

INF011 – Padrões de Projeto

12 – Decorator

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas



Decorator

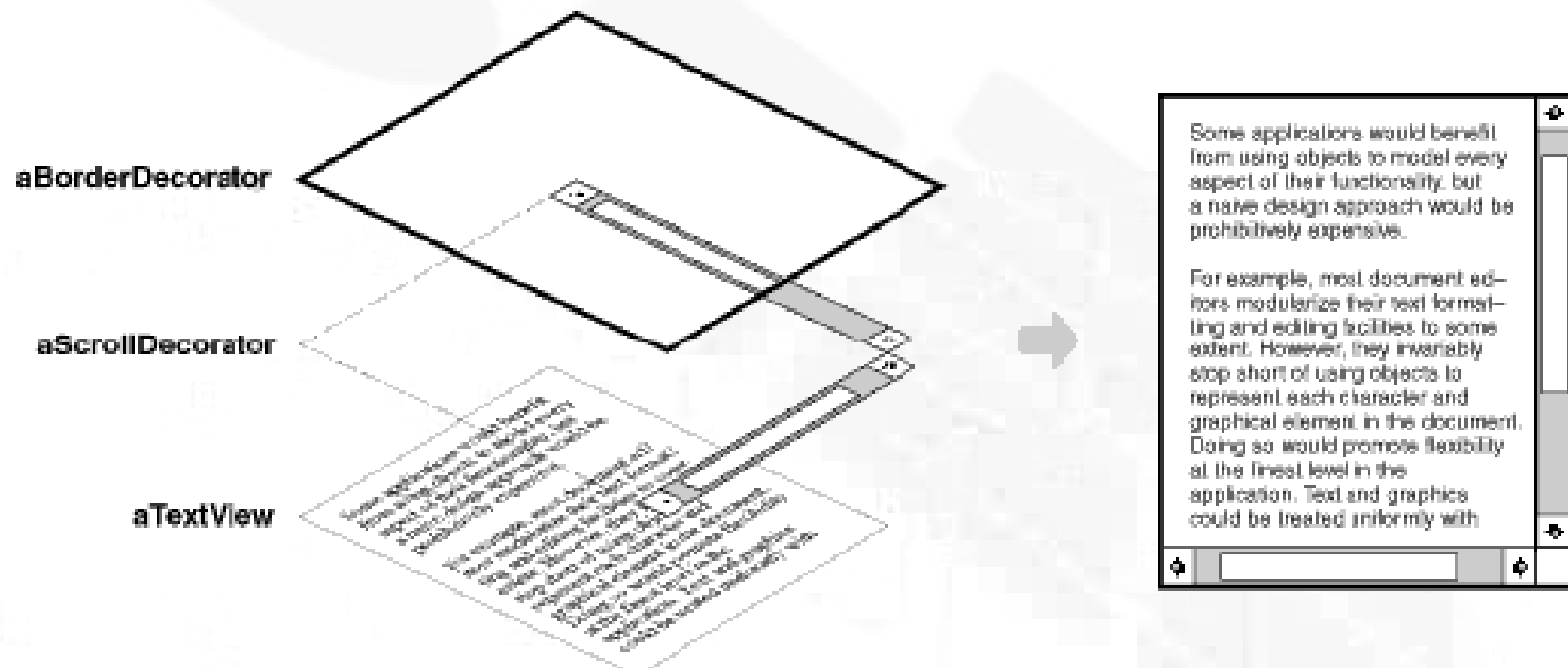
- Propósito:
 - Anexar, de forma dinâmica, responsabilidades adicionais a um objeto. Representa uma alternativa flexível à herança de implementação ao realizar extensão de funcionalidades
- Também conhecido como: Wrapper
- Motivação:
 - Às vezes é necessário adicionar responsabilidades somente a objetos específicos, ao invés de a classes inteiras
 - Ex: um toolkit gráfico pode permitir acrescentar bordas ou scrolling a qualquer widget
 - Pode-se herdar de uma classe que implementa a borda, porém toda instância da sub-classe terá uma borda

Decorator

- Motivação:
 - Entretanto, a solução é inflexível pois a escolha da borda é realizada de forma estática, o cliente não pode controlar como e quando decorar o widget com uma borda
 - Uma abordagem mais flexível é encapsular o widget em um outro objeto, que adiciona a borda (decorator)
 - O decorator expõe a mesma interface do widget de modo que sua presença é transparente para o cliente
 - O decorator repassa as requisições para o widget e pode realizar ações adicionais (tais como o desenho da borda) antes ou depois do repasse
 - Pode-se utilizar decorators aninhados de forma recursiva, provendo portanto um número ilimitado de responsabilidades adicionadas

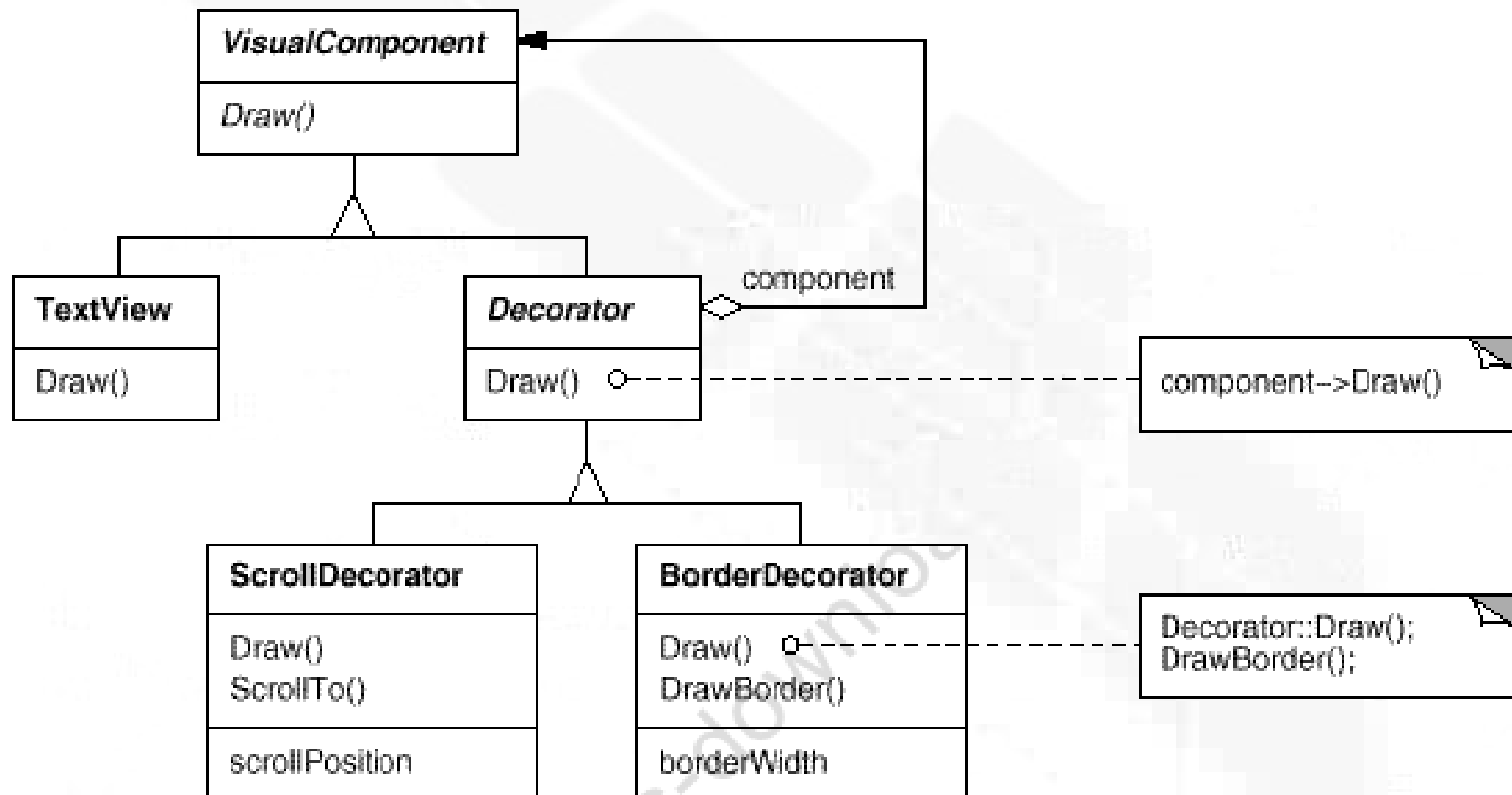
Decorator

- Motivação:



Decorator

- Motivação:

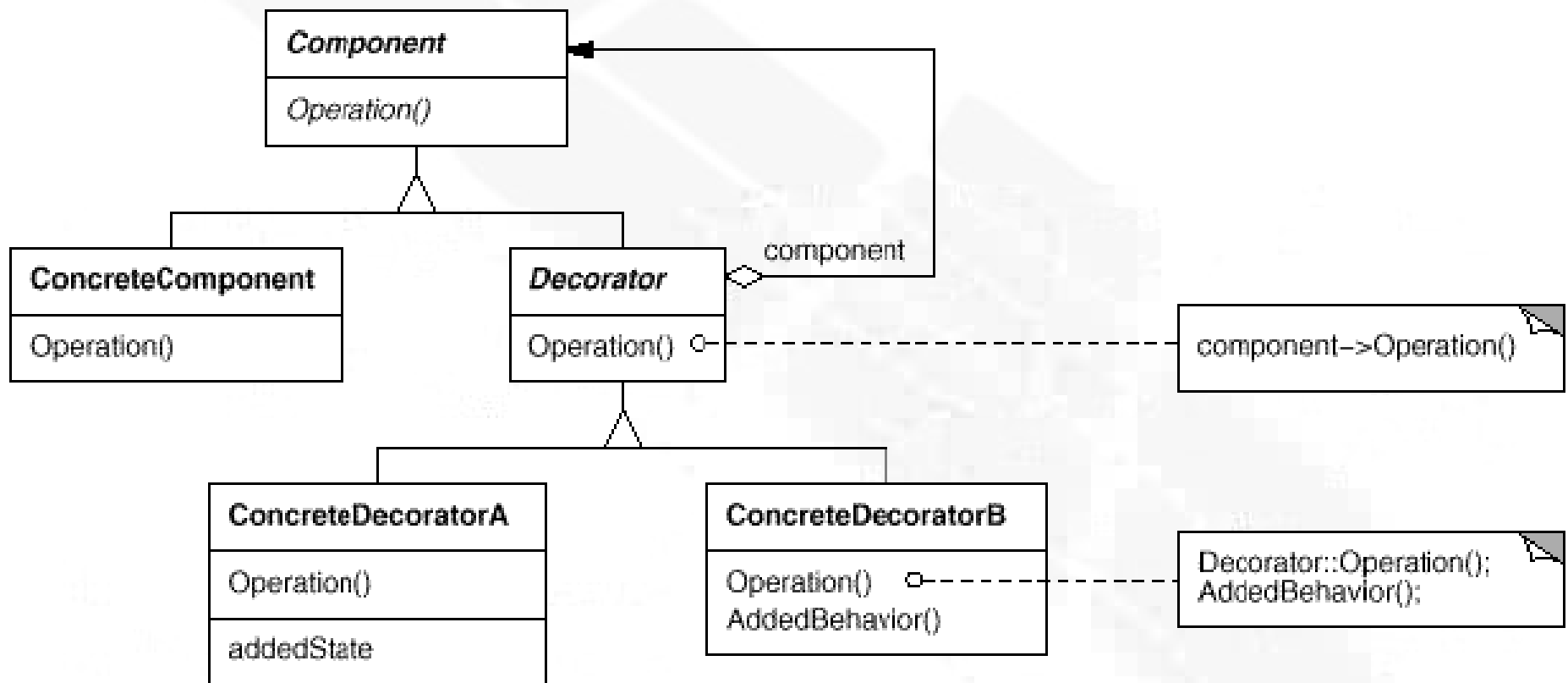


Decorator

- Aplicabilidade:
 - Deseja-se adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos
 - Deseja-se ter responsabilidades que podem ser retiradas de um objeto específico
 - Quando extensão por herança de implementação é impraticável:
 - Explosão de sub-classes gerada pela combinação de um grande número de extensões independentes
 - Definição da classe não disponível

Decorator

- Estrutura:



Decorator

- Participantes:
 - Component (VisualComponent):
 - Define a interface dos objetos que podem ter responsabilidades a eles adicionadas dinamicamente
 - ConcreteComponent (TextView):
 - Define um objeto no qual pode-se anexar responsabilidades adicionais
 - Decorator:
 - Mantém uma referência para um objeto Component e define uma interface em conformidade com a interface de Component
 - ConcreteDecorator (BorderDecorator, ScrollDecorator):
 - Adiciona responsabilidades ao componente

Decorator

- Colaborações:
 - O decorator repassa requisições ao seu objeto Component. Pode, opcionalmente, realizar operações adicionais antes ou depois do repasse

Decorator

- Consequências:
 - Mais flexível que herança estática (de implementação):
 - Responsabilidades podem ser adicionadas/removidas em run-time simplesmente anexando/desanexando decorators
 - Herança iria demandar uma nova classe para cada responsabilidade adicional
 - Pode-se adicionar uma propriedade duas vezes (ex: borda dupla). Herdaríamos da classe que implementa a borda duas vezes ?

Decorator

- Consequências:
 - Evita classes carregadas de funcionalidades no alto da hierarquia:
 - O Decorator oferece uma abordagem pay-as-you-go
 - Ao invés de tentar suportar todas as funcionalidades previstas em uma classe configurável extremamente complexa define-se uma classe simples e adiciona-se funcionalidades de forma incremental, através de objetos decorator
 - A aplicação não paga por funcionalidades que não utiliza
 - É fácil definir novos decorators independente das classes que eles estendem

Decorator

- Consequências:
 - O decorator e o seu componente não são idênticos:
 - Um decorator atua como um encapsulamento transparente mas, sob um ponto de vista de identidade de objeto, não é igual ao seu componente
 - Não deve-se depender da identidade de objetos quando utilizando decorators
 - Grande quantidade de objetos pequenos:
 - Um projeto que usa o Decorator geralmente resulta em um sistema formado por diversos objetos pequenos parecidos, que diferem na forma com que foram conectados
 - Pode ser difícil de compreender e depurar

Decorator

- Implementação:
 - Conformidade de interface:
 - A interface de um objeto decorator deve estar em conformidade com a interface do componente que ele decora
 - Portanto, decorators concretos devem herdar de uma classe comum
 - Omitindo a classe abstrata Decorator:
 - Pode-se omitir a classe abstrata do decorator quando necessita-se adicionar apenas uma responsabilidade
 - Desempenho das classes componente:
 - A interface comum dos componentes e decorators deve focar em serviços, não em armazenamento de dados

Decorator

- Código exemplo:

```
class VisualComponent {  
public:  
    VisualComponent();  
  
    virtual void Draw();  
    virtual void Resize();  
    // ...  
};
```

Decorator

- Código exemplo:

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Decorator

- Código exemplo:

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

Decorator

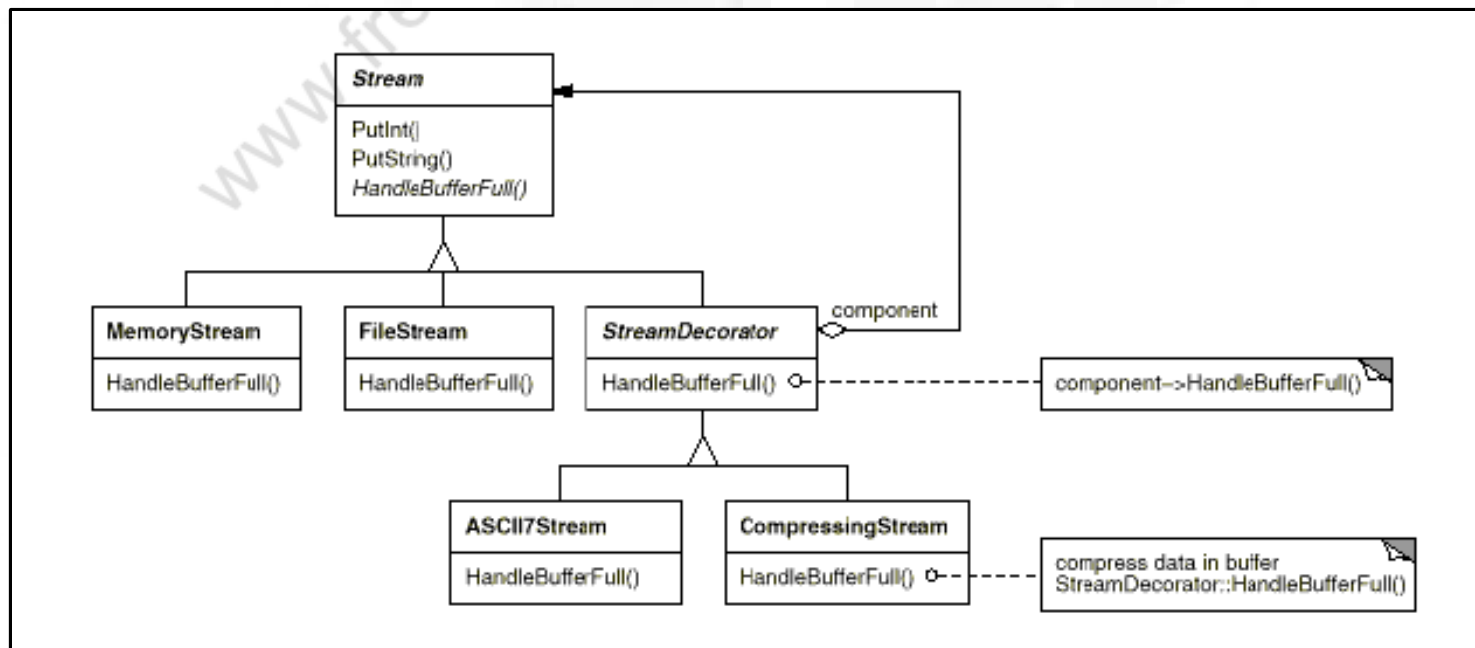
- Código exemplo:

```
void Window::SetContents (VisualComponent* contents) {  
    // ...  
}  
  
Window* window = new Window;  
TextView* textView = new TextView;  
  
window->SetContents(textView);
```

```
new BorderDecorator(  
    new ScrollDecorator(textView), 1
```

Decorator

- Usos conhecidos:
 - InterViews
 - ET++
 - Facilidades de I/O



Decorator

- Padrões relacionados:
 - O Decorator é diferente do Adapter no sentido que o Decorator somente muda as responsabilidades do objeto e não a sua interface. O Adapter dá ao objeto uma interface completamente diferente
 - Um Decorator pode ser visto como um Composite degenerado com somente um componente. Entretanto, o Decorator adiciona responsabilidades, não tem a intenção de realizar agregações
 - Um Decorator permite que você troque a “casca” de um objeto. Um Strategy permite que você troque o comportamento interno de um objeto

INF011 – Padrões de Projeto

12 – Decorator

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

