Programação Orientada a Objetos

Padrões Estruturais

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.

ybadoo

Classificação dos padrões do GoF

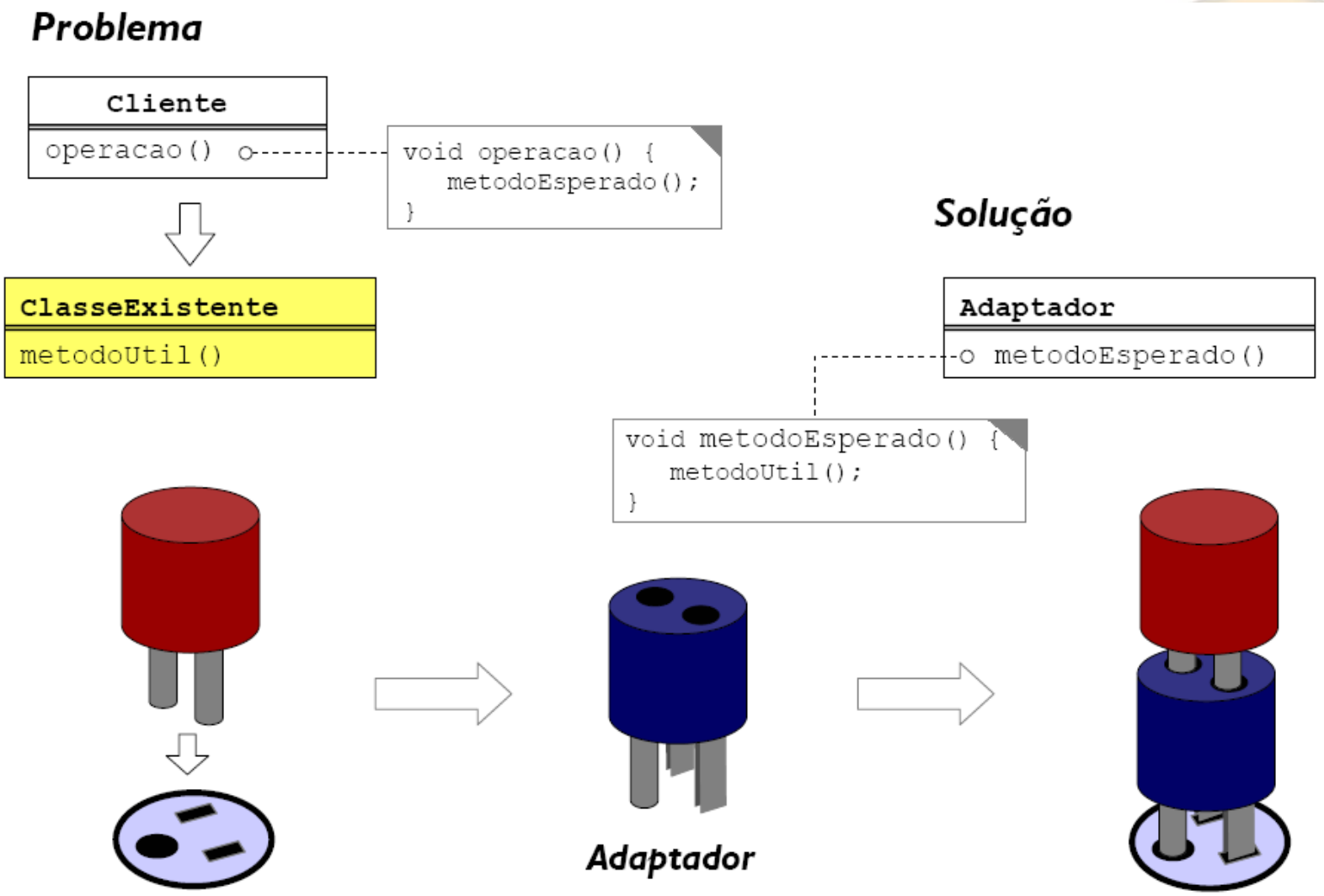
		Propósito		
		1. Criação	*2. Estrutura*	*3. Comportamento*
Escopo	Classe	*Factory Method*	*Class Adapter*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Object Adapter* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility*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

Adapter

"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."

ybadoo

Problema e solução

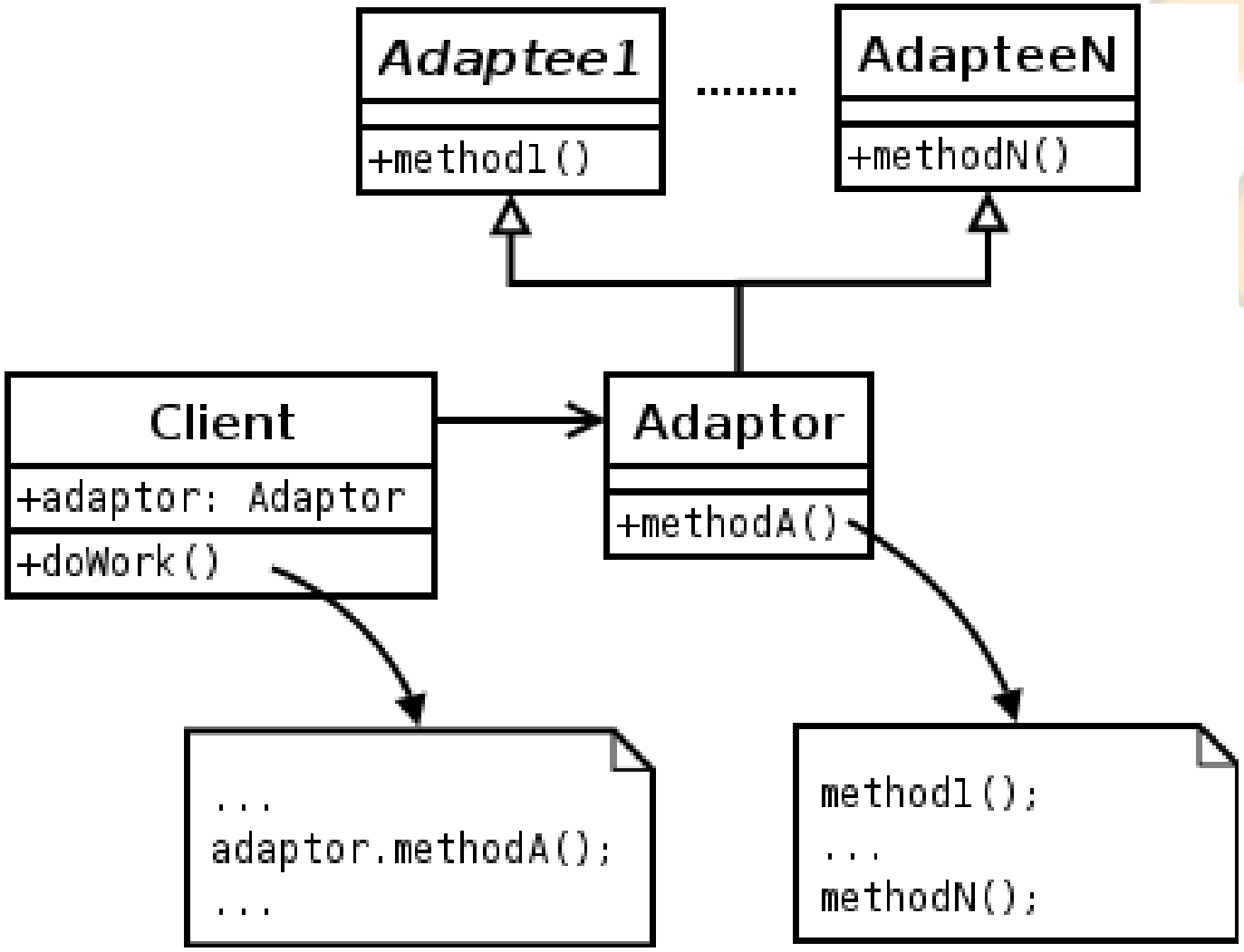


Estrutura

- *Class Adapter*: usa herança múltipla.
- *Client*:
  - Colabora com objetos compatíveis com a interface de *Target*.
- *Target*:
  - Define a interface específica do domínio que *Client* usa.
- *Adaptee*:
  - Define uma interface existente que necessita ser adaptada.
- *Adapter*:
  - Adapta a interface do *Adaptee* à interface de *Target*.

ybadoo

Class Adapter

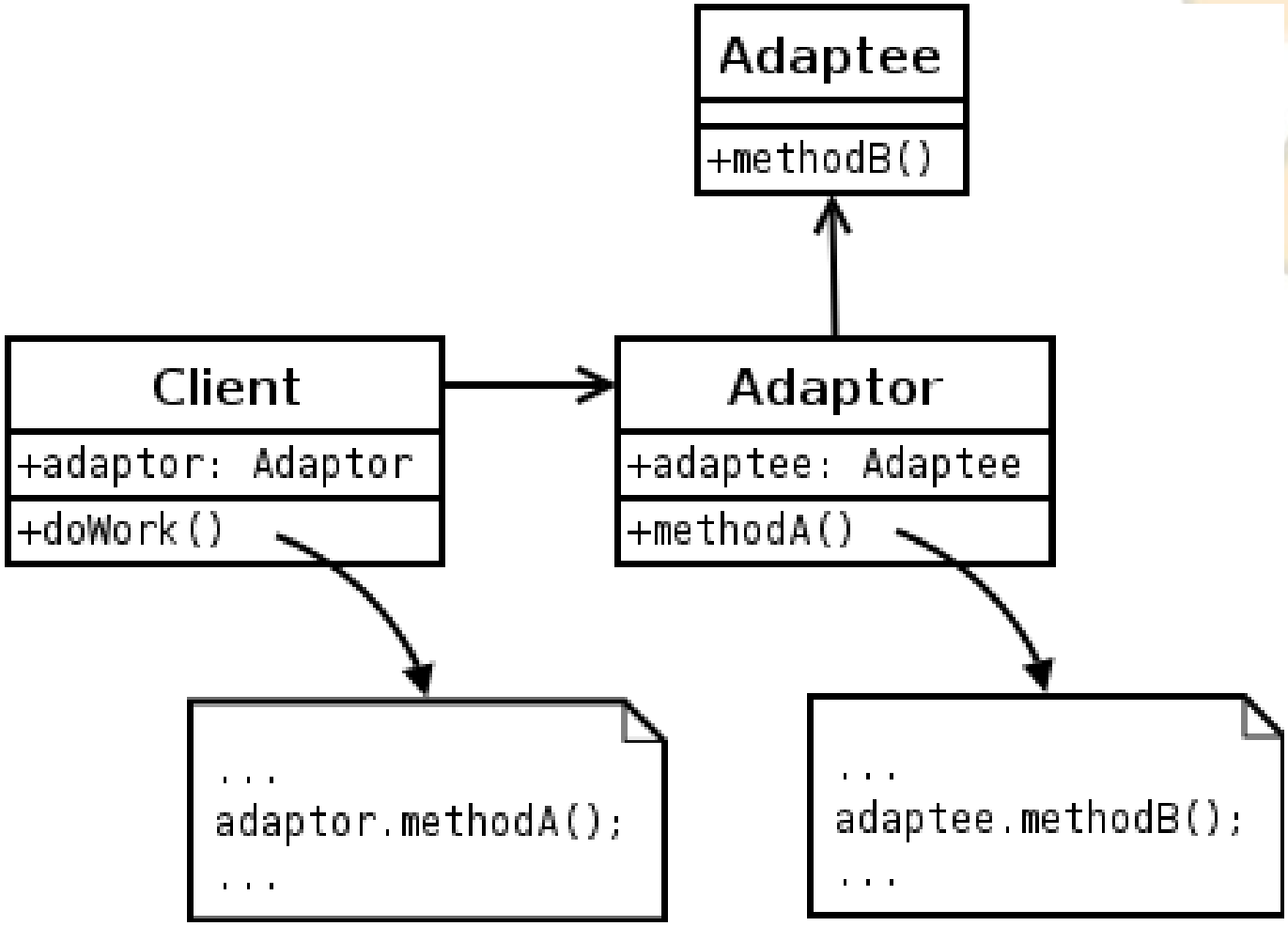


Estrutura

- *Object Adapter*: usa composição.
- Única solução se *Target* não for uma interface em Java.
- *Adapter* possui referência para o objeto que terá sua interface adaptada (instância de *Adaptee*).
- Cada método de *Target* chama o(s) método(s) correspondente(s) na interface adaptada.

ybadoo

Object Adapter



Colaborações

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

ybadoo

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 o cliente não usa uma interface Java ou classe abstrata não pode ser estendida.

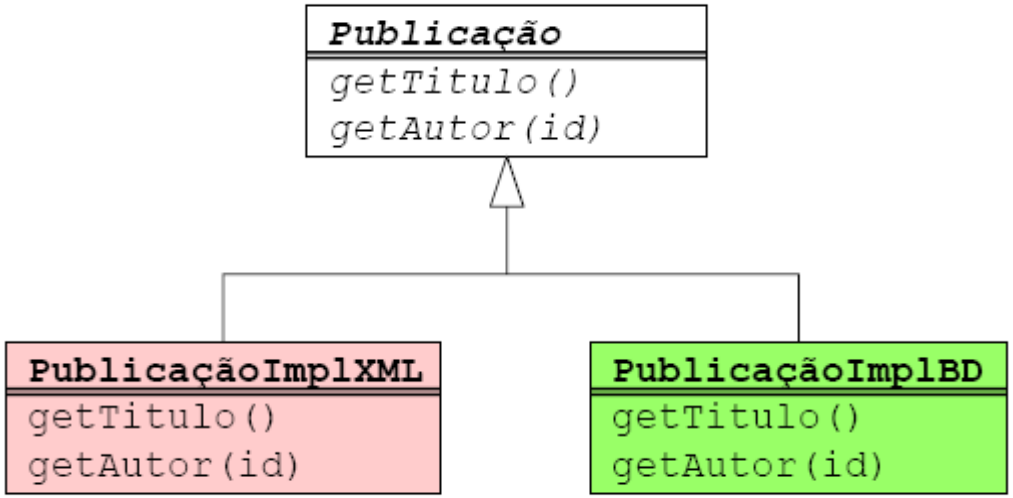
ybadoo

Bridge

“Desacoplar uma abstração de sua implementação para que os dois possam variar independentemente.”

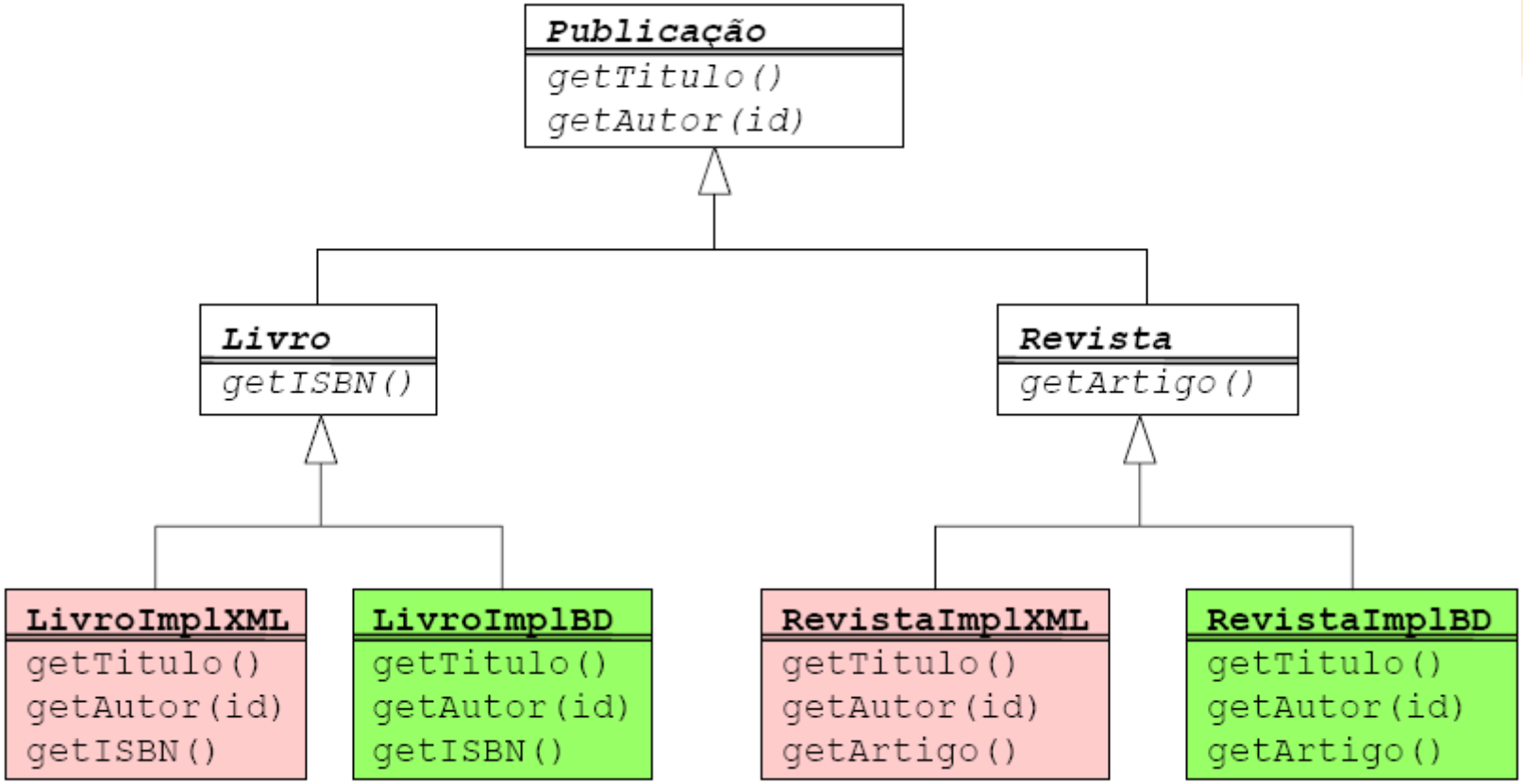
Problema (1/2)

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



Problema (2/2)

- Mas herança complica a implementação:



Estrutura (1/2)

- *Abstraction*:
  - Define a interface da abstração.
  - Mantém uma referência para um objeto do tipo *Implementor*.
- *RefinedAbstraction*:
  - Estende a interface definida por *Abstraction*.
- *Implementor*:
  - Define a interface para as classes de implementação. Esta interface não precisa corresponder exatamente à interface de *Abstraction*; de fato, as duas interfaces podem ser bem diferentes. A interface de *Implementor* fornece somente operações primitivas e *Abstraction* define operações de nível mais alto baseadas nestas primitivas.

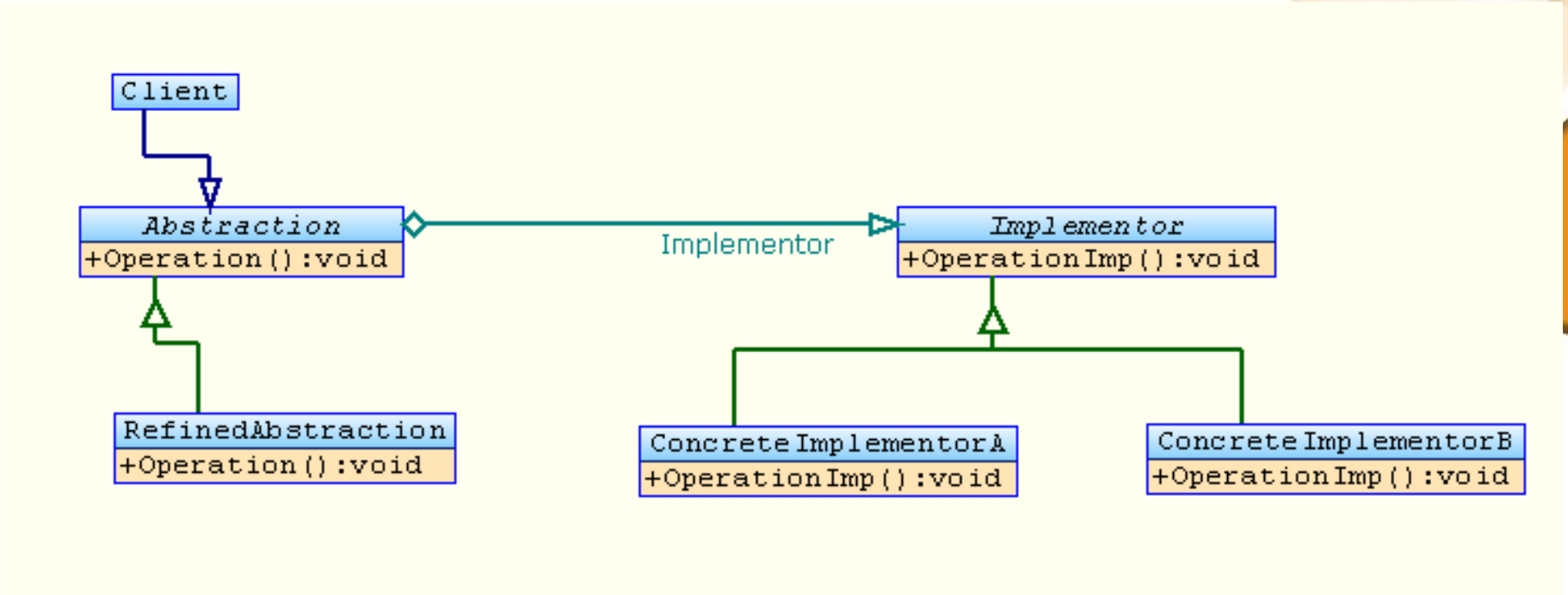
ybadoo

Estrutura (2/2)

- *ConcreteImplementador*:
  - Implementa a interface de *Implementor* e define sua implementação concreta.

ybadoo

Diagrama UML



Quando usar?

- Quando for necessário evitar uma ligação permanente entre a interface e a 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.

ybadoo

Diferenças

- O *Adapter* faz as coisas trabalharem depois de projetadas, enquanto a *Bridge* os faz trabalhar antes.
- A *Bridge* possibilita bastante independência na abstração e na implementação.
- O *Adapter* é feito para que classes com interfaces incompatíveis possam trabalhar juntas.

ybadoo

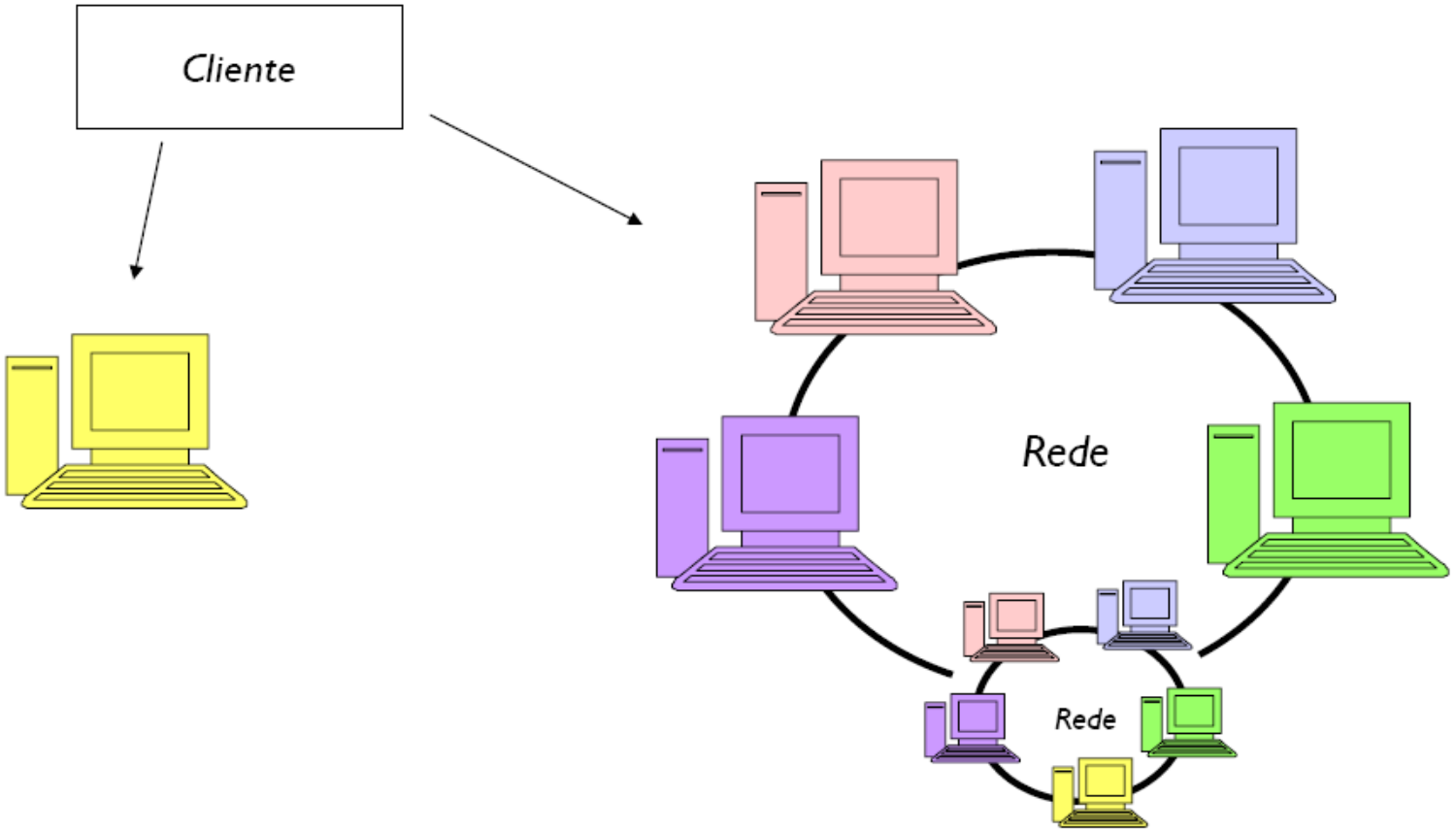
Composite

“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.”

ybadoo

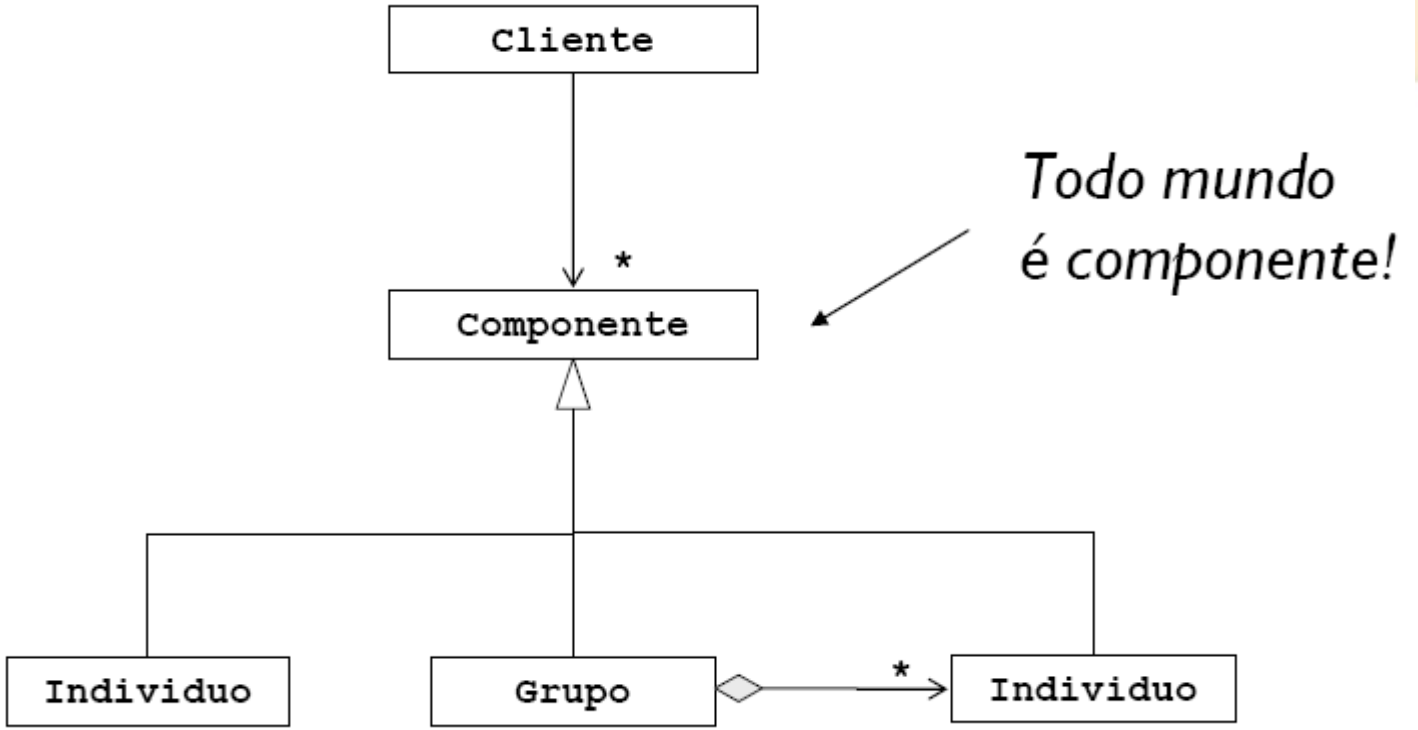
Problema

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



Solução

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



ybadoo

Estrutura (1/2)

- *Component*:
  - Declara a interface para os objetos na composição.
  - Implementa comportamento por falta para a interface comum a todas as classes, conforme apropriado.
  - Declara uma interface para acessar e gerenciar os seus componentes-filho.
- *Leaf*:
  - Representa objetos-folha na composição.
  - Define comportamento para objetos primitivos na composição.

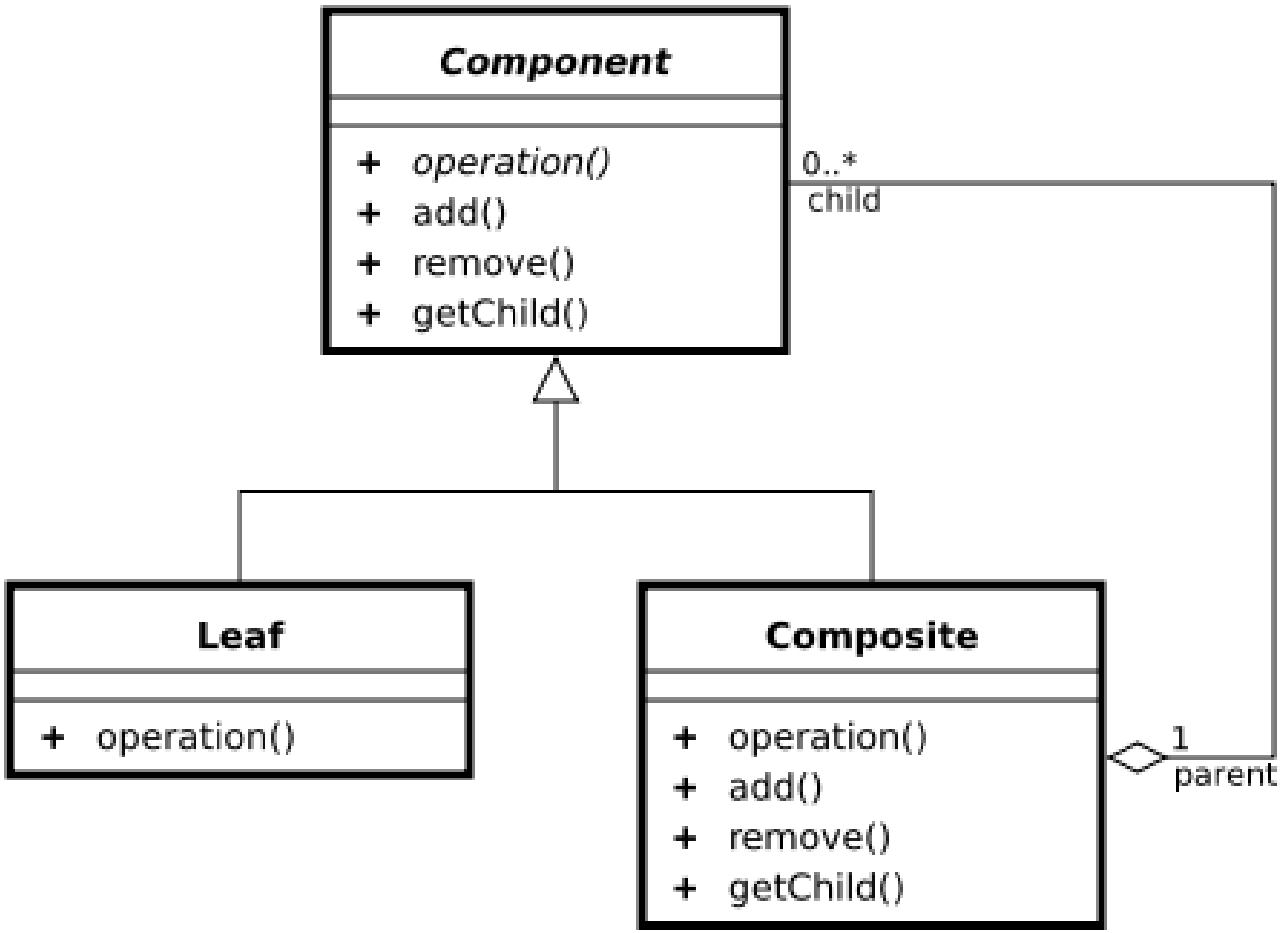
ybadoo

Estrutura (2/2)

- *Composite*:
  - Define comportamento para componentes que têm filhos.
  - Armazena os componentes-filho.
  - Implementa as operações relacionadas com os filhos presentes na interface de *Component*.
- *Client*:
  - Manipula objetos na composição através da interface de *Component*.

ybadoo

Diagrama UML



Quando usar?

- Sempre que houver necessidade de tratar um conjunto como um indivíduo.
- Funciona melhor se relacionamentos entre 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.

ybadoo

Consequências

- *Composite* define hierarquias de objetos que podem se tornar crescentemente complexas.
- Torna o cliente simples, pois ele não precisa conhecer a classe exata dos objetos.
- Torna fácil criar novas classes de componentes.
- Mas projeto fica excessivamente genérico:
  - Dificuldade em restringir os membros de objetos compostos.

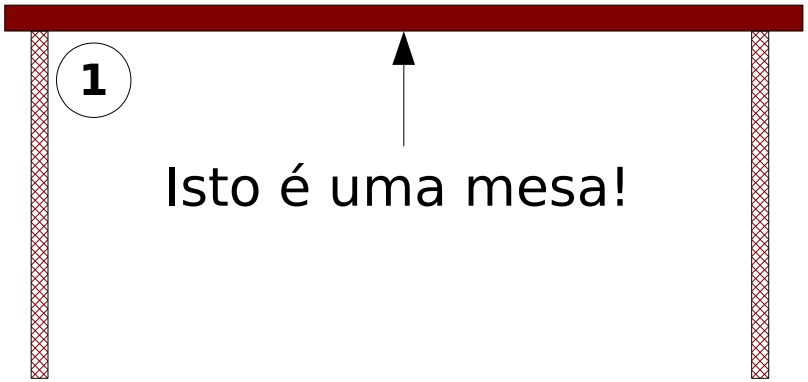
ybadoo

Decorator

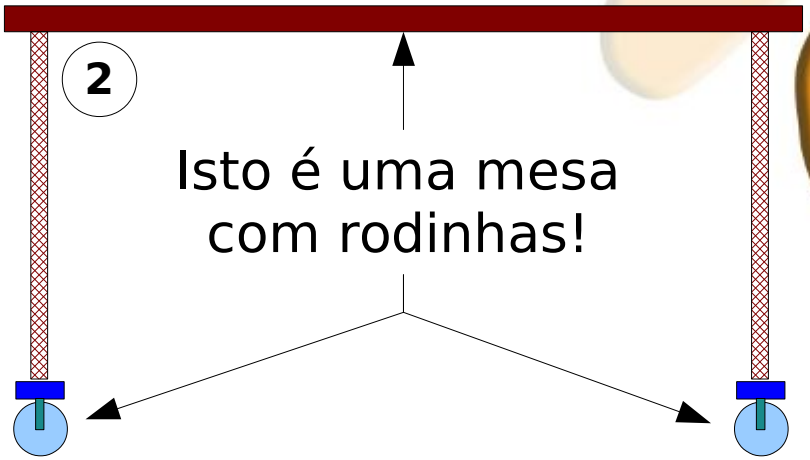
“Anexar responsabilidades adicionais a um objeto dinamicamente. Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade.”

ybadoo

Problema



Queremos modelar estas mesas em termos de objetos



A aplicação pode, em algum momento, usar a mesa pura, em outro, com rodinhas, em outro, com rede de *ping pong,* etc.

ybadoo

Usar herança?

- Solução mais “natural”.
- Adição estática de responsabilidades.
- Cliente não tem como controlar como e quando colocar rodinhas, a rede de *ping pong* ou outras coisas:
  - Responsabilidade é atribuída a toda a classe e não a cada objeto.
- Não é facilmente extensível.
- E se quisermos usar a mesa para jantar?

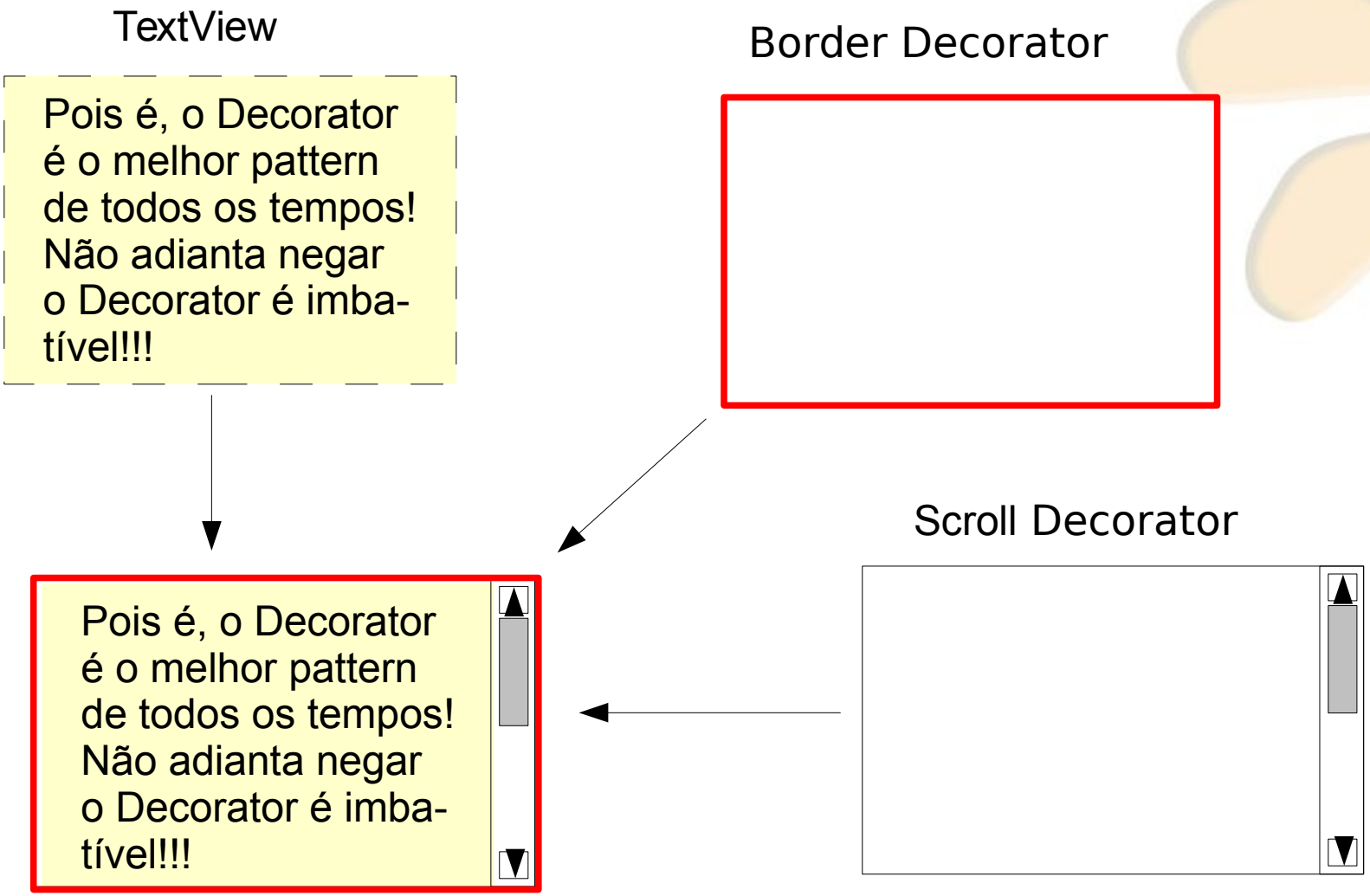
ybadoo

Porque não decorar?

- Encobrir o objeto mesa em outro objeto que adiciona rodinhas, rede de *ping pong*, etc.
- O objeto que “engloba” a mesa se chama *decorator*.
- Solução mais flexível pois podemos aproveitar os *decorators* para outros móveis e combiná-los de inúmeras formas.

ybadoo

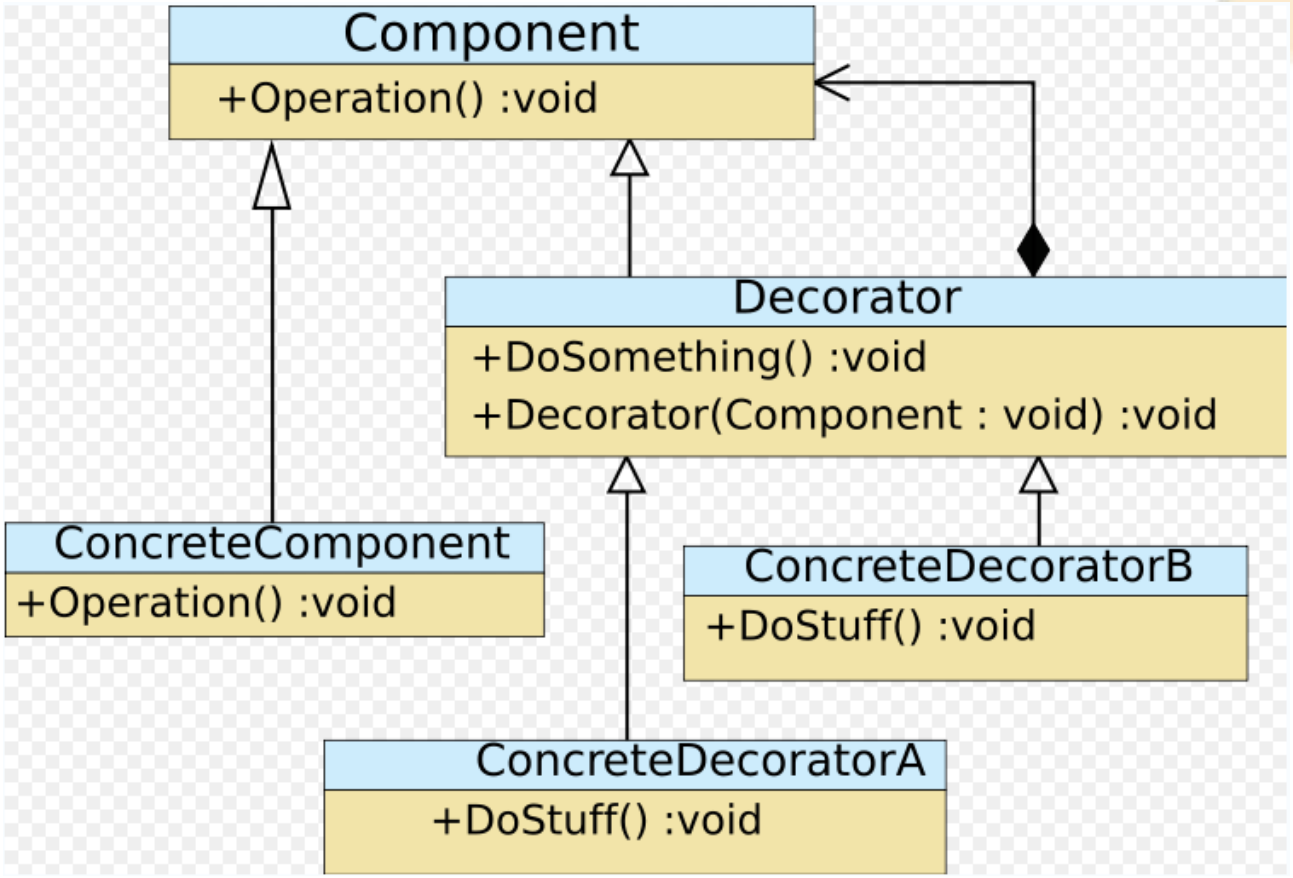
Exemplificando



Estrutura

- *Component*:
  - Define a interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente.
- *ConcreteComponent*:
  - Define um objeto para o qual responsabilidades adicionais podem ser atribuídas.
- *Decorator*:
  - Mantém uma referência para um objeto *Component* e define uma interface que segue a interface de *Component*.
- *ConcreteDecorator*:
  - Acrescenta responsabilidades ao componente.

Diagrama UML



Consequências (1/2)

- O *decorator* implementa a mesma interface dos componentes que ele decora:
  - Transparência para os clientes.
- *Decorators* podem ser aninhados recursivamente permitindo um número ilimitado de funcionalidades.
- O *decorator* adiciona funcionalidades de forma transparente.

ybadoo

Consequências (2/2)

- Maior flexibilidade que herança.
- Adição de responsabilidades por demanda.
- Transparência para o cliente:
  - Tomar cuidado com o estado do objeto.
- Aumenta o número de objetos nas aplicações:
  - Relação entre extensibilidade e manutenibilidade.

ybadoo

Aplicabilidade

- Quando se quer adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos.
- Quando a utilização de herança para estender funcionalidades gera uma explosão de subclasses para atender às diversas combinações.

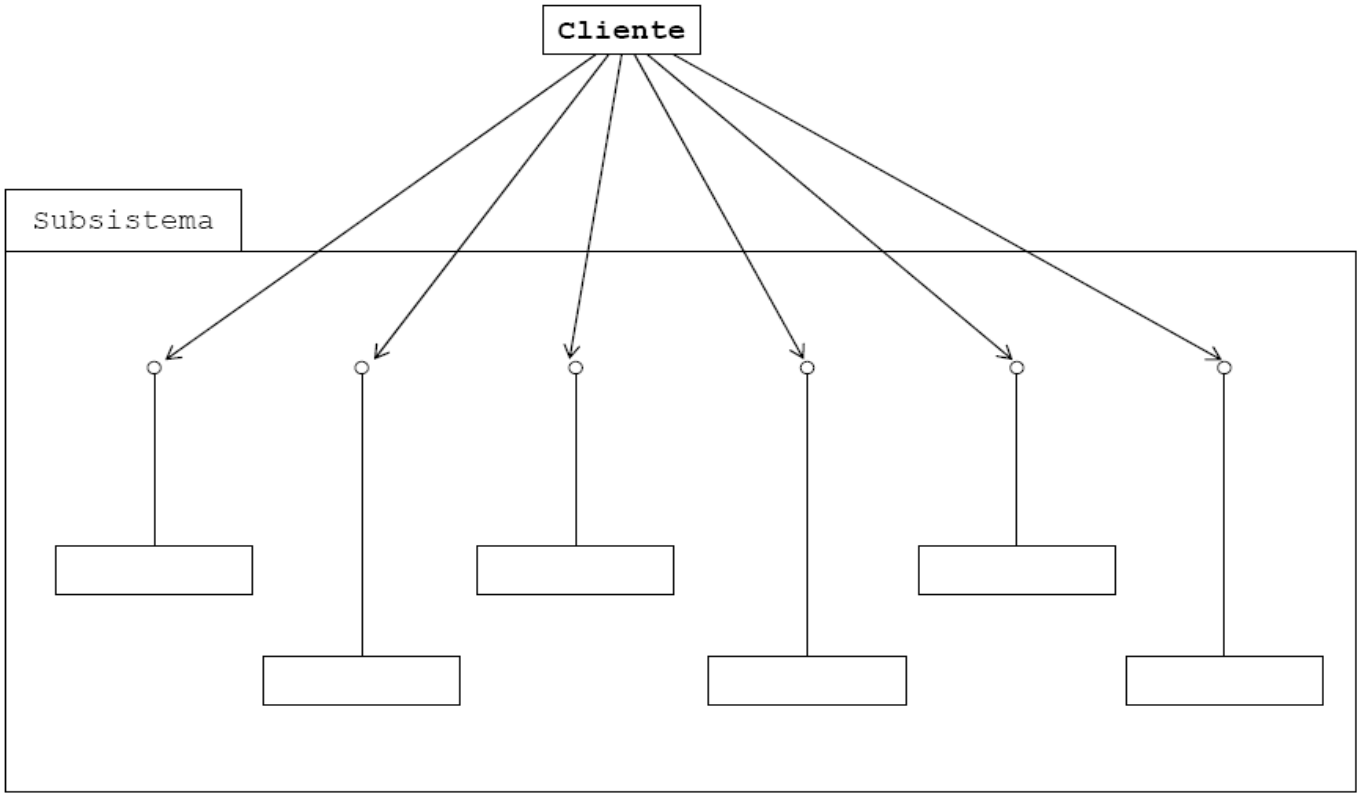
Façade

“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.”

ybadoo

Problema

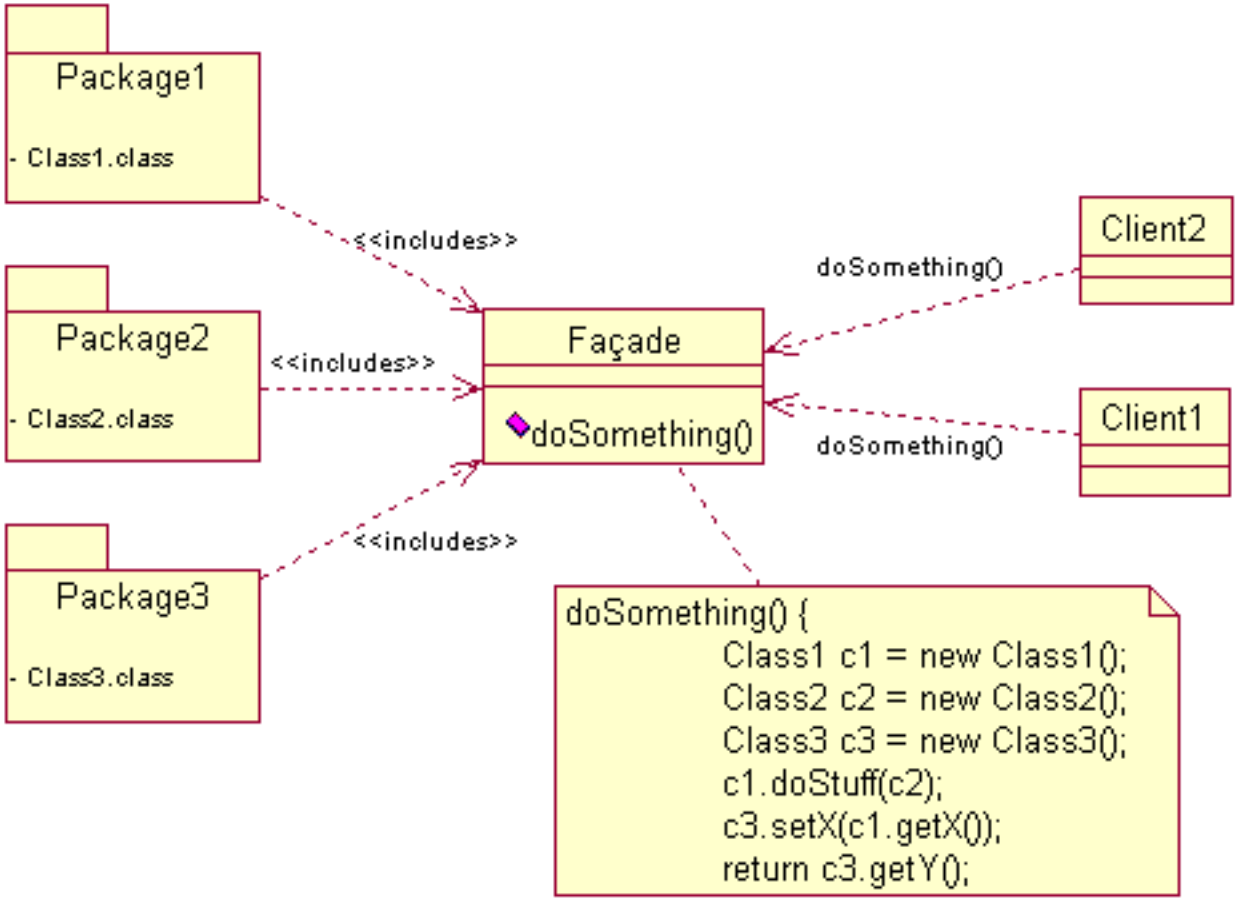
- Cliente precisa saber muitos detalhes do subsistema para utilizá-lo.



Estrutura

- *Façade*:
  - Conhece quais as classes do subsistema responsáveis pelo atendimento de uma solicitação.
  - Delega solicitações de clientes a objetos apropriados do subsistema.
- Classes de subsistema:
  - Implementam a funcionalidade do subsistema.
  - Encarregam-se do trabalho atribuído a elas pelo objeto *Façade*.
  - Não têm conhecimento da *Façade*, ou seja, elas não mantêm referências para a mesma.

Diagrama UML



Aplicabilidade

- Este padrão é usado quando:
  - Deseja-se prover uma interface simples para um subsistema complexo.
  - Existem muitas dependências entre clientes e as classes de implementação de uma abstração.
  - Deseja-se separar o subsistema em camadas.

ybadoo

Consequências

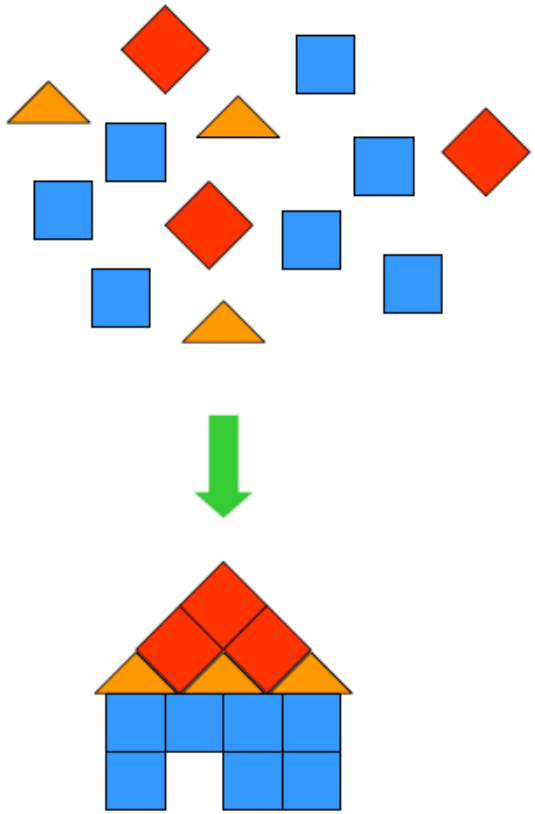
- Esconde os componentes do subsistema dos clientes:
  - Ao reduzir o número de objetos com os quais os clientes lidam, fazendo o subsistema mais fácil de se usar.
- Reduz o acoplamento entre o subsistema e seus clientes:
  - Pode eliminar dependências complexas ou circulares. Isto pode ser importante quando o cliente e o subsistema são implementados independentemente.
- Não impede que as aplicações usem as classes do subsistema, se precisarem:
  - Assim, pode-se escolher entre facilidade do uso e generalidade.

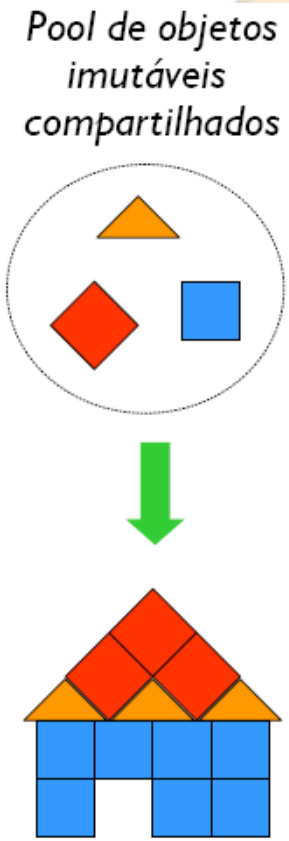
Flyweight

"Usar compartilhamento para suportar grandes quantidades de objetos refinados eficientemente."

ybadoo

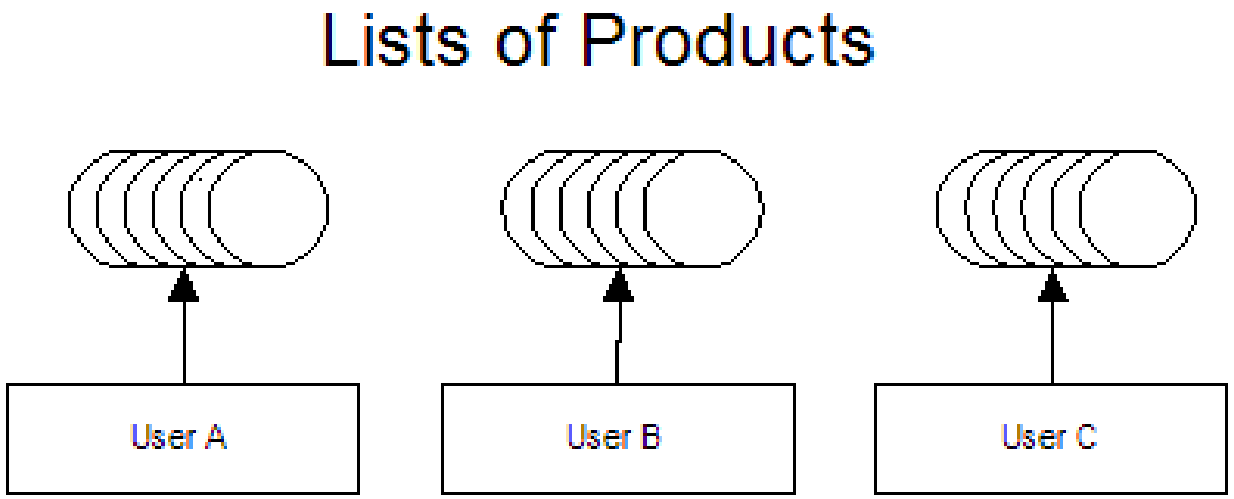
Problema



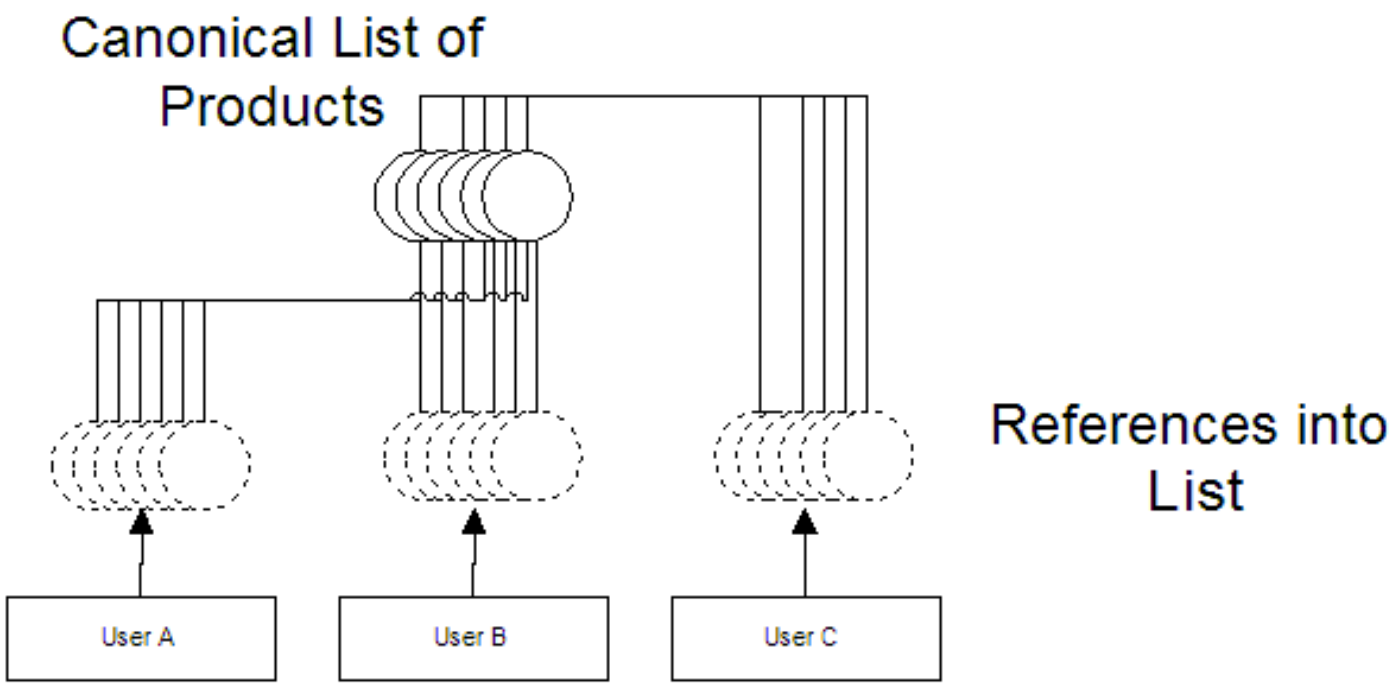


ybadoo

Exemplo - Problema



Exemplo - Solução



Estrutura (1/3)

- *Flyweight*:
  - Declara uma interface através da qual *flyweights* podem receber e atuar sobre estados extrínsecos.
- *ConcreteFlyweight*:
  - Implementa a interface de *Flyweight* e acrescenta armazenamento para estados intrínsecos, se houver.
  - Um objeto *ConcreteFlyweight* deve ser compartilhável.
  - Qualquer estado que ele armazene deve ser intrínseco, ou seja, deve ser independente do contexto do objeto *ConcreteFlyweight*.

Estrutura (2/3)

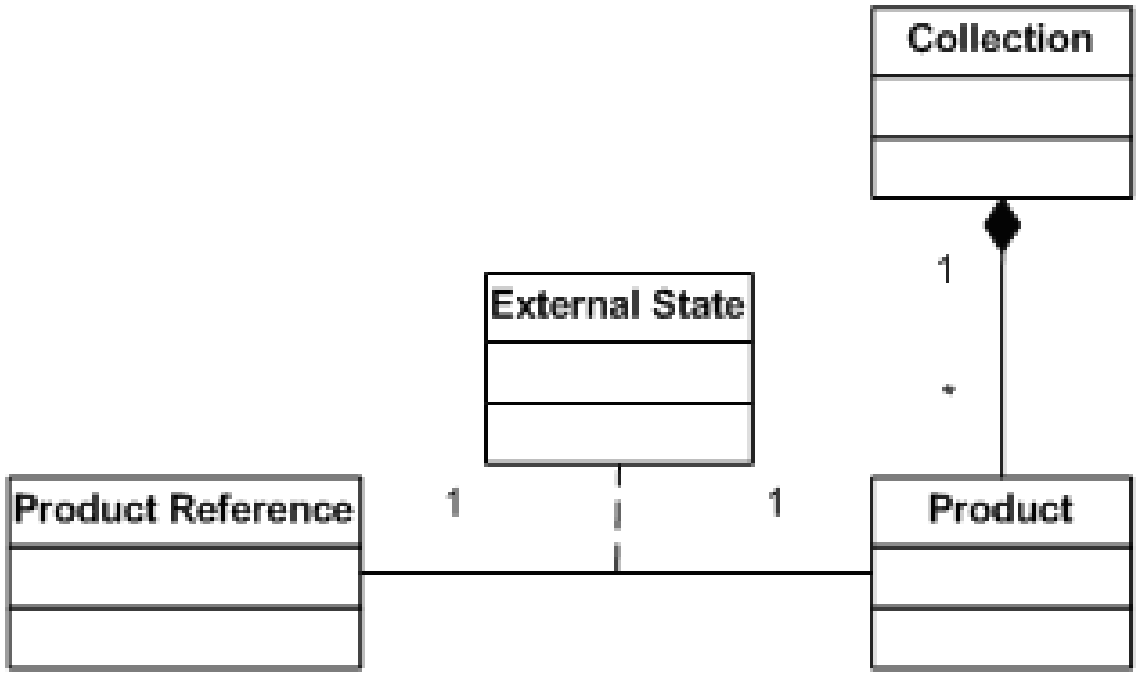
- *UnsharedConcreteFlyweight*:
  - Nem todas as subclasses de *Flyweight* necessitam ser compartilhadas.
  - A interface de *Flyweight* habilita o compartilhamento, ela não força ou garante o mesmo.
  - É comum para objetos *UnsharedConcreteFlyweight* não compartilhar objetos *ConcreteFlyweight* como filhos em algum nível da estrutura de objetos *Flyweight*.
- *FlyweightFactory*:
  - Cria e gerencia objetos *Flyweight*.
  - Garante que os *flyweight* sejam compartilhados apropriadamente.
  - Quando um cliente solicita um *flyweight*, um objeto *FlyweightFactory* fornece uma instância existente ou cria uma, se nenhuma existir.

ybadoo

Estrutura (3/3)

- *Client*:
  - Mantém uma referência para *flyweight*(s).
  - Computa ou armazena o estado extrínseco do *flyweight*(s).

Diagrama UML



Prós e contras

- *Flyweight* é uma solução para construção de aplicações usando objetos imutáveis:
  - Ideal para objetos que oferecem serviços (guardados em caches e em *pools*).
  - Ideal para objetos que podem ser usados para construir outros objetos.
- Problemas:
  - Possível impacto no desempenho (se houver muitas representações diferentes, elas não podem ser alteradas, e é preciso criar muitos objetos).

ybadoo

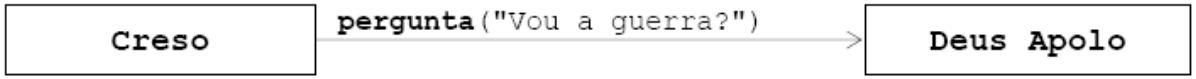
Proxy

“Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro.”

ybadoo

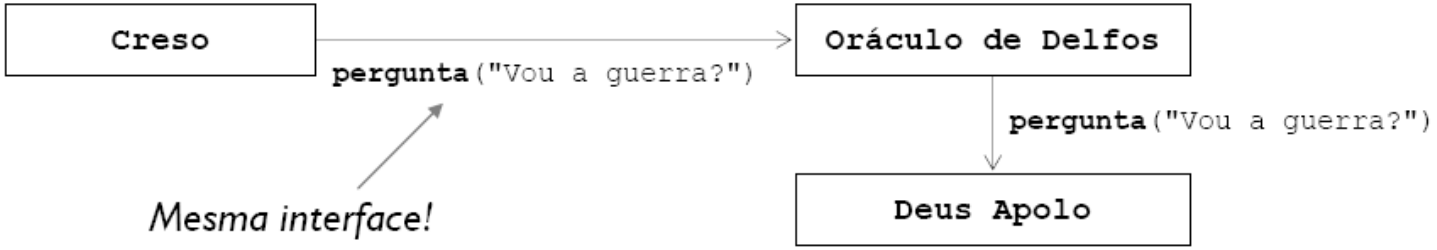
Problema

- *Sistema quer utilizar objeto real...*



- *Mas ele não está disponível (remoto, inaccessível, ...)*

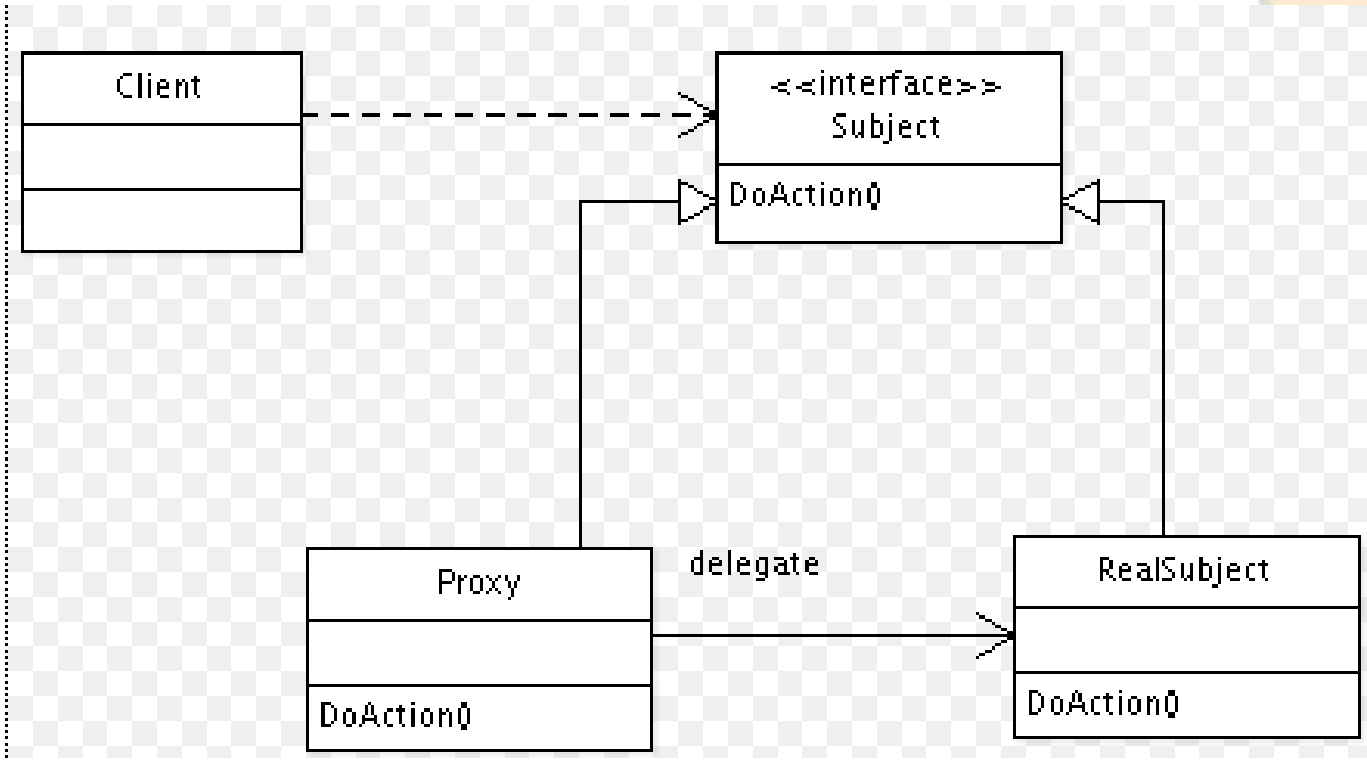
- *Solução: arranjar um **intermediário** que saiba se comunicar com ele eficientemente*



Estrutura

- 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.

Diagrama UML



ybadoo

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.

Resumo (1/2)

- *Adapter*:
  - Adaptar uma interface existente para um cliente.
- *Bridge*:
  - Implementar um design que permita total desacoplamento entre interface e implementação.
- *Composite*:
  - Tratar composições e unidade uniformemente.
- *Decorator*:
  - Para acrescentar recursos e comportamento a um objeto existente, receber sua entrada e poder manipular sua saída.

ybadoo

Resumo (1/2)

- *Façade*:
  - Simplificar o uso de uma coleção de objetos.
- *Flyweight*:
  - Quando for necessário reutilizar objetos visando desempenho.
- *Proxy*:
  - Quando for preciso um intermediário para o objeto real.

ybadoo