Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

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;
- Adapter;
- Bridge;
- Composite;
- Decorator;
  - Interceptors / AOP;
- Façade;
  - DAO;
- Flyweight;
- Proxy;
- Conclusões.

Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Introdução

Estrutura de objetos

- Padrões de estrutura: como classes e objetos se organizam para formar estruturas maiores?
  - Padrões de classe: herança;
  - Padrões de objeto: composição:
    - Maior flexibilidade pela capacidade de mudança em tempo de execução.

Padrões de Estrutura

"Padrões de estrutura com escopo de classe usam herança para compor interfaces ou implementações. Os com escopo de objeto descrevem formas de compor objetos para realizar novas funcionalidades."

Escopo de Classe	Adapter
Escopo de Objeto	Bridge
	Composite
	Decorator
	Façade
	Flyweight
	Proxy

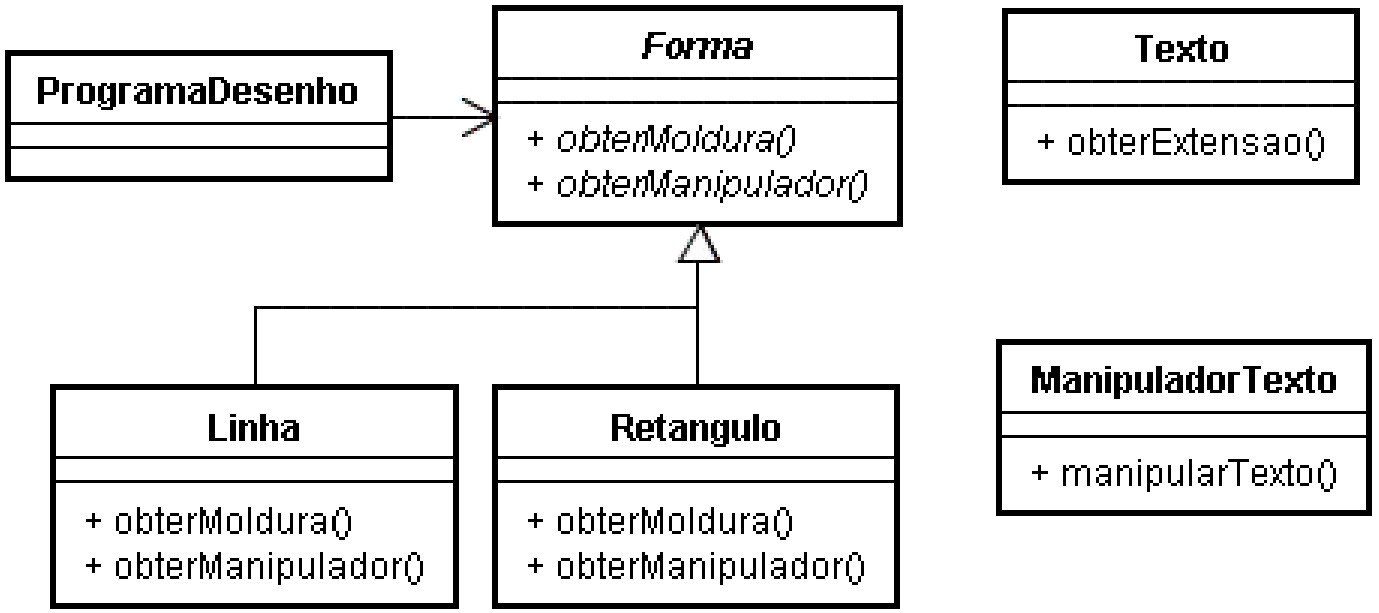
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Adapter
(Adaptador)
Estrutural / Objeto e Classe

Descrição

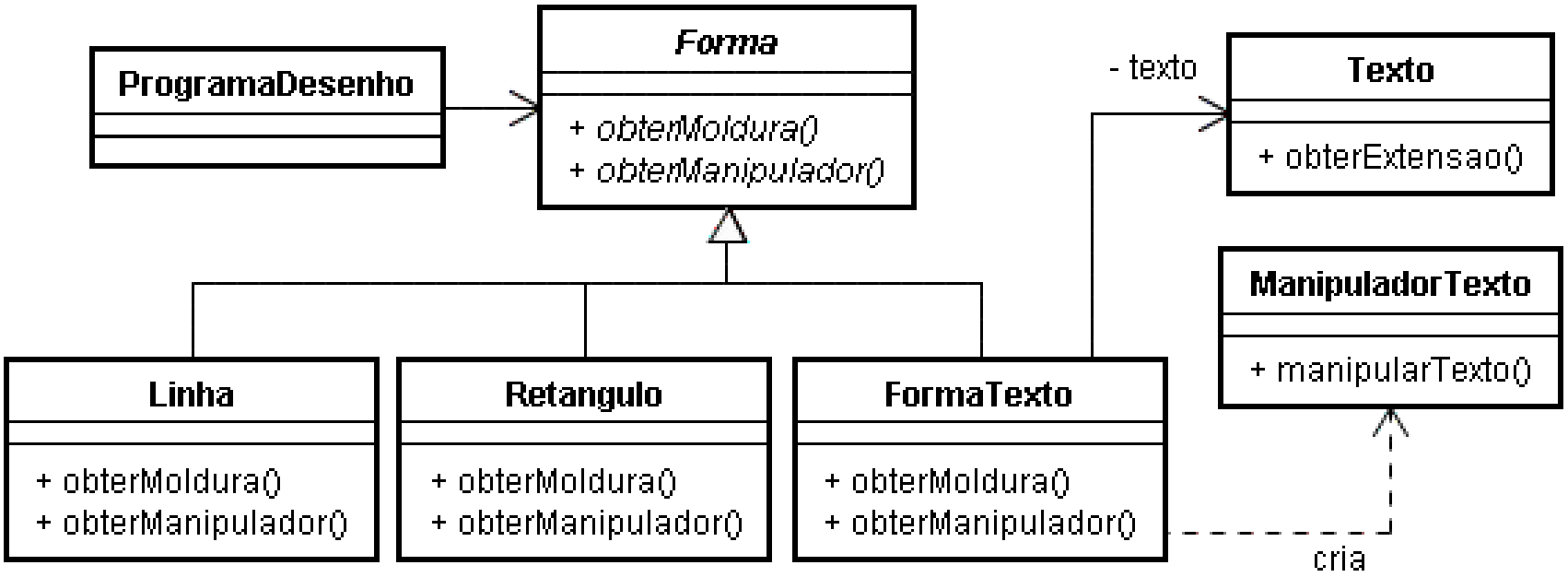
- Intenção:
  - Converter a interface de uma classe em outra interface esperada pelo cliente. Permite que classes com interfaces incompatíveis possam colaborar.
- Também conhecido como:
  - Wrapper.

O problema



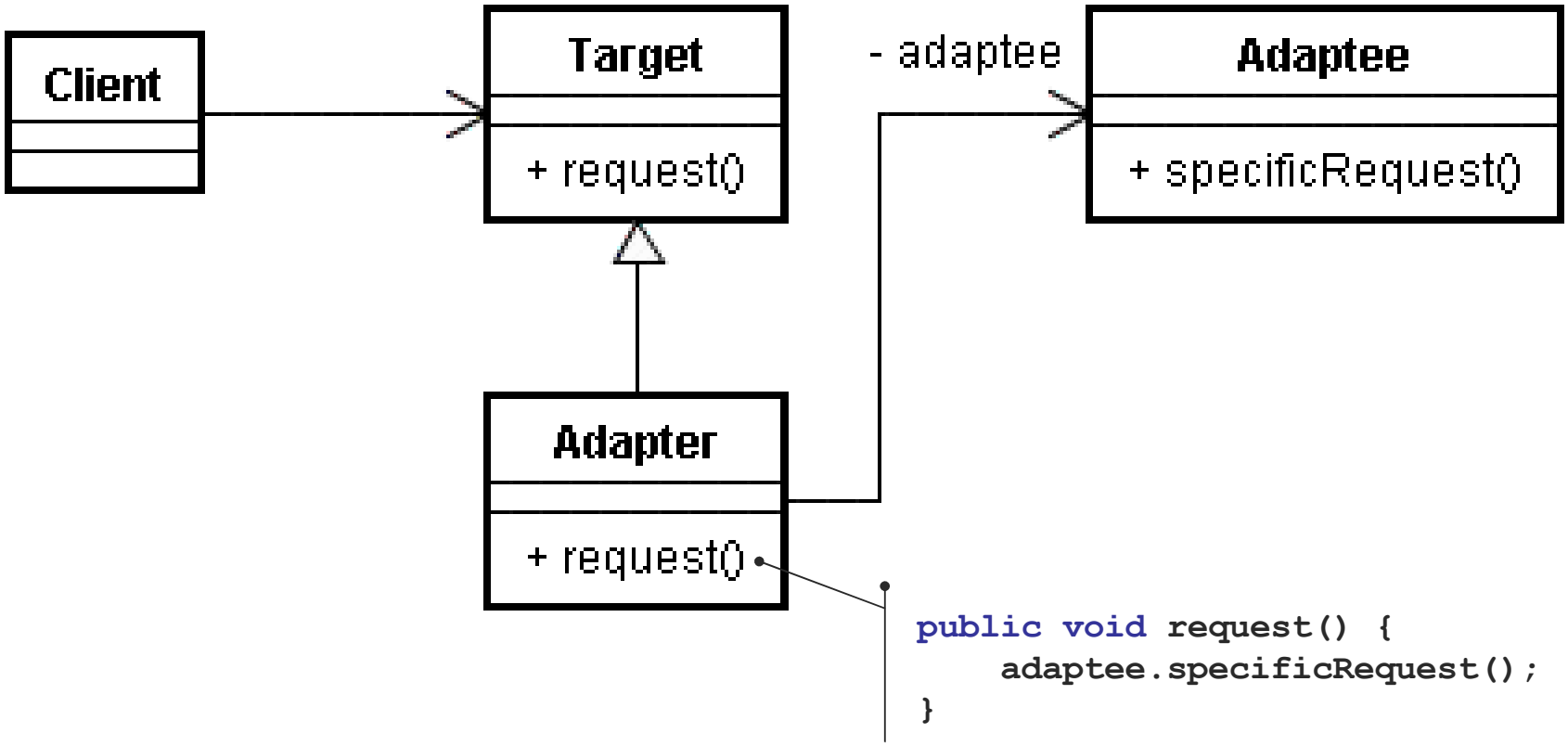
- Existe uma ferramenta gráfica de texto pronta, mas o programa de desenho só trabalha com formas.

A solução

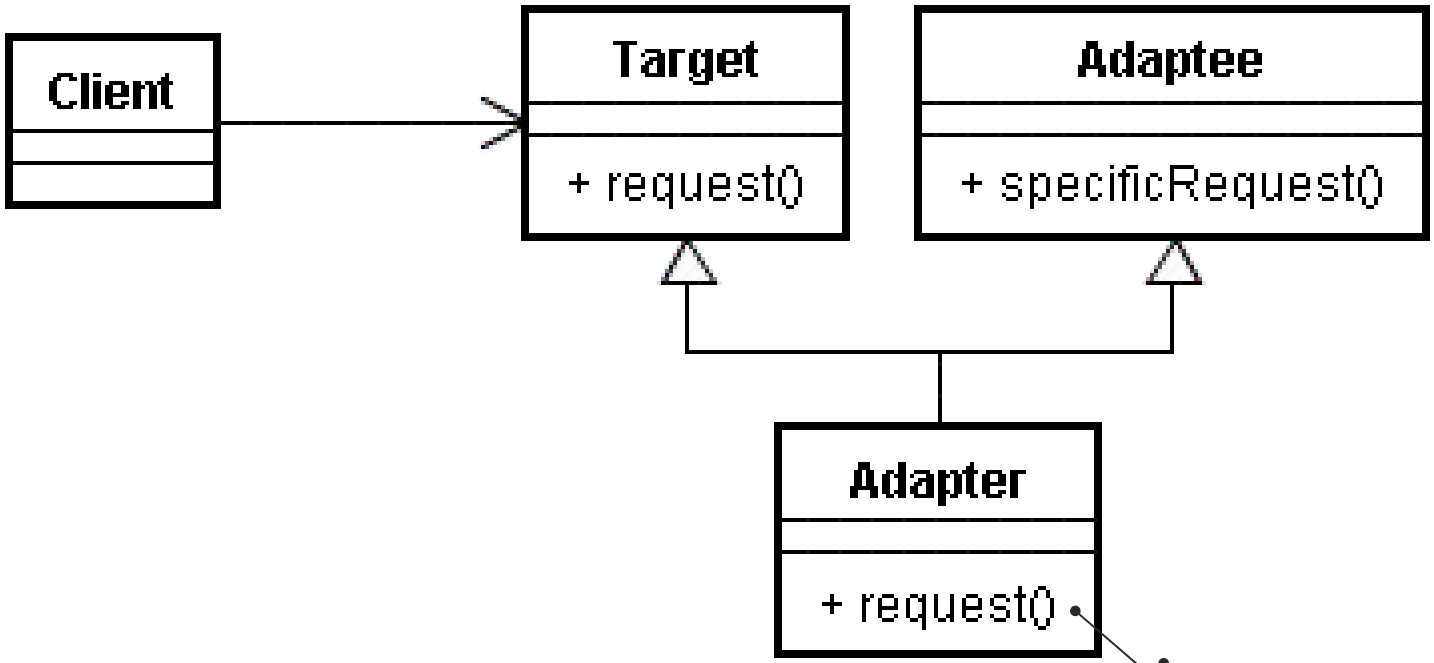


- FormaTexto adapta a classe pronta à interface esperada pelo programa de desenho.

Estrutura – Padrão de objeto

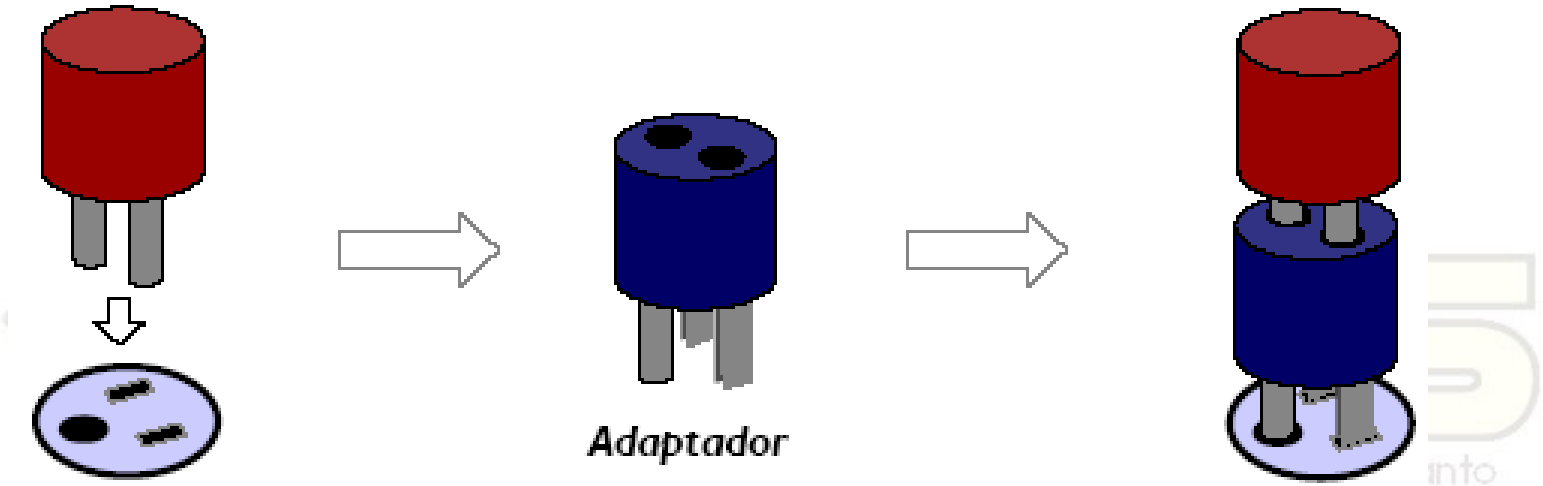


Estrutura – Padrão de classe




```
public void request() {
    specificRequest();
}
```

Analogia



Usar este padrão quando...

- você quer usar uma classe já pronta que possui uma interface diferente da que você precisa;
- você quer criar uma classe reutilizável já prevendo que a situação acima ocorrerá no futuro.

Vantagens e desvantagens

- Adapter de Classe:
  - Não funciona bem quando se quer adaptar uma hierarquia de classes;
  - Permite que o adaptador sobrescreva algumas funções do adaptado.

Vantagens e desvantagens

- Adapter de Objeto:
  - Permite o uso de um único adaptador para uma hierarquia de classes adaptadas;
  - É mais difícil sobrescrever funções do adaptado.

Exemplos em Java

- As classes envoltório (Integer, Boolean, Float, etc.);
- Adaptadores de eventos de interface gráfica (AWT/Swing):
  - java.awt.MouseAdapter adapta java.awt.MouseListener a uma interface mais simples.

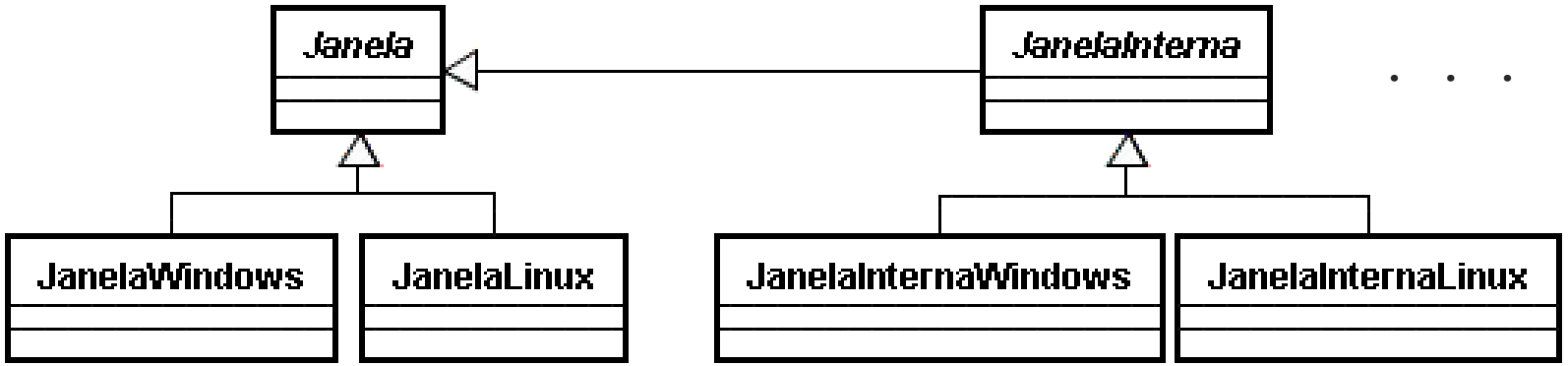
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Bridge
(Ponte)
Estrutural / Objeto

Descrição

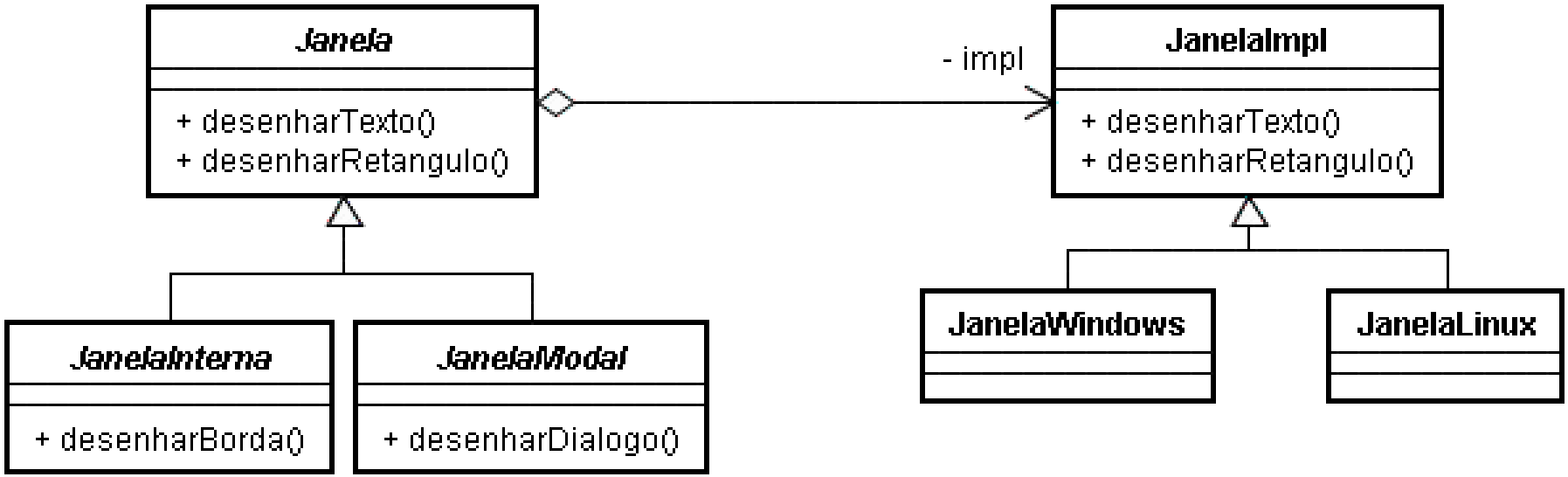
- Intenção:
  - Desacoplar uma abstração de sua implementação para que ambos possam variar independentemente.
- Também conhecido como:
  - Handle, Body.

O problema



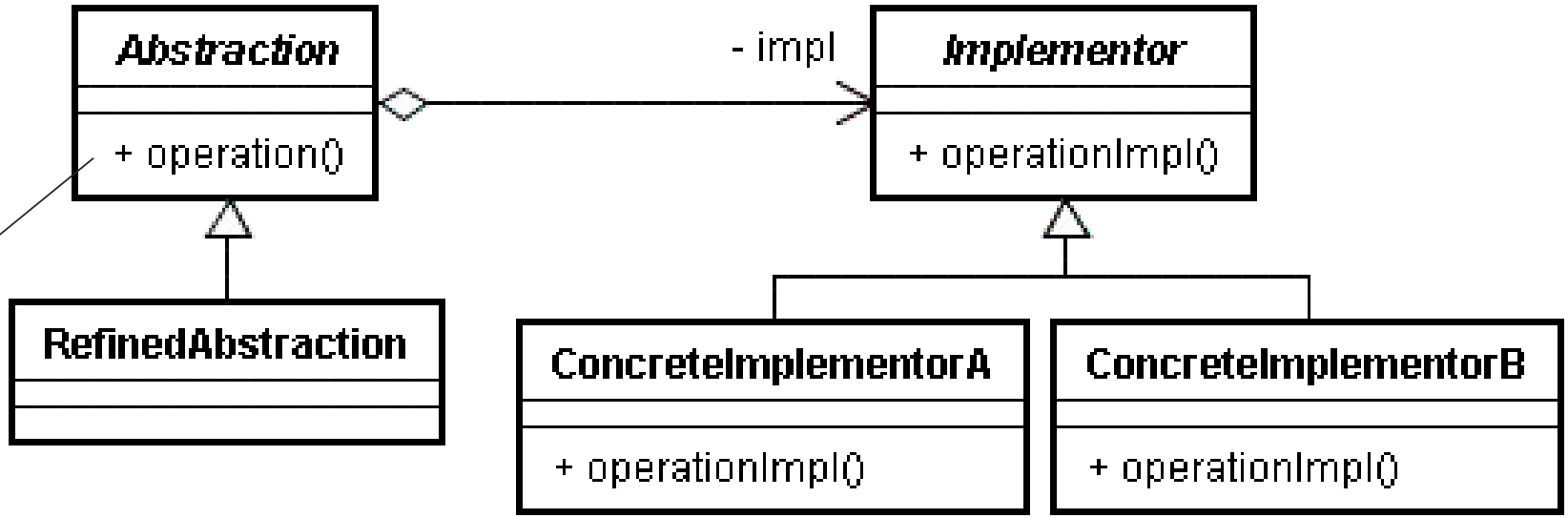
- Componentes gráficos devem ser implementados para várias arquiteturas;
- Cada novo componente exige várias implementações;
- Cada nova arquitetura mais ainda.

A solução



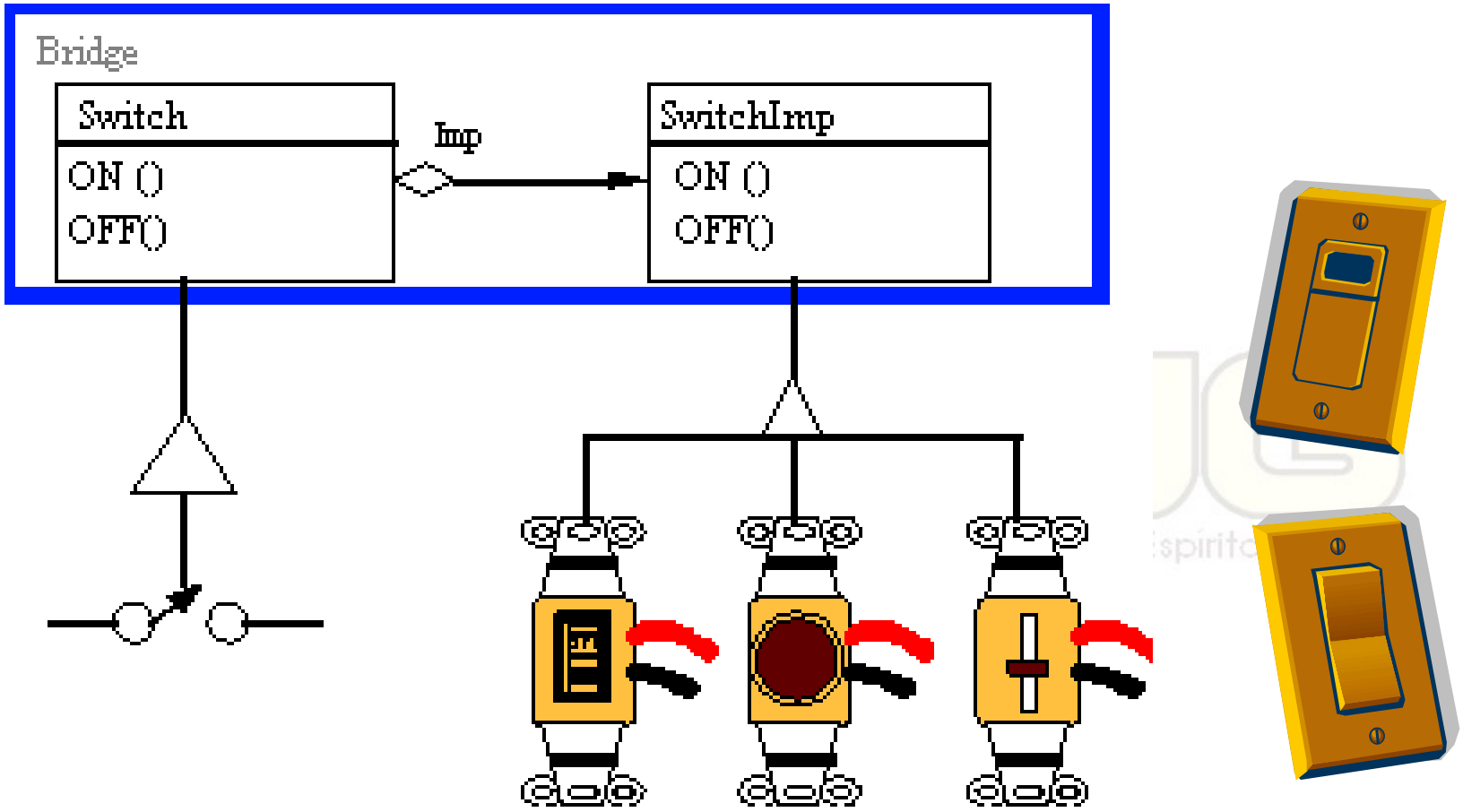
- Janela concentra métodos que utilizam recursos específicos de plataforma;
- Subclasses utilizam os métodos de Janela para implementar itens específicos.

Estrutura




```
public void operation() {
    impl.operationImpl();
}
```

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 evitar uma ligação permanente entre a abstração e a implementação;
- tanto a abstração quanto a implementação possuem subclasses;
- mudanças na implementação não devem afetar o código do cliente;
- sua atual solução gera uma proliferação de classes (exemplo).

Vantagens e desvantagens

- Desacopla a implementação:
  - Podendo até mudá-la em tempo de execução.
- Melhora a extensibilidade:
  - É possível extender a abstração e a implementação separadamente.
- Esconde detalhes de implementação:
  - Clientes não precisam saber como é implementado.

Exemplos em Java

- Drivers JDBC são bridges:
  - Connection, Statement, ResultSet, etc. são abstrações;
  - Suas implementações (driver) são escondidas do cliente.

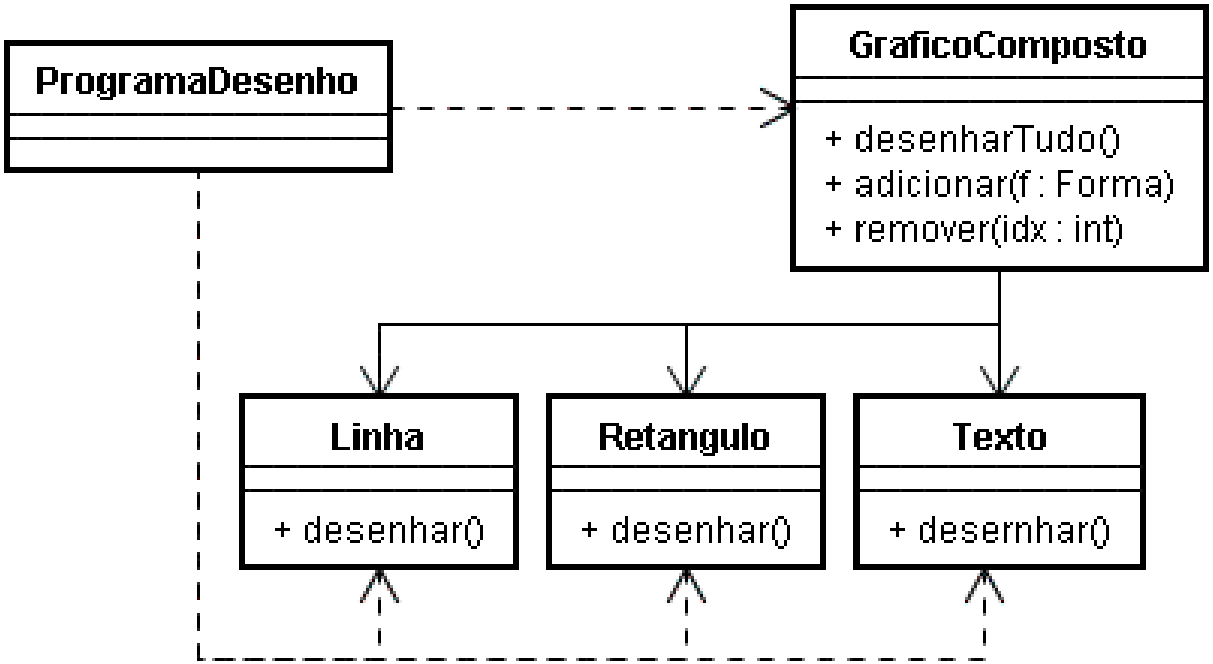
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Composite
(Composto)
Estrutural / Objeto

Descrição

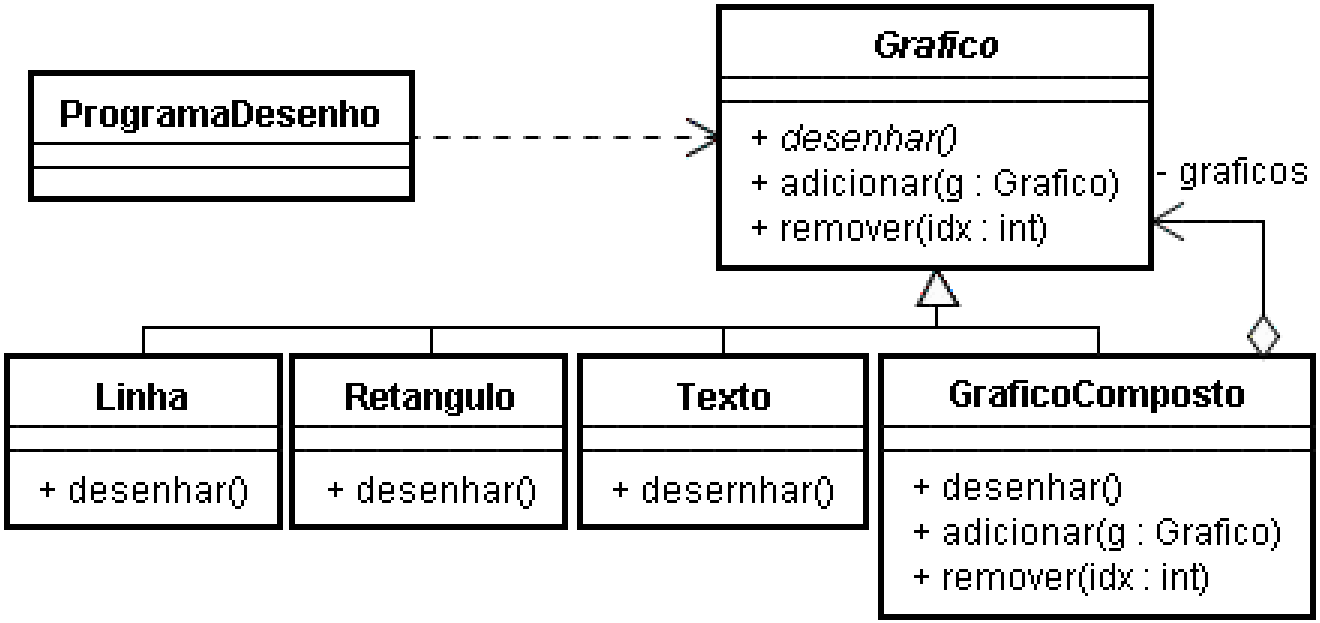
- Intenção:
  - Compor objetos em estruturas de árvore para representar hierarquias todo-parte. Permite que clientes trate objetos individuais e compostos de maneira uniforme.

O problema



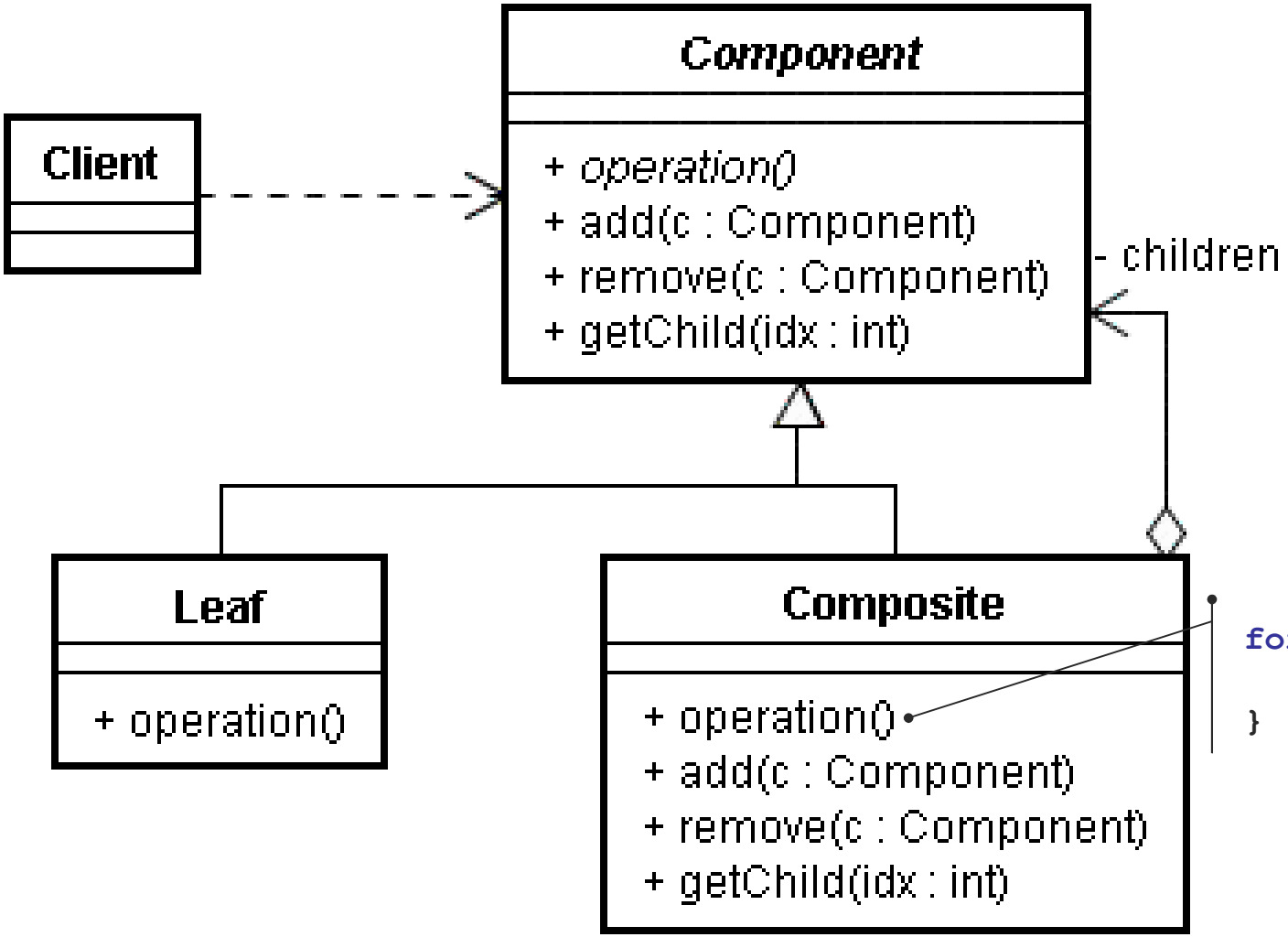
- Existem gráficos que são compostos de outros gráficos;
- O programa tem que conhecer cada um deles, o que complica o código.

A solução

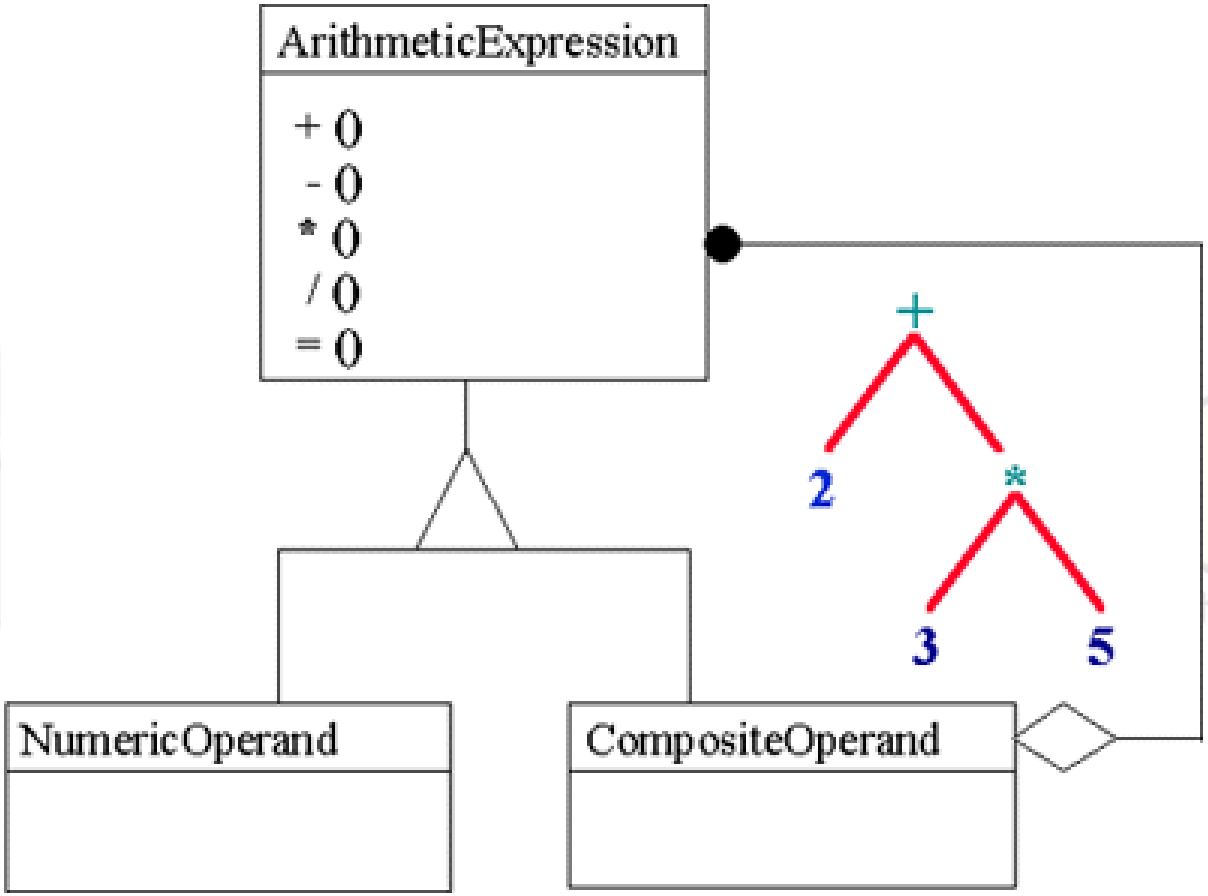


- A classe abstrata representa tanto gráficos simples quanto compostos;
- Programa só precisa conhecer Grafico.

Estrutura


Component
+ operation()
+ add(c : Component)
+ remove(c : Component)
+ getChild(idx : int)
Client
- children
Leaf
+ operation()
Composite
+ operation()
+ add(c : Component)
+ remove(c : Component)
+ getChild(idx : int)
for (c : children) {
c.operation();
}

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 representar hierarquias todo-parte;
- quiser que clientes ignorem a diferença entre objetos simples e objetos compostos.

Vantagens e desvantagens

- Define hierarquias todo-parte:
  - Objetos podem ser compostos de outros objetos e assim por diante.
- Simplifica o cliente:
  - Clientes não se preocupam se estão lidando com compostos ou individuais.
- Facilita a criação de novos membros:
  - Basta estar em conformidade com a interface comum a todos os componentes.
- Pode tornar o projeto muito genérico:
  - Qualquer componente pode ser criado, não há como usar checagem de tipos para restringir.

Transparência x segurança

- Composite viola o princípio de herança:
  - Leaf IS-A Component é falso, pois add(), remove(), etc. não fazem sentido para Leaf.
  - Mais transparência (tratamento uniforme);
  - Menos segurança (checagem de tipos).
- Paliativos (opcionais):
  - Defina Component como classe abstrata e seus métodos com implementação vazia;
  - Defina getComposite() para retornar a si mesmo se for composto e null caso contrário.

Outras preocupações

- Direção na navegação:
  - Filhos -> pais? Pais -> filhos?
- Ciclos:
  - Recursão não pode gerar loops infinitos.
- Multiplicidade:
  - Um filho pode ter múltiplos pais?
- Limpeza:
  - Deleção em cascata?

Exemplos em Java

- A implementação DOM (org.w3c.dom):
  - Componente: Node;
  - Um node pode ser o documento, um elemento (tag), um atributo, uma porção de texto, etc.;
  - Alguns nodes podem conter outros.

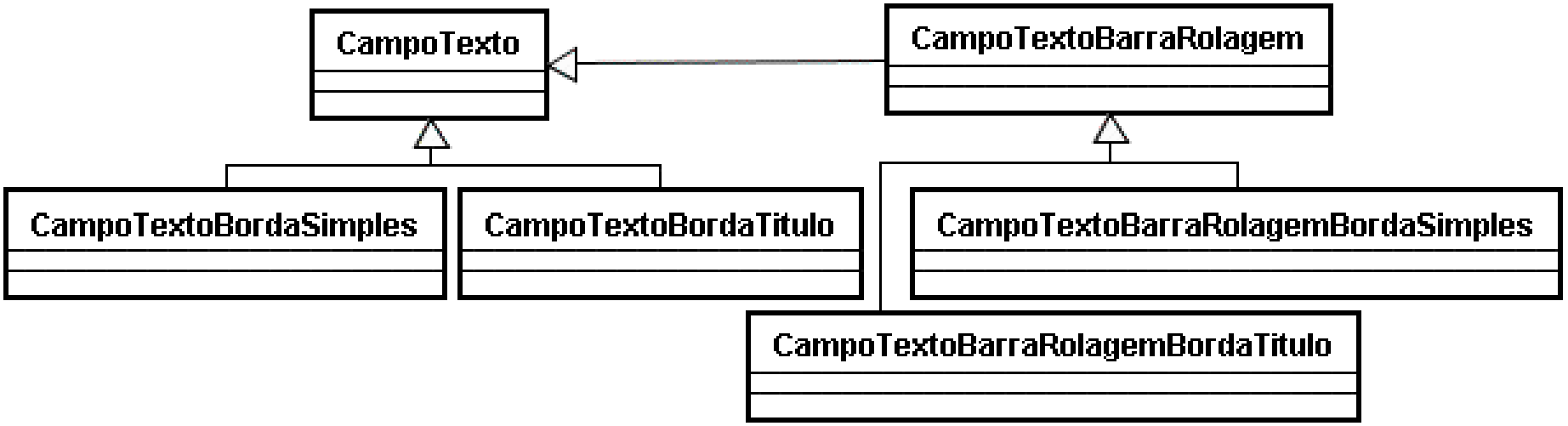
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Decorator
(Decorador)
Estrutural / Objeto

Descrição

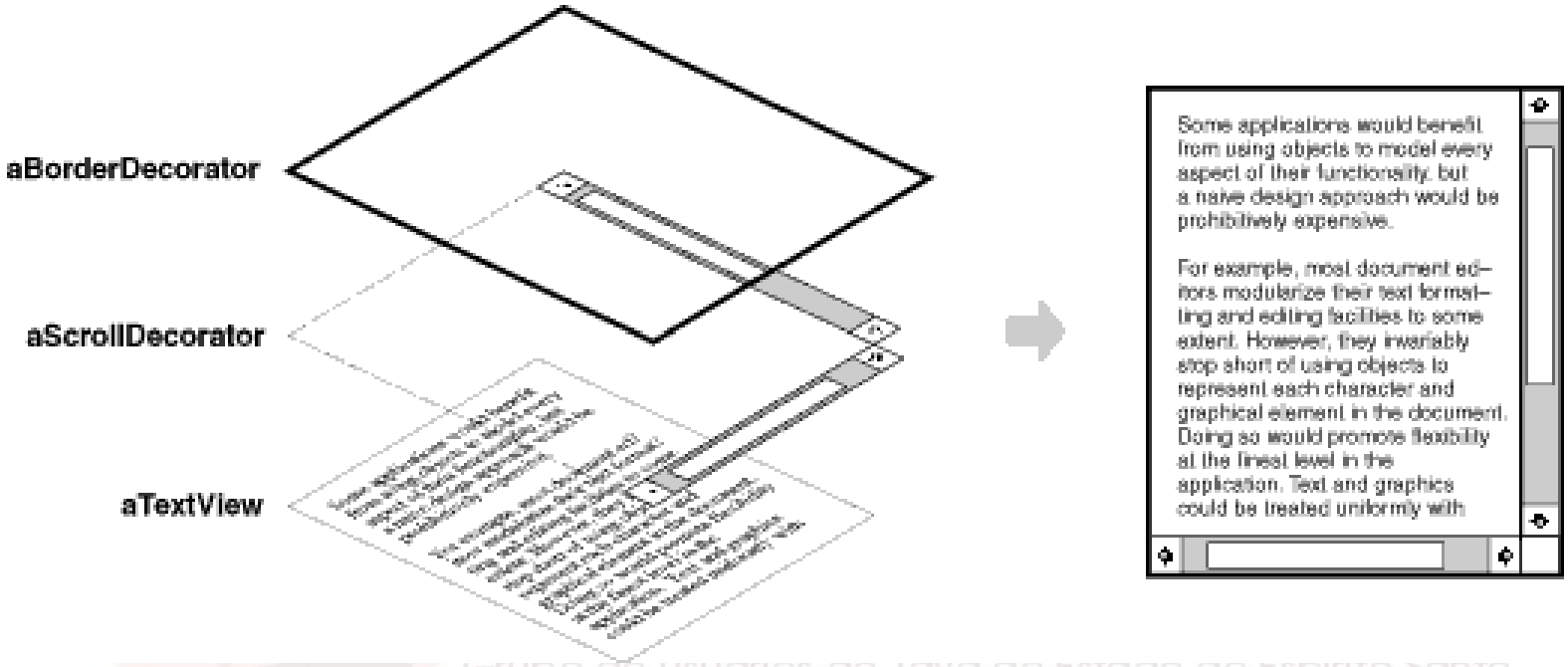
- Intenção:
  - Anexar funcionalidades adicionais a um objeto dinamicamente. Provê uma alternativa flexível à herança como mecanismo de extensão.
- Também conhecido como:
  - Wrapper

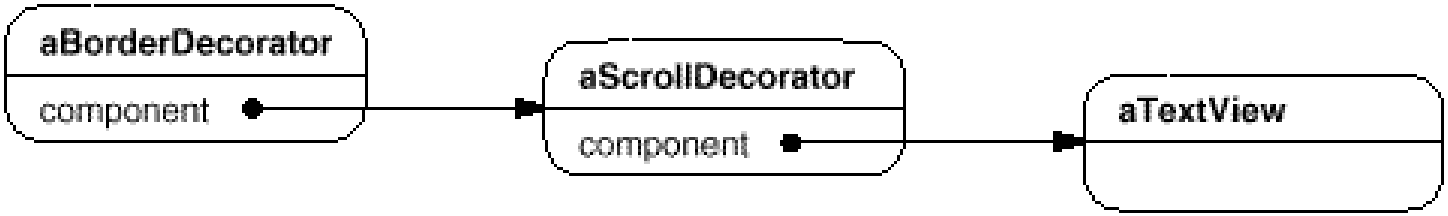
O problema



- Adicionar funcionalidade por herança é inflexível e prolifera classes.

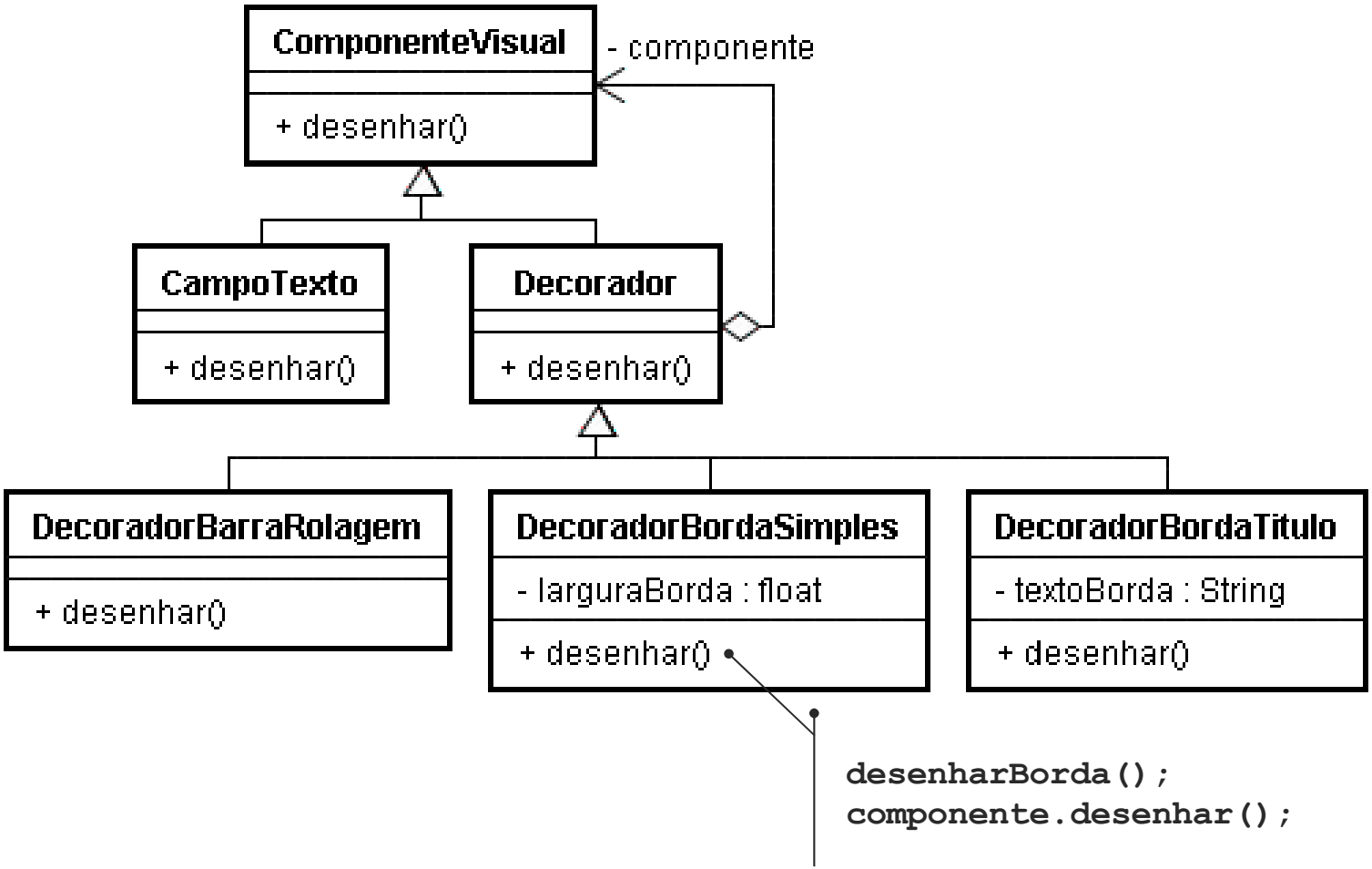
A solução



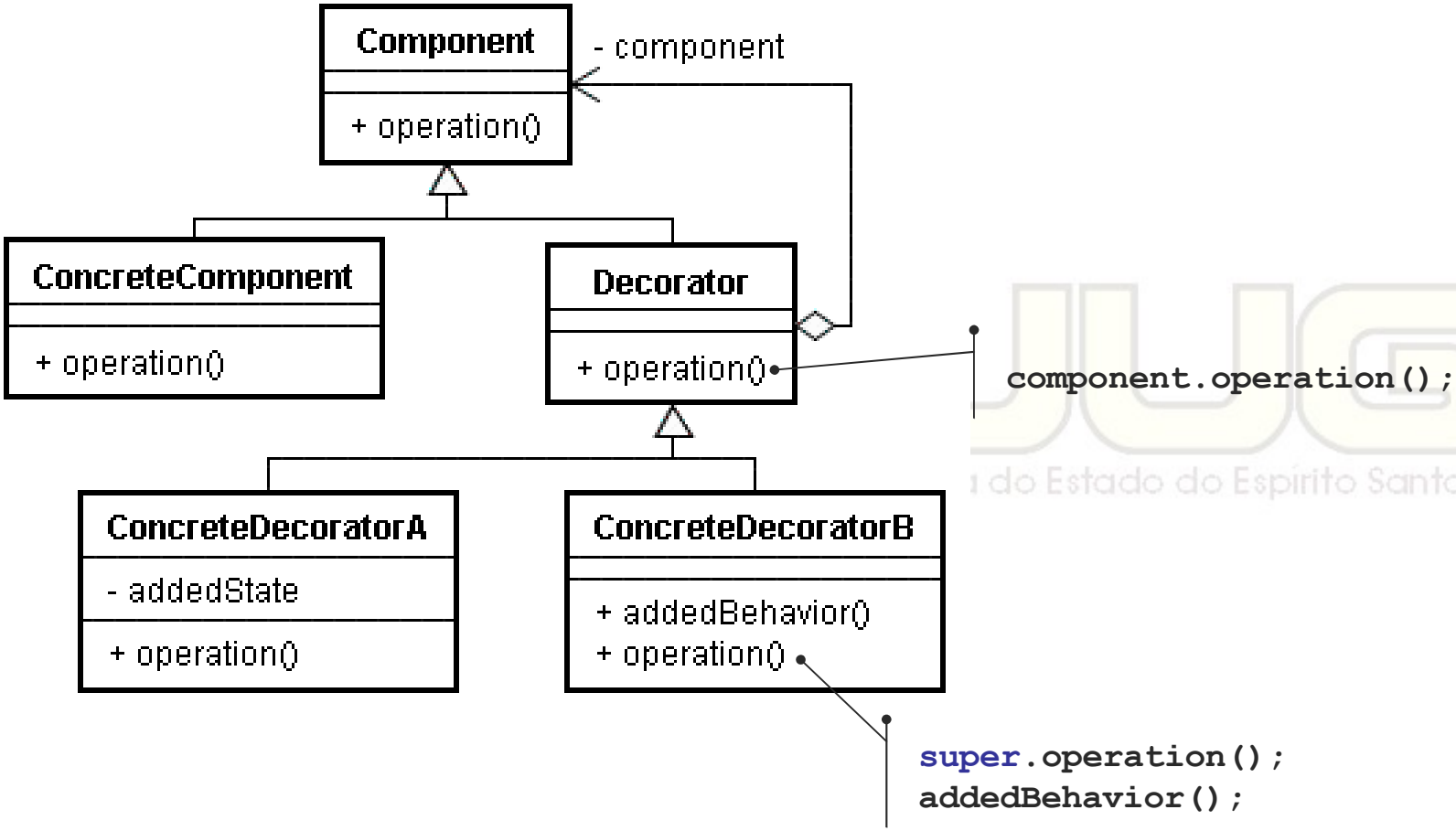


- Componentes adicionados por cima de outros, decorando-os.

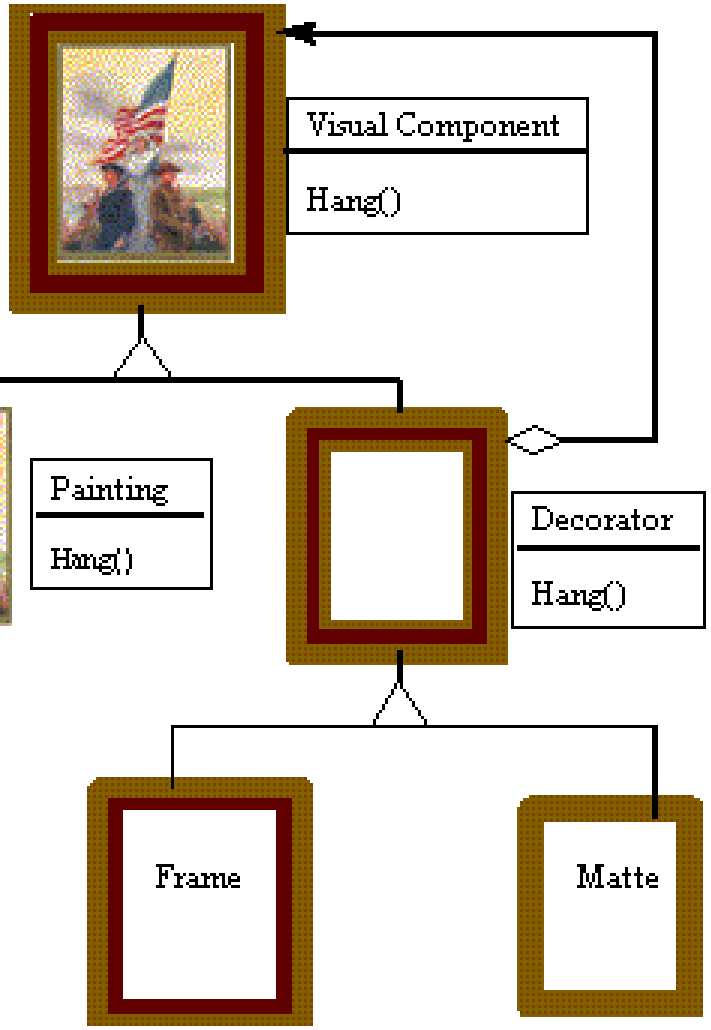
A Solução


ComponenteVisual
+ desenhar()
- componente
CampoTexto
+ desenhar()
Decorador
+ desenhar()
DecoradorBarraRolagem
+ desenhar()
DecoradorBordaSimples
- larguraBorda : float
+ desenhar()
DecoradorBordaTitulo
- textoBorda : String
+ desenhar()
desenharBorda();
componente.desenhar();

Estrutura


Component
+ operation()
- component
ConcreteComponent
+ operation()
Decorator
+ operation()
component.operation();
ConcreteDecoratorA
- addedState
+ operation()
ConcreteDecoratorB
+ addedBehavior()
+ operation()
super.operation();
addedBehavior();

Analogia



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

Usar este padrão quando...

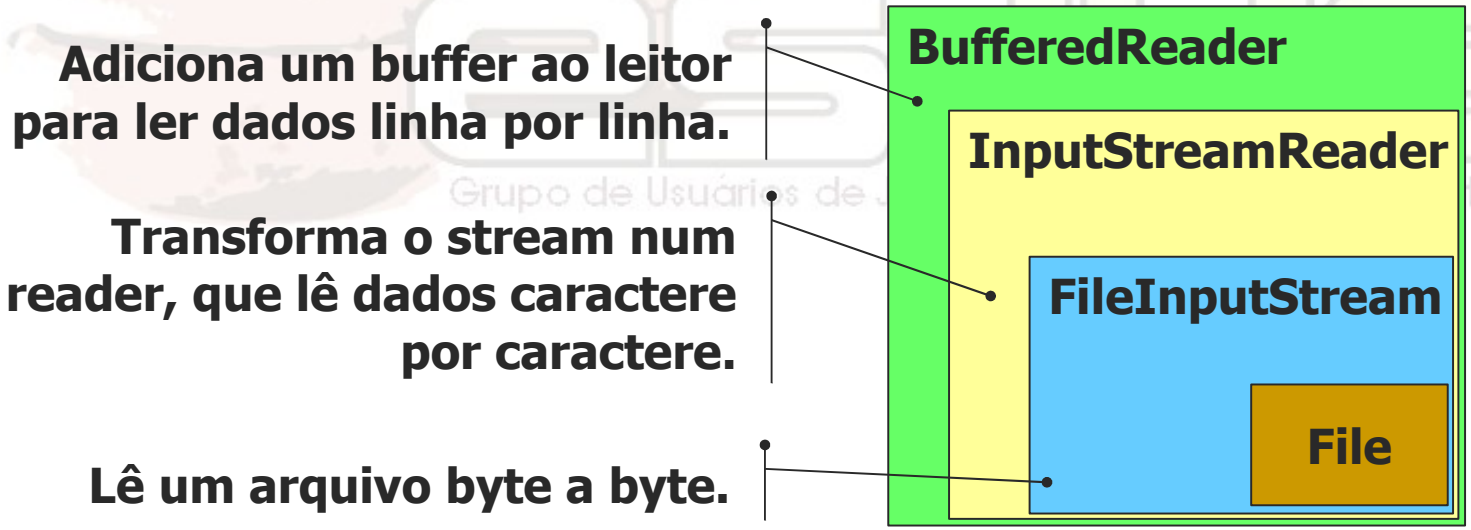
- quiser adicionar funcionalidade dinamicamente e transparentemente;
- quiser adicionar funcionalidade que pode depois ser desativada;
- extensão por herança é impraticável (não disponível ou produziria uma explosão de subclasses, como no exemplo).

Vantagens e desvantagens

- Mais flexibilidade do que herança:
  - Podem ser adicionadas/removidas em tempo de execução;
  - Pode adicionar duas vezes a mesma funcionalidade.
- O decorador é diferente do componente:
  - A identidade do objeto não pode ser usada de forma confiável.
- Muitos objetos pequenos:
  - Um projeto que utiliza Decorator pode vir a ter muitos objetos pequenos e parecidos.

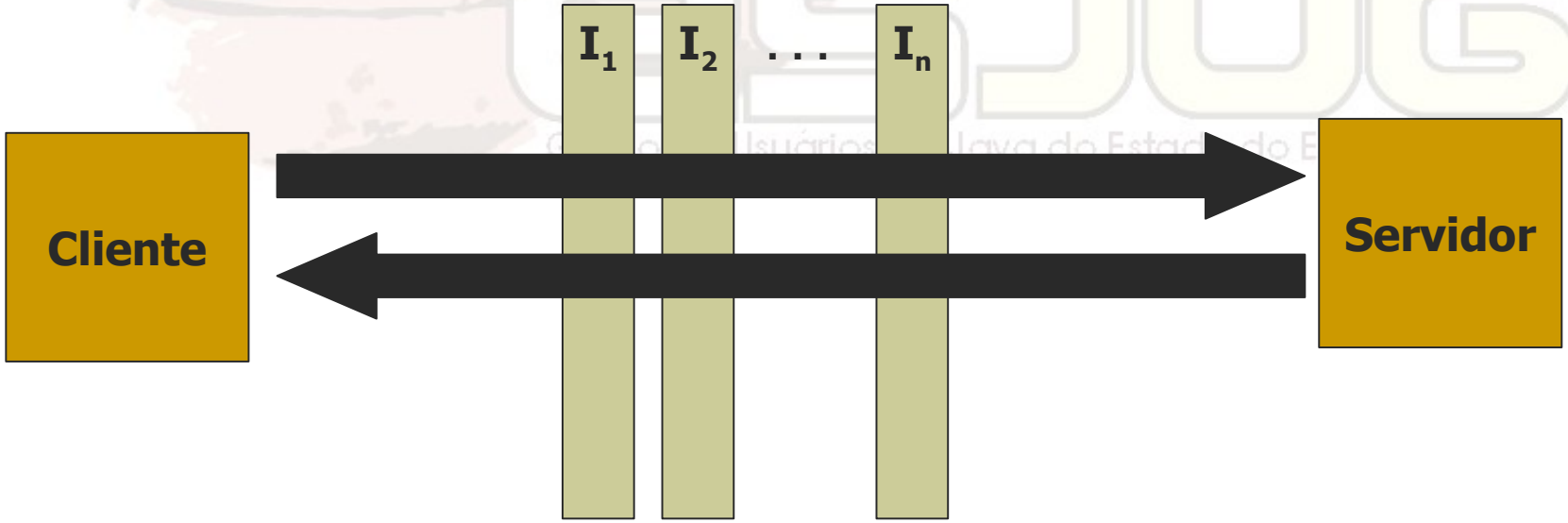
Exemplos em Java

- Os componentes gráficos de Java (AWT/Swing) não usam decorator!
- Os streams do Java I/O usam:

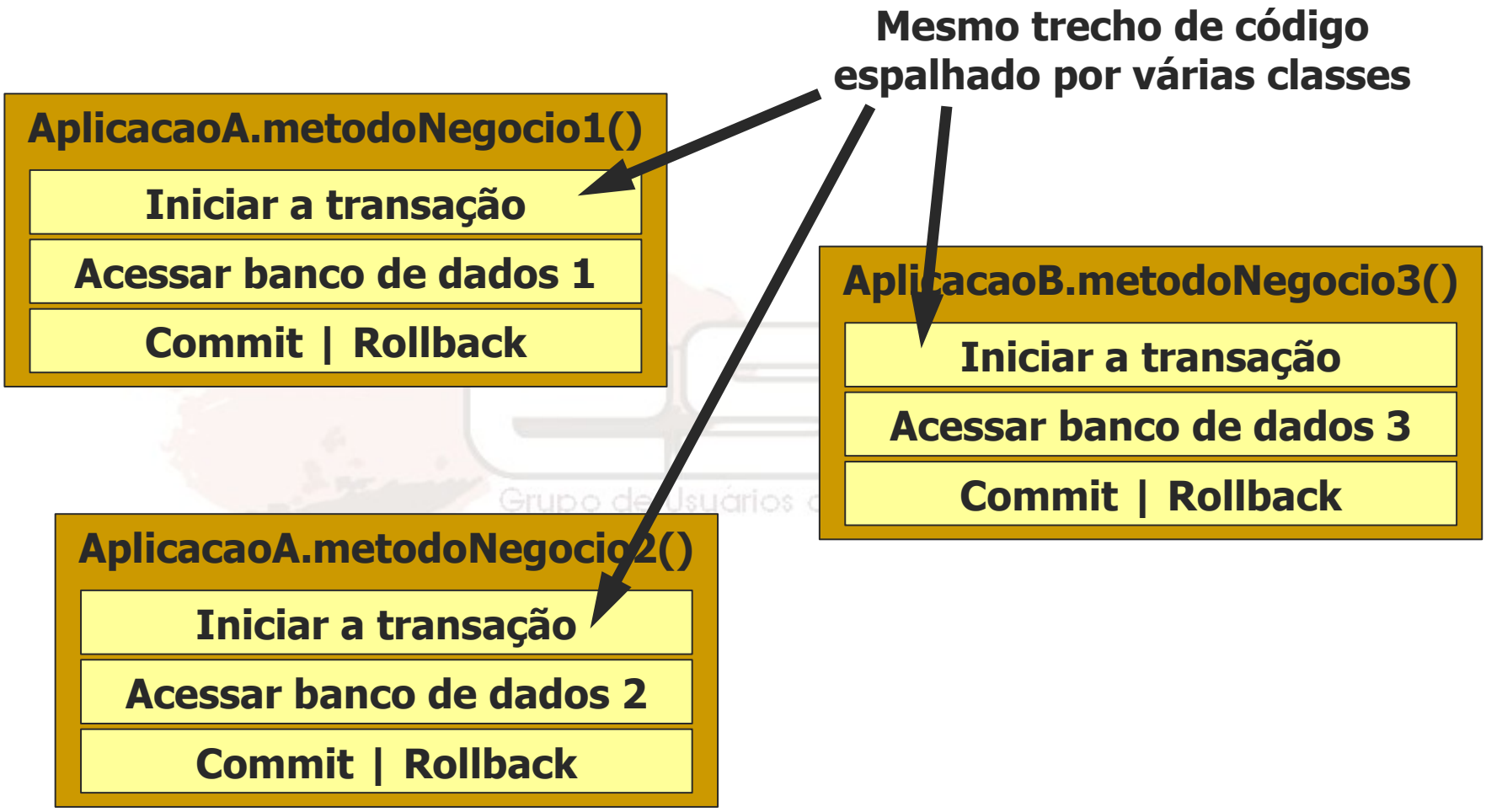


Interceptadores (Interceptors)

- Idéia semelhante aos decoradores;
- Interceptam uma requisição e podem executar código antes e/ou depois.



Programação Orientada a Aspectos (AOP)



Programação Orientada a Aspectos (AOP)

Aspectos são separados e implementados uma única vez.

AplicacaoA.metodoNegocio1()

Acessar banco de dados 1

AplicacaoA.metodoNegocio2()

Acessar banco de dados 2

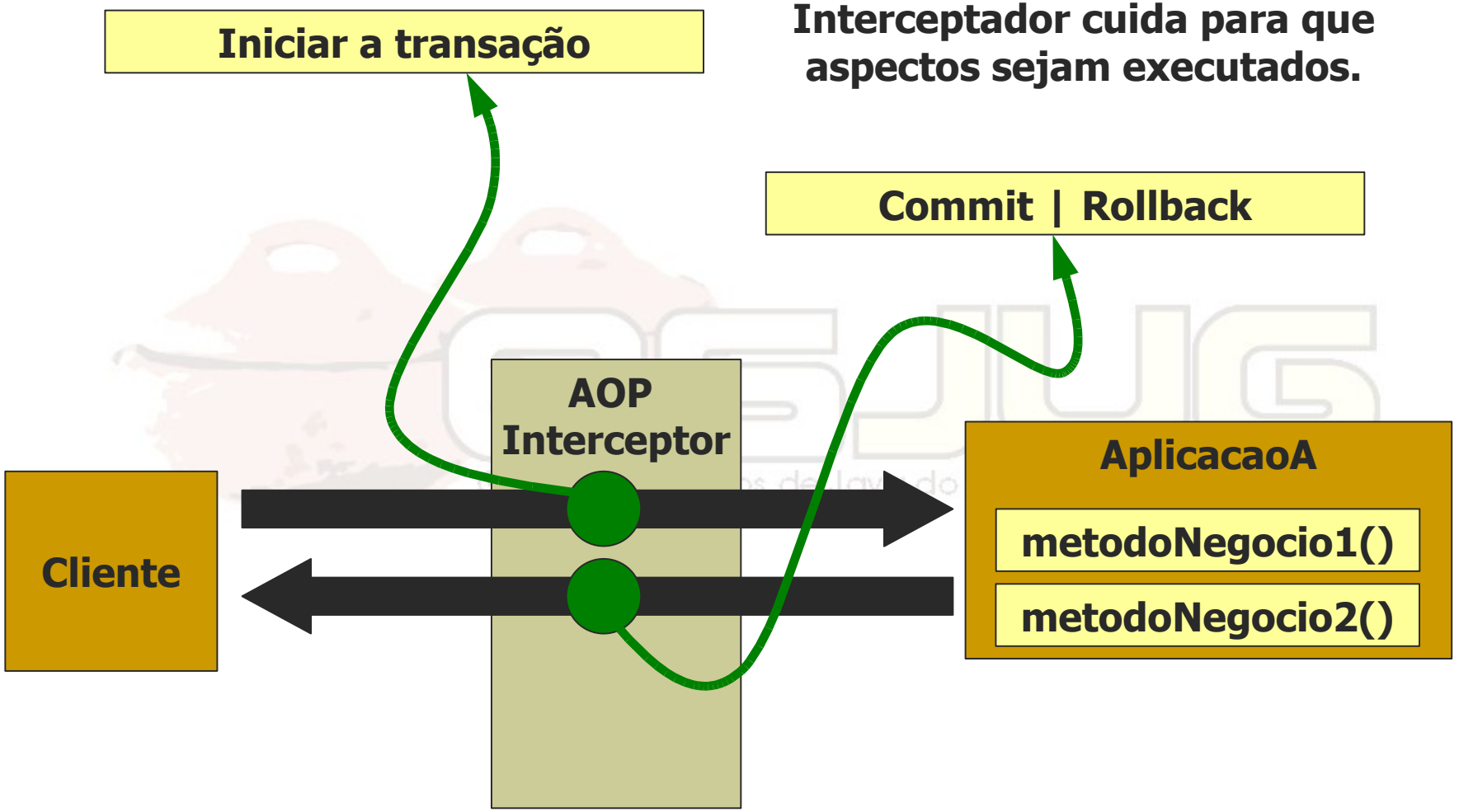
AplicacaoB.metodoNegocio3()

Acessar banco de dados 3

Iniciar a transação

Commit | Rollback

Programação Orientada a Aspectos (AOP)



Código exemplo – cronômetro

```
interface Componente { void executarTarefa(); }

class ComponenteConcreto implements Componente {
    public void executarTarefa() { Thread.sleep(2000); }
}

class Interceptador implements Componente {
    private Componente componente;
    public Interceptador(Componente componente) {
        this.componente = componente;
    }
    public void executarTarefa() {
        long antes = System.currentTimeMillis();
        componente.executarTarefa();
        long depois = System.currentTimeMillis();
        System.out.println((depois - antes) + " ms");
    }
}
```

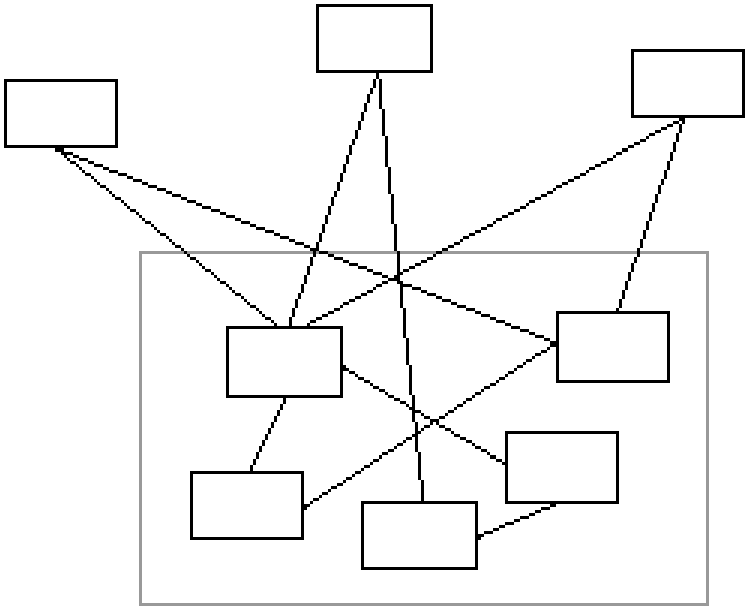
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Façade
(Fachada)
Estrutural / Objeto

Descrição

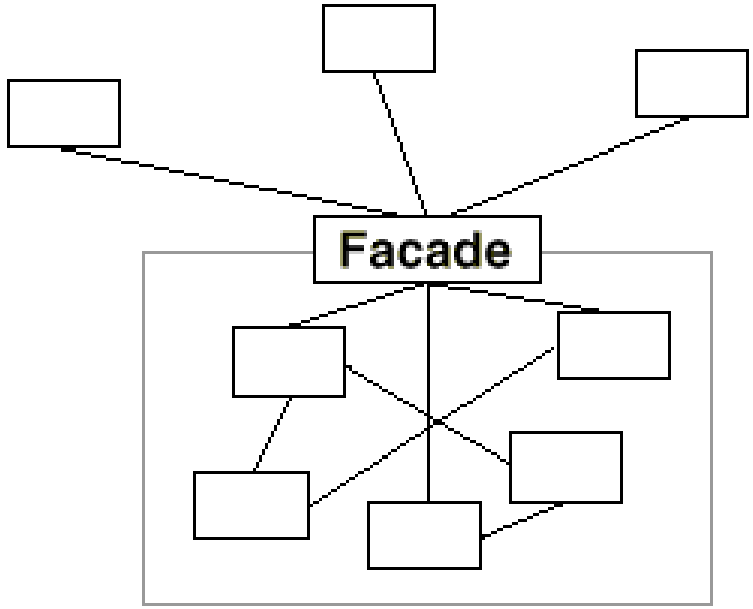
- Intenção:
  - Prover uma interface unificada para um conjunto de interfaces de um subsistema. Define uma interface de mais alto nível para tornar o uso dos subsistemas mais fácil.

O problema



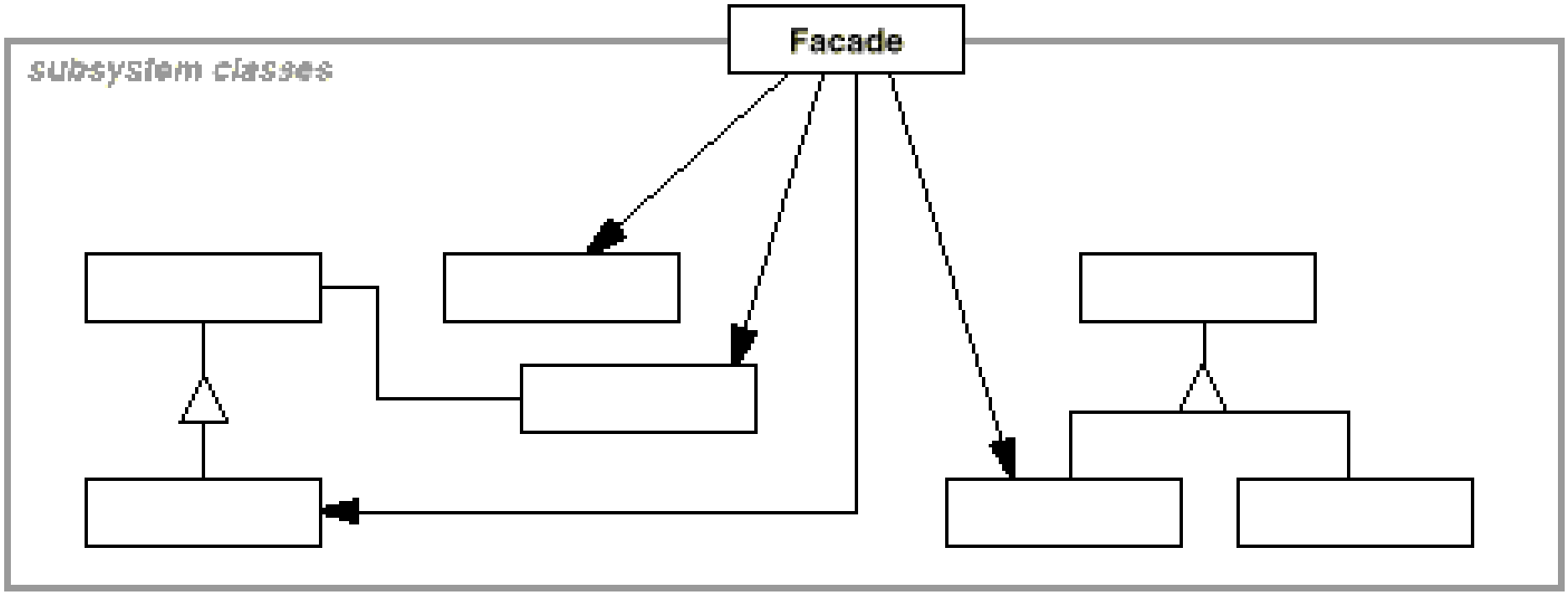
- Clientes acessam vários subsistemas;
- Mudanças em algum subsistema demandam alterações em diversos clientes.

A solução

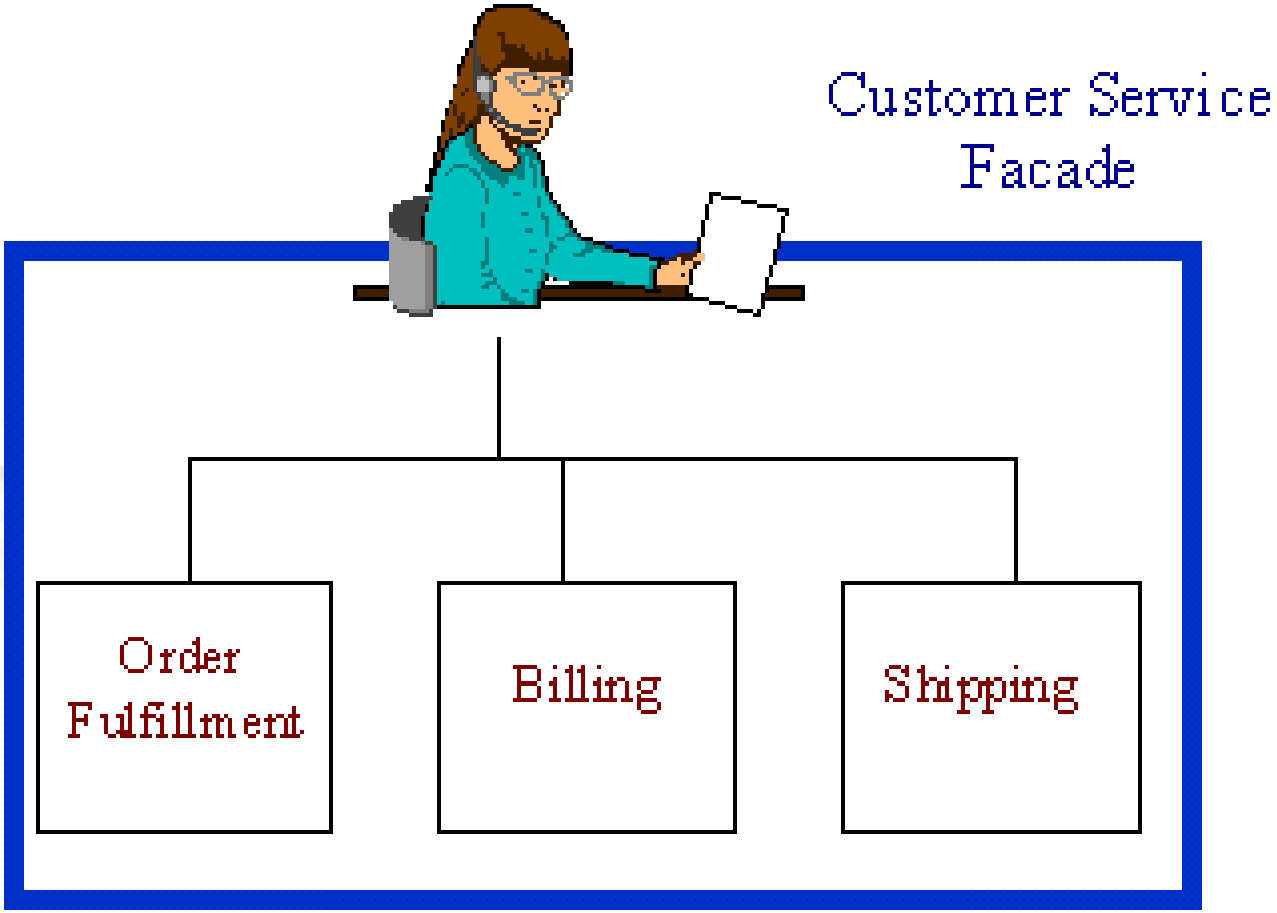


- Introdução de um objeto fachada que provê uma interface simplificada e única ao sistema.

Estrutura



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 prover uma interface simples para um subsistema complexo;
- diminuir a dependência direta entre o cliente e classes internas do seu sistema;
- desenvolver seu sistema em múltiplas camadas, cada uma com sua fachada.

Vantagens e desvantagens

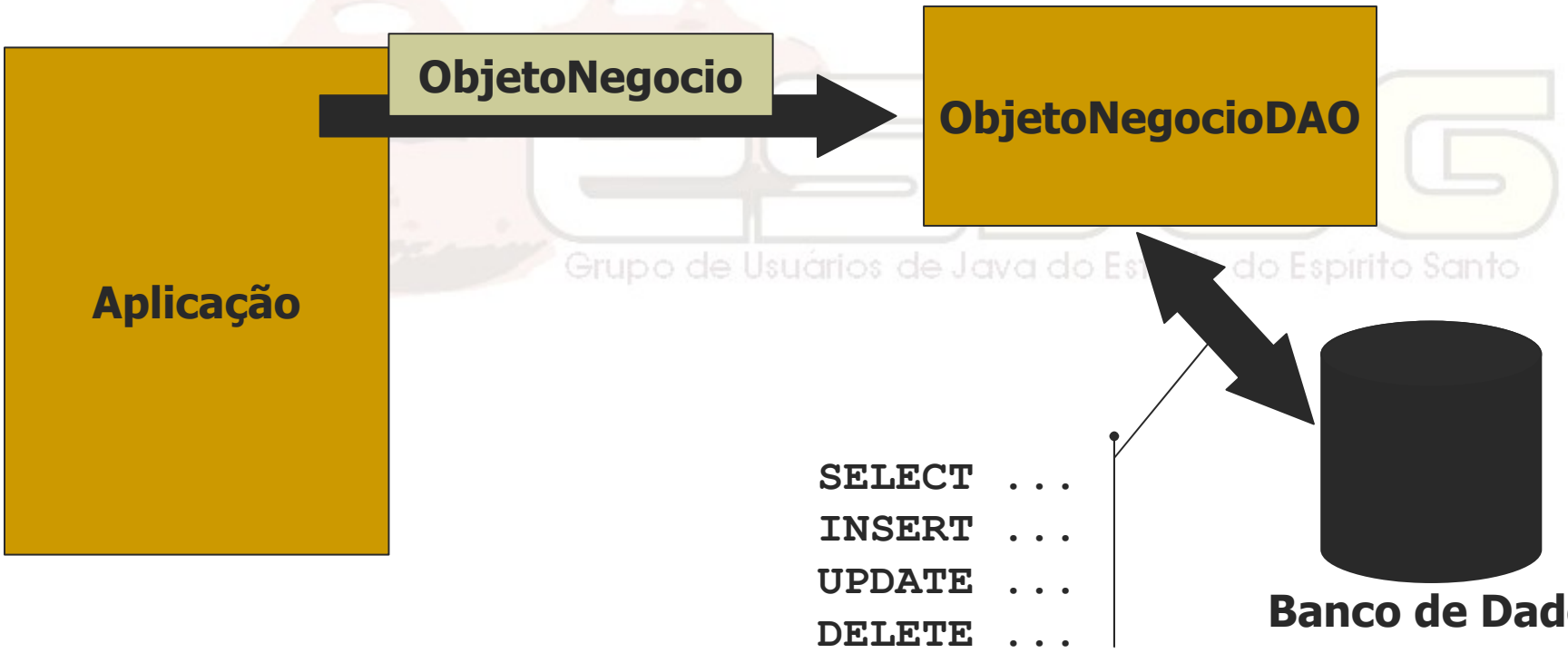
- Facilita a utilização do sistema:
  - Cliente só precisa conhecer a fachada.
- Promove acoplamento fraco:
  - Pequenas mudanças no subsistema não mais afetam o cliente.
- Versatilidade:
  - Quando necessário, clientes ainda podem acessar o subsistema diretamente (se quiser permitir isto).

Façade e Singleton

- Façade geralmente é implementado como Singleton;
- Pode não ser o caso se o sistema tiver multiplos usuários e cada um usar uma fachada separada;
- Fachada só com métodos estáticos é chamada de Utilitário.

Data Access Object

- O padrão DAO pode ser considerado uma fachada para o acesso a dados.



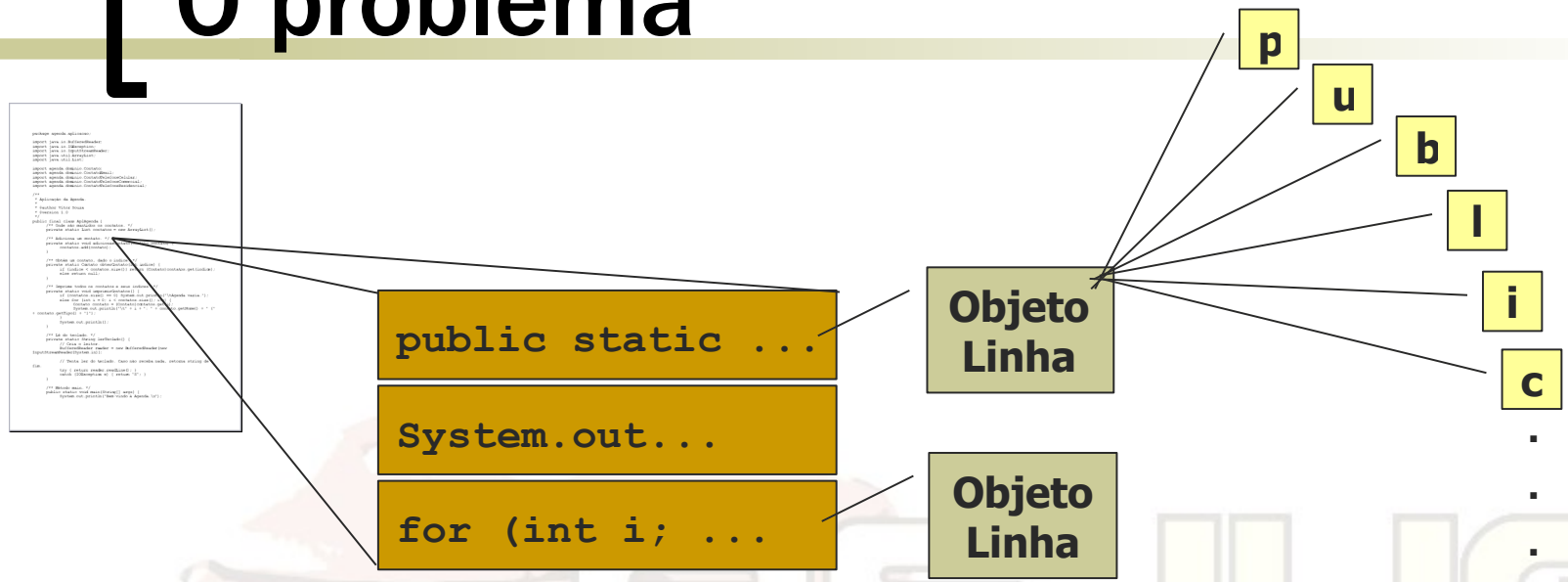
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Flyweight
(Peso mosca)
Estrutural / Objeto

Descrição

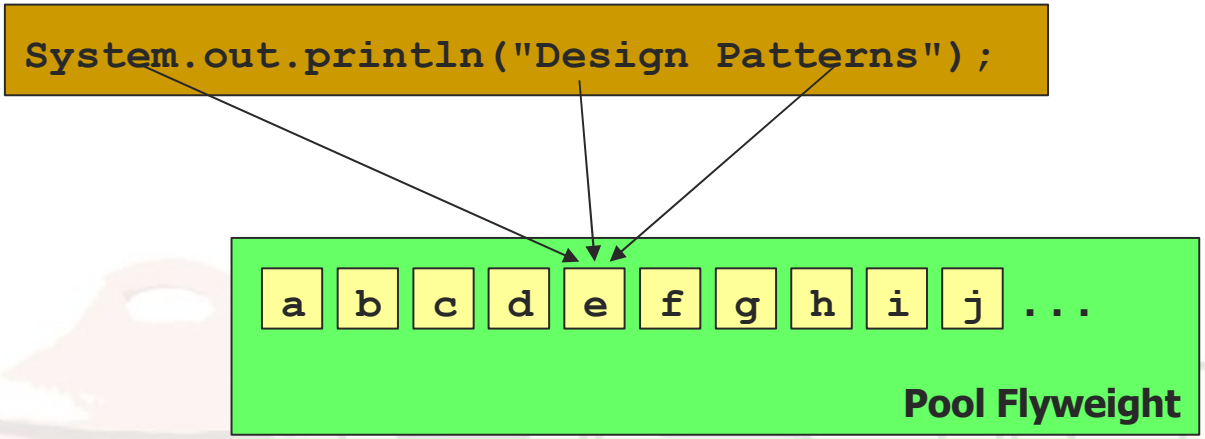
- Intenção:
  - Implantar compartilhamento de objetos de granularidade muito pequena para dar suporte ao uso eficiente de grande quantidade deles.

O problema



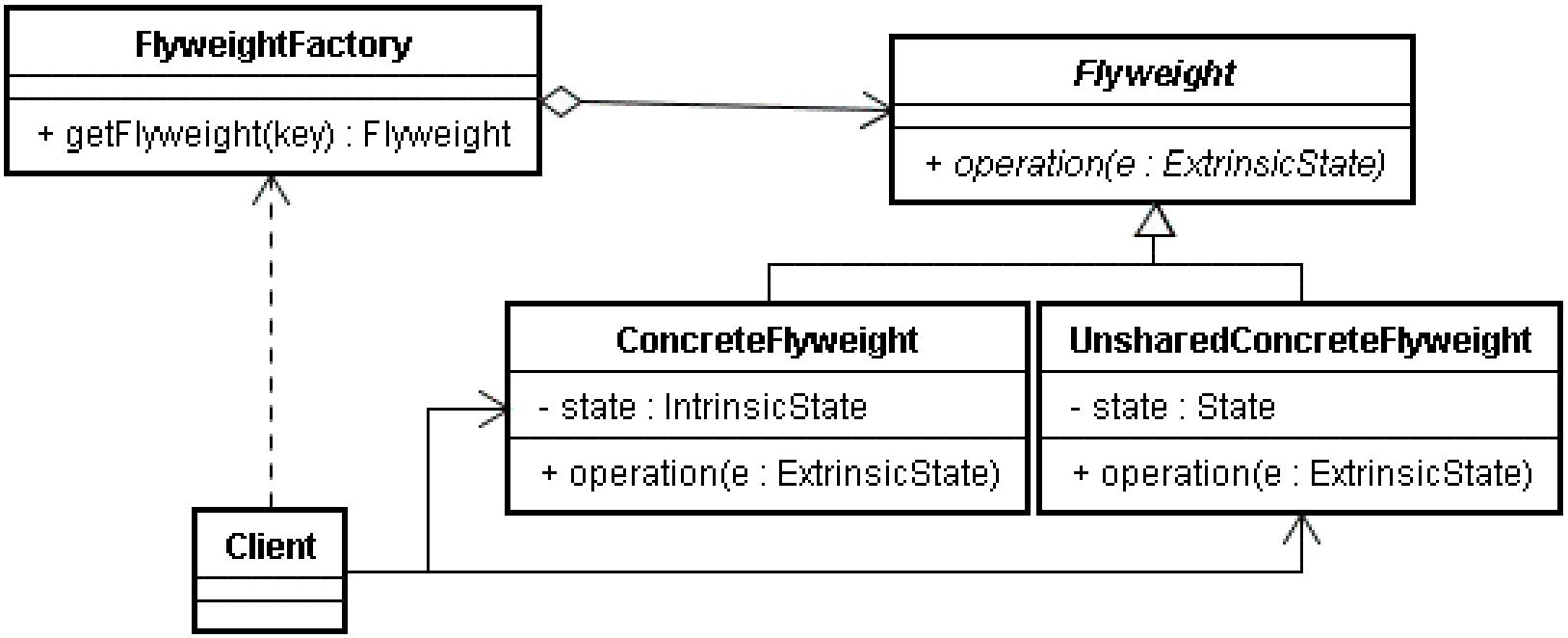
- Desenvolver um editor de texto onde cada caractere é representado por um objeto:
  - Granularidade muito pequena;
  - Não haverá recursos (memória) suficiente para textos grandes.

A solução

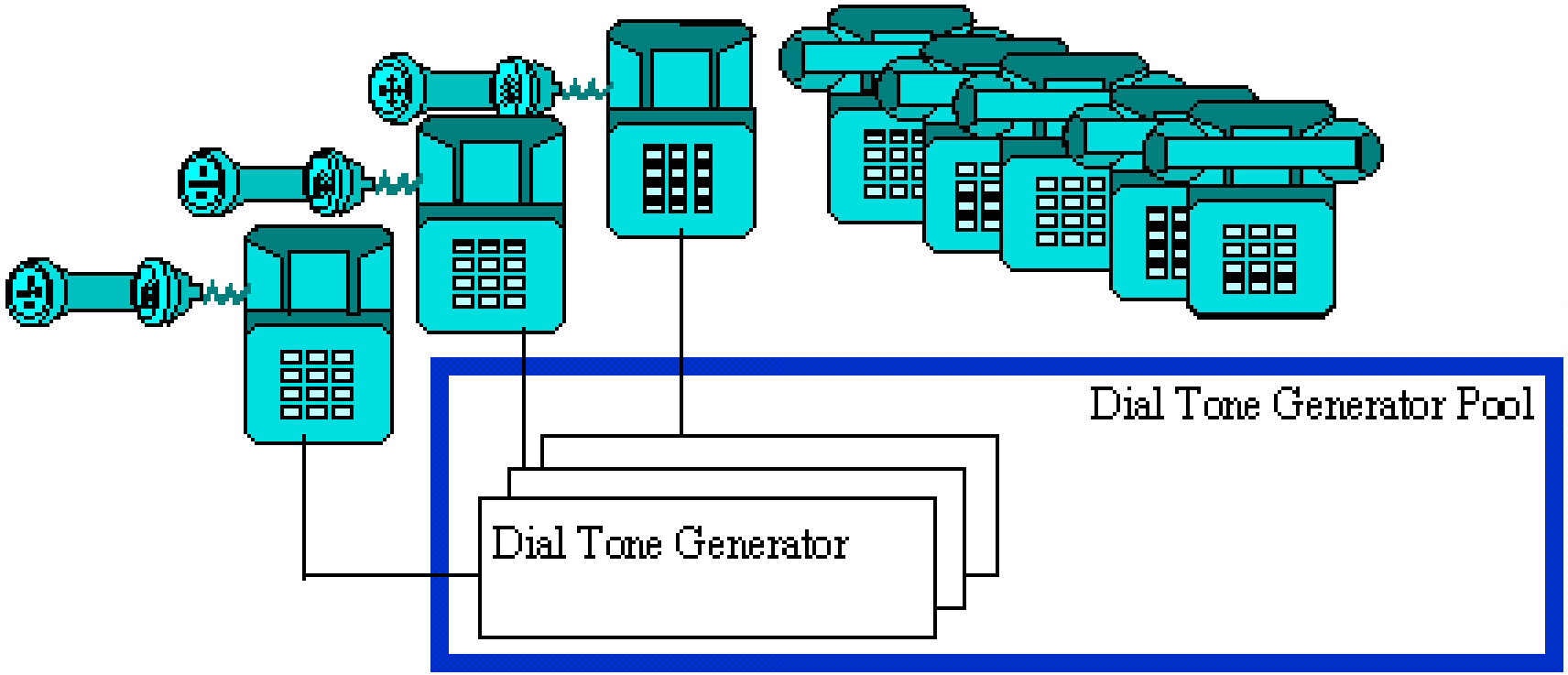


- Monta-se um pool de objetos compartilhados;
- Cada caractere tem um objeto. Com 100 objetos (tabela ASCII) poderíamos montar textos de qualquer tamanho.

Estrutura


FlyweightFactory
+ getFlyweight(key) : Flyweight
Flyweight
+ operation(e : ExtrinsicState)
ConcreteFlyweight
- state : IntrinsicState
+ operation(e : ExtrinsicState)
UnsharedConcreteFlyweight
- state : State
+ operation(e : ExtrinsicState)
Client

Analogia



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

Usar este padrão quando...

- todas as condições forem verdadeiras:
  - A aplicação usa um grande número de objetos;
  - O custo de armazenamento é alto por causa desta quantidade;
  - O estado dos objetos pode ser externalizado;
  - Objetos podem ser compartilhados assim que seu estado é externalizado;
  - A aplicação não depende da identidade.

Vantagens e desvantagens

- Custo x benefício:
  - Custo de recuperar o objeto compartilhado e transferir seu estado externalizado;
  - Benefício de economia de recursos.

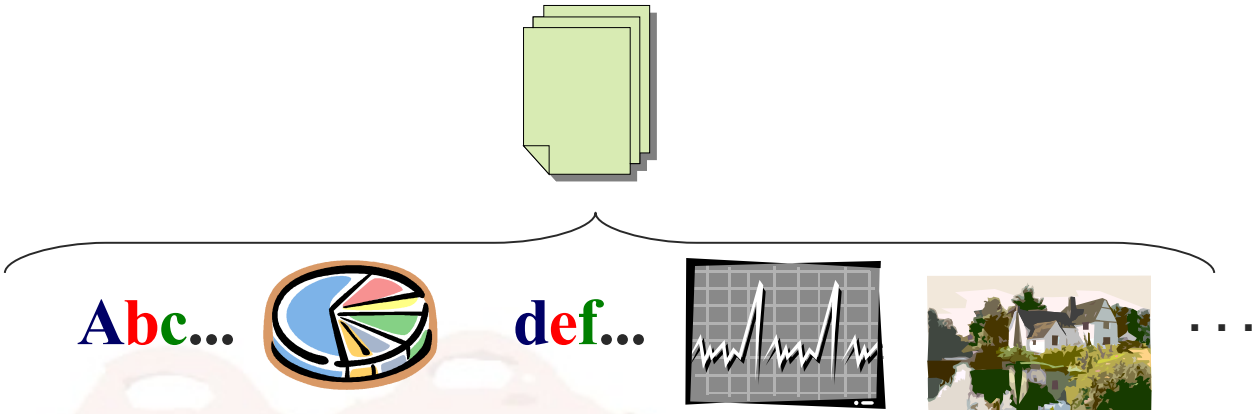
Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Proxy
(Procurador)
Estrutural / Objeto

Descrição

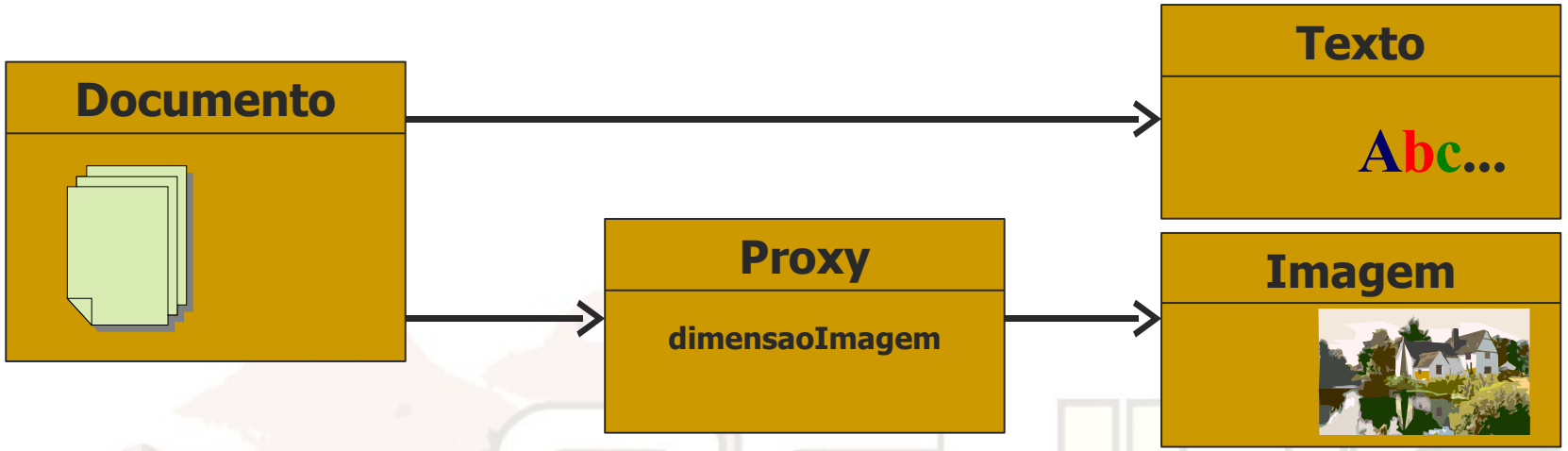
- Intenção:
  - Prover um representante ou ponto de acesso que controle o acesso a um objeto.
- Também conhecido como:
  - Surrogate.

O problema



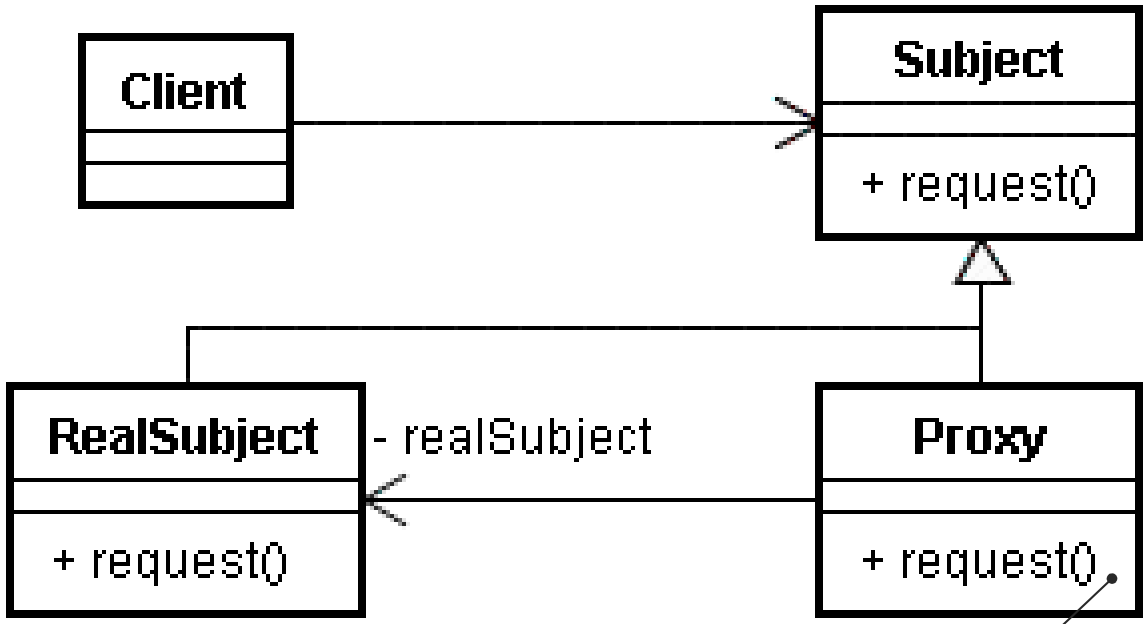
- Considere um editor de texto multimídia (texto e imagens):
  - Carregar todas as imagens do texto assim que ele é aberto pode demorar;
  - Nem todas as imagens aparecem na primeira página, muitas estão "escondidas" mais abaixo.

A solução



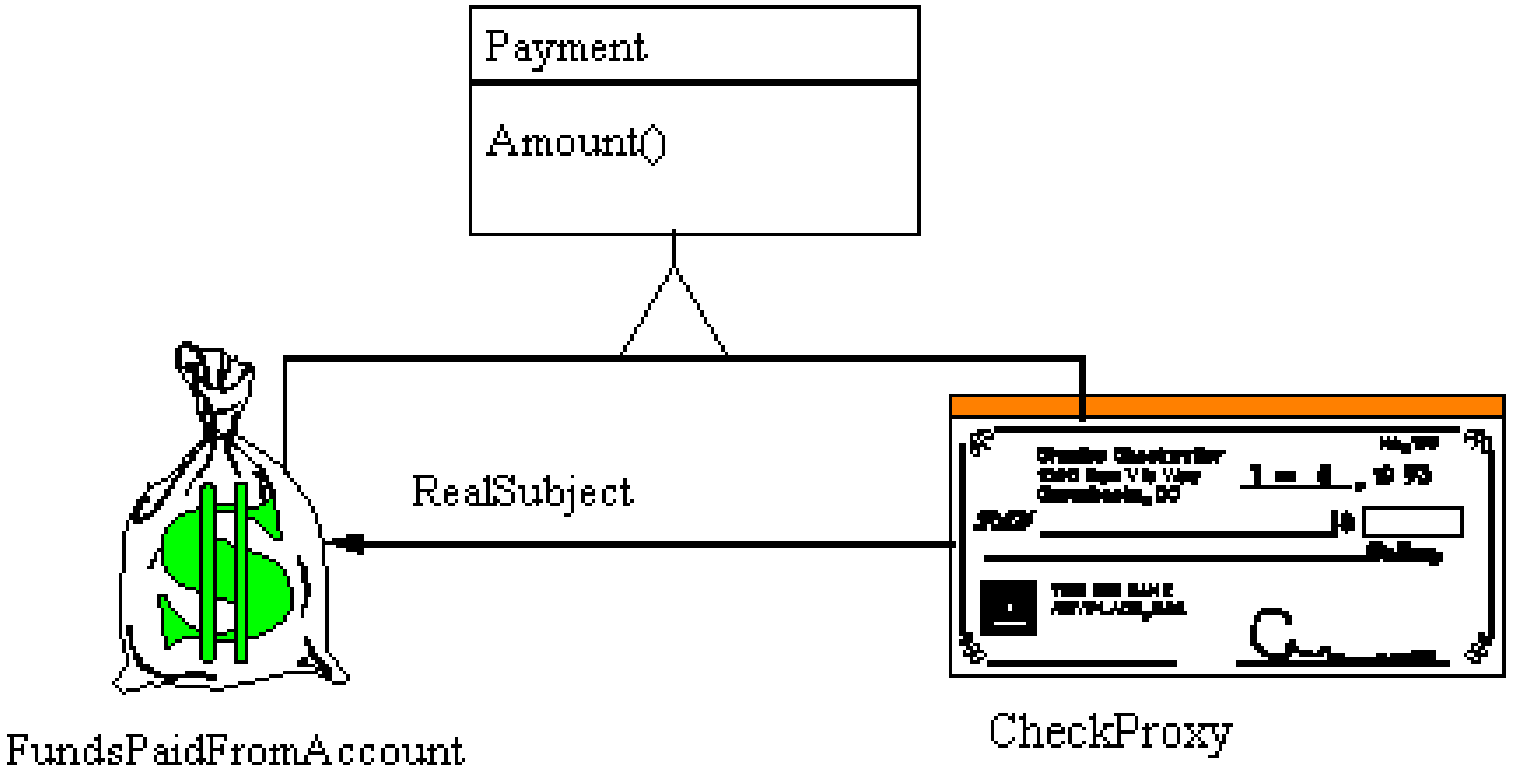
- Documento instancia um objeto proxy, que possui referência à imagem;
- Assim que necessário, proxy carrega a imagem – *lazy loading*.

Estrutura




```
return realSubject.request();
```

Analogia



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

Usar este padrão quando...

- precisar de um acesso mais versátil a um objeto do que um ponteiro:
  - Remote proxy (acesso remoto);
  - Virtual proxy (exemplo da imagem);
  - Protection proxy (controla acesso).

Vantagens e desvantagens

- Adiciona um nível de separação:
  - Transparência na execução de ações de carregamento de objetos.

Dynamic proxy em Java

- Introduzido na versão 1.3;
- Permite a substituição de objetos por proxies de forma transparente;
- Ex.: Hibernate.

Curso - Padrões de Projeto
Módulo 3: Padrões de Estrutura

Vítor E. Silva Souza
vitorsouza@gmail.com

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