Programação Orientada a Objetos

Padrões Comportamentais

Cristiano Lehrer, M.Sc.

ybadoo

Classificação dos Padrões de Projeto

- Propósito – o que o padrão faz:
  - Padrões de criação: abstraem o processo de criação de objetos a partir da instanciação de classes.
  - Padrões estruturais: tratam da forma como classes e objetos estão organizados para a formação de estruturas maiores.
  - Padrões comportamentais: preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos.
- Escopo – em que o padrão de projeto é aplicado:
  - Padrões de classes: em geral estáticos, definidos em tempo de compilação.
  - Padrões de objetos: em geral dinâmicos, definidos em tempo de execução.

Classificação dos padrões do GoF

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

ybadoo

Interpreter

"Dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa representação para interpretar sentenças da linguagem."

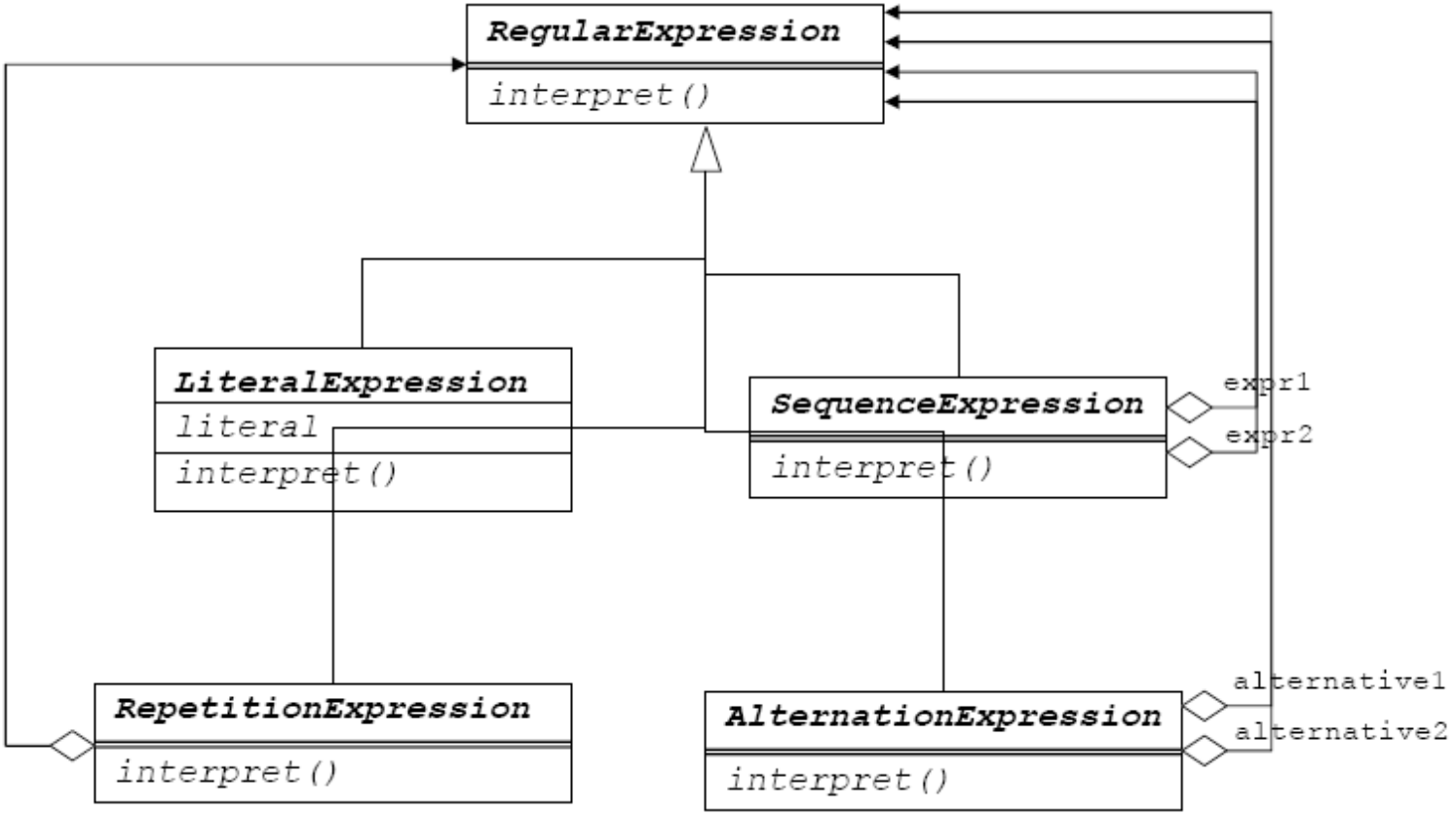
ybadoo

Problema

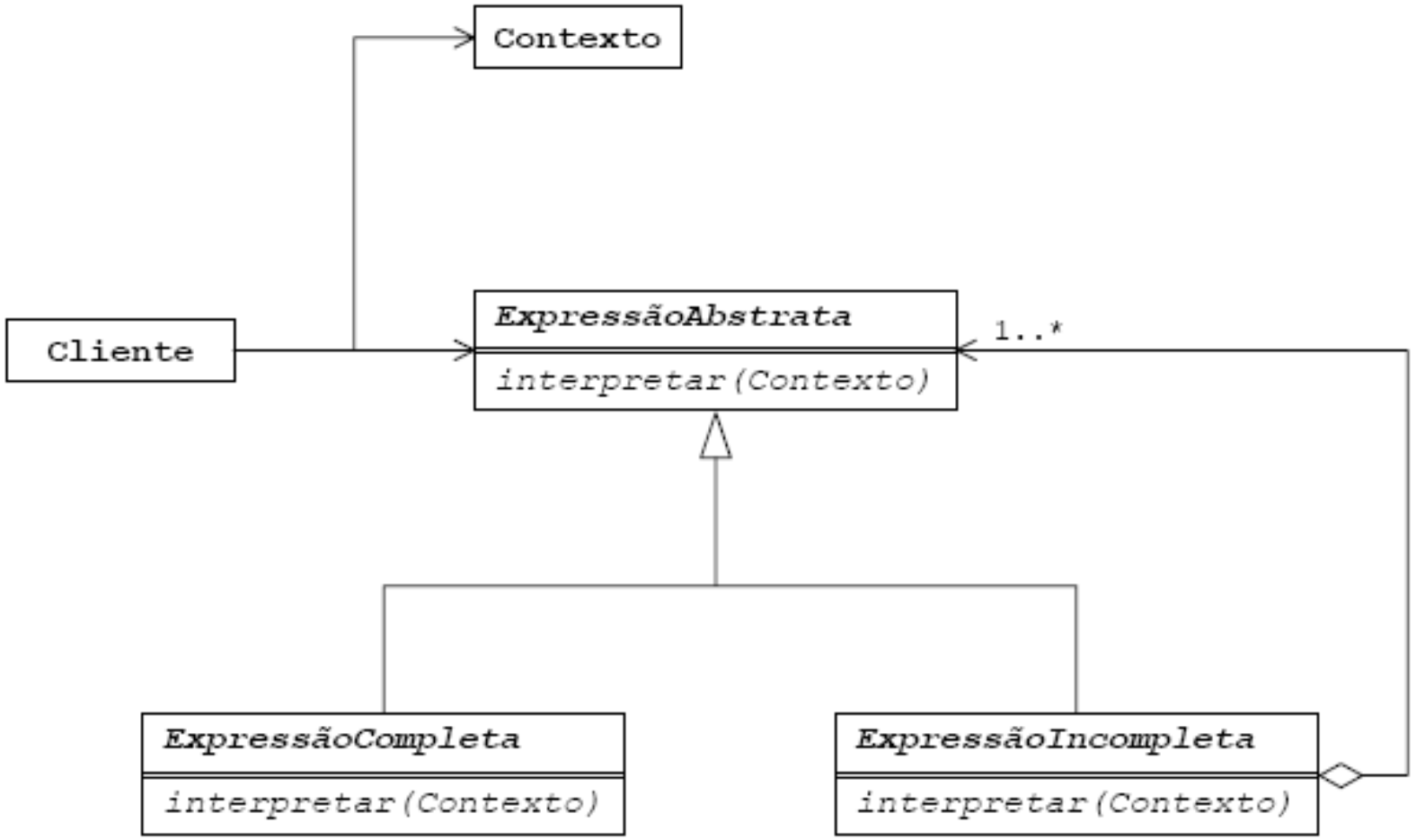
- Se comandos estão representados como objetos, eles poderão fazer parte de algoritmos maiores:
  - Vários padrões repetitivos podem surgir nesses algoritmos.
  - Operações como iteração ou condicionais podem ser frequentes:
    - Representá-las como objetos *Command*.
- Solução em OO:
  - Elaborar uma gramática para calcular expressões compostas por objetos:
    - *Interpreter* é uma extensão do padrão *Command* (ou um tipo de *Command*; ou uma micro-arquitetura construída com base em *Commands*) em que toda uma lógica de código pode ser implementada com objetos.

ybadoo

Exemplo [GoF]



Estrutura UML



Participantes (1/3)

- *AbstractExpression*:
  - Declara uma operação abstrata Interpretar comum a todos os nós na árvore sintática abstrata.
- *TerminalExpression*:
  - Implementa uma operação Interpretar associada aos símbolos terminais da gramática.
  - É necessária uma instância para cada símbolo terminal em uma sentença.
- *Context*:
  - Contém informação que é global para o interpretador.

Participantes (2/3)

- *NonTerminalExpression*:
  - É necessária uma classe desse tipo para cada regra $R::R_1R_2...R_n$ da gramática.
  - Mantém variáveis de instância do tipo *AbstractExpression* para cada um dos símbolos R_1 a R_n.
  - Implementa uma operação Interpretar para símbolos não-terminais da gramática. *Interpret* chamará a si próprio recursivamente nas variáveis que representam R_1 a R_n.

Participantes (3/3)

- *Client*:
  - Constrói (ou recebe) uma árvore sintática abstrata que representa uma determinada sentença na linguagem definida pela gramática.
  - A árvore sintática abstrata é montada a partir de instâncias das classes *NonTerminalExpression* e *TerminalExpression*.
  - Invoca a operação Interpretar.

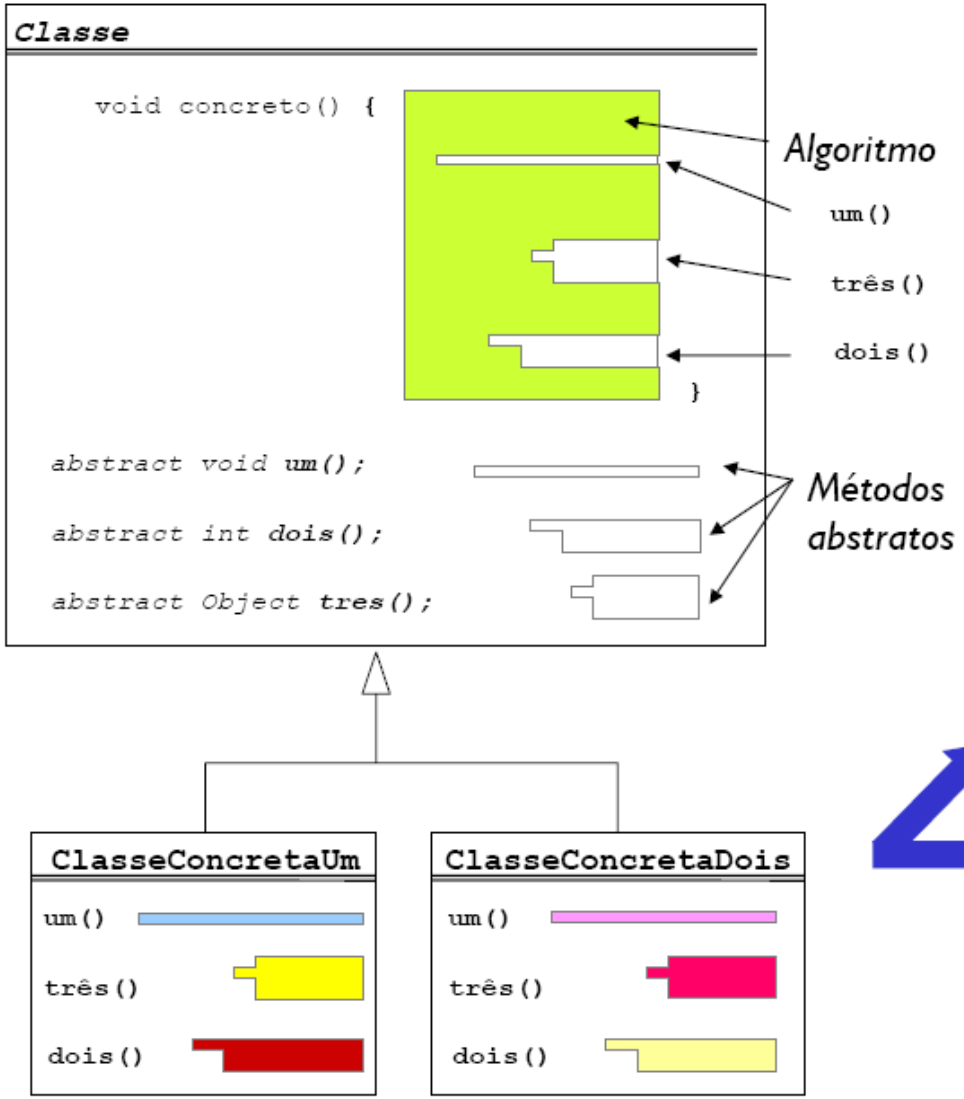
ybadoo

Template Method

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

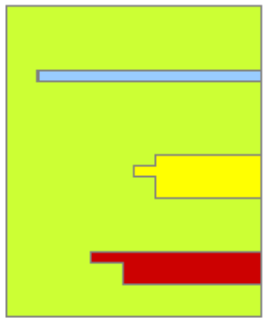
ybadoo

Problema

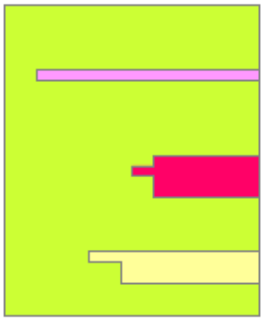


Algoritmos resultantes

```
Classe x =
  new ClasseConcretaUm()
x.concreto()
```



```
Classe x =
  new ClasseConcretaDois()
x.concreto()
```

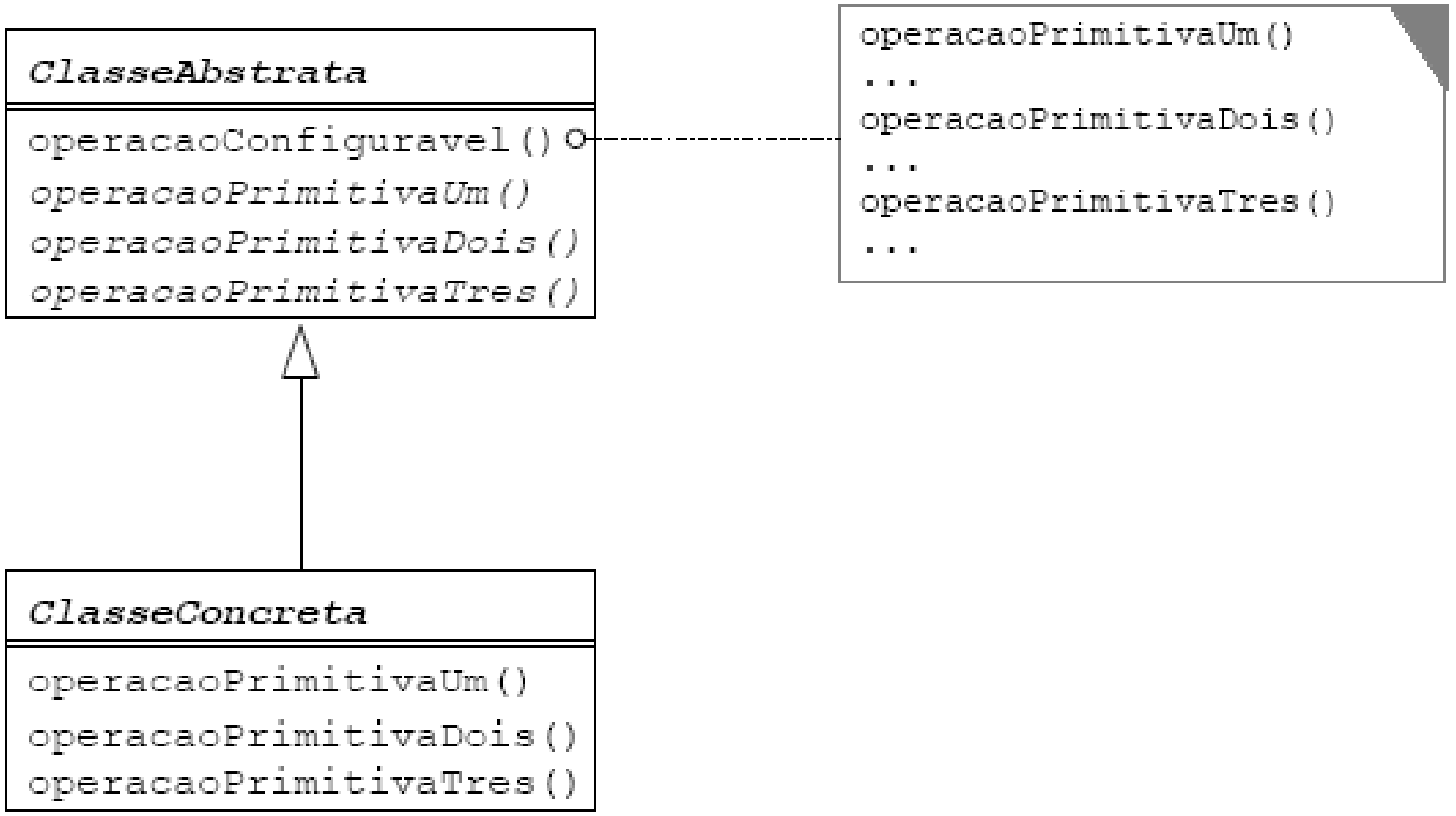


Solução

- O que é um *Template Method*:
  - Um *Template Method* define um algoritmo em termos de operações abstratas que subclasses sobrepõem 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.

ybadoo

Estrutura UML



Participantes

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

ybadoo

Chain of Responsibility

“Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.”

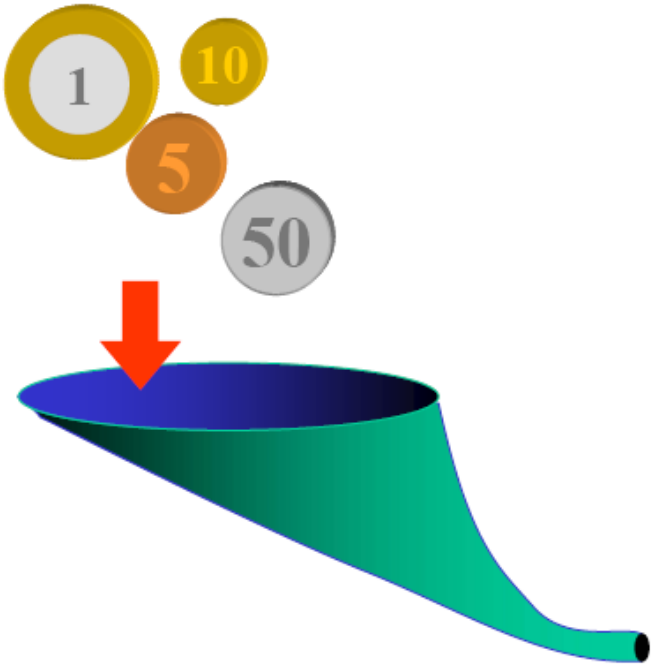
ybadoo

Problema (1/2)

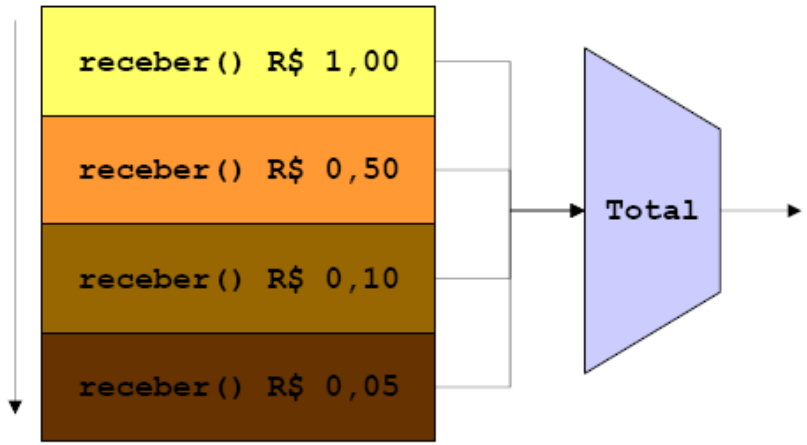
- Permitir que vários objetos possam servir a uma requisição ou repassá-la.
- Permitir divisão de responsabilidades de forma transparente.

ybadoo

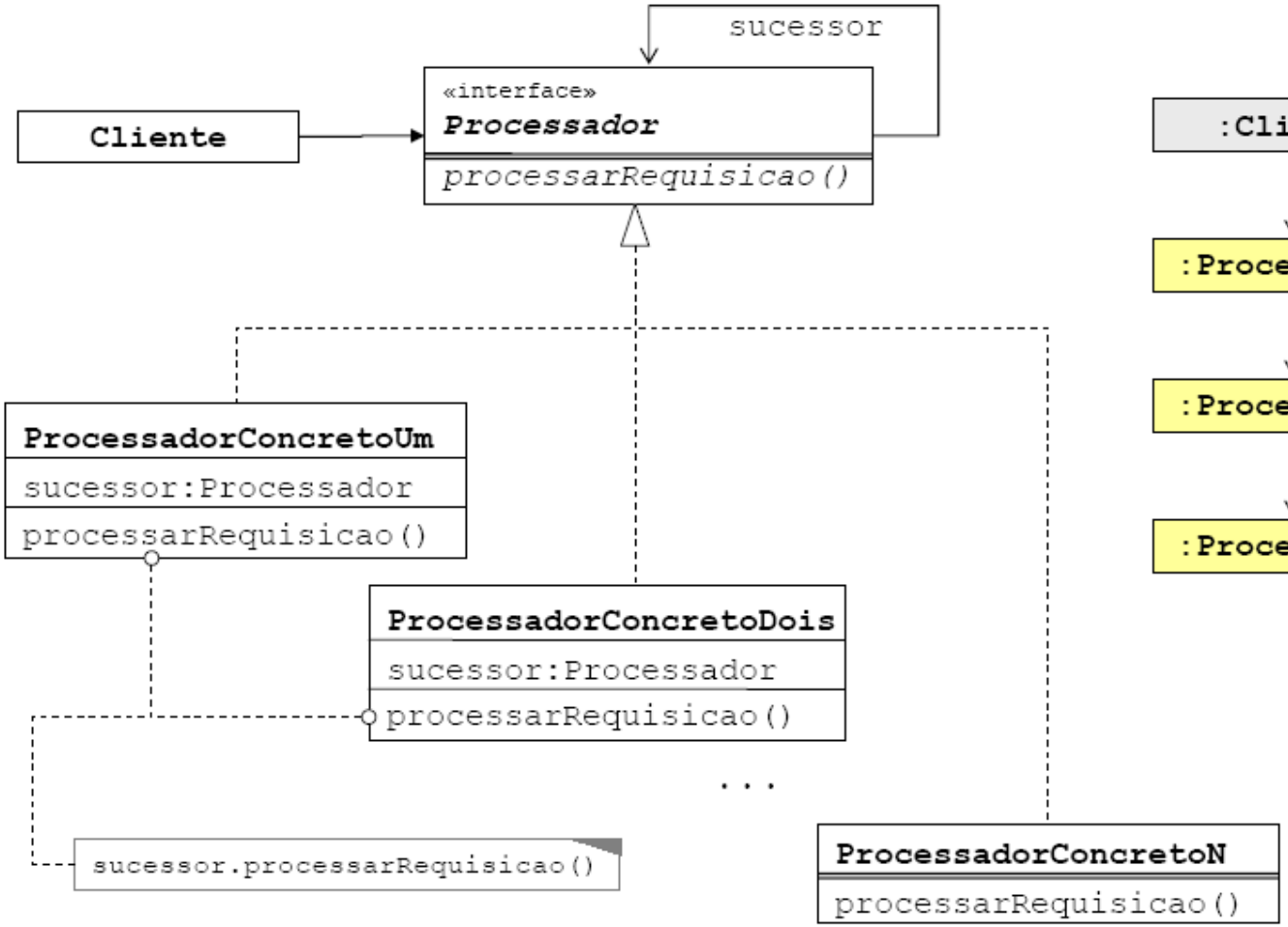
Problema (2/2)



*Um objeto pode ser uma **folha** ou uma **composição** de outros objetos*



Estrutura UML



Participantes

- *Handler*:
  - Define uma interface para tratar solicitações.
  - (opcional) implementa o elo (link) ao sucessor.
- *ConcreteHandler*:
  - Trata de solicitações pelas quais é responsável.
  - Pode acessar seu sucessor.
  - Se o *ConcreteHandler* pode tratar a solicitação, ele assim o faz; caso contrário, ele repassa a solicitação para o seu sucessor.
- *Client*:
  - Inicia a solicitação para um objeto ConcreteHandler da cadeia.

ybadoo

Estratégias

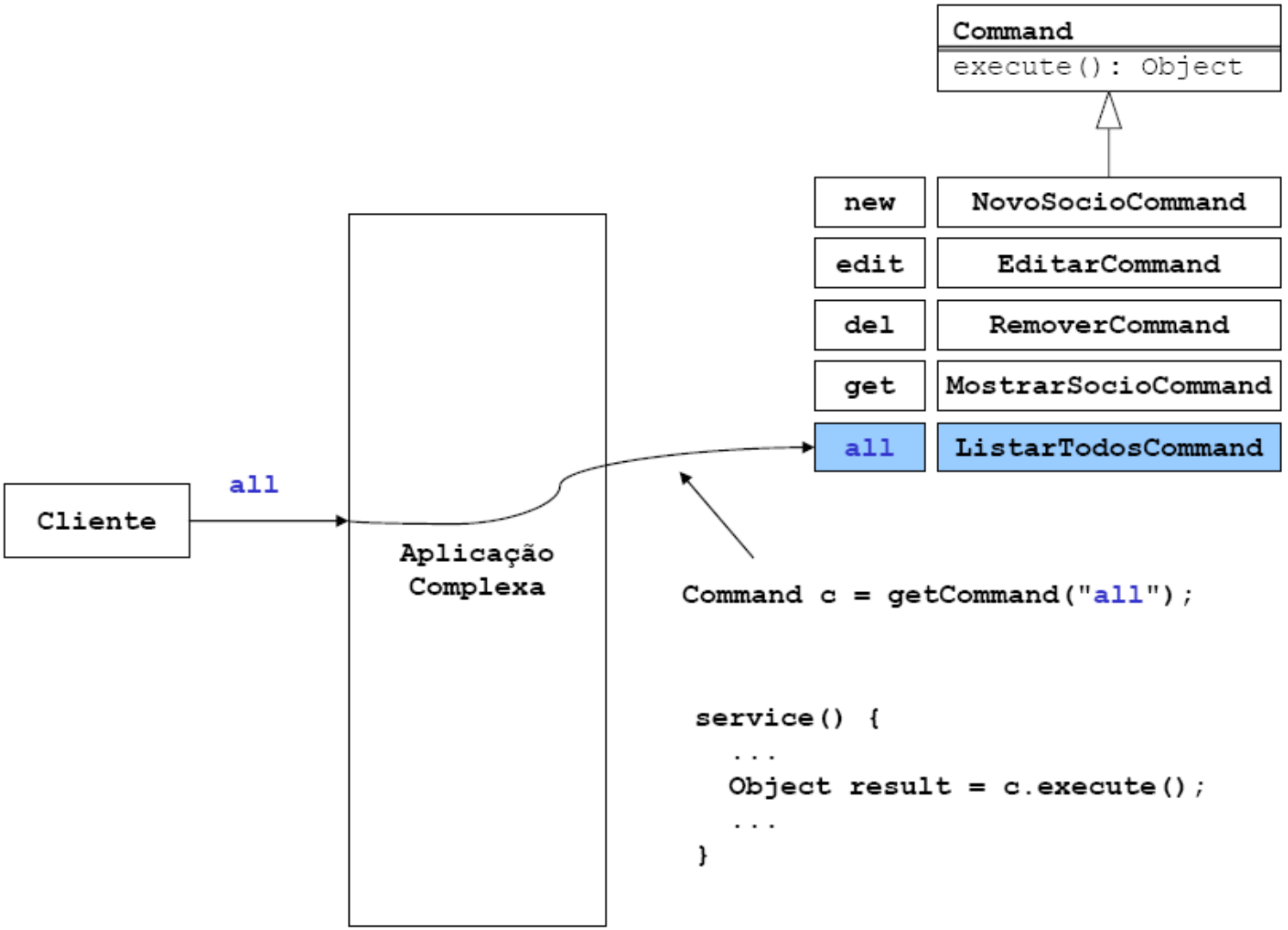
- Pode-se implementar um padrão de várias formas diferentes. Cada forma é chamada de estratégia.
- *Chain of Responsibility* pode ser implementada com estratégias que permitem maior ou menor acoplamento entre os participantes:
  - Usando um mediador:
    - Só o mediador sabe quem é o próximo participante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

ybadoo

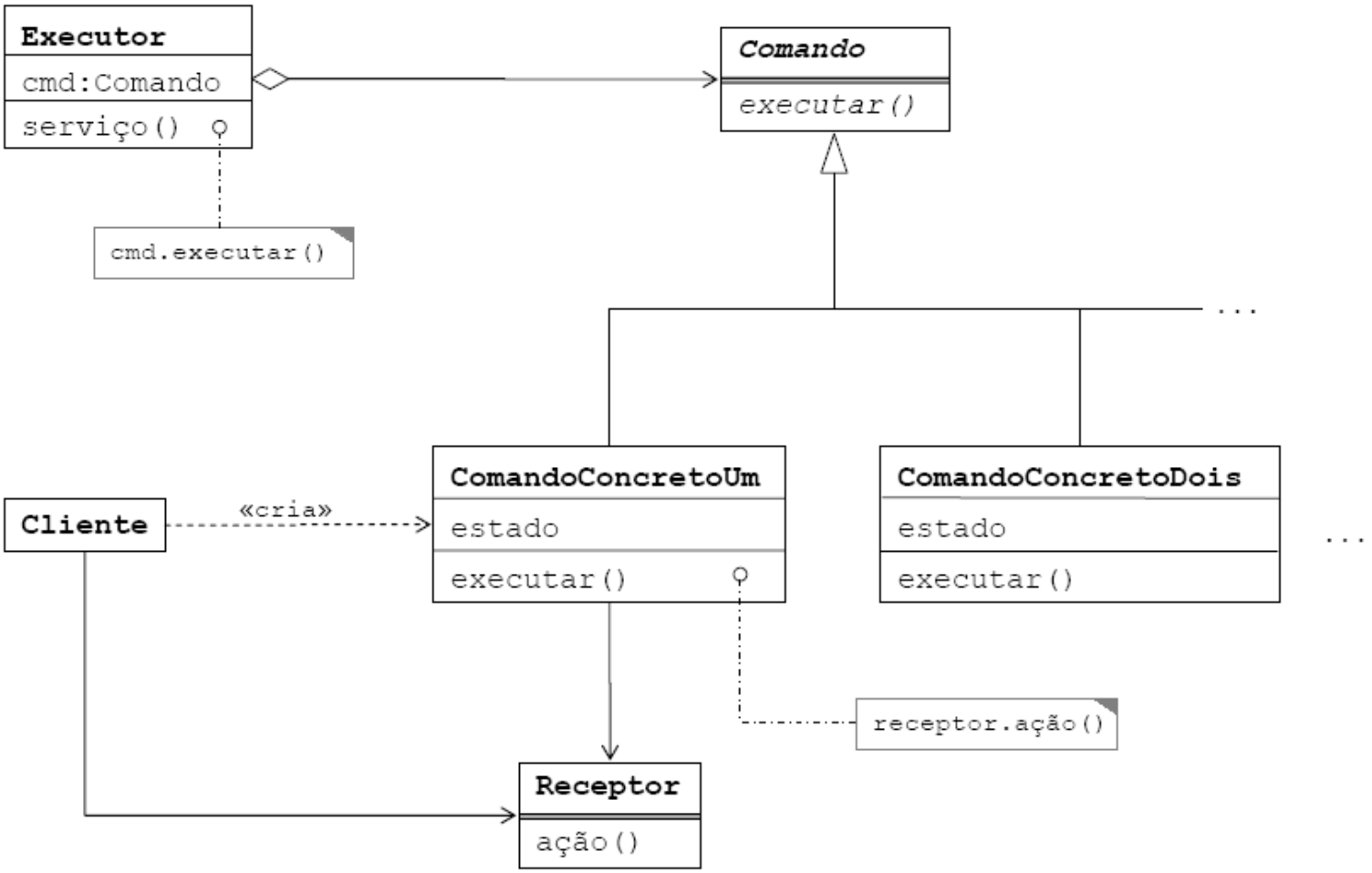
Command

“Encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas.”

Problema



Estrutura UML



Participantes (1/2)

- *Command*:
  - Declara uma interface para a execução de uma operação.
- *ConcreteCommand*:
  - Define uma vinculação entre um objeto *Receiver* e uma ação.
  - Implementa Execute através da invocação da(s) correspondente(s) operação(ões) no *Receiver*.
- *Client*:
  - Cria um objeto *ConcreteCommand* e estabelece o seu receptor (*Receiver*).

ybadoo

Participantes (2/2)

- *Invoker*:
  - Solicita ao *Command* a execução da solicitação.
- *Receiver*:
  - Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um *Receiver*.

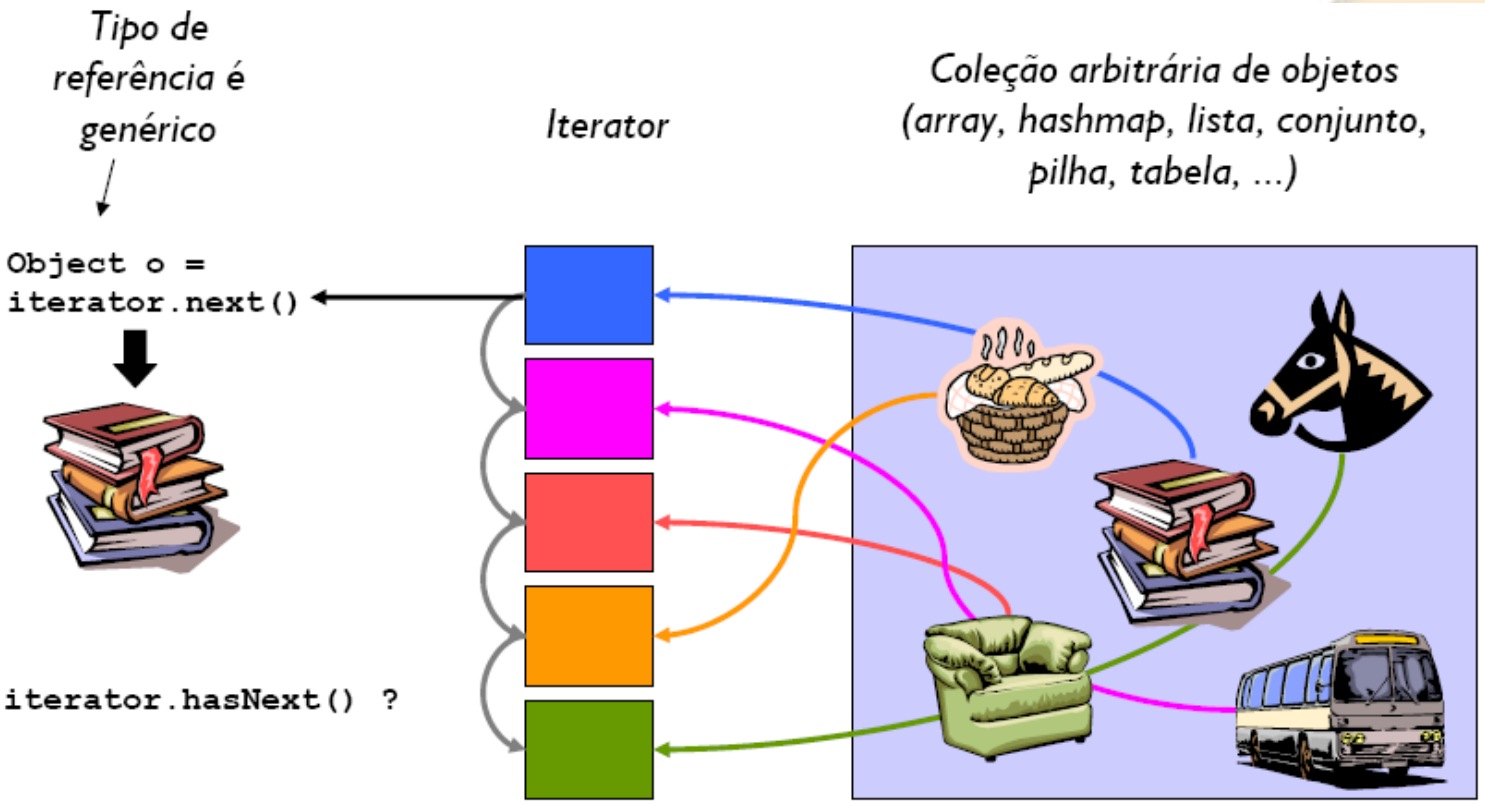
ybadoo

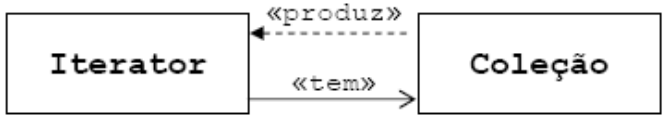
Iterator

“Fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente.”

ybadoo

Problema



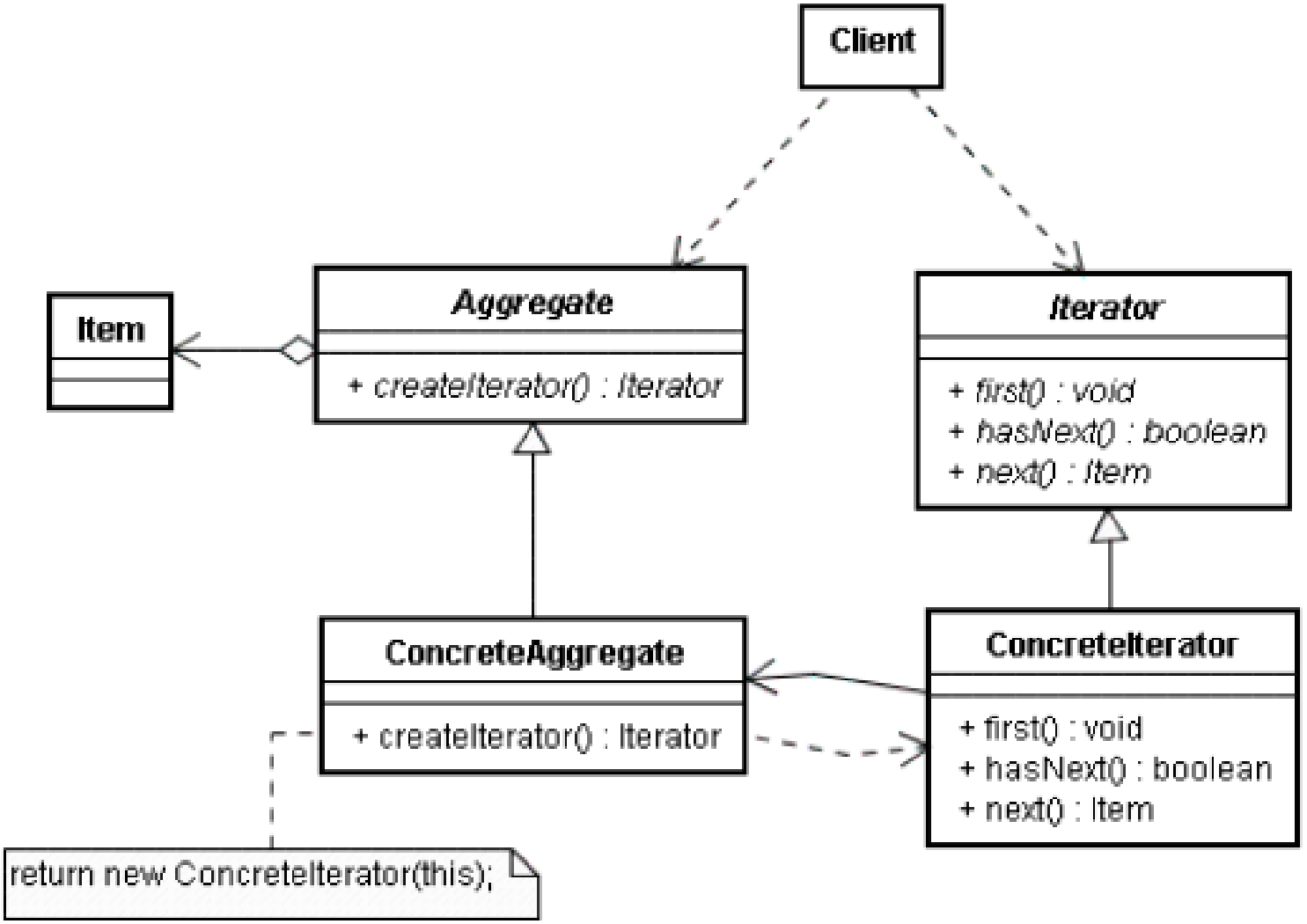


Para que serve?

- *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*.
  - *Iterator()* é um exemplo de *Factory Method*.

ybadoo

Estrutura UML



Participantes

- *Iterator*:
  - Define uma interface para acessar e percorrer elementos.
- *ConcreteIterator*:
  - Implementa a interface de *Iterator*.
  - Mantém o controle da posição corrente no percurso do agregado.
- *Aggregate*:
  - Define uma interface para a criação de um objeto *Iterator*.
- *ConcreteAggregate*:
  - Implementa a interface de criação do *Iterator* para retornar uma instância do *ConcreteIterator* apropriado.

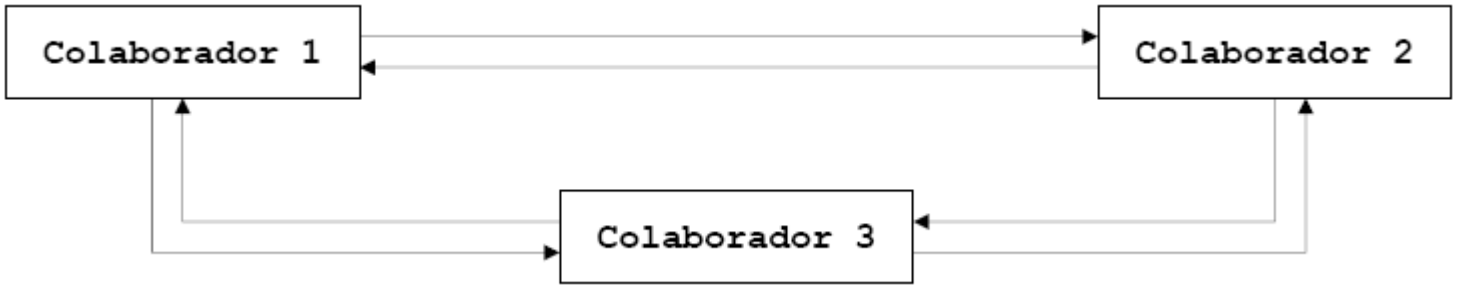
ybadoo

Mediator

“Definir um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente.”

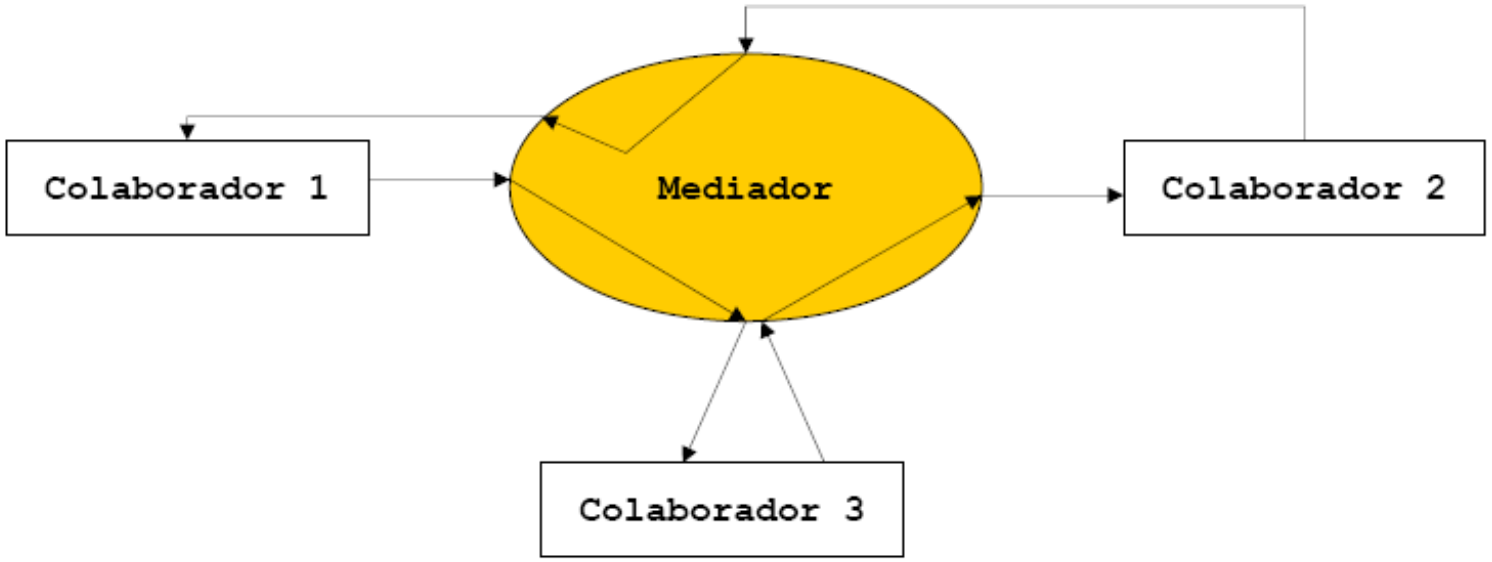
Problema

- 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.

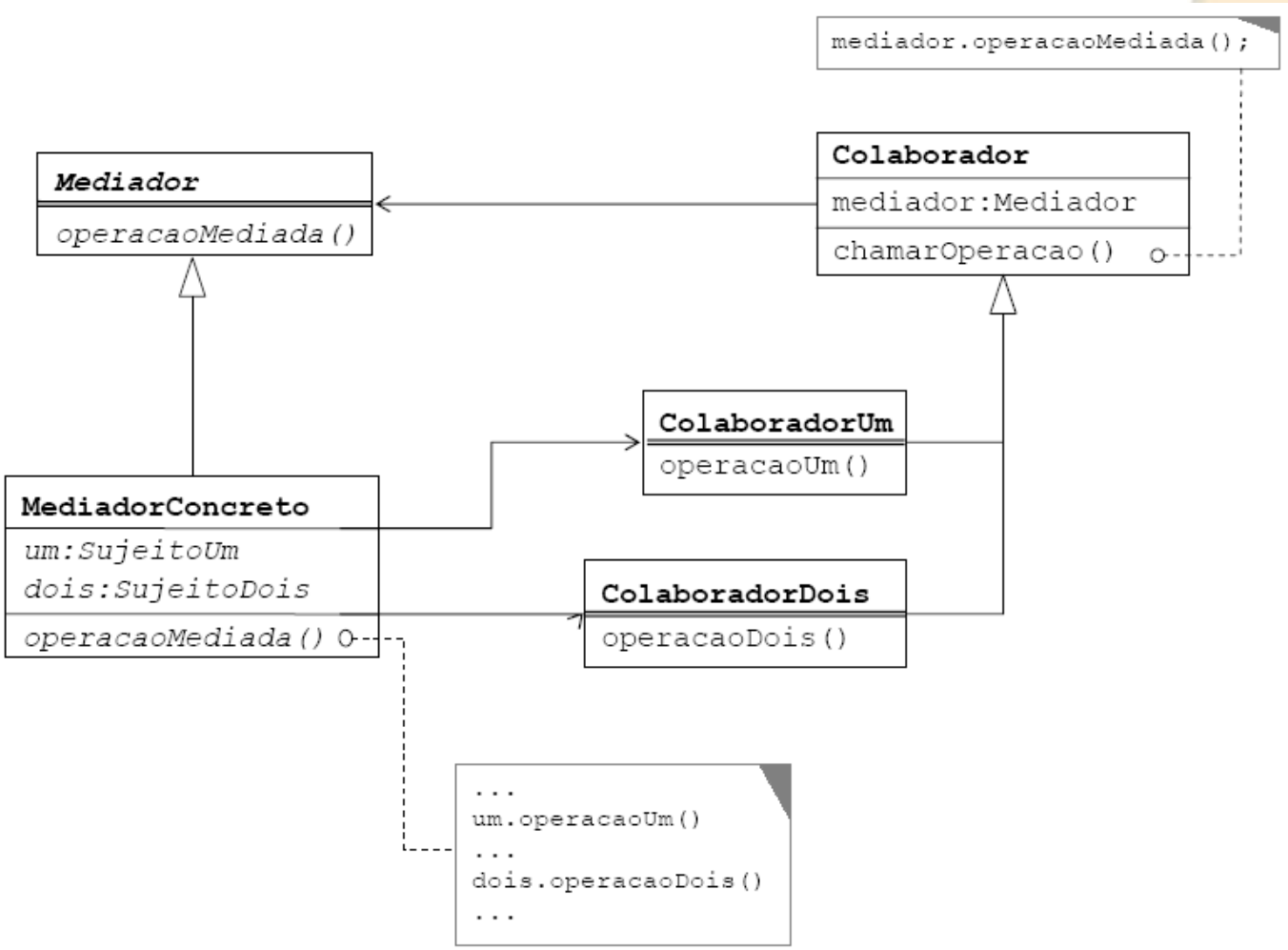


Descrição da solução

- 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).

ybadoo

Estrutura UML



Participantes

- *Mediator*:
  - Define uma interface para comunicação com objetos de classe *Colleague*.
- *ConcreteMediator*:
  - Implementa comportamento cooperativo através da coordenação de objetos de classe *Colleague*.
- *Colleague* classes:
  - Cada classe *Colleague* conhece seu objeto *Mediator* de outra forma.
  - Cada colega se comunica com o seu mediador sempre que, de outra forma, teria se comunicado com outro colega.

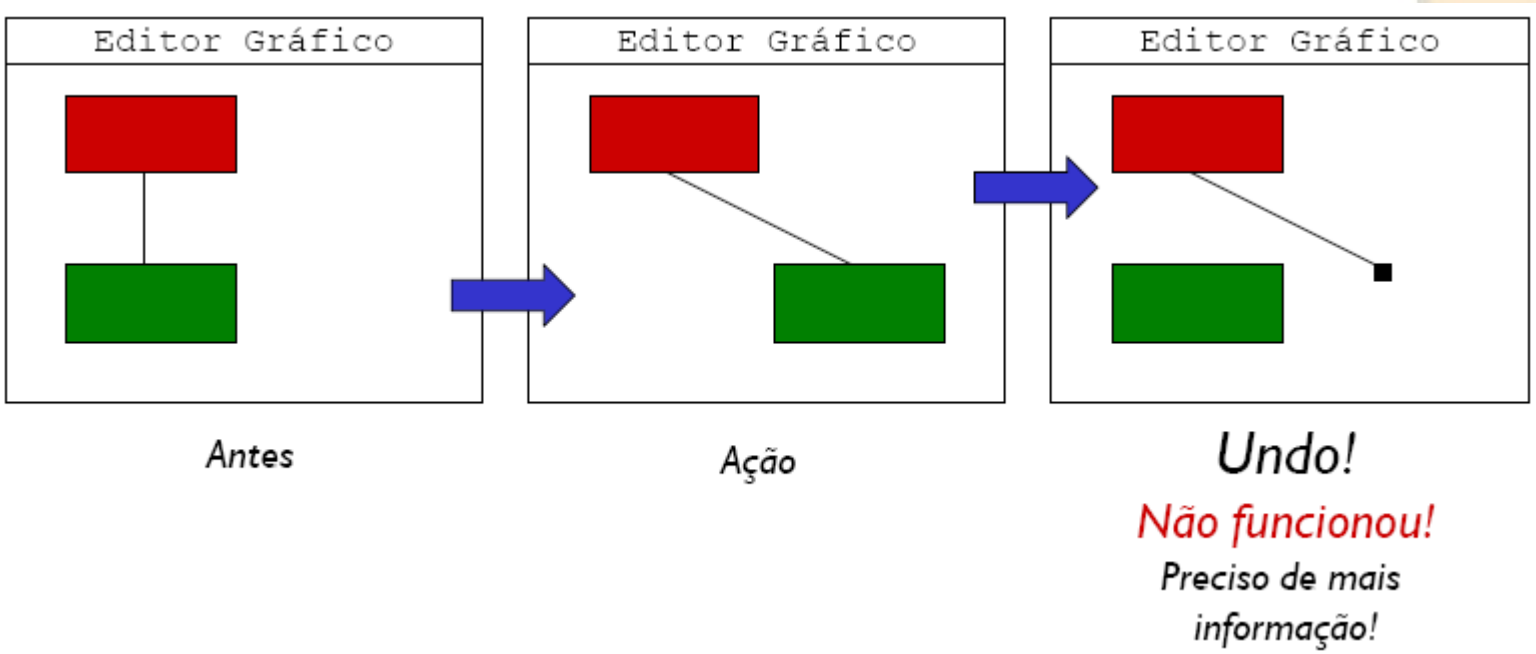
Memento

“Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde.”

ybadoo

Problema

- É preciso guardar informações sobre um objeto suficientes para desfazer uma operação, mas essas informações não devem ser públicas.



Solução

- Um memento é um pequeno repositório para guardar estado dos objetos:
  - Pode-se usar outro objeto, um *string*, um arquivo.
- Memento guarda um *snapshot* no estado interno de outro objeto – a Fonte:
  - Um mecanismo de Undo irá requisitar um memento da fonte quando ele necessitar verificar o estado desse objeto.
  - A fonte reinicializa o memento com informações que caracterizam seu estado atual.
  - Só a fonte tem permissão para recuperar informações do memento (o memento é opaco aos outros objetos).

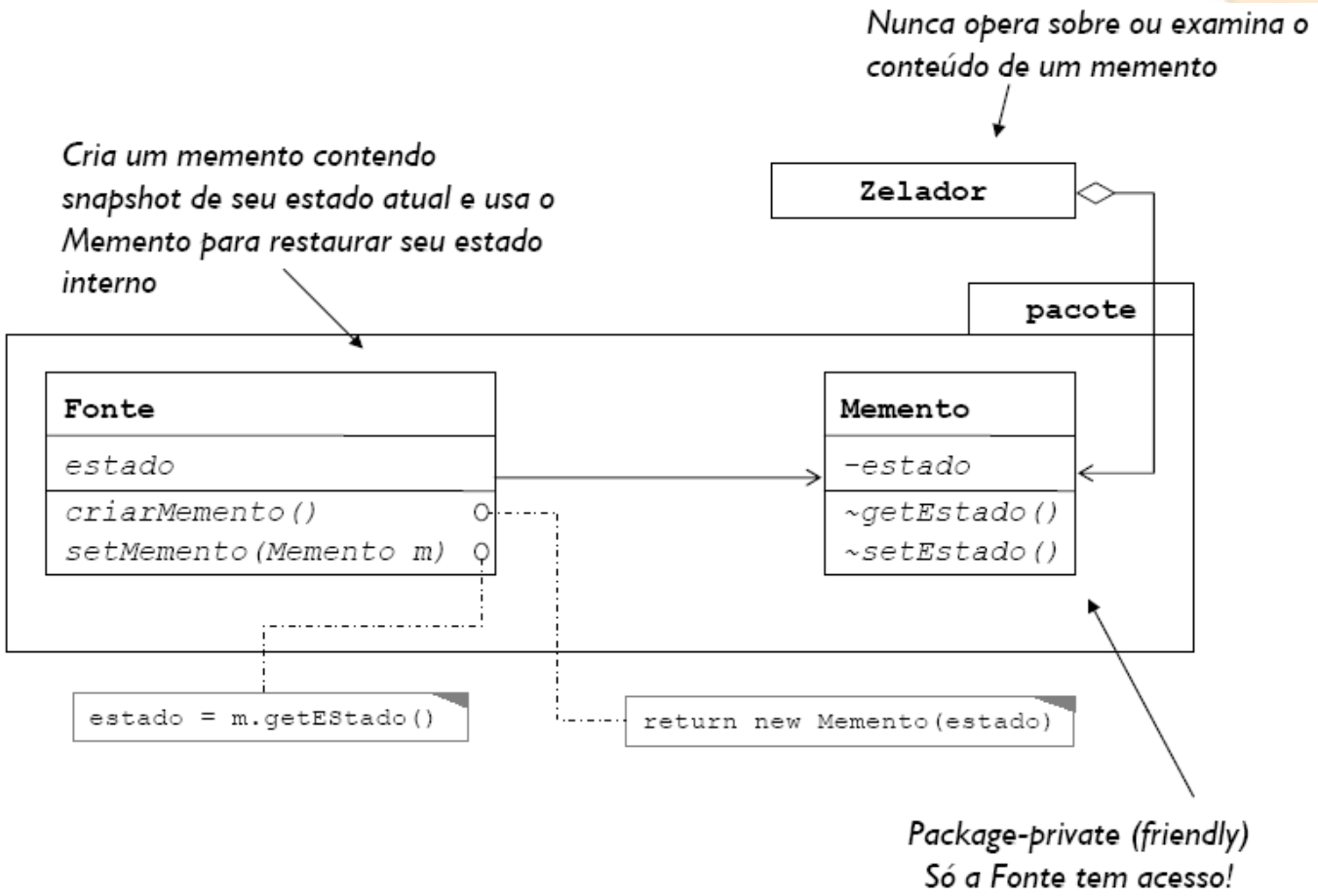
ybadoo

Quando usar?

- Use memento quando:
  - Um *snapshot* do (parte do) estado de um objeto precisa ser armazenada para que ele possa ser restaurado ao seu estado original posteriormente.
  - Uma interface direta para se obter esse estado iria expor detalhes de implementação e quebrar o encapsulamento do objeto.

ybadoo

Estrutura UML



Participantes

- *Memento*:
  - Armazena o estado interno do objeto *Originator*.
  - O memento pode armazenar pouco ou muito do estado interno do originador, conforme necessário e segundo critérios do seu originador.
- *Originator*:
  - Cria um memento contendo um instantâneo do seu estado interno corrente.
  - Usa o memento para restaurar o seu estado interno.
- *Caretaker*:
  - É responsável pela custódia do memento.
  - Nunca opera ou examina os conteúdos de um memento.

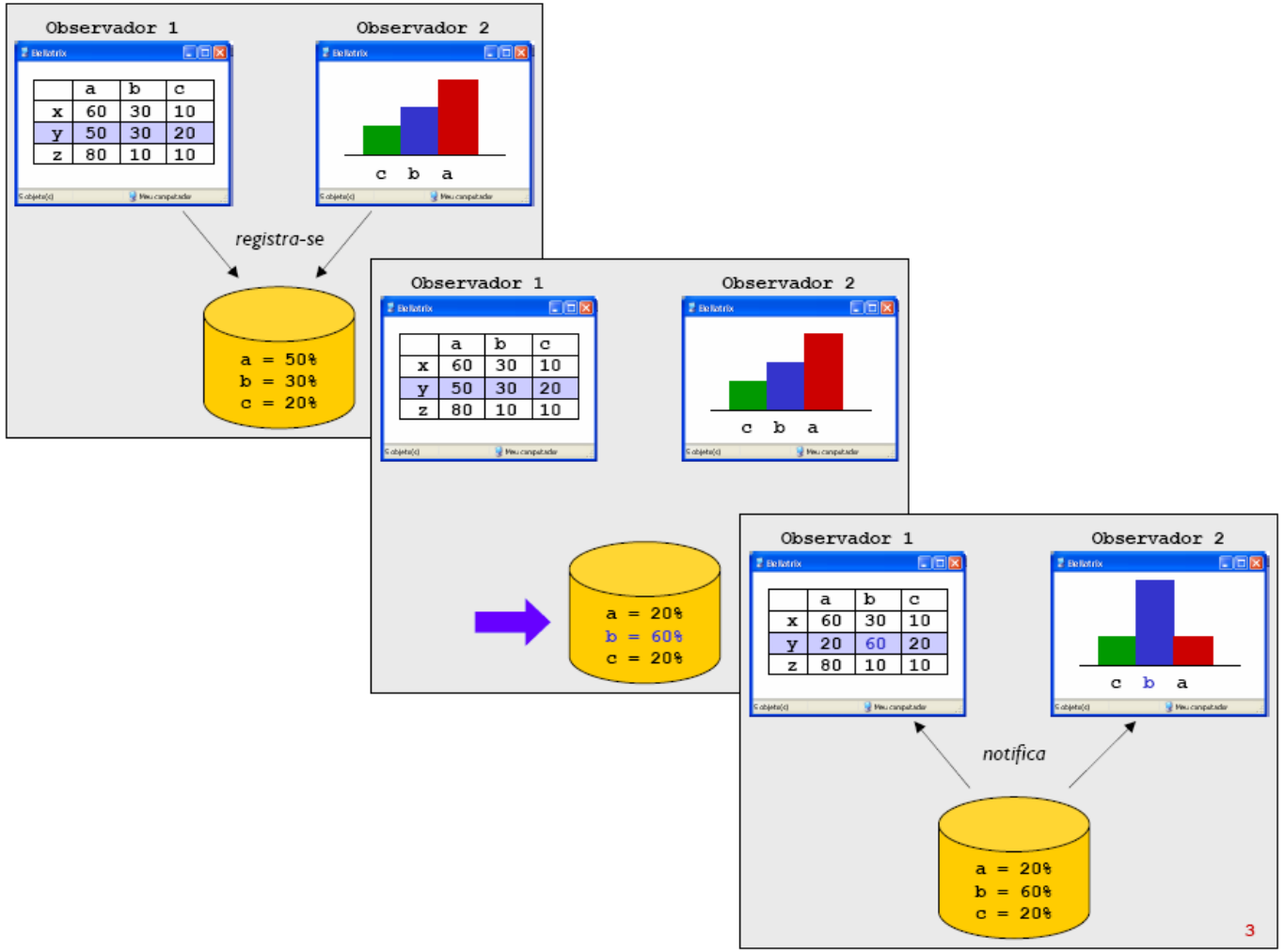
ybadoo

Observer

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

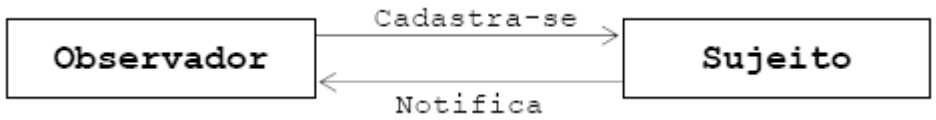
ybadoo

Problema (1/2)



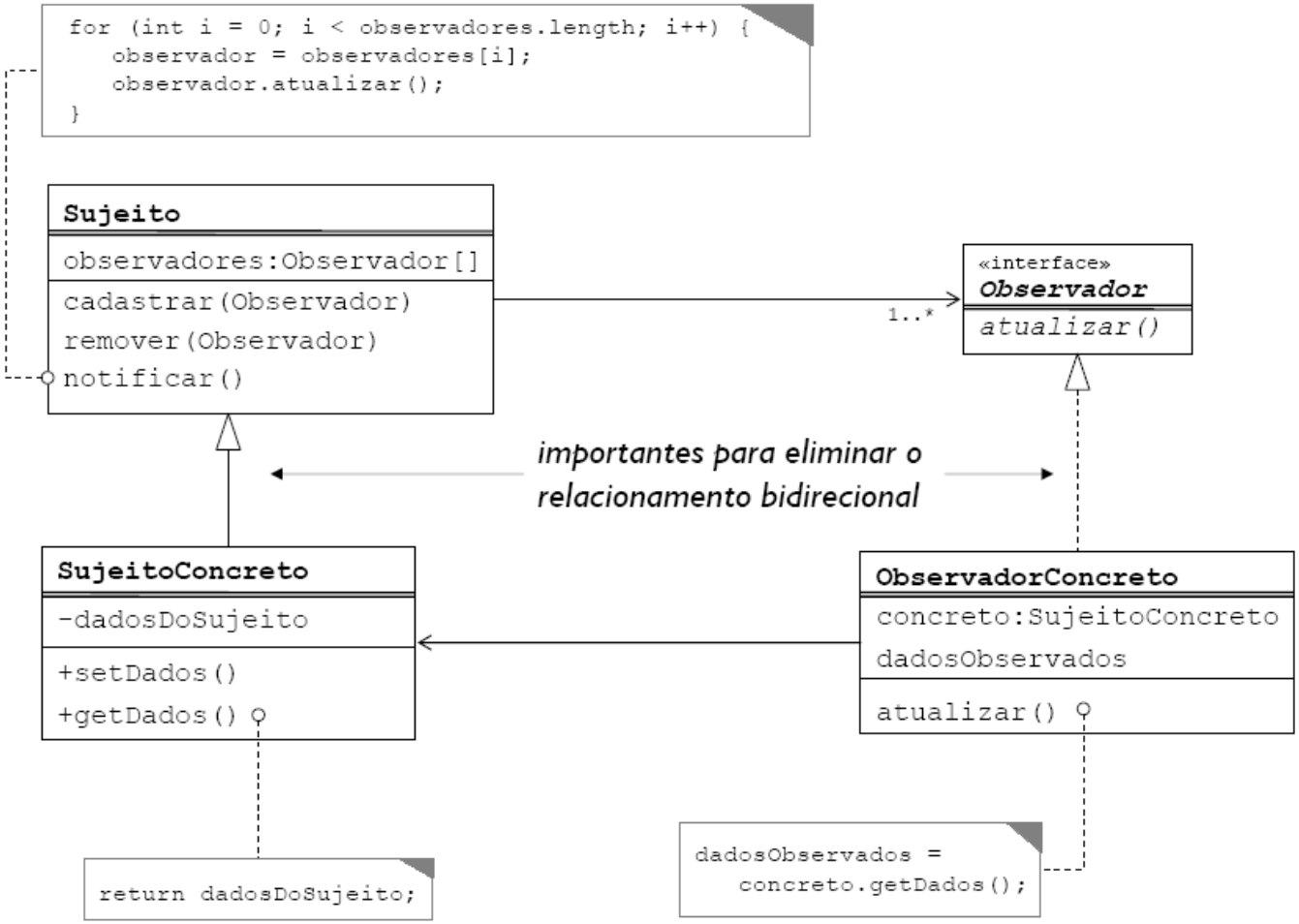
Problema (2/2)

- 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?



ybadoo

Estrutura UML



Participantes (1/2)

- *Subject*:
  - Conhece os seus observadores.
  - Um número qualquer de objetos *Observer* pode observar um *subject*.
  - Fornece uma interface para acrescentar e remover objetos para associar e desassociar objetos *observer*.
- *Observer*:
  - Define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um *Subject*.

ybadoo

Participantes (2/2)

- *ConcreteSubject*:
  - Armazena estados de interesse para objetos *ConcreteObserver*.
  - Envia uma notificação para os seus observadores quando seu estado muda.
- *ConcreteObserver*:
  - Mantém uma referência para um objeto *ConcreteSubject*.
  - Armazena estados que deveriam permanecer consistentes com os do *Subject*.
  - Implementa a interface de atualização de *Observer*, para manter seu estado consistente com o do *subject*.

ybadoo

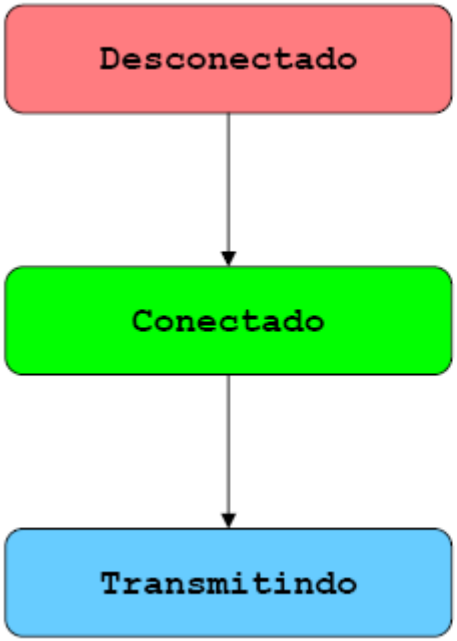
State

“Permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe.”

ybadoo

Problema

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



:Objeto

operação

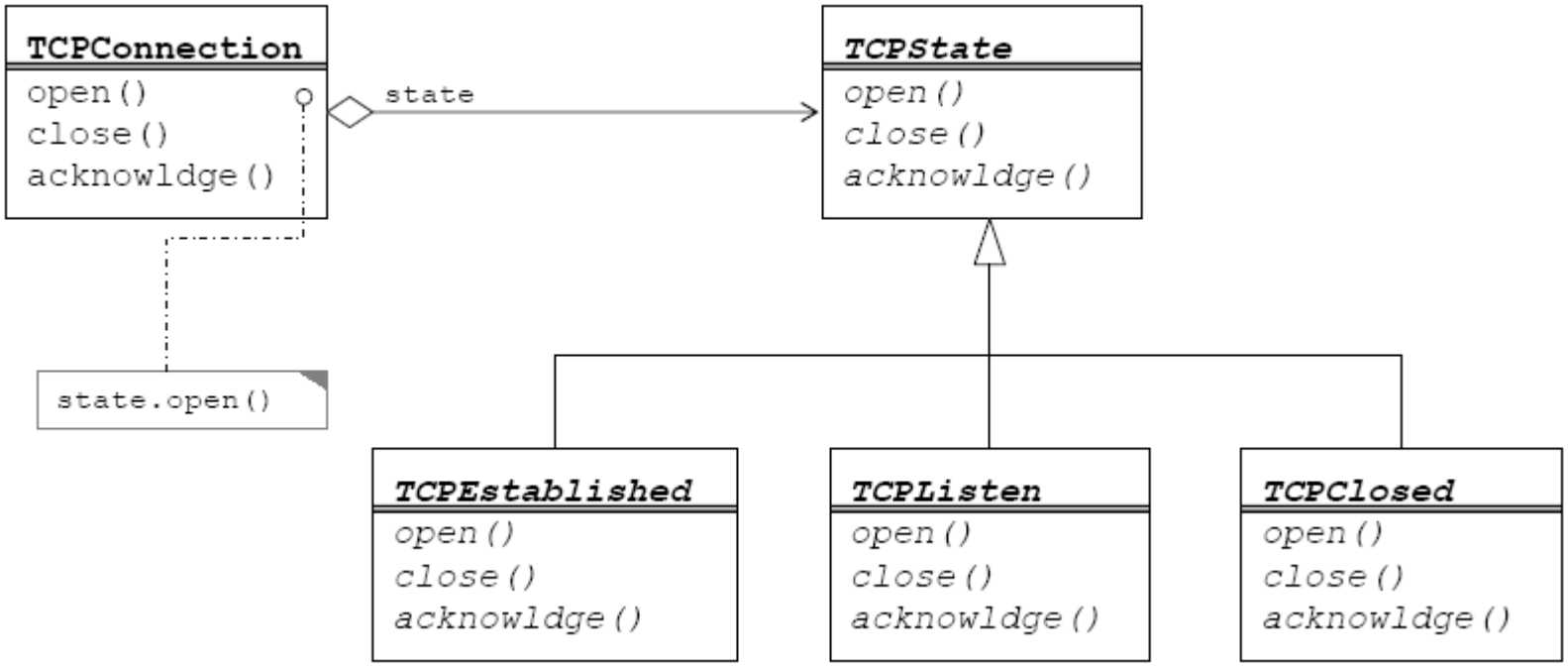
```
if (estado == desconectado) {
  façaIsto();
} else if (estado == conectado) {
  façaAquilo();
} else {
  faça();
}
```

operação

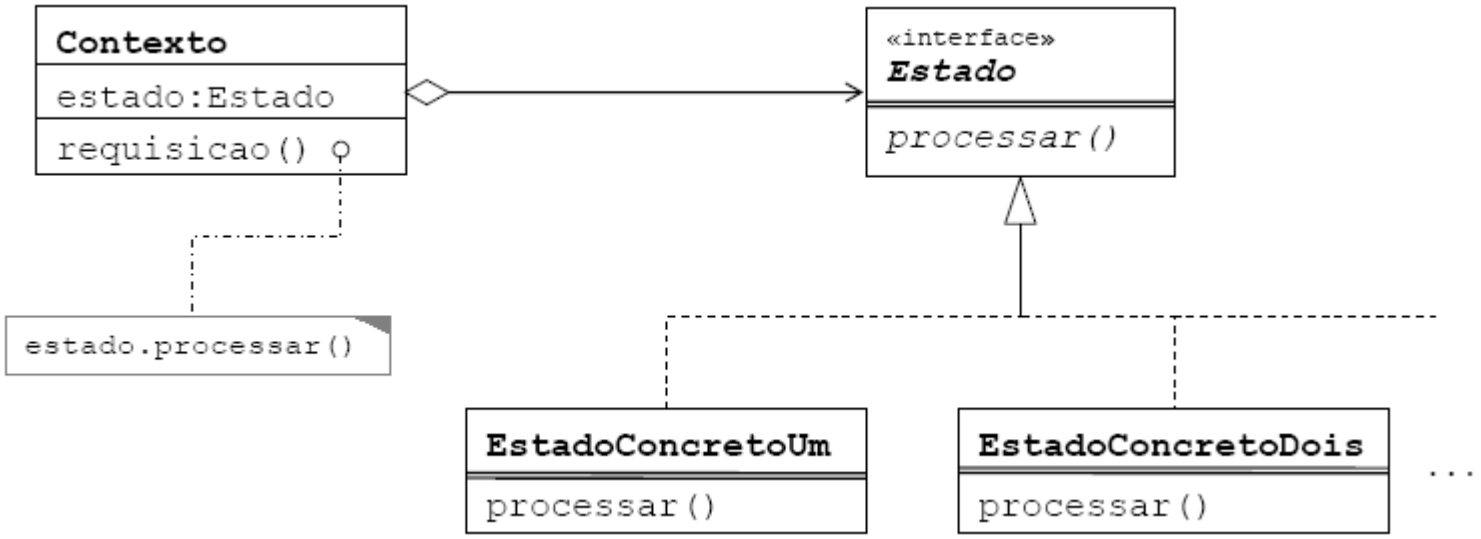
```
estado.faça()
```

Exemplo [GoF]

- Sempre que a aplicação mudar de estado, o objeto *TCPConnection* muda o objeto *TCPState* que está usando.



Estrutura UML



Participantes

- *Context*:
  - Define a interface de interesse para os clientes.
  - Mantém uma instância de uma subclasse *ConcreteState* que define o estado corrente.
- *State*:
  - Define uma interface para encapsulamento associado com um determinado estado do *Context*.
- *ConcreteState*:
  - Cada subclasse implementa um comportamento associado com um estado do *Context*.

ybadoo

Strategy

“Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.”

ybadoo

Problema

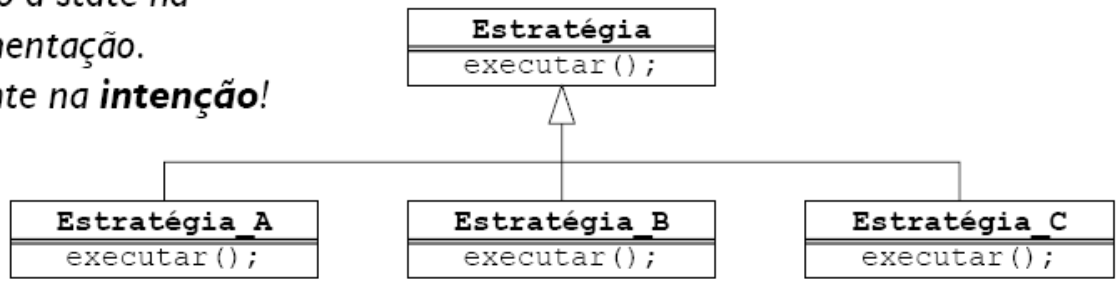
Várias estratégias, escolhidas de acordo com opções ou condições

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

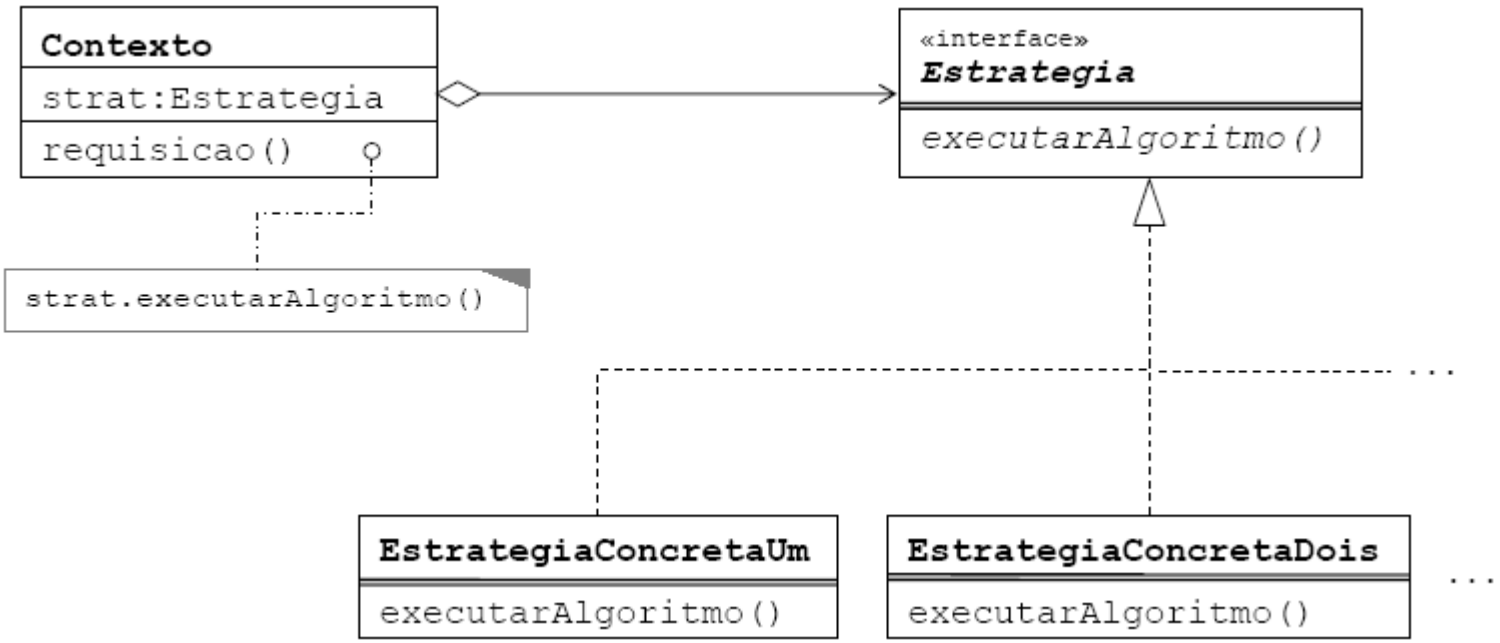
```
if (guerra && inflação > META) {
   plano = new Estrategia_C();
} else if (guerra && recessão) {
   plano = new Estrategia_B();
} else {
   plano = new Estrategia_A();
}
```

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

Idêntico a state na implementação.
*Diferente na **intenção**!*



Estrutura UML



Participantes

- *Strategy*:
  - Define uma interface comum para todos os algoritmos suportados.
  - *Context* usa esta interface para chamar o algoritmo definido por uma *ConcreteStrtegy*.
- *ConcreteStrategy*:
  - Implementa o algoritmo usando a interface de *Strategy*.
- *Context*:
  - É configurado com um objeto *ConcreteStrategy*.
  - Mantém uma referência para um objeto *Strategy*.
  - Pode definir uma interface que permite a *Strategy* acessar seus dados.

ybadoo

Quando usar?

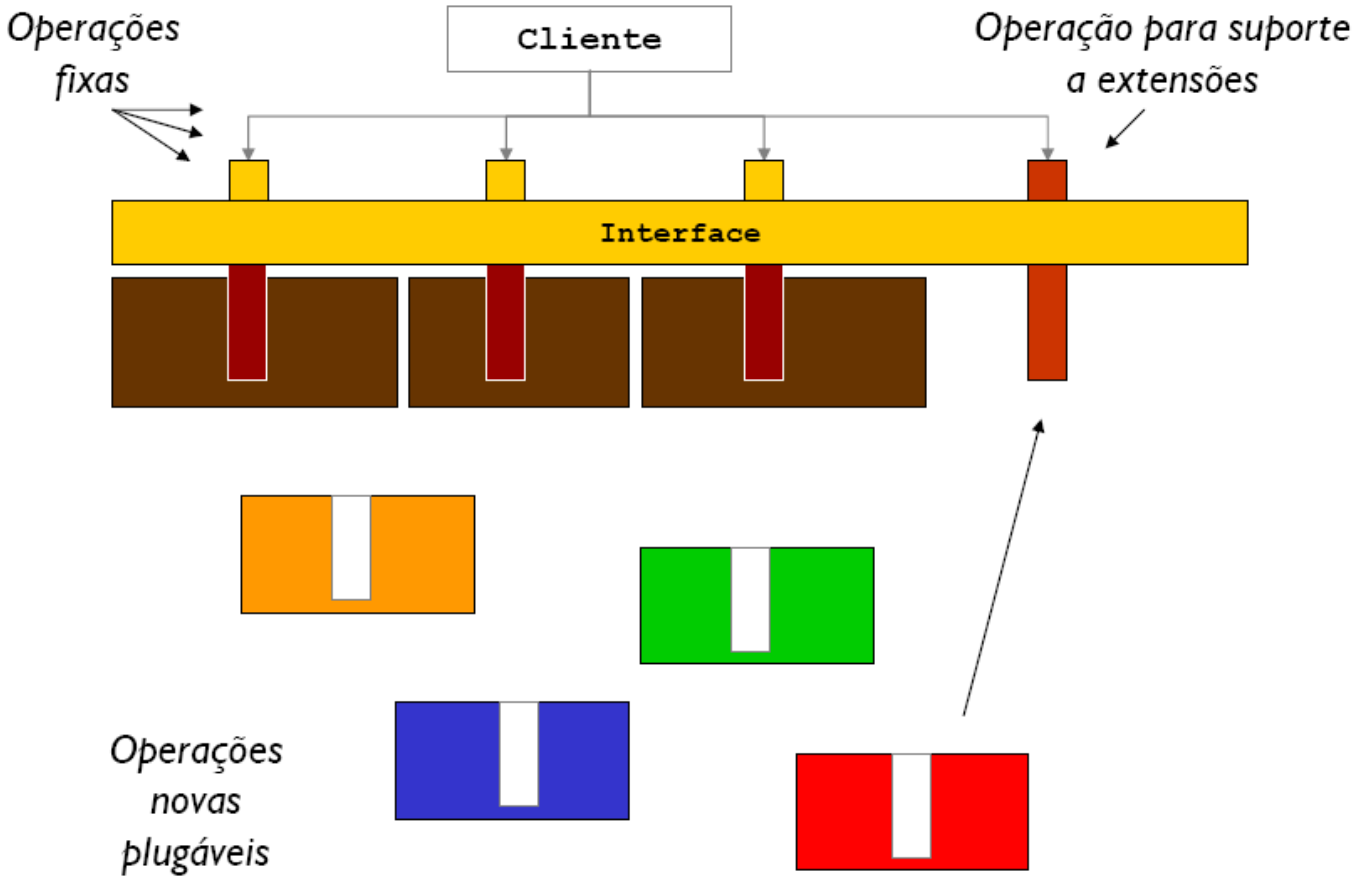
- Quando classes relacionadas forem diferentes apenas no seu comportamento:
  - *Strategy* oferece um meio para configurar a classe com um entre vários comportamentos.
- Quando você precisar de diferentes variações de um mesmo algoritmo.
- Quando um algoritmo usa dados que o cliente não deve conhecer.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

ybadoo

Visitor

“Representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.”

Problema

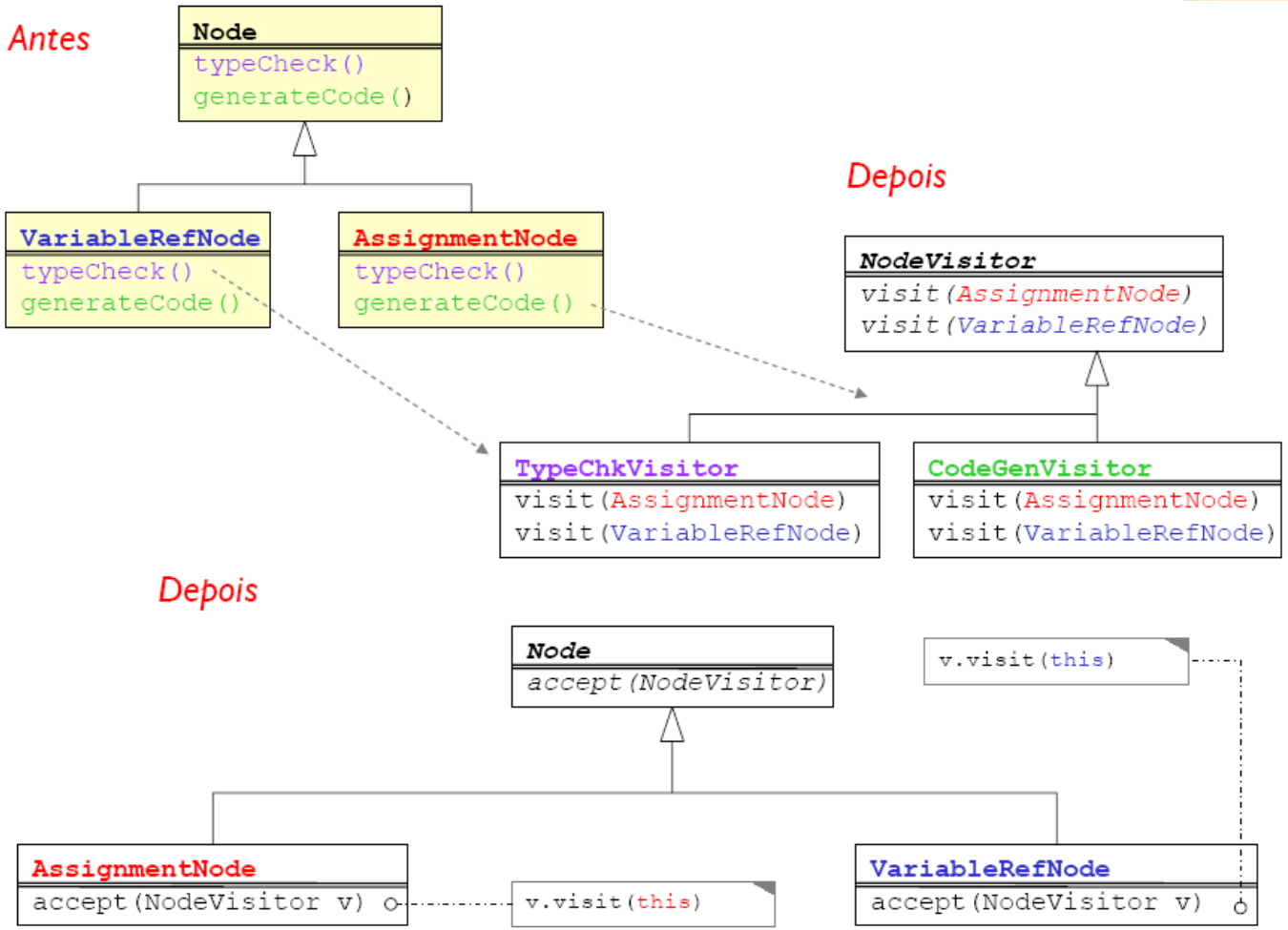


Para que serve?

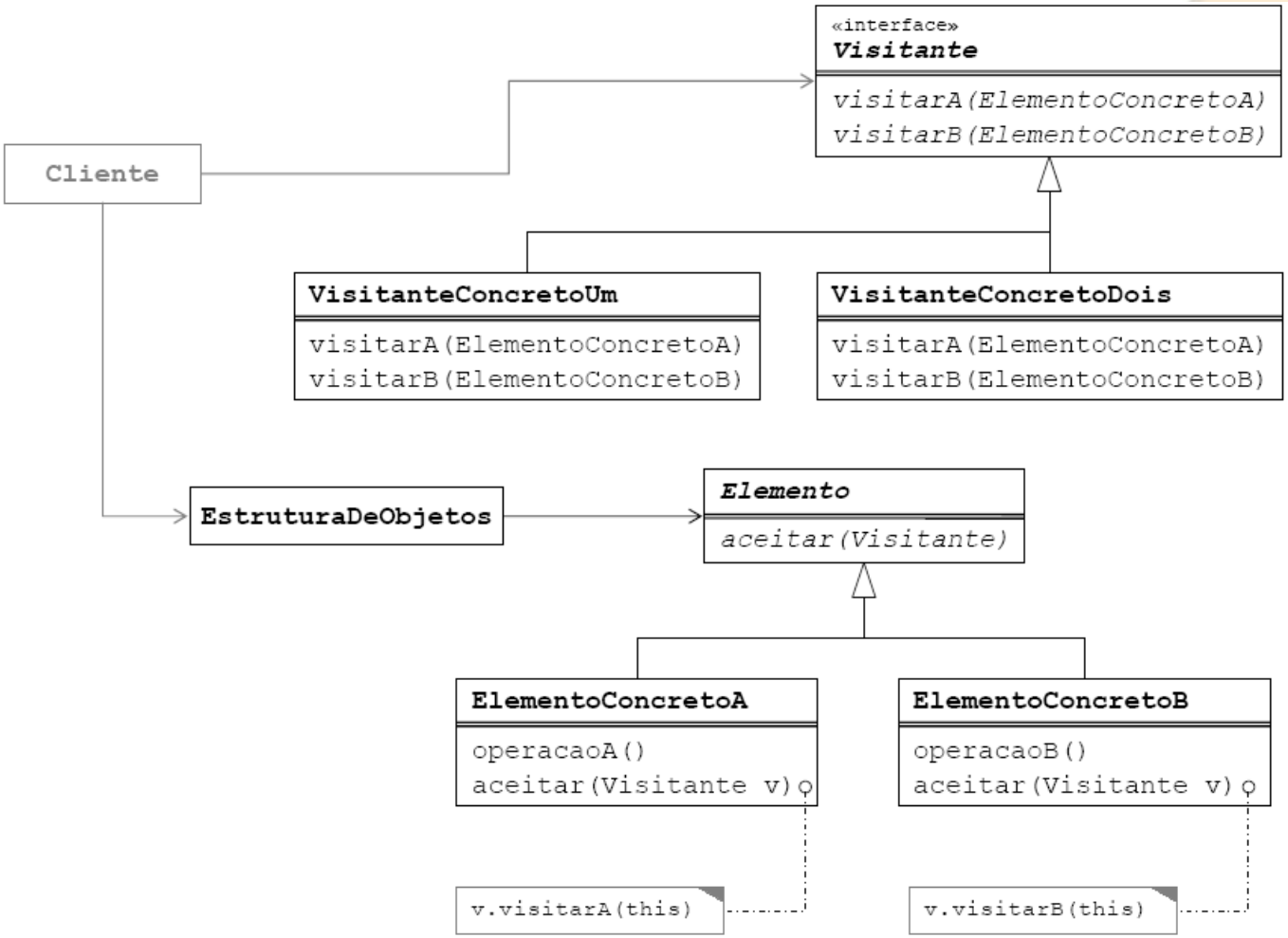
- *Visitor* permite:
  - Plugar nova funcionalidade em objetos sem precisar mexer na estrutura de herança.
  - Agrupar e manter operações relacionadas em uma classe e aplicá-las, quando conveniente, a outras classes (evitar espalhamento e fragmentação de interesses).
  - Implementar um *Iterator* para objetos não relacionados através de herança.

ybadoo

Exemplo [GoF]



Estrutura UML



Participantes (1/3)

- *Visitor*:
  - Declara uma operação *Visit* para cada classe *ConcreteElement* na estrutura do objeto.
  - O nome e a assinatura da operação identifica a classe que envia a solicitação *Visit* ao visitante.
  - Isto permite ao visitante determinar a classe concreta do elemento que está sendo visitado.
  - Então, o visitante pode acessar o elemento diretamente através da sua interface específica.
- *Element*:
  - Define uma operação *Accept* que aceita um visitante como um argumento.

Participantes (2/3)

- *ConcreteVisitor*:
  - Implementa cada operação declarada por *Visitor*.
  - Cada operação implementa um fragmento do algoritmo definido para a correspondente classe de objeto na estrutura.
  - *ConcreteVisitor* fornece o contexto para o algoritmo e armazena o seu estado local.
  - Este estado, frenquentemente, acumula resultados durante o percurso da estrutura.
- *ConcreteElement*:
  - Implementa uma operação *Accept* que aceita um visitante como um argumento.

Participantes (3/3)

- *ObjectStructure*:
  - Pode enumerar seus elementos.
  - Pode fornecer uma interface de alto nível para permitir ao visitante visitar seus elementos.
  - Pode ser ou um composto, ou uma coleção, tal como uma lista ou um conjunto.

ybadoo

Prós e contras

- Vantagens:
  - Facilita a adição de novas operações.
  - Agrupa operações relacionadas e separa operações não relacionadas: reduz espalhamento de funcionalidades e embaralhamento.
- Desvantagens:
  - Dá trabalho adicionar novos elementos na hierarquia: requer alterações em todos os *Visitors*.
  - Se a estrutura muda com frequência, não use!
  - Quebra de encapsulamento: métodos e dados usados pelo *Visitor* têm de estar acessíveis.

ybadoo

Resumo (1/3)

- *Interpreter*:
  - Para realizar composição com comandos e desenvolver uma linguagem de programação usando objetos.
- *Template Method*:
  - Para compor um algoritmo feito por métodos abstratos que podem ser completados em subclasses.
- *Chain of Responsibility*:
  - Quando uma requisição puder ou precisar ser tratada por um ou mais entre vários objetos.
- *Command*:
  - Para representar um comando (ação imperativa do cliente).

Resumo (2/3)

- *Iterator*:
  - Para navegar em uma coleção elemento por elemento.
- *Mediator*:
  - Para controlar a interação entre dois objetos independentes.
- *Memento*:
  - Para armazenar o estado de um objeto sem quebrar o encapsulamento.
  - O uso típico deste padrão é na implementação de operações de Undo.
- *Observer*:
  - Quando houver necessidade de notificação automática.

ybadoo

Resumo (3/3)

- *State*:
  - Para representar o estado de um objeto.
- *Strategy*:
  - Para representar um algoritmo (comportamento).
- *Visitor*:
  - Para estender uma aplicação com novas operações sem que seja necessário mexer na interface existente.

ybadoo