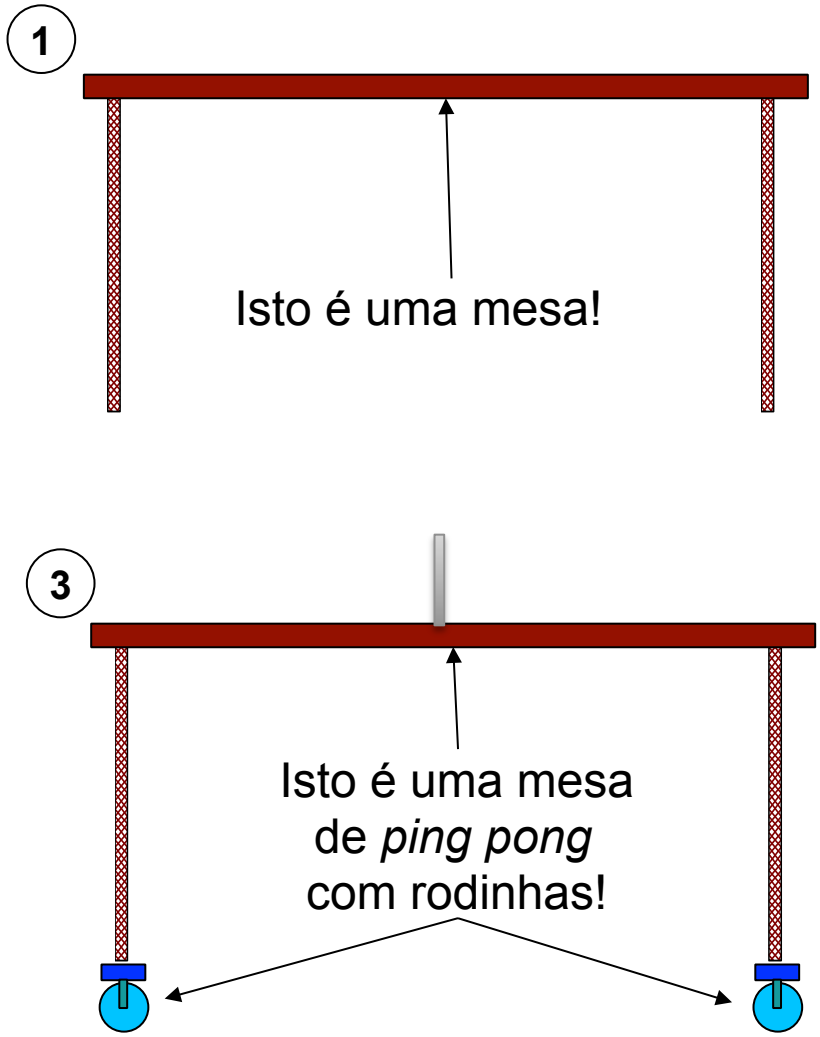
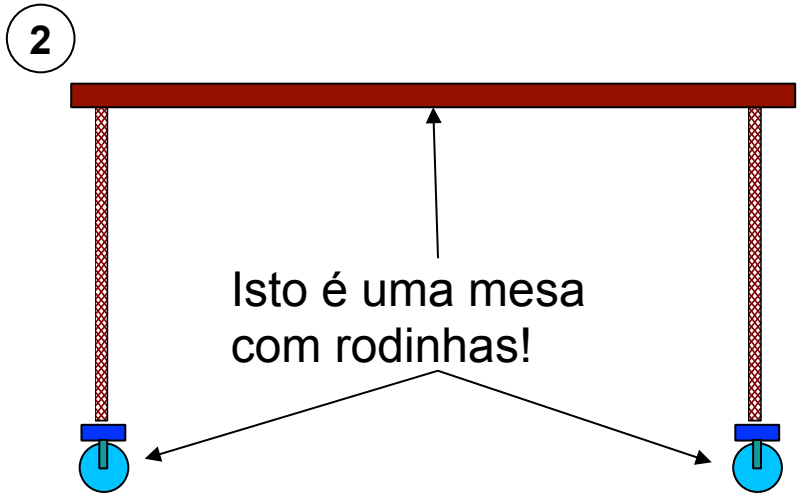
Padrões de Projeto

Decorator

Motivação

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.

Usar herança?

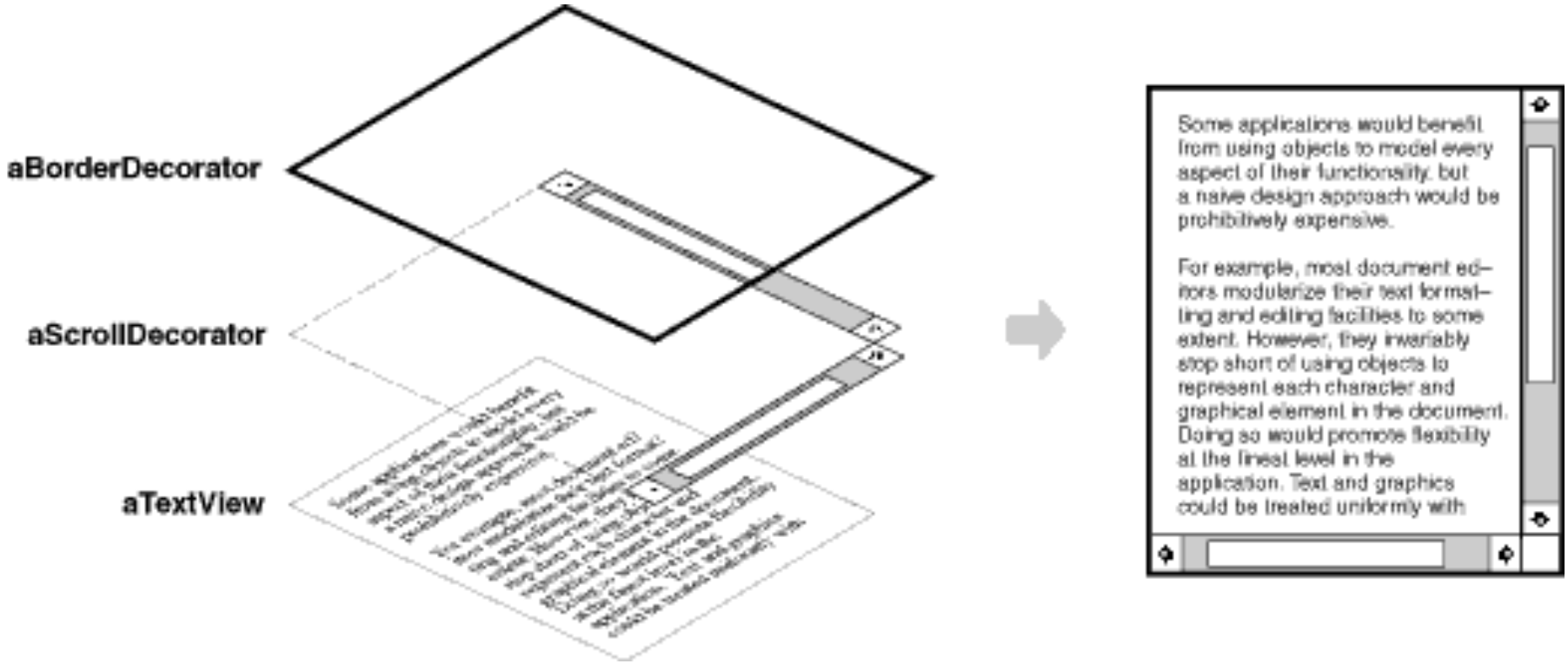
- Solução mais “natural”
- Adição estática de responsablidades
- Cliente não tem como controlar como e quando colocar rodinhas, a rede de ping-pong ou outras coisas
  - Responsabilidade é atribuída a classe e não a cada objeto
- Não é facilmente extensível
- E se quisermos usar a mesa para jantar?

Por que não decorar?

- Embelezar a mesa com objeto que adiciona rodinhas, outro objeto que adiciona rede de ping-pong, etc.
- Cada objeto que "embeleza" a mesa é um *decorator*
- Solução mais flexível pois podemos aproveitar os decorators para outros móveis e combiná-los de inúmeras formas
  - Composição de objetos

Motivação

- Em um editor de documentos, queremos "embelezar" a interface do usuário (UI)
  - Vamos adicionar uma borda à área de edição
  - Vamos adicionar barras de rolagem (scroll bars)



Motivação

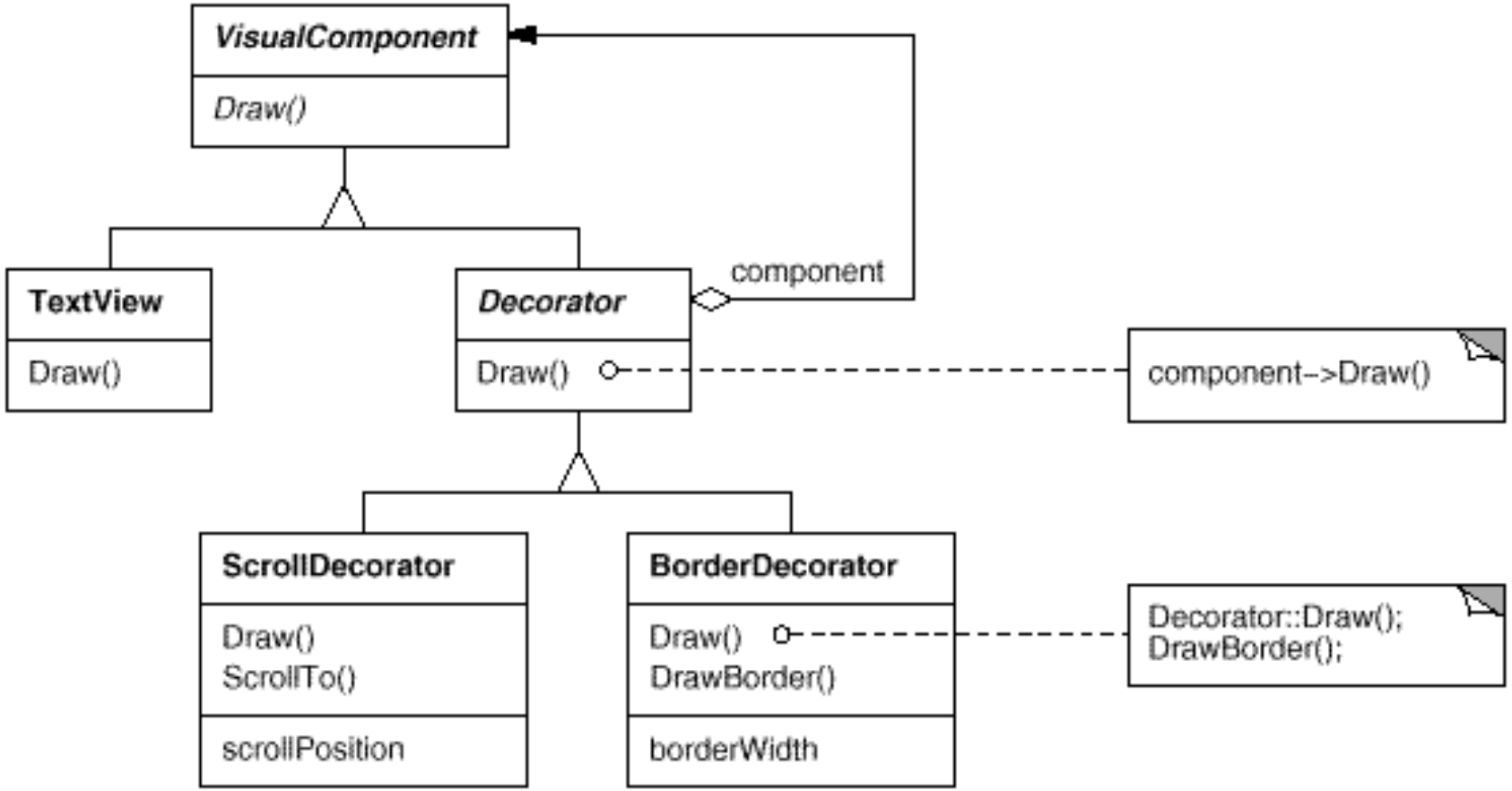
- Desejamos adicionar responsabilidades a objetos, individualmente, e não a uma classe
- **Não** vamos usar herança para adicionar este embelezamento
  - Não poderíamos mudar o embelezamento em tempo de execução
  - Para n tipos de embelezamento, precisaríamos de 2n classes para ter todas as combinações

- Teremos mais flexibilidade se outros objetos da UI não souberem que está havendo embelezamento.

Motivação

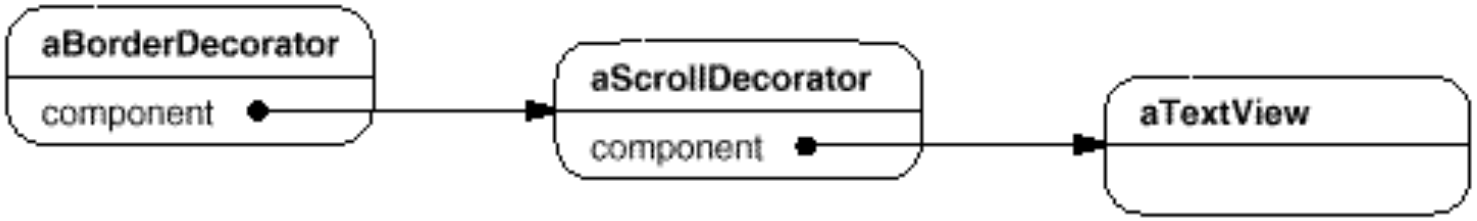
- Usaremos composição de objetos, mas quais objetos devem participar da composição?
- O embelezamento em si será um objeto (digamos uma instância da classe Border) que embelezará uma instância de TextView
  - quem vai compor quem?
- Uma restrição importante é que clientes devem tratar apenas de objetos VisualComponent e não deveriam saber se um TextView tem uma borda ou não.

- Decorator é uma classe abstrata para componentes visuais que embelezam outros componentes visuais
- As classes ScrollDecorator e BorderDecorator são subclasses de Decorator



Sobre o Decorator

- 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
  - O component não conhece seus decorators



Decorator

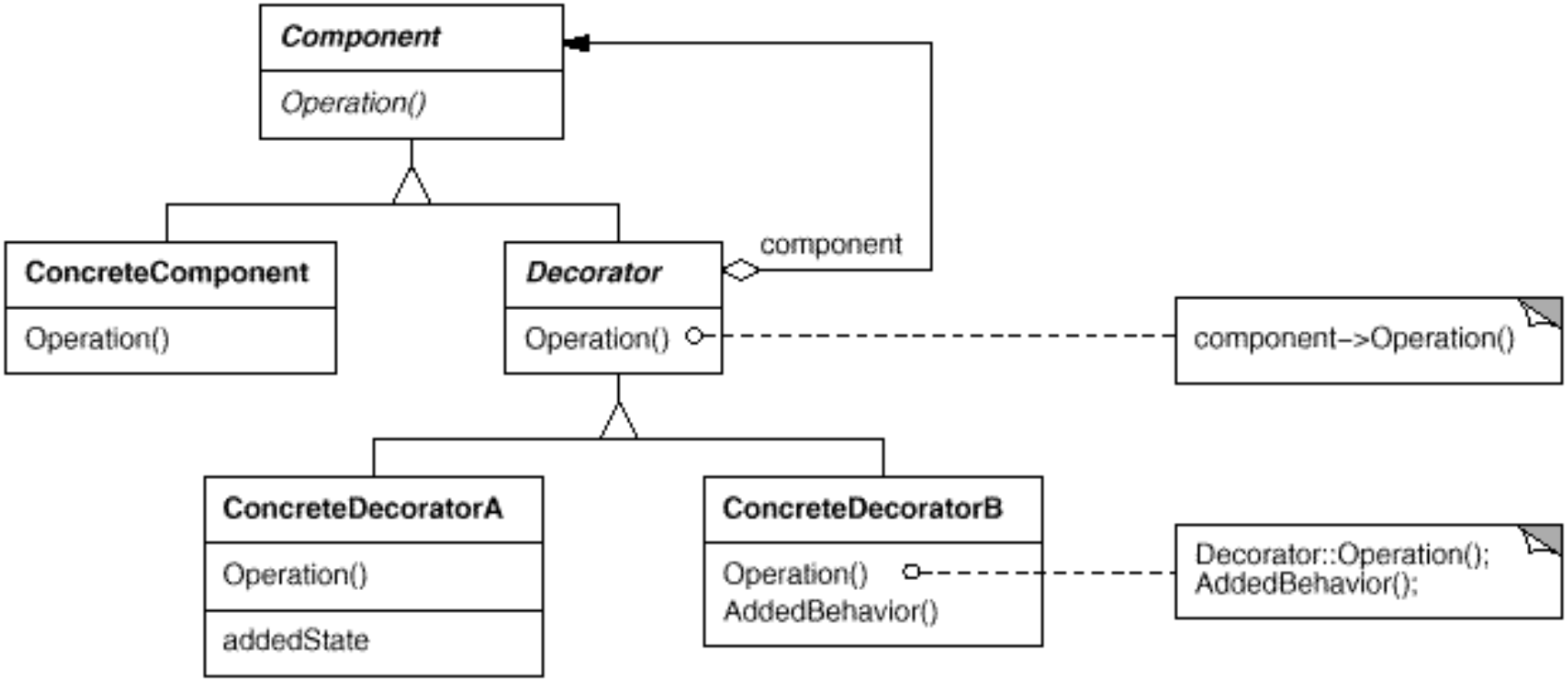
Object Structural

- Objetivo (Intent)
  - Adicionar responsabilidades a um objeto dinamicamente. Decoradores oferecem uma alternativa flexível ao uso de herança para estender funcionalidade.
- Também conhecido como (AKA)
  - Wrapper
- Motivação

Quando usar (Applicability)

- Use o padrão Decorator
  - Para adicionar responsabilidades a objetos individualmente e de modo transparente, ou seja, sem afetar outros objetos.
  - Quando há responsabilidades que podem ser retiradas.
  - Quando a utilização de herança para estender funcionalidades gera uma explosão de subclasses para atender às diversas combinações.

Estrutura


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

Participantes

- Component
  - define a interface para objetos que podem ter responsabilidades acrescentadas a eles 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.

Colaborações

- Decorator encaminha pedidos para o objeto Component que ele embeleza
- Decorator pode, opcionalmente, realizar operações adicionais antes e após encaminhar o pedido.

Consequências

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

Implementação

- Observar conformidade entre as interfaces do Decorator e os componentes
- Decorators abstratos podem ser omitidos se houver apenas um tipo de embelezamento
- Manter simplicidade na superclasse Component!
- Alternativa: utilizar o padrão Strategy para encapsular as diversas funcionalidades adicionais se Component for pesado.

Exemplo de código (1)

Arquivo: VisualComponent.java

```
package decorator;

public interface VisualComponent {
      public void draw();
}
```

Exemplo de código (2)

Arquivo: TextView.java

```
package decorator;

public class TextView implements VisualComponent {
    public TextView() {
    }

    public void draw() {
        //Código para desenhar o TextView
    }
}
```

Exemplo de código (3)

Arquivo: Decorator.java

```
package decorator;

public class Decorator implements VisualComponent
  {
    protected VisualComponent vc;
    public Decorator(VisualComponent vc) {
        this.vc = vc;
    }

    public void draw() {
        vc.draw();              // delega
    }
}
```

Exemplo de código (4)

Arquivo: BorderDecorator.java

```
package decorator;

public class BorderDecorator extends Decorator {
    private int width;

    public BorderDecorator(VisualComponent vc, int width) {
        super(vc);
        this.width = width;
    }

    public void draw() {
        super.draw();
        drawBorder();
    }

    private void drawBorder() {
        /* Codigo para desenhar a borda
           em torno do VisualComponent */
    }
}
```

Exemplo de código (5)

Arquivo: ScrollDecorator.java

```
package decorator;

public class ScrollDecorator extends Decorator {
    public ScrollDecorator(VisualComponent vc) {
        super(vc);
    }

    public void draw() {
        super.draw();
        drawScrollBar();
    }

    private void drawScrollBar() {
        /* Codigo para desenhar a scrollbar
           em torno do VisualComponent */
    }
}
```

Exemplo de código (6)

Arquivo: Window.java

```
package decorator;

public class Window {
    private VisualComponent contents;

    public void setContents(VisualComponent vc) {
        contents = vc;
    }
}
```

Exemplo de código (7)

Arquivo: Program.java

```
package decorator;

public class Program {
    public static void main(String[] args) {
        Window w = new Window();
        w.setContents(new BorderDecorator(
                          new ScrollDecorator(
                              new TextView()), 1));
    }
}
```

Usos conhecidos (Known Uses)

- Diversos toolkits de interfaces gráficas OO

Padrões relacionados

- Adapter (139): mudanças de interface X mudanças nas responsabilidades.
- Composite (163): decorator como composite degenerado (apenas um componente)? Decorator adiciona responsabilidades – não serve para agregação de objetos.
- Strategy (315): skin X guts.

Referências

- Livro do GOF
- Design Patterns Java Workbook
- Material do prof. Jacques Sauvé, UFCG
- Material do Grupo aSide @ UFBA