Mestrado em Sistemas Moveis

IHC - modulo 2

António Nestor Ribeiro

Universidade do Minho

2004/2005

Design Patterns

- **Patterns de criação**

Objectivos:

- abstrair o processo de instanciação.
- tornar o sistema independente da forma como os objectos são criados, representados e compostos
- adaptar-se ao facto de os sistemas, ao tornarem-se mais complexos, dependerem cada vez mais da composição do que da herança

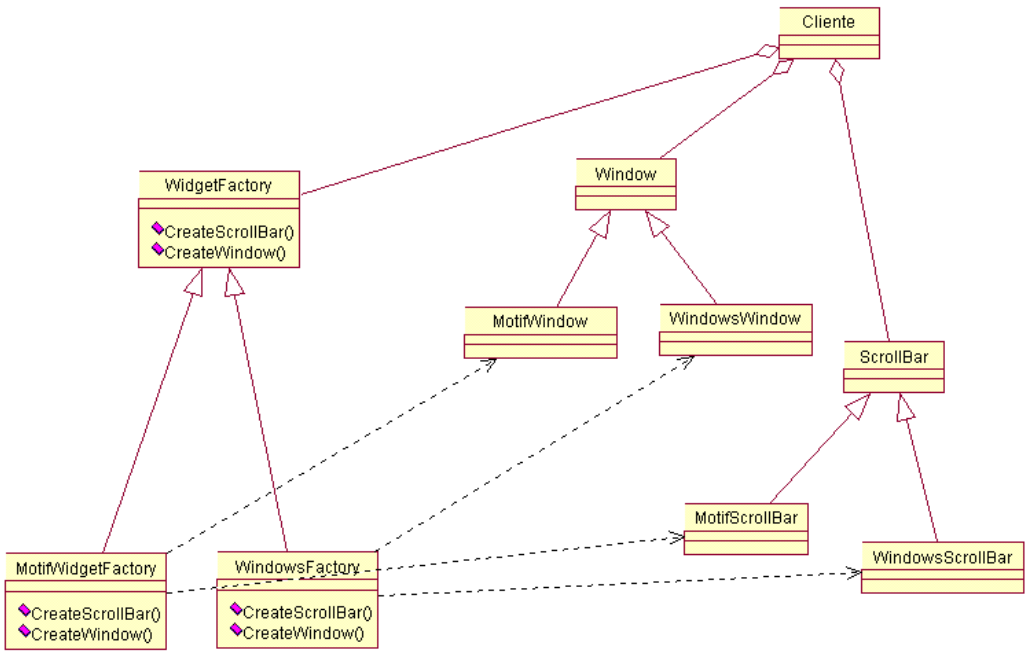
O que fornecem:

- encapsulam conhecimento de que concretização de classes é que o sistema usa
- capacidade de esconder a forma como as instâncias são criadas e compostas

Design Patterns - Abstract Factory

Objectivo: Fornecer um interface para a criação de famílias de objectos sem conhecer a sua implementação concreta.

Motivação: Considere-se um exemplo da implementação de um sistema gráfico que deve funcionar em mais do que um ambiente (ex: Windows, Mac, Motif, etc).

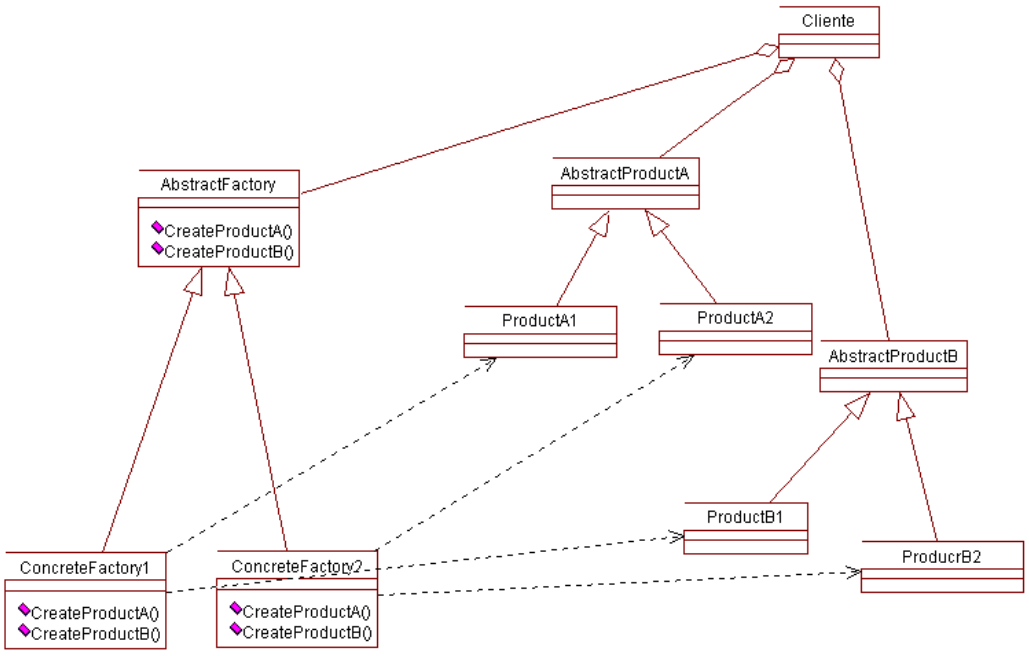


Design Patterns - Abstract Factory (cont.)

Aplicação: usar quando:

- um sistema deva ser independente da maneira como os seus produtos são criados e compostos
- quando o sistema deva ser configurado para mais do que uma família de produtos
- quando se quer fornecer uma biblioteca de classes de produtos e apenas se quer dar a conhecer a interface destes (a sua API)

Estrutura:



Design Patterns - Abstract Factory (cont.)

Participantes:

- AbstractFactory (WidgetFactory), declara um interface para as operações que criam objectos de produtos abstractos
- ConcreteFactory (MotifWidgetFactory, WindowsWidgetFactory), implementam as operações sobre implementações concretas
- AbstractProduct (Window, ScrollBar), declaram um interface para um dado tipo
- ConcreteProduct (MotifWindow, MotifScrollBar),
  - definem uma concretização de objecto da fábrica abstracta
  - implementam a interface de AbstractProduct
- Cliente, usa apenas os interfaces declarados em AbstractFactory e AbstractProduct

Design Patterns - Abstract Factory (cont.)

Colaborações:

- apenas uma única instância de ConcreteFactory é criada em tempo de execução
- AbstractFactory delega a criação de instâncias a cada ConcreteFactory

Consequências:

- Isola as concretizações (as classes dos objectos que realmente existem). O Cliente não vê as implementações, mas apenas interfaces. Logo não aparecem no código do cliente.
- Torna as mudanças de famílias de produtos relativamente fáceis.
- Promove a consistência entre os produtos. Cada produto é criado isoladamente e possui todo o comportamento necessário para a sua interacção.
- Como suportar novos tipos de produtos com interfaces diferentes??

Design Patterns - Abstract Factory (cont.)

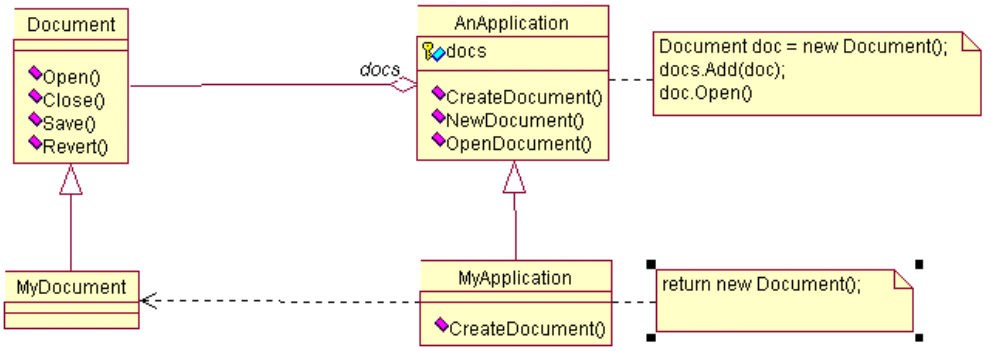
Análise do Impacto das alterações:

- adicionar um novo tipo de producto implica alterar a AbstractFactory e todas as subclasses que dela dependem.
- um modo mais flexível, mas menos seguro, de fazer isto é a adição de um parâmetro que especifica o tipo de producto a criar
- assim bastaria um único método, por exemplo `make(...)`. Claro que esta é uma solução mais adequada a linguagens com *dynamic binding*.

Design Patterns - Factory Method

Objectivo: Definir um interface para a criação de um objecto, deixando para as subclasses a decisão de quem instanciar.

Motivação: Considere-se uma framework para aplicações que gerem múltiplos documentos. A classe AnAplication apenas sabe que deve instanciar um documento, mas não sabe de que tipo é que ele vai ser.

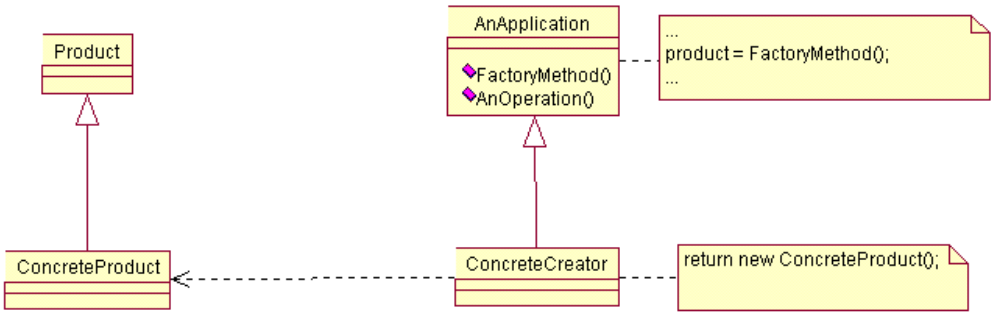


Design Patterns - Factory Method (cont.)

Aplicação: usar quando:

- uma classe não consegue antecipar os objectos que deve criar
- a classe deseja que seja a subclasse a especificar o que cria
- quem está a implementar quer que construir uma base de conhecimento em que sejas as subclasses a manter o conhecimento de como se cria uma instância.

Estrutura:



Design Patterns - Factory Method (cont.)

Participantes:

- Product (Document) - define o interface dos objectos que FactoryMethod cria
- ConcreteProduct (MyDocument) - implementa a interface dos produtos
- Creator (Application) - declara o método de *fabrico*, que devolve um objecto do tipo Product. Esta classe pode não ser abstracta, e ter um código *standard* já implementado.
- ConcreteCreator (MyApplication) - reescreve o método de *fabrico* de modo a que devolva uma verdadeira instância de ConcreteProduct.

Design Patterns - Factory Method (cont.)

Consequências:

- uma potencial desvantagem reside no facto de sempre que seja necessário criar um novo ConcreteProduct se tenha que especializar a classe Creator.
- torna o mecanismo de herança natural, *i.e.*, criar um objecto num esquema destes é sempre mais simples do que criar um objecto directamente.

Implementação:

- As duas maiores variações deste *pattern* consistem em:
  - Creator é uma classe abstracta e não providencia nenhuma implementação ⇒ é necessário criar uma subclasse
  - Creator já tem uma implementação
- Utilização de métodos de *fabrico* parametrizados

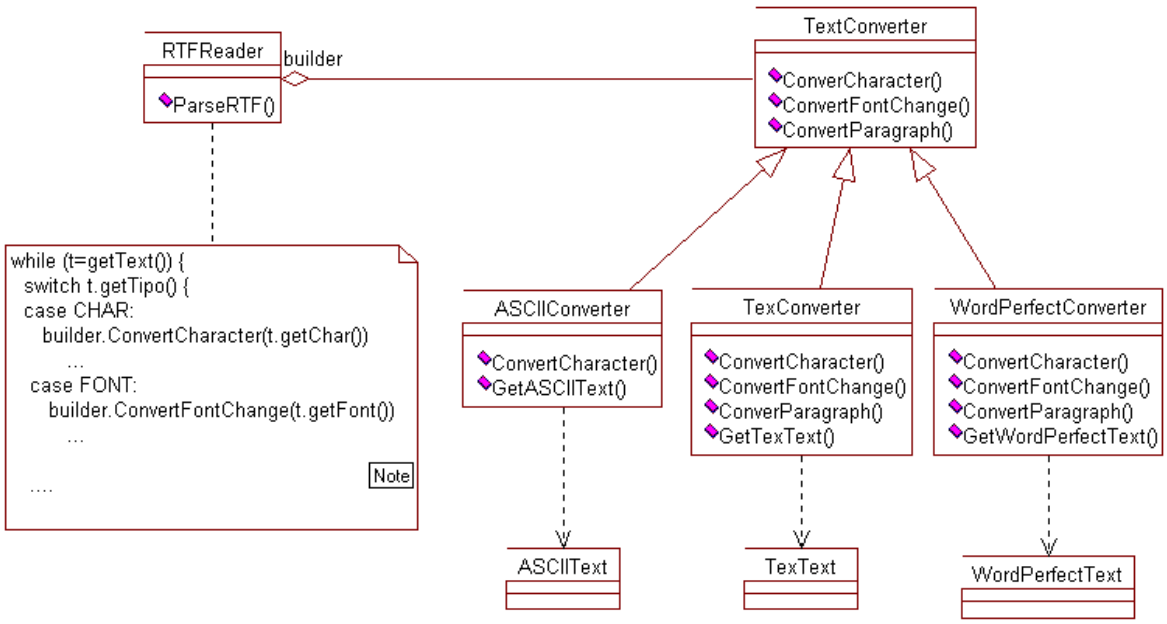
```
Product Create(ProdID id) {

  ...
  if (id == ProdutoTipoA) return new ProductA();
  if (id == ProdutoTipoB) return new ProductB();
  ...
}
```

Design Patterns - Builder

Objectivo: separa a construção de um objecto complexo da sua representação, de modo a que o mesmo processo origine representações diferentes.

Motivação: Considere-se o exemplo seguinte, em que existe um processo de transformação de texto RTF para diferentes formatos (Ascii, LaTeX, WordPerfect, etc). A problemática aqui reside no facto de o número de representações destino ser potencialmente grande, logo havendo a necessidade de prever diferentes tipos de conversão.

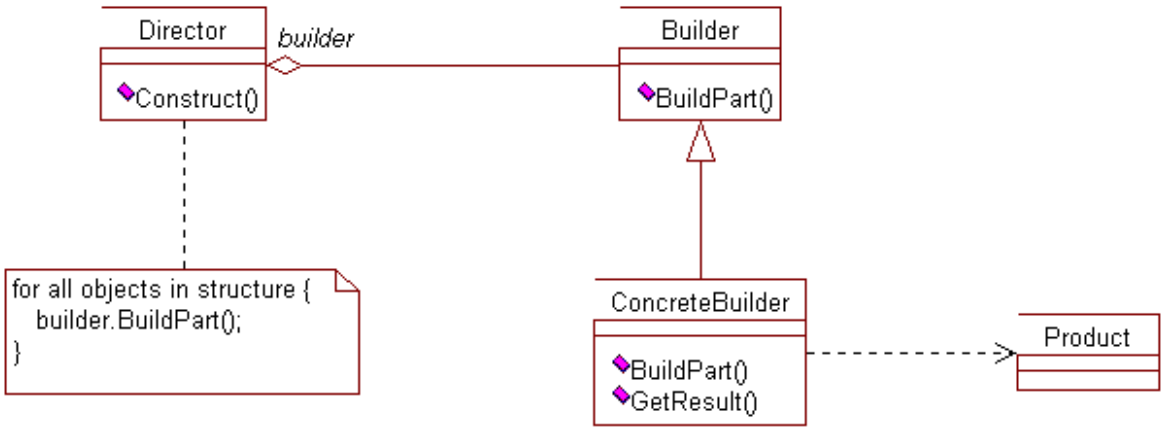


Design Patterns - Builder (cont.)

Aplicação:

- o algoritmo de criação de uma representação é independente da representação do objecto.
- o processo de construção deve admitir representações diferentes para o objecto a construir.

Estrutura:



Design Patterns - Builder (cont.)

Participantes:

- Builder (TextConverter) - especifica um interface abstracto para criar as partes de um objecto
- ConcreteBuilder (AsciiConverter, etc)
  - constroi e junta as partes implementando o interface
  - fornece um interface para ler um producto daquele tipo
- Director (RTFReader) - constroi um objecto utilizando o interface de Builder
- Product (AsciiText, etc)
  - representam o objecto a ser construído
  - incluem classes que definem as partes desses objectos

Design Patterns - Builder (cont.)

Colaborações:

1. O cliente cria o objecto Director e configura-o com o Builder desejado.
2. O Director notifica o builder sempre que uma parte do producto deva ser construído.
3. O Builder recebe os pedidos do Director e adiciona as partes ao Product a criar.
4. O cliente recebe o objecto Product do Builder.

Consequências:

- permite variações na representação interna.
- isola código distinto para a representação e para a criação.

Design Patterns

Comentário aos *patterns* de criação:

- Permitem maior flexibilidade nos seguintes aspectos:
  - o que é criado,
  - quando é criado,
  - quem cria e
  - como é criado.
- Existem duas maneiras de parametrizar um sistema pelas classes de objectos que ele cria:

  1ª - especializar a classe que cria os objectos. Corresponde à utilização do *pattern* FactoryMethod.

  2ª - fazer com que o sistema se baseie na composição, *i.e.*, definir um objecto que é responsável por conhecer as classes das implementações. Este é um aspecto chave dos *patterns* AbstractFactory e Builder. Todos eles criam uma fábrica de objectos cuja responsabilidade é criar objectos.

Design Patterns

Patterns estruturais

- preocupam-se em como as classes e os objectos são compostos por forma a criarem estruturas maiores.
- Existem dois tipos de patterns estruturais:
  1. de classe - estes padrões usam a noção de herança de modo a comporem interfaces e implemetações.
  2. de objecto - descrevem modos de composição de objectos de modo a que seja possível a descrição de novas funcionalidades. A vantagem destes padrões advém do facto de permitirem alterar a composição em tempo de execução.

Design Patterns - Bridge

Objectivo: Separar a interface, o nível de abstração, da implementação, com o propósito de que as duas possam evoluir independentemente.

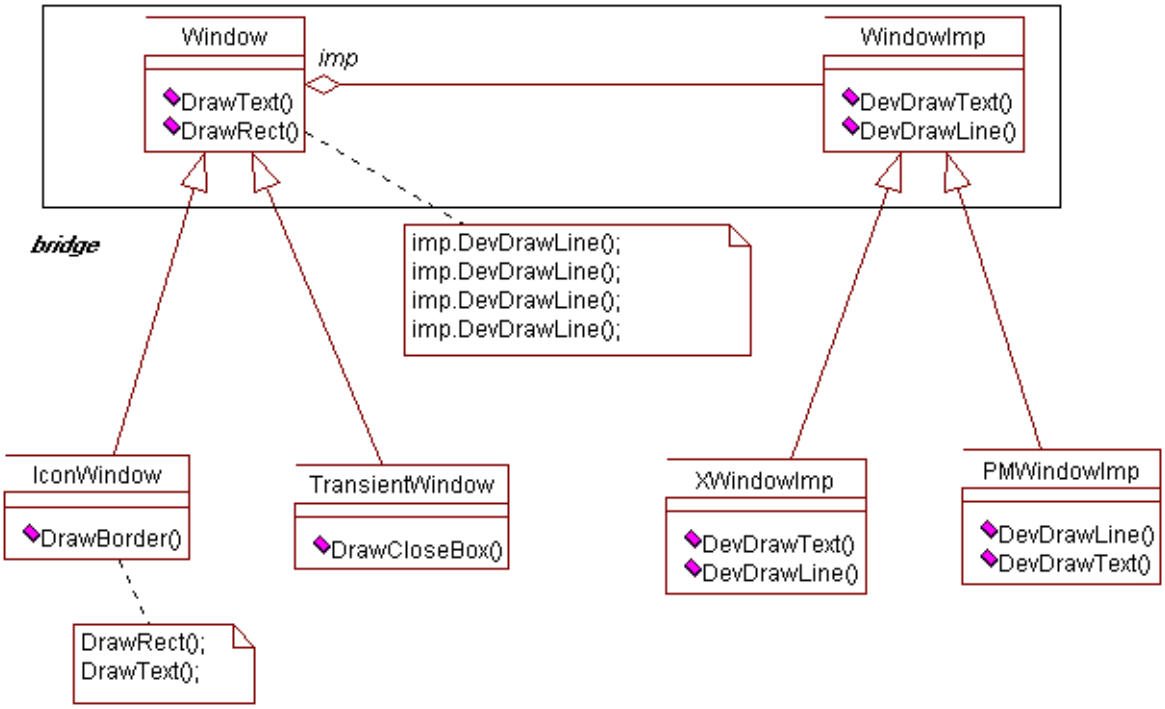
Motivação:

Quando uma interface pode ter mais do que uma implementação, a maneira mais simples de tratar esta situação é recorrendo a esquemas de herança. No entanto esta solução não é suficientemente flexível. Além de que obriga a criar subclasses para situações diferentes, pode obrigar a criar código dependente da solução tornando assim difícil portar o código para outras plataformas (aplicações).

Considere-se a situação de termos que criar código para a implementação de janelas (windows), e que as janelas são implementadas de modo diferente consoante a camada de apresentação.

O *pattern* Bridge foca estes problemas e a solução que propõe consiste na separação entre as hierarquias de abstração e de implementação.

Design Patterns - Bridge (cont.)



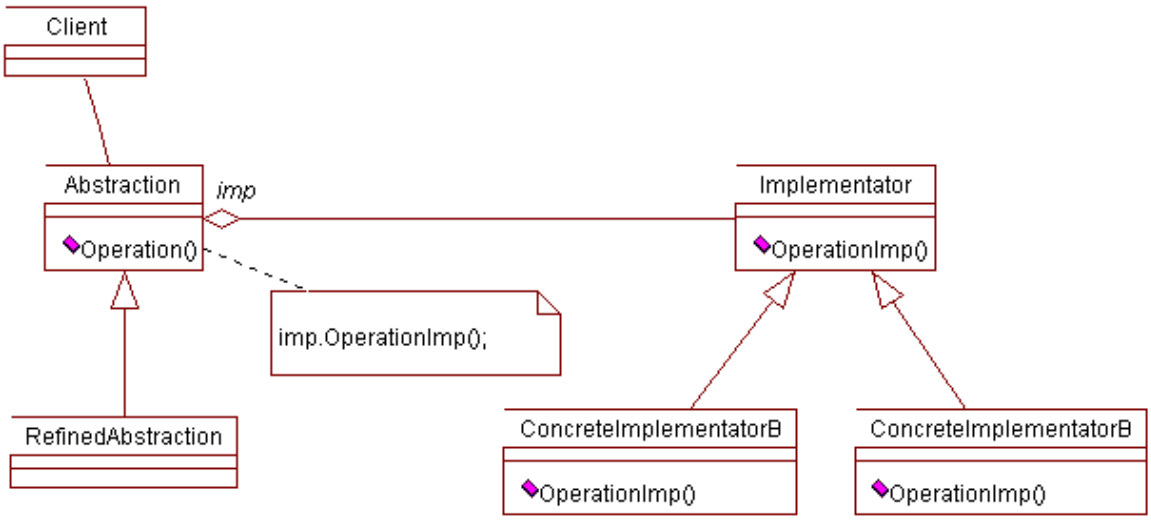
Aplicação:

- usar quando se quer evitar uma permanente associação entre a abstração e a implementação. Isto é particularmente verdade quando a implementação só é escolhida em tempo de execução.
- tanto as abstrações como as implementações podem ser extensiveis através do mecanismo de herança. Nessa situação o *pattern* deixa que as duas hierarquias evoluam de modo diferente.

Design Patterns - Bridge (cont.)

- modificações nas abstrações e nas implementações não se devem reflectir no cliente

Estrutura:



Participantes:

- Abstraction (Window)
  - define a interface da abstração
  - mantem uma referência para o objecto que constitui uma implementação
- RefinedAbstraction (IconWindow) - acrescenta funcionalidade ao interface

Design Patterns - Bridge (cont.)

- Implementator (WindowImp) -define o interface para as implementações. Este interface não precisa de corresponder exactamente ao interface da abstração. Normalmente a implemetação define primitivas e a abstração disponibiliza operações de alto nível baseadas nessas mesmas primitivas.
- ConcreteImplementator (XWindowImp, ...) - implementam os métodos específicos para uma situação concreta.

Consequências:

- Separação entre interface e implementação - permitindo que a implementação só seja conhecida em tempo de execução. Permite-se também que um objecto mude a sua implementação em tempo de execução.

  É também possível que se compile uma das hierarquias sem que a outra seja avisada.
- esconde a implementação dos clientes (recordar Abstract Factory, sendo que uma Abstract Factory pode criar e configurar uma concretização de Bridge).

Design Patterns - Composite

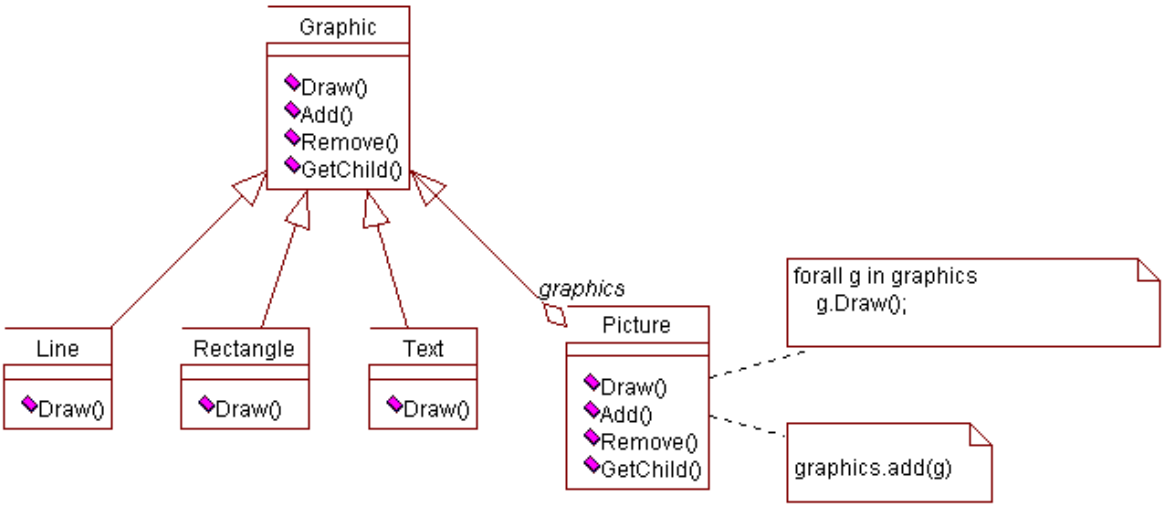
Objectivo: Compôr objectos em estruturas em árvore para se representar hierarquias do tipo *parte de*. Permite que se trate da mesma forma objectos singulares e também composições de objectos.

Motivação: Considere-se que temos, mais uma vez, uma aplicação gráfica, que permite a construção de diagramas complexos a partir de elementos mais simples. Seja por exemplo um diagrama composto por elementos simples como linhas, rectângulos, pedaços de texto e outros, e também por diagramas.

O problema reside no facto de que a interface dos elementos simples é diferente do interface dos elementos compostos, apesar de para o cliente tal distinção não ser necessária.

O *pattern* Composite tem como objectivo a representação uniforme dos elementos contidos e dos seus contentores.

Design Patterns - Composite (cont.)

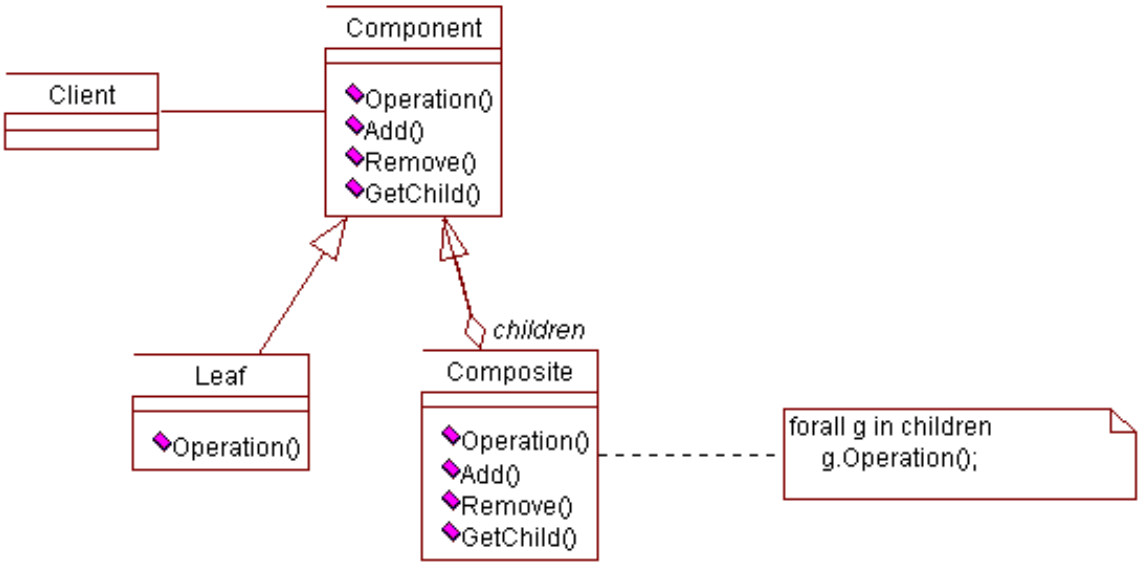


Aplicação:

- querem-se representar hierarquias em que esteja reflectida a composição de elementos
- quer-se que os clientes não se apercebam da diferença entre elementos contidos e os seus contentores. Os clientes tratarão todos os objectos de forma uniforme

Design Patterns - Composite (cont.)

Estrutura:



Participantes:

- Component (Graphic)
  - declara o interface dos objectos na composição
  - implementa o comportamento por omissão para todas as classes
  - declara um interface para acesso e gestão dos elementos contidos

Design Patterns - Composite (cont.)

- Leaf (Rectangle, Line, ...) - componentes não compostos
- Composite (Picture)
  - define o comportamento dos elementos compostos
  - guarda os elementos contidos
  - implementa operações sobre os elementos contidos

Colaborações:

- Os clientes usam a interface de Component para interagirem com os objectos. Se o recipiente da mensagem for um objecto do tipo Leaf então a mensagem é logo tratada. Caso contrário, se for um Composite, este encaminha a mensagem para um dos seus elementos filho.

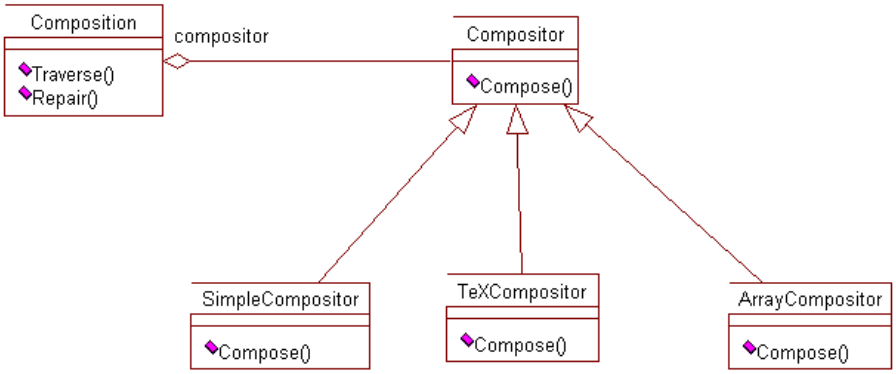
Consequências:

- o cliente não tem conhecimento do tipo de objecto com o qual interage
- o código dos clientes torna-se assim mais simples
- torna simples a extensão a novos tipos de componentes, mas também limita o facto de o cliente querer restringir alguns componentes

Design Patterns - Strategy

Objectivo: Definir uma família de algoritmos, encapsular cada um deles tornando-os assim reutilizáveis em mais do que uma situação. Permite assim que clientes diferentes possam usar algoritmos (estratégias) diferentes.

Motivação: Suponha-se o exemplo de construção de um editor de texto, em que existe a necessidade de separar uma stream de texto por linhas. Claro que a separação das linhas depende da aplicação cliente. Esta pode precisar de marcas, de separação física, etc. Torna-se também difícil alterar novos algoritmos de partição de linhas quando estes estão embebidos na aplicação cliente.

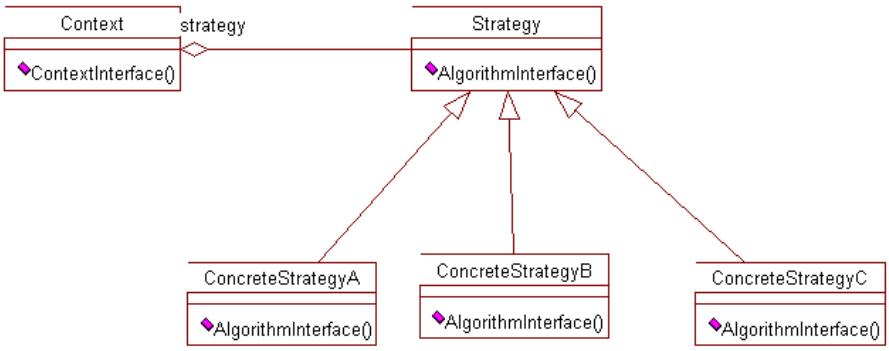


Design Patterns - Strategy (cont.)

Aplicação:

- muitas classes que estão relacionadas são diferentes no comportamento. Pode-se desta maneira fazer com que as classes sejam parametrizadas pelo seu comportamento.
- quando se necessita de variações de um algoritmo
- um algoritmo usa dados que não devem ser conhecidos do cliente
- uma classe pode exibir muitos comportamentos

Estrutura:



Design Patterns - Strategy (cont.)

Participantes:

- Strategy - declara um interface comum para todos os algoritmos suportados.
- ConcreteStrategy - implementa o algoritmo cuja interface respeita Strategy
- Context - é configurada para utilizar um algoritmo concreto e mantém uma referência para o algoritmo.

Consequências:

- permite detectar funcionalidades semelhantes. Favorece assim a criação de famílias de algoritmos
- apresenta uma alternativa ao esquema usual de herança. A classe é parametrizada pelo algoritmo
- permite eliminar expressões condicionais, na escolha do algoritmo
- aumenta o número de objectos e o peso (memória e comunicação) das aplicações.