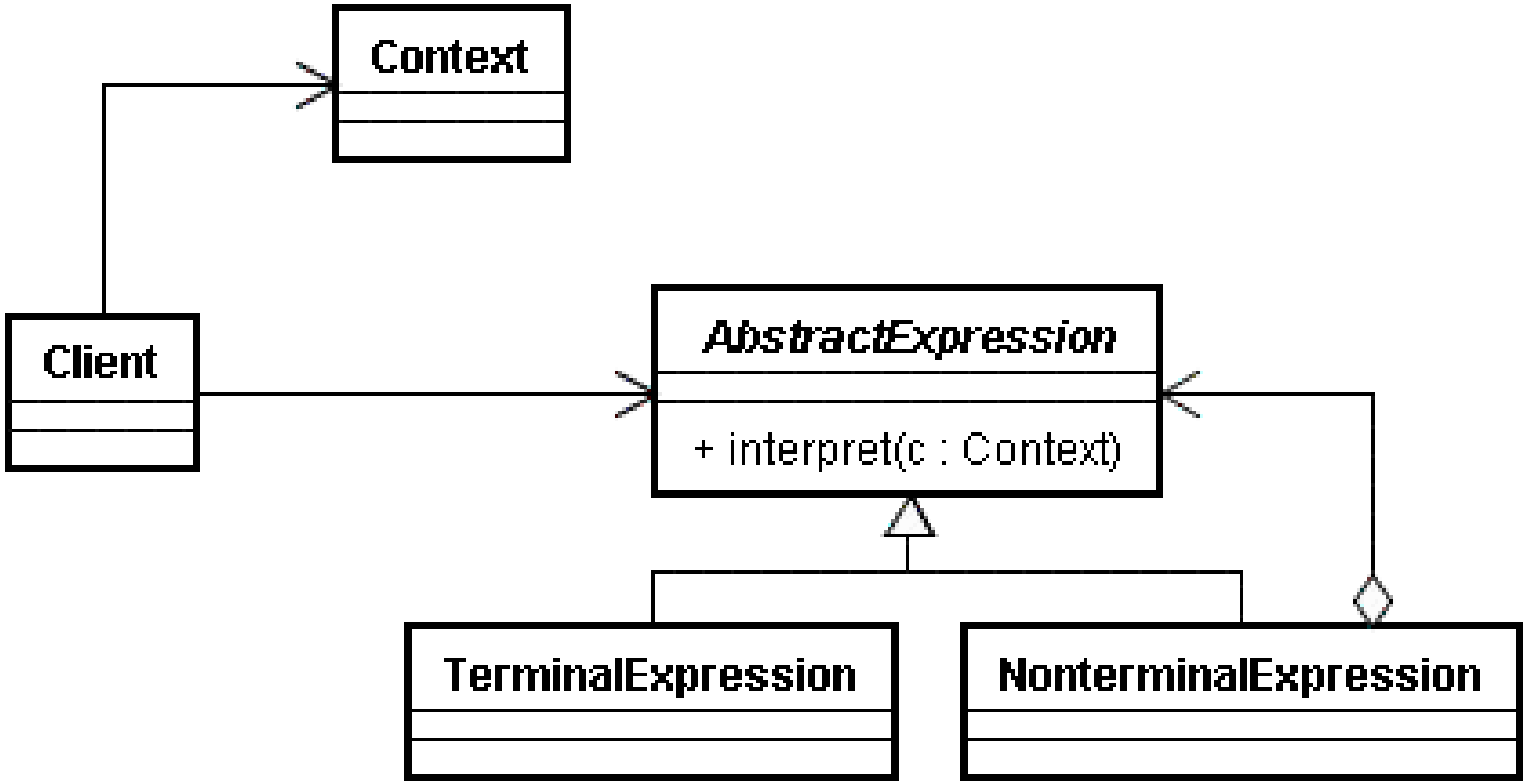
- Algumas tarefas ocorrem com tanta frequência de formas diferentes que é interessante criar uma linguagem só para definir este tipo de problema.

A solução

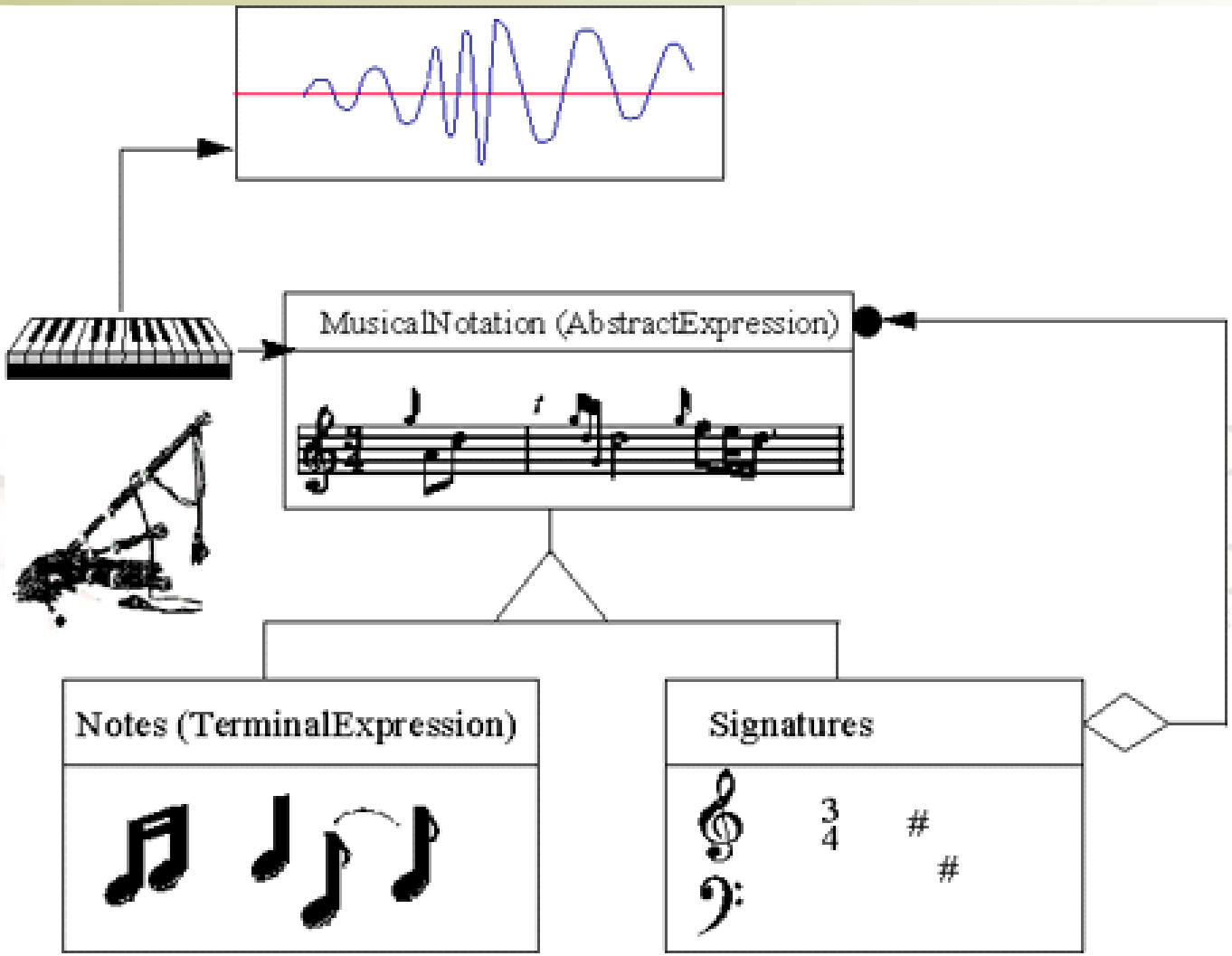
```
// Verificar se o e-mail é válido.
regexp = "[a-zA-Z0-9_.-]@[a-zA-Z0-9_.-].com.br";
if (email.matches(regexp)) {
        // E-mail válido.
}
else {
        // E-mail inválido.
}
```

- Expressões regulares são um exemplo: gramática criada somente para verificar padrões em Strings;
- É criado um interpretador para a nova linguagem.

Estrutura


Context
Client
AbstractExpression
+ interpret(c : Context)
TerminalExpression
NonterminalExpression

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- existe uma linguagem a ser interpretada que pode ser descrita como uma árvore sintática;
- Funciona melhor quando:
  - A linguagem é simples;
  - Desempenho não é uma questão crítica.

Vantagens e desvantagens

- É fácil mudar e estender a gramática:
  - Pode alterar expressões existentes, criar novas expressões, etc.;
  - Implementação é simples, pois as estruturas são parecidas.
- Gramáticas complicadas dificultam:
  - Se a gramática tiver muitas regras complica a manutenção.

Interpreter e Command

- Interpreter pode ser usado como um refinamento de Command:
  - Comandos são escritos numa gramática criada para tal;
  - Interpreter lê esta linguagem e cria objetos Command para execução.

Exemplos em Java

- Strings contém um método matches() para verificar expressões regulares;
- As classes Pattern e Matcher (java.util.regex) também efetuam o mesmo trabalho.

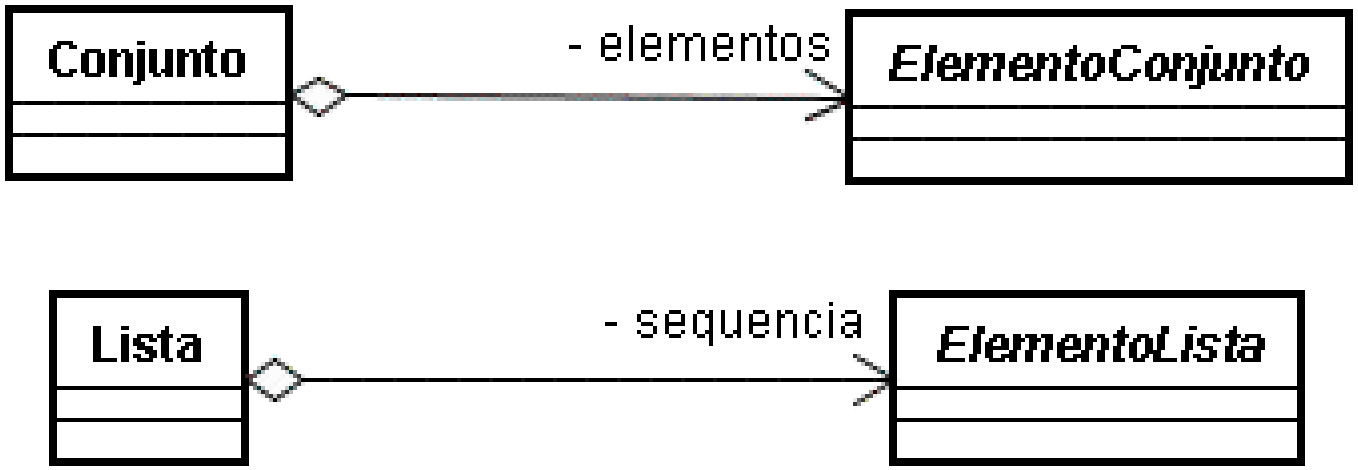
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Iterator
(Iterador)
Comportamento / Objeto

Descrição

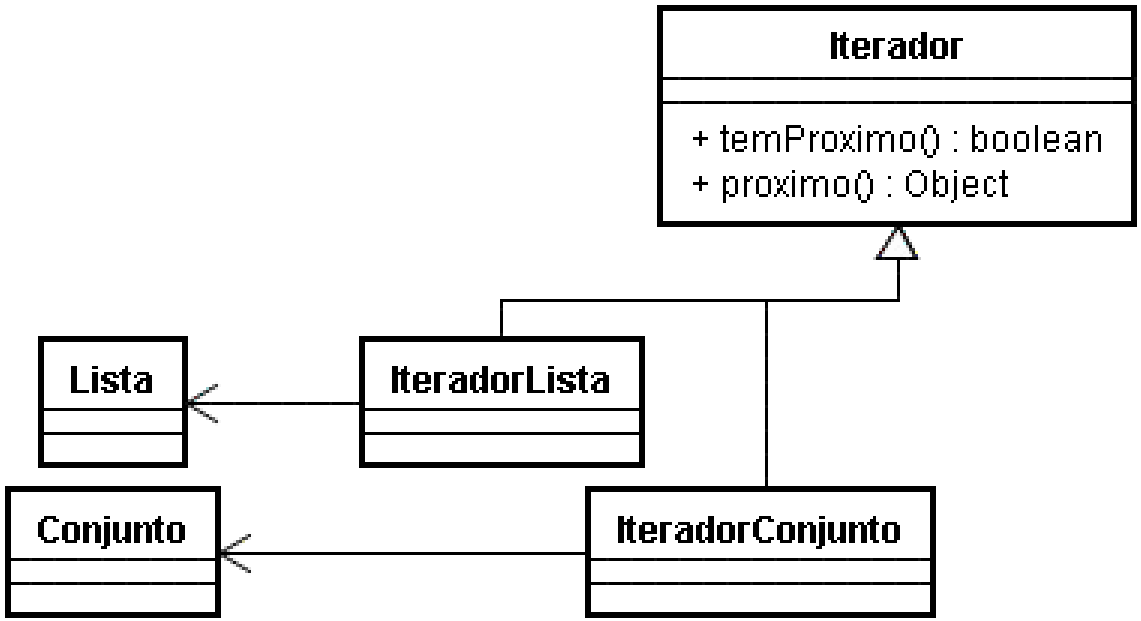
- Intenção:
  - Prover uma forma de acessar os elementos de um conjunto em sequência sem expor a representação interna deste conjunto.
- Também conhecido como:
  - Cursor.

O problema



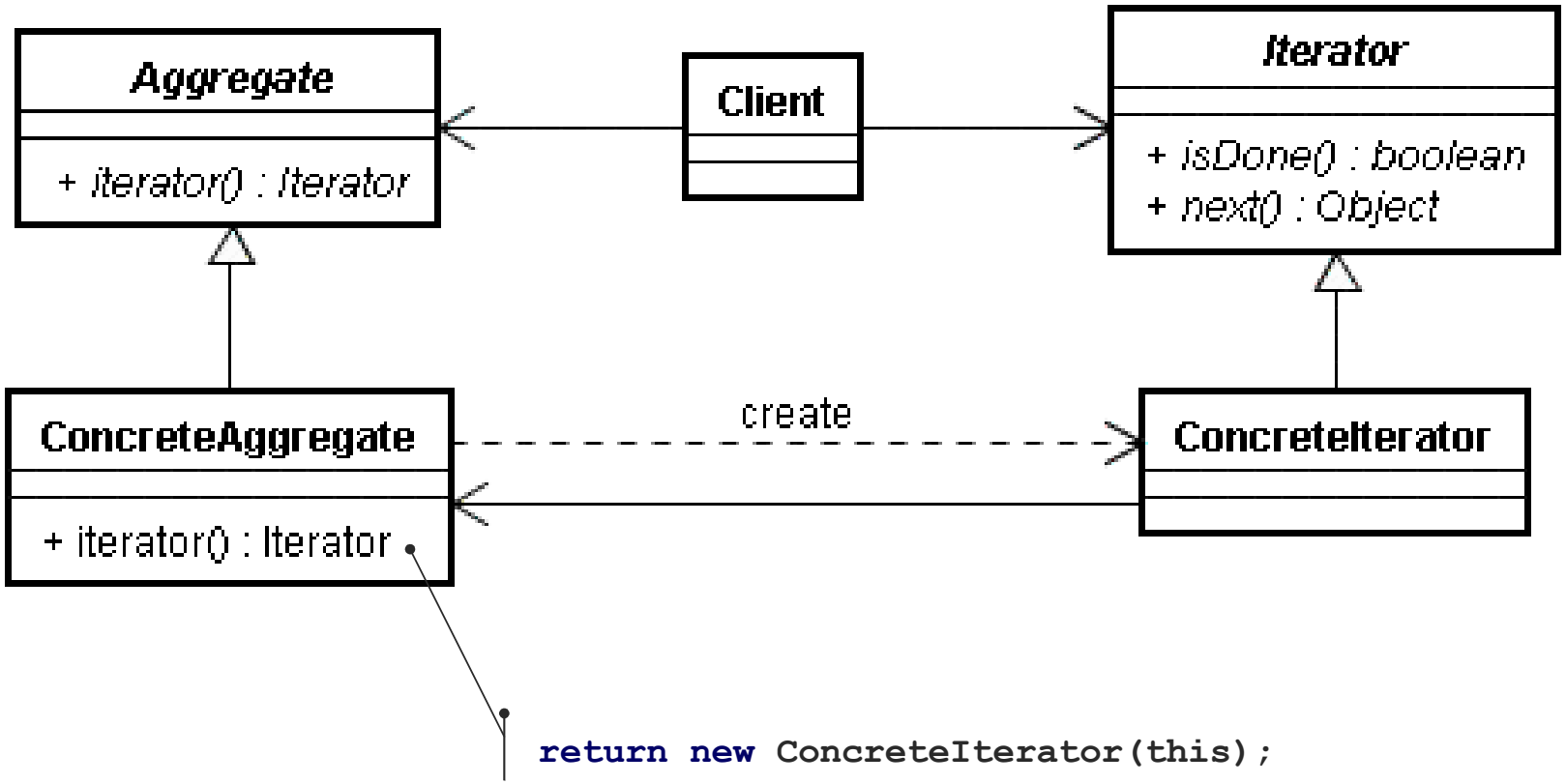
- Cliente precisa acessar os elementos;
- Cada coleção é diferente e não se quer expor a estrutura interna de cada um para o Cliente.

A solução

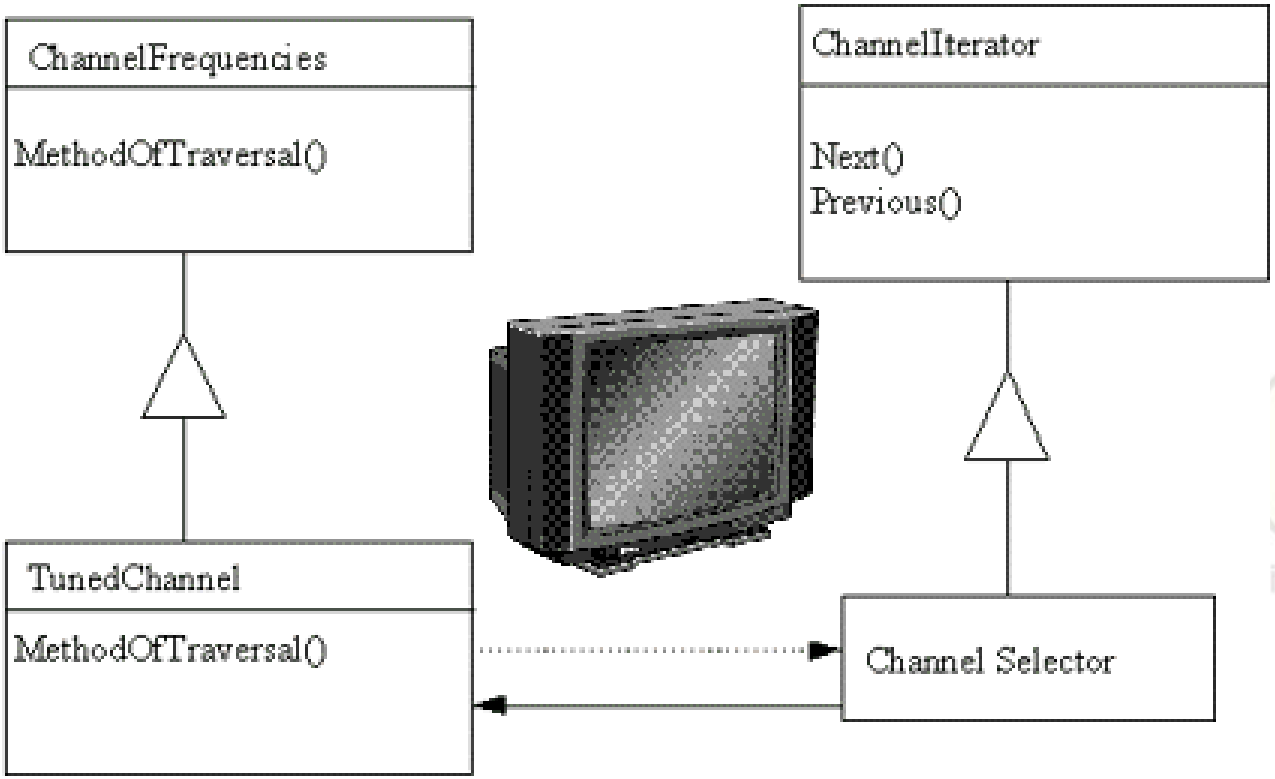


- Iterador provê acesso sequencial aos elementos, independente da coleção.

Estrutura


Aggregate
+ iterator() : Iterator
Client
Iterator
+ isDone() : boolean
+ next() : Object
ConcreteAggregate
+ iterator() : Iterator
create
ConcreteIterator
return new ConcreteIterator(this);

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- quiser acessar objetos agregados (coleções) sem expor a estrutura interna;
- quiser prover diferentes meios de acessar tais objetos;
- quiser especificar uma interface única e uniforme para este acesso.

Vantagens e desvantagens

- Múltiplas formas de acesso:
  - Basta implementar um novo iterador com uma nova lógica de acesso.
- Interface simplificada:
  - Acesso é simples e uniforme para todos os tipos de coleções.
- Mais de um iterador:
  - É possível ter mais de um acesso à coleção em pontos diferentes.

Exemplos em Java

- Iterator, da API Collections (java.util);
- Vários outros:
  - javax.swing.text.ElementIterator;
  - java.text.StringCharacterIterator;
  - Etc.

Java 1.4:

```
Iterator iter = colecao.iterator();
while (iter.hasNext()) {
  Object o = iter.next();
}
```

Java 5:

```
for (Object o : colecao) {

}
```

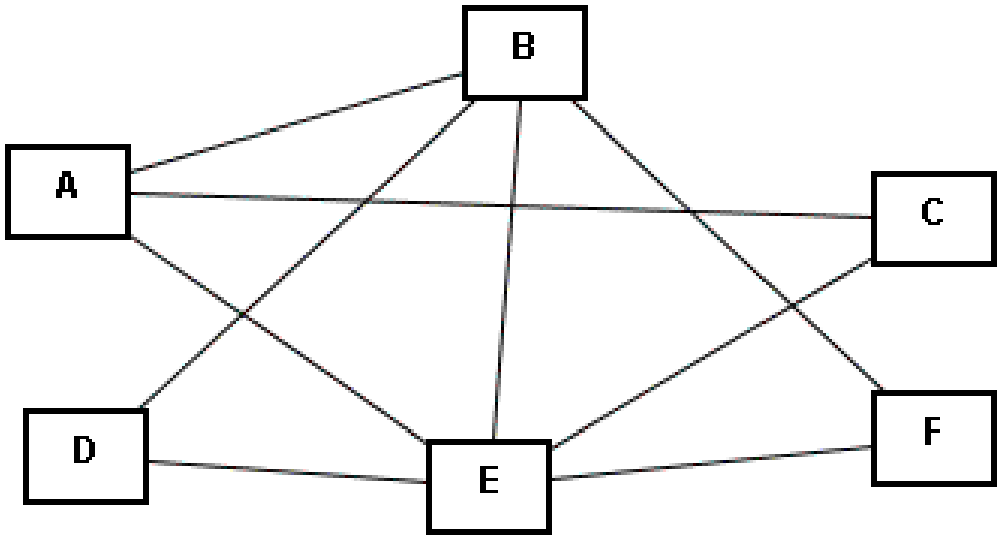
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Mediator
(Mediador)
Comportamento / Objeto

Descrição

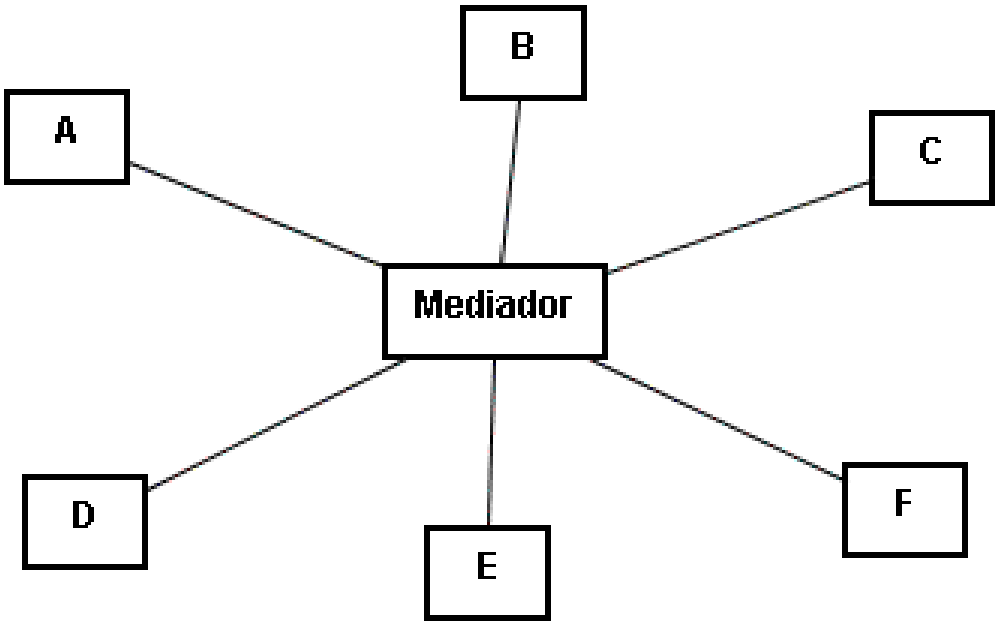
- Intenção:
  - Definir um objeto que encapsula a informação de como um conjunto de outros objetos interagem entre si. Promove o acoplamento fraco, permitindo que você altere a forma de interação sem alterar os objetos que interagem.

O problema



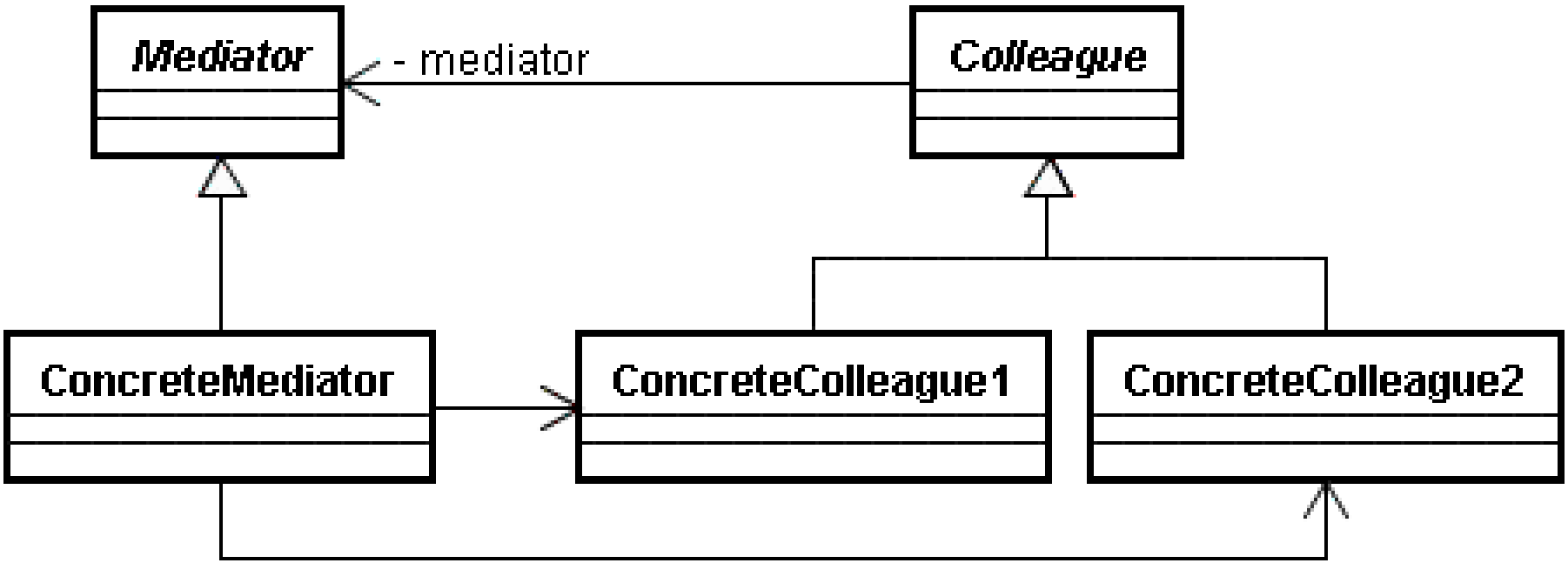
- Modelagem OO encoraja a distribuição de responsabilidades;
- Esta distribuição resulta em um emaranhado de conexões.

A solução

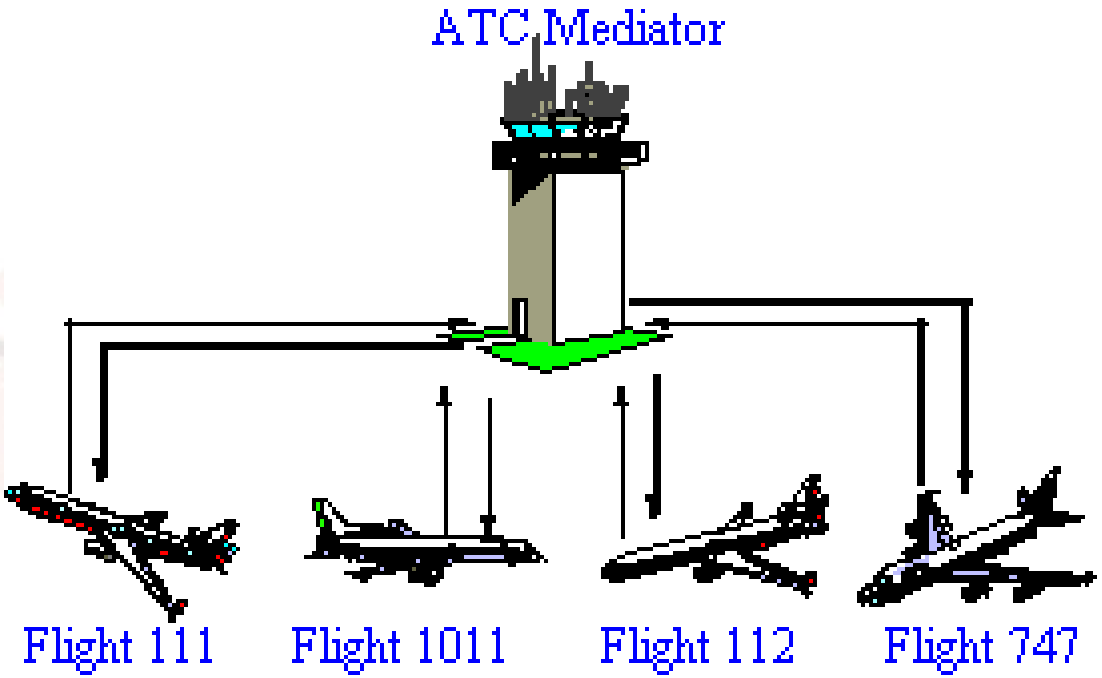


- Um mediador assume a tarefa de realizar a comunicação entre os muitos objetos.

Estrutura


Mediator
- mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- um conjunto de objetos se comunica de uma forma bem determinada, porém complexa;
- reutilizar uma classe é difícil pois ela tem associação com muitas outras;
- um comportamento que é distribuído entre várias classes deve ser extensível sem ter que criar muitas subclasses.

Vantagens e desvantagens

- Limita extensão por herança:
  - Para estender ou alterar o comportamento, basta criar uma subclasse do mediador.
- Desacopla objetos:
  - Desacoplamento promove o reuso.
- Simplifica o protocolo:
  - Relações Colleagues x Mediator são mais simples de manter do que muitas espalhadas;
  - Fica mais claro como os objetos interagem.
- Exagero pode levar a sistema monolítico.

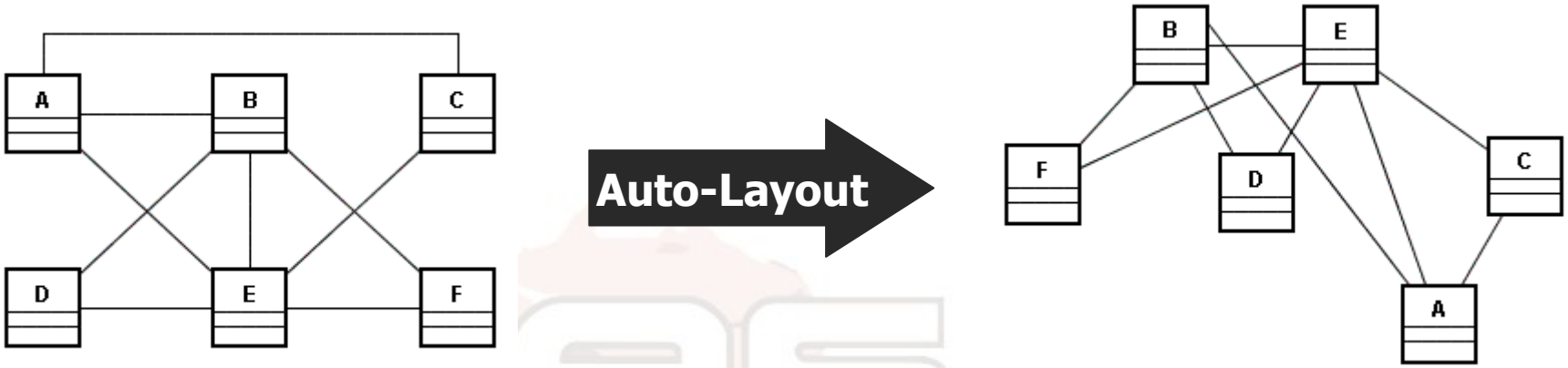
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Memento
(Recordação)
Comportamento / Objeto

Descrição

- Intenção:
  - Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto para que possa ser restaurado posteriormente.
- Também conhecido como:
  - Token.

O problema

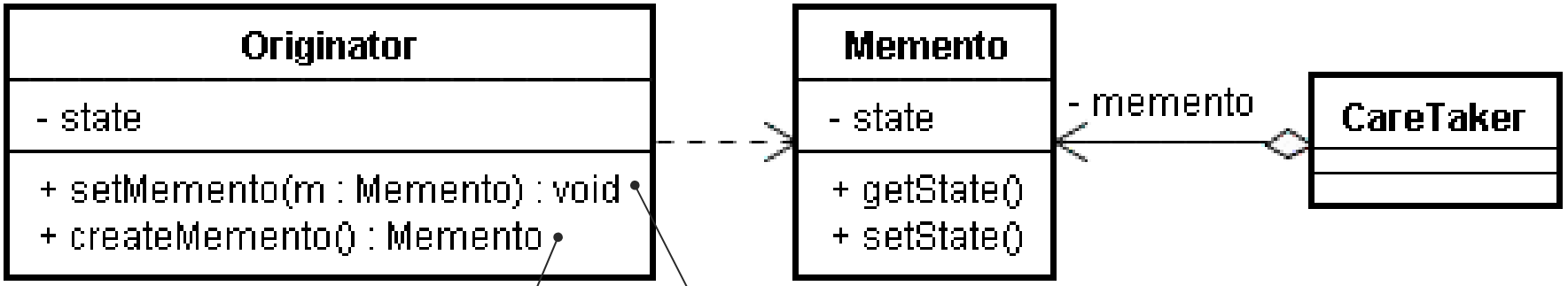


- Para dar suporte a operação de *undo*, é preciso armazenar o estado do(s) objeto(s) antes de uma determinada operação.

A solução

- Se o estado (pontos x,y; tamanho, etc.) de cada objeto (cada classe e linhas) foi armazenado, basta restaurá-los.

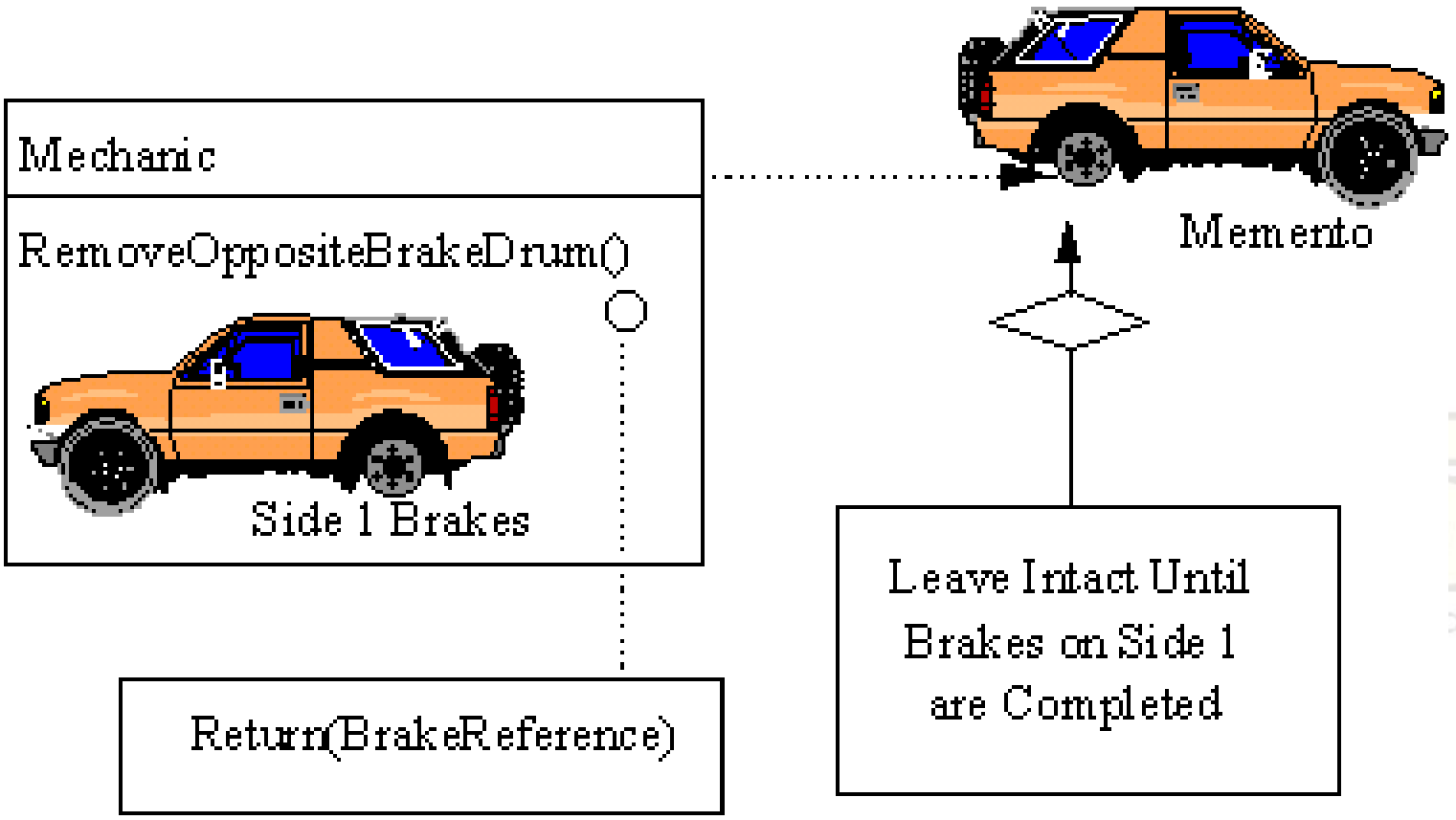
Estrutura




```
return new Memento(state);
```

```
state = m.getState();
```

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- o estado do objeto (ou de parte dele) deve ser armazenado para ser recuperado no futuro;
- uma interface direta para obtenção de tal estado iria expor a implementação e quebrar o encapsulamento.

Vantagens e desvantagens

- Preserva o encapsulamento:
  - Retira do objeto original a tarefa de armazenar estados anteriores;
  - Caretaker não pode expor a estrutura interna do objeto, a qual tem acesso;
  - No entanto pode ser difícil esconder este estado em algumas linguagens.
- Pode ser caro:
  - Dependendo da quantidade de estado a ser armazenado, pode custar caro.

Persistência

- Memento persistente?
  - Em banco de dados?
  - Serializado em disco?
  - Convertido em texto (XML)?
  - Etc.
- Memento só em memória?

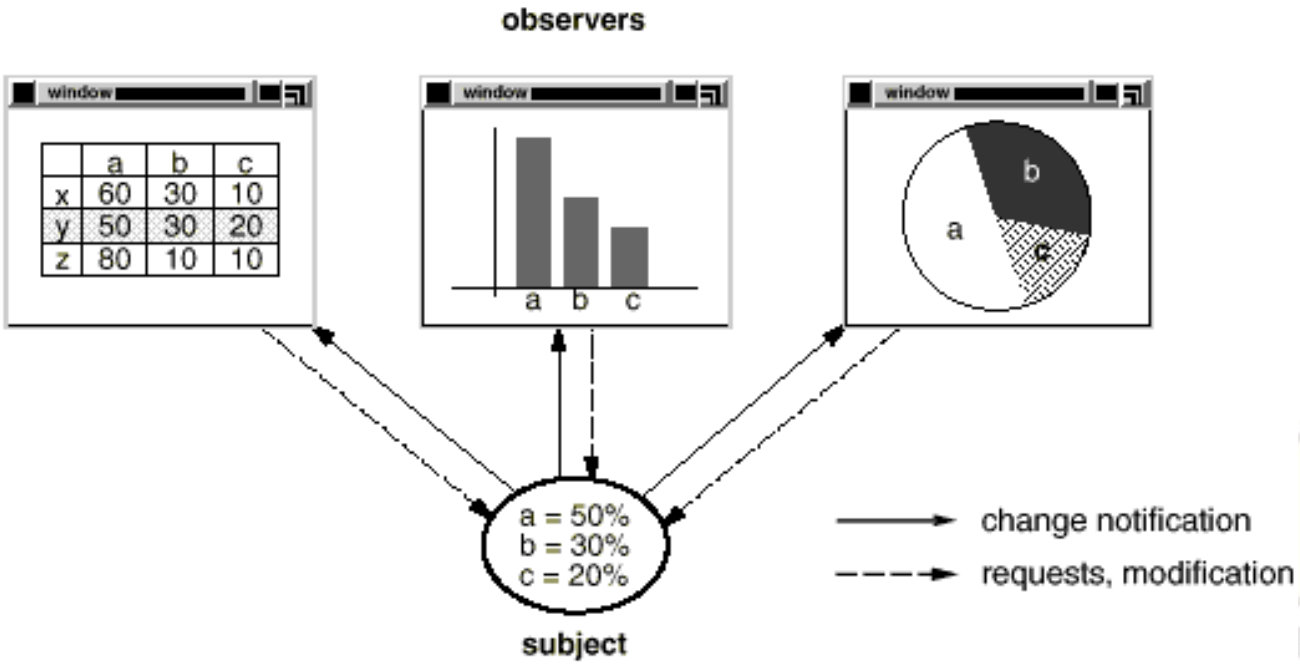
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Observer
(Observador)
Comportamento / Objeto

Descrição

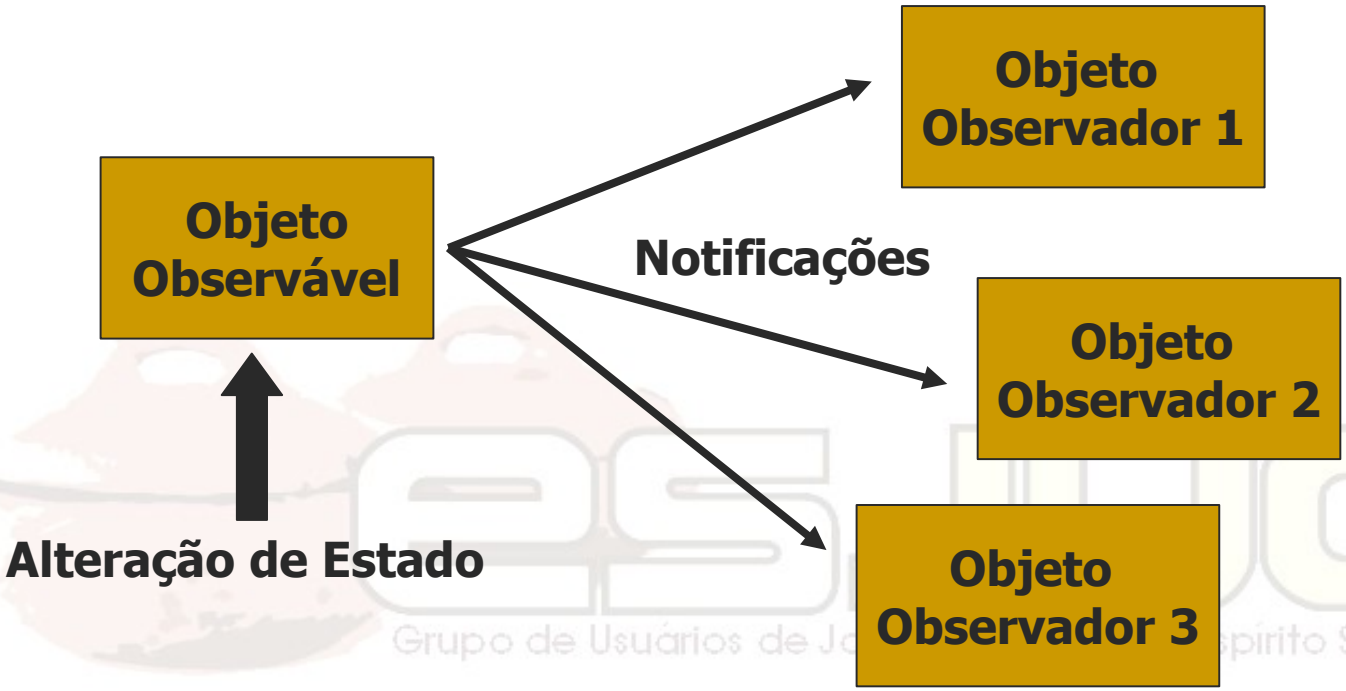
- Intenção:
  - Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, os outros são notificados e se atualizam.
- Também conhecido como:
  - Dependents, Publish-Subscribe.

O problema



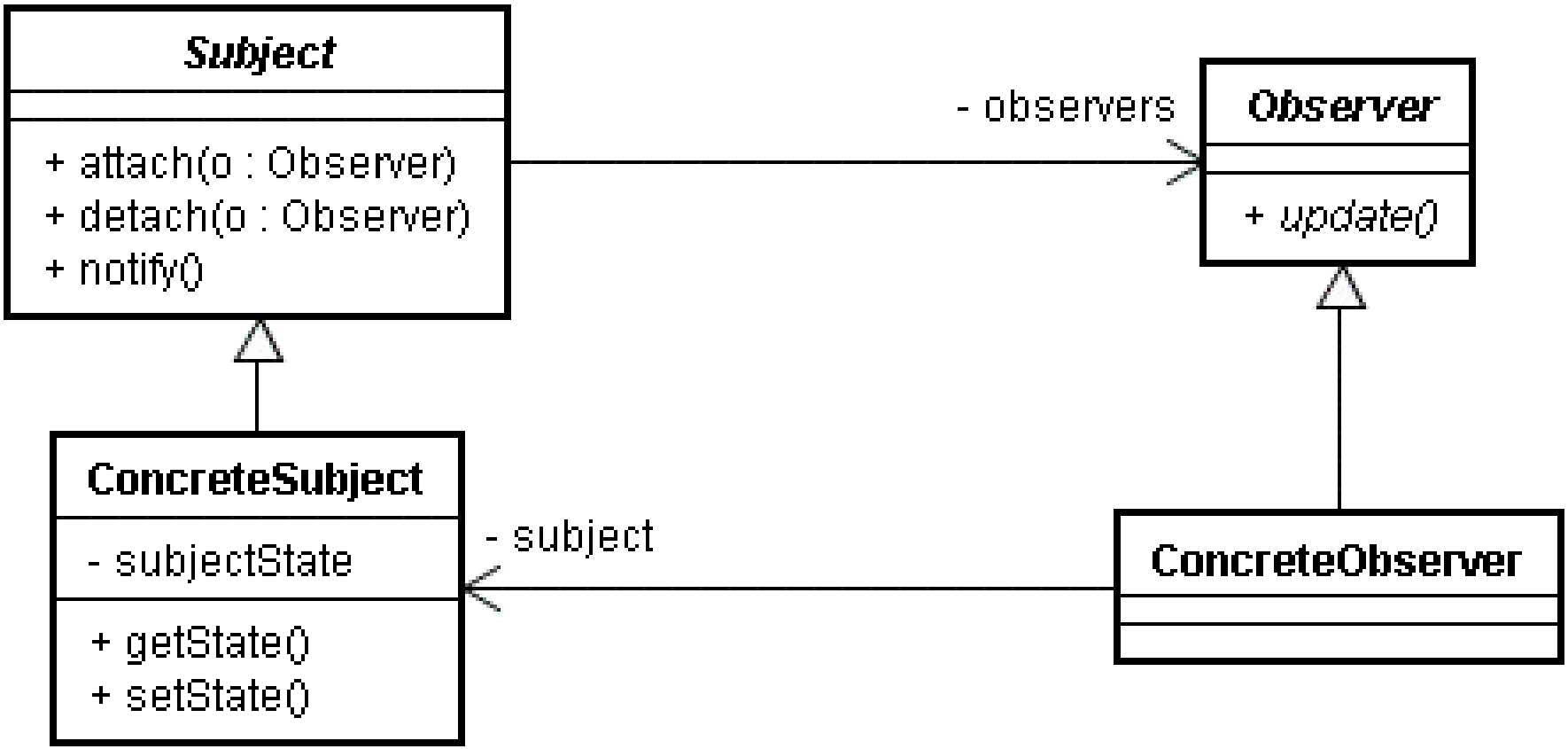
- O efeito colateral de distribuir responsabilidade entre objetos é manter a consistência entre eles.

A solução

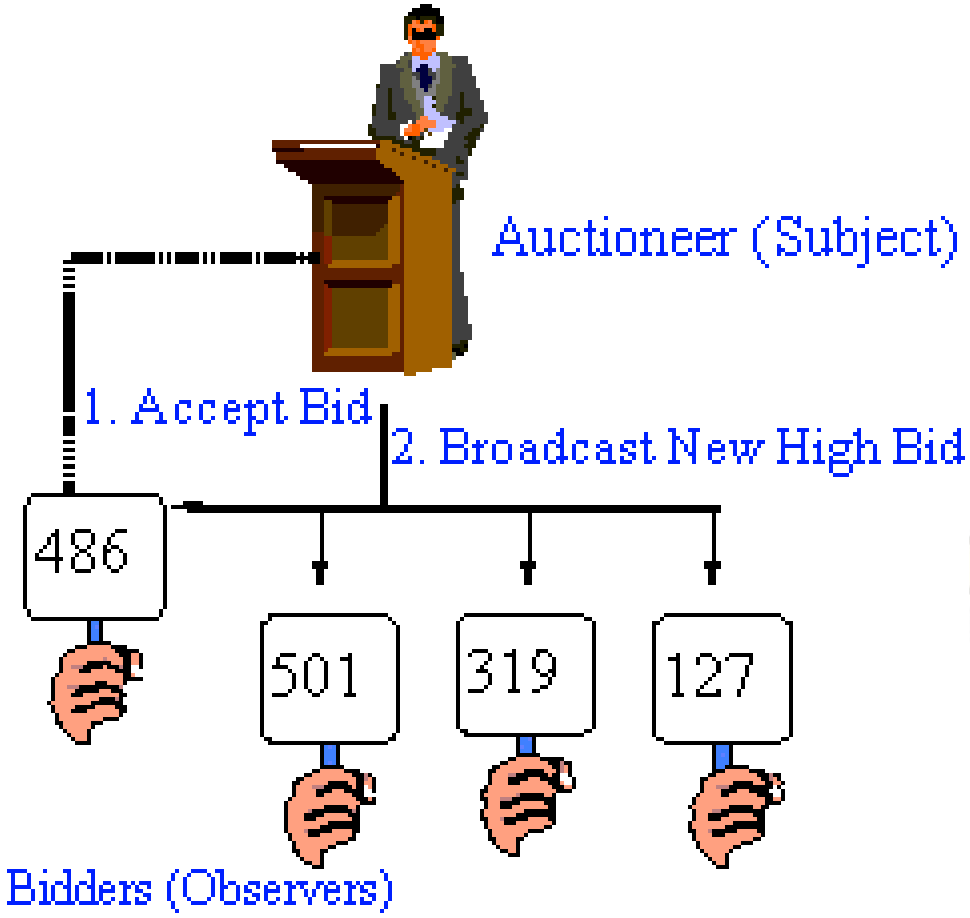


- Objeto observável registra os observadores e os notifica sobre qualquer alteração.

Estrutura


Subject
+ attach(o : Observer)
+ detach(o : Observer)
+ notify()
- observers
Observer
+ update()
ConcreteSubject
- subjectState
+ getState()
+ setState()
- subject
ConcreteObserver

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

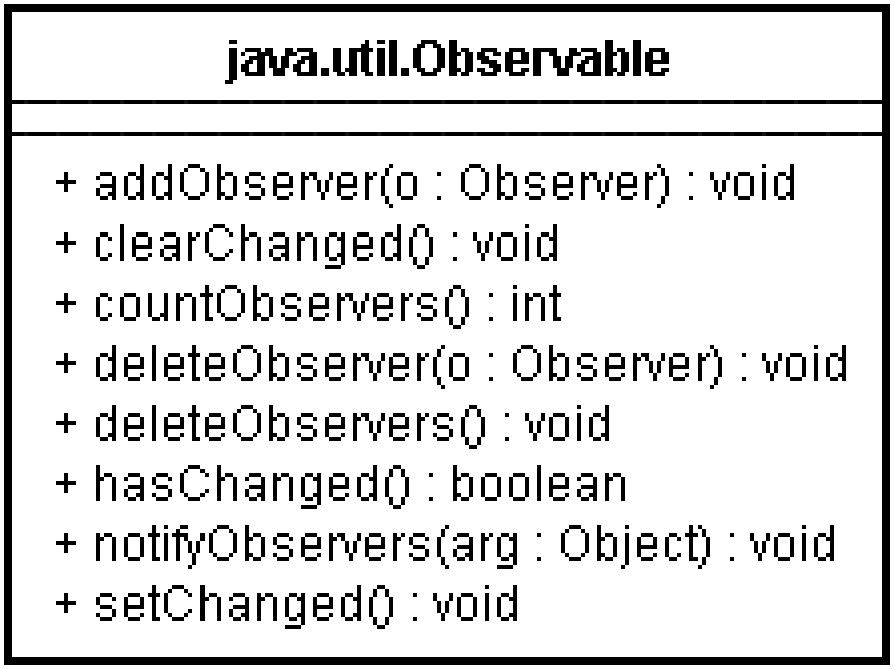
Usar este padrão quando...

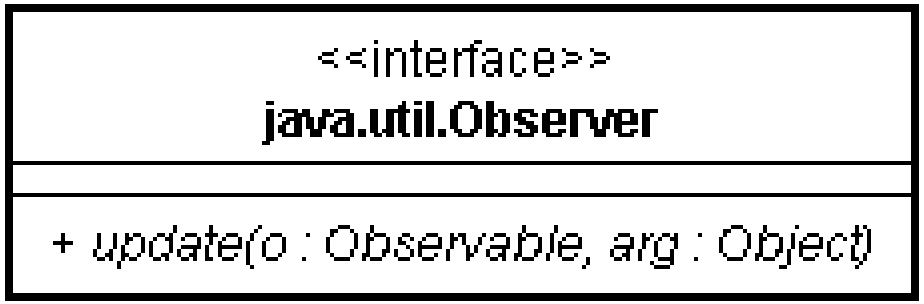
- uma abstração possui dois aspectos e é necessário separá-los em dois objetos para variá-los;
- alterações num objeto requerem atualizações em vários outros objetos não-determinados;
- um objeto precisa notificar sobre alterações em outros objetos que, a princípio, ele não conhece.

Vantagens e desvantagens

- Flexibilidade:
  - Observável e observadores podem ser quaisquer objetos;
  - Acoplamento fraco entre os objetos: não sabem a classe concreta uns dos outros;
  - É feito broadcast da notificação para todos, independente de quantos;
  - Observadores podem ser observáveis de outros, propagando em cascata.

Observer em Java






```
public class ClasseX
             extends Observable {

  alterarEstado(Object e) {
    /* Altera o estado. */

    setChanged();
    notifyObservers(e);
  }
}
```

Veja módulo 5, exemplo de MVC.

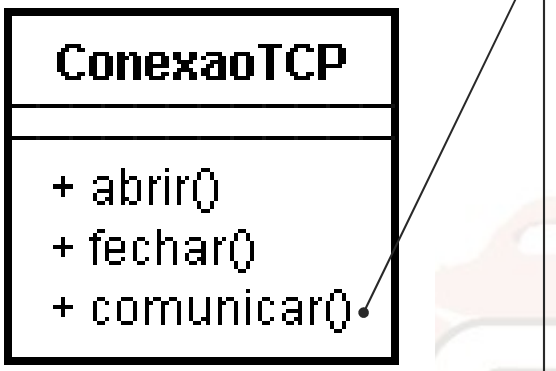
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

State
(Estado)
Comportamento / Objeto

Descrição

- Intenção:
  - Permitir que um objeto altere seu comportamento quando muda de estado interno. O objeto aparenta mudar de classe.
- Também conhecido como:
  - Objects for States

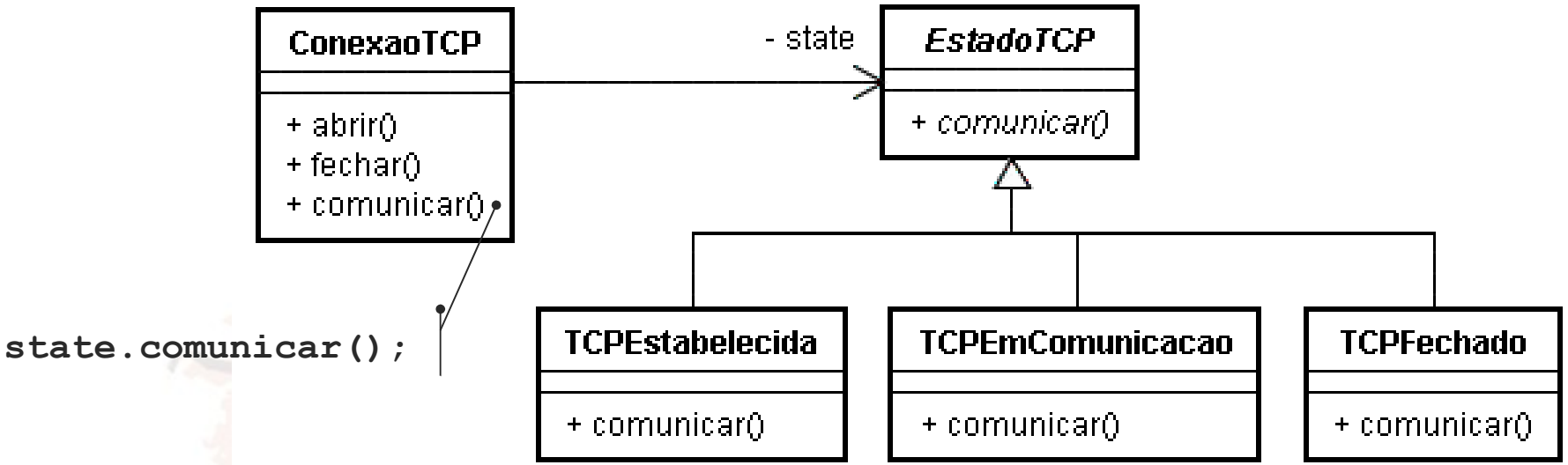
O problema




```
if (conexaoAberta()) {
  // Envia a comunicação.
}
else if (recebendoDados()) {
  // Aguarda terminar a comunicação.
}
else if (conexaoFechada()) {
  // Erro ou reconexão automática.
}
```

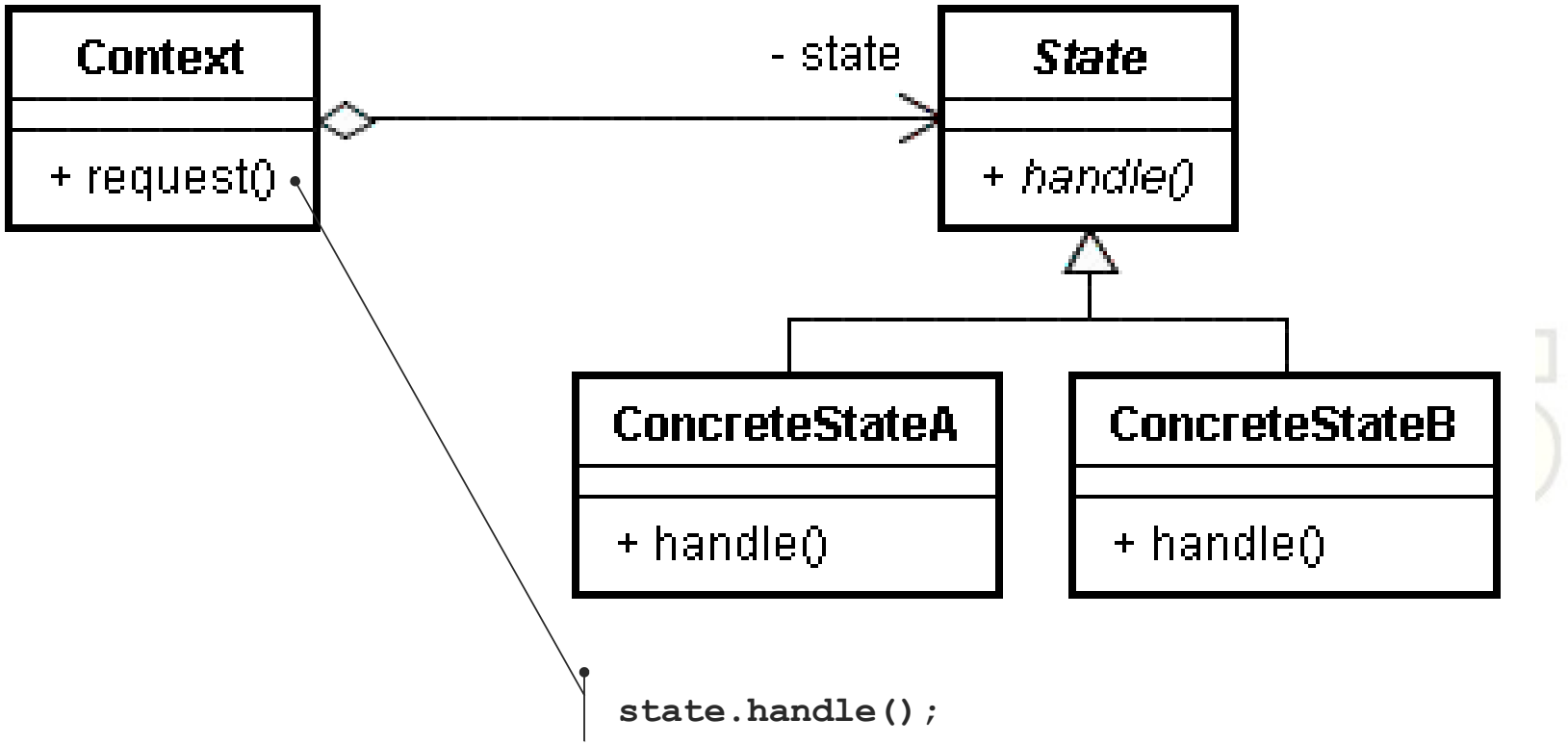
- Um objeto responde diferentemente dependendo do seu estado interno.

A solução

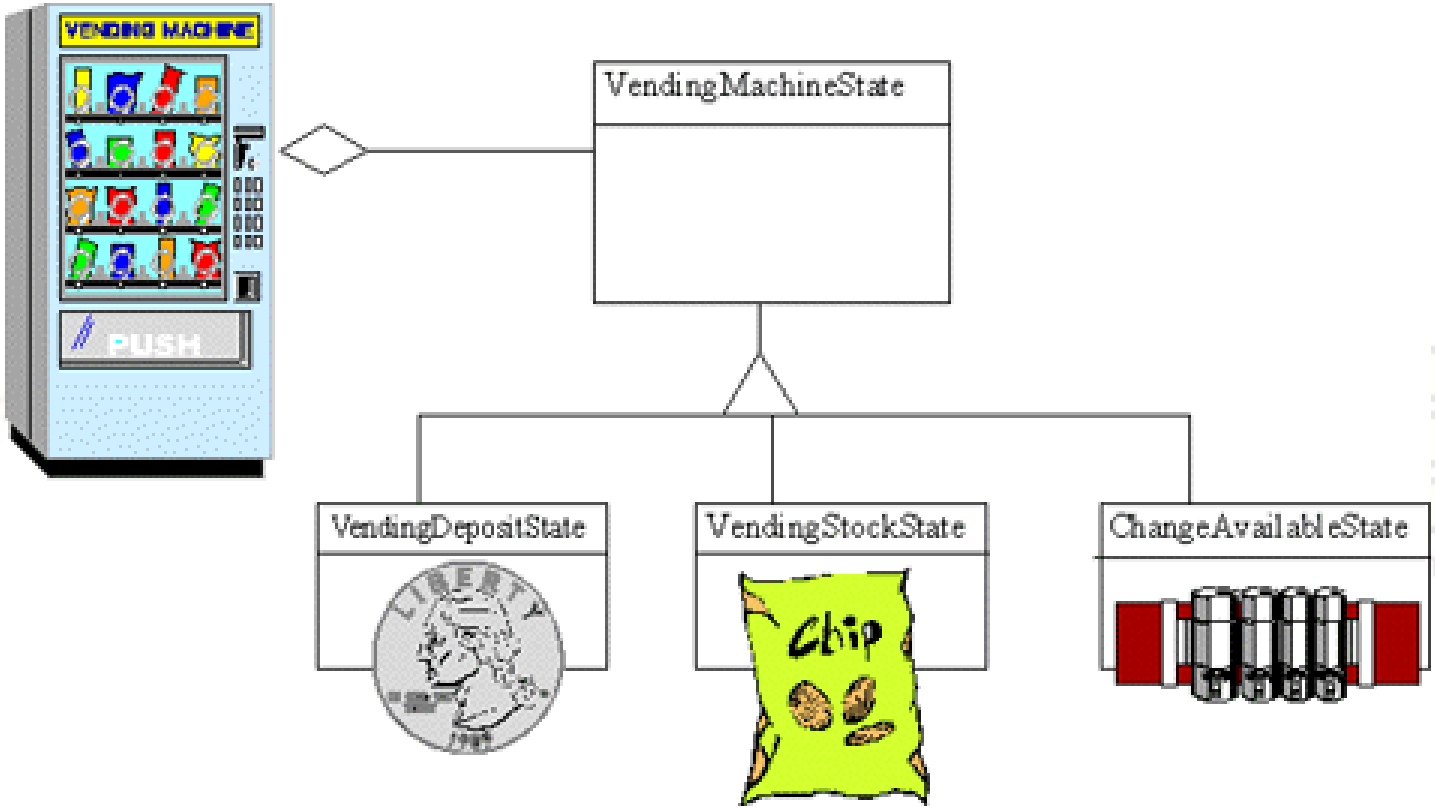


- Conexão possui um objeto que representa seu estado e implementa o método que depende deste estado.

Estrutura


Context
+ request()
- state
State
+ handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()
state.handle();

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- o comportamento de um objeto depende do seu estado, que é alterado em tempo de execução;
- operações de um objeto possuem condicionais grandes e com muitas partes (sintoma do caso anterior).

Vantagens e desvantagens

- Separa comportamento dependente de estado:
  - Novos estados/comportamentos podem ser facilmente adicionados.
- Transição de estados é explícita:
  - Fica claro no diagrama de classes os estados possíveis de um objeto.
- States podem ser compartilhados:
  - Somente se eles não armazenarem estado em atributos.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Strategy
(Estratégia)
Comportamento / Objeto

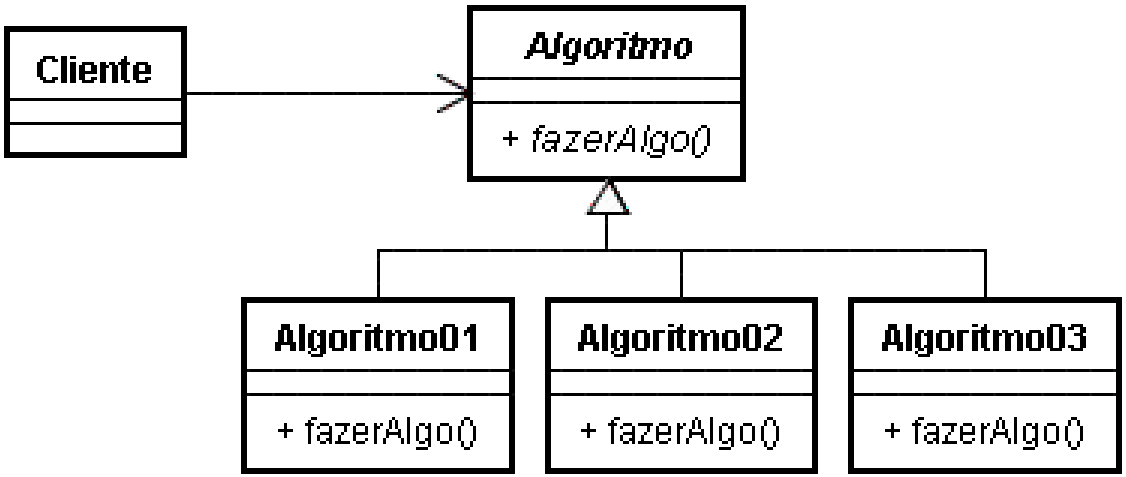
Descrição

- Intenção:
  - Definir uma família de algorítmos e permitir que um objeto possa escolher qual algorítmo da família utilizar em cada situação.
- Também conhecido como:
  - Policy.

O problema

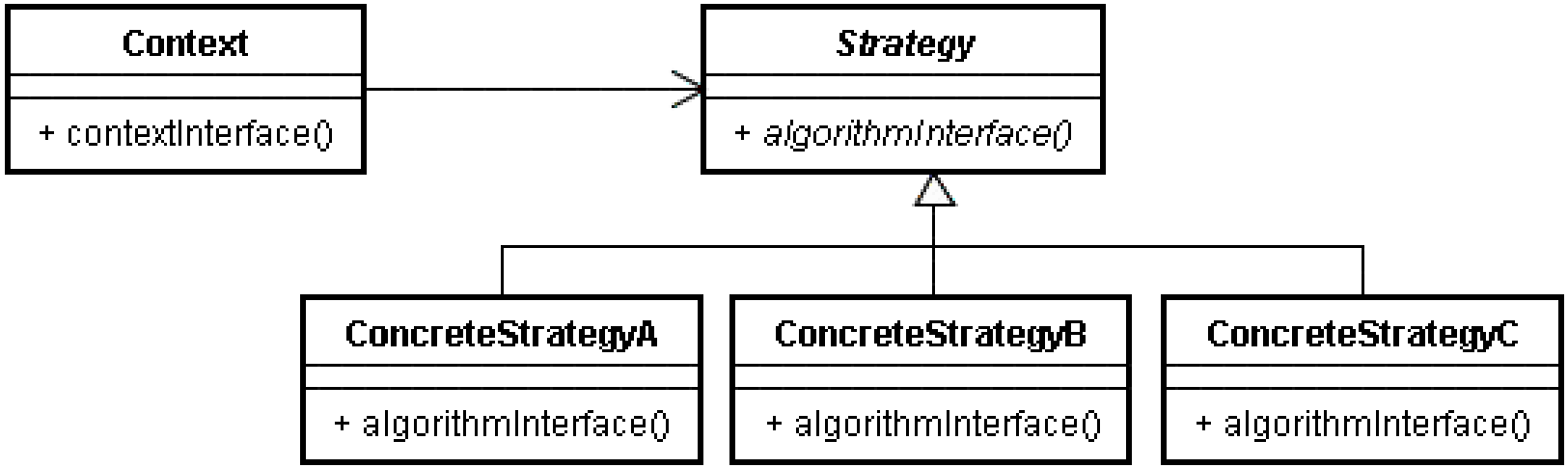
- Existem problemas que possuem vários algorítmos que os solucionam;
  - Ex.: quebrar um texto em linhas.
- Seria interessante:
  - Separar estes algorítmos em classes específicas para serem reutilizados;
  - Permitir que sejam intercambiados e que novos algorítmos sejam adicionados com facilidade.

A solução

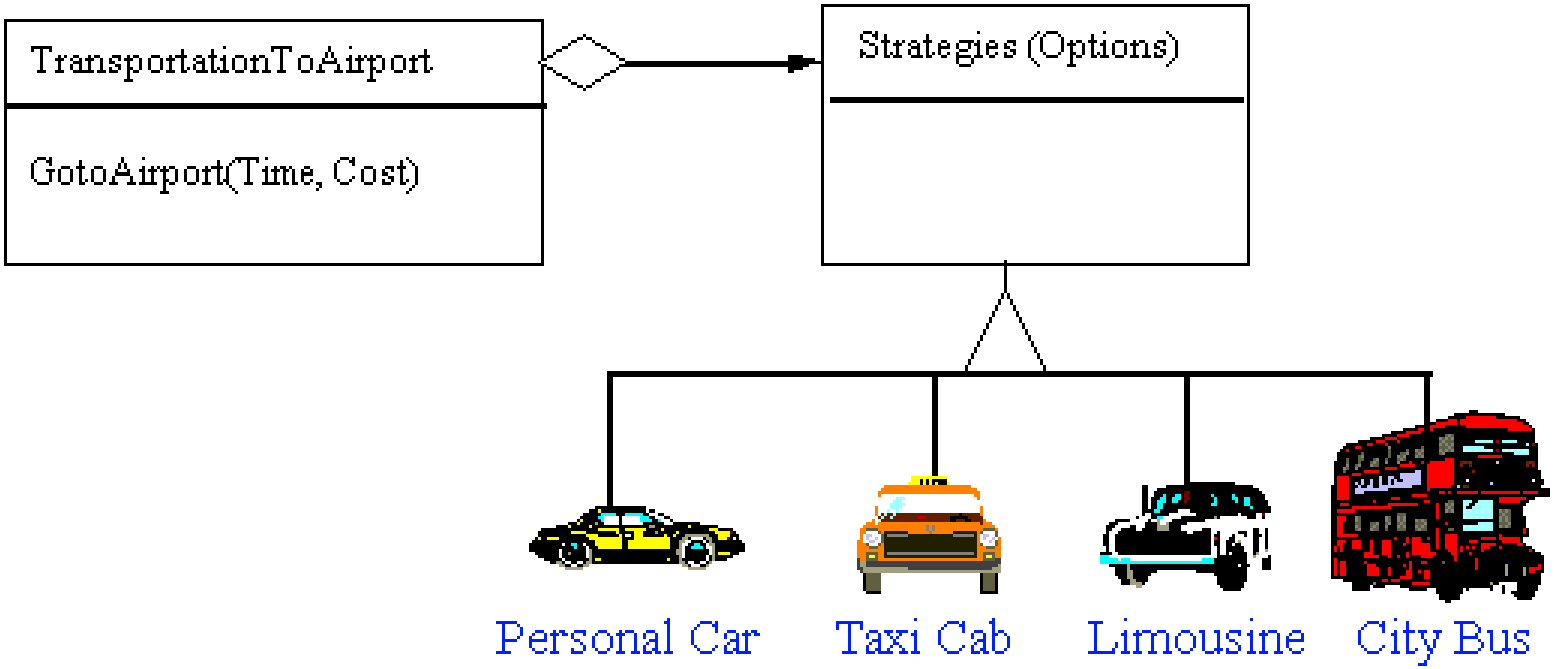


- Comportamento é encapsulado em objetos de uma mesma família;
- Similar ao padrão State, no entanto não representa o estado do objeto.

Estrutura


Context
+ contextInterface()
Strategy
+ algorithmInterface()
ConcreteStrategyA
+ algorithmInterface()
ConcreteStrategyB
+ algorithmInterface()
ConcreteStrategyC
+ algorithmInterface()

Analogia



Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- várias classes diferentes diferem-se somente no comportamento;
- você precisa de variantes de um mesmo algorítmo;
- um algoritmo utiliza dados que o cliente não deve conhecer;
- uma classe define múltiplos comportamentos, escolhidos num grande condicional.

Vantagens e desvantagens

- Famílias de algorítmos:
  - Beneficiam-se de herança e polimorfismo.
- Alternativa para herança do cliente:
  - Comportamento é a única coisa que varia.
- Eliminam os grandes condicionais:
  - Evita código monolítico.
- Escolha de implementações:
  - Pode alterar a estratégia em runtime.

Vantagens e desvantagens

- Clientes devem conhecer as estratégias:
  - Eles que escolhem qual usar a cada momento.
- Parâmetros diferentes para algoritmos diferentes:
  - Há possibilidade de duas estratégias diferentes terem interfaces distintas.
- Aumenta o número de objetos:
  - Este padrão aumenta a quantidade de objetos pequenos presentes na aplicação.

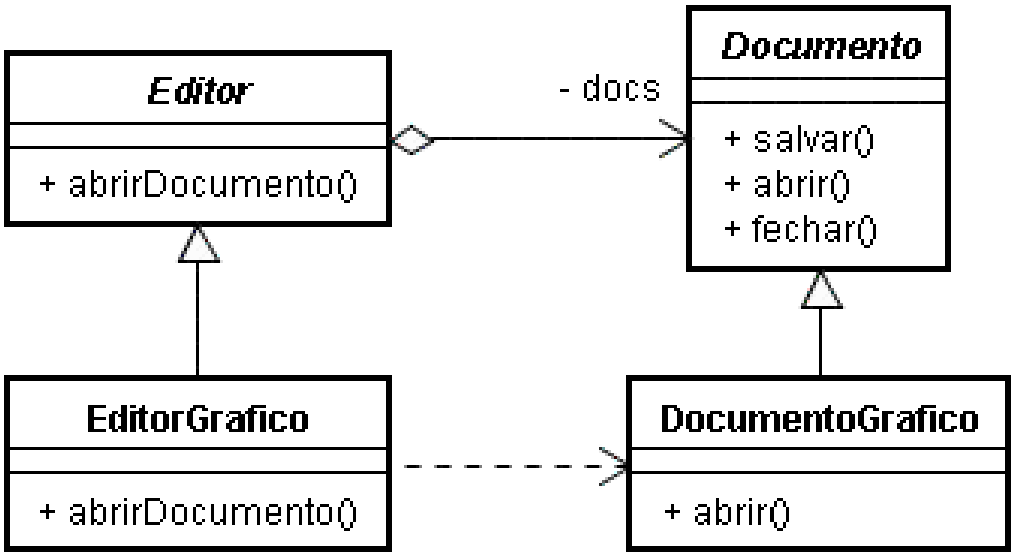
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Template Method
(Método Modelo)
Comportamento / Classe

Descrição

- Intenção:
  - Definir o esqueleto de um algorítmo numa classe, delegando alguns passos às subclasses. Permite que as subclasses alterem partes do algorítmo, sem mudar sua estrutura geral.

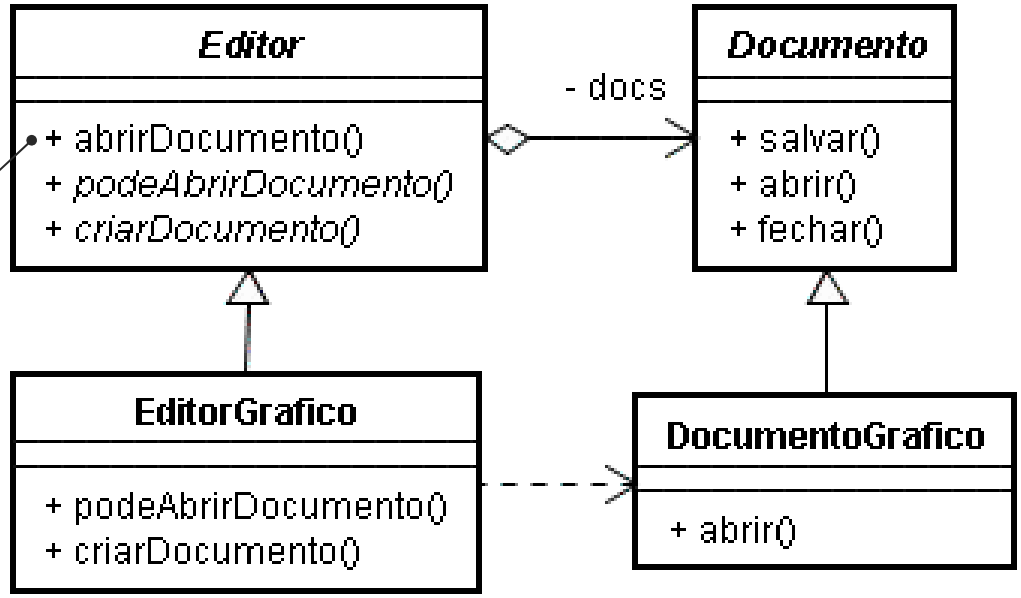
O problema



- Alguns passos de abrirDocumento() e abrir() são iguais para todo Editor e Documento;
- EditorGrafico e DocumentoGrafico têm que sobrescrever todo o método.

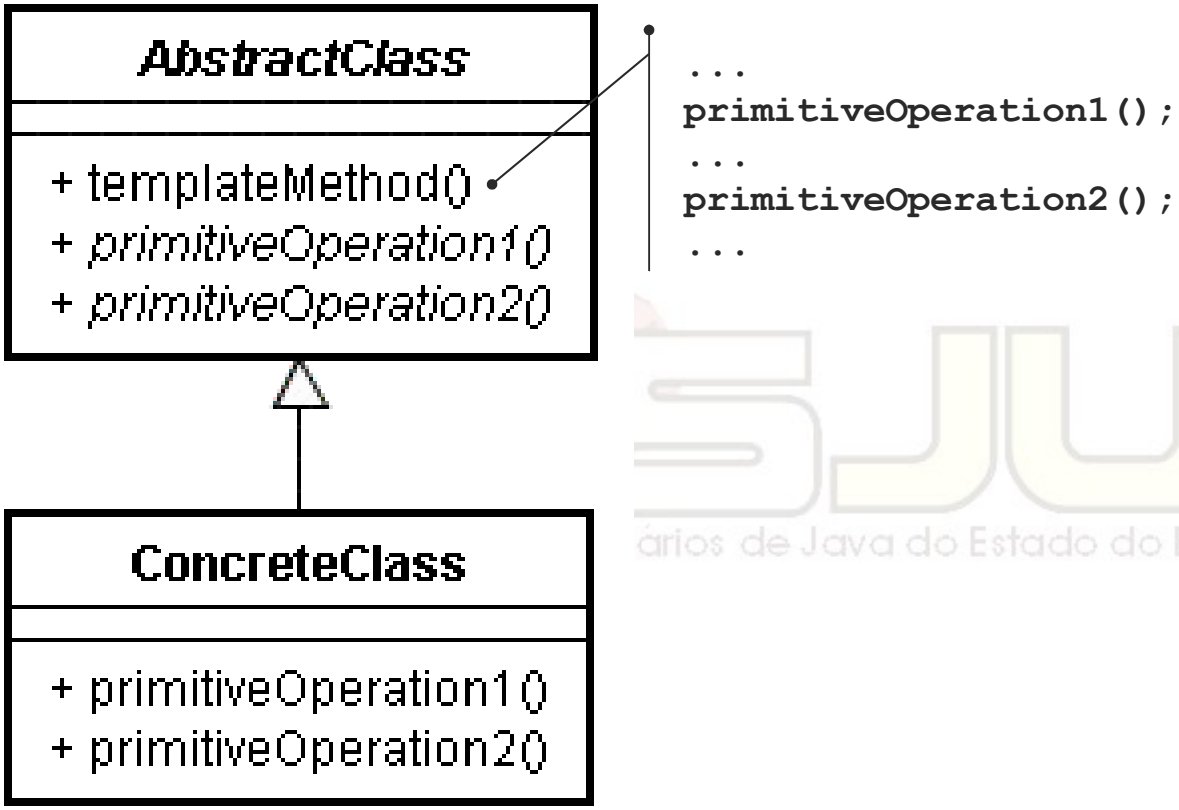
A solução

```
if (podeAbrirDocumento()) {
  Documento d =
          criarDocumento();
  docs.add(d);
}
```

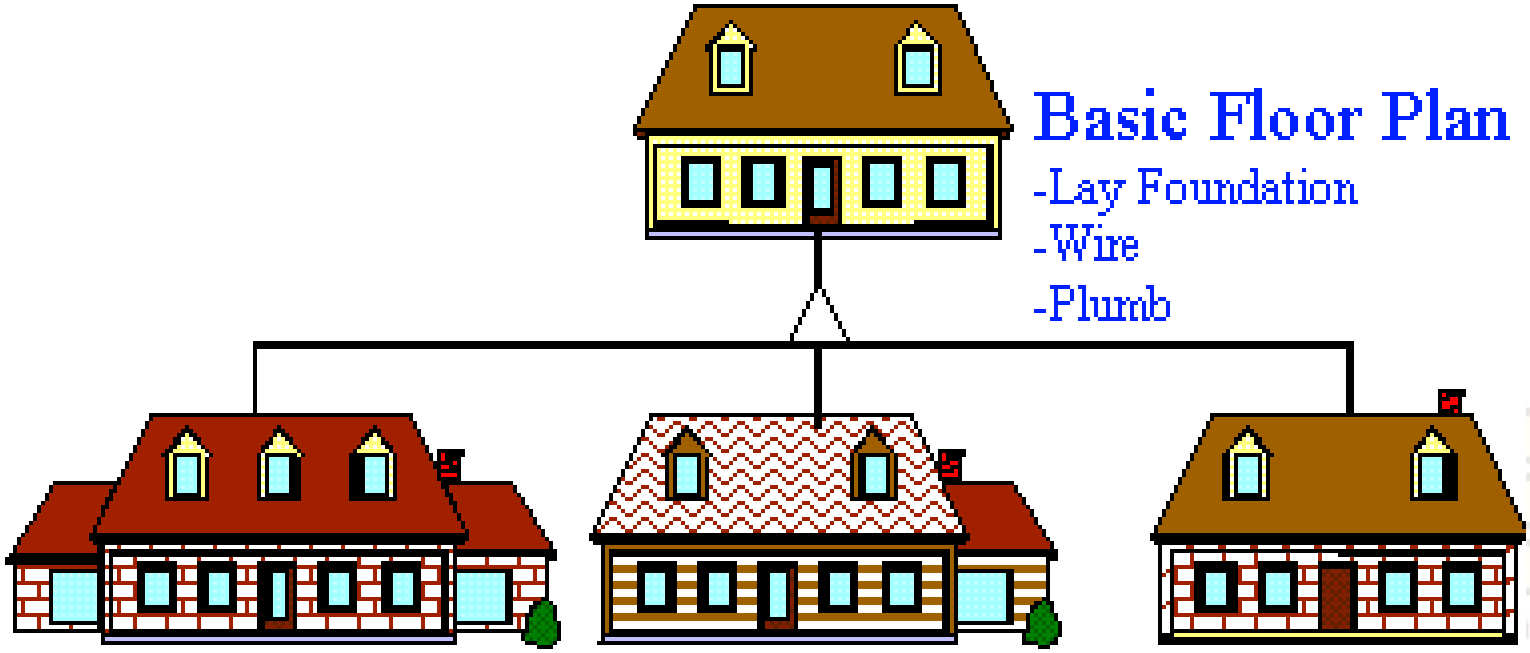


- Método é implementado em Editor, chamando métodos abstratos que são implementados em EditorGrafico.

Estrutura



Analogia



Variations added to Template Floor Plan

Non-Software Examples of Software Design Patterns by Michael Duell
http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

Usar este padrão quando...

- quiser implementar partes invariantes de um algorítmo na superclasse e deixar o restante para as subclasses;
- comportamento comum de subclasses deve ser generalizado para evitar duplicidade de código;
- quiser controlar o que as subclasses podem estender (métodos finais).

Vantagens e desvantagens

- Reuso de código:
  - Partes de um algorítmo são reutilizadas por todas as subclasses.
- Controle:
  - É possível permitir o que as subclasses podem estender (métodos finais).
- Comportamento padrão extensível:
  - Superclasse pode definir o comportamento padrão e permitir sobrescrita.

Exemplo em Java

- java.util.Comparator e métodos que o utilizam (ex.: Arrays.sort()):
  - Lógica de ordenação implementada em Arrays.sort();
  - Comparação entre objetos (x1 > x2?) implementada no Comparator;
  - Pode ser considerado um "Template Method" com escopo de objeto, visto que não usa herança, mas sim delegação.

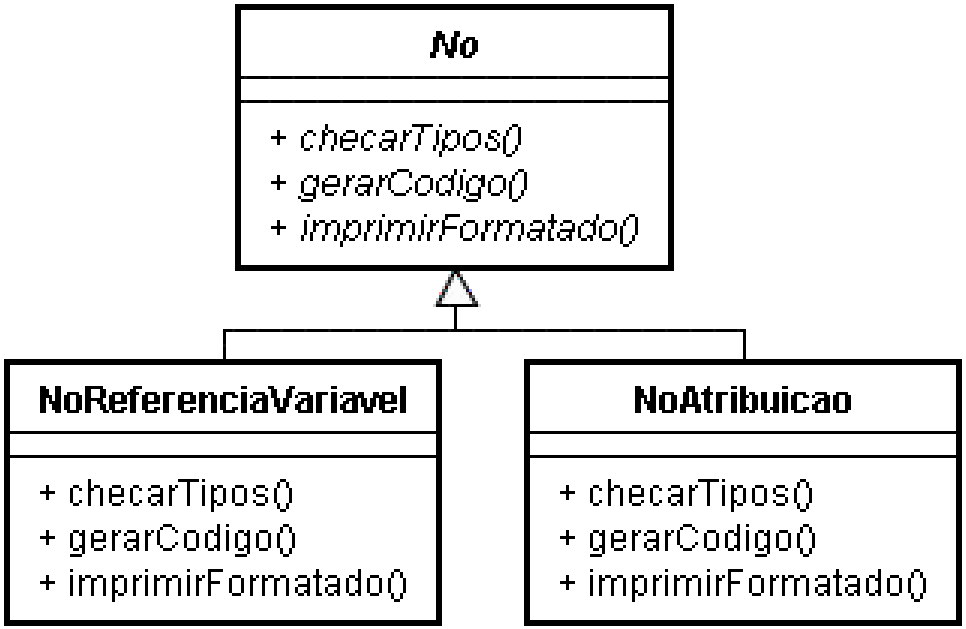
Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Visitor
(Visitante)
Comportamento / Objeto

Descrição

- Intenção:
  - Representar uma operação a ser efetuada em objetos de uma certa classe como outra classe. Permite que você defina uma nova operação sem alterar a classe na qual a operação é efetuada.

O problema

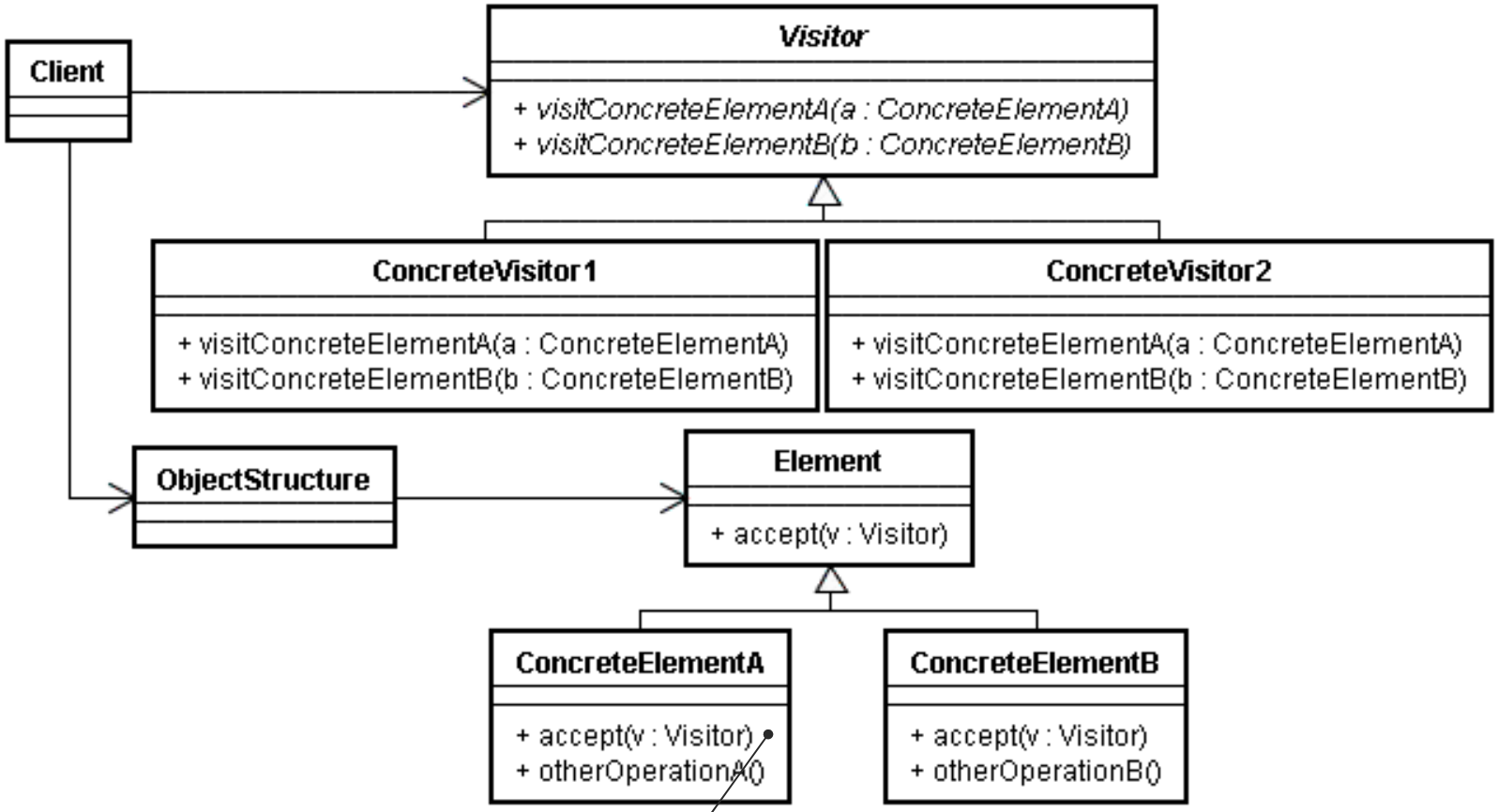


- Um compilador representa o código como uma árvore sintática. Para cada nó precisa realizar algumas operações;
- Misturar estas operações pode ser confuso.

A solução

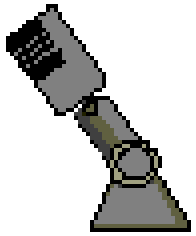
- As operações possíveis viram classes;
- Cada uma deve tratar todos os parâmetros (nós) possíveis para aquela operação;
- Nós agora possuem somente uma operação: aceitar(v: VisitanteNo).

Estrutura



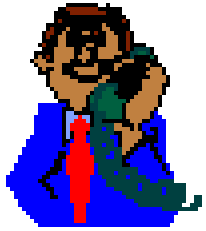

```
v.visitConcreteElementA(this);
```

Analogia



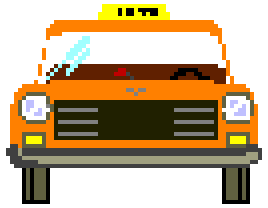
Cab Company Dispatcher

(Object Structure is List of Customers)



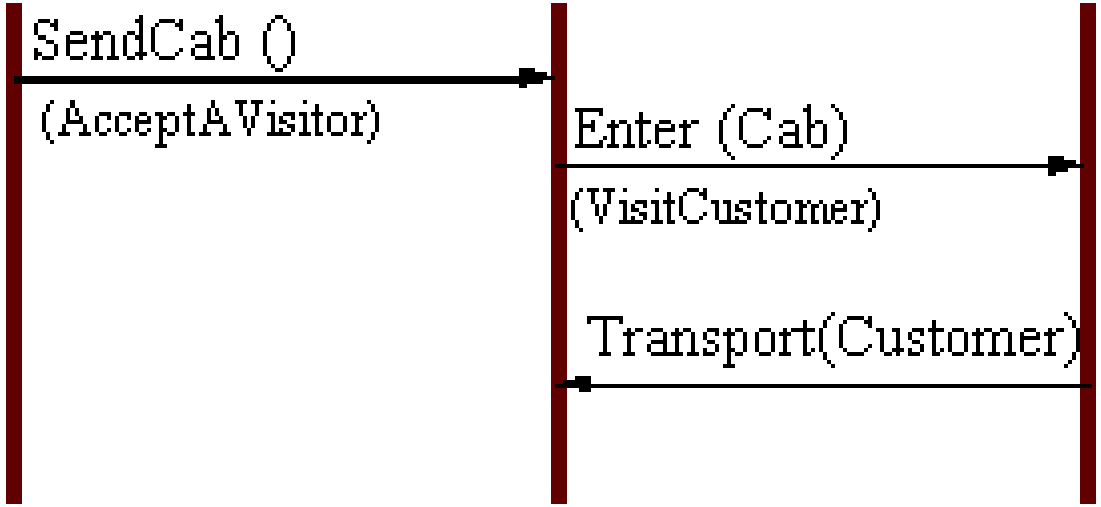
Customer

(Concrete Element of Customer List)



Taxi

(Visitor)



Usar este padrão quando...

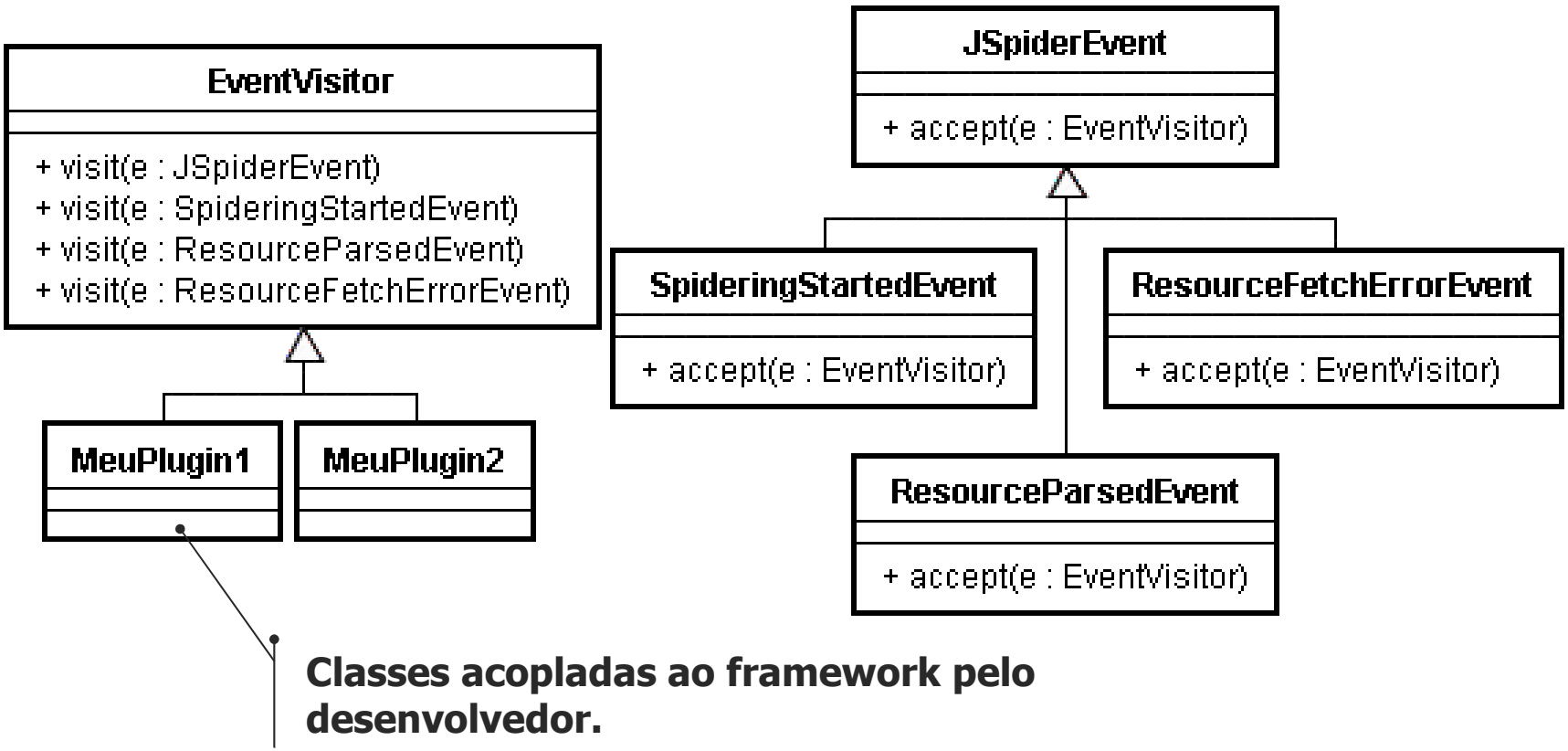
- uma estrutura de objetos contém muitas classes com muitas operações diferentes;
- quiser separar as operações dos objetos-alvo, para não "poluir" seu código;
- o conjunto de objetos-alvo raramente muda, pois cada novo objeto requer novos métodos em todos os visitors.

Vantagens e desvantagens

- Organização:
  - Visitor reúne operações relacionadas.
- Fácil adicionar novas operações:
  - Basta adicionar um novo Visitor.
- Difícil adicionar novos objetos:
  - Todos os Visitors devem ser mudados.
- Transparência:
  - Visite toda a hierarquia transparentemente.
- Quebra de encapsulamento:
  - Pode forçar a exposição de estrutura interna para que o Visitor possa manipular.

Exemplos em Java

JSpider Crawler Event Model



Classes acopladas ao framework pelo desenvolvedor.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Conclusões

Flexibilidade e extensão

- Padrões permitem variar um comportamento ou estendê-lo:
  - Strategy: implementações de um algorítmo;
  - State: comportamento diferente para estados diferentes;
  - Mediator: como um conjunto de objetos se comunica;
  - Iterator: como apresentar um conjunto de objetos agregados em sequência;

Flexibilidade e extensão

- Chain of Responsibility: quantidade de objetos que colabora para uma requisição;
- Template Method: partes de um algorítmo;
- Command: parâmetros e comportamento das ações;
- Etc.

Outras opções

- Encapsular a comunicação (Mediator) ou distribuí-la livremente (Observer)?
- É necessário desacoplar um objeto cliente do objeto que responde à sua requisição? Como fazê-lo?
  - Command, Observer, Mediator, Chain;
- Combinação de padrões.

Curso - Padrões de Projeto
Módulo 4: Padrões de Comportamento

Vítor E. Silva Souza
vitorsouza@gmail.com

http://www.javablogs.com.br/page/engenho
http://esjug.dev.java.net