Composite

Pronunciação

- Pronunciação americana: compÓsit
- Pronunciação canadense (Britânica): cÓmposit

Um problema a resolver: editor de documentos

- Para introduzir este padrão (e alguns outros), usaremos o exemplo do projeto de um editor de documentos *WYSIWYG* (What You See Is What You Get)
  - Semelhante a Word, por exemplo
  - Outros exemplos do padrão Composite no Swing de Java
- O editor pode misturar texto e gráficos usando várias opções de formatação
- A redor da área de edição estão os menus, scroll bars, barras de ferramentas, etc.
- O primeiro problema de design que queremos atacar é como representar a *estrutura* do documento
  - Essa estrutura afeta o resto da aplicação já que a edição, formatação, análise textual, etc. deverão acessar a representação do documento
- Um documento é um arranjo de elementos gráficos básicos
  - Caracteres, linhas, polígonos e outras figuras
- O usuário normalmente não pensa em termos desses elementos gráficos mas em termos de estruturas físicas
  - Linhas, colunas, figuras, tabelas e outras subestruturas
  - As subestruturas podem ser compostas de outras subestruturas, etc.
- O usuário também pode pensar na estrutura lógica

(frase, parágrafos, seções, etc.)

- Não vamos considerar isso aqui mas a solução que usaremos se aplica a esta situação também

- A interface do usuário (UI) do editor deve permitir que o usuário manipule tais subestruturas diretamente
  - Por exemplo, o usuário pode tratar um diagrama como uma unidade em vez de uma coleção de elementos gráficos primitivos
  - O usuário deve manipular uma tabela como uma unidade e não como um amontoado de texto e gráficos
- Para permitir tal manipulação, usaremos uma representação interna que case com a estrutura física do documento
- Portanto, a representação interna deve suportar:
  - A manutenção da estrutura física do documento (o arranjo de texto e gráficos em linhas, colunas, tabelas, ...)
  - A geração e apresentação visual do documento
  - Mapear posições da tela para elementos da representação interna. O editor vai saber para o que o usuário está apontando
- Tem algumas outras restrições no projeto
  - Texto e gráficos devem ser tratados uniformemente
    - A interface deve permitir embutir texto em gráficos e vice-versa
    - Um gráfico não deve ser tratado como caso especial de texto, nem texto como caso especial de gráfico, senão teremos mecanismos redundantes de manipulação, formatação, etc.
  - A implementação não deve ter que diferenciar entre elementos únicos e grupos de elementos na representação interna
    - O editor deve tratar elementos simples e complexos de forma uniforme permitindo assim documentos arbitrariamente complexos
    - O décimo elemento da linha 5, coluna 2 pode ser um caractere único ou um diagrama complexo

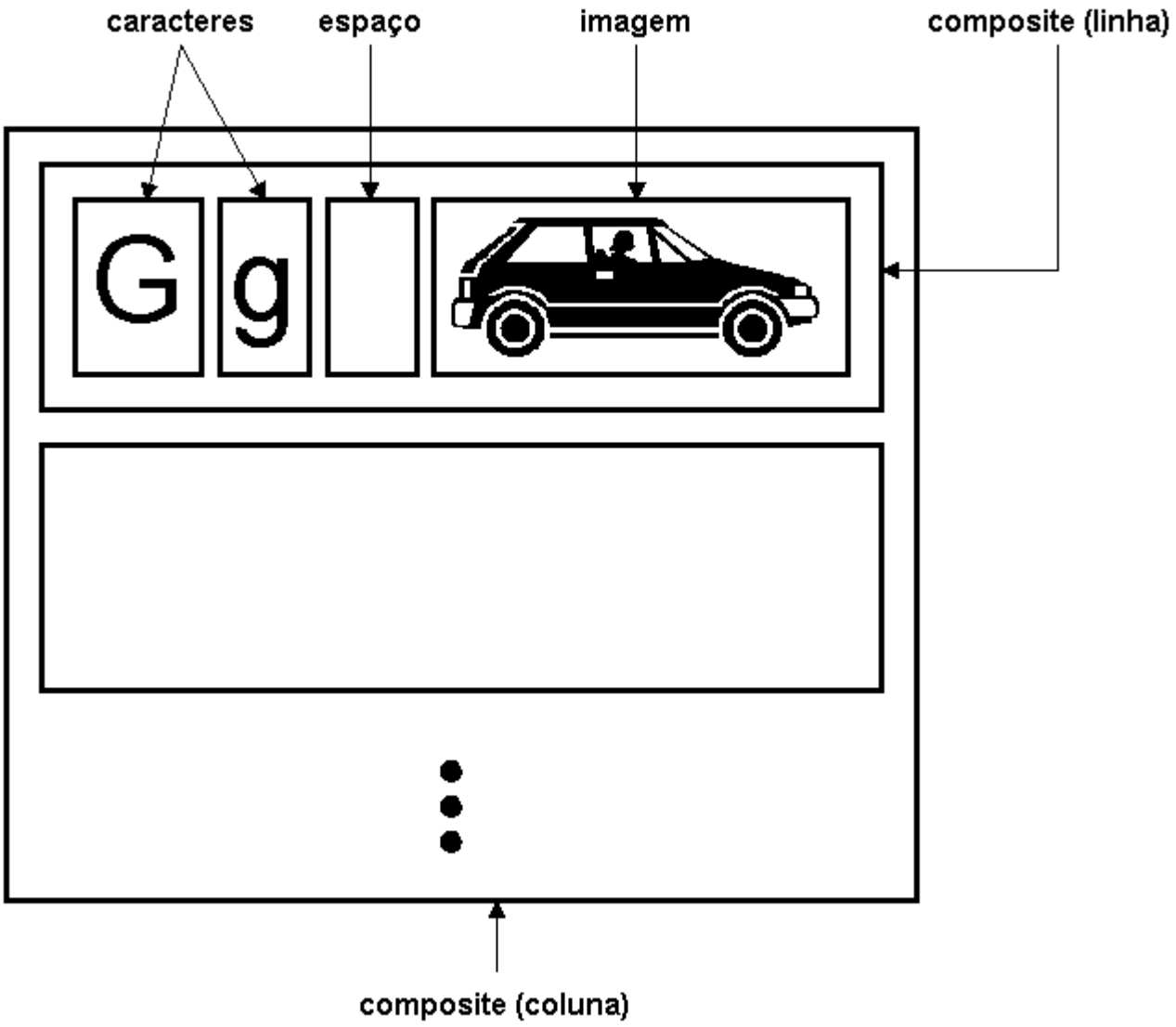
com muitos elementos

- Basta que o elemento saiba se desenhar, possa dar suas dimensões: ele pode ter qualquer complexidade

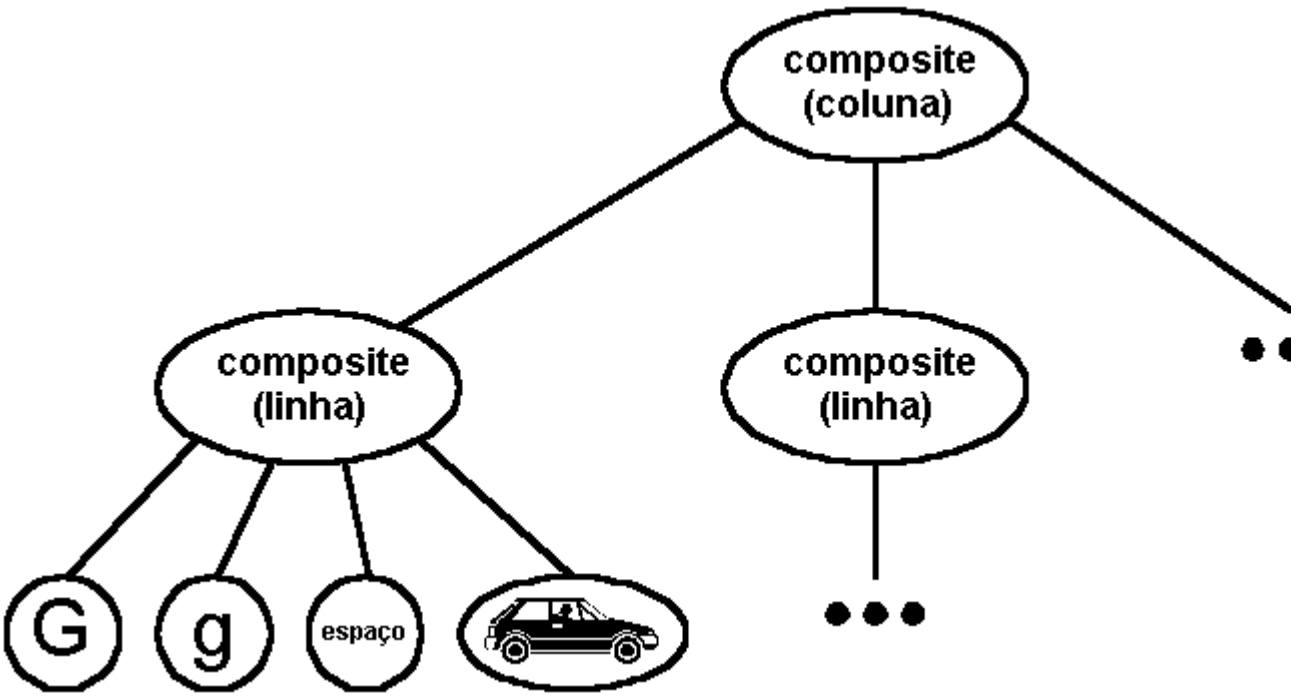
- Por outro lado, queremos analisar o texto para verificar a grafia, hifenizar, etc. e não podemos verificar a grafia de gráficos ou hifenizá-los

Composição recursiva

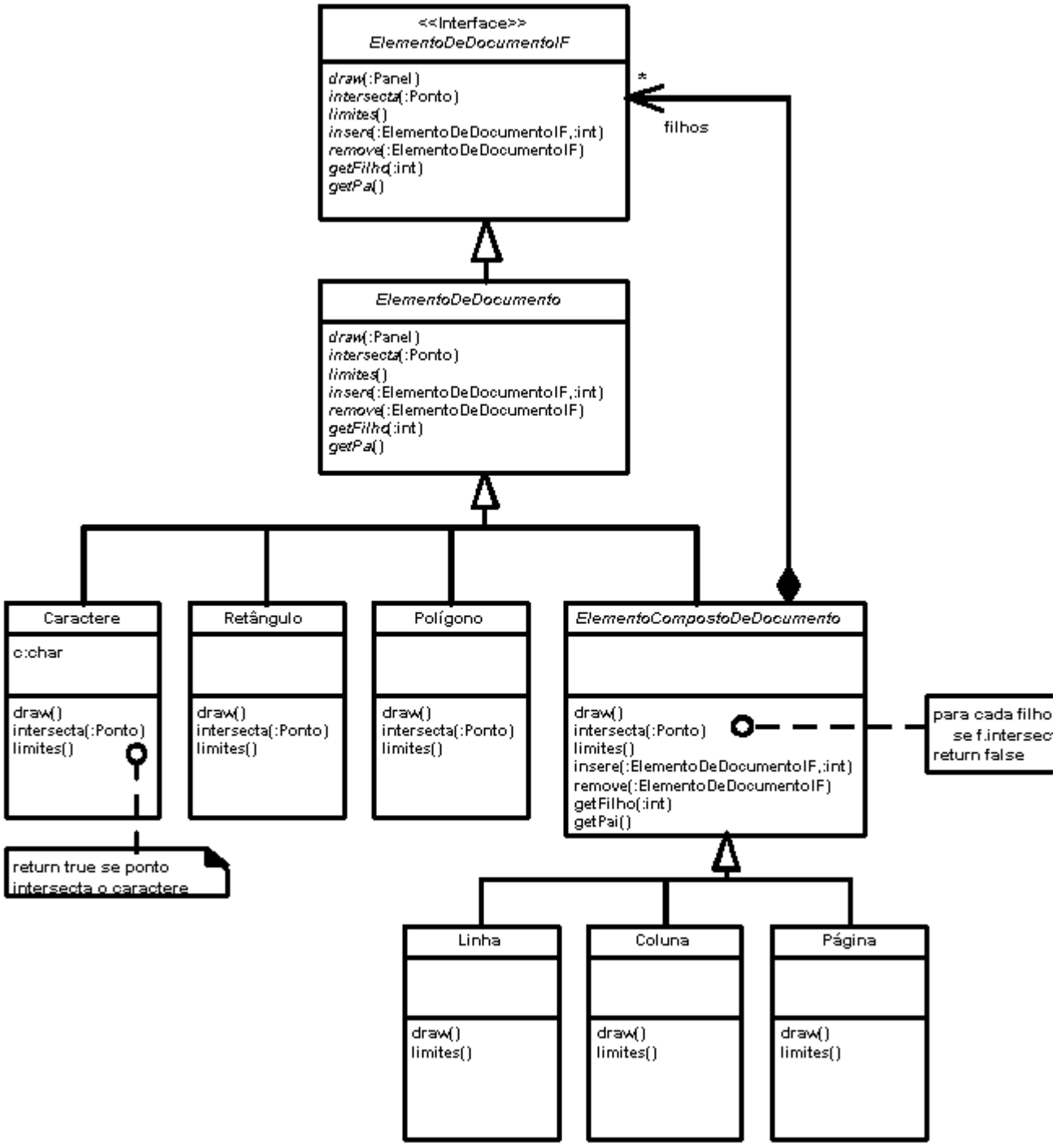
- Uma forma comum de representar informação estruturada hierarquicamente é através da *Composição Recursiva*
  - Permite construir elementos complexos a partir de elementos simples
- Aqui, a composição recursiva vai permitir compor um documento a partir de elementos gráficos simples
  - Começamos formando linhas a partir de elementos gráficos simples (caracteres e gráficos)
  - Múltiplas linhas formam colunas
  - Múltiplas colunas formam páginas
  - Múltiplas páginas formam documentos
  - etc.



- Podemos representar essa estrutura física usando um objeto para cada elemento
  - Isso inclui elementos visíveis e elementos estruturais (linhas, colunas)
  - A estrutura de objetos seria como abaixo
    - Na prática, talvez um objeto não fosse usado para cada caractere por razões de eficiência



- Podemos agora tratar texto e gráficos uniformemente
- Podemos ainda tratar elementos simples e compostos uniformemente
- Teremos que ter uma classe para cada tipo de objeto e essas classes terão que ter a mesma interface (para ter uniformidade de tratamento)
  - Uma forma de ter interfaces compatíveis é de usar herança
- Definimos uma interface "ElementoDeDocumentoIF" e uma classe abstrata "ElementoDeDocumento" para todos os elementos que aparecem na estrutura de objetos
  - Suas subclasses definem elementos gráficos primitivos (caracteres e gráficos) e elementos estruturais (linhas, colunas, frames, páginas, documentos, ...)
  - Parte da hierarquia de classes segue abaixo



- Elementos de documentos têm três responsabilidades
  - Sabem se desenhar (draw)
  - Sabem o espaço que ocupam (limites)
  - Conhecem seus filhos e pai
- Subclasses de ElementoDeDocumento devem redefinir certas operações tais como draw()
- Um ElementoDeDocumentoIF pai deve freqüentemente saber quanto espaço seus filhos ocupam para posicioná-los de forma a não haver sobreposição

- Isso é feito com o método limites() que retorna o retângulo que contém o elemento
- A operação intersecta() serve para saber se um ponto intersecta o elemento
  - Usado quando o usuário clica com o mouse
- A classe abstrata ElementoCompostoDeDocumento implementa certos métodos comuns através da aplicação sucessiva do método aos filhos
  - É o caso de intersecta()
- Todos os elementos têm uma mesma interface para gerenciar os filhos (inserir, remover, etc.)
- Este padrão de projeto chama-se Composite e será discutido mais detalhadamente agora

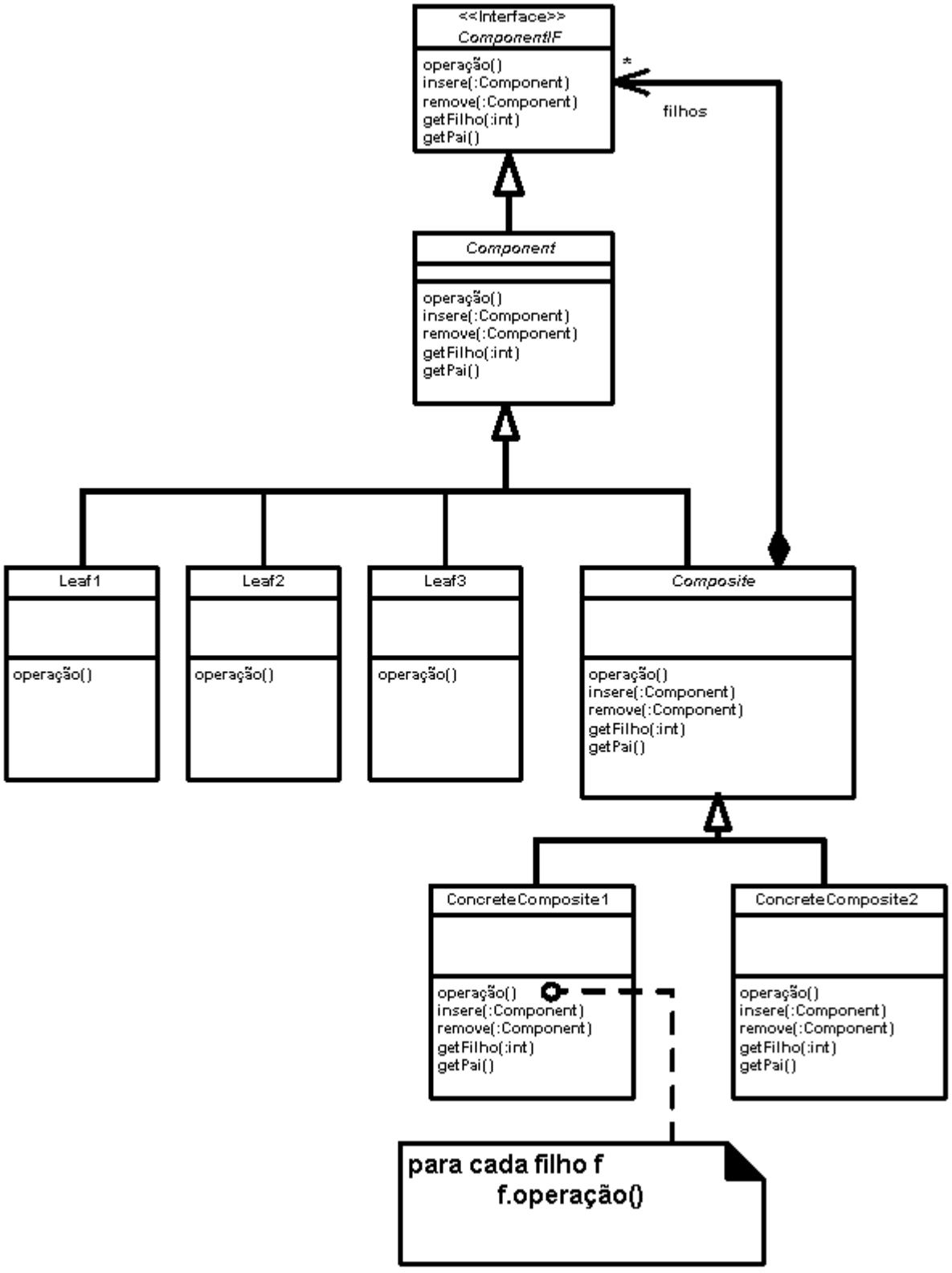
O Padrão Composite

Objetivo

- Compor objetos em estruturas em árvore para representar hierarquias Parte-Todo
- Composite permite que clientes tratem objetos individuais e composições uniformemente

Participantes

- Os nomes genéricos dados às classes abstratas são **Component** e **Composite**
- Os nomes genéricos dados às classes concretas são **Leaf** e **ConcreteComposite**



Conseqüências do uso de Composite

- Objetos complexos podem ser compostos de objetos mais simples recursivamente
  - O cliente pode tratar objetos simples ou compostos da mesma forma: simplifica o cliente

- Facilita a adição de novos componentes: o cliente não tem que mudar com a adição de novos objetos (simples ou compostos)
  - Exemplo: no editor, como adicionar "Comentários" do tipo Word? O que pode estar contido num comentário?
- Do lado negativo, o projeto fica geral demais
  - É mais difícil restringir os componentes de um objeto composto
  - Por exemplo, acima, podemos compor linhas com linhas ou com documentos, etc. o que não faz sentido
  - O sistema de tipagem da linguagem não ajuda a detectar composições erradas
    - A solução é verificar em tempo de execução

Considerações de implementação

- Adicionar referência ao pai de um objeto pode simplificar o caminhamento na estrutura
  - Onde adicionar a referência ao pai? Normalmente é colocada na classe abstrata Component
    - Exercício: colocar esta referência na figura acima
  - As subclasses herdam a referência e os métodos que a gerenciam
- Compartilhamento de componentes
  - Útil para reduzir as necessidades de espaço
  - Por exemplo, caracteres iguais poderiam compartilhar objetos
  - Fazer isso complica se os componentes só puderem ter um único pai
  - O padrão "flyweight" mostra como resolver a questão
- Maximização da interface de Component
  - Exercício para casa: certos livros colocam os métodos de gerenciamento de filhos apenas na classe Composite porque uma folha não tem filhos!

- Você concorda ou discorda com a maximização da interface de Component? Por quê? Em outras palavras, é melhor ter uma interface idêntica para folhas e Composite (transparência para o cliente) ou interfaces diferentes (segurança de não fazer besteiras como adicionar um filho a uma folha, o que seria capturado pelo compilador)?
- Se mantivermos as interfaces de Component e Composite diferentes, como o cliente pode testar se um objeto é folha ou composto?

- Onde são armazenados os filhos?
  - Nós os colocamos em Composite mas eles poderiam ser colocados em Component
  - A desvantagem é a perda de espaço para essa referência para folhas
- Quando os filhos devem ter uma ordem especial, deve-se cuidar deste aspecto
  - Usar um *iterator* é uma boa idéia
- Cache de informação
  - As classes Composite podem manter em cache informação sobre seus filhos de forma a eliminar (curto-circuitar) o caminhamento ou pesquisa nos filhos
  - Um exemplo: um Composite poderia manter em cache os limites do conjunto de filhos de forma a não ter que recalcular isso sempre
    - Quando um filho muda, a cache deve ser invalidada
    - Neste caso, os filhos devem conhecer o pai para avisar da mudança
- Como armazenar os filhos?
  - ArrayList, LinkedList, HashMap (qualquer coleção razoável)

Uso do padrão na API Java: O pacote java.awt.swing

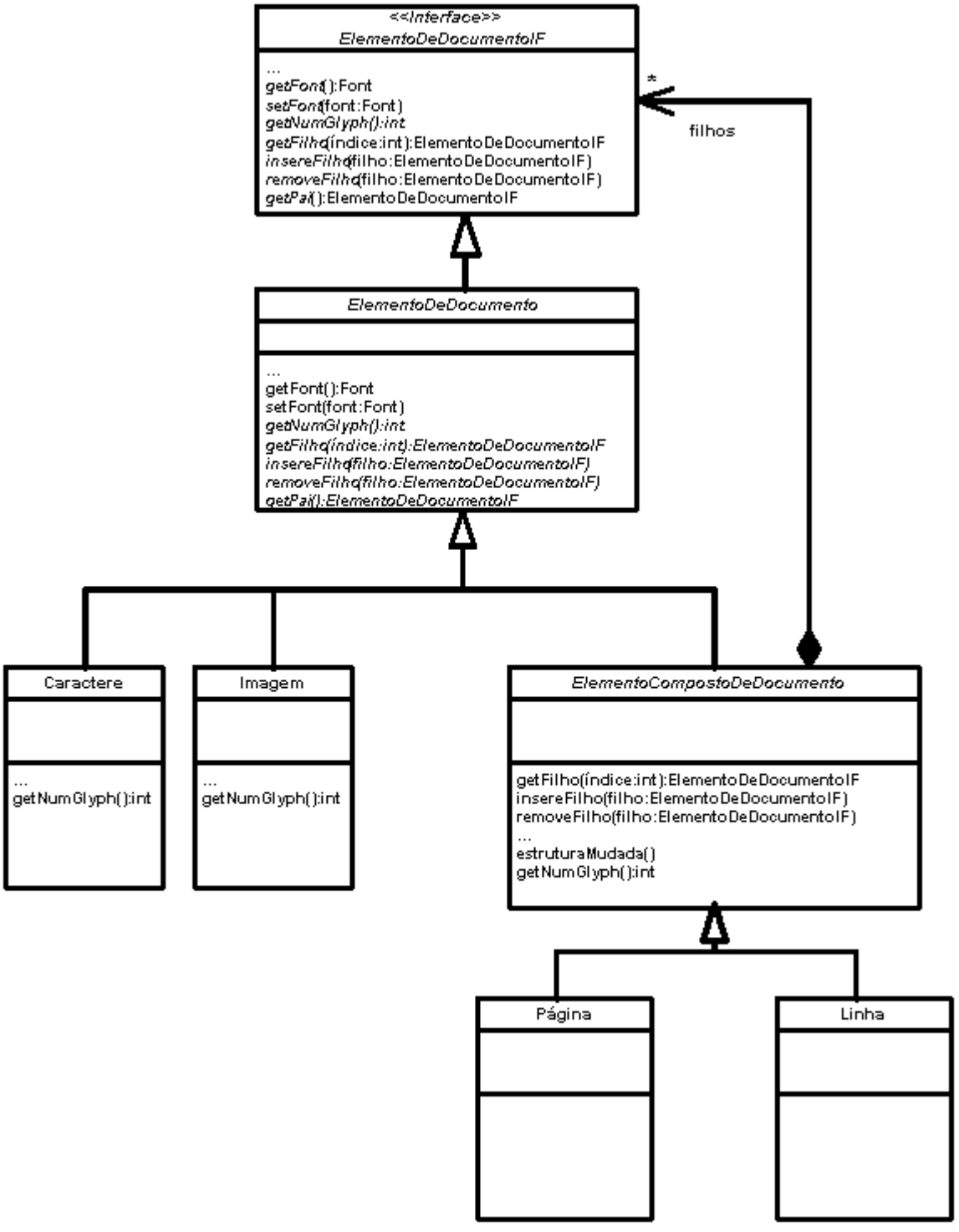
- Usa o padrão Composite com a classe abstrata

"Component" e o Composite sendo a classe "Container"

- Folhas podem ser Label, TextField, Button
- Composites concretos são Panel, Frame, Dialog

Exemplo de código: um editor de documentos

- O diagrama de classes segue



```
interface ElementoDeDocumentoIF {
    ElementoDeDocumentoIF getPai();
    /**
     * Retorna o font associado a este objeto.
     * Se não houver font, retorna o font do pai.
     * Se não houver pai, retorna null.
     */
    Font getFont();

    /**
     * Associa um font a este objeto.
     */
    void setFont(Font font);

    /**
     * Retorna o número de glyphs que este objeto contém.
     */
    int getNumGlyph();

    /**
     * Retorna o filho deste objeto na posição dada.
     */
    ElementoDeDocumentoIF getFilho(int índice)
            throws ÉFolhaException;

    /**
     * Faça o ElementoDeDocumentoIF dado um filho deste obje
     */
    void insereFilho(ElementoDeDocumentoIF filho)
            throws ÉFolhaException;

    /**
     * remove o ElementoDeDocumentoIF dado
     * da lista de filhos deste objeto.
     */
    void removeFilho(ElementoDeDocumentoIF filho)
            throws ÉFolhaException;
} // interface ElementoDeDocumentoIF

abstract Class ElementoDeDocumento
            implements ElementoDeDocumentoIF {
    // Este é o font associado ao objeto
    // Se for nulo, o font é herdado do pai
    private Font font;

    // o container deste objeto
```

```
    ElementoDeDocumentoIF pai;
    ...
    public ElementoDeDocumentoIF getPai() {
        return pai;
    }

    public Font getFont() {
        if(font != null) {
            return font;
        } else if(pai != null) {
            return pai.getFont();
        } else {
            return null;
        }
    } // getFont()

    public void setFont(Font font) {
        this.font = font;
    } // setFont()

    public abstract int getNumGlyph();

    public ElementoDeDocumentoIF getFilho(int índice)
            throws ÉFolhaException {
        throw new ÉFolhaException("Folha não tem filhos");
    }

    /**
     * Faça o ElementoDeDocumentoIF dado
     * um filho deste objeto.
     */
    public void insereFilho(ElementoDeDocumentoIF filho)
            throws ÉFolhaException {
        throw new ÉFolhaException("Folha não tem filhos");
    }

    /**
     * Remove o ElementoDeDocumentoIF dado
     * da lista de filhos deste objeto.
     */
    public void removeFilho(ElementoDeDocumentoIF filho) {
            throws ÉFolhaException {
        throw new ÉFolhaException("Folha não tem filhos");
    }
} // class ElementoDeDocumento

abstract class ElementoCompostoDeDocumento
            extends ElementoDeDocumento
```

```
        implements ElementoDeDocumentoIF {
// Os filhos deste objeto
private Collection filhos = new Vector();

// Valor em cache de getNumGlyph
private int cacheNumGlyph;
// Validade de cacheNumGlyph
private boolean cacheValida = false;

/**
 * Retorna o filho deste objeto na posição dada.
 */
public ElementoDeDocumentoIF getFilho(int índice) {
    return (ElementoDeDocumentoIF)filhos.get(índice);
} // getFilho()

/**
 * Faça o ElementoDeDocumentoIF dado
 * um filho deste objeto.
 */
public synchronized void insereFilho(ElementoDeDocumento
    synchronized(filho) {
        filhos.add(filho);
        filho.pai = this;
        estruturaMudada();
    } // synchronized
} // insereFilho()

/**
 * Remove o ElementoDeDocumentoIF dado
 * da lista de filhos deste objeto.
 */
public synchronized void removefilho(ElementoDeDocumento
    synchronized(filho) {
        if(this == filho.pai) {
            filho.pai = null;
            filhos.remove(filho);
            estruturaMudada();
        }
    } // synchronized
} // removeFilho()
...
/**
 * Uma chamada a esta função significa que um filho mudo
 * o que invalida a cache de informação que o objeto man
 * sobre seus filhos.
 */
public void estruturaMudada() {
```

```
        cacheValida = false;
        if(pai != null) {
            pai.estruturaMudada();
        }
    } // estruturaMudada()

    /**
     * Retorna o número de glyphs que este objeto contém.
     */
    public int getNumGlyph() {
        if(cacheValida) {
            return cacheNumGlyph;
        }
        cacheNumGlyph = 0;
        for(int i = 0; i < filhos.size(); i++) {
            cacheNumGlyph += ((ElementoDeDocumentoIF)filhos.
        } // for
        cacheValida = true;
        return cacheNumGlyph;
    } // getNumGlyph()
} // class ElementoCompostoDeDocumento

class Caractere extends ElementoDeDocumento
            implements ElementoDeDocumentoIF {
    ...
    /**
     * Retorna o número de glyphs que este objeto contém.
     */
    public int getNumGlyph() {
        return 1;
    } // getNumGlyph()
} // class Caractere

class Imagem extends ElementoDeDocumento
            implements ElementoDeDocumentoIF {
    ...
    /**
     * Retorna o número de glyphs que este objeto contém.
     */
    public int getNumGlyph() {
        return 1;
    } // getNumGlyph()
} // class Imagem

class Página extends ElementoCompostoDeDocumento {
            implements ElementoDeDocumentoIF {
    ...
```

```
} // Página
```

- Alguns comentários
  - getFont() usa a informação de seu pai
  - Um objeto composto usa uma cache para saber quantos glyphs compõem o objeto
  - synchronized é usado ao mexer com dados que podem ser compartilhados para possibilitar uma implementação multi-threaded

Perguntas finais para discussão

- Você usaria o padrão Composite se não tivesse uma hierarquia parte-todo? Em outras palavras, se apenas alguns objetos têm filhos e quase todos os outros objetos de uma coleção são folhas (uma folha não tem filho), você ainda usaria o padrão Composite para modelar tais objetos?

Referências adicionais

- **A look at the Composite design pattern**

programa