Programação Orientada a Objetos II

Aula 06

Prof. Leandro Nogueira Couto
UFU – Monte Carmelo

Recapitulando

- Vimos até aqui os **Padrões de Projeto Creacionais**:
  - Factory Method
  - Abstract Factory
  - Singleton
  - Prototype
  - Builder
- Relacionados com a **instanciação e criação** de objetos

Padrões Estruturais

- Iremos agora falar de **Padrões de Projeto Estruturais**
- Facilitam o projeto/design do sistema
- Promovem soluções para implementar **relações** entre entidades
- Métodos eficientes para **organização** que preservam princípios da POO

Padrões Estruturais

- Iremos agora falar de **Padrões de Projeto Estruturais**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

Adapter

Adapter

- Imagine que:
- Queremos realizar a **comunicação com uma classe previamente projetada**, mas sem alterá-la demais.
- Precisamos **converter a interface da classe** para outra interface que nosso cliente espera.
- Um componente já pronto "da prateleira" oferece uma **funcionalidade interessante**, mas a "visão de universo" do componente **não é compatível** com a filosofia e arquitetura do sistema sendo desenvolvido.

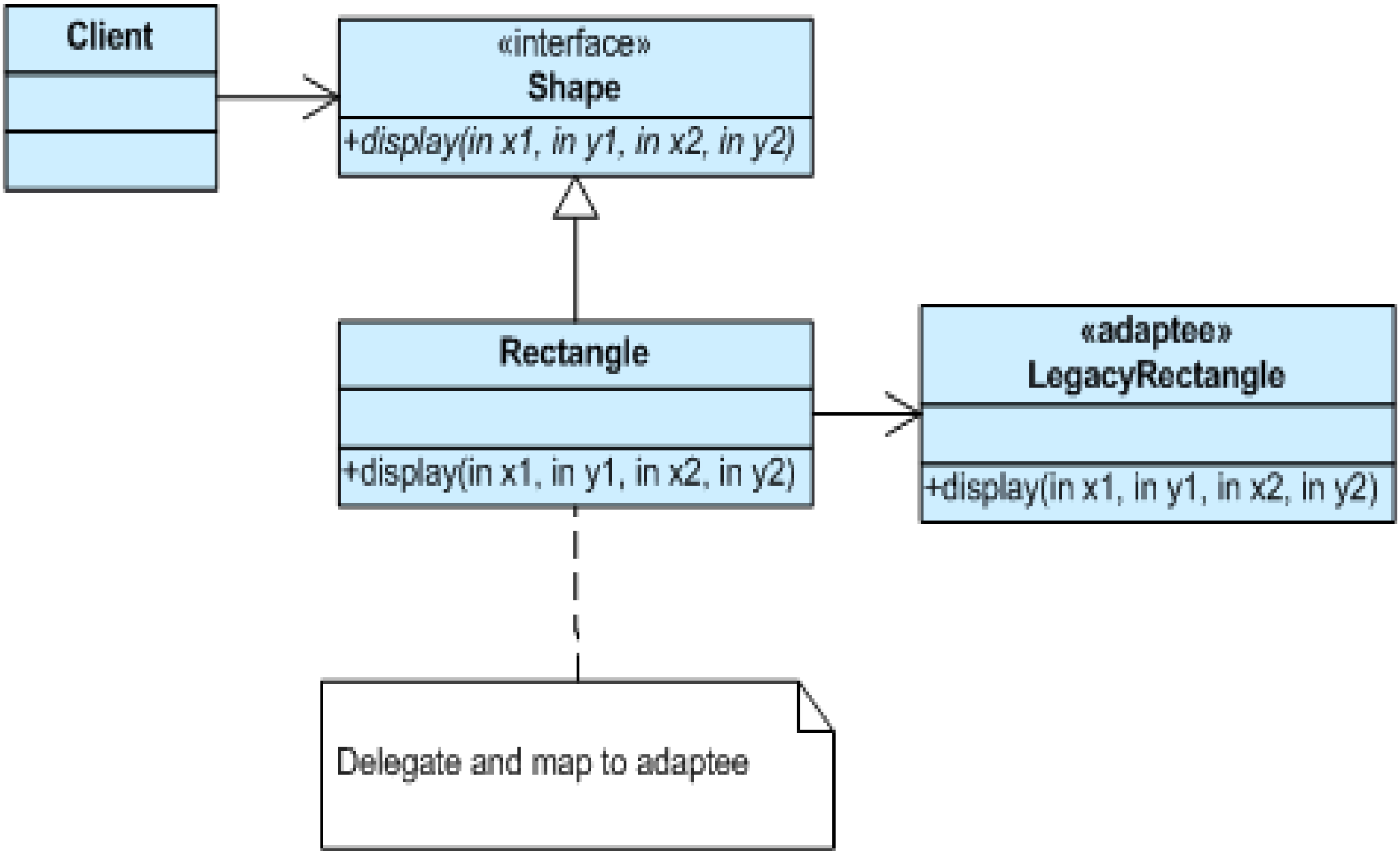
Adapter

- O Adapter permite que **classes trabalhem juntas** que de outra forma não poderiam por causa de interfaces incompatíveis.
- **Embala** uma classe existente em uma nova interface: **Wrapper**.
- Faz o casamento de um **componente antigo** em um **sistema novo**.
- Adiciona-se uma camada que **Traduz chamadas** e **retornos** de uma classe para outra

Adapter

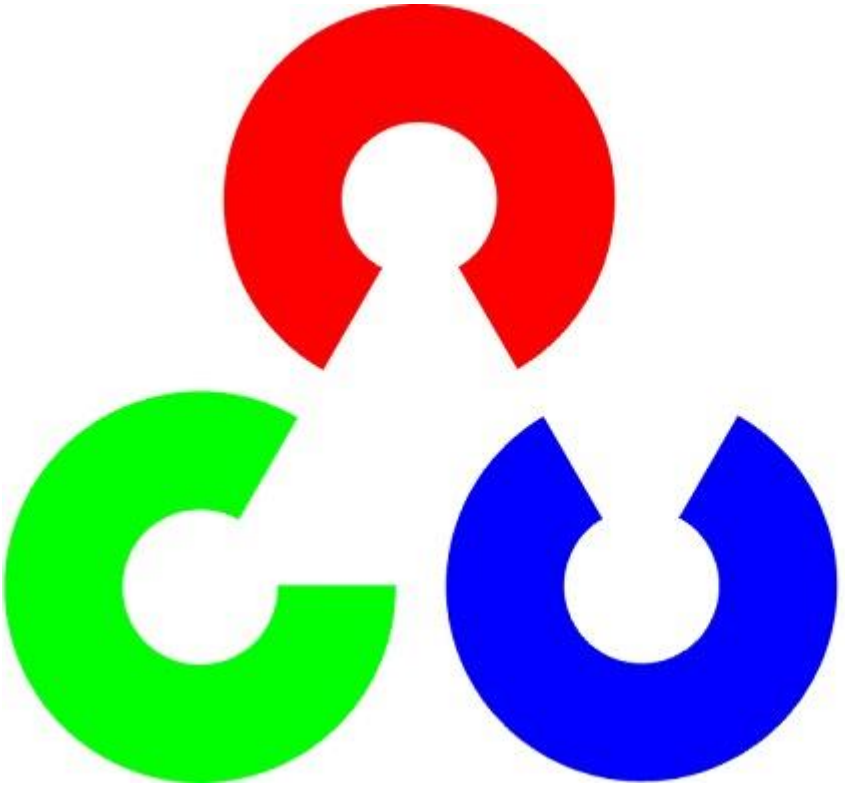
- O reuso de software é uma tarefa complexa.
- Sempre há algo errado ou incompatível entre o "velho" e o "novo": dimensão, filosofia de projeto, *timing* e sincronização, etc.
- Exemplo:
  - Uma classe **Rectangle** tem um método **display()** que espera receber “x, y, w, h” como parâmetros. Mas o **Cliente** quer passar “x e y superior esquerdo” e “x e y inferior direito”.
  - Essa incongruência é resolvida, sem mudar as classes originais, com um nível extra de abstração: um objeto **Adapter**.

Adapter


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in x2, in y2)
Delegate and map to adaptee

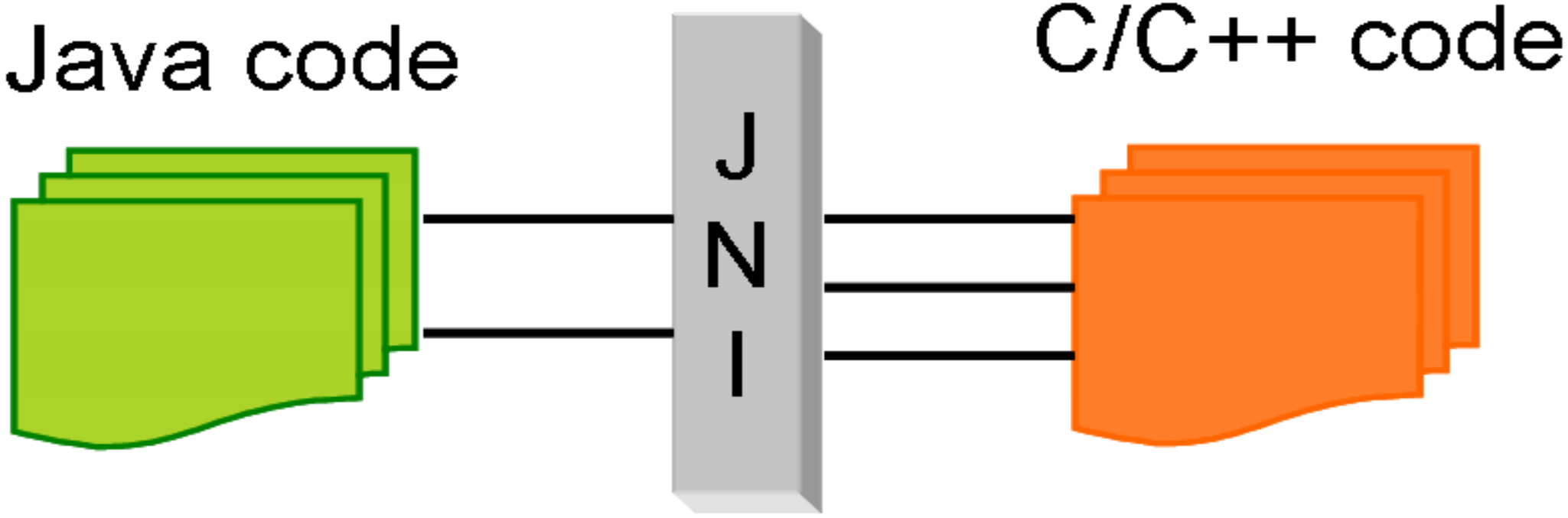
Adapter

- Exemplo: JavaCV



Adapter

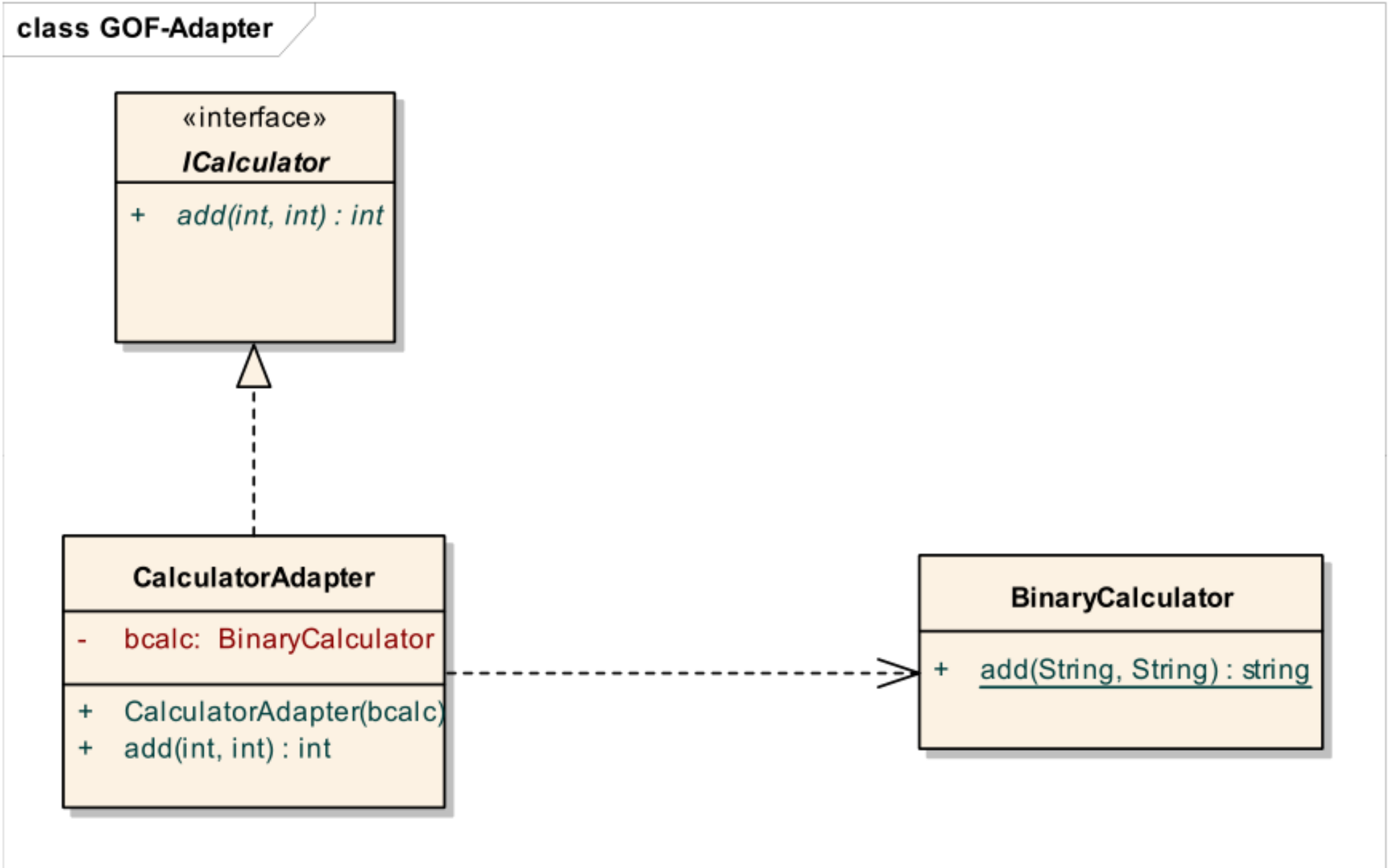
- Exemplo: Java Native Interface



Adapter

- Outro exemplo:
- Usar uma calculadora binária como uma calculadora decimal

Adapter

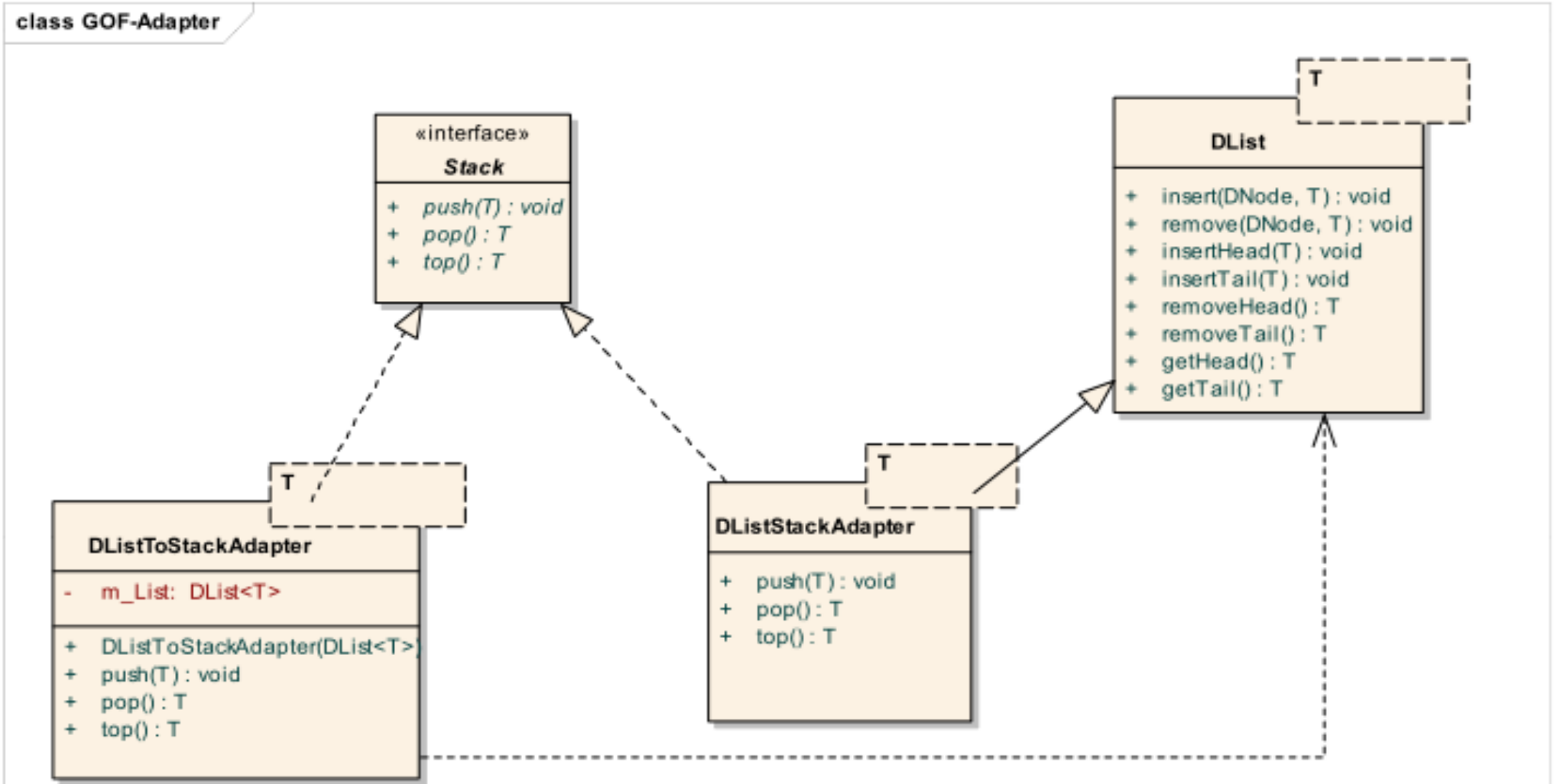

class GOF-Adapter
«interface»
ICalculator
+ add(int, int) : int
CalculatorAdapter
- bcalc: BinaryCalculator
+ CalculatorAdapter(bcalc)
+ add(int, int) : int
BinaryCalculator
+ add(String, String) : string

Adapter

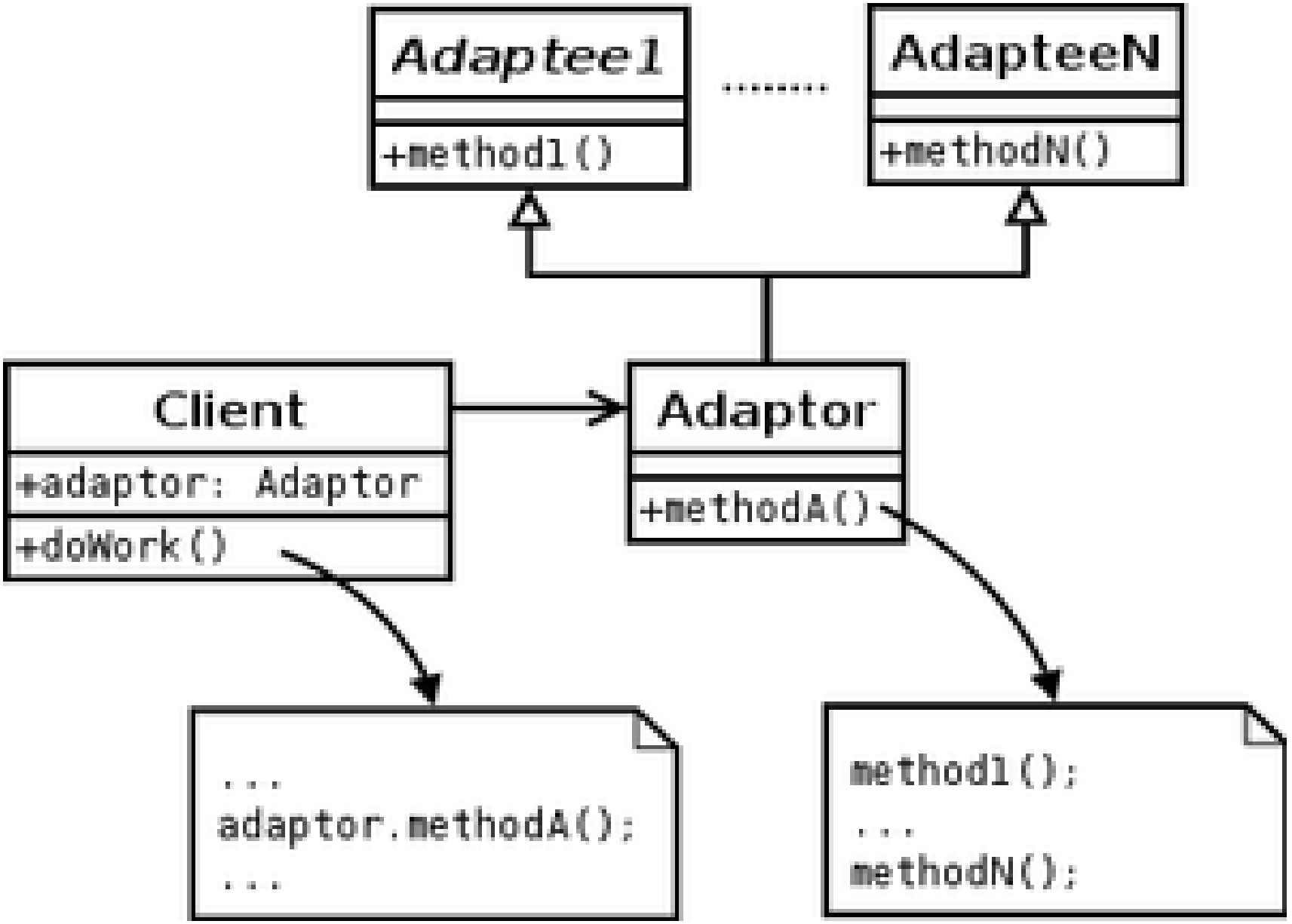
```
public interface ICalculator{
    public int add(int ia , int ib);
}
public class BinaryCalculator {
    public static string add(String sa,String sb){   //…
    }
}
public class CalculatorAdapter implements ICalculator {
    private BinaryCalculator bcalc;
    public CalculatorAdapter(bcalc c){
        bcalc = c;
    }
   public int add(int ia, int ib){
        String result;
        result = bcalc.add(Integer.toBinaryString(ia),   Integer.toBinaryString(ib),
        //converts binary string to a decimal representation return is value
        return Integer.valueOf(result,10).intValue();
```

Adapter

- Dois exemplos de classes adaptadoras (traduzindo de lista para pilha). Uma baseada em uso (**DListToStackAdapter**) e outra em herança múltipla (**DListStackAdapter**)



Adapter


Adaptee1
+method1()
.....
AdapteeN
+methodN()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+methodA()
...
adaptor.methodA();
...
method1();
...
methodN();

Adapter

- Passos
  - Identifique os participantes: o componente que ao qual precisamos nos adequar (o Cliente) e o componente que precisa se adaptar.
  - Identique a interface que o cliente requer.
  - Projete um embrulho/embalagem/wrapper que faça o casamento entre o adaptado e o Cliente.
  - O embrulho/wrapper "tem uma" instância da classe adaptada.
  - O embrulho/wrapper mapeia a interface do cliente pra interface do adaptado.
  - O cliente pode usar a nova interface (do wrapper)

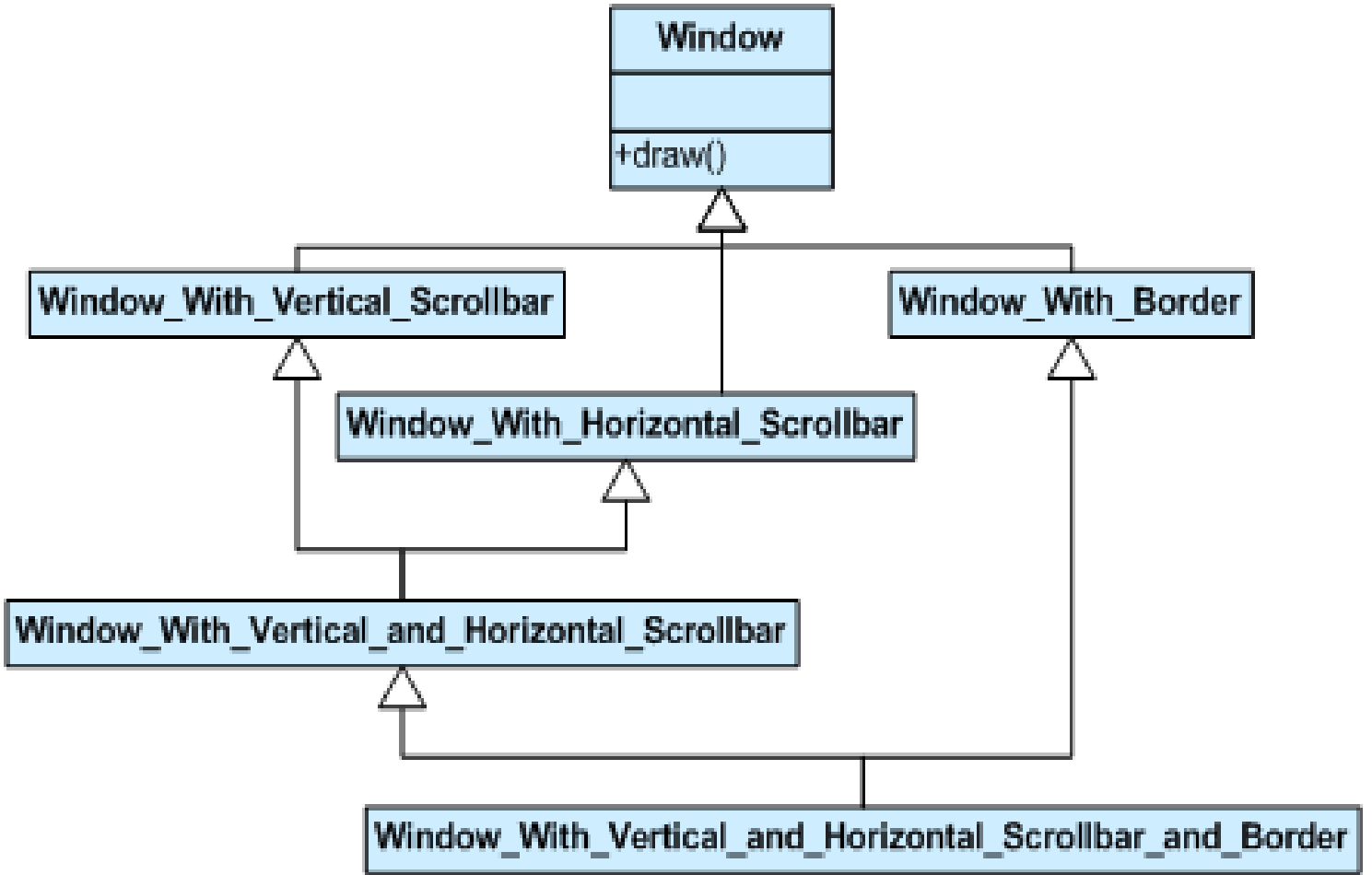
Decorator

Decorator

- Imagine que:
- Queremos adicionar comportamentos a um objeto em tempo de execução, sem usar herança. É possível?

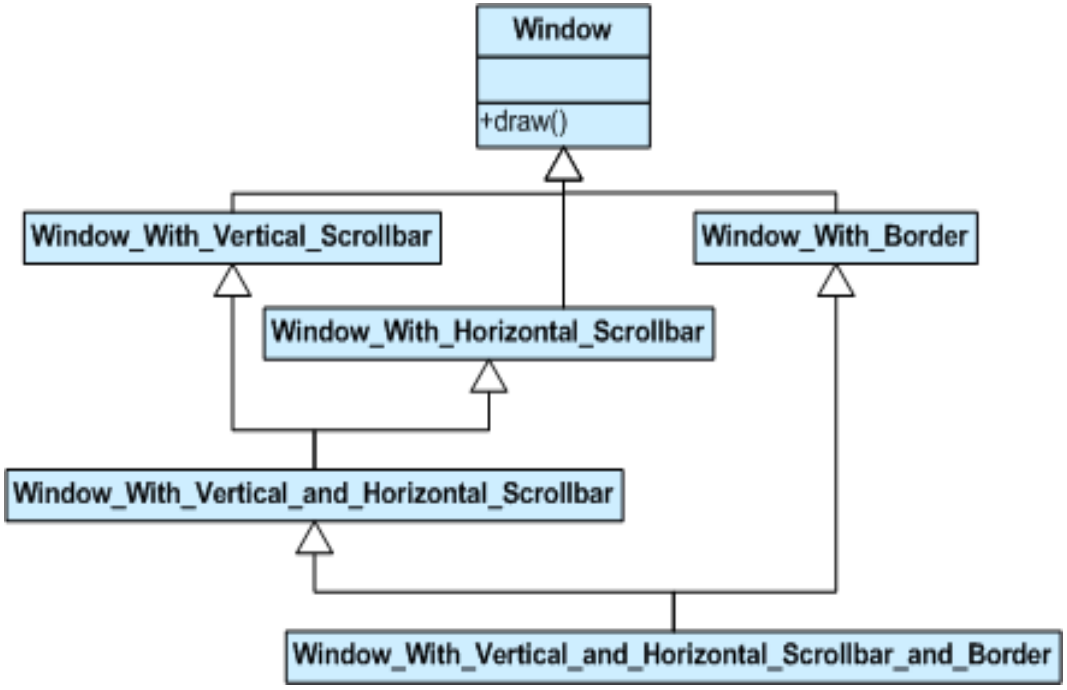
Decorator

- Exemplo: Suponha que você está trabalhando num gerenciador de interface e deseja suportar que o usuário adicione bordas e barras de rolagem às janelas. Uma possibilidade é a seguinte hierarquia:

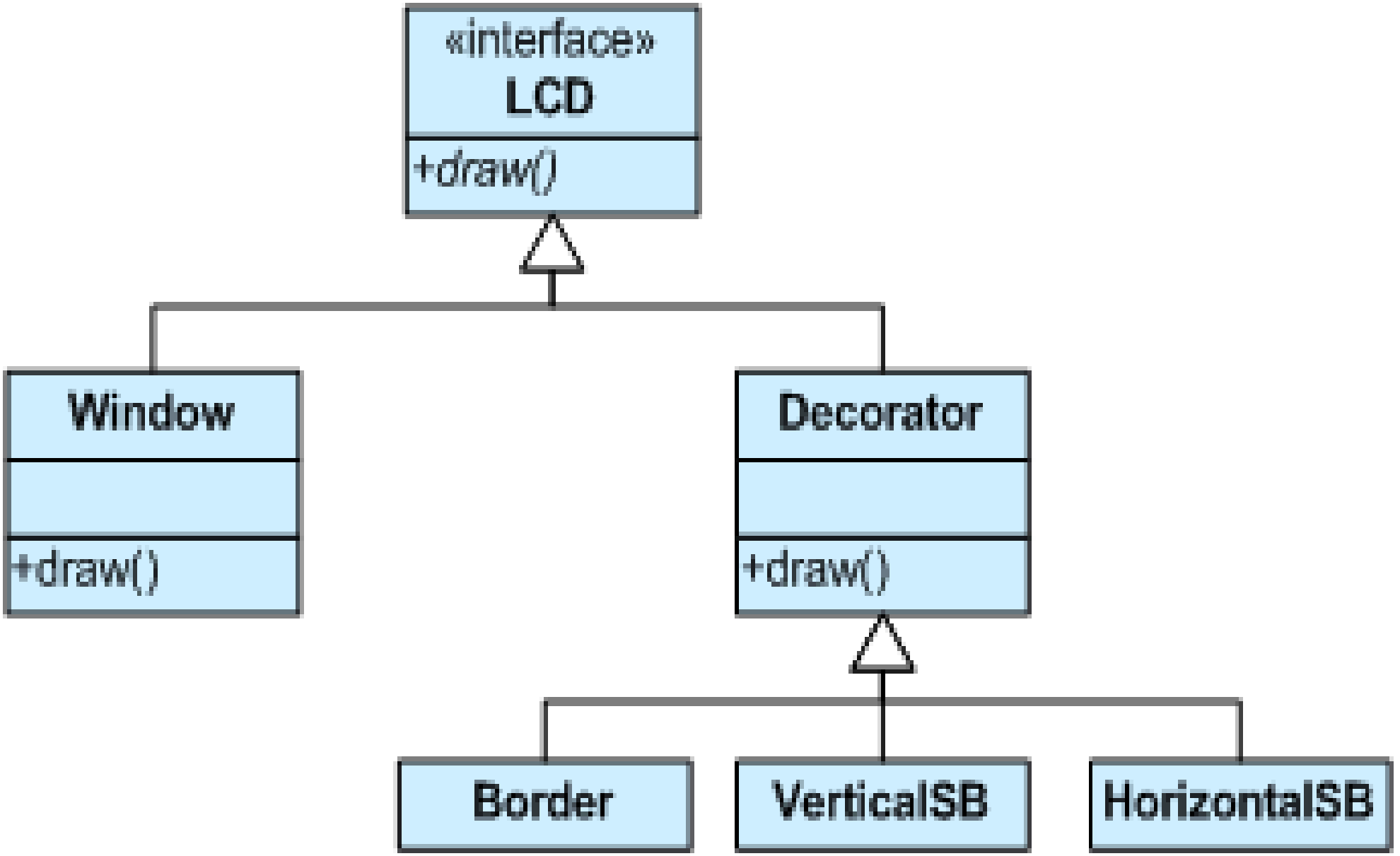


Decorator

- Parece um tanto... inadequado!
  - E se quisermos adicionar a opção de uma janela "com botão de minimizar" nesse projeto?
- Gostaríamos de poder dar ao objeto Window qualquer combinação de "features" que quisermos, sem precisar de uma subclasse toda vez.
- É isso que o Decorator propõe fazer.



Decorator

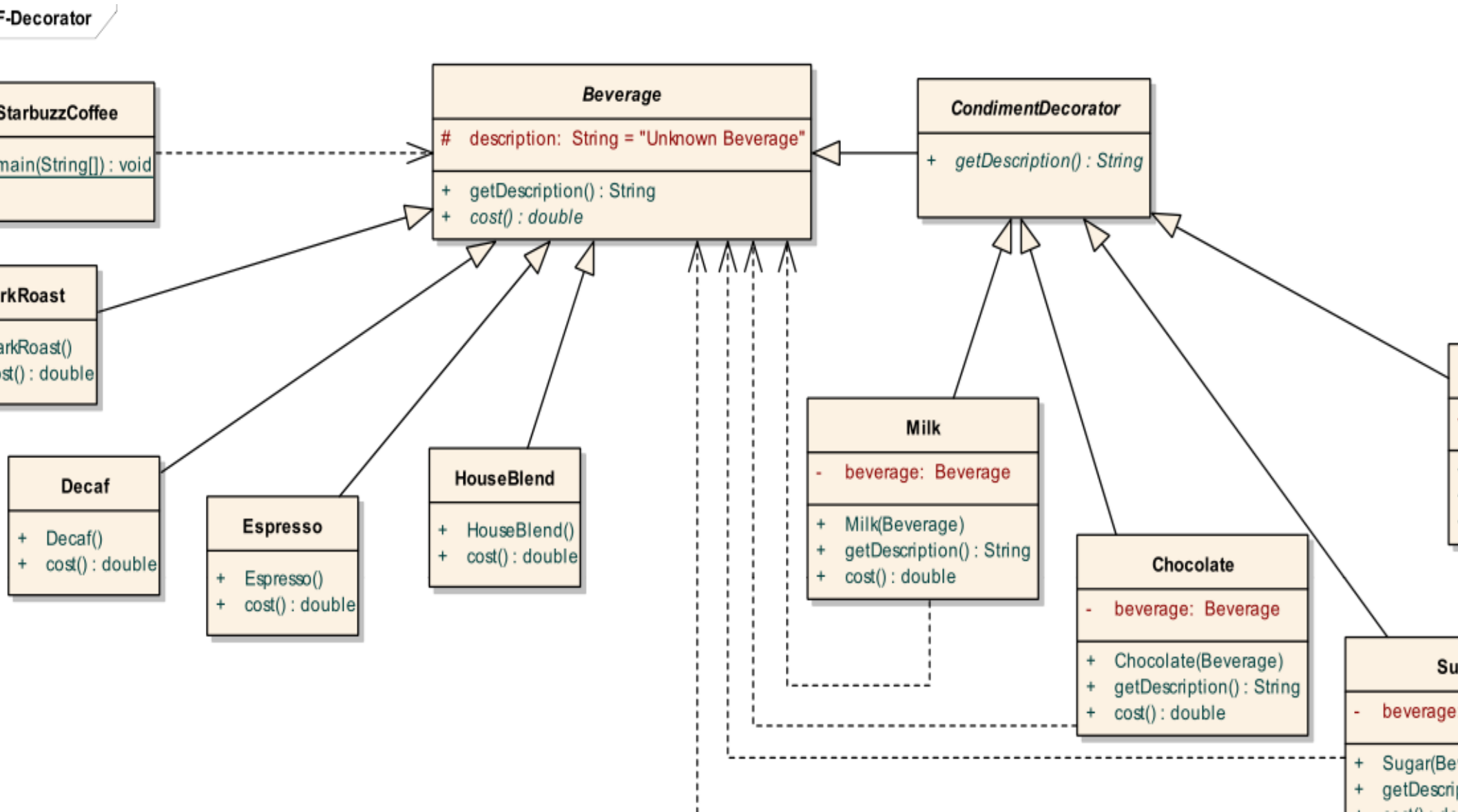

«interface»
LCD
+draw()
Window
+draw()
Decorator
+draw()
Border
VerticalSB
HorizontalSB

Decorator

```
Widget*  aWidget = new BorderDecorator(
                       new HorizontalScrollBarDecorator(
                         new VerticalScrollBarDecorator(
                           new Window( 80, 24 ))));
aWidget->draw();
```

Decorator

- Exemplo do Use a Cabeça – Padrões de Projeto

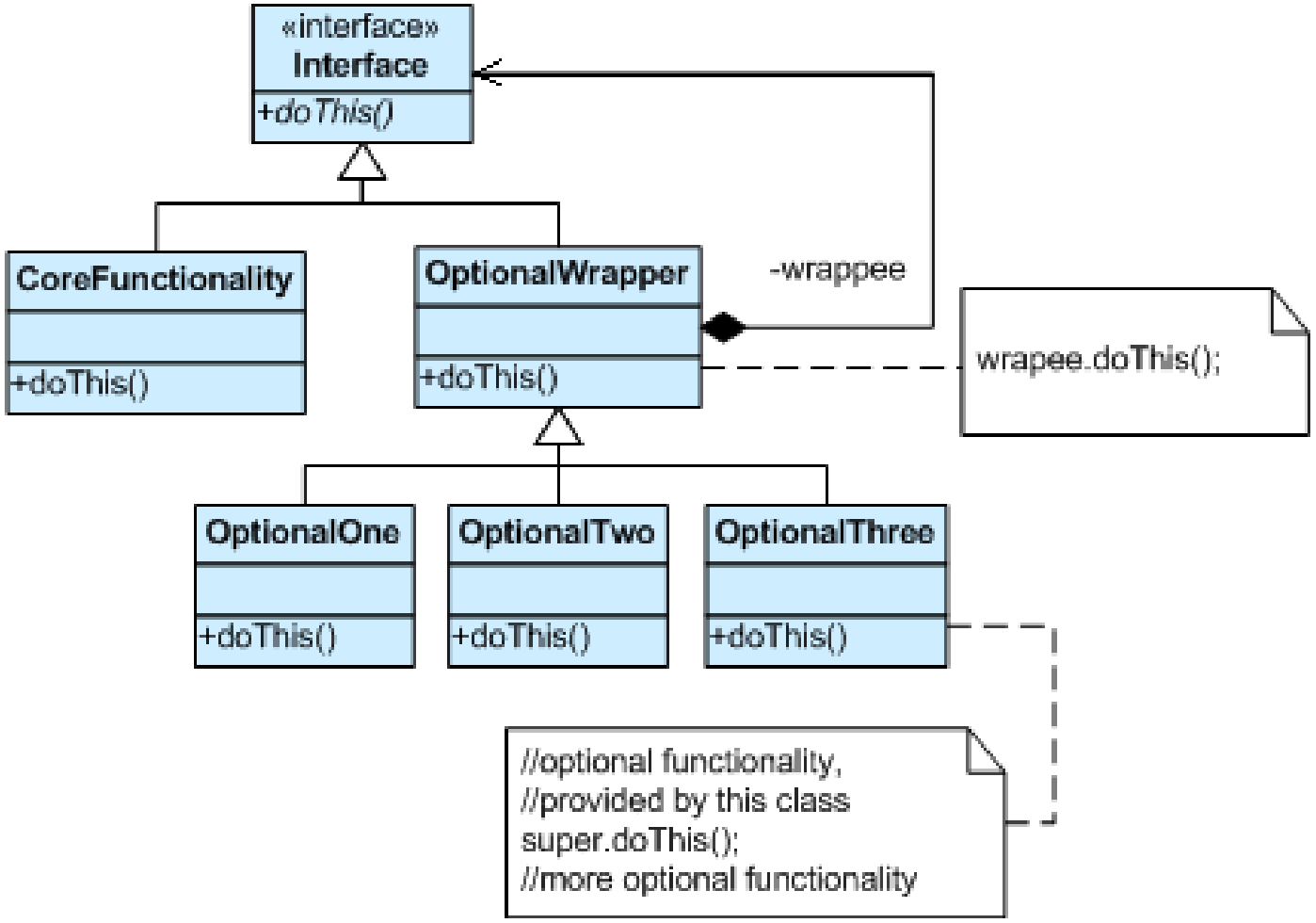


Decorator

- **Encapsulamos** o objeto original dentro de uma interface estilo wrapper! (O Decorator também é chamado de wrapper às vezes)
- Tanto os **objetos decoradores** (scrollbar horizontal, scrollbar horizontal, border) e o **objeto principal** (window) **herdam dessa interface abstrata**. A interface usa **composição recursiva** para permitir que um número ilimitado de camadas sejam adicionadas ao objeto central.
- Não adicionamos métodos, mas **adicionamos responsabilidades ao objeto**.
- Note que o objeto está escondido agora atrás da interface. Acessá-lo se torna mais problemático

Decorator

- O Cliente está interessado em CoreFunctionality.doThis(). Ele pode ou não estar interessado em OptionalOne.doThis() e OptionalTwo.doThis().



Decorator

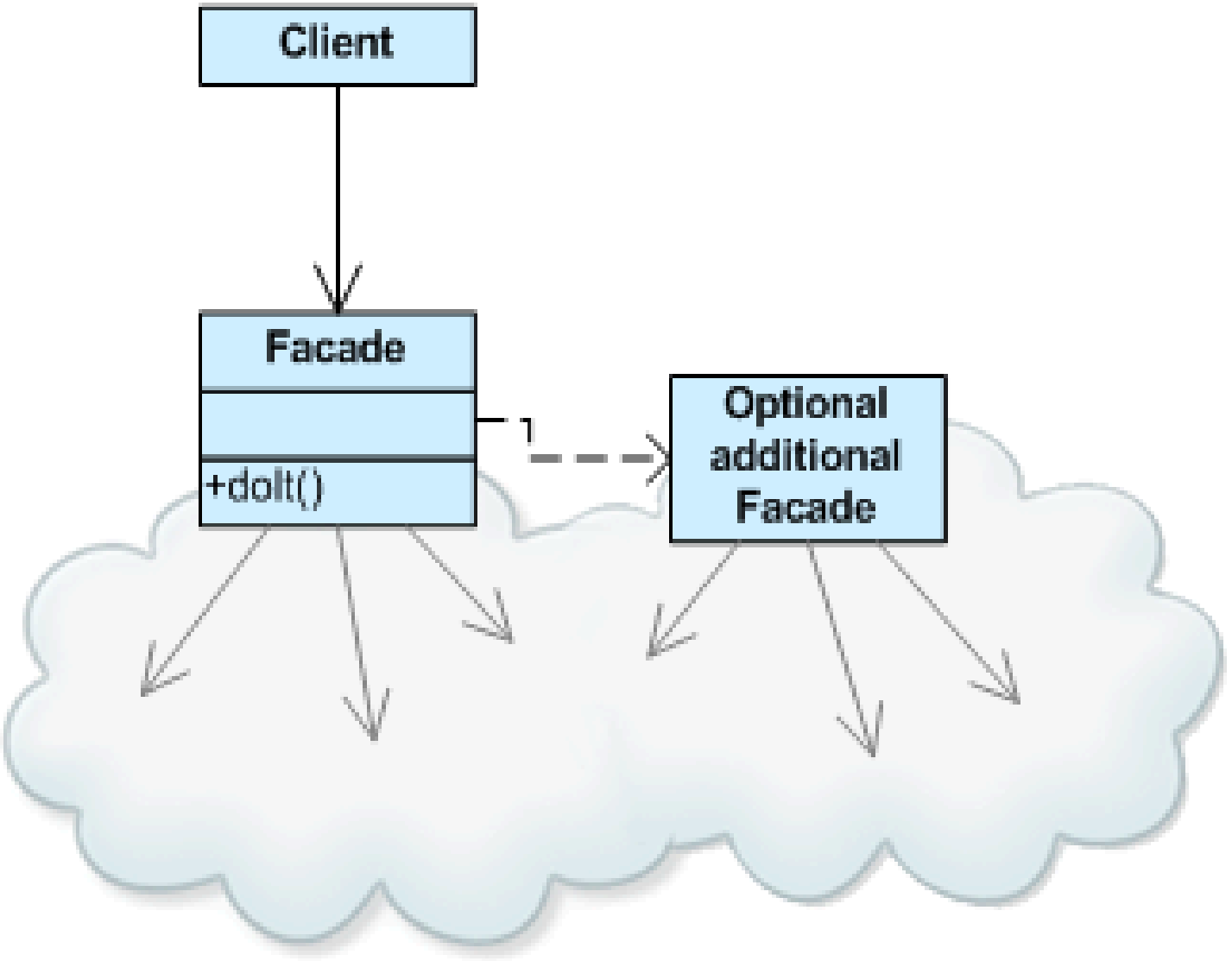
- Passos
  - Certifique-se que o contexto é: uma única classe central (não-opcional), vários adicionais opcionais, e uma interface comum a tudo isso.
  - Crie uma interface “Mínimo Denominador Comum” que torna todas as classes intercambiáveis.
  - Crrie uma segunda classe base (Decorator) pra suportar os opcionais.
  - A classe central e o Decorador herdam da interface Mínimo Denominador Comum.
  - Cada derivado do Decorator declara uma relação de composição com a interface MDC, e a classe inicializa a o membro interface no construtor.
  - Defina uma classe derivada do Decorator para cada opcional.

Facade

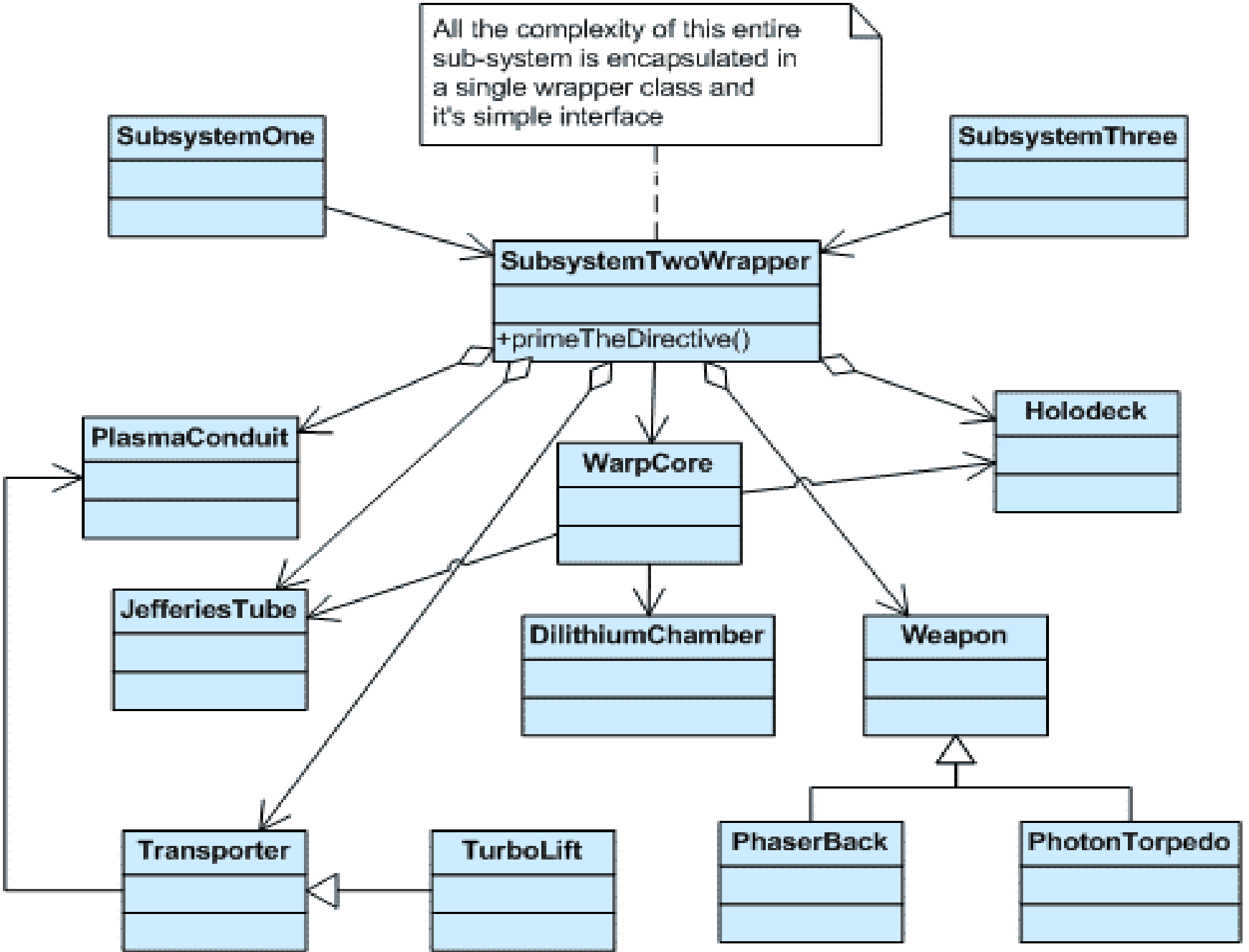
Facade

- Escrito como **Facade** ou **Façade**
- Fornece uma **interface comum**, uma "fachada", para um grupo de classes de um subsistema, facilitando o seu uso
- Embrulha um subsistema complicado em uma interface mais simples
- Motivação: **Reduzir o acomplamento entre sistemas**
  - Se temos um sistema complexo, com vários pontos de acesso, o que acontece se ele tem muitos Clientes?
  - Com Facade, deixamos o sistema mais modular

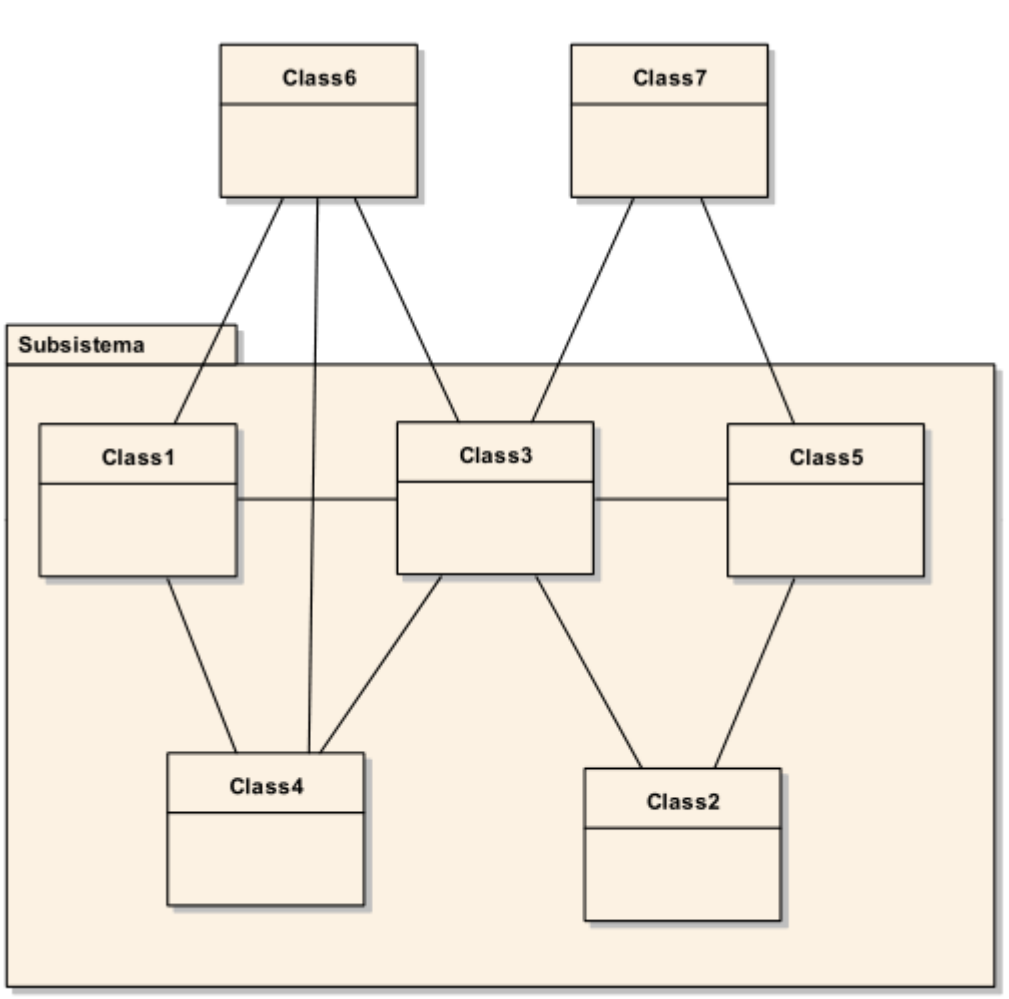
Facade

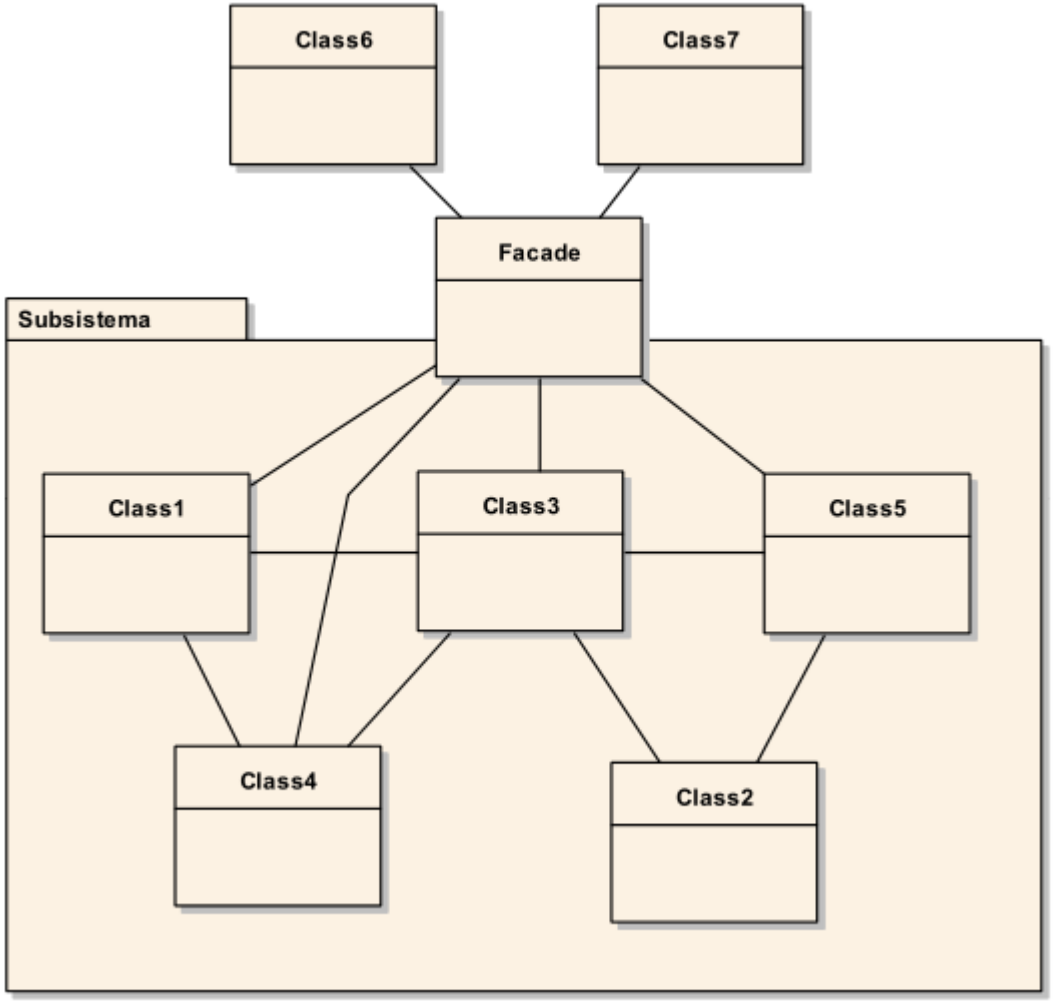


Facade


All the complexity of this entire sub-system is encapsulated in a single wrapper class and it's simple interface
SubsystemOne
SubsystemThree
SubsystemTwoWrapper
+primeTheDirective()
PlasmaConduit
WarpCore
Holodeck
JefferiesTube
DilithiumChamber
Weapon
Transporter
TurboLift
PhaserBack
PhotonTorpedo

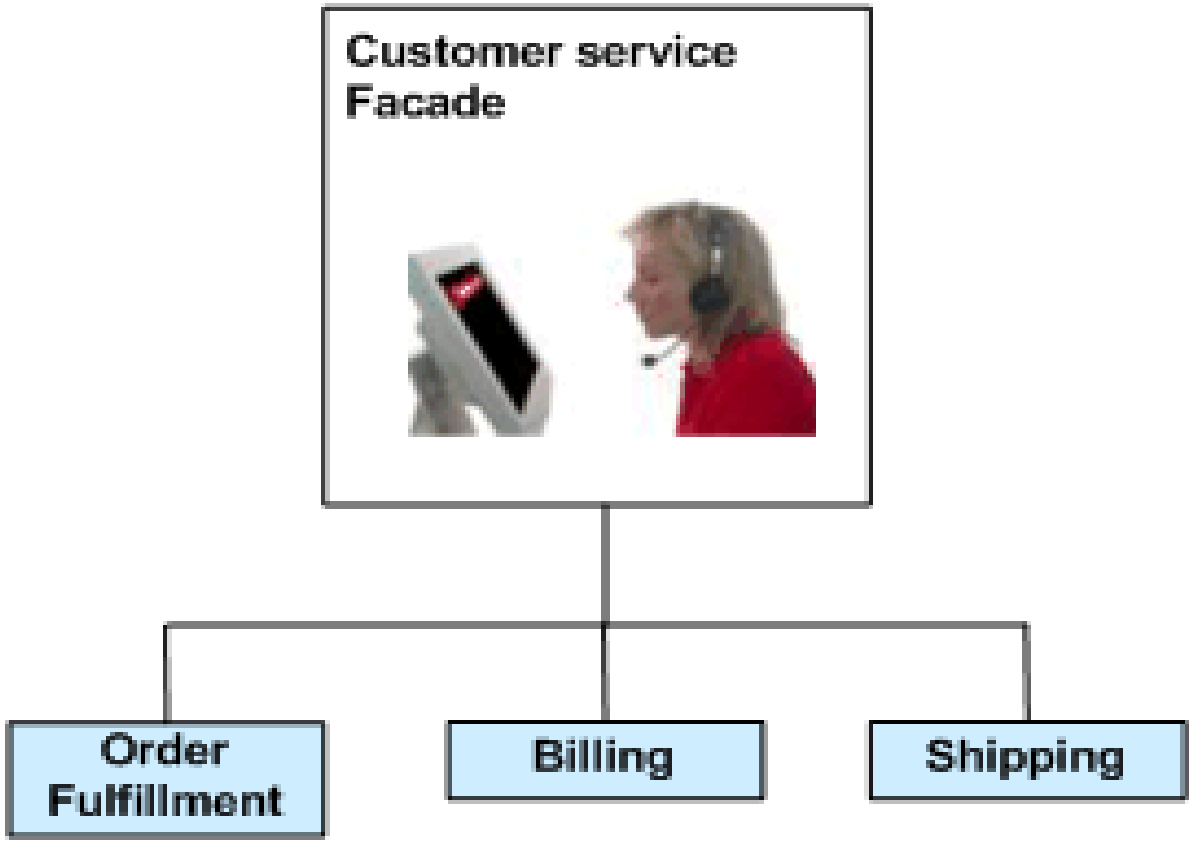
Facade


Class6
Class7
Subsistema
Class1
Class3
Class5
Class4
Class2


Class6
Class7
Facade
Subsistema
Class1
Class3
Class5
Class4
Class2

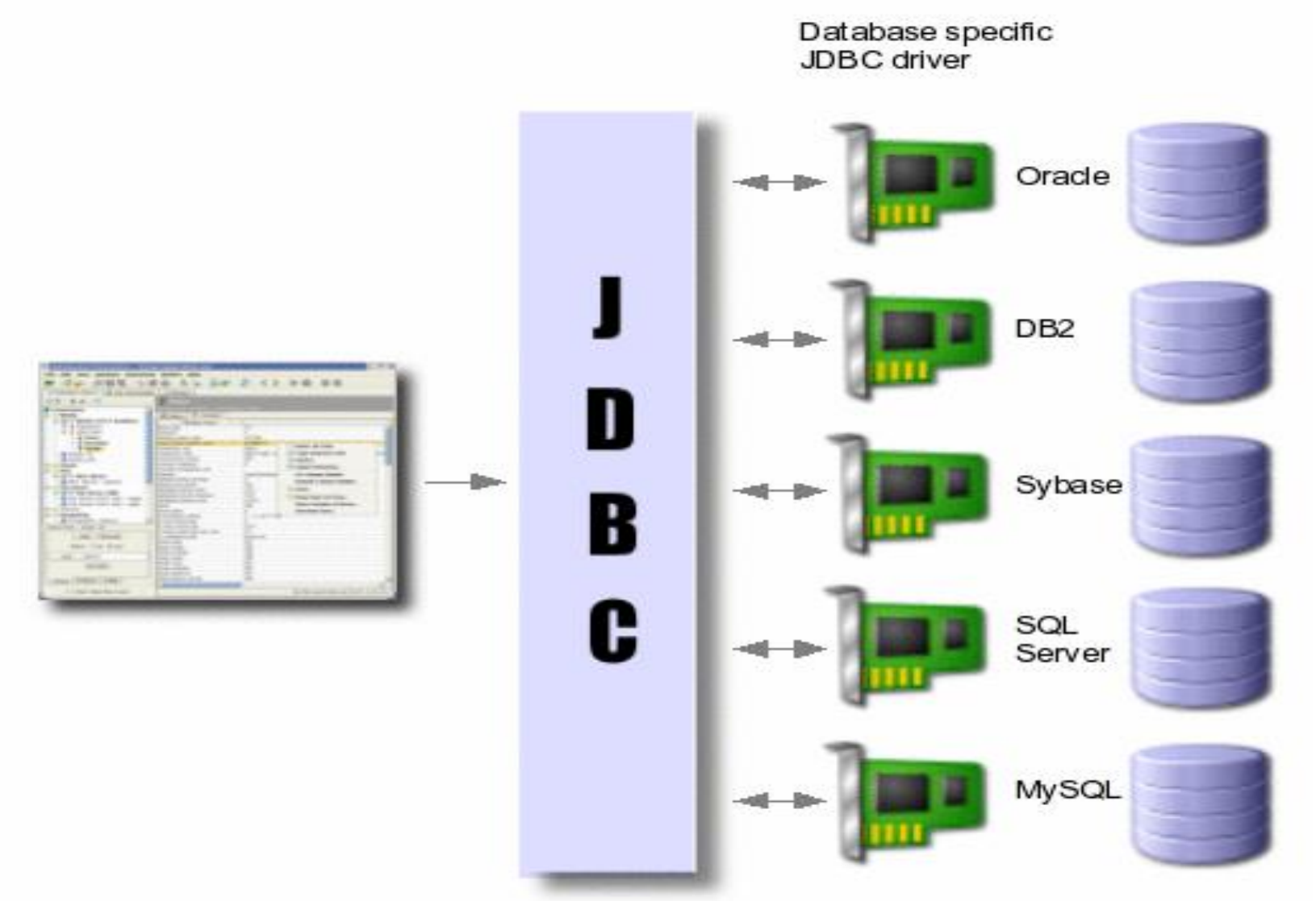
Facade

- Analogia: Sistema de atendimento ao usuário



Facade

- Exemplo: JDBC (Java DataBase Connectivity)

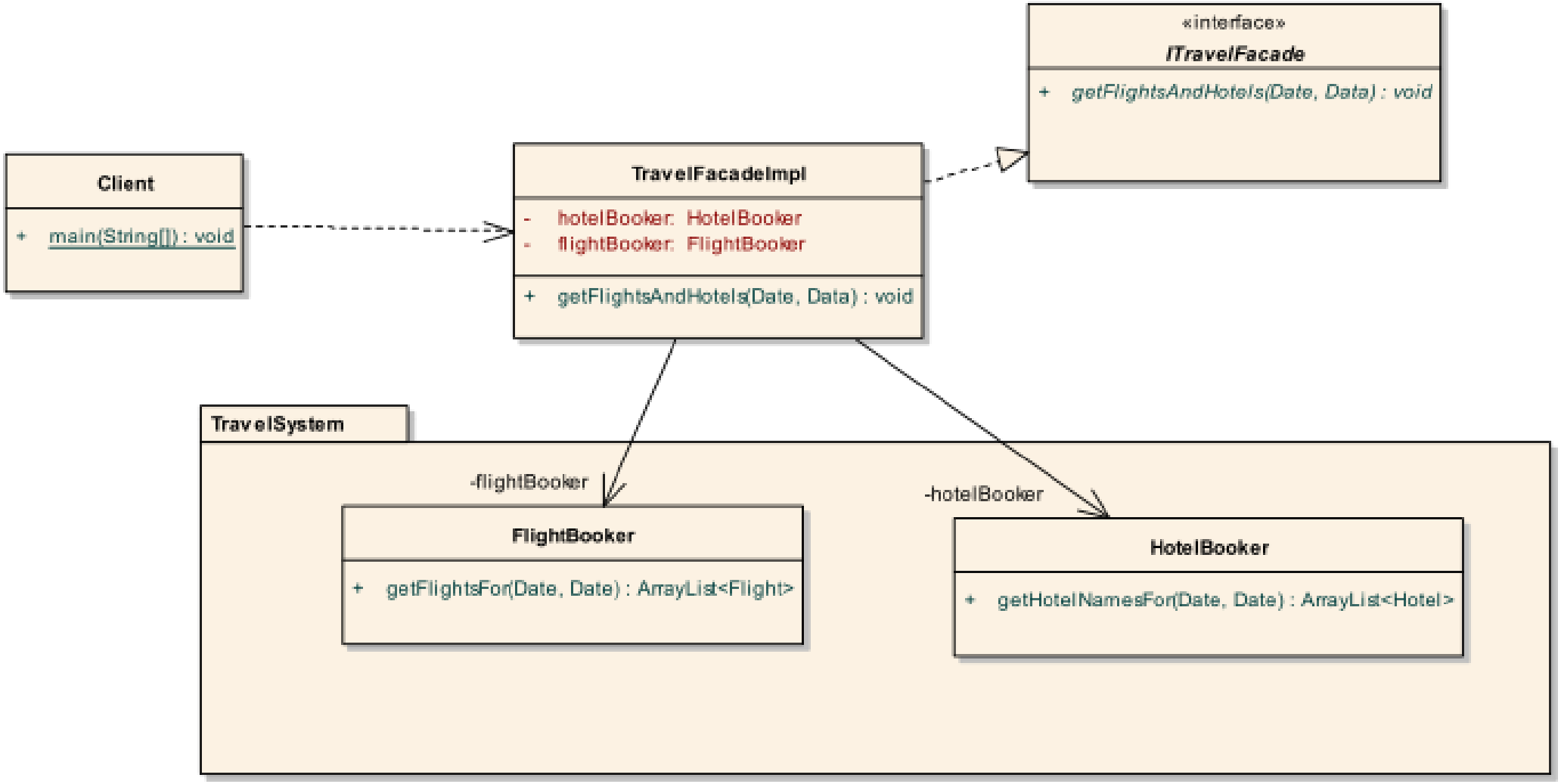


Facade

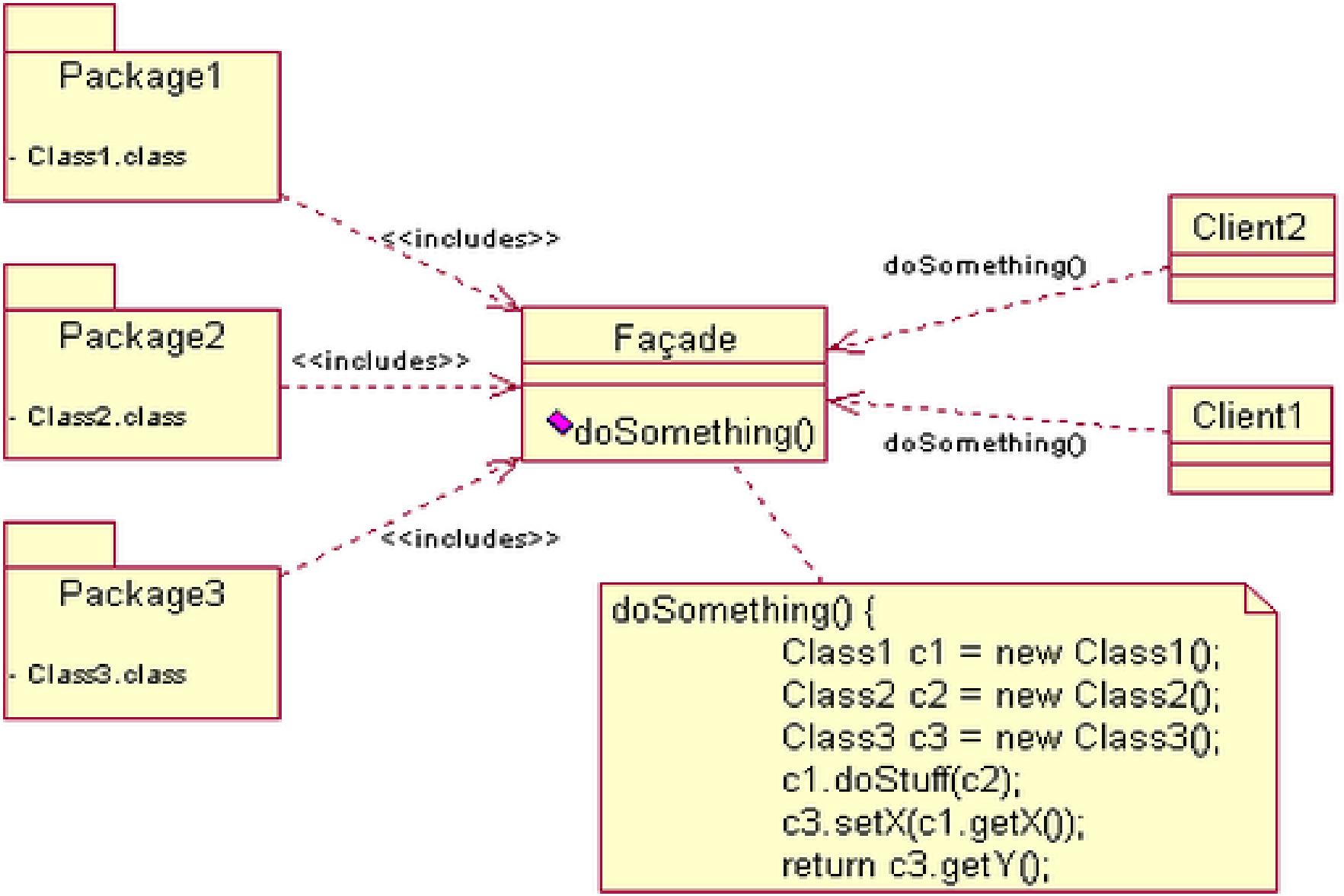
- Cuidados:
  - Se o Facade é o único ponto de acesso a um sub-sistema, ele servirá de limitador das operações, características e flexibilidade de Clientes com o sub-sistema.
  - Pode assim restringir quem precisa ser mais detalhista
  - O Facade não deve se tornar um anti-padrão como um "objeto deus", que sabe de tudo

Facade

- Recomenda-se definir o Facade como uma interface e então fornecer sua implementação concreta



Facade


Package1
- Class1.class
Package2
- Class2.class
Package3
- Class3.class
<<includes>>
<<includes>>
<<includes>>
Façade
doSomething()
Client2
doSomething()
Client1
doSomething()
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();

Facade

- Passos
  - Identificar uma interface unificadora e mais simples para o sistema.
  - Projetar uma classe ‘wrapper’ que implementa essa interface, encapsulando o sub-sistema.
  - O wrapper Facade resume em um lugar a complexidade e colaborações do componente, delegando tarefas pros métodos apropriados.
  - O Cliente está "acoplado" apenas à Facade.
  - Considere se a organização pode ser melhorada com outros Facade.
  - Similar ao Abstract Factory (esconde especificidades concretas). Usualmente implementado como Singleton

Pergunta

- **Pergunta**: Então a diferença entre o Adapter e o Façade é que o Adapter "embrulha" uma classe e o Façade pode representar várias classes?

- **Resposta**: Não! Lembre-se, o **Adapter** muda a interface de uma ou mais classes pra uma interface que o Cliente espera. Enquanto a maioria dos exemplos em livros mostra o Adapter adaptando uma classe só, é possível querer adaptar várias classes pra prover uma interface que o Cliente saiba usar.

- Da mesma forma, um **Façade** pode prover uma interface simplificada para uma única classe com uma interface muito complexa.

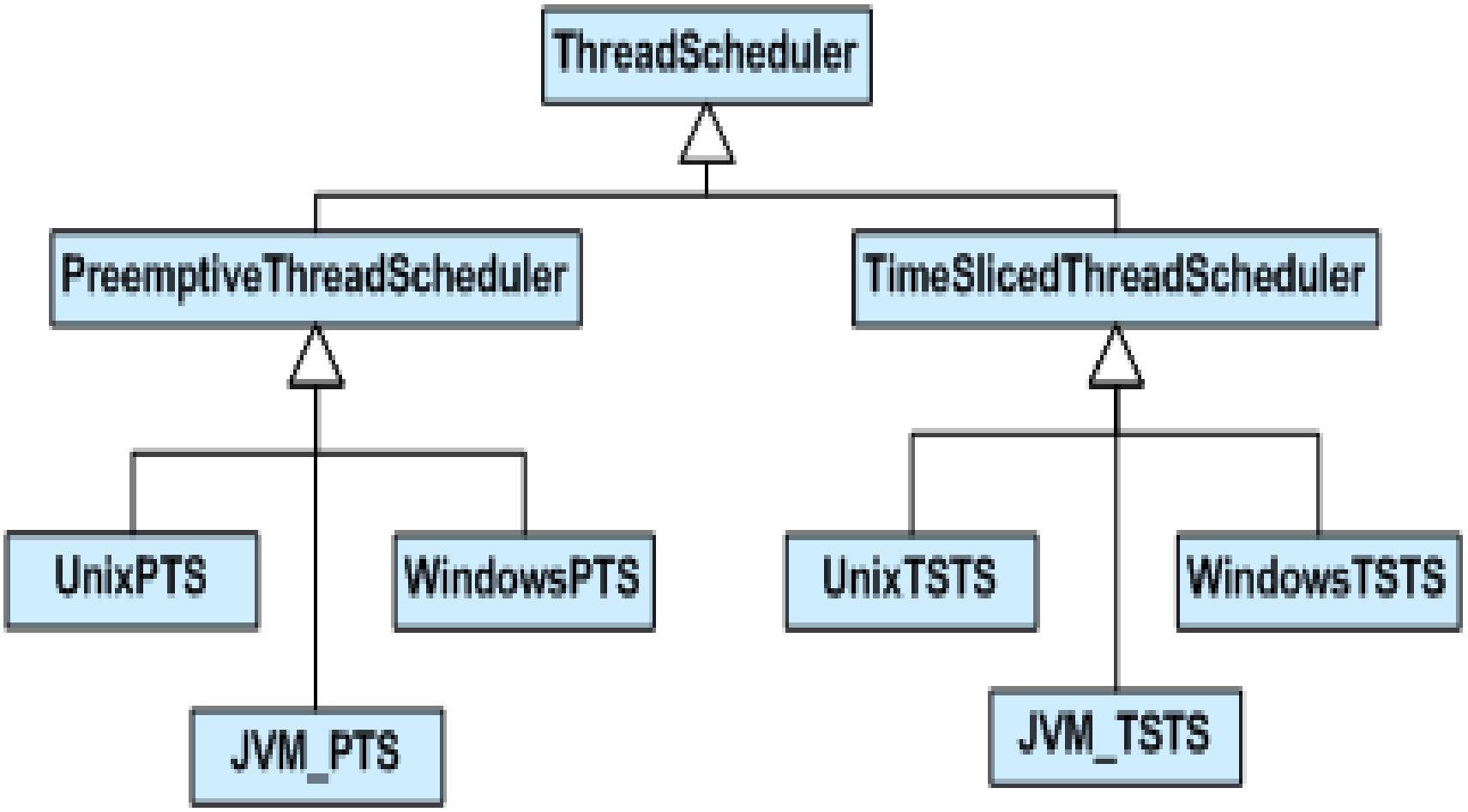
- A diferença entre os dois não está no número de classes, mas na **intenção**.

Bridge

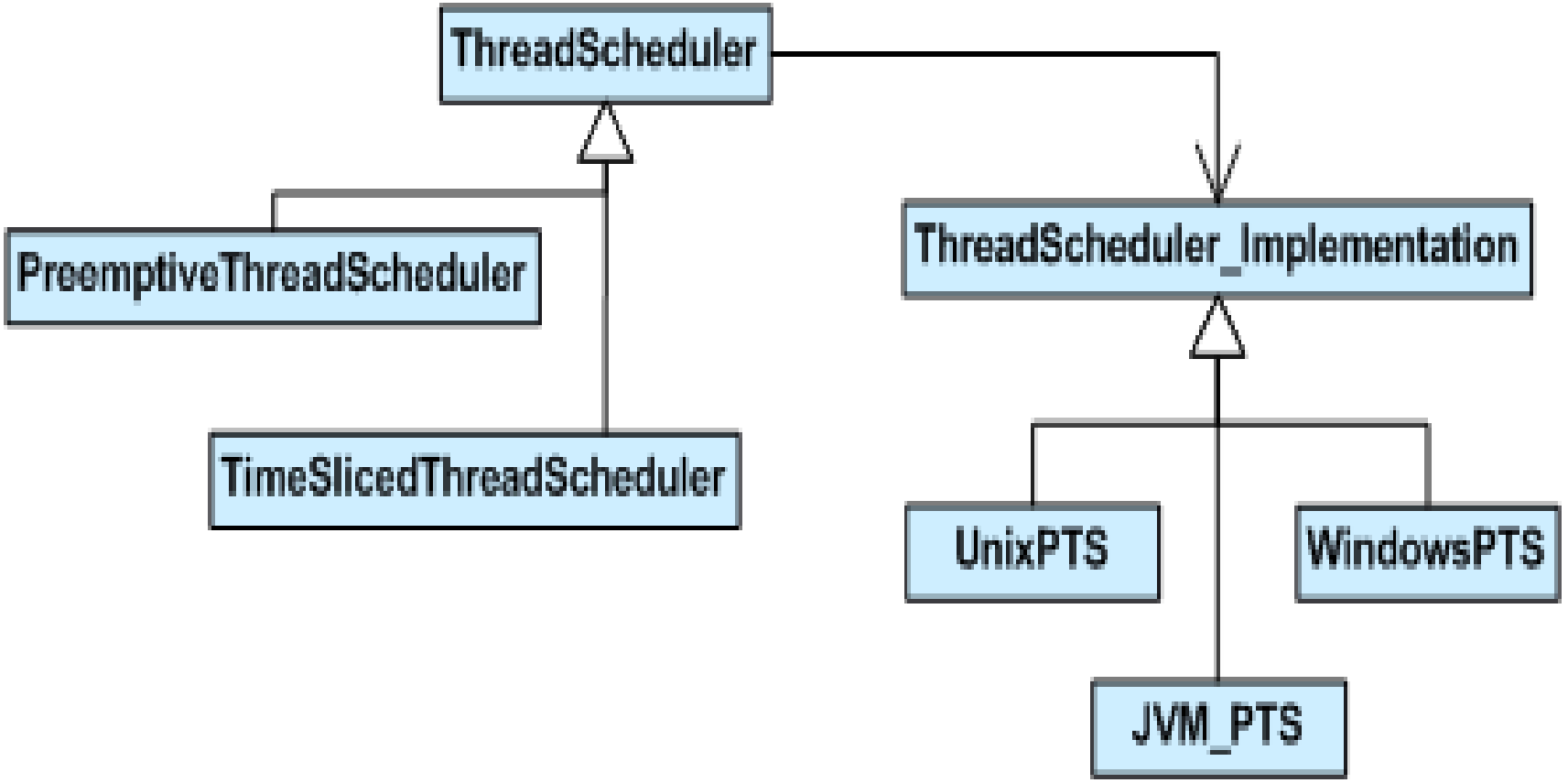
Bridge

- Objetivo
  - Desacoplar a **abstração** da **implementação** para os dois variarem independentemente
  - Cada um pode ser estendido de forma independente
- Motivação
  - Quando é possível a presença de **mais de uma implementação** para uma determinada abstração
- Aplicação
  - Evitar uma ligação forte entre a abstração e a implementação
  - Permitir que uma implementação seja escolhida em tempo de execução

Bridge


ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
UnixPTS
WindowsPTS
JVM_PTS
UnixTSTS
WindowsTSTS
JVM_TSTS

Bridge


ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
ThreadScheduler_Implementation
UnixPTS
WindowsPTS
JVM_PTS

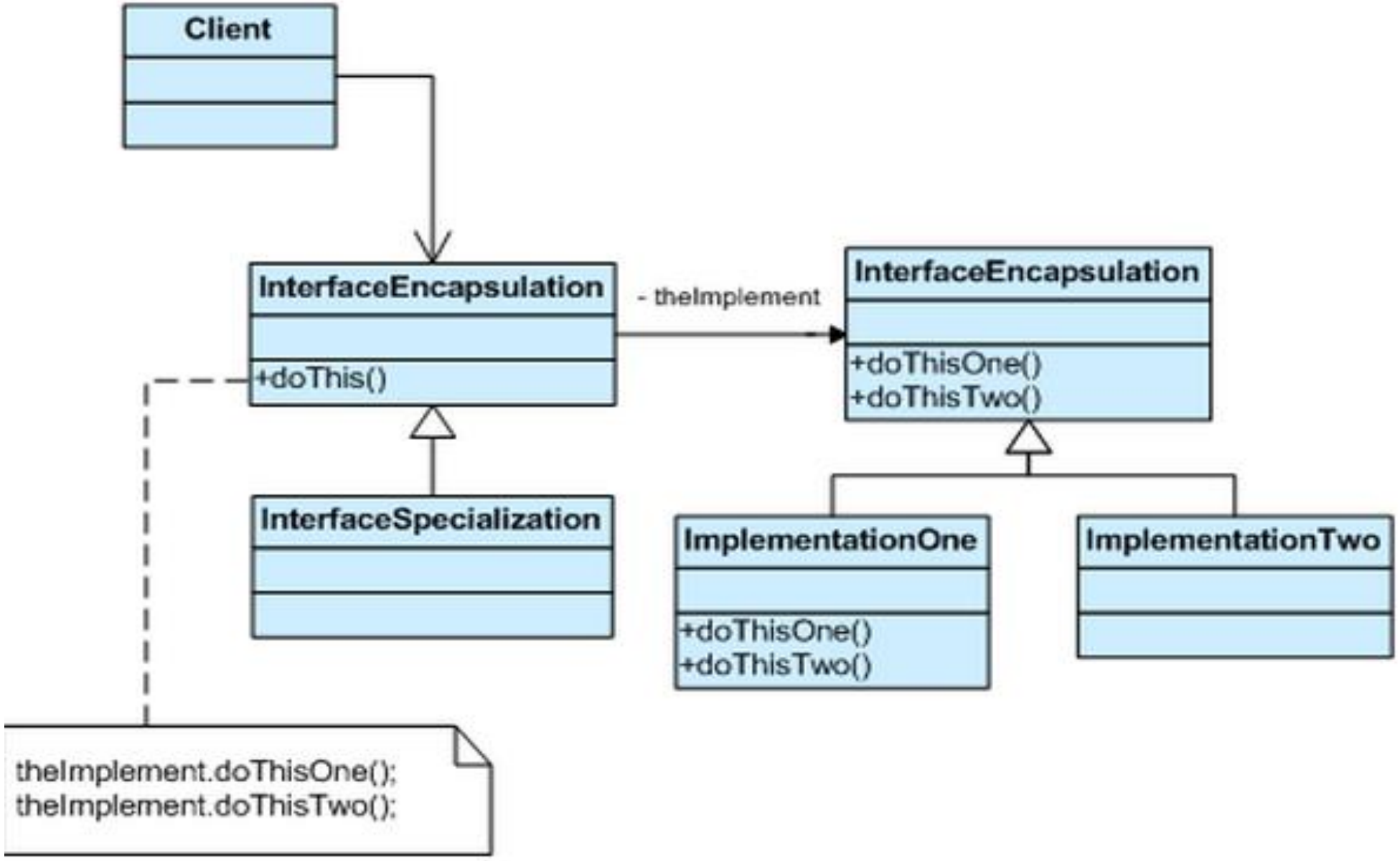
Bridge

- Decompõe os elementos da interface e os da implementação do componente em hierarquias de classe independentes ("ortogonais").
- A classe de interface tem um "ponteiro" (uma referência) pra classe abstrata de implementação.
- Esse "ponteiro" é inicializado com uma instância de uma classe de implementação concreta, mas todas as interações subsequentes entre a classe de interface e a de implementação são baseadas na abstração oferecida pela superclasse da implementação (ver figura anterior).
- O cliente interage com a interface, e esta delega as chamadas pra classe de implementação.

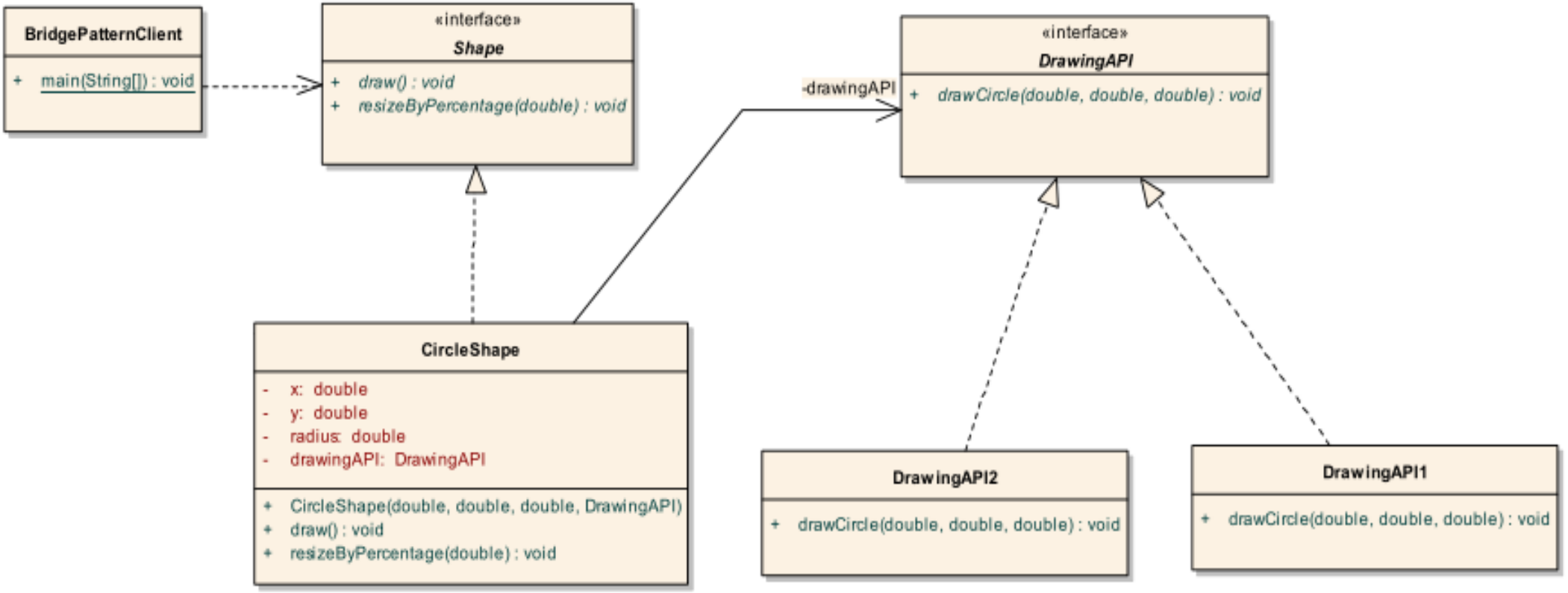
Bridge

- Assim a interface permanece enquanto o corpo da implementação está encapsulado e pode evoluir, mudar, ser trocado, etc.
- Implementação pode até ser escolhida em tempo real (é possível introduzir compatibilidade)
- Modelo maçaneta/mecanismo

Bridge


Client
InterfaceEncapsulation
+doThis()
- theImplement
InterfaceEncapsulation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
theImplement.doThisOne();
theImplement.doThisTwo();

Bridge


BridgePatternClient
+ main(String[]) : void
«interface»
Shape
+ draw() : void
+ resizeByPercentage(double) : void
-drawingAPI
«interface»
DrawingAPI
+ drawCircle(double, double, double) : void
CircleShape
- x: double
- y: double
- radius: double
- drawingAPI: DrawingAPI
+ CircleShape(double, double, double, DrawingAPI)
+ draw() : void
+ resizeByPercentage(double) : void
DrawingAPI2
+ drawCircle(double, double, double) : void
DrawingAPI1
+ drawCircle(double, double, double) : void

Bridge

- Passos
  - Decida se existem duas dimensões ortogonais no domínio (podem ser divididas). Pode ser abstração/plataforma, domínio/estrutura ou interface/implementação.
  - Projete a separação do que o Cliente quer e o que as plataformas oferecem.
  - Projete uma interface mínima, necessária, e suficiente, pra desacoplar a abstração e implementação.
  - Defina uma subclasse dessa interface pra cada implementação.
  - Crie a **superclasse** da interface que **"tem um"** objeto de **implementação** e delega funcionalidades pra ele (ver figura).
  - Defina especializações (subclasses) da interface, se necessário.

Resumo

- O **Adapter** faz as coisas funcionarem depois de **serem** projetadas; O **Bridge** faz as coisas funcionarem **antes** de serem projetadas. O Bridge é projetado antes para separar classes, pra que independentemente. O Adapter é "retroprojetado" depois para juntar classes não relacionadas.
- O Adapter muda a interface, o Decorator aprimora as responsabilidades do objeto.
- Facade define uma nova interface, enquanto o Adapter reusa uma antiga interface.