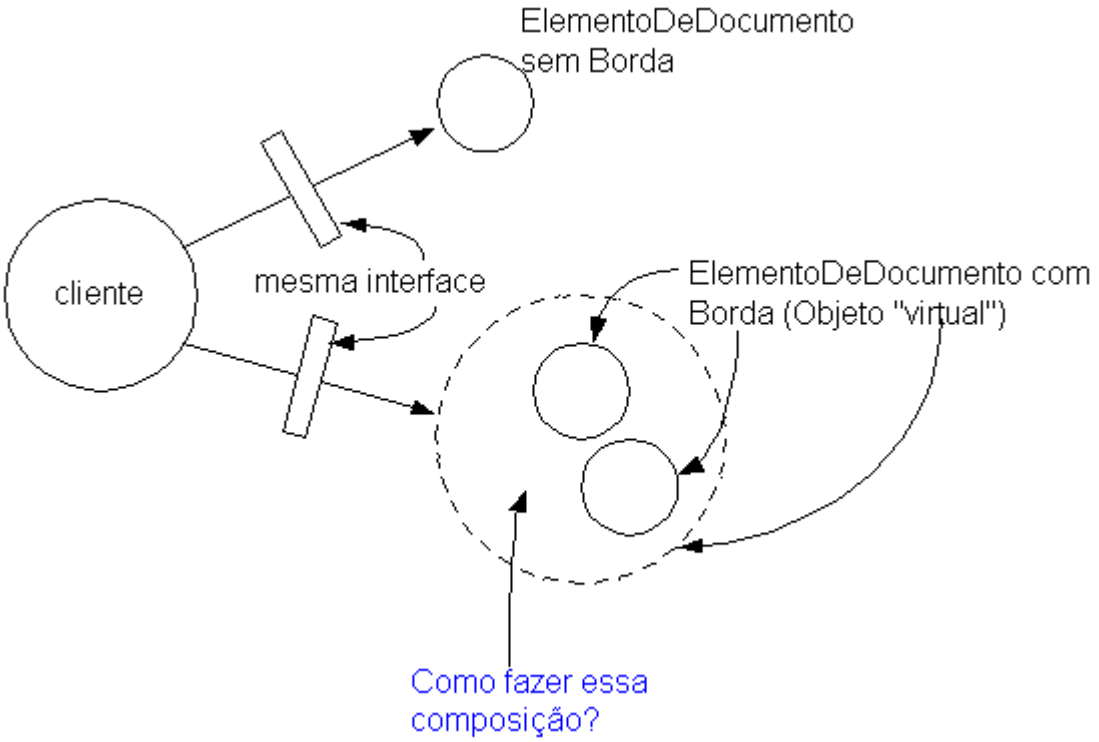
Decorator

Introdução

- No editor de documentos *WYSIWYG*, vamos embelezar a interface do usuário (UI)
  - Vamos adicionar uma borda à área de edição
  - Vamos adicionar barras de rolagem (*scroll bars*)
- *Não* vamos usar herança para adicionar este embelezamento
  - Não poderíamos mudar o embelezamento em tempo de execução
  - Para n tipos de embelezamento, precisaríamos de 2^n classes para ter todas as combinações
- Teremos mais flexibilidade se outros objetos da UI não souberem que está havendo embelezamento!

Inclusão transparente

- Usaremos composição em vez de herança evita os problemas mencionados, mas quais objetos devem participar da composição?
- O embelezamento em si será um objeto (digamos uma instância da classe Borda)
- Isso nos dá dois objetos para fazer a composição: ElementoDeDocumento e Borda
  - Observe que, desta forma, podemos ter uma borda a redor de qualquer ElementoDeDocumento e não só a redor da área de edição
- Devemos decidir agora quem vai compor quem
  - Uma restrição importante é que clientes devem tratar apenas objetos ElementoDeDocumento e não deveriam saber se um ElementoDeDocumento tem uma Borda ou não!
  - Veja a situação abaixo



- Na figura acima, clientes tratam ElementoDeDocumento uniformemente
  - Se um cliente manda um ElementoDeDocumento sem Borda se desenhar não vai aparecer borda, caso contrário, vai aparecer borda, mas o cliente não sabe
- Perguntamos novamente: "Quem vai compor quem?"
  - A Borda pode conter o ElementoDeDocumento
    - Faz sentido já que a Borda engloba o ElementoDeDocumento na tela
  - O ElementoDeDocumento pode conter a Borda
    - Isso implica em mudar a classe ElementoDeDocumento para que ela conheça a Borda
- Usaremos a primeira escolha de forma a manter o código que trata bordas inteiramente na classe Borda sem mexer nas outras classes
- Como construir a classe Borda?
  - Já que a Borda vai incluir o ElementoDeDocumento, é a Borda que será "vista" pelo cliente
  - Mas o cliente só trata com ElementoDeDocumento!
  - Isso implica que a Borda tem a mesma interface

que ElementoDeDocumento

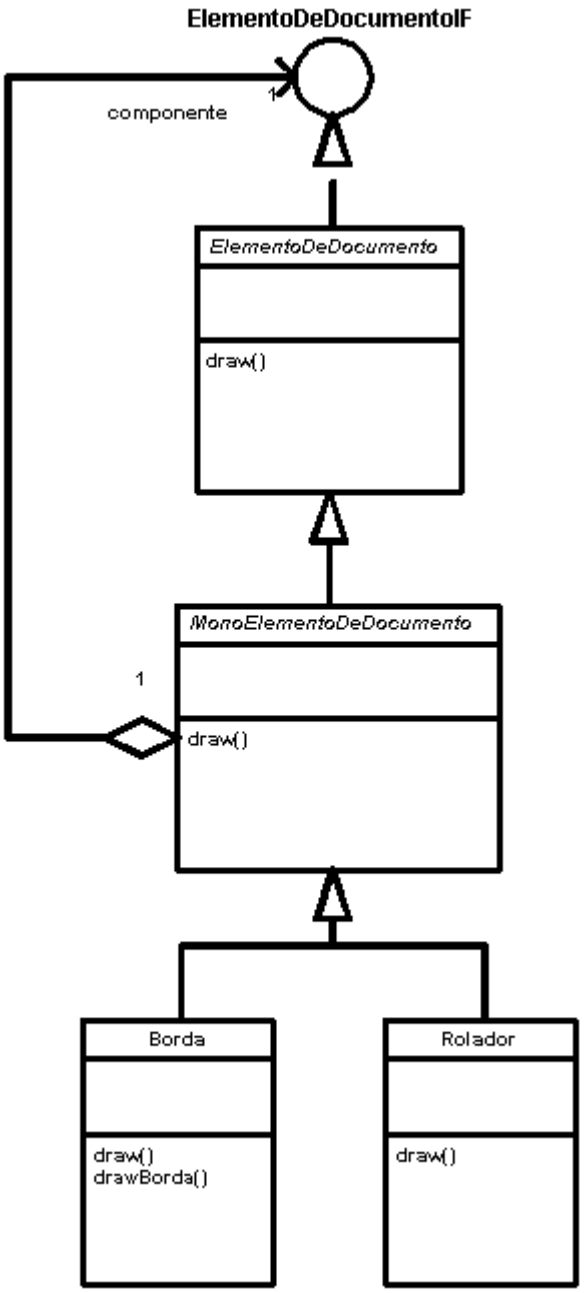
- Fazemos Borda uma subclasse de ElementoDeDocumento para garantir este relacionamento

- Isso também faz sentido porque a borda ter uma aparência gráfica na tela e tudo que tem aparência gráfica é ElementoDeDocumento

- Este conceito chama-se "Inclusão Transparente"
  - Composição com um único filho; e
  - Interfaces compatíveis
- Clientes não sabem se estão tratando do objeto original (ElementoDeDocumento) ou do seu incluidor (Borda)
- O incluidor delega as operações para o objeto incluído, mas aumenta o comportamento fazendo seu trabalho antes ou depois da delegação da operação

A classe MonoElementoDeDocumento

- Definimos uma subclasse de ElementoDeDocumento chamada MonoElementoDeDocumento
- MonoElementoDeDocumento é uma classe abstrata para todos os ElementoDeDocumento de embelezamento

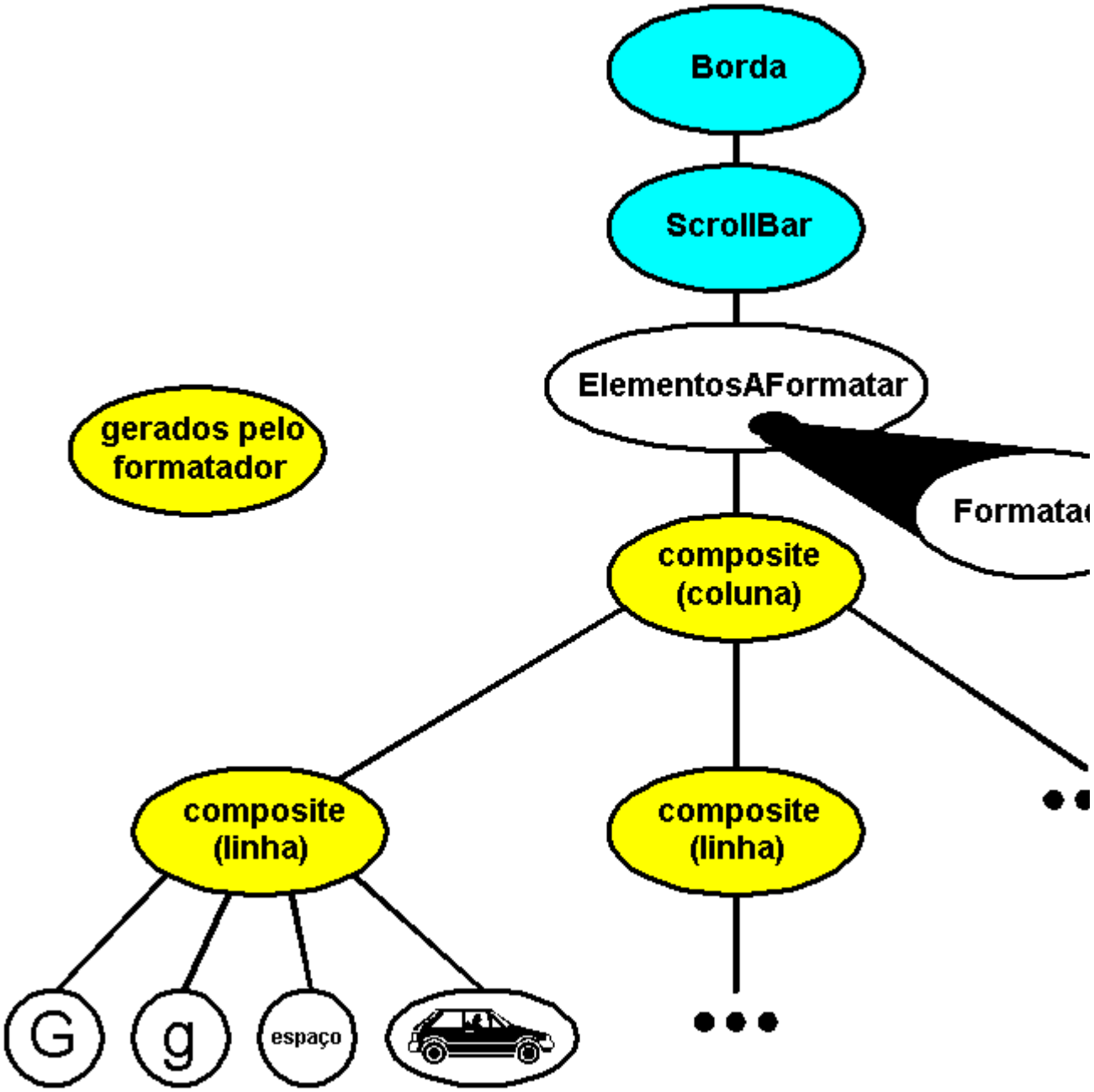


- MonoElementoDeDocumento armazena uma referência a um único ElementoDeDocumento (daí "mono") e encaminha todos os pedidos para ele
  - Isso é chamado *Forwarding* ou Delegação
- Isso faz com que MonoElementoDeDocumento seja completamente transparente aos clientes
- A classe MonoElementoDeDocumento vai reimplementar pelo menos um dos métodos de ElementoDeDocumento para fazer seu trabalho
- Exemplo: Borda redefine o método draw():

```
public abstract class MonoElementoDeDocumento extends Elemen
```

```
    private ElementoDeDocumentoIF componente;
    public void draw() {
        componente.draw();
    }
}
public class Borda extends MonoElementoDeDocumento {
    ...
    public void draw() {
        super.draw();
        drawBorda();
    }
    ...
}
```

- Observe que Borda *estende* o método do pai para desenhar a borda
  - Já que chama o método do pai antes de fazer seu trabalho
- A classe ScrollBar também é um embelezamento
  - Ela desenha seu Componente em posições diferentes dependendo de duas barras de rolagem
  - Ela adiciona as barras de rolagem como embelezamento
  - Ao desenhar seu Componente, ScrollBar pede ao sistema gráfico para fazer "clipping" do Componente nos limites apropriados
    - As partes fora dos limites não aparecerão
- Agora, como adicionar uma Borda *e* um ScrollBar?
  - Compomos os ElementosAFormatar dentro de um ScrollBar e este dentro de uma Borda
    - Se compuséssemos a Borda dentro do ScrollBar, a borda seria rolada também
      - Se isso for o comportamento desejado, essa ordem deve ser usada
  - A estrutura de objetos aparece abaixo



- Observe que os decoradores compõem um único Componente
- Se tentássemos embelezar mais de uma coisa de cada vez, teríamos que misturar vários tipos de ElementoDeDocumento com decoradores
  - Teríamos embelezamento de linha, embelezamento de coluna, etc.
  - O jeito que fizemos parece melhor
- Este padrão chama-se Decorator
  - Embelezamento usando inclusão transparente
  - Observe que o embelezamento pode, na realidade, ser qualquer adição de responsabilidades e não um

"embelezamento visual"

- Ex. embeleza uma árvore de sintaxe com ações semânticas
- Ex. embeleza uma máquina de estados finitos com novas transições

O padrão Decorator

Objetivo

- Adicionar responsabilidades dinamicamente a um objeto
- Decoradores provêem uma alternativa flexível à herança para estender funcionalidade
- Permitem adicionar responsabilidades a um objeto e não a uma classe inteira

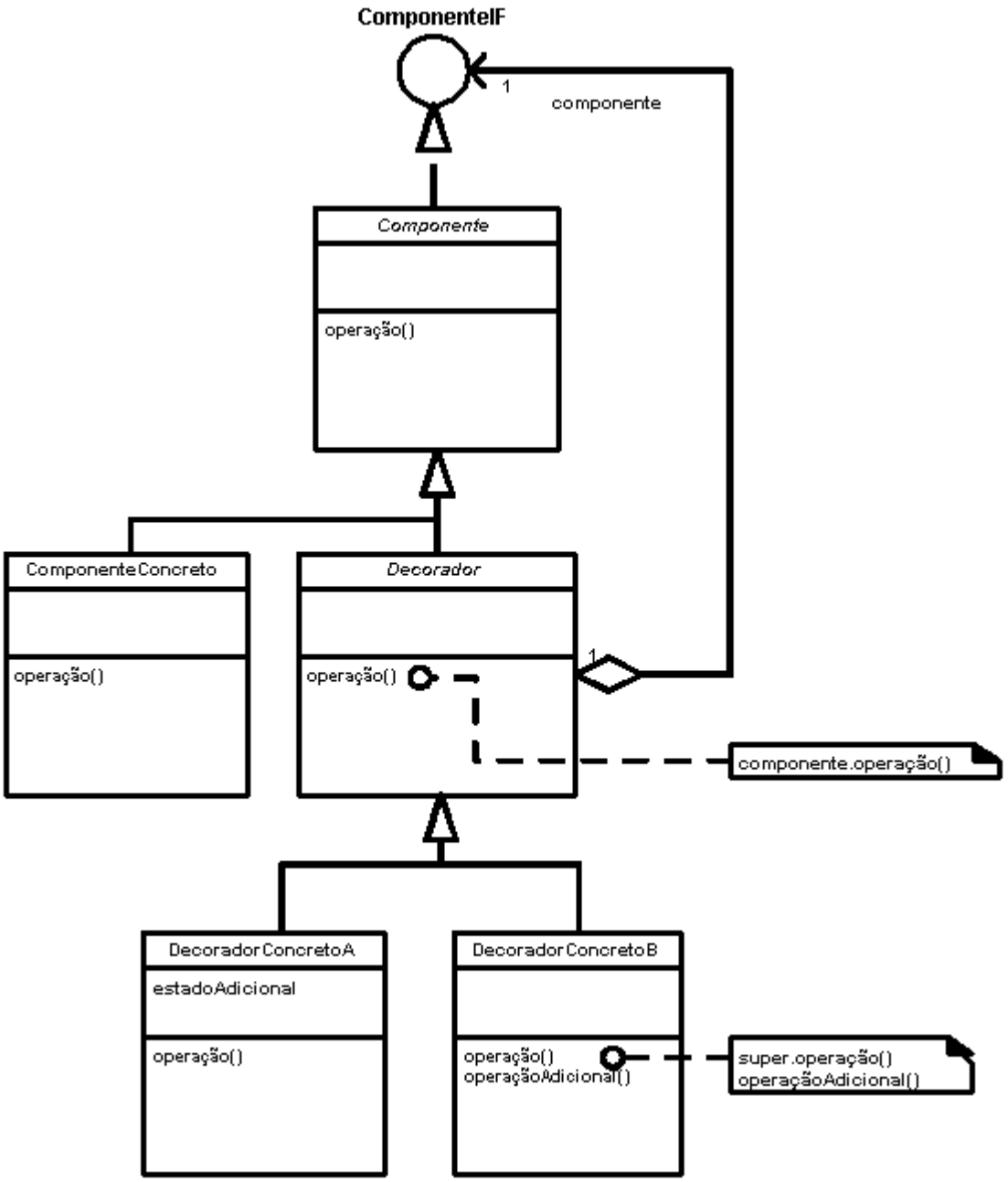
Também chamado de

- Wrapper

Quando usar o padrão Decorator?

- Para adicionar responsabilidades dinamicamente a objetos individuais e transparentemente (sem afetar outros objetos)
- Quando há responsabilidades que podem ser retiradas
- Quando a herança geraria uma explosão de subclasses
- Quando a herança seria uma boa alternativa mas a definição da classe está escondida ou não disponível para herança

Estrutura genérica



Participantes

- **ComponenteIF** (ex. ElementoDeDocumentoIF)
  - Define a interface de objetos que podem ter responsabilidades adicionadas dinamicamente
- **Componente** (ex. ElementoDeDocumento)
  - Possível classe abstrata para fatorar o código comum dos ComponenteConcreto
- **ComponenteConcreto** (ex. ElementosAFormatar)
  - Define um objeto ao qual se pode adicionar

responsabilidades adicionais

- **Decorador** (ex. MonoElementoDeDocumento)
  - Mantém uma referência a um objeto ComponenteIF e define uma interface igual à do Componente
- **DecoradorConcreto** (ex. Borda, Rolador)
  - Adiciona responsabilidades ao Componente

Colaborações entre objetos

- O Decorador encaminha pedidos ao Componente
- Pode opcionalmente adicionar operações antes ou depois deste encaminhamento

Conseqüências do uso do padrão Decorator

- Mais flexível do que herança estática
  - Pode adicionar responsabilidades dinamicamente
  - Pode até adicionar responsabilidades duas vezes! (ex. borda dupla)
- Evita classes com features demais no topo da hierarquia
  - Em vez de tentar prever todos os features numa classe complexa e customizável, permite definir classes simples e adicionar funcionalidade de forma incremental com objetos decoradores
  - Desta forma, as aplicações não têm que pagar pelos features que não usam
- Ponto negativo: o Decorador e o objeto incluso não têm a mesma identidade
  - Portanto, cuidado com a identidade de objetos: um Decorador adicionado a um objeto vai mudar o objeto com o qual o cliente lida (em termos de identidade)
- Ponto negativo: muitos pequenos objetos
  - Fácil de customizar por programadores que entendem o projeto mas pode ser difícil entender e

depurar

Considerações de implementação

- A interface do decorador deve ser igual à interface do objeto sendo decorado
- Se tiver que adicionar apenas uma única responsabilidade, pode-se remover o decorador abstrato
  - Neste caso, o decorador concreto pode tratar do forwarding para o Componente incluso
- Mantendo Componentes enxutos
  - A classe Componente deve ser mantida enxuta e os dados ser definidos nos Componentes concretos
  - Caso contrário, os decoradores (que herdam de Componente) ficam "pesados"

Decorator versus Strategy

- Decorator muda a "pele" de um objeto enquanto Strategy muda as "entranhas"
- Strategy é melhor quando a classe Componente é pesada
  - Neste caso, o decorador seria pesado demais
  - Com Strategy, parte do comportamento de Componente é repassado para outro objeto (que implementa um algoritmo)
  - O Componente "sabe" da existência do(s) objeto(s) de estratégia mas não sabe da existência de decoradores
- Exemplo: bordas com Strategy
  - O Componente pode mandar um objeto Borda (que ele conhece) desenhar a borda
  - O Componente pode manter uma lista de objetos de estratégia para fazer várias coisas, o que é equivalente a ter decoradores recursivos
  - Neste caso, o objeto Borda encapsula a estratégia de desenhar uma borda
- Ver a figura abaixo

funcionalidade estendida com decoradores

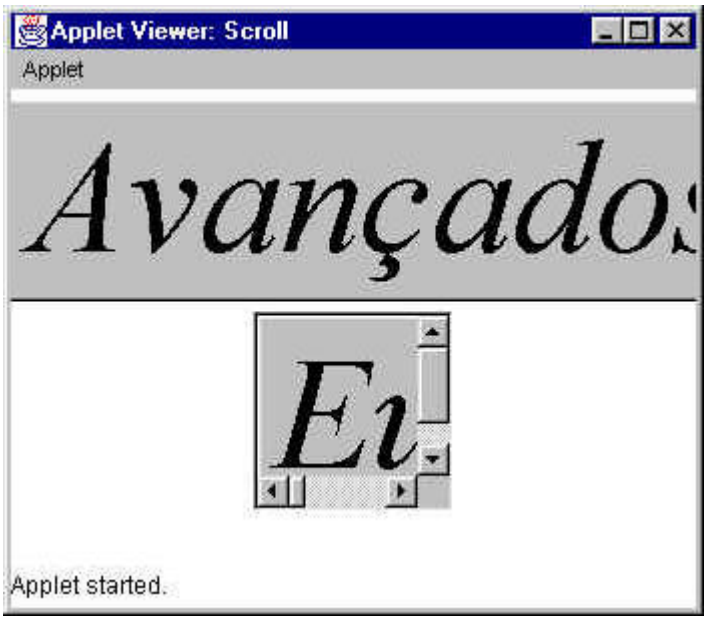
funcionalidade estendida com estratégias

Exemplo de código na API Java

- Uso de scroll bars
- ScrollPane() é um decorador que contém um Component e adiciona barras de rolagem
  - Não usei Swing para poder mostrar o resultado num applet e vários browsers ainda não suportam Swing
  - Para Swing, a classe é um JScrollPane.
- Exemplo de código segue:

```
import java.awt.*;
import java.applet.*;

public class Scroll extends Applet {
    public void init() {
        ScrollPane spane = new ScrollPane();
        Button botaoGrande1 =
            new Button("Eu adoro Metodos Avançados de Progra
        botaoGrande1.setFont(new Font("Serif", Font.ITALIC,
        Button botaoGrande2 =
            new Button("Eu adoro Metodos Avançados de Progra
        botaoGrande2.setFont(new Font("Serif", Font.ITALIC,
        spane.add(botaoGrande1);
        add(botaoGrande2);    // sem scroll bars
        add(spane);           // com scroll bars
    }
}
```



- Applet roda aqui

Referências adicionais

- http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-letters.html
  - JTable usando decoradores para filtrar e ordenar

```
// exemplo 1
TableFilterDecorator filterDecorator =
      new TableHighPriceFilter(new TableSortDecorator(table.

// exemplo 2
TableSortDecorator sortDecorator =
      new TableSortDecorator (new TableHighPriceFilter (tabl
```

- Decorate your Java code

Pergunta final para discussão

- Na seção de implementação do Decorator Pattern, os autores (Gamma et al.) escrevem: *A interface de um objeto decorador deve estar de acordo com a interface do Componente que ele decora*.

  Agora, considere um objeto A decorado pelo objeto B. Haja vista que o objeto B decora o objeto A, o objeto

B compartilha uma interface com o objeto A. Se algum cliente recebe uma instância do objeto decorado (B) e tenta chamar um método em B que não faça parte da interface de A, isto significa que o objeto B não é mais um decorador, no sentido estrito do padrão? Além do mais, porque é importante que a interface de um objeto decorador esteja de acordo com a interface do objeto que ele decora?

- Vê se você acha algo parecido com Decorators na classe Collections de Java. Explique.
- Explique como as classes de entrada/saída de Java usam o padrão Decorator.

programa