Padrão *Composite*

Contexto

A construção de UI (*user interfaces*) é um exemplo de aplicação do padrão *Composite*:

- Componentes da UI (`Component` ou `JComponent`) são armazenados em contentores da UI (`java.awt.Container`);
- Se um painel (instancia de `JPanel`) pode conter componentes então é porque **é um contentor da UI** (subtipo de `java.awt.Container`);
- Mas se um painel por sua vez pode ser adicionado a um contentor da UI, então é porque **também é um componente UI**.

Pode um contentor ser também um elemento desse mesmo contentor?

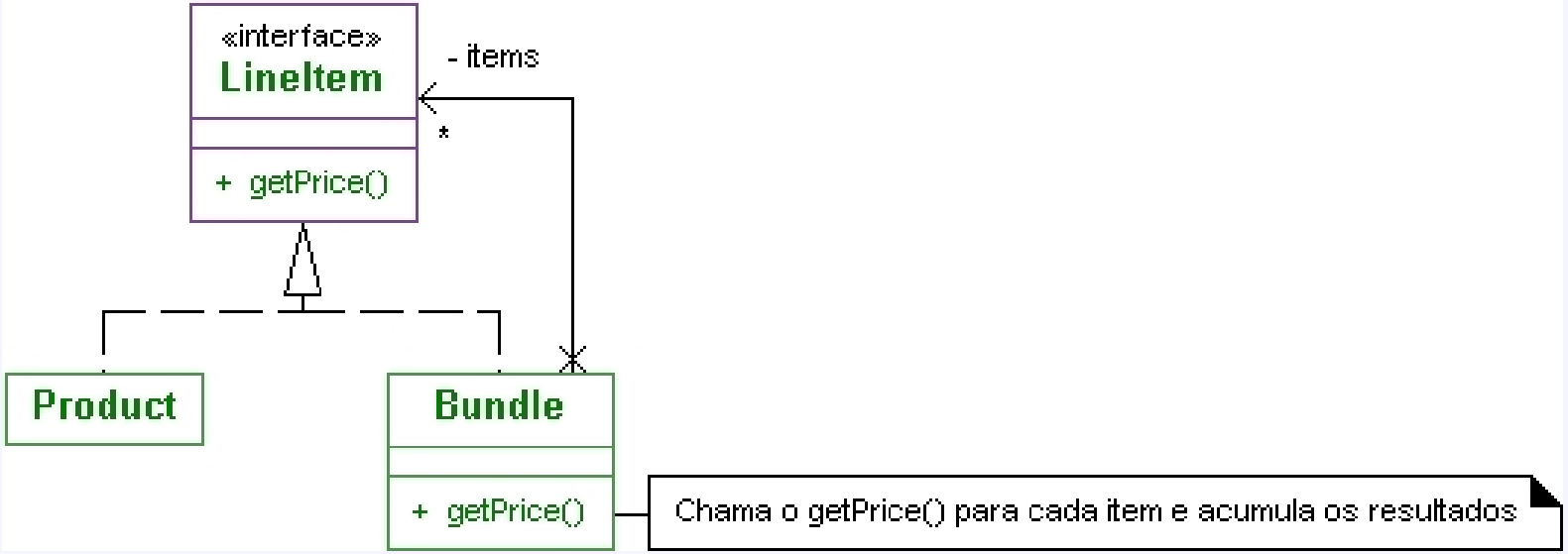
Objectivo:

➔ O padrão ***Composite*** define como se deve **combinar diferentes objectos num único objecto**, que tem o mesmo comportamento que cada uma das suas partes.

Solução

- Uma das características deste padrão é a forma como um método do objecto composto faz o seu trabalho. Este deve aplicar o método a todos os seus objectos primitivos e combinar os resultados.

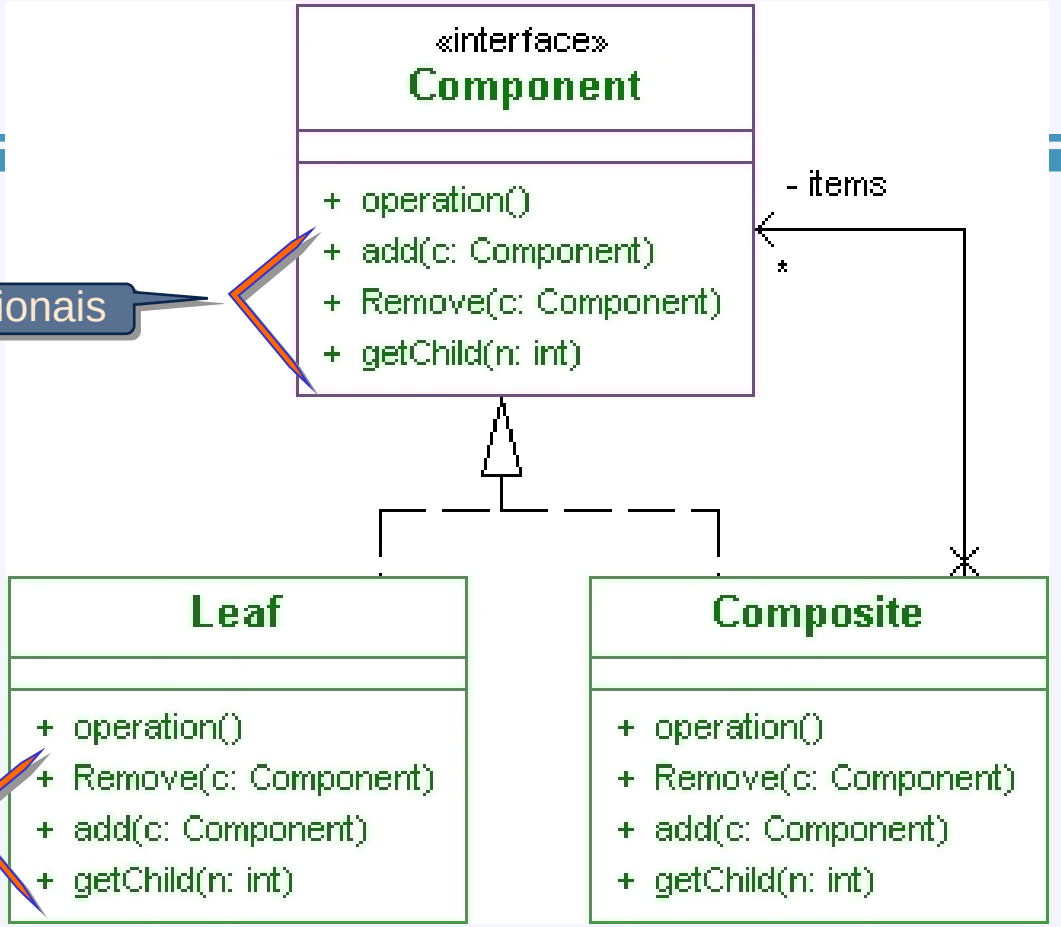
Exemplo 1:



Exemplo 2:

- Outro exemplo similar é a computação do tamanho preferível de um contentor (`getPreferredSize()`). O contentor tem que obter o tamanho preferível de todos os seus componentes e combinar os resultados.

Padrão *Composite*



Característica:

➔ A implementação de `operation()` no tipo composto deve aplicar esse método em todos os componentes que o constituem e combinar os seus resultados.
Exemplo: em `Bundle` os métodos `getPrice()`.

Padrão *Composite* - Participantes

Nome do Participante	Descrição
`Component` (`java.awt.Component`)	Abstracção sobre os componentes que podem ser contentores (`Composite`) ou uma parte de um contentor sem filhos (`Leaf`).
`Composite` (`java.awt.Container`)	• Tem uma colecção de objectos individuais (`Component`); • A implementação de uma operação da interface `Component` deve ser aplicado a cada um dos objectos contidos e combinar os seus resultados.
`Leaf` (`JButton, JTextArea, etc`)	Um objecto individual sem filhos.
`operation()` (`getPreferredSize(), etc`)	Método especificado pela interface `Component` e implementado por `Leaf` e `Composite`.

Padrão *Composite*

Característica	Descrição
Nome	*Composite*
Categoria	Estrutura – Objectos
Objectivo	Compõe objectos numa estrutura em árvore que representa uma hierarquia do tipo parte/todo. Os clientes tratam os objectos individuais e compostos uniformemente.
Aplicabilidade	• Objectos individuais podem ser combinados em objectos compostos. • Os clientes podem ignorar as diferenças entre objectos compostos e individuais.
Nome alternativo	...

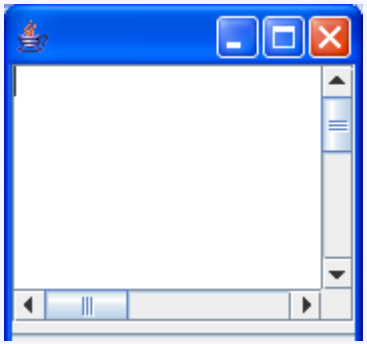
Padrão *Decorator*

Contexto

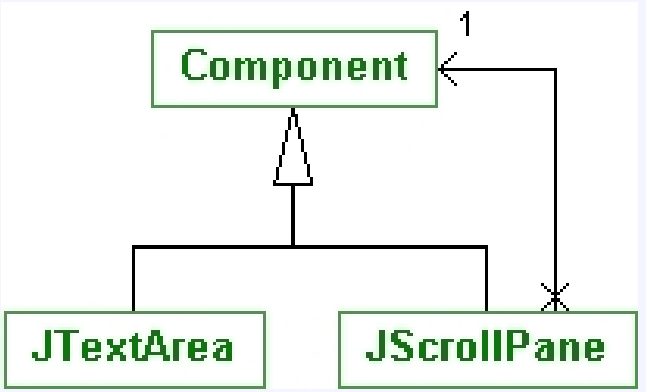
- Quando um contentor contém mais informação que aquela que é possível apresentar na sua área visível é necessário acrescentar-lhe *scroll bars*.
- As *scroll bars* fazem todo o sentido em componentes como áreas de texto, listas, etc, mas outros componentes, como por exemplo uma tabela, podem não fazer sentido ter *scroll bars*.

Exemplo de adição de ***scroll bars*** a uma caixa de texto:

```
JTextArea textArea = new JTextArea(20, 40);
JScrollPane scroller = new JScrollPane(textArea);
```



- Porque as *scroll bars* adicionam funcionalidade à caixa de texto, estas são entendidas como uma "**decoração**"



Solução

- Qualquer componente pode ser decorado com um `ScrollPane`, não apenas as áreas de texto.
- Um aspecto chave de `ScrollPane` é que este pode **decorar qualquer componente, continuando a ser um componente**. Como tal toda a funcionalidade de `Component` ainda se aplica às *scroll bars*.
- Outro aspecto chave do padrão *Decorator* é que **o componente "decorado" é completamente passivo**. A caixa de texto não tem que conhecer, nem fazer nada para adquirir as *scroll bars*.

➔ O padrão ***Decorator*** define como implementar **uma classe que adiciona funcionalidade a outra classe, mantendo a sua interface**.

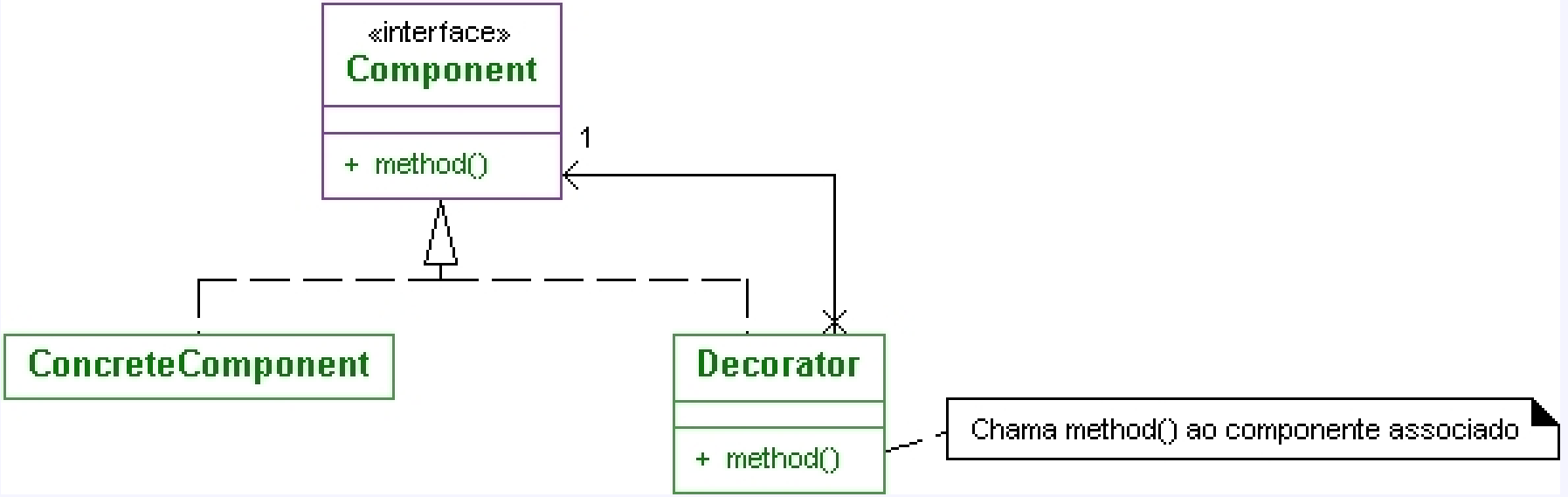
Alternativa desvantajosa:
Outra opção no desenho da caixa de texto era fazer com que a classe `JTextArea` fosse responsável por fornecer as *scroll bars* (é o caso da classe `TextArea` do *package* `java.awt`). Desvantagens:

- **Repetição de código** por todos os componentes que disponibilizem o mesmo mecanismo ou perigo de **"explosão do número" de tipos** (exemplo hipotético: `ListMagnifiable`, `ListScrollable`, etc).
- **Impossibilidade de prever futuras decorações** que se pretendam adicionar aos componentes.

Padrão *Decorator* - Estrutura

- A implementação de um método da *interface* `Component` na classe `Decorator` implica chamar esse mesmo método ao componente associado e combinar o resultado com o efeito da decoração.

  Exemplo: o método `paint` de `ScrollPane` pinta o componente decorado chamando-lhe o `paint` e acrescenta o desenho das *scroll bars*.



Padrão *Decorator* - Participantes

Nome do Participante	Descrição
`Component` (`java.awt.Component`)	Interface que define uma abstracção sobre um componente.
`Decorator` (`JScrollPane`)	• Objecto que manipula um componente e decora-o de uma dada forma; • Quando implementa um método da interface `Component`, deve aplicar esse método ao componente decorado e combinar o resultado com o efeito da decoração.
`ConcreteComponent` (`JTextArea, JList, etc`)	Implementação concreta de `Component`.
`method()` (`paint(), etc`)	Método especificado pela interface `Component`

Padrão *Decorator*

Característica	Descrição
Nome	*Decorator*
Categoria	Estrutura – Objectos
Objectivo	Define como implementar **uma classe que adiciona funcionalidade a outra classe, mantendo a sua interface**.
Aplicabilidade	• Adicionar funcionalidade ao comportamento de uma classe, identificada por `Component`; • O componente decorado continua a ser usado da mesma forma que um simples componente; • **O componente não tem responsabilidades na decoração**; • Mantém em aberto a possibilidade de criar no futuro novas decorações que possam ser adicionadas a este componente.
Nome alternativo	...

Exemplo: padrão *Decorator* no *package* ***io***

- Outro exemplo de aplicação do padrão *Decorator* é o conjunto de "**filtros**" da biblioteca de IO do Java.

  Exemplo:
  - A classe `Reader` suporta as operações básicas de leitura, tais como ler um único carácter ou um *array* de caracteres.
  - A classe `FileReader` implementa os mesmo métodos, lendo os caracteres a partir de um ficheiro, contudo continua a não disponibilizar um método para ler uma linha completa.

➔ A classe `BufferedReader` pode adicionar a qualquer `Reader` a capacidade de ler linha a linha. O seu método `readLine` mantém a chamada ao método `read` do seu `Reader` interno até que encontre o carácter de fim de linha. Nessa altura retorna a instância de `String` que tem todos os caracteres lidos.

Exemplo de utilização:

```
BufferedReader in = new BufferedReader(new FileReader("input.txt"));
String firstLine = in.readLine();
```

Exemplo: padrão *Decorator* no *package* ***io***…

Nome do Participante	Nome no java.io
`Component`	`Reader`
`Decorator`	`BufferedReader`
`ConcreteComponent`	`FileReader`
`method()`	`read()`

Atenção

Em GUI nem tudo o que serve de "decoração" segue o padrão *decorator*.
Ex: Border.