Programação Orientada a Objetos II

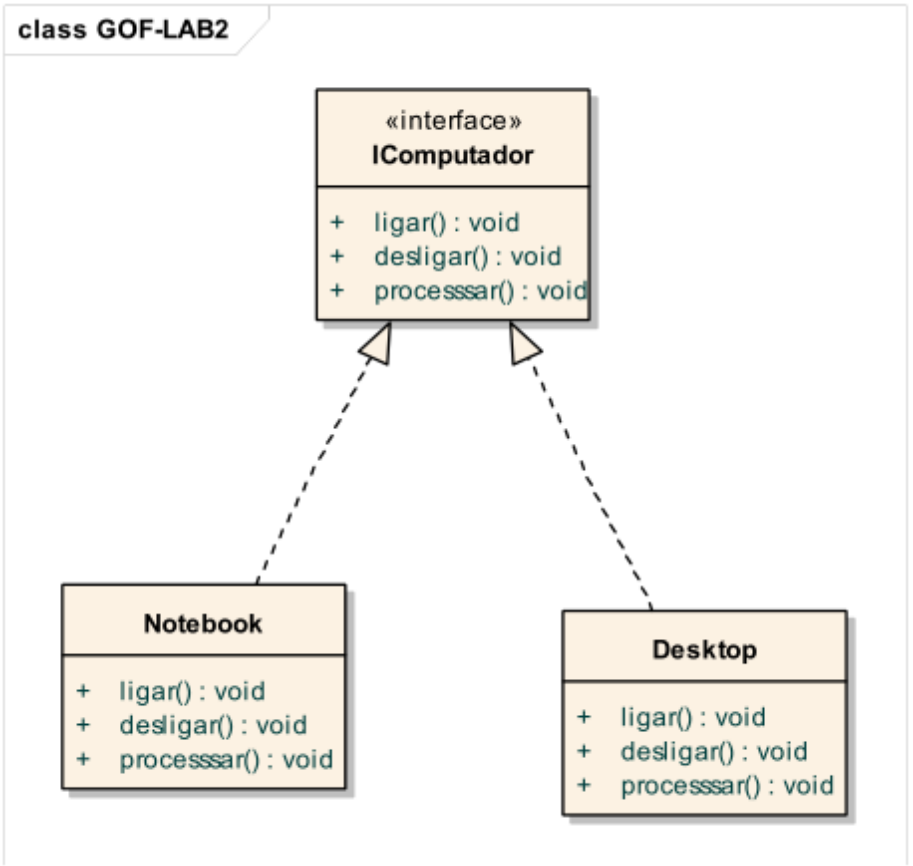
Aula 07

Prof. Leandro Nogueira Couto
UFU – Monte Carmelo
05/2013

Exercício

- Vamos fazer um sistema de compra de computadores (vide website da Dell)
- Considere que é possível montar **Desktops**, **Notebooks** e **Ultrabooks**
- Para esse sistema, desejamos poder modificar as máquinas padrão escolhendo diferentes partes melhoradas de um computador (GPU, CPU, Memória, HD), **com diferentes preços e diferentes descrições**, para montar uma especificação de um PC desejada pelo usuário
- Considere e implemente a **restrição**: o produto montado só pode possuir um de cada item (e.g. apenas uma GPU)
- Use o Padrão de Projeto **Decorator**
- O sistema deve permitir obter o custo de uma máquina decorada e obter a descrição da máquina

Exercício



- Ignore os métodos da interface; considere que o método que teremos são **custo()**.
- Queremos poder construir um **Notebook** ou **Desktop** com uma variedade de diferentes **Componentes**.
- No final da montagem (no método **main**) queremos saber o preço do computador montado.

Recapitulando

- Vimos até aqui alguns **Padrões de Projeto Estruturais**:
  - Adapter
  - Bridge
  - Decorator
  - Facade
- Design Patterns relacionados com a organização e relações entre os objetos
- Preservar princípios de POO
- **Discussão**: projetar com Design Patterns ou usar para corrigir problemas?

Padrões Estruturais

- Veremos agora mais alguns Padrões de Projeto Estruturais:
  - Composite
  - Flyweight
  - Proxy

Composite

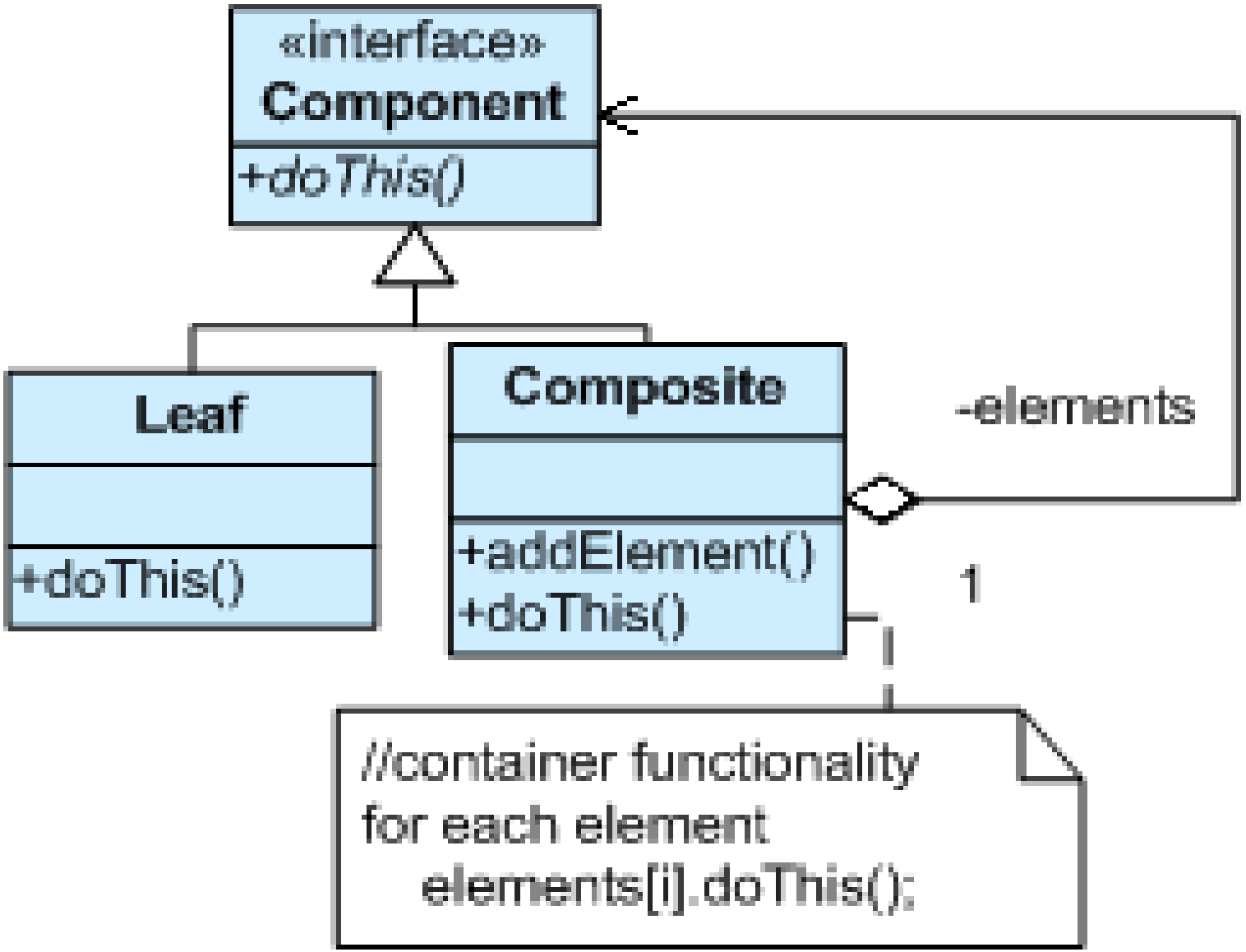
Composite

- Imagine que:
- Queremos manipular de forma uniforme uma coletânea hierárquica de objetos **primitivos** e **compostos**.
- "Perguntar" (query) o tipo do objeto antes de processar não é desejável.
- O processamento de cada tipo de objeto é diferente.

Composite

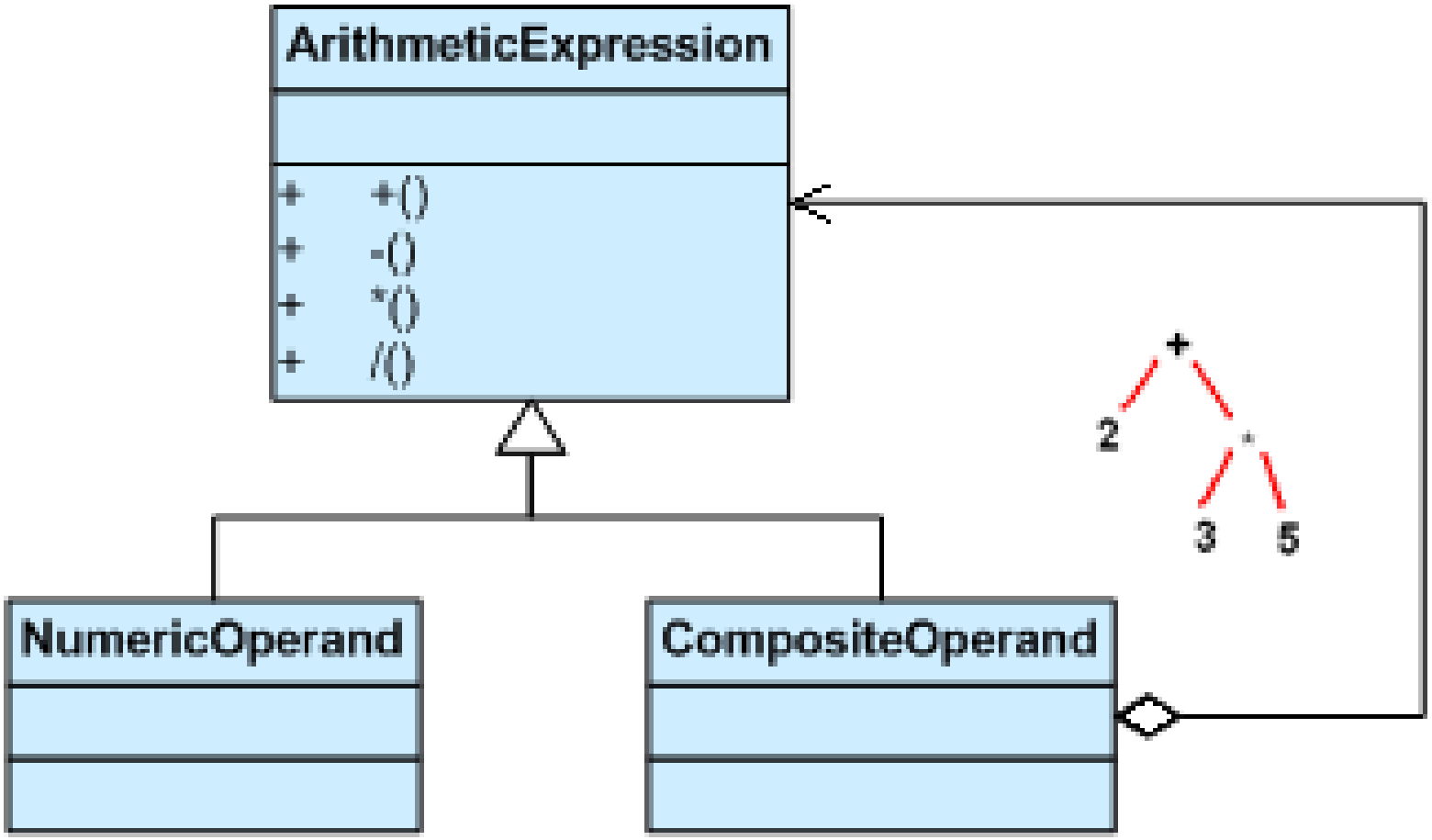
- **Usar quando**: quando se tem o**bjetos compostos** que tem **componentes**, cada um dos quais podendo ser também um **objeto composto**
- Um **diretório** contém elementos que podem ser **arquivos** os outros **diretórios**
- Estrutura estilo **árvore**
- Como queremos métodos para manipular os "filhos" [e.g. addFilho(), removeFilho()], devemos definir esses métodos na classe Composta. Mas como queremos tratar Primitivos e Compostos de forma uniforme, precisamos mover esses métodos para uma **classe Component abstrata**.

Composite


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
+addElement()
+doThis()
-elements
1
//container functionality
for each element
elements[i].doThis();

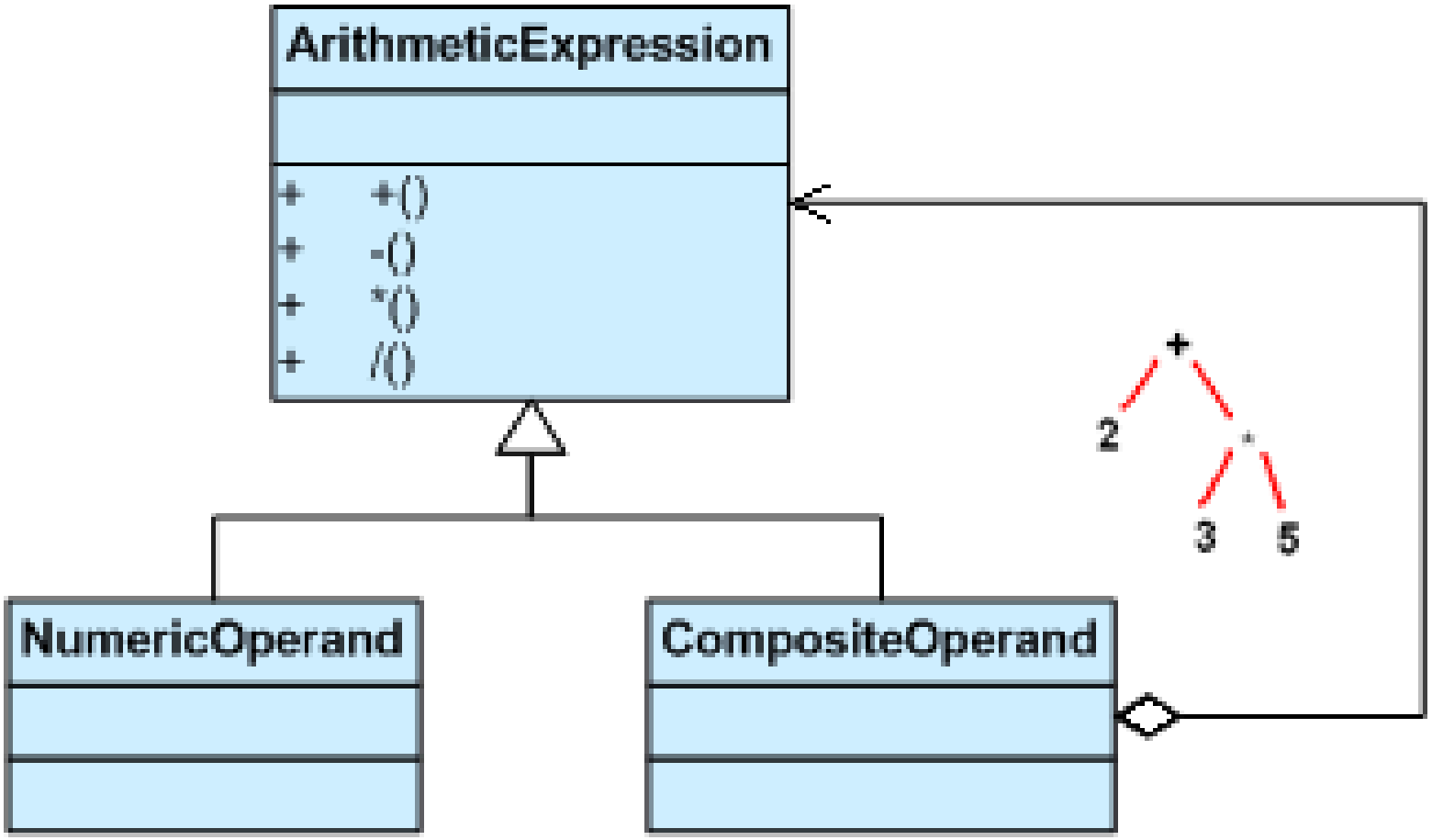
Composite

- Exemplo: menus que contém itens, cada um dos quais pode ser um (sub)menu.
- Um exemplo: expressões aritméticas:



Composite

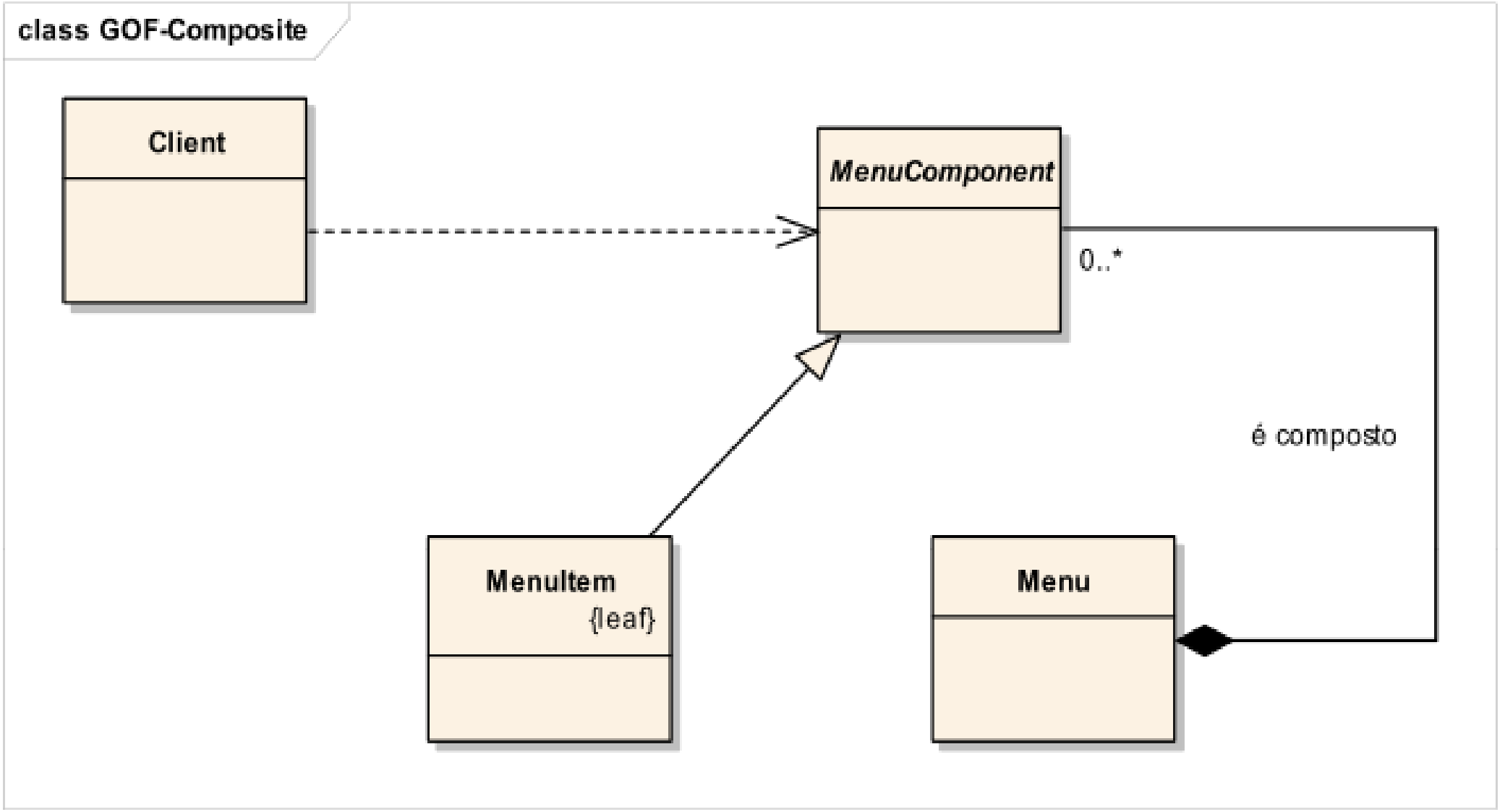
- Um exemplo: expressões aritméticas. Como fica o **cálculo da expressão final** nesse caso? Que métodos são necessários? calcularResultado()



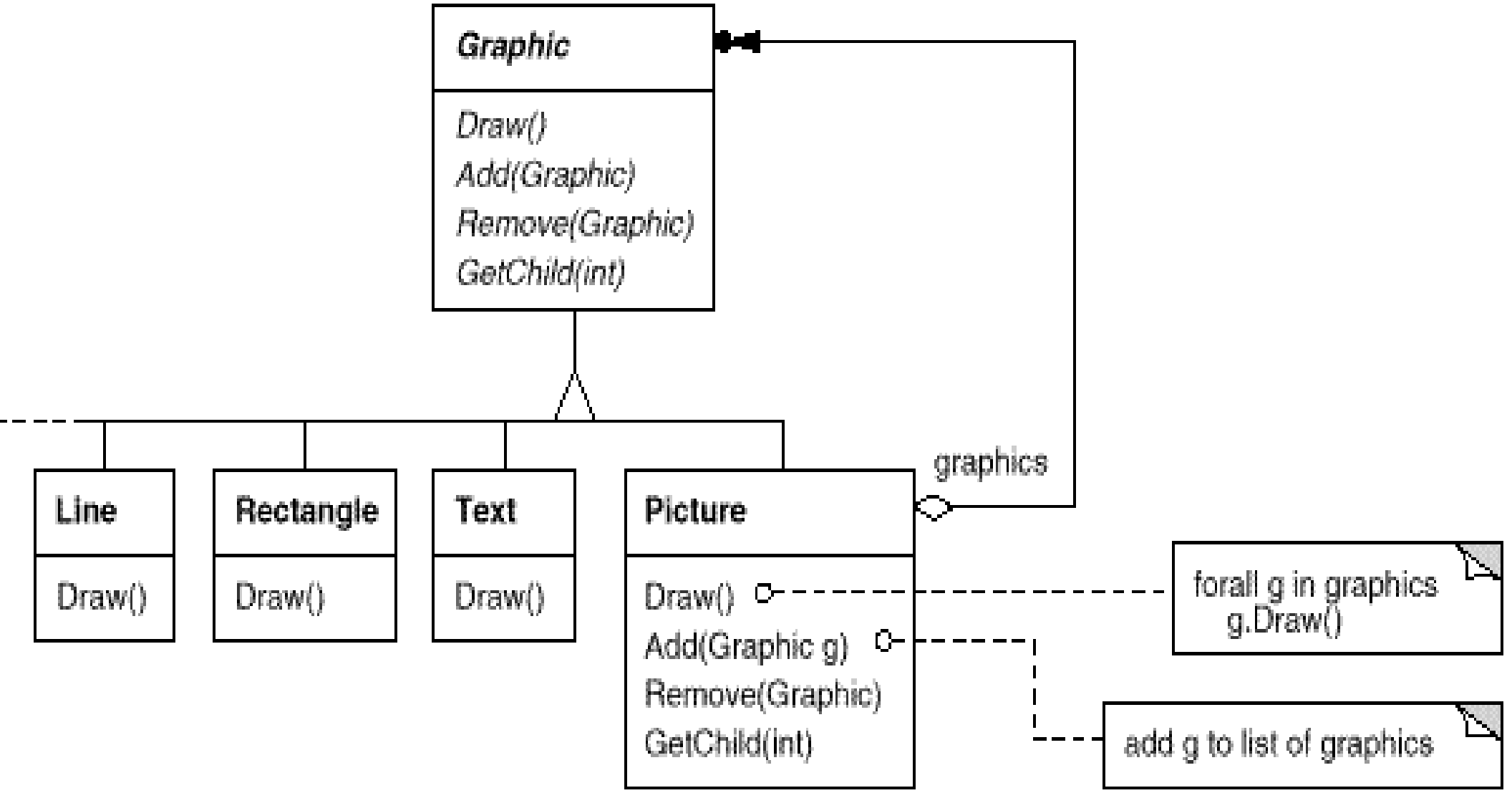
Composite

- Participantes:
- Component – Abstração para folhas e Composite. Por exemplo, um sistema de arquivos define move, copy, rename, e getSize para arquivos e pastas.
- Leaf – Folhas não tem filhos. Implementam serviços definidos na interface Component.
- Composite - Um Composite armazena componentes filho e implementa os métodos definidos na inteface Component. Usualmente implementa métodos para Adicionar(), Remover() ou Get() dos seus Componentes.
- Client – Manipula os objetos usando a interface Component.

Composite


class GOF-Composite
Client
MenuComponent
0..*
é composto
MenuItem
{leaf}
Menu

Composite


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

Composite

- Passos:
- Assegure que seu problema é sobre como representar relações hierárquicas "parte-todo".
- Divida seu domínio em conceitos que sejam classes containers e classes contidas.
- Crie uma interface “mínimo denominador comum”, ou seja que tem métodos aplicáveis a containers e contidos.
- As classes container e contido herdam da interface.
- Todo container tem uma relação "tem um" um-para-muitos com a interface (composição).
- Classes container tratam o polimorfismo para delegar pros seus objetos contidos.
- Decida onde vão seus métodos de gerenciamento de filhos [e.g. addChild(), removeChild()]. Ver discussão Transparência vs Segurança logo adiante.

Composite

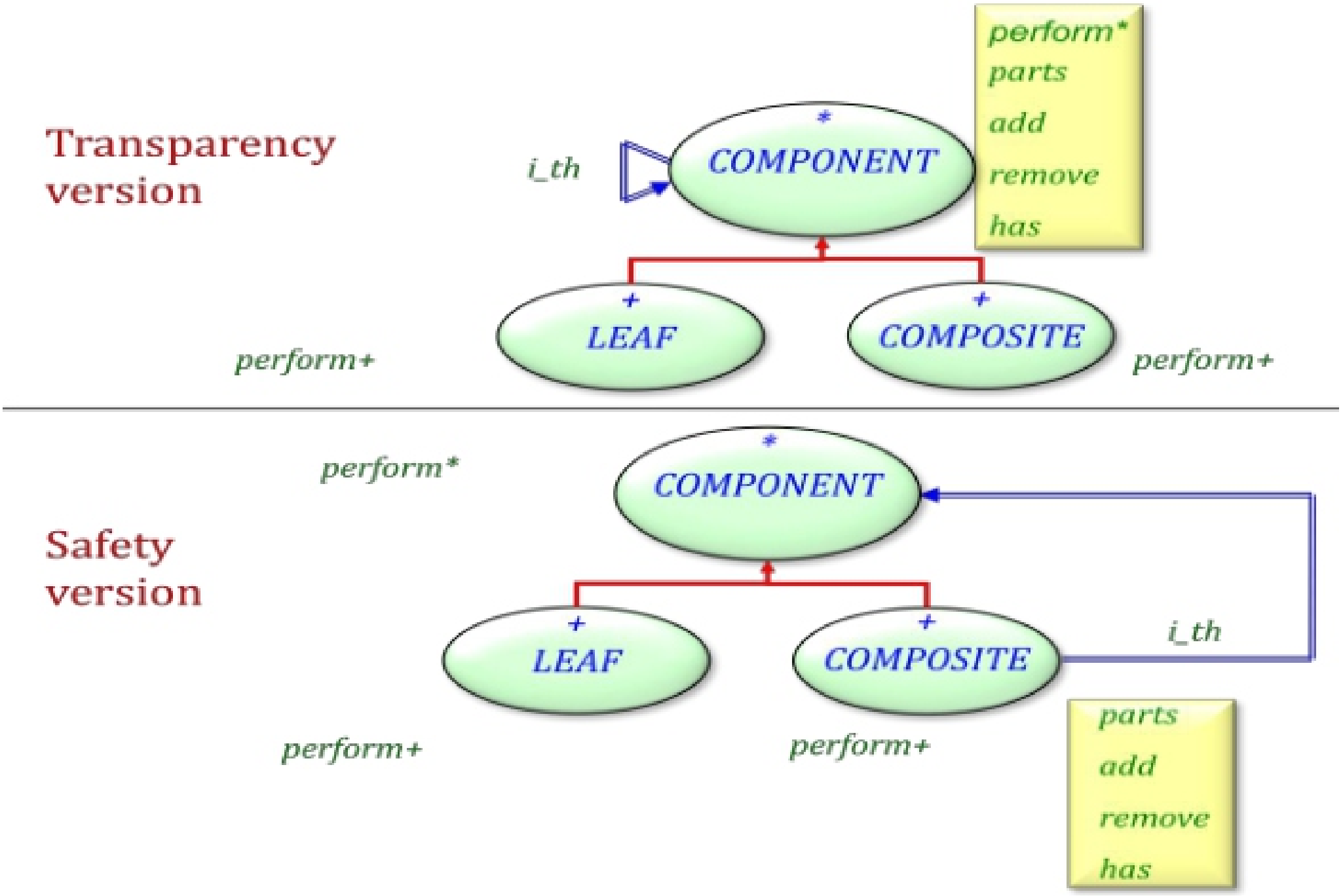
- Composite e Decorator tem diagramas de estrutura similares, refletindo o fato de que ambos dependem de composição recursiva para organizar um número de objetos abritrário (ou "tão grande quanto se queira")

- Pode ser útil construir um Composite usando o padrão creacional Builder
  - Construção complexa do Composite
  - Embalado num Builder que previne erros na construção e separa a instanciação. Podemos com um Builder só ter vários tipos de Composites.

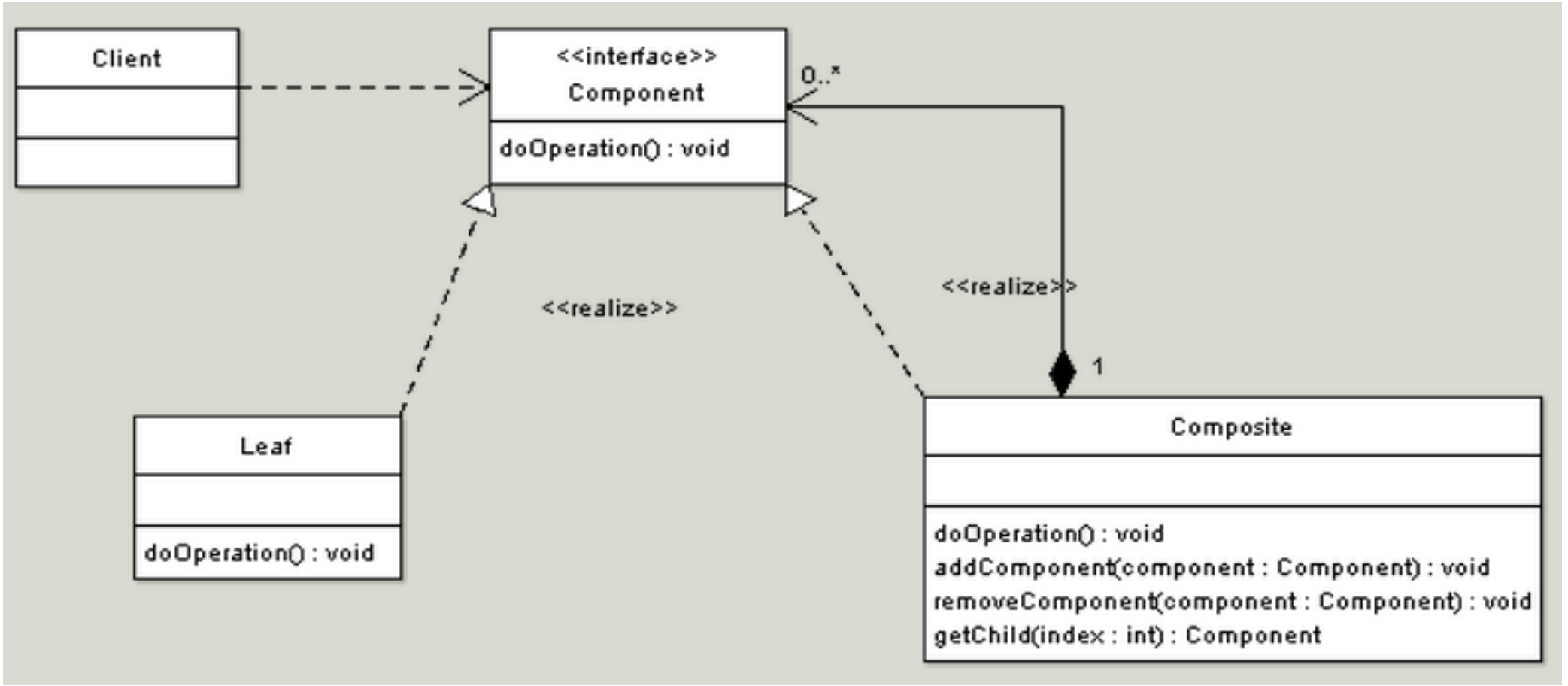
Composite

- **Transparência vs Segurança**
- Poder tratar uma coleção heterogênea de objetos com **transparência** requer que a interface “gerenciamento de filhos” seja definida na raíz da hierarquia de classe Composta (a classe abstrata Component).
- Porém, isso custa em **segurança**, pois Clientes podem tentar fazer coisas sem sentido como adicionar ou remover objetos de objetos folha. Pra combater isso, a interface de gerenciamento de filhos é declarada na classe Composite, mas assim perde-se transparência porque folhas e Composites agora tem interfaces diferentes.

Composite


Transparency version
perform*
parts
add
remove
has
*
COMPONENT
i_th
+
LEAF
+
COMPOSITE
perform+
perform+
Safety version
perform*
*
COMPONENT
+
LEAF
+
COMPOSITE
i_th
perform+
perform+
parts
add
remove
has

Composite


Client
<<interface>>
Component
doOperation() : void
0..*
<<realize>>
<<realize>>
1
Leaf
doOperation() : void
Composite
doOperation() : void
addComponent(component : Component) : void
removeComponent(component : Component) : void
getChild(index : int) : Component

Composite

- É interessante ter habilidade do cliente conseguir realizar operações no objeto sem precisar saber se o objeto é uma folha ou não.
- O Composite não te força a tratar todos os Components como Composites. Ele só diz pra você colocar todas as operações que se quer tratar uniformemente na classe Component.
- Exemplo:

  Copiar um arquivo para um diretório faz sentido, mas copiar um arquivo para outro arquivo não faz sentido

  Por outro lado, **renomear** faz sentido para um arquivo ou para um diretório

Flyweight

Flyweight

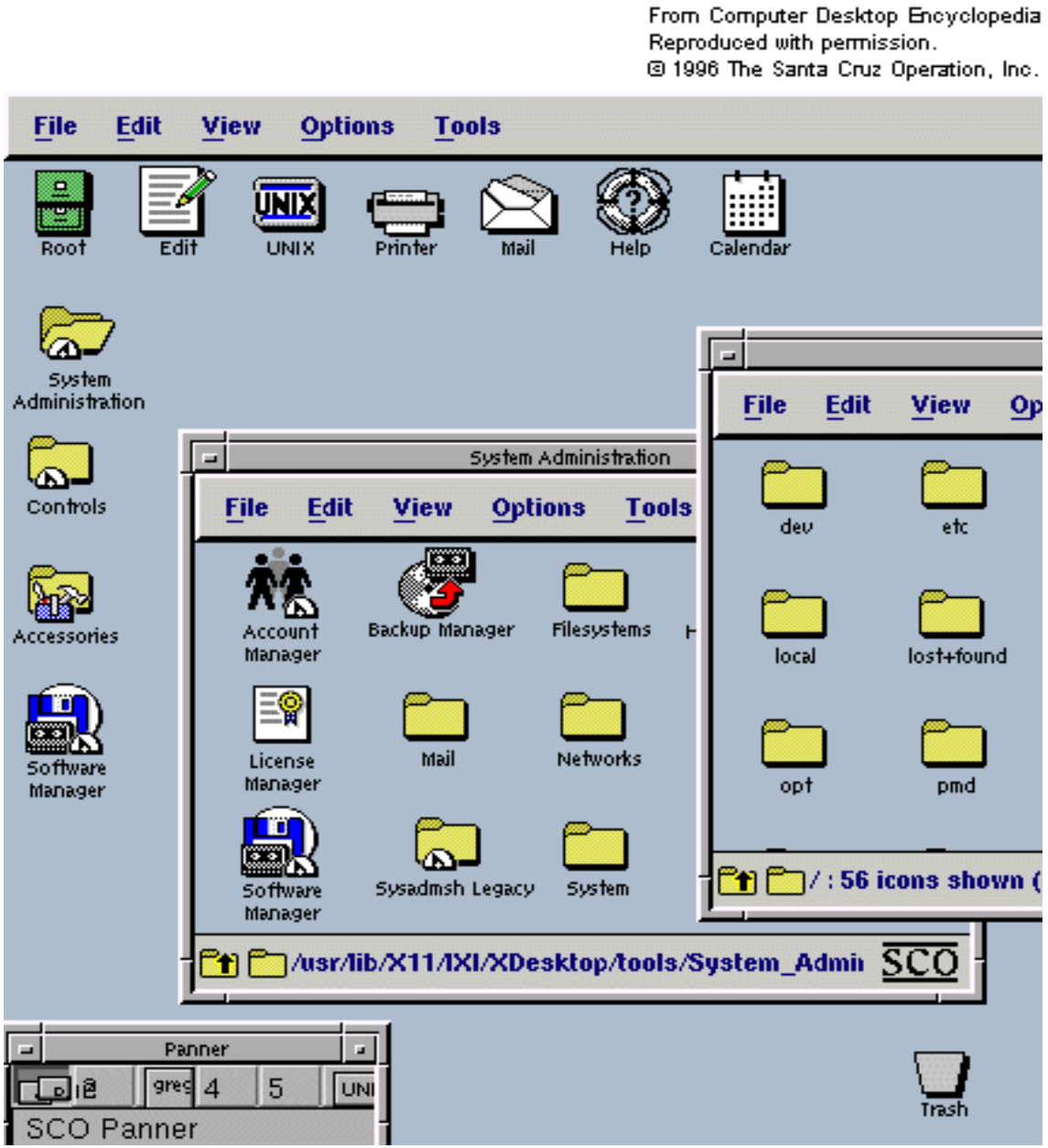
- Objetivo
  - Permitir compartilhamento de uso de funcionalidades para suportar um alto número de objetos "granulares" de forma eficiente.
  - Manter objetos "**peso-leve**", minimizando o custo de memória através do compartilhamento
  - Partes do estado do objeto podem ser compartilhadas, por exemplo, sendo armazenadas em estruturas externas e passadas pros objetos Flyweight (peso-leve) temporariamente quando estes são usados.

Flyweight

- Projetar o sistema até o nível máximo de granularidade pode oferecer flexibilidade máxima, mas o custo em termos de performance é proibitivo
- Trade-off: abre mão da flexibilidade por desempenho
- Normalmente usado em conjunto com uma **filosofia de reuso** de recursos em nome de ganhos de performance, ou seja, se o **projeto** levar em consideração a necessidade de **economia**, o uso do Flyweight é ainda mais eficiente

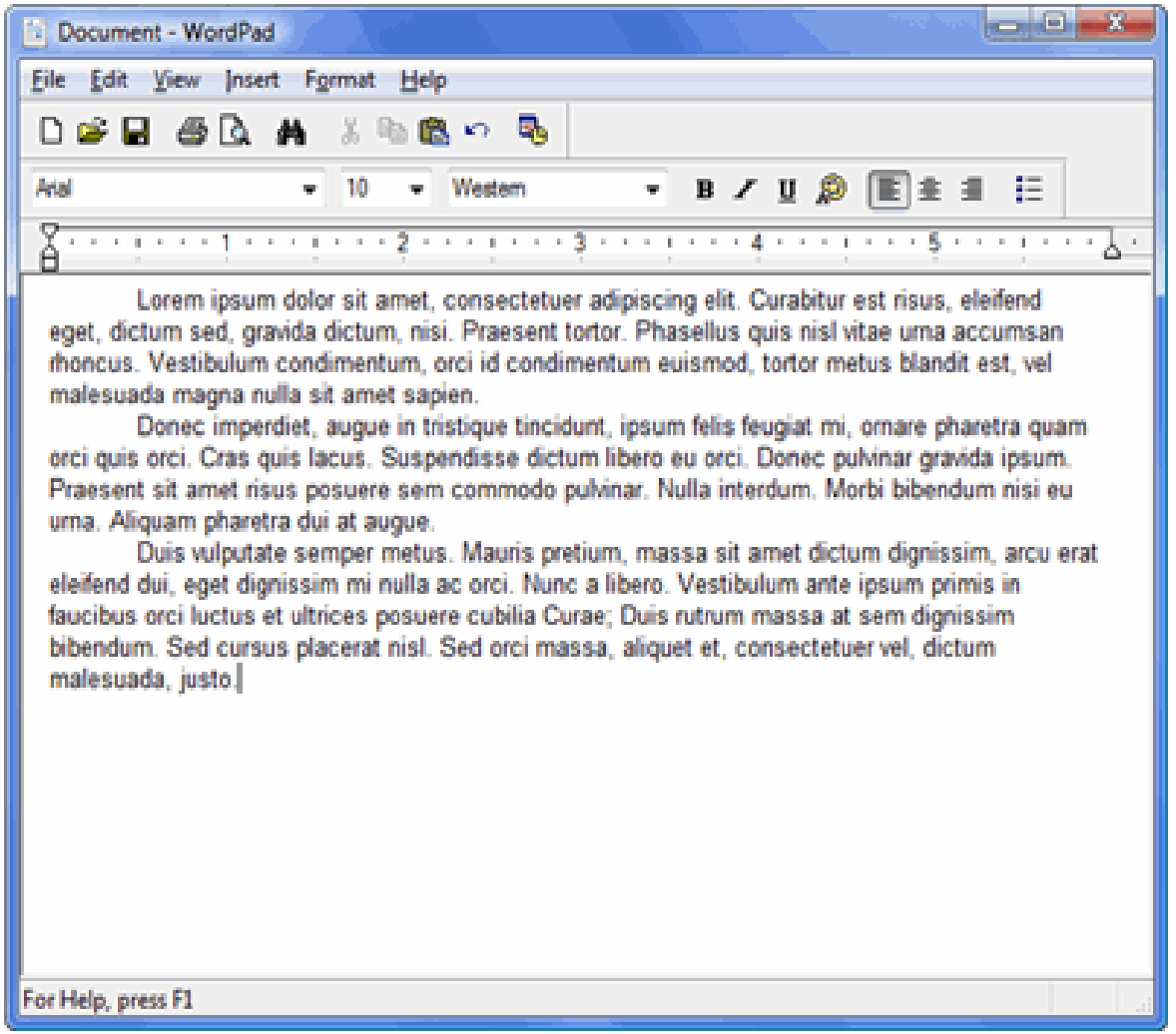
Flyweight

- Exemplo: Motif GUI



Flyweight

- Exemplo: Editor de Texto



Flyweight

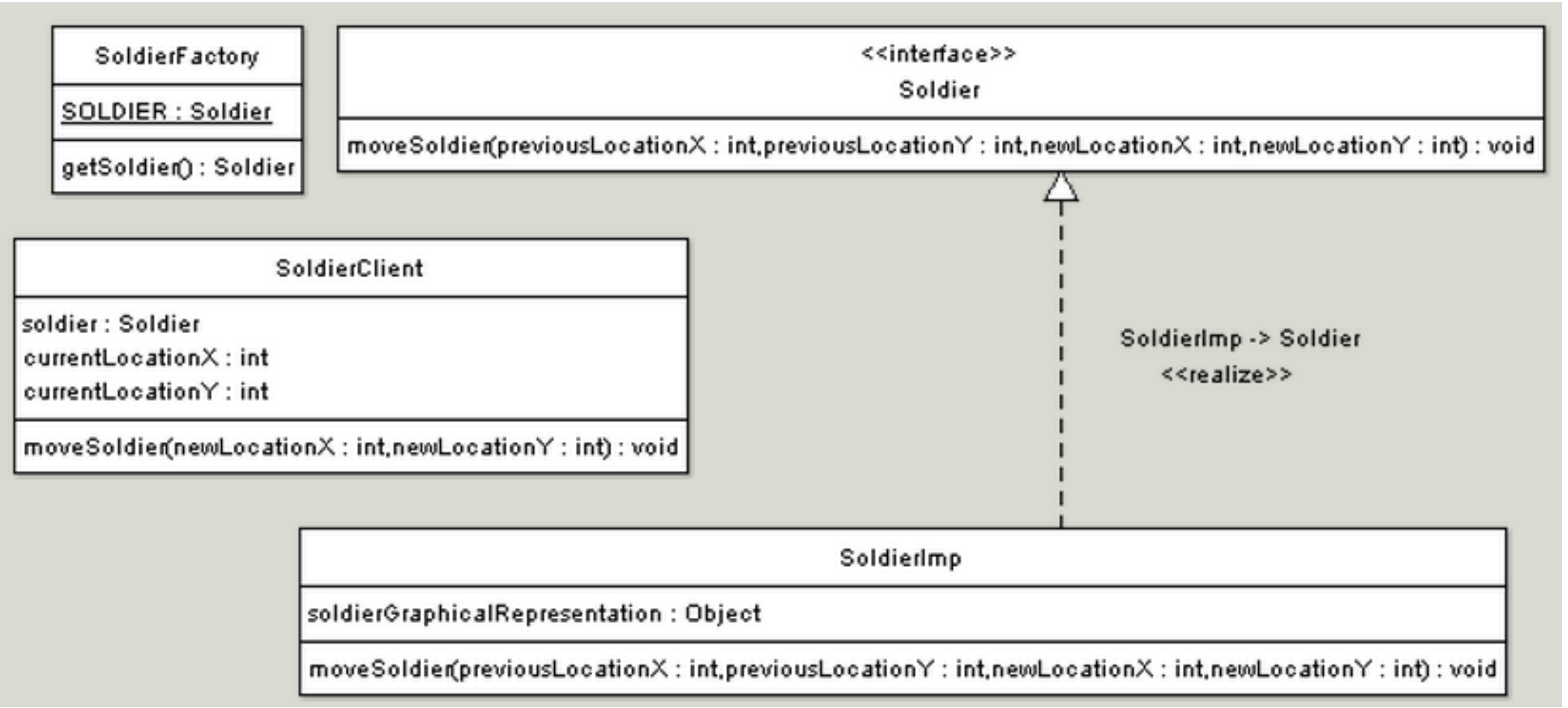
- Exemplo: Editor de Texto
  - Preciso armazenar o gráfico para cada caractere? O consumo de memória nesse caso seria absurdo!

Flyweight

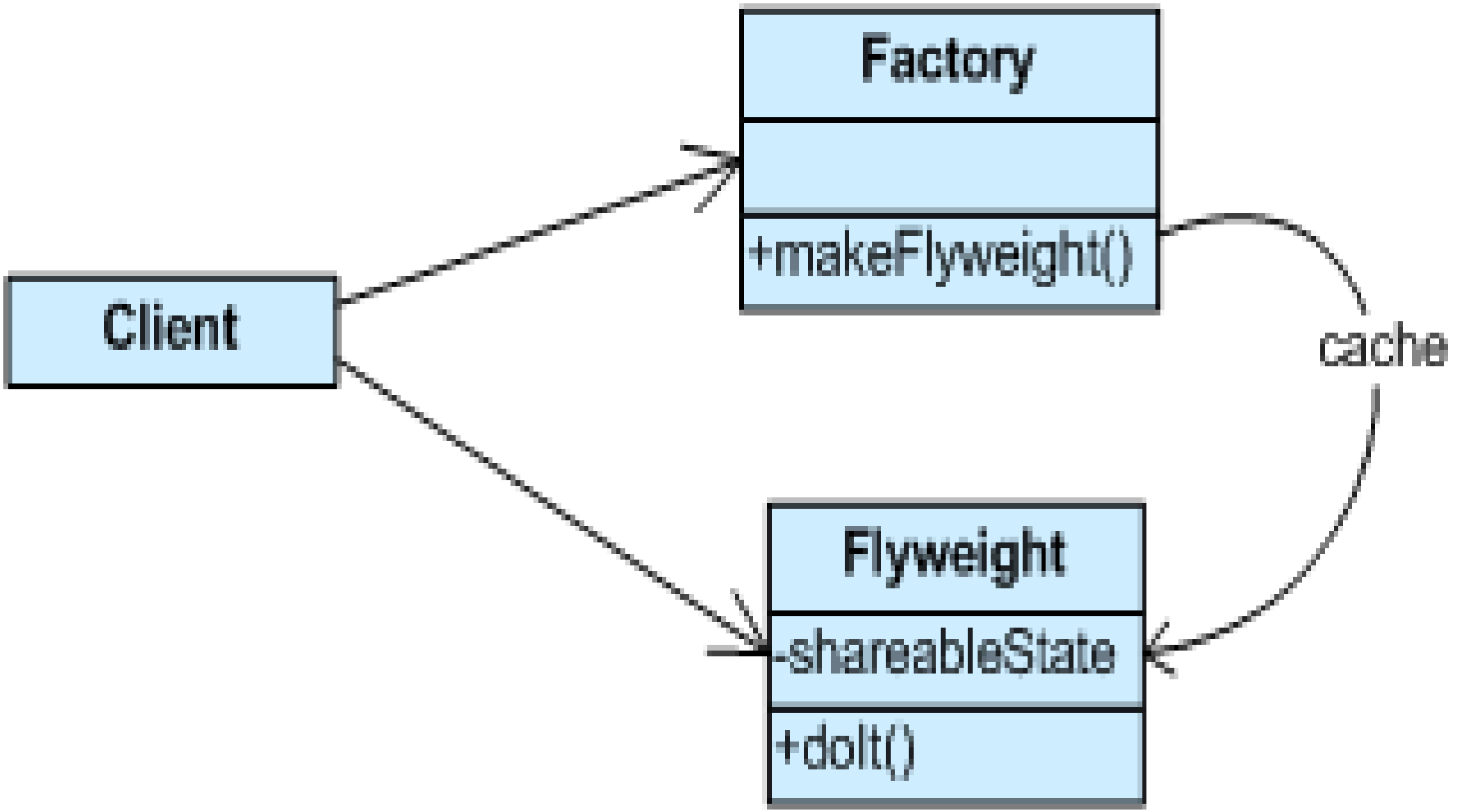
- Exemplo: Jogos de Estratégia

Flyweight

- Exemplo: Jogos de Estratégia



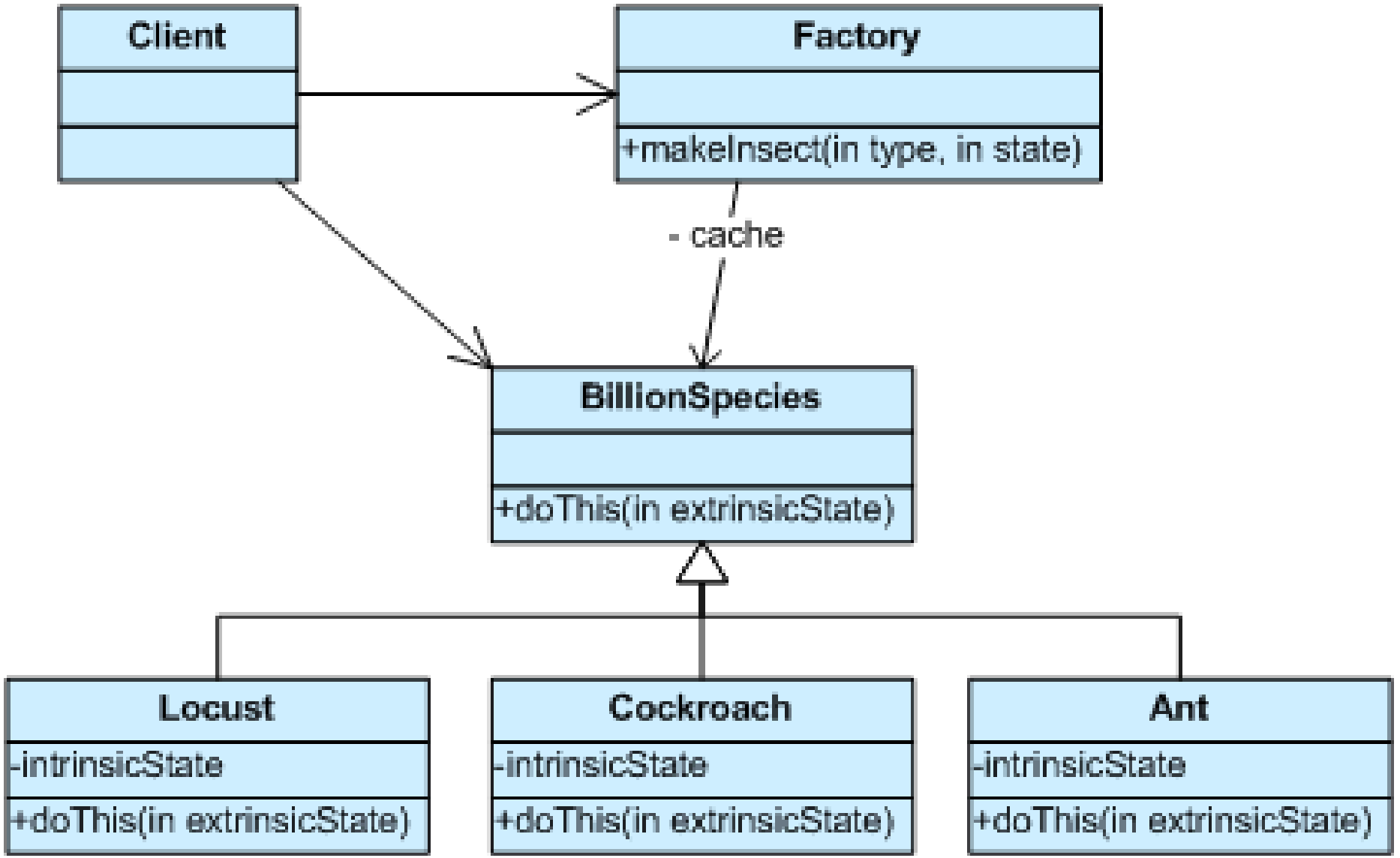
Flyweight


Factory
+makeFlyweight()
Client
cache
Flyweight
-shareableState
+doIt()

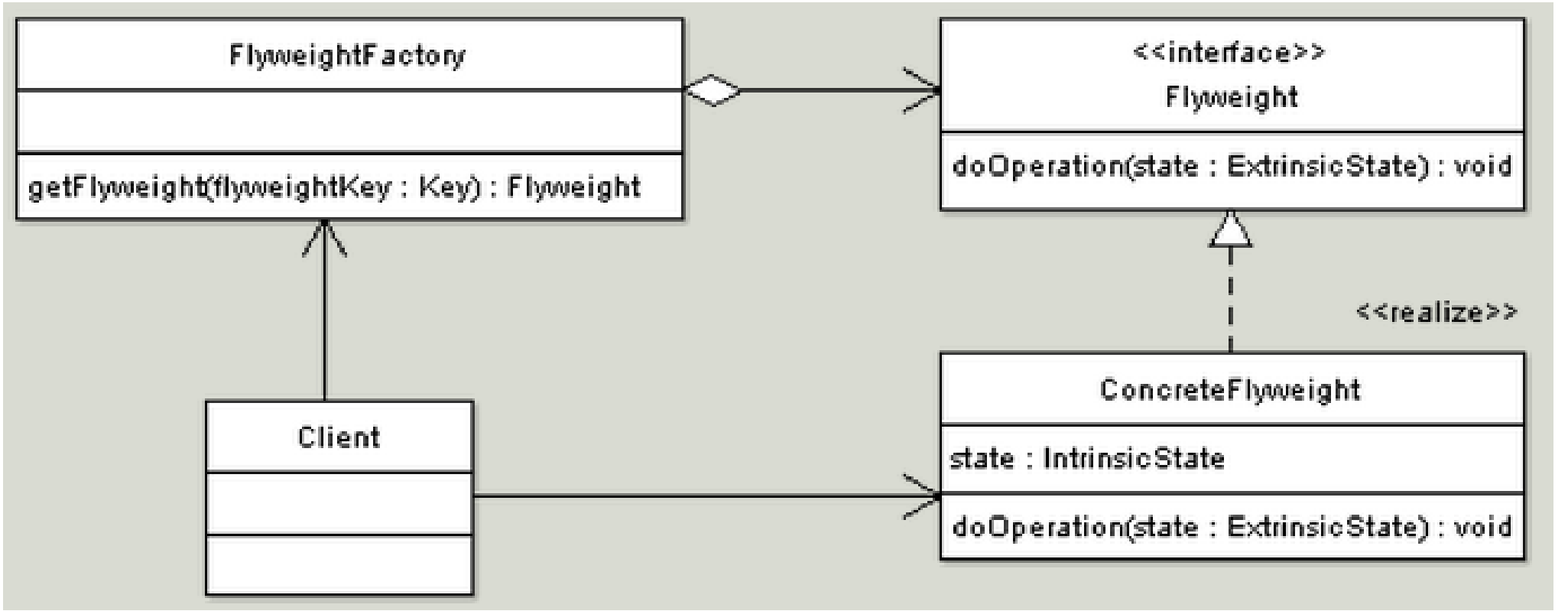
Flyweight

- Cada objeto **Flyweight** é composto por duas partes: a parte dependente de estado (**intrínseca**) e a parte independente de estado (**extrínseca**)
- A parte **extrínseca** é armazenada (compartilhada) no objeto Flyweight. O estado **intrínseco** é armazenado ou computado pelos objetos clientes, e passados pro Flyweight quando suas operações são invocadas.

Flyweight


Client
Factory
+makeInsect(in type, in state)
- cache
BillionSpecies
+doThis(in extrinsicState)
Locust
-intrinsicState
+doThis(in extrinsicState)
Cockroach
-intrinsicState
+doThis(in extrinsicState)
Ant
-intrinsicState
+doThis(in extrinsicState)

Flyweight


FlyweightFactory
getFlyweight(flyweightKey : Key) : Flyweight
<<interface>>
Flyweight
doOperation(state : ExtrinsicState) : void
<<realize>>
Client
ConcreteFlyweight
state : IntrinsicState
doOperation(state : ExtrinsicState) : void

Flyweight

- Participantes
- **Flyweight** – Declara uma interface através da qual flyweights podem receber e responder ao estado extrínseco.
- **ConcreteFlyweight** – Implementa a interface Flyweight e guarda estado intrínseco. O ConcreteFlyweight deve ser compartilhável. No exemplo do jogo de guerra, a repreentação gráfica é estado intrínseco, mas localização e "vida" são estados extrínsecos. O soldado anda, e o comportamento de movimento manipula o estado externo (localização) pra atualizar a posição.
- **Client** – O Cliente mantém referências pros flyweights e computa e mantém estado extrínseco.

Flyweight

- Participantes
- **FlyweightFactory** – A Factory cria e gerencia os flyweights. Também assegura o compartilhamento do Flyweight. **Só cria novos objetos se preciso**, senão retorna referência pra uso.

  No exemplo do jogo de guerra, a Factory pode criar um Flyweight Soldado ou um Flyweight General. Quando o Client pede um soldado pra Factory, ela checa pra ver se já tem um Soldado armazenado no "pool". Se sim, ele é retornado, se não, um Soldado é criado, adicionado ao "pool" e retornado ao Client. A próxima vez que o Client pedir um Soldado, já vai haver um criado.

Flyweight

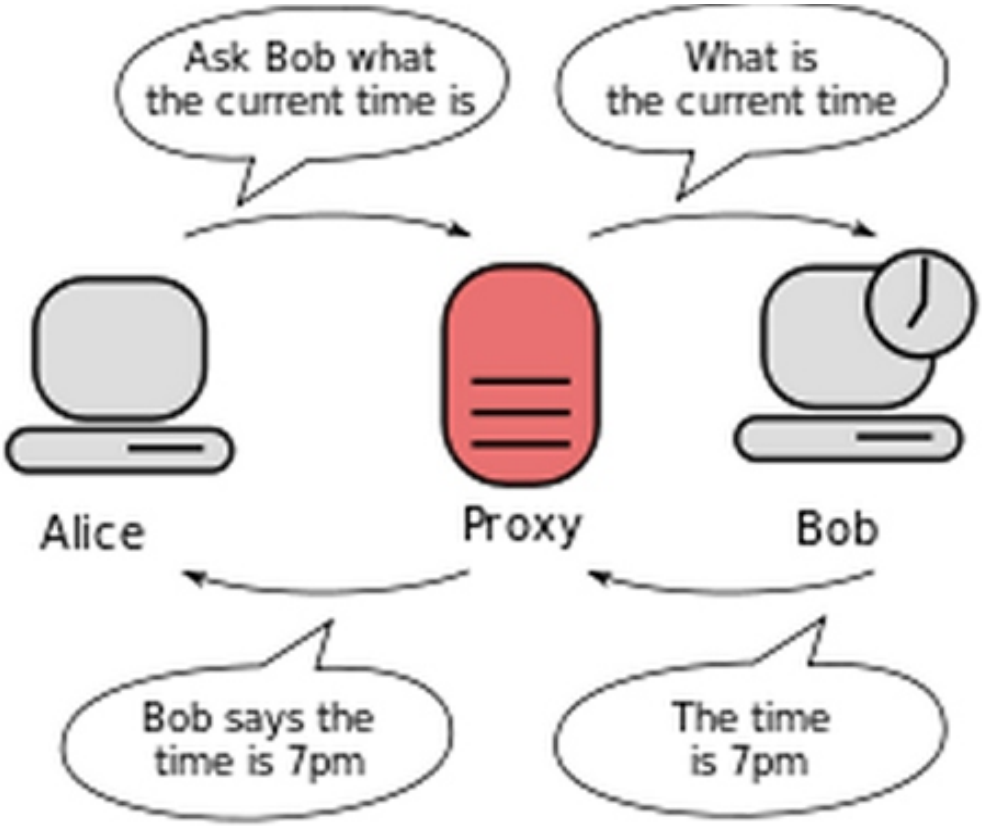
- Passos
- Certifique-se que a carga de objertos é um problema, e o Cliente pode absorver alguma da responsabilidade.
- Divida a classe alvo em estado compartilhável e não compartilhável.
- Remova o estado não-compartilhavel dos atributos da classe, e adicione-o à lista de atributos dos métodos que precisam.
- Crie uma Factory que guarde e reuse instâncias da classe.
- O Cliente pede objetos para a Factory, sem usar o **new**.
- O Cliente ou uma terceira classe deve buscar ou computar o estado não-compartilhável, fornecendo-o pros métodos da classe.

Flyweight

- Java usa **flyweight** para fazer **auto-boxing** de tipos primitivos
  - Conversão de int pra Integer
  - Suponha que temos um vetor de 500 mil zeros...
  - Qual o tamanho no Java de um int? E de um Integer?
  - Lembrando que Java usa **passagem por valor** (https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value/12429953#12429953)
- Pode-se usar Singleton no Factory para garantir que só uma instância de cada Flyweight é criada (lembre-se, o problema é que as instâncias são "pesadas")
- Semelhante ao Object Pool. Quais as diferenças?
  - No Object Pool o problema é a criação do objeto. O objeto pode mudar de estado e depois ser resetado. Não há separação de estado compartihável

Proxy

Proxy


Ask Bob what the current time is
What is the current time
Alice
Proxy
Bob
Bob says the time is 7pm
The time is 7pm

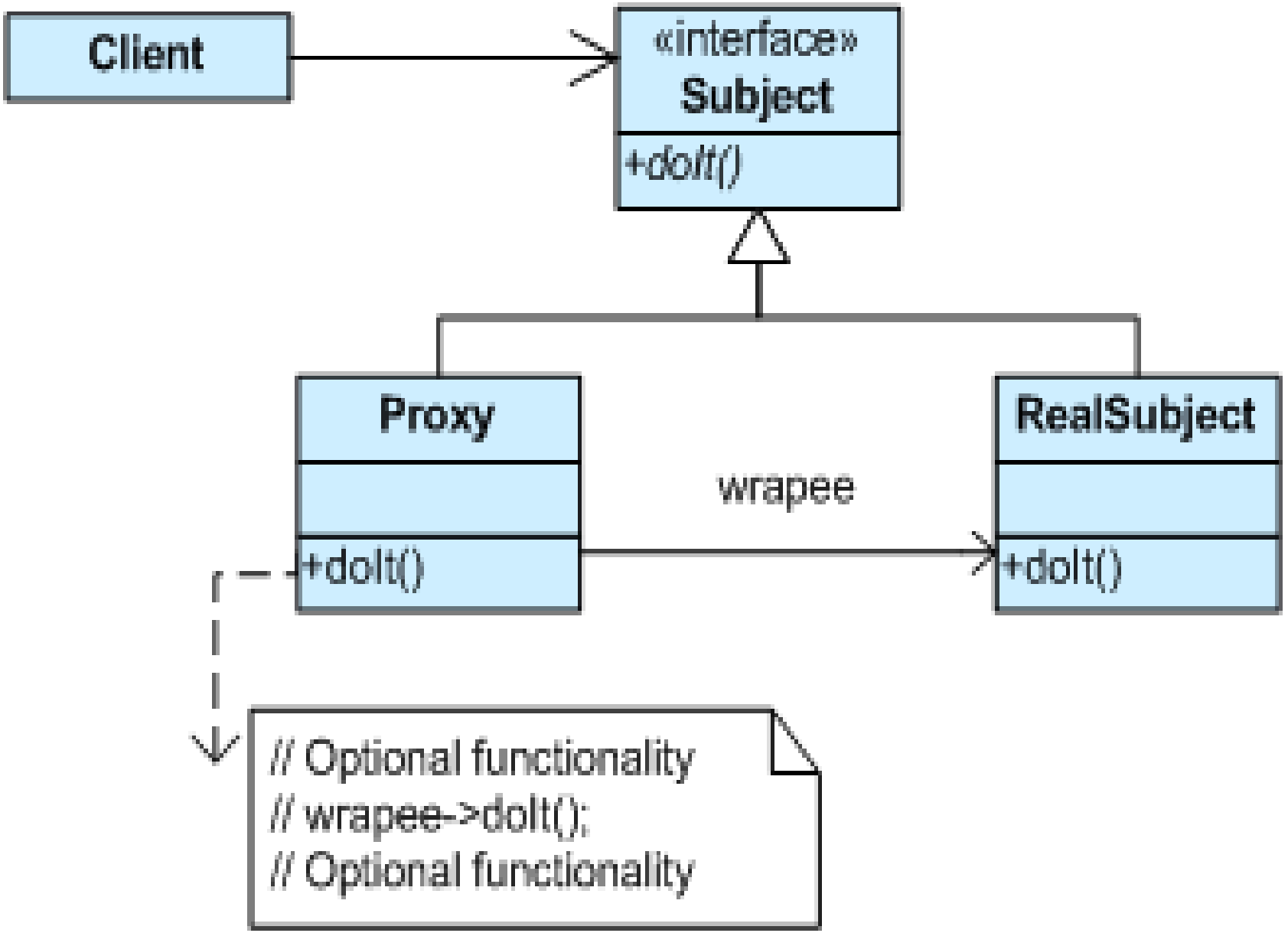
Proxy

- Objetivo
  - Controlar acesso a um objeto através de um objeto intermediário
  - Uma camada a mais de abstração para suportar acesso inteligente
  - Por exemplo, se precisamos só de poucos métodos de objetos custosos, só inicializamos esses objetos quando precisarmos deles. Até lá vamos precisar de objetos leves intermediários pra oferecer a mesma interface dos objetos pesados.
- Proxy = "Procurador", embaixador, substituto

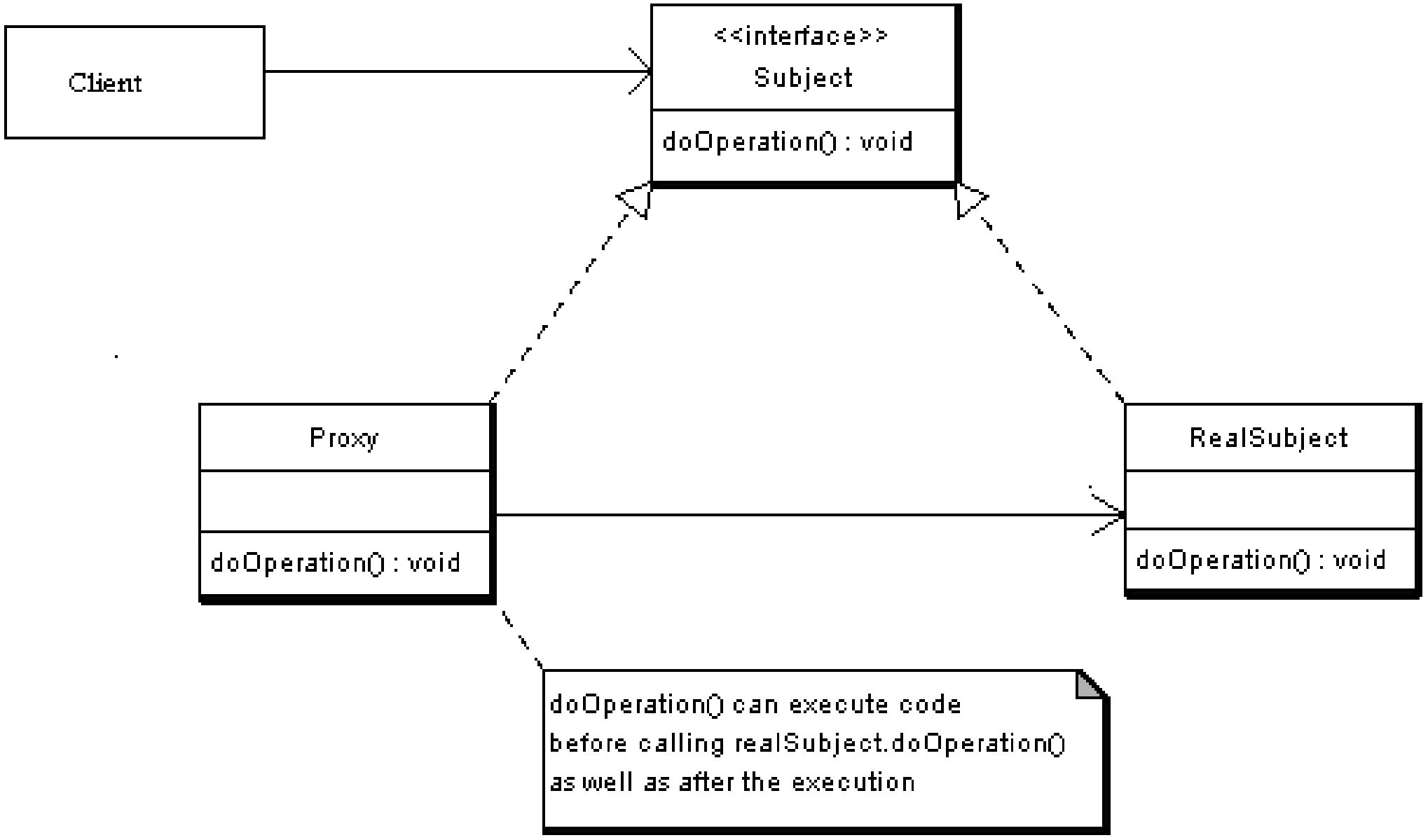
Proxy

- Situações de uso:
  - 1) Um Proxy é um **"placeholder"** pra um objeto mais pesado, provendo a interface dele. O objeto real só é criado quando alguém precisa ou acessa (**lazy init.**).
  - 2) Num **sistema distribuído**, um Proxy remoto provê um representante local de um objeto que está em outro lugar na rede. Chamado frequentemente de "stub" nesses casos (CORBA e RPC).
  - 3) Um Proxy **protetor controla acesso** a um objeto sensível, checando se quem chamou tem permissões antes de repassar o pedido.
  - 4) Um Proxy inteligente pode realizar **tarefas adicionais** quando o objeto principal é referenciado

Proxy


Client
«interface»
Subject
+doIt()
Proxy
+doIt()
wrapee
RealSubject
+doIt()
// Optional functionality
// wrapee->doIt();
// Optional functionality

Proxy


Client
<<interface>>
Subject
doOperation() : void
Proxy
doOperation() : void
RealSubject
doOperation() : void
doOperation() can execute code
before calling realSubject.doOperation()
as well as after the execution

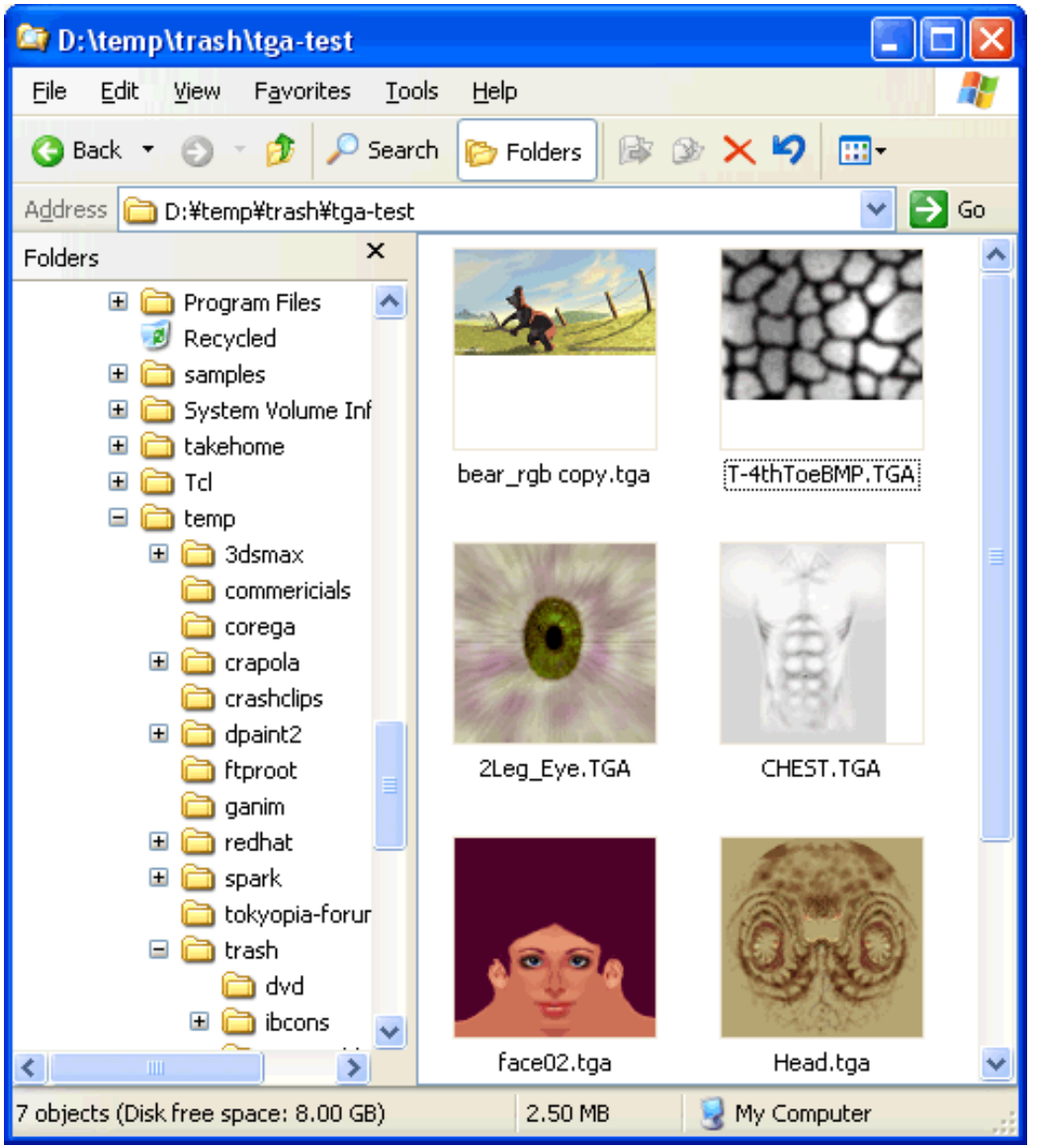
Proxy

- Participantes
  - **Subject** - Interface implementada pelo RealSubject representando suas operações/serviços. O Proxy também implementa essa interface, para poder ser usado no lugar o RealSubject quando necessário.
  - **Proxy** – Mantém a referência de acesso ao RealSubject. Controla accesso ao RealSubject e pode ser responsável por instanciá-lo e destruí-lo.

    Outras responsabilidades dependem do tipo de Proxy.
  - **RealSubject** – O objeto real que o Proxy representa.

Proxy

- Exemplo:
  - Pense num File System Explorer
  - Navegando uma página de imagens, é preciso poder ver o nome dos arquivos, talvez até um "thumbnail" (miniatura)
  - A imagem, porém, só deve ser carregada se o usuário clicar ou pedir para abrir



Proxy

- Passos
  - Identifique o aspecto que deve estar sob um **wrapper** ou **mediador**.
  - Defina uma **interface** que faça o Proxy e o componente original "visualmente" **idênticos**.
  - Considere definir um objeto para **encapsular a decisão** de se referência para o Proxy ou para o objeto original é desejável.
  - A classe **wrapper** (Proxy) tem uma **referência** para a classe real e implementa a interface.
  - Cada método do Proxy realiza sua **contribuição** (checagem de segurança, carrega objeto pesado, verifica concorrência, etc), e **delega** a operação pro objeto **embrulhado/protegido**

Proxy

- **Adapter** oferece uma interface diferente ao objeto. O **Proxy** oferece a mesma interface.