Tópicos Avançados de Programação Orientada aos Objectos

Reutilização em Sistemas de Objectos

Design Patterns, Frameworks, Componentes

António Nestor Ribeiro

Universidade do Minho

Design Patterns

- Reutilização = *"não fazer tudo outra vez de novo!"*
  - quando se encontra uma **boa** solução reutilizá-la sempre que possível
  - permite basear os novos requisitos, e respectivas soluções, em soluções anteriormente testadas
- *Design Patterns* → mecanismo de reutilização de especificação e arquitectura
- Catálogo de Gamma, Johnson, *et all*: conjunto (não completo) dos *design patterns* que são possiveis de identificar.

  Limitações: não existem patterns para:
  - concorrência
  - paralelismo
  - real-time systems
  - object databases

Design Patterns

O que é um design pattern e *em que consiste*??

- **nome** - incremento no vocabulário (novos termos)
- **problema** - designa quando se aplica o *pattern*. Explica o problema e o contexto em que ele ocorre
- **solução** - elementos que constituem o *pattern*, as suas relações, responsabilidades e colaborações. (não é necessariamente código final)
- **consequências** - resultados e compromissos da aplicação do *pattern*.

A que nível nos situamos??

- Listas Ligadas, HashTables, Estruturas Dados, ...
- Nível a que nos situamos
- Sistemas Complexos

Objectivo: descrições de objectos e classes que são "alteradas" para resolver situações num determinado contexto.

Design Patterns

O MVC do SMALLTALK (Model, View, Controller) utiliza mais do que um *pattern*.

Patterns utilizados:

- Observer
- Composite
- Strategy

Como se descreve um *Design Pattern*??

⇒ notação gráfica → não captura as necessidades, compromissos, alternativas, *etc.*

Organização do catálogo:

- Nome e Classificação
- Objectivo
- *Alias* (outras designações)
- Motivação
- Aplicação
- Estrutura
- Participantes
- Colaboração (relacionamentos)
- Consequências
- Implementação
- Exemplos Código
- Utilizações Conhecidas
- *Patterns* relacionados

Design Patterns

Tipos de *Patterns*:

1. **Criação** (Creational)
2. **Estruturais** (Structural)
3. **Comportamento** (Behavioural)

Que *patterns* é que temos?

Creational	**Structural**	**Behavioural**
Factory Method	Adapter	Interpreter
		Template Method
Abstract Factory	Adapter	Chain of Responsability
Builder	Bridge	Command
Prototype	Composite	Iterator
Singleton	Decorator	Mediator
	Facade	Memento
	Flyweight	Observer
	Proxy	State
		Strategy
		Visitor

Design Patterns

Exemplos de *patterns*:

- Creational
  - Abstract Factory - interface para criar famílias de objectos sem conhecer as suas concretizações
  - Builder - Separa a construção de um objecto complexo da sua representação de modo a que o processo de construção possa criar representações diferentes.
  - Factory Method - define um interface para criar um objecto, mas deixa que sejam as subclasses a definir quem se instancia.
  - ...
- Structural
  - Adapter - Converte o interface de uma classe noutro interface que é o esperado. Compatibiliza interfaces.
  - Composite - compõe objectos em estruturas em árvore de modo a representar a relação *part-of*.
  - ...
- Behavioural
  - Observer - Define uma relação *um para muitos* entre objectos, para que quando um objecto mude essa mudança seja notificada aos outros.
  - Template Method - define um *template* de algoritmo numa operação, deixando às subclasses o preenchimentos de alguns passos.
  - ...

Design Patterns

Além da reutilização por *patterns*, a reutilização podia ser também conseguida através:

- toolkits - quando se incorporam classes de uma, ou mais, bibliotecas. Os toolkits são um conjunto de classes de uso geral que podem ser reutilizadas.

  ⇓

  quem escreve o toolkit nao esta em condições de saber o que se passa na aplicação em que são invocados

  ⇓

  importante: *não assumir compromissos a montante*

  ⇓

  exemplo: Java Beans

- frameworks - conjunto de classes cooperantes que formam um suporte para reutilização num campo específico. Exemplo: *patterns* para negócio ou para gestão.

Design Patterns

A framework utilizada dita a arquitectura da aplicação, visto que define toda a estrutura (classes e objectos), relações e até o controlo da aplicação.

Vantagem: por alguma coisa a correr *imediatamente*.

Desvantagem: mudanças na framework são dificilmente digeridas

Quais são as principais diferenças entre as frameworks e os *design patterns*?

1. O nível de abstracção de um *pattern* é superior ao de uma framework
2. Os *patterns* são elementos arquitecturais mais pequenos
3. Os *patterns* são **menos especializados** que as frameworks

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

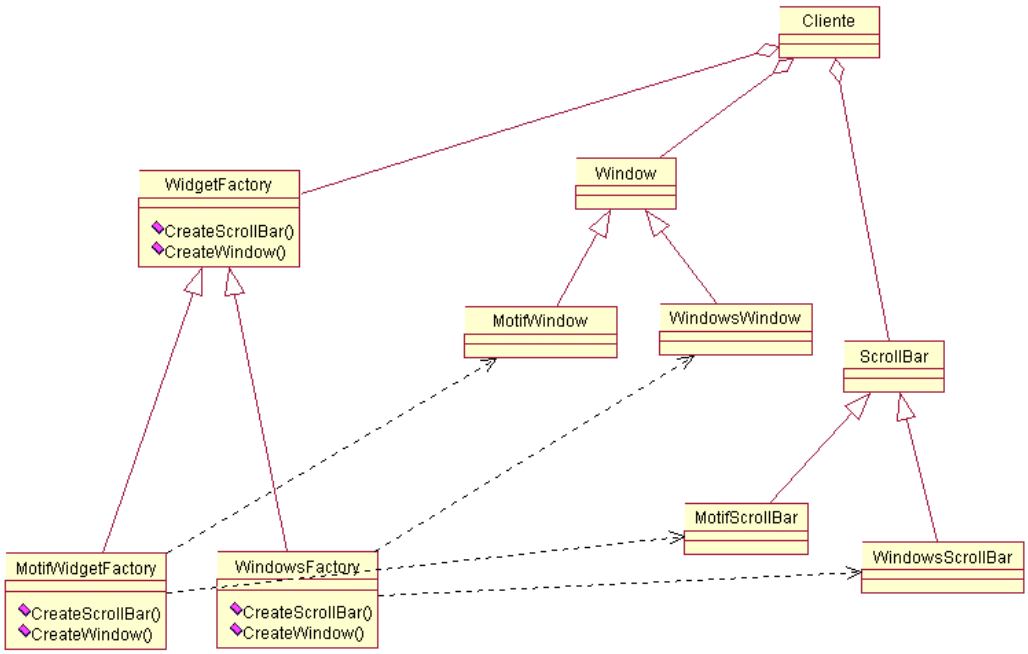
O que fornecem:

- 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).

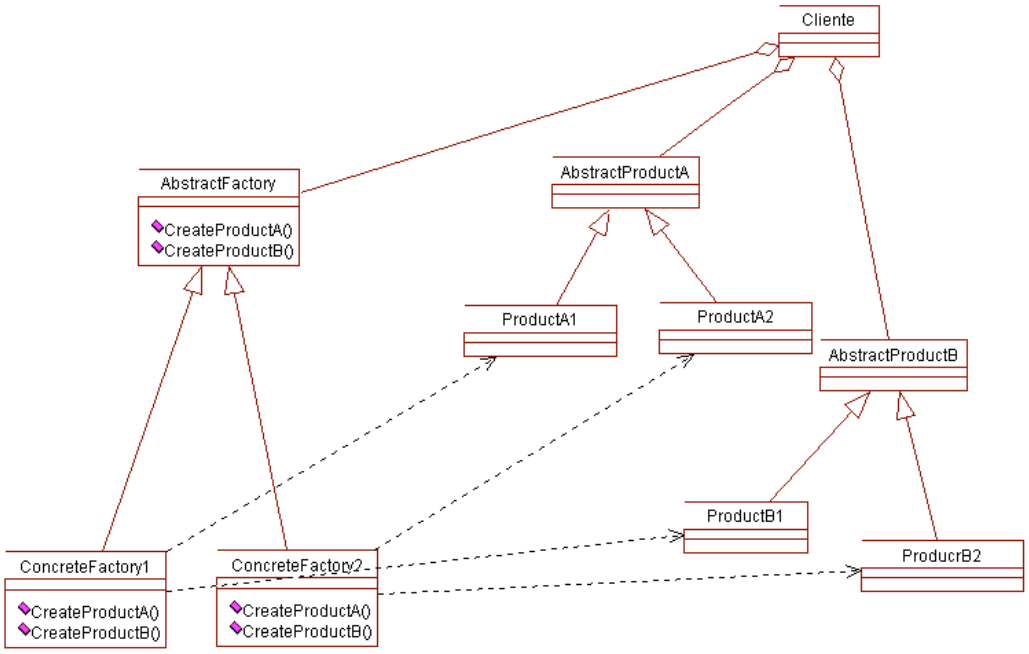


Design Patterns - Abstract Factory (cont.)

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.)

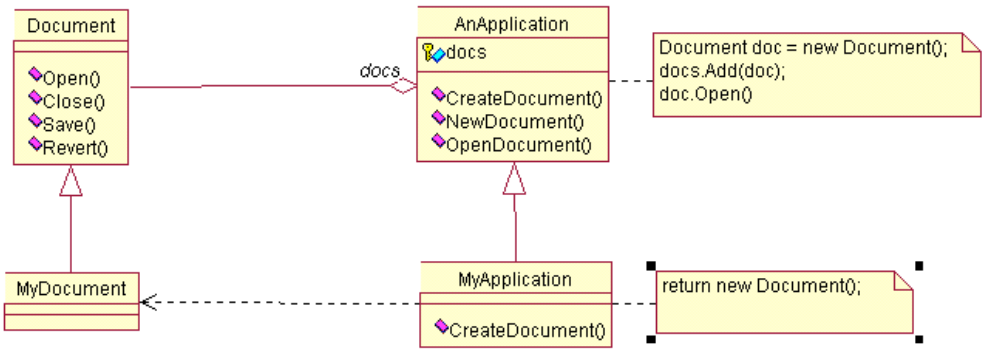
Análise do Impacto das alterações:

- adicionar um novo tipo de producto implica alterar a AbstractFactory e todas as subclasses que dela dependem.
- um modo mais flexível, mas menos segura, 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.

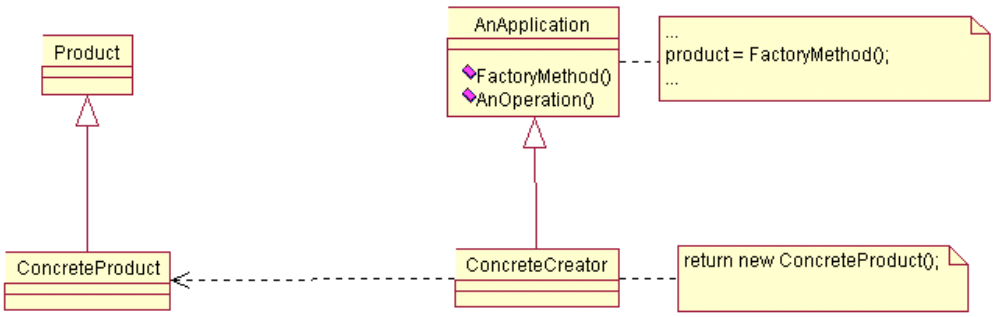


Design Patterns - Factory Method (cont.)

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 construir uma base de conhecimento em que sejam 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 sublcasse
  - Creator já tem uma implementação
- Utilização de métodos de *"fabrico"* parametrizados

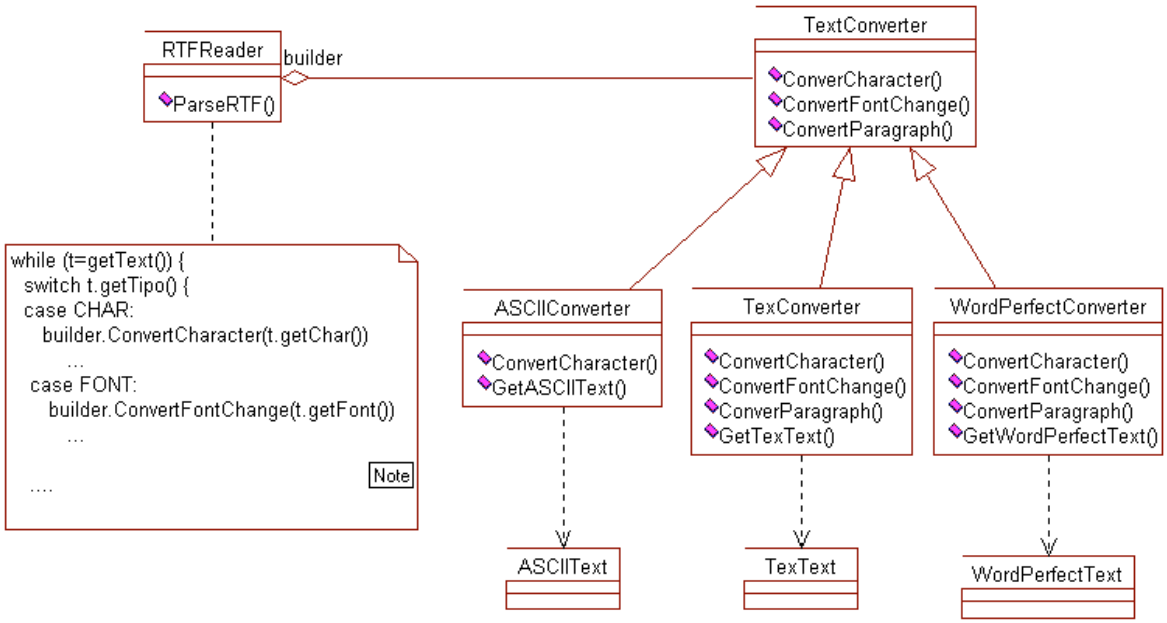
```
Product Create(ProdID id) {

  ...
  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.

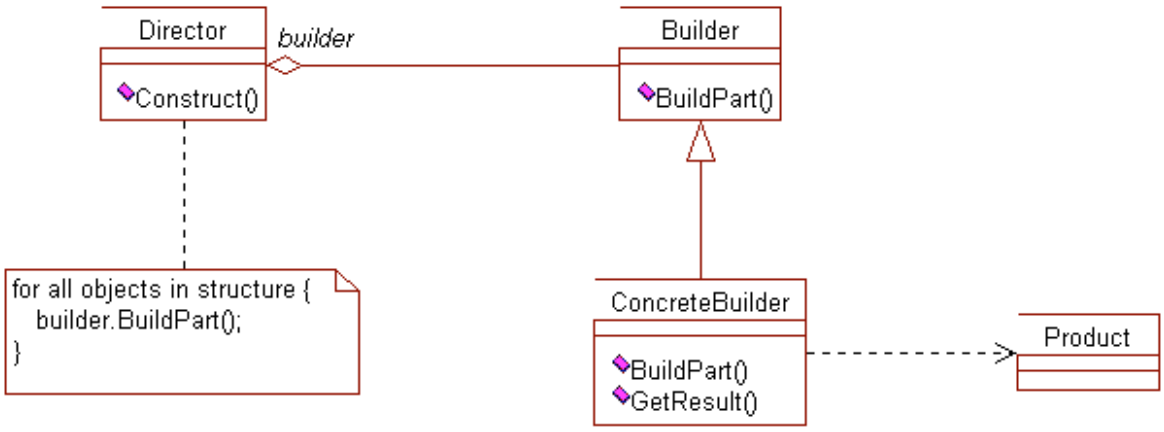


Design Patterns - Builder (cont.)

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 - Adapter

Objectivo: Converter a interface de uma classe noutra interface que a aplicação cliente espera.

Motivação:

Considere-se um editor gráfico que permite desenhar e arrangar elementos gráficos em imagens e diagramas. Cada objecto gráfico nesta aplicação é subclasse (é do tipo de) de `Shape`.

Pode porém acontecer que existam já classes feitas para representar alguns tipos de elementos do editor gráfico, mas que essas classes tenham sido feitas sem saberem da existência de `Shape`.

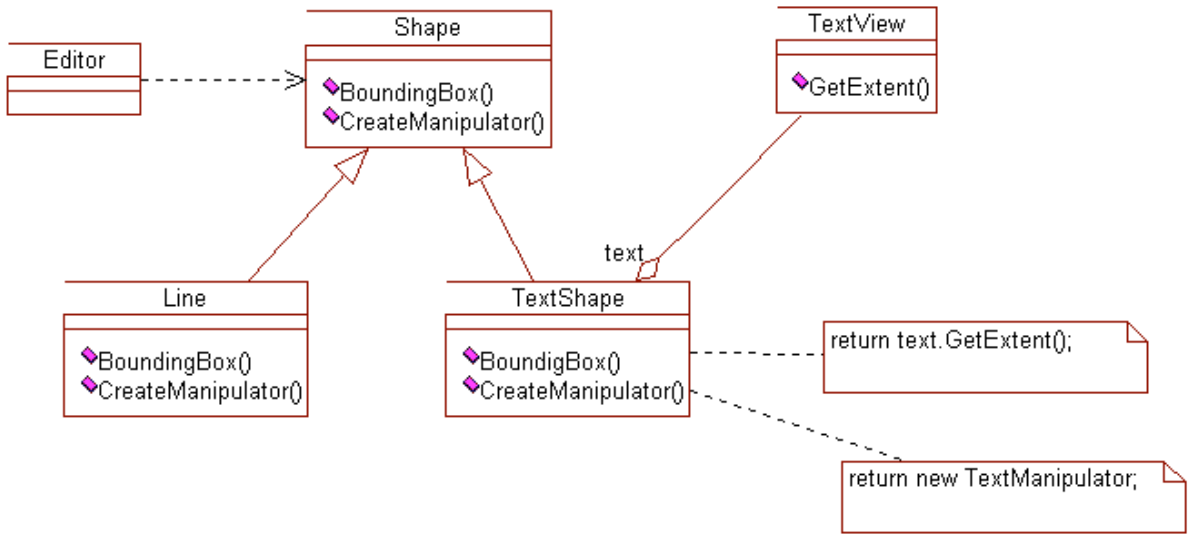
Considere-se que existe uma classe, `TextView`, já implementada e que não é subclasse de `Shape`, logo não tem uma interface compatível.

A solução passa pela criação de uma classe **adaptadora**, `TextShape`, que adapte o interface de `TextView`.

Design Patterns - Adapter (cont.)

Isto pode ser feito de duas maneiras:

1. herdando a interface de `Shape` e a implementação de `TextView`
2. compondo uma instância de `TextView` com `TextShape` e implementando `TextShape` em função da interface de `TextView`



Design Patterns - Adapter (cont.)

Aplicação:

- quando for necessário usar uma classe existente, mas o seu interface não é o que se pretende
- quando se quer criar uma classe reutilizável que possa interactuar com classes que não tenham interface compatível
- **(object adapter)** quando é necessário usar várias subclasses (existentes), mas não é possível adaptar a interface de cada uma.

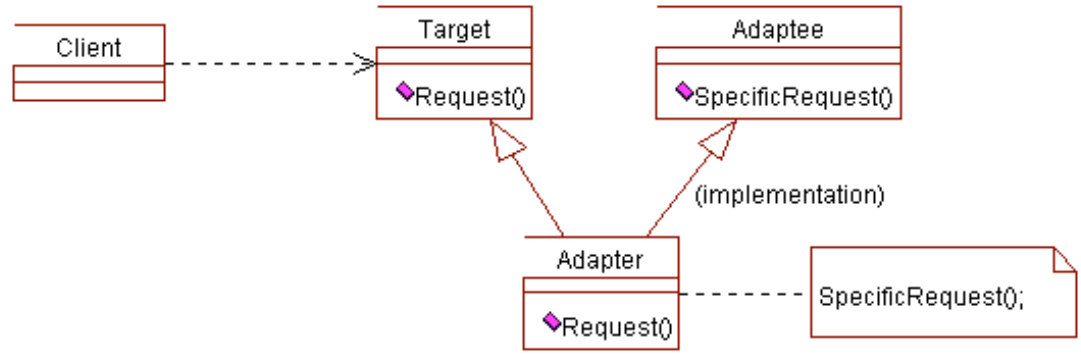
Estrutura:

Este *pattern* apresenta duas soluções possíveis, uma para composição a nível de classes e outra a nível de objectos.

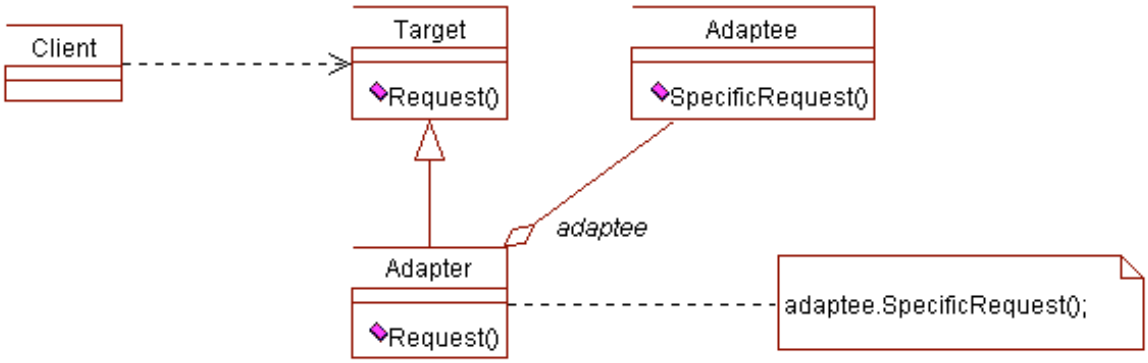
A sua estrutura corresponde assim aos seguintes casos:

Design Patterns - Adapter (cont.)

Class Adapter



Object Adapter



Design Patterns - Adapter (cont.)

Participantes:

- Target (Shape) - define a interface que o cliente usa
- Adaptee (TextView) - define a interface existente e que necessita de adaptação
- Adapter (TextShape) - adapta a interface de Adaptee à de Target

Colaborações:

- os cliente apenas chamam operações na instância de `Adapter`. A instância adaptadora é que invoca os métodos necessários na instância adaptada.

Consequências:

- Adaptador de Classes
  - a adaptação é feita a uma classe concreta. Logo o processo não é transitivo e aplicável às suas subclasses.
  - deixa que o adaptador reescreva algum do comportamento da adaptada (uma vez que é subclasse desta).

Design Patterns - Adapter (cont.)

- Adaptador de Objectos
  - um único adaptador pode trabalhar com muitos adaptados. O adaptador pode acrescentar funcionalidade ao adaptado.
  - torna mais complicada a tarefa de redefinição de funcionalidade do adaptado. Requer a criação de subclasses do adaptado e a referência passar a ser feita para os objectos das subclasses.

Design Patterns - Bridge

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

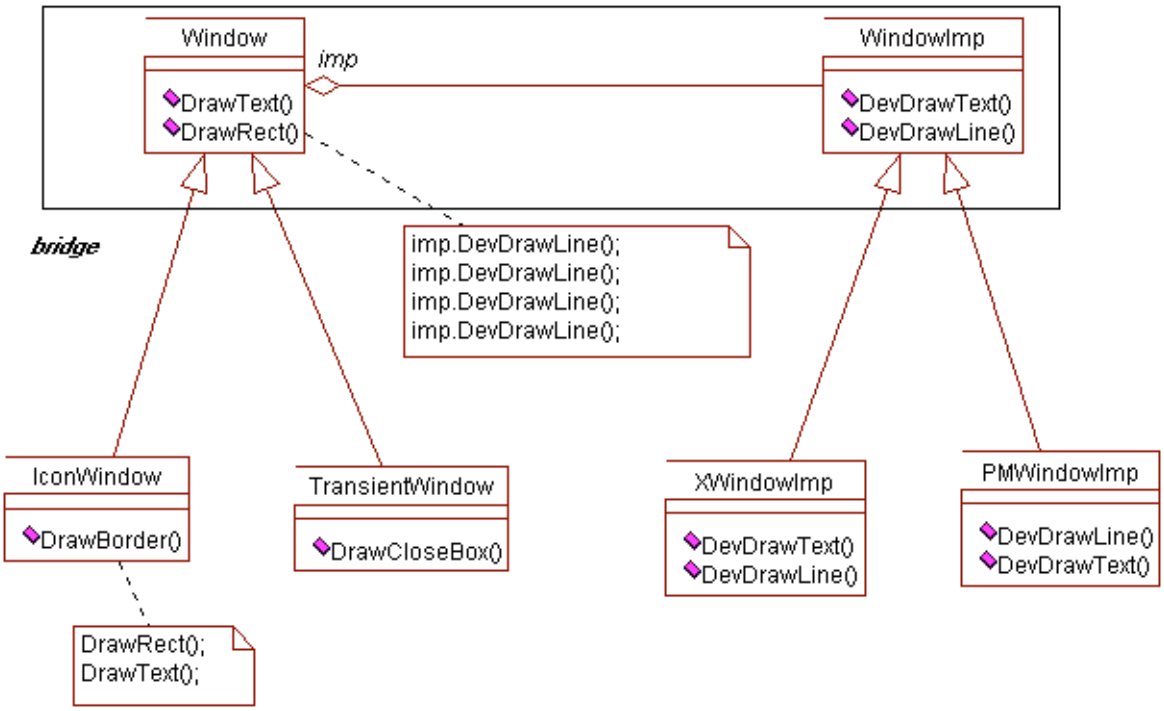
Motivação:

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.)



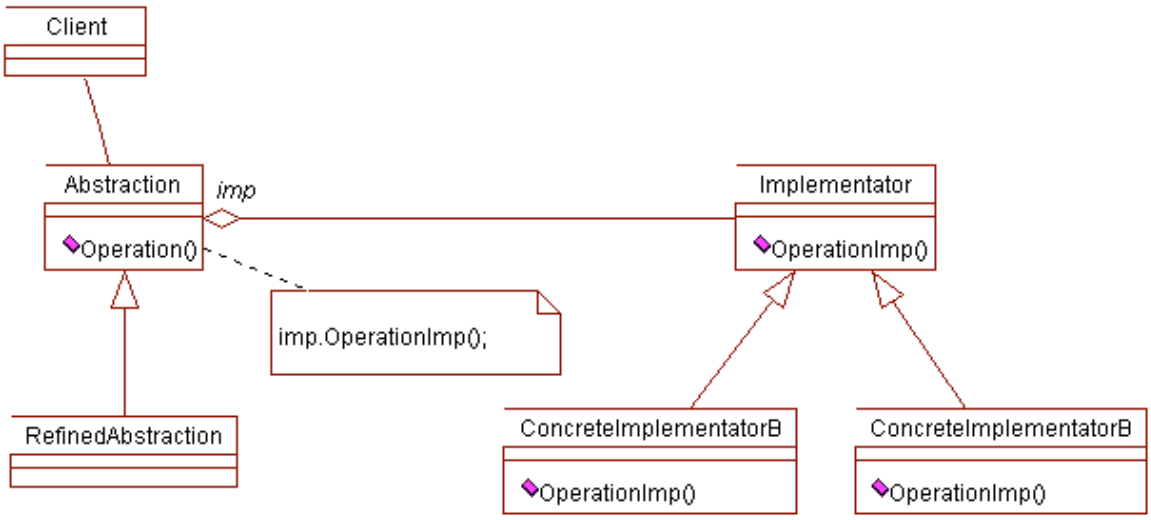
Aplicação:

- 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

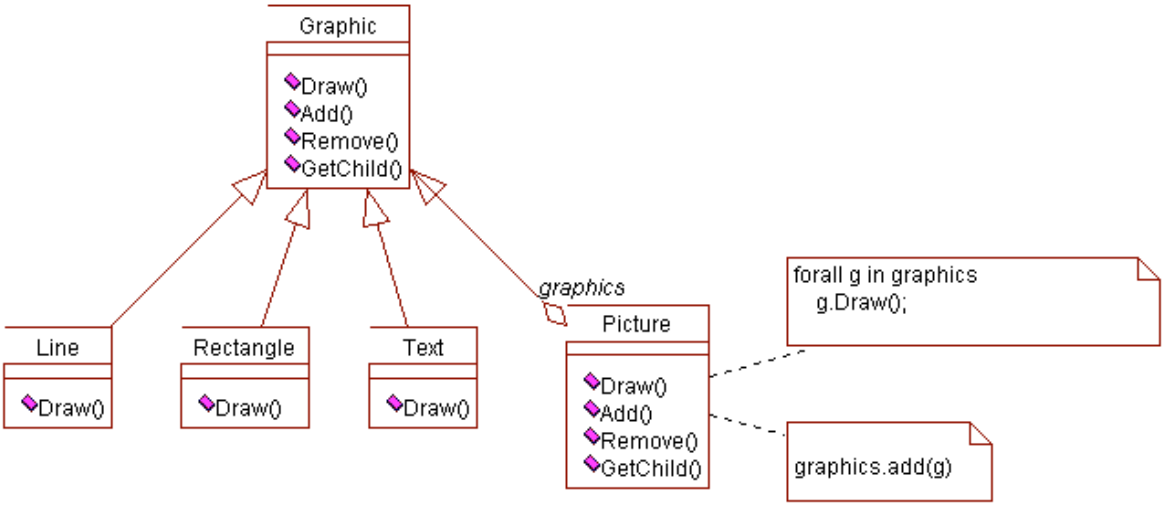
Objectivo: Compôr objectos em estruturas em árvore para se representar hierarquias do tipo *parte de*. Permite que se trate da mesma forma objectos singulares e também composições de objectos.

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.)

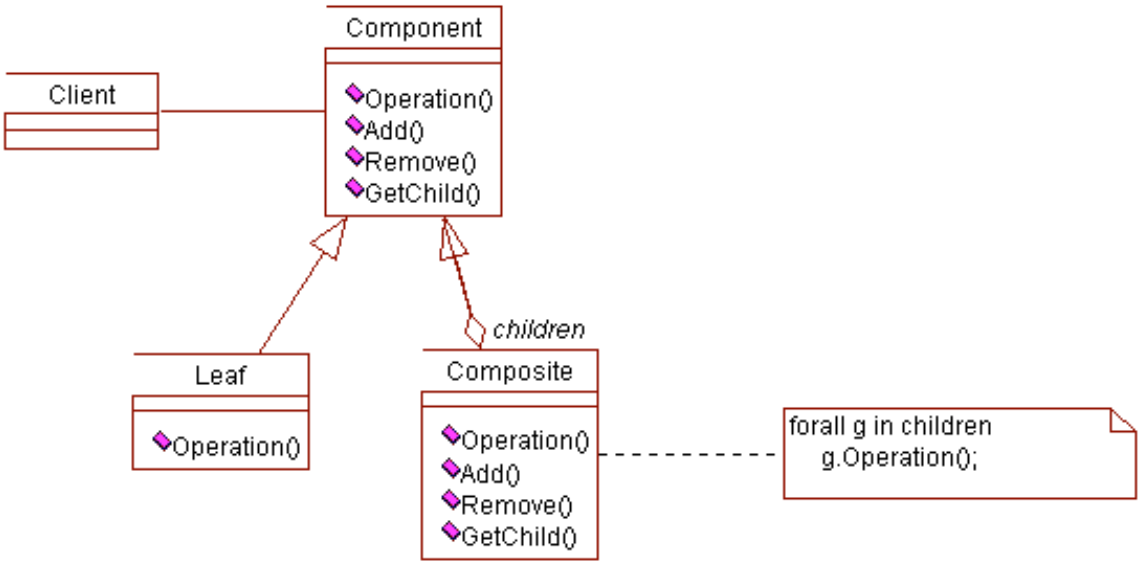


Aplicação:

- 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:

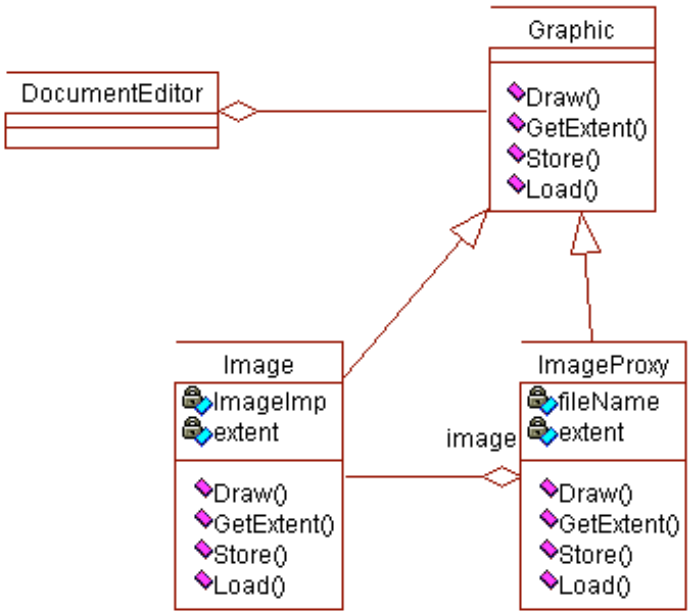
- o cliente não tem conhecimento do tipo de objecto com o qual interage
- o código dos clientes torna-se assim mais simples
- torna simples a extensão a novos tipos de componentes, mas também limita o facto de o cliente querer restringir alguns componentes

Design Patterns - Proxy

Objectivo: Providenciar um objecto que seja um contentor para um outro.

Motivação: Imagine-se um editor de texto que não apresenta as imagens que nele estão contidas, por isso constituir uma tarefa "pesada"aquando da edição. É assim necessário ter um contentor para a imagem, que só se associará à imagem quando tal for necessário.

A solução reside assim no uso de um outro objecto, um objecto **proxy**, que durante um certo tempo assume a identidade do objecto que aí deveria estar.

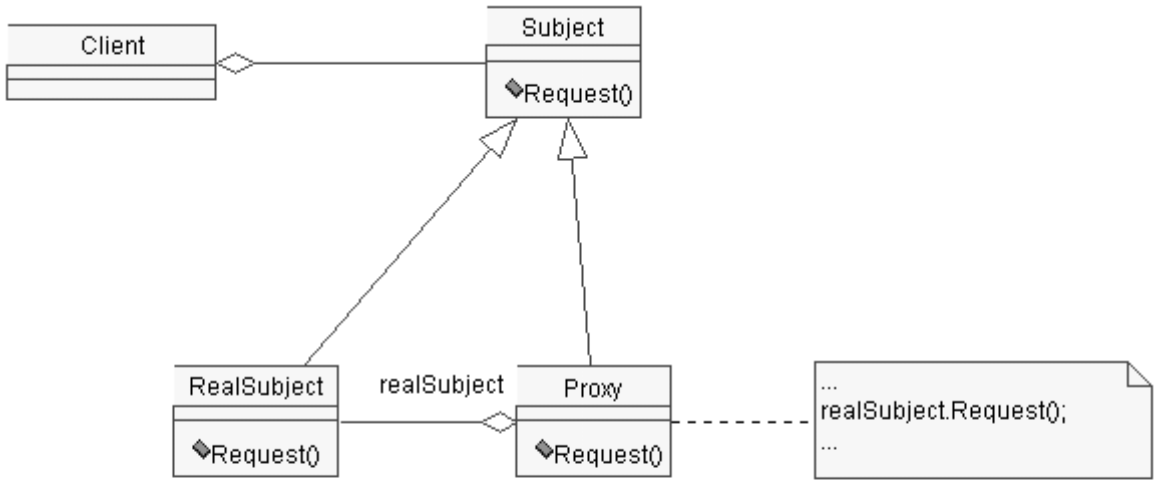


Design Patterns - Proxy (cont.)

Aplicação:

- quando se precisa de um mecanismo mais versátil e sofisticado para referenciar um objecto do que um simples apontador
- quando se necessita de "alguma"proteção para o objecto que se quer referenciar. O objecto **proxy** funciona aqui como uma *"firewall"*
- permite criar objectos *"caros"*
  apenas quando necessário

Estrutura:



Design Patterns - Proxy (cont.)

Consequências:

- o *pattern* representa um nível de indirecção no acesso ao objecto. Isto tem as seguintes vantagens:
  - um proxy remoto pode esconder que o objecto reside num outro espaço
  - um proxy virtual pode realizar algum tipo de optimização
  - permite efectuar algum tipo de operação interna aquando do acesso ⇒ Encapsulamento

Patterns Semelhantes:

O Proxy funciona com o mesmo espirito que o Adapter. Contudo o Proxy pode não adaptar toda a interface, mas apenas algumas partes.

Design Patterns

Discussão dos Patterns estruturais

- várias semelhanças entre os vários *patterns*
- existem poucas alternativas para a estruturação:
  - herança simples e múltipla, no caso dos patterns de classe
  - composição e agregação, no caso dos patterns de objectos

Comparação entre o Adapter e o Bridge

- Existem atributos comuns:
  - nível de indirecção
  - passagem de mensagens
- Diferenças ao nível dos requisitos:
  - o Adapter está apostado na resolução da adaptação de interfaces (compatibilidades). Não se preocupa em como esses interfaces são implementados, ou como podem evoluir.
  - o Bridge concentra-se na dualidade entre uma abstração e as suas possíveis implementações.

Design Patterns

Estas diferenças levam a que o Adapter e o Bridge sejam utilizados em diversas fases do ciclo de desenvolvimento do software.

O Adapter *torna-se necessário* quando se percebe que existem implementações incompatíveis.

O utilizador de um Bridge sabe *à priori* que uma abstração pode ter várias implementações e que elas podem evoluir independentemente.

- O Adapter é utilizado para fazer as coisas funcionarem **depois** de feitas
- O Bridge faz com que as coisas funcionem mesmo **antes** de elas existirem (em definitivo)