Aulas Práticas da Disciplina de

Programação com Objectos

Ana Cardoso Cachopo

Ano Lectivo 2013/2014

Conteúdo

1 Aula P01 — Apresentação; Tipos de dados em C

Sumário:

- Apresentação, há exercícios de avaliação em cada aula prática
- Notação UML mínima para classes
- Programas em C: ficheiros .h e .c, compilação, linkagem
- Exemplos de definição de tipos de dados abstractos
- Vantagens da reutilização de conceitos

Resumo:
Apresentação

- Que linguagens de programação conhecem

Ano	Disciplina	Linguagem
1.1	FP	Python
1.2	IAED	C
1.2	AC	Assembly
2.1	PO	Java
2.1	SO	C

UML

- As classes são representadas por rectângulos com três "fatias".
- Na primeira fatia têm o nome da classe.
- Na segunda fatia têm os atributos ou variáveis. Nesta aula consideramos só os nomes dos atributos.
- Na terceira fatia têm os métodos ou funções. Nesta aula consideramos todos os métodos, incluindo os construtores, destruidores, getters e setters, mas apenas os seus nomes.

Programas em C

- Têm ficheiros .h e .c para organizar melhor o código de programas grandes. Tendo a interface no .h, podem usar as funções definidas no .c respectivo (mais comprido), mas sem ter que incluir todo o .c. Também serve para esconder funções implementadas no .c, mas que não são tornadas públicas no .h.
- O programa principal fica no `main.c`, que deve devolver um inteiro. Normalmente é zero, a não ser que haja algum erro.
- No Mac, o C não vinha de raíz. Tive que ir à AppStore fazer download do Xcode, que é free. Depois, foi preciso abrir a aplicação e ir às Preferências no menu Xcode, tab Downloads, instalar as Command Line Tools. Depois disto, passei a poder usar o compilador de C no terminal, com o comando `gcc`.
- Também posso usar o compilador de C++ com o comando `g++`. Não sei se ficou instalado por causa do Xcode ou se já estava disponível.

Vantagens da reutilização de código

- Em vez de usar copy-paste, devemos pensar como é que podemos reutilizar o nosso código de uma forma inteligente.
- Lembrar que "um **bom** programador é um **bom** preguiçoso, não é um preguiçoso qualquer".
- Se houver um bug no código copiado, temos que ir à procura e corrigir o erro várias vezes no resto do programa. Se tivermos reutilizado o código, mas sem o copiar, só precisamos de corrigir uma vez. Para além disso, a abstração das características comuns normalmente ajuda-nos a encontrar melhores formas de resolver os problemas.

Nestes exercícios

- Os exercícios mostram a definição de tipos de dados abstractos e de algumas instâncias. Estas instâncias são semelhantes aos objectos suportados por linguagens como o C++ ou o Java mas, como estão implementados em C, algumas das operações têm de ser definidas explicitamente pelo programador.
- `#ifndef __ANIMAL_H__`
  `#define __ANIMAL_H__`
  As directivas para o compilador começam com `#`. Neste caso, como o tipo Animal pode ser usado por mais do que um tipo e por isso incluído em mais do que um ficheiro, é bom estilo de programação dizer que se a constante `__ANIMAL_H__` não está definida, define-a e continua até ao `#endif`. Da próxima vez que este ficheiro for incluído por outro, como a constante já está definida, passa para o `#endif`, que neste caso está imediatamente antes do fim do ficheiro e por isso não precisa de repetir trabalho.
- A gestão de memória é feita pelo programador: tem que alocar a memória para os elementos dos tipos e libertá-la quando já não é necessária. É preciso cuidado para não ter fugas de memória.
- `int equalsAnimal(Animal animal1, Animal animal2);`
  O C não tem booleanos, por isso usamos um inteiro em que zero representa *falso* e os outros inteiros representam *verdadeiro*.
- `const char *getAnimalName(Animal animal);`
  Não existem strings, estas são representadas por ponteiros para `char`. Os nomes nos exercícios têm `const` porque não mudam de comprimento ao longo da execução do programa.
- `#include <string.h>`
  `#include <stdlib.h>`
  `#include <stdio.h>`
  `#include "Animal.h"`
  Inclusão das bibliotecas standard para lidar com strings, alocação de memória e escrita para o ecran, para além dos headers deste tipo.
- `Animal animal = (Animal)malloc(sizeof(struct animal));`
  Determina o tamanho da variável `animal` que é uma estrutura, aloca memória para um elemento deste tipo, faz um typecast para animal `(Animal)` e atribui o resultado à variável `animal`, do tipo `Animal`.
- `strcpy(animal->_name, name);`
  Como o nome é uma string, ou seja, um ponteiro para `char`, se fizesse a atribuição `animal->_name = name;` estava a copiar o ponteiro e não o conteúdo do nome.
- `!strcmp(getAnimalName(animal1), getAnimalName(animal2))`
  Para comparar strings usa-se o `strcmp`, que devolve zero quando elas são iguais, por isso é necessário negar usando o `!`, pois queremos que os animais possam ser iguais quando têm o mesmo nome.
- `printf("a1==a1? %s\n", equalsAnimal(a1, a1) ? "yes": "no");`
  A string do `printf` é o resultado de uma versão abreviada do if.

2 Aula P02 — Introdução aos objectos

Sumário:

- Análise e modelação em UML
- Implementação de classes simples em Java

Resumo:

Objectos

- Um objecto tem um estado, comportamento e identidade.
- Um objecto tem uma interface.
- Classe $\neq$ Instância (Pessoa $\neq$ Maria).

UML

- `+` — public
- `#` — protected
- `-` — private
- Seta a tracejado: relação de dependência

Java

- Instalar JDK (Java Development Kit) e não apenas o JRE (Java Runtime Environment).
- Uma classe é uma implementação de um tipo, que fica num ficheiro e pode ter vários métodos.
- A classe `Object` é a superclasse de todas as classes em Java.
- Qualquer programa é uma coleção de classes, normalmente cada classe pública está no seu ficheiro (mas dentro de uma classe podem ser definidas classes internas, que ficam no mesmo ficheiro).
- Definição de classes: `class` nome `extends` superclase.
- Se não estender nenhuma classe, estende implicitamente `Object`.
- Em Java, se definir uma classe mas não definir nenhum construtor, é criado automaticamente um construtor sem argumentos, chamado construtor default. Se for definido pelo menos um construtor, o construtor default já não é definido automaticamente.
- Os tipos primitivos não são `Objects`.
- Auto-boxing — conversão automática dos tipos primitivos para os respectivos wrappers. Também existe o auto-unboxing.
- Fazer *downcast* corresponde a passar de um tipo para um seu sub-tipo. Fazer *upcast* corresponde a usar um elemento de um sub-tipo num local onde se espera um elemento de um tipo mais acima na hierarquia, por exemplo, um cão no lugar de um animal.
- Todas as instruções têm que terminar com `;` (ponto e vírgula).
- Comentários "normais" estão entre `/* text */` e são ignorados pelo compilador.
- Comentários de apenas uma linha começam com `//` e o compilador ignora tudo até ao fim da linha.

- Comentários com documentação para ser usados pelo javadoc `/** documentation */` são ignorados pelo compilador mas o JDK usa-os para preparar a documentação que é gerada automaticamente.
- A maioria dos espaços em branco são ignorados.
- As variáveis são declaradas com um nome e um tipo. Existem *variáveis de instância* que são declaradas nas classes, e *variáveis locais*, que são declaradas nos métodos. Se não forem inicializadas, as `variáveis de instância` ficam com o "zero" do tipo correspondente por omissão. Se não forem inicializadas, as *variáveis locais* não têm valor, por isso o código não vai compilar se não existir um valor inicial. É boa prática inicializar todas as variáveis.
- As classes e os métodos são definidos dentro de chavetas.
- Em Java os inteiros e os boleanos não são compatíveis, por isso os inteiros não podem ser usados directamente nos testes, só com operadores que devolvam boleanos.
- == identidade em Java, usado para comparar tipos primitivos ou referências. Diferente de = atribuição. Para comparar dois objectos diferentes que podem ter o mesmo conteúdo, por exemplo duas strings com os mesmos caracteres, usar o método `equals()`.
- Os métodos sem `return` são `void`.
- Todas as aplicações Java têm que ter pelo menos uma classe e um método `main`. Cada classe pode ter o seu próprio `main`, por exemplo para testar a classe, mas a aplicação principal só precisa de um.
- O método `main` é sempre `public static`, pode ser `int` ou `void` e recebe sempre um array de strings `(string[] args)`.
- Anotações fora dos comentários são para o compilador: `@param` é um marcador para a documentação em Java; `@override` diz que esté a sobrepor-se a um método herdado (para no caso de me enganar a escrever o nome, por exemplo, o compilador poder dar erro).
- Convenções: tudo maiúscula para atributos de classe (`static`).
- **Keywords de acesso** de Java que podem estar aplicadas aos atributos ou aos métodos das classes:
  - `public` — todos têm acesso. Os métodos que são para ser usados fora da classe são `public`.
  - `private` — só o código da classe pode aceder, são detalhes de implementação da classe. Normalmente os atributos ou variáveis de instância são `private`, porque são manipulados pelos `getters` e `setters`.
  - `protected` — só podem ser acedidos dentro da classe e suas subclasses.
  - *nada* — todas as classes da mesma package podem aceder.
  - `static` — nas variáveis e métodos significa que são variáveis ou métodos de classe. As variáveis e métodos de classe existem, mesmo que não exista nenhuma instância da classe. São usados com o nome da classe e não de uma instância. As variáveis e métodos `static` não podem usar variáveis nem métodos de instância (não static).
  - `final` — numa variável significa que não pode ser modificada. Num método significa que não pode ser redefinido nas subclasses. Numa classe significa que não pode ter subclasses. As interfaces não podem ser `final`.

Tipos primitivos e wrappers

Tipo	Dimensão (bits)	Mínimo	Máximo	Wrapper
boolean	-	-	-	Boolean
char	16	Unicode 0	Unicode 216-1	Character
byte	8	-128	+127	Byte
short	16	-215	+215-1	Short
int	32	-231	+231-1	Integer
long	64	-263	+263-1	Long
float	32	IEEE 754	IEEE 754	Float
double	64	IEEE 754	IEEE 754	Double
void	-	-	-	Void

3 Aula P03 — Hierarquias simples

Sumário:

- Construção de objectos
- Realização de hierarquias de classes simples
- Polimorfismo
- Overloading e overriding de métodos

Resumo:

Herança

- Com a herança podemos facilmente evitar a duplicação de código.
- Uma subclasse `extends` a sua superclasse (relação IS-A, que só tem um sentido e é transitiva).
- Uma subclasse herda todas as variáveis de instância e métodos `public` e `protected` da superclasse, mas não herda os `private`.
- Os métodos herdados podem ser redefinidos (`override`), mas as variáveis de instância apenas podem ser especializadas, embora isso raramente seja necessário.
- Quando um método herdado é redefinido (`override`) e é chamado com uma instância da subclasse, é a versão da subclasse que é chamada.

Polimorfismo

- Quando se define uma superclasse para um grupo de classes, qualquer subclasse dessa classe pode aparecer onde era esperada a superclasse (por exemplo, se esperar um animal, posso colocar lá um cão, que ele vai ter todas as características de um animal).
- Em particular, se tiver um método que tem um argumento declarado como pertencendo à superclasse, posso passar-lhe qualquer instância de uma sua subclasse. Mesmo que a subclasse ainda não estivesse definida no início. Por exemplo, se definir novas subclasses de animal, tudo o que usar o animal continua a funcionar, sem ter que alterar o código.

Overriding $\neq$ overloading

- Quando se faz `override` de um método, está-se a assinar um contrato:
  - Os argumentos têm que ser os mesmos, e os tipos de retorno têm que ser compatíveis.
  - O método não pode ser menos acessível (por exemplo, não pode ser `public` e passar a `private`).
- Quando se faz `overload` cria-se um novo método. São dois métodos com o mesmo nome mas assinaturas diferentes. Têm o mesmo nome, mas não têm nada a ver com herança e polimorfismo. São dois métodos diferentes. Por exemplo, o + para somar números ou para concatenar strings.
  - É obrigatório mudar a lista de argumentos.
  - O tipo de retorno pode ser diferente, desde que também se mudem os argumentos.

  - Pode-se mudar os níveis de acesso em qualquer direcção (porque é um método diferente, não tem que cumprir nenhum contrato).

Java

- As classes internas não devem ser usadas fora da classe onde são definidas.
- Uma classe sem a keyword `public` não é pública e só pode ser usada/subclassada por outras classes dentro da mesma package.
- Uma classe não pode ter subclasses se for `final`. Pode-se declarar toda a classe ou apenas alguns métodos como `final`. Isto acontece com classes que não queremos que possam ser modificadas, em geral por questões de segurança.
- Se uma classe só tiver construtores `private` não pode ter subclasses.
- Usa-se `super(<args>)` para chamar o construtor da superclasse. Nos construtores, a chamada ao método da superclasse tem que estar no início.
- Usa-se `super.nomeMetodo(<args>)` para chamar o método `nomeMetodo` da superclasse.
- Usa-se `this` para referir o próprio objecto. Pode ser `this(<args>)` para chamar outro dos "meus" construtores ou `this.nomeMetodo(<args>)` para chamar o "meu" método `nomeMetodo`.
- O Java tem a ferramenta `javadoc` que cria automaticamente documentação em html para as classes.
- `@param` — é um marcador para a documentação em Java que indica os parâmetros do método.
- `@return` — é um marcador para a documentação em Java que indica o tipo de retorno do método.
- `@see` — é um marcador para a documentação em Java que diz que devem ser consultados outros métodos.
- `@Override` — diz que esté a sobrepor-se a um método herdado (para o caso de me enganar a escrever o nome, por exemplo).
- `@SuppressWarnings("nls")` — suprime warnings do compilador relacionados com caracteres acentuados.

Nestes exercícios

- Os vários construtores não aparecem no UML.

4 Aula P04 — Hierarquias simples v2

Sumário:

- Packages
- Classes abstractas
- Overloading e overriding de métodos

Resumo:

Unix

- `echo $CLASSPATH` — mostrar o valor da variável de ambiente `CLASSPATH`.
- `export CLASSPATH=.` — alterar a variável para passar a incluir a directoria corrente.

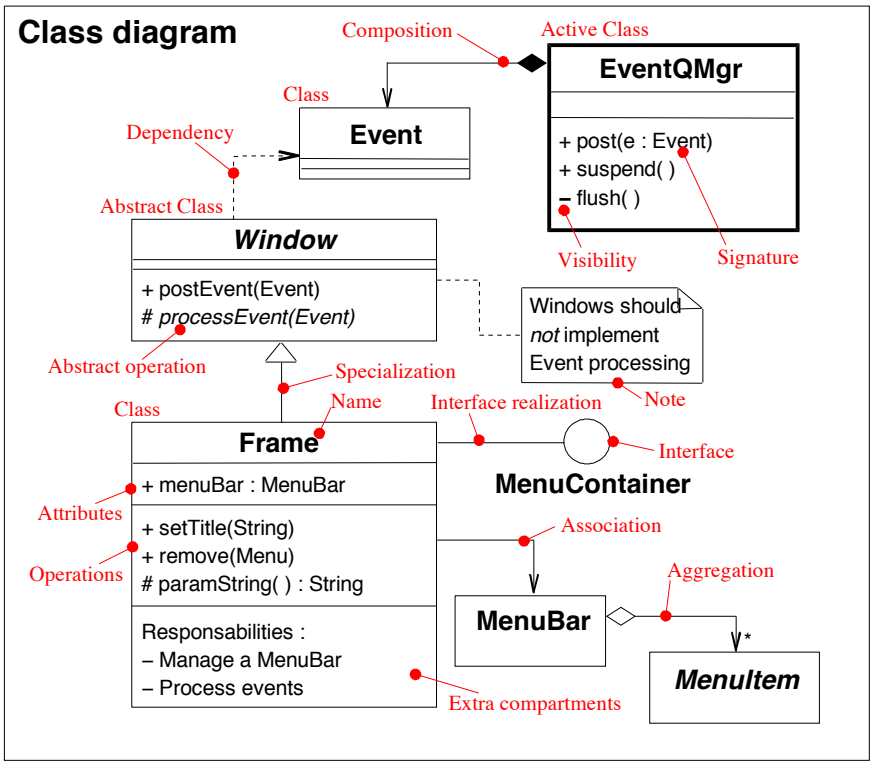
CVS

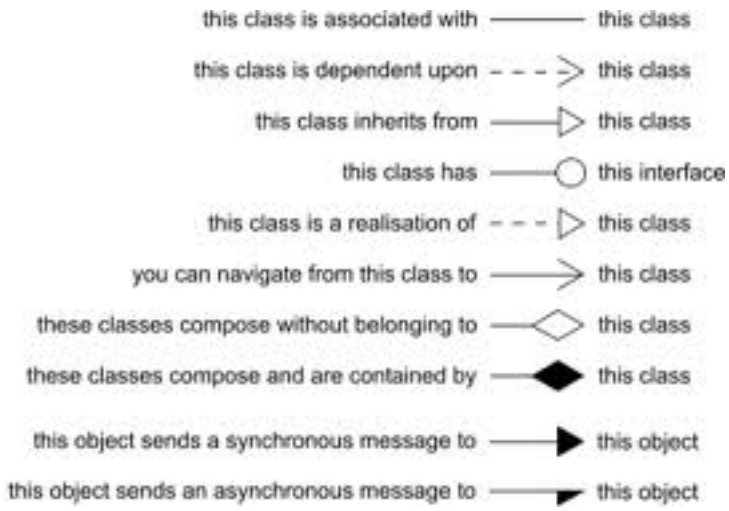
- `cvs ci` — check in = commit, opção `-m` para dar mensagem explicativa, que depois aparece nos logs.
- `cvs up` — update, para ficar com a versão mais recente.
- `cvs add` — adiciona ficheiros ao repositório.
- `p` — patch

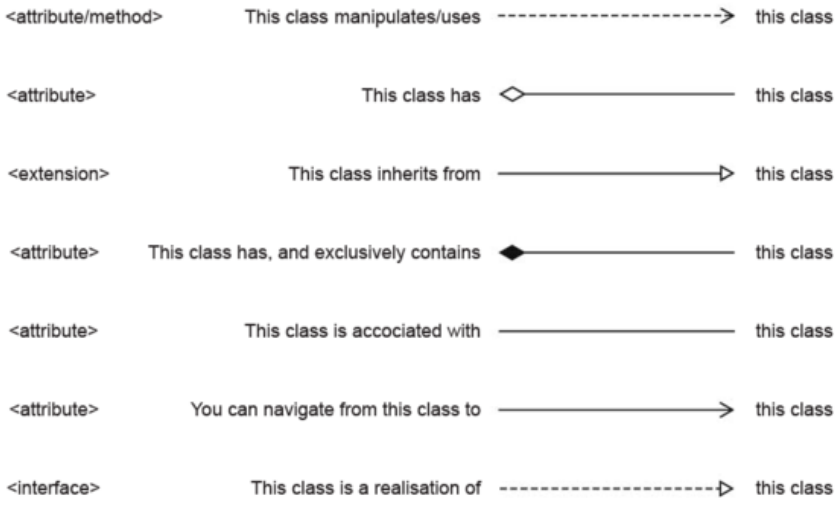
UML

- Classes abstractas em itálico em computador.
- Feitas à mão, as classes abstractas, interfaces e enumerados aparecem com «`abstract`», «`interface`» e «`enum`», respectivamente.
- Agregação vs composição (há quem use sempre só associação com aridade).
- Agregação: os As têm Bs, mas os Bs fazem sentido fora do âmbito dos As, por exemplo, uma escola tem pessoas.
- Composição: os As são feitos de Bs. Mas quando os As são destruidos os Bs também desaparecem (mais complicado, menos usado).
- Nas ligações, nas pontas são os papeis e as aridades, no meio é a relação.

Imagens úteis encontradas na net







Java

- `package` nome — indica que a classe que vai ser definida está dentro da package com o nome dado. O ficheiro `.java` deve estar dentro de uma directoria com o nome da package.
- `import nomepackage.*` — importa todas as classes que estão na package nomeada. Podia dizer explicitamente o nome de cada classe a importar. Usa-se na classe que tem o main.
- `void` $\neq$ `null`: `void` é um tipo, `null` é um valor. `void` é usado como tipo de retorno dos métodos, por exemplo. `null` é usado para fazer comparações, por exemplo para saber se uma variável foi inicializada.
- As classes abstractas podem ter métodos abstractos e métodos não abstractos. As classes abstractas não podem ter instâncias. Os métodos abstractos têm que ser redefinidos nas subclasses (que não sejam também elas abstractas). As classes não abstractas só podem ter métodos não abstractos (o que faz sentido, pois podem ter instâncias e nesse caso os métodos abstractos não iriam ser redefinidos e as instâncias não os saberiam aplicar).

Nestes exercícios

- Na `MagicLamp` o método `rub` retorna um `Genie`, que é uma classe abstracta. É a primeira vez que retornamos algo que não seja um tipo primitivo do Java (a não ser nos construtores). Aqui, incrementa-se as esfregadelas antes do `if`, pois queremos retornar os génios e depois de retornar não podemos fazer mais nada. Retornamos um génio ou demónio, dependendo das condições actuais da lâmpada. No UML existem as relações de dependência todas por causa deste método.
- Na `MagicLamp` o primeiro `equals` faz override do método herdado e chama o método de baixo, que foi definido usando overloading. Não é uma chamada recursiva, é uma chamada a outro método por causa do overloading. Nas outras aulas tínhamos o código todo no método de cima, aqui separámos por questões de eficiência, se considerarmos que fazemos muitas comparações entre lâmpadas.
- A classe `Genie` tem que existir porque o método `rub` precisa de um tipo de retorno. Eventualmente até podia não ser abstracta, mas para poder retornar um génio de alguma das sua subclasses sem saber à partida qual, tem que ter uma classe mais genérica para poder ter o tipo correspondente.
- No `Genie` o construtor é `protected` para não se poder criar instâncias, só as subclasses é que o podem usar.
- No `Genie` o método `doGrantWish` podia ser abstracto e depois redefinido obrigatoriamente nas várias subclasses, nem que fosse com um print.
- O `RecyclableDemon` redefine o método `canGrantWish` para dizer que concede desejos desde que não tenha sido reciclado.
- Na aplicação inicializa-se um array de génios com as várias chamadas ao método `rub` da lâmpada, com o número de desejos relevante. No resultado da execução vê-se que o número de desejos de cada génio nem sempre é respeitado.

5 Aula P05 — Hierarquias; Interfaces

Sumário:

- Interfaces
- Classes abstractas

Resumo:

UML

- As interfaces aparecem em itálico ou com «`interface`».
- A seta para as interfaces é a tracejado e com a ponta como a do is-a.

Java

- As interfaces não são classes, por isso não podem ter instâncias.
- As interfaces não podem ter atributos.
- As interfaces apenas declaram os métodos, não os implementam.
- As classes podem *implementar* interfaces, as interfaces podem *estender* outras interfaces.
- Pode existir uma hierarquia de interfaces, mas só de interfaces, nunca há classes nem instâncias misturadas.
- As classes abstractas e as interfaces são semelhantes, pois definem uma interface e adiam a implementação dos métodos para mais tarde e nenhumas podem ter "instâncias directas". No entanto, nas classes abstractas podem-se implementar alguns métodos, ao passo que nas interfaces os métodos só podem ser declarados. As classes abstractas podem ter atributos e as interfaces não podem.
- Se uma classe implementa uma interface, tem que implementar todos os seus métodos. No entanto, pode ser com métodos herdados de outras classes. Ou seja, as instâncias da classe de baixo têm que saber responder a todos os métodos da interface, mas pode ser à custa de métodos herdados de classes mais acima na hierarquia.
- Se uma classe implementar uma interface mas não definir todos os métodos da interface, tem que ser abstracta. Os métodos que não forem definidos são automaticamente considerados abstractos.
- Tabela comparativa entre classes, classes abstractas e interfaces:

	Classe	Classe Abstracta	Interface
Pode ter instâncias	✓	×	×
Pode ter atributos	✓	✓	×
Pode declarar métodos sem os definir	×	✓	✓
Pode definir métodos	✓	✓	×

Nestes exercícios

- `Animal` e `NamedAnimal` são classes abstractas, que não podem ter instâncias.
- `Predator` e `Prey` são interfaces, só declaram métodos.
- `Dog extends NamedAnimal implements Predator` — a classe `Dog` herda de `NamedAnimal` e implementa a interface `Predator`.

- Métodos `caught` e `eat` nos cães e gatos não deviam estar nos predadores? Ver comentários de cão e gato.
- Dois ciclos `for` da aplicação, variável de controlo declarada antes, para poder ser usada nos dois ciclos.

6 Aula P06 — Classes internas

Sumário:

- Classes internas
- Interfaces `Comparable` e `Comparator`
- Interfaces `Iterable` e `Iterator`

Resumo:

UML

- As classes internas representam-se com "lolipops" com cruzes.
  O "lolipop" fica na classe maior.
- As constantes (`final`) escrevem-se com maiúsculas.
- Os atributos de classe (`static`) são sublinhados.
- Não é preciso colocar no UML as interfaces de Java implementadas, a não ser que se queira alterar alguma coisa.

Java

- Usam-se classes internas para esconder o seu código do exterior. Quando se pretende que uma classe implemente de várias maneiras uma interface, podem definir-se várias classes e cada uma implementa a interface à sua maneira. Se essas classes não forem usadas no resto do programa, podem ser classes internas e assim o seu código fica escondido do exterior.
- As classes internas podem usar todos os métodos e atributos da classe externa, mesmo que estes sejam `private`, tal como se tivessem sido declarados na classe interna.
- Uma instância da uma classe interna não `static` fica ligada à instância da classe externa que a criou. A classe externa pode criar instâncias das suas classes internas da mesma forma que cria instâncias de outras classes.
- Cria-se uma classe interna `static` quando se sabe que ela não vai precisar de aceder a variáveis nem métodos da classe externa que não sejam `static` e se pretende ter apenas uma versão sincronizada para todas as instâncias da classe externa. Uma vantagem adicional de dizer que são `static` é poupar memória, pois só existe uma classe interna para todas as instâncias da classe externa.
- Podem-se criar classes internas anónimas quando só se quer usar uma vez. É como se fosse uma lambda function.
- Podem-se criar classes internas nas classes externas, num dos seus métodos ou num dos seus blocos de código.
- É possível criar instâncias de classes internas a partir de código exterior à classe externa, mas não é muito comum.

```
class Foo {
  public static void main (String[] args) {
    MyOuter outerObj = new MyOuter();
    MyOuter.MyInner innerObj = outerObj.new MyInner(); }
}
```

- The **`Throwable`** class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause. For the purposes of compile-time checking of exceptions, `Throwable` and any subclass of `Throwable` that is not also a subclass of either `RuntimeException` or `Error` are regarded as checked exceptions.
- `public class Throwable extends Object implements Serializable`
- Os `Throwable` podem ser `Error` ou `Exception`.
- Os `Error` não estão previstos na execução do programa.
- An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The `ThreadDeath` error, though a "normal"condition, is also a subclass of `Error` because most applications should not try to catch it. A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, `Error` and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.
- The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch. The class `Exception` and any subclasses that are not also subclasses of `RuntimeException` are checked exceptions. Checked exceptions need to be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.
- A estrutura é a seguinte:

```
try {
  ...
}
catch(tipoDeThrowable) {
  ...
}
```

- Pode haver vários `catch`, normalmente do mais específico para o mais genérico, pois são executados por ordem.

Nestes exercícios

- A interface `java.lang.Comparable<T>` usa-se quando os objectos têm uma maneira "natural" de serem comparados, por exemplo números, strings, datas, etc.
- Usam-se classes internas para os Comparators por conveniência da ocultação de código.
- Na tabela com classes internas, `MAX_COMPARATOR` e `LENGTH_COMPARATOR` são `final` e portanto são constantes e devem aparecer com maiúscula.
- Também são `static`, por isso são variáveis de classe e não de instância, ou seja, existem independentemente de haver ou não instâncias de `Table` e são as mesmas para todas as instâncias da classe `Table`, por isso poupa-se memória.
- As classes internas `MaxComparator` e `LengthComparator` são criadas num contexto estático, por isso não ficam ligadas a nenhuma instância de `Table`.

7 Aula P07 — Entradas e Saídas

Sumário:

- Entradas e saídas
- Serialização de objectos em Java

Resumo:

UML

- Não é preciso colocar as interfaces do Java nos exercícios de avaliação.

Java

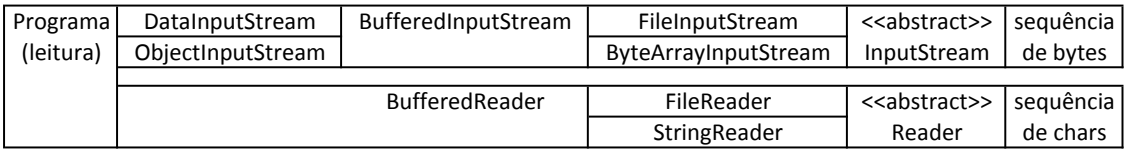
- Os objectos têm estado e comportamento. O comportamento é definido pela classe, mas o estado faz parte de cada objecto individual. É o estado dos objectos individuais que precisa de ser gravado.
- Se os dados forem ser usados apenas pelo programa de Java que os gerou, usar `serializable`. É a opção mais segura, mas não vai ser fácil para um humano perceber o ficheiro. Quando um objecto é serializado, todos os objectos que ele refere a partir de variáveis de instância são também serializados automaticamente. A serialização grava todo o grafo de objectos, com todos os objectos referenciados nas variáveis de instância a partir do objecto a serializar.
- Se os dados puderem ser usados por outros programas, usar ficheiros de texto, por exemplo com um objecto por linha e os atributos separados por vírgulas. O ficheiro fica fácil de ler, mas a leitura e escrita ficam mais sujeitas a erros.
- Serializar objectos:
  1. Criar um `FileOutputStream`, que sabe ligar-se a ficheiros e criar ficheiros. Se o ficheiro não existir, é criado automaticamente.
     `FileOutputStream fileStream = new FileOutputStream("NomeFich");`
  2. Criar um `ObjectOutputStream`, que sabe escrever objectos, mas não se consegue ligar-se directamente a um ficheiro. A isto chama-se `chaining` de uma stream para outra.
     `ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);`
  3. Escrever os objectos, serializando-os.
     `objectStream.writeObject(meuObjecto);`
  4. Fechar o `ObjectOutputStream`, o que fecha os outros automaticamente.
     `objectStream.close();`
- A API de I/O do Java tem as *connection streams*, que representam uma ligação a uma fonte ou a um destino (ficheiro, socket, etc), e as *chain streams*, que não se conseguem ligar elas próprias e precisam de ser "acorrentadas" (*chained*) a uma *connection stream*. Normalmente, são necessárias pelo menos duas *streams* para fazer alguma coisa útil: uma para representar a ligação e outra para chamar os métodos. Isto porque as *connection streams* são a um nível demasiado baixo, por exemplo, escrevem *bytes*. Para escrevermos objectos precisamos da *chain stream*.

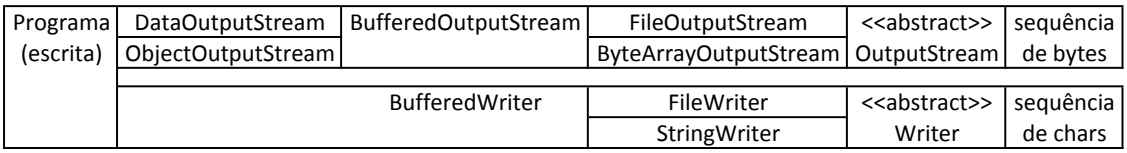
- A interface `serializable` não tem métodos para implementar, apenas indica que os objectos desse tipo podem ser serializados. Está no `java.io`, por isso é preciso fazer `import java.io.*`.
- Quando uma classe é serializável, todas as classes das suas variáveis têm que ser serializáveis, senão dá erro de compilação. Se não quiser serializar alguma das variáveis, é preciso dizer que ela é `transient`, para ser ignorada durante a serialização.
- Se o mesmo objecto for atingido mais do que uma vez no grafo de objectos a serializar, o Java apercebe-se e só o grava uma vez.
- Des-serializar objectos:
  1. Criar um `FileInputStream`, se o ficheiro não existir dá erro.
     `FileInputStream fileStream = new FileInputStream("NomeFich");`
  2. Criar um `ObjectInputStream`.
     `ObjectInputStream os = new ObjectInputStream(fileStream);`
  3. Ler os objectos, tendo o cuidado de não ler mais do que os que foram escritos, pela ordem em que foram escritos. São lidos objectos, depois é preciso dizer de que tipo são para os poder usar. Mas é preciso que as classes de todos os objectos que vão ser lidos estejam disponíveis para a JVM.
     `Object meuObjecto = os.readObject();`
  4. Pode ser necessário fazer `cast` dos objectos. Convém ter em atenção que isto não faz o construtor voltar a correr, queremos que o objecto seja restaurado para o estado em que estava quando foi serializado e não para como estava quando foi criado. Se alguma das superclasses não for serializável, o seu construtor e o de todas as suas superclasses vai correr. Isto significa que estas vão reinicializar o seu estado. As variáveis de instância ficam com os valores do estado serializado. As variáveis `transient` ficam a `null` ou com os valores por omissão dos tipos respectivos. As variáveis da classe (`static`) não são serializadas e quando o objecto é recuperado passa a depender das variáveis de classe que existam nessa altura.
     `Gato g1 = (Gato) meuObjecto;`
  5. Fechar o `ObjectInputStream`.
     `os.close();`
- Escrever uma `String` para um ficheiro de texto é semelhante a escrever um objecto, mas usando um `FileWriter` em vez de um `FileOutputStream`, sem o ligar a um `ObjectOutputStream`.

```
import java.io.*;
class WriteAFile {
  public static void main (String[] args) {
    try {
      FileWriter writer = new FileWriter(``Foo.txt'');
      writer.write(``hello foo!'');
      writer.close();
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

- A classe `java.io.File` representa um ficheiro no filesystem, mas os seus metadados (nome, se é uma directoria, permissões, etc.), e não representa o conteúdo do ficheiro.
- Os `BufferedWriter` são mais eficientes que os `FileWriter` porque vão escrevendo para um buffer e só quando este está cheio é que escrevem para disco. São mais eficientes porque a escrita em memória é muito mais eficiente do que a escrita em disco. Mas é possível fazer `flush()` quando se quer antecipar a escrita para disco.
  `BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));`

- Algumas classes para leitura e escrita em Java:





8 Aula P08 — Collections; Escrita de testes JUnit

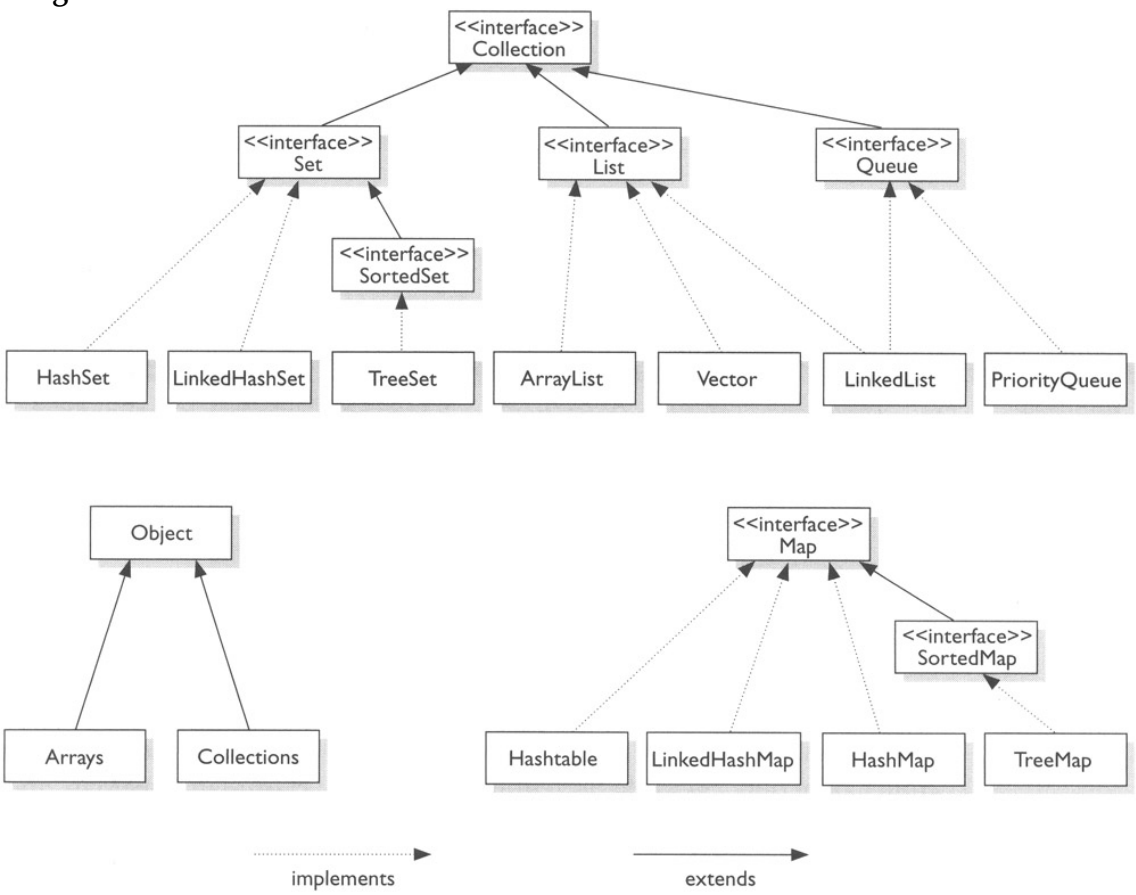
Sumário:

- Collections em Java
- Escrita de testes JUnit

Resumo:

Java

- **Imagem útil encontrada na net**



- Em baixo à esquerda, `Collections` e `Arrays` (plural) são classes que implementam "métodos úteis", todos eles `static` e polimórficos, como por exemplo `sort` ou `search`. Os métodos recebem como argumentos objectos do tipo `Collection` ou `Map`.
- As `Collection` correspondem a aglomerados de objectos e os `Map` correspondem a aglomerados de objectos organizados pela sua chave.
- Os `Set` não têm elementos repetidos e podem ser ordenados ou não.
- As "folhas" de cada "árvore" correspondem às várias implementações das interfaces acima.
- A `Hashtable` foi substituida pelo `HashMap` porque está desactualizada porque era pouco eficiente, pois pedia locks das tabelas completas. O `HashMap` não suporta concorrência. As versões concorrentes mais recentes pedem lock apenas dos buckets necessários em cada momento e por isso são mais eficientes.

JUnit

- **Imagem útil e texto encontrados na net (Junit cheatsheet)**

Method annotations:

tag	description

`@Test`
`@Test(timeout = time)`
`@Test(expected = exception.class)`
	Turns a public method into a JUnit test case. Adding a timeout will cause the test case to fail after **time** milliseconds. Adding an expected exception will cause the test case to fail if **exception** is not thrown.

`@Before`	Method to run before every test case
`@After`	Method to run after every test case
`@BeforeClass`	Method to run once, before any test cases have run
`@AfterClass`	Method to run once, after all test cases have run

Assertion methods:

method	description
`assertTrue(test)`	fails if the Boolean test is `false`
`assertFalse(test)`	fails if the Boolean test is `true`
`assertEquals(expected, actual)`	fails if the values are not equal
`assertSame(expected, actual)`	fails if the values are *not* the same (by ==)
`assertNotSame(expected, actual)`	fails if the values are the same (by ==)
`assertNull(value)`	fails if the given value is *not* `null`
`assertNotNull(value)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

Each method can also be passed a string to display if it fails, e.g.

```
assertEquals("message", expected, actual)
```

- A unit test is a piece of code written by a developer that executes a specific functionality in the code which is tested. A unit test targets a small unit of code, e.g. a method or a class, (local tests). Unit tests ensure that the code works as intended. They are also very helpful to ensure that the code still works as intended in case you need to modify code for fixing a bug or extending functionality. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.
- The entire goal is FAILURE ATOMICITY – the ability to know exactly what failed when a test case did not pass.
- Tests should be self-contained and not care about each other: JUnit assumes that all test methods can be executed in an arbitrary order. Therefore tests should not depend on other tests.
- You cannot test everything! Instead, think about: boundary cases, empty cases, error cases, behaviour in combination (but not to excess).
- Each test case should test ONE THING: 10 small tests are better than 1 test 10x as large. Rule of thumb: 1 assert statement per test case. Try to avoid complicated logic.
- Torture tests are ok, but only in addition to simple tests.
- JUnit best practices:
  - Use descriptive test names.
  - Add a default timeout to every test.
  - Use private methods to get rid of redundant test code.
- Create test suites using `@RunWith` and `@Suite.SuiteClasses` to run tests for several classes at once. If you have several test classes you can combine them into a test suite. Running a test suite will execute all test classes in that suite in the specified order.
- Build quick arrays and collections using array literals:

  - `int[] quick = new int[] {1, 2, 3, 4};`
  - `List<Integer> list = Arrays.asList(7, 4, -3, 18);`
  - `Set<Integer> set = new HashSet<Integer>(Arrays.asList(5, 6, 10));`
- O nome da classe de teste deve terminar com `Test`. Os nomes dos métodos de teste devem começar com `test`. Os `runners` executam todos os métodos cujo nome comece por `test`.

9 Aula P09 — Padrão de desenho “Composite”

Sumário:

- Padrão de desenho “Composite”

Resumo:

Princípios de desenho

- Os padrões de desenho são soluções para problemas de desenho de software que se encontram frequentemente no desenvolvimento de aplicações reais.
- Os 23 padrões do livro dos “Gang of Four” geralmente são considerados a fundação para os outros padrões de desenho. São caracterizados em três grupos: de criação, estruturais e comportamentais.
- Uma classe deve ter apenas uma razão para mudar. Cada responsabilidade de uma classe é uma área potencial de mudança. Mais do que uma responsabilidade significa mais do que uma razão para mudar.

Padrões de desenho

From book: Design Patterns, Gamma, Helm, Johnson, Vlissides

- Design patterns are classified by two criteria. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioural purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

- The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static — fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns"are those that focus on class relationships (Factory Method, Adapter, Interpreter, Template Method). Note that most patterns are in the Object scope.

- Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes.
Builder	Separates object construction from its representation.
Factory Method	Creates an instance of several derived classes.
Prototype	A fully initialized instance to be copied or cloned.
Singleton	A class of which only a single instance can exist.
Structural Patterns	
Adapter	Match interfaces of different classes.
Bridge	Separates an object's interface from its implementation.
Composite	A tree structure of simple and composite objects.
Decorator	Add responsibilities to objects dynamically.
Facade	A single class that represents an entire subsystem.
Flyweight	A fine-grained instance used for efficient sharing.
Proxy	An object representing another object.
Behavioural Patterns	
Chain of Responsibility	A way of passing a request between a chain of objects.
Command	Encapsulate a command request as an object.
Interpreter	A way to include language elements in a program.
Iterator	Sequentially access the elements of a collection.
Mediator	Defines simplified communication between classes.
Memento	Capture and restore an object's internal state.
Observer	A way of notifying change to a number of classes.
State	Alter an object's behavior when its state changes.
Strategy	Encapsulates an algorithm inside a class.
Template Method	Defer the exact steps of an algorithm to a subclass.
Visitor	Defines a new operation to a class without change.

Relações entre os padrões de desenho

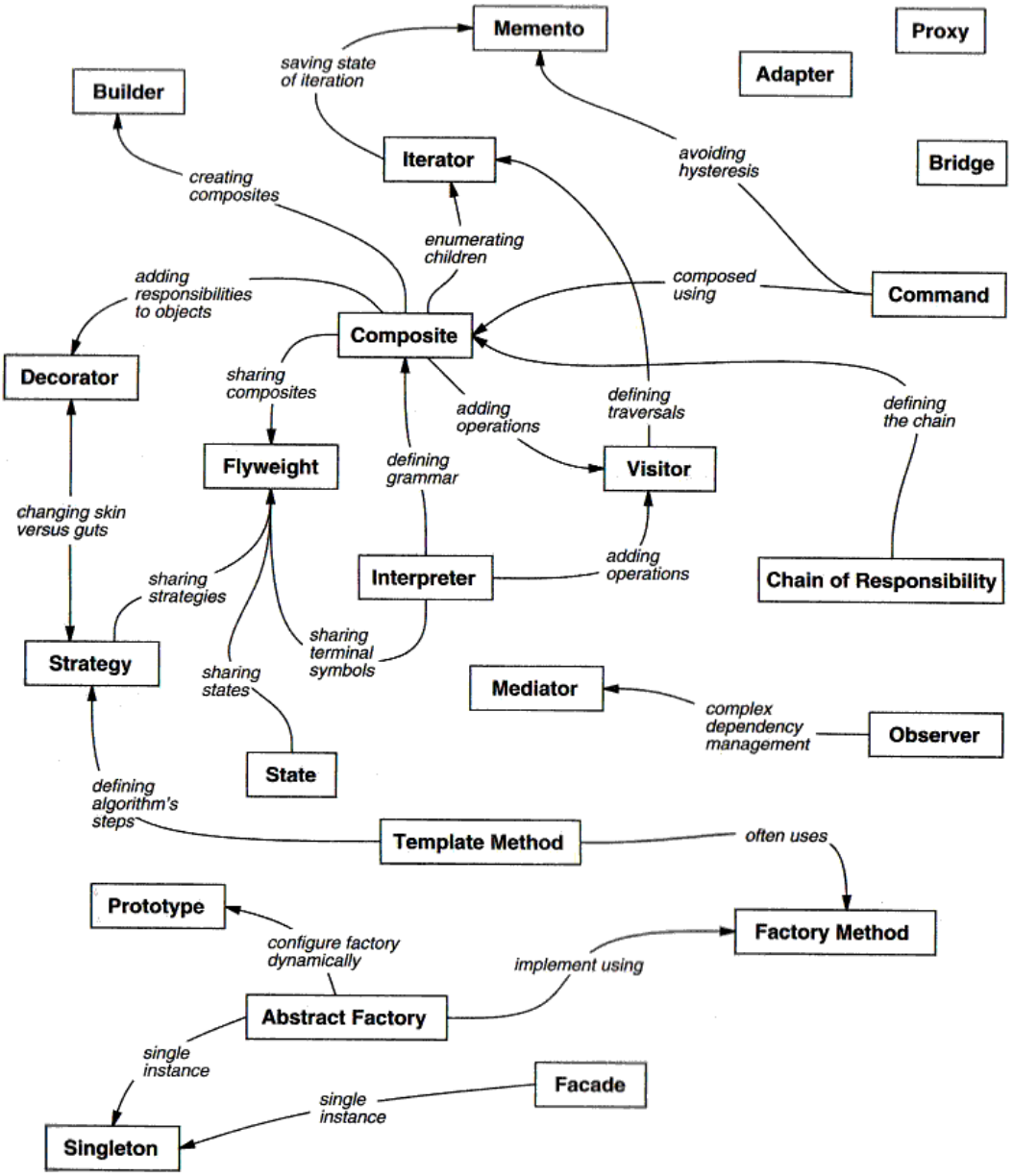


Figure 1.1: Design pattern relationships

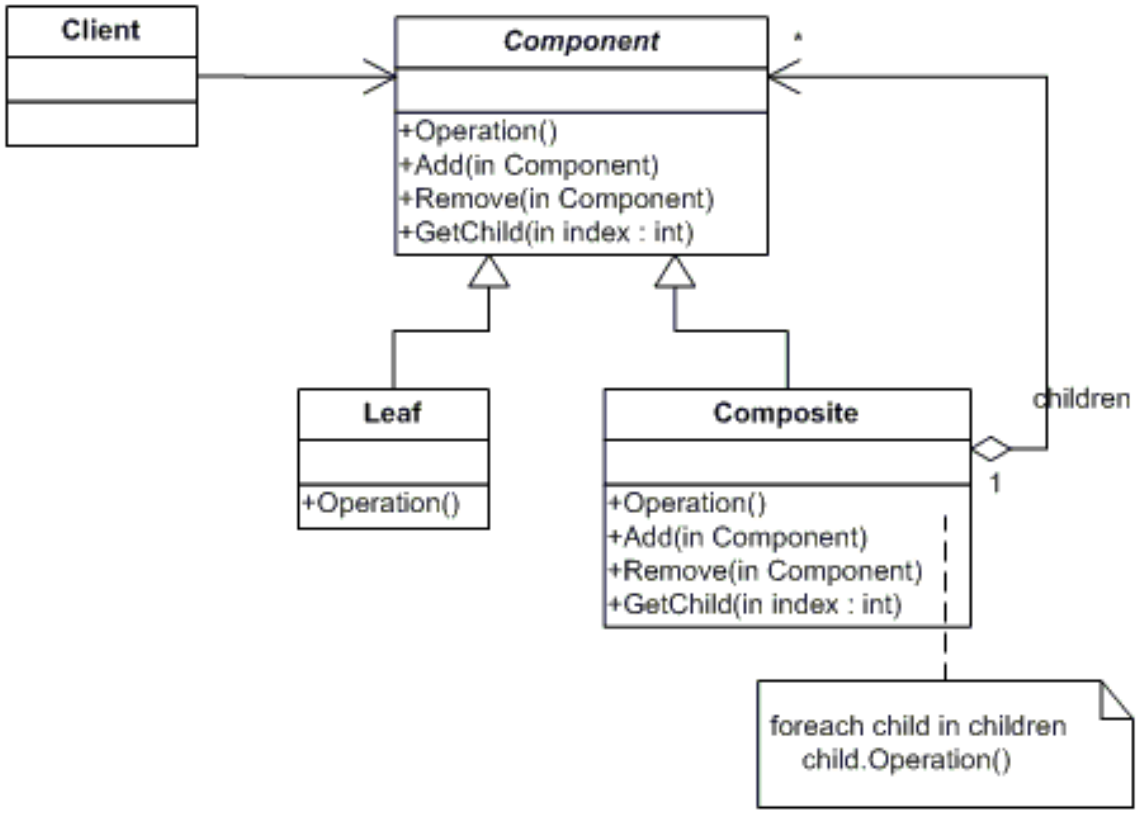
Composite — A tree structure of simple and composite objects.

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Use this pattern when:
  - You want to represent part-whole hierarchies of objects.
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- The classes and/or objects participating in this pattern are:
  - Component (DrawingElement) — Declares the interface for objects in the compo-

sition. Implements default behavior for the interface common to all classes, as appropriate. Declares an interface for accessing and managing its child components. (optional) Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

- Leaf (PrimitiveElement) — Represents leaf objects in the composition. A leaf has no children. Defines behavior for primitive objects in the composition.
- Composite (CompositeElement) — Defines behavior for components having children. Stores child components. Implements child-related operations in the Component interface.
- Client (CompositeApp) — Manipulates objects in the composition through the Component interface.

- UML



10 Aula P10 — Padrões de desenho “Strategy” e “State”

Sumário:

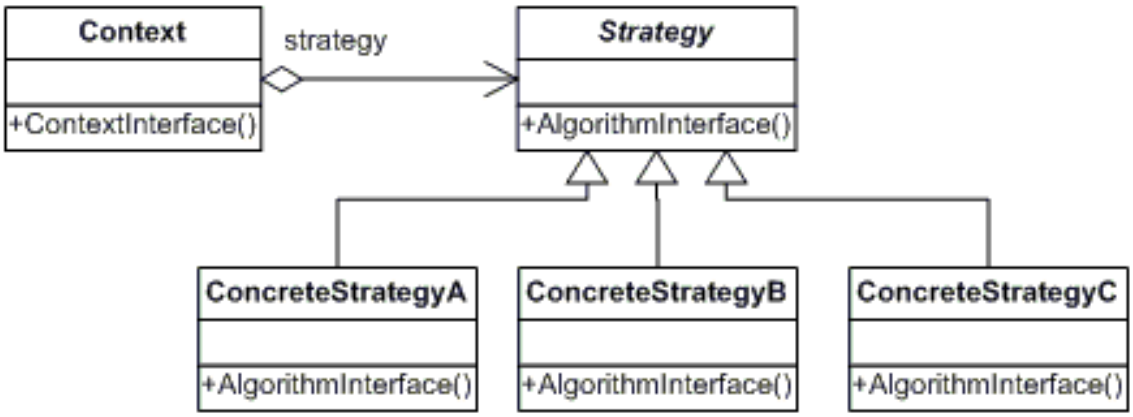
- Padrões de desenho “Strategy” e “State”

Resumo:

From book: Design Patterns, Gamma, Helm, Johnson, Vlissides

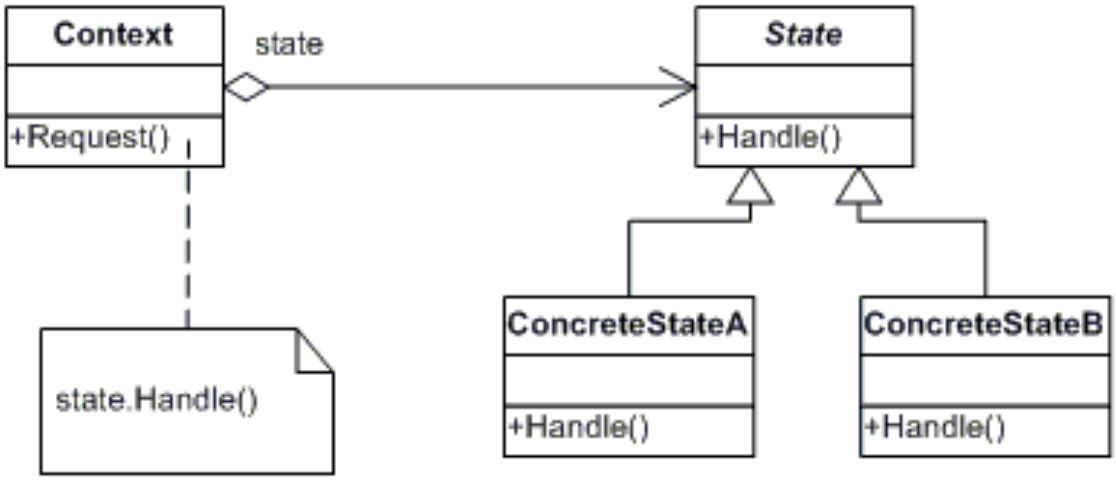
Strategy — Encapsulates an algorithm inside a class.

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Use this pattern when:
  - Many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
  - You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
  - An algorithm uses data that clients shouldn’t know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviours, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
- The classes and/or objects participating in this pattern are:
  - Strategy (SortStrategy) — declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
  - ConcreteStrategy (QuickSort, ShellSort, MergeSort) — implements the algorithm using the Strategy interface.
  - Context (SortedList) — is configured with a ConcreteStrategy object. Maintains a reference to a Strategy object. May define an interface that lets Strategy access its data.
- UML



State — Alter an object's behaviour when its state changes.

- Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- Use this pattern when:
  - An object's behaviour depends on its state, and it must change its behaviour at runtime depending on that state.
  - Operations have large, multipart conditional statements that depend onthe object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.
- The classes and/or objects participating in this pattern are:
  - Context (Account) — Defines the interface of interest to clients. Maintains an instance of a ConcreteState subclass that defines the current state.
  - State (State) — Defines an interface for encapsulating the behaviour associated with a particular state of the Context.
  - Concrete State (RedState, SilverState, GoldState) — Each subclass implements a behaviour associated with a state of Context.
- UML



Comentários

- Vantagens de ter o Strategy (Disciplina, MetodoAvaliacao, MetTeorico, MetPratico) relativamente a ter várias subclasses (Disciplina, DiscTeorica, DiscPratica): as várias estratégias são reutilizáveis por várias classes; assim uma disciplina pode trocar de método de avaliação em runtime, mas uma instância não pode trocar a sua classe; se uma classe tem vários Strategy é fácil fazer combinações (Disciplina, MetodoAvaliacao, TipoAulas, CorpoDocente), mas se fosse subclasses tinha que haver muitas subclasses, uma para cada combinação. Pode fazer sentido ter subclasses, mesmo usando o Strategy, quando que quer limitar os comportamentos possíveis.
- Apesar de terem diagramas UML iguais, o Strategy e o State são diferentes. O State altera o estado das instâncias da classe que o usa. O State é um Strategy que permite mudar de estratégia em runtime.

Nestes exercícios

- O Strategy do exercício dos gatos está no Collections, que pode usar vários Comparators para fazer a ordenação. Percebia-se melhor se houvesse uma ordenação por idade, uma ordenação por nome, etc.
- No padrão State, criar uma classe para cada estado possível. A interface da classe abstracta deve ter um método para cada acção que pode ser executada e cada subclasse deve implementar esses métodos. Em alguns casos, a execução de um método pode levar à mudança de estado.
- No semáforo, se o State não fosse uma classe interna tinha que ter um atributo com o semáforo com que foi criado ou então cada método recebê-lo como argumento, ou então o setter do estado do semáforo tinha que poder ser usado pela classe State, sendo package protected.
- O semáforo e o State têm métodos com os mesmos nomes porque o semáforo delegou o tratamento das operações no State. Cada método do semáforo escreve qualquer coisa, chama o método correspondente do State e escreve mais qualquer coisa. Mas que é responsável por fazer o trabalho é o State. É mais fácil de perceber se os métodos tiverem o mesmo nome.
- Na minha solução para a máquina (avaliação), todas as subclasses de State são internas à máquina e isso facilita imenso a escrita do código, muito mais do que ter a classe State interna ao semáforo mas depois as subclasses serem externas, como está no wiki.
- Como fazer sem ser com `new` para cada mudança de estado? Na classe Semaforo/Maquina, ter variáveis do tipo State para cada um dos estados possíveis. Se os estados puderem ser iguais para todos os Semaforos/Maquinas, podem até ser constantes de classe (`static`). Se tiverem que ser diferentes para cada instância de Semaforo/Maquina, no construtor fazer `new` para cada um dos tipos de estado e atribuir a essas variáveis, e depois quando se muda de estado usar os valores dessas variáveis.

11 Aula P11 — Padrão de desenho "Visitor"

Sumário:

- Padrão de desenho "Visitor"

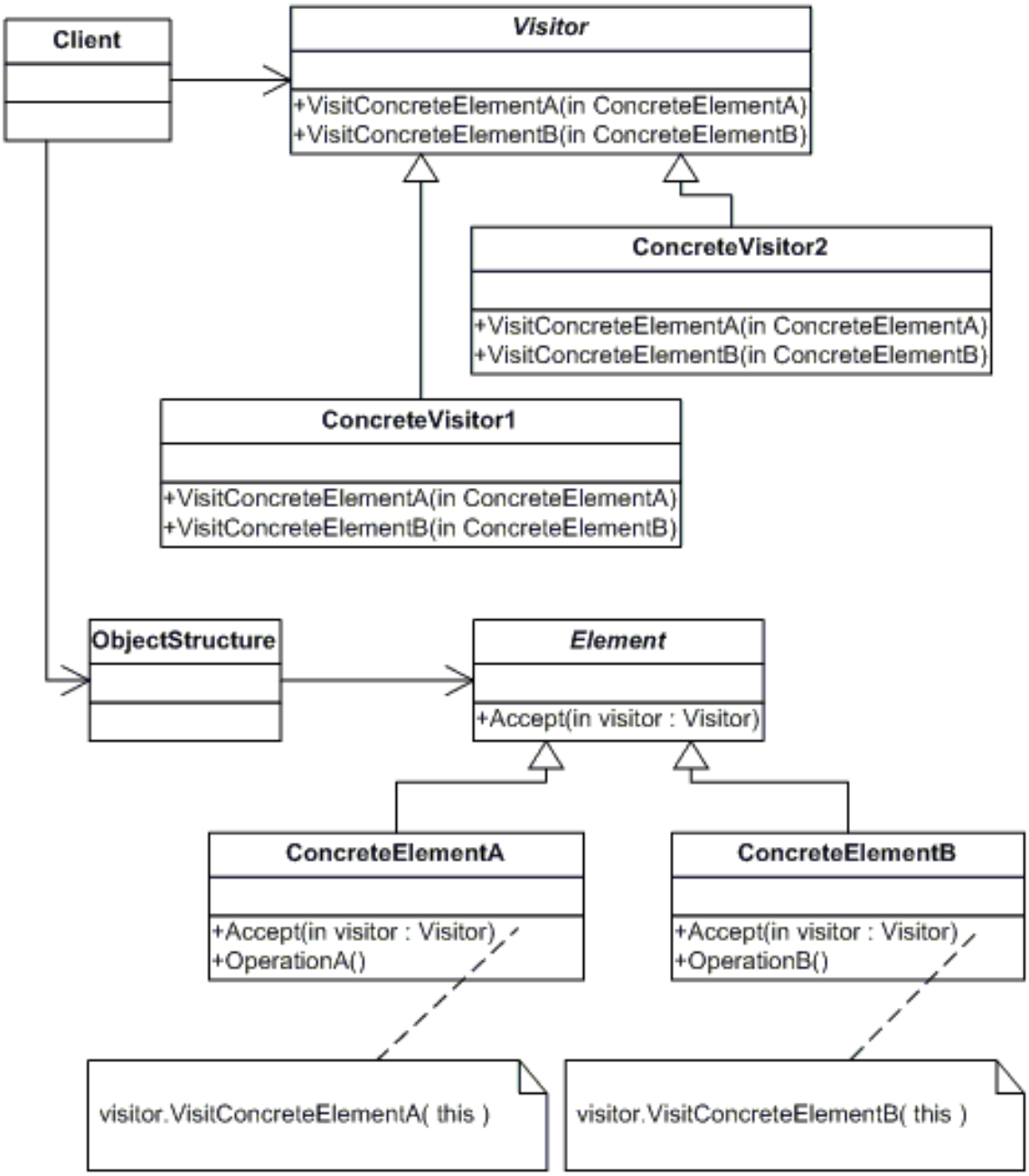
Resumo:

From book: Design Patterns, Gamma, Helm, Johnson, Vlissides

Visitor — Defines a new operation to a class without change.

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Use this pattern when:
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting"their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.
- The classes and/or objects participating in this pattern are:
  - Visitor (Visitor) — declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface.
  - ConcreteVisitor (IncomeVisitor, VacationVisitor) — implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
  - Element (Element) — defines an Accept operation that takes a visitor as an argument.
  - ConcreteElement (Employee) — implements an Accept operation that takes a visitor as an argument.
  - ObjectStructure (Employees) — can enumerate its elements. May provide a high-level interface to allow the visitor to visit its elements. May either be a Composite (pattern) or a collection such as a list or a set.

- UML



- Some of the benefits and liabilities of the Visitor pattern are as follows:
  - Visitor makes adding new operations easy. Visitors make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor. In contrast, if you spread functionality over many classes, then you must change each class to define a new operation.
  - A visitor gathers related operations and separates unrelated ones. Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses. That simplifies both the classes defining the elements and the algorithms defined in the visitors. Any algorithm-specific data structures can be hidden in the visitor.
  - Adding new ConcreteElement classes is hard. The Visitor pattern makes it hard to add new subclasses of Element. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class. Sometimes a default implementation can be provided in Visitor that

can be inherited by most of the ConcreteVisitors, but this is the exception rather than the rule. So the key consideration in applying the Visitor pattern is whether you are mostly likely to change the algorithm applied over an object structure or the classes of objects that make up the structure. The Visitor class hierarchy can be difficult to maintain when new ConcreteElement classes are added frequently. In such cases, it's probably easier just to define operations on the classes that make up the structure. If the Element class hierarchy is stable, but you are continually adding operations or changing algorithms, then the Visitor pattern will help you manage the changes.

– Visiting across class hierarchies. An iterator can visit the objects in a structure as it traverses them by calling their operations. But an iterator can't work across object structures with different types of elements. Visitor does not have this restriction. It can visit objects that don't have a common parent class. You can add any type of object to a Visitor interface.

12 Aula P12 — Padrões de desenho “Decorator” e “Adapter”

Sumário:

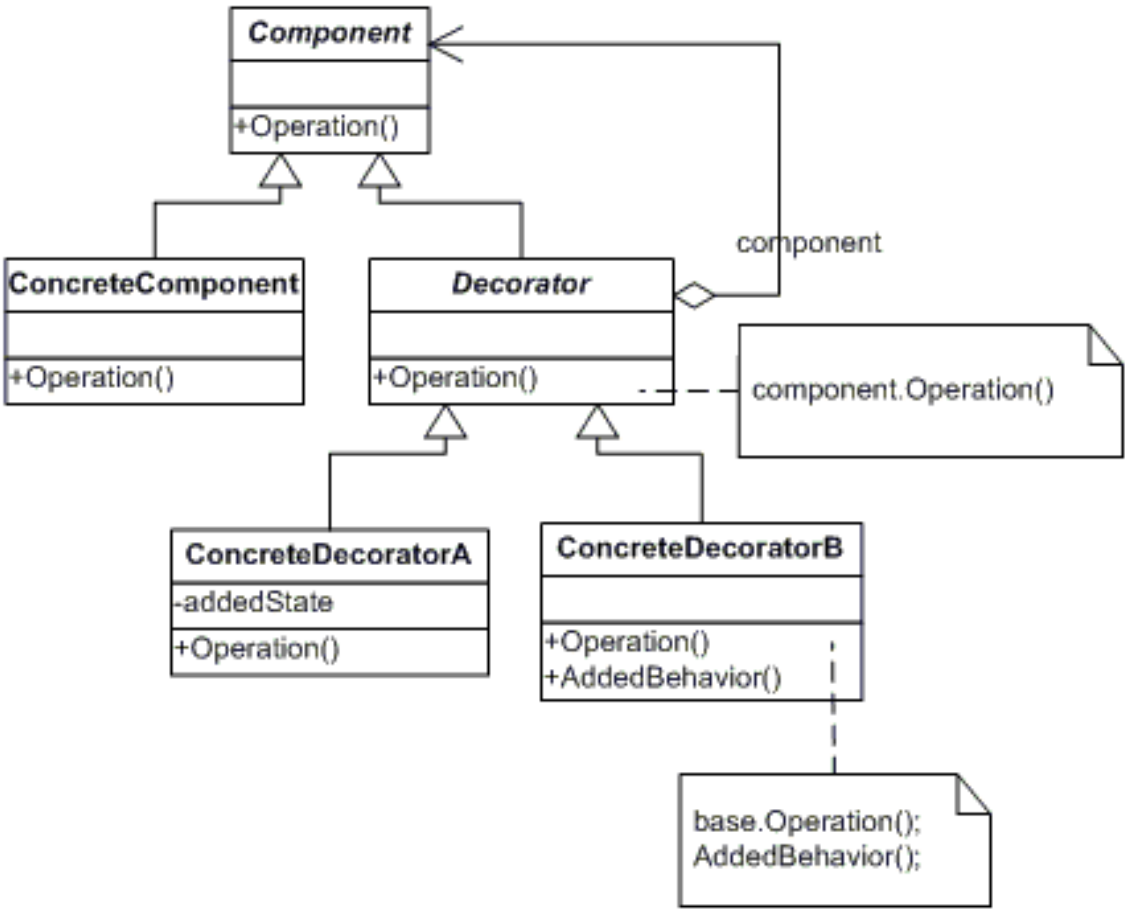
- Padrões de desenho “Decorator” e “Adapter”

Resumo:

From book: Design Patterns, Gamma, Helm, Johnson, Vlissides

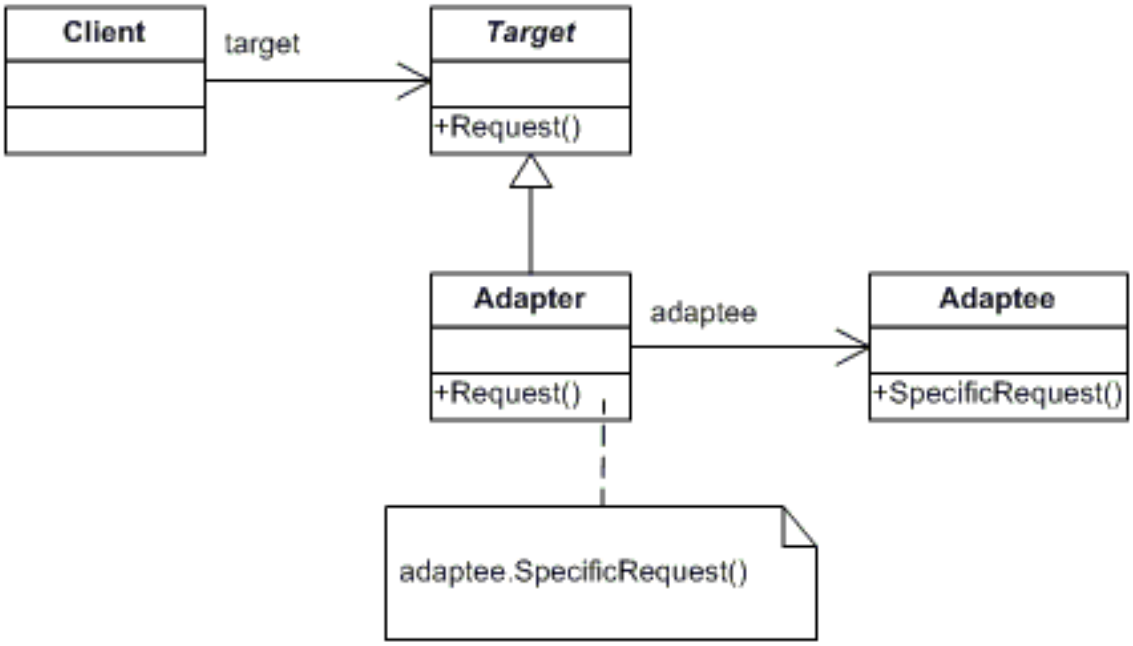
Decorator — Add responsibilities to objects dynamically.

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Use this pattern when:
  - You need to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - For responsibilities that can be withdrawn.
  - Extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.
- The classes and/or objects participating in this pattern are:
  - Component (LibraryItem) — defines the interface for objects that can have responsibilities added to them dynamically.
  - ConcreteComponent (Book, Video) — defines an object to which additional responsibilities can be attached.
  - Decorator (Decorator) — maintains a reference to a Component object and defines an interface that conforms to Component’s interface.
  - ConcreteDecorator (Borrowable) — adds responsibilities to the component.
- UML



Adapter — Match interfaces of different classes.

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Use this pattern when:
  - You want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - (Object adapter only) You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.
- The classes and/or objects participating in this pattern are:
  - Target (ChemicalCompound) — defines the domain-specific interface that Client uses.
  - Adapter (Compound) — adapts the interface Adaptee to the Target interface.
  - Adaptee (ChemicalDatabank) — defines an existing interface that needs adapting.
  - Client (AdapterApp) — collaborates with objects conforming to the Target interface.
- UML



Comentários

- Já tínhamos visto um caso em que o Java usa um Decorator, na leitura e escrita de ficheiros/streams.
- Gosto do exemplo da formatação de textos para o Decorator.
- Usar o Decorator tem várias vantagens, pois não obriga a ter muitas subclasses para ter combinações de formatações. Por essa razão, também evita a duplicação de código. Adicionalmente, podemos mudar as formatações dinamicamente, coisa que não seria possível com subclasses.
- Para o Adapter, gosto do exemplo da aplicação que usa patos, mas eles acabam e tem que passar a usar perus, mas não podemos alterar nem a aplicação nem os patos.
- A nova classe criada no Adapter implementa a interface da classe que se pretende e que o cliente quer usar.
- Tem que ter um objecto da classe adaptada lá dentro e os métodos fazem a "tradução" entre os métodos da interface pretendida e os da classe efectivamente usada.
- O cliente e a classe adaptada não são alterados, nem precisam de saber da existência do Adapter.

Nestes exercícios

- Criei mais umas variáveis nos textos formatados, para poder mostrar como é que se podem acrescentar e remover Decorators dinamicamente.
- Quando temos `2.3f` significa que é um `float`. Os `float` ocupam menos espaço que os `double`, por isso se não tivermos `f` temos um erro de compilação.
- Se for ao contrário, o compilador não dá erro, mas faz arredondamentos: `double a = 2.3f; System.out.println(a);`, por exemplo, dá: `2.299999952316284`.

13 Aula P13 — Avaliação do projecto

Sumário:

- Teste prático do projecto

Resumo:
Tirado da página da PO

O teste prático é realizado individualmente e avalia a capacidade de efectuar alterações ao código entregue.

Para o processo de avaliação, assume-se que o repositório CVS contém um projecto passível de ser construído, i.e., que não apresenta erros de compilação.

Teste Prático Tipo

As seguintes cinco perguntas constituem um enunciado tipo para o teste prático:

1. Compilar e executar um pequeno programa em Java. Esta pergunta é eliminatória.
2. Altere o comando de visualizar produtos para que indique o número de produtos antes de os listar.
3. Acrescente uma nova opção ao menu de gestão de produtos que permita visualizar apenas o produto com o identificador especificado pelo utilizador.
4. Acrescente um novo tipo de veículo, DVD, caracterizado por um identificador único (cadeia de caracteres), um título (cadeia de caracteres), um género (cadeia de caracteres) e um limite de idade (inteiro).
5. Realize um teste JUnit que verifica que a funcionalidade de registo de produtos funciona de forma correcta.

14 Aula P14 — Diagramas de sequência UML

Sumário:

- Diagramas de sequência em UML

Resumo:
From: `http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/`

UML sequence diagrams are used to show how objects interact in a given situation. An important characteristic of a sequence diagram is that time passes from top to bottom : the interaction starts near the top of the diagram and ends at the bottom (i.e. Lower equals Later). A popular use for them is to document the dynamics in an object-oriented system. For each key collaboration, diagrams are created that show how objects interact in various representative scenarios for that collaboration.

An object should be named only if at least one of the following applies:

- You want to refer to it during the interaction as a message parameter or return value.
- You don't mention its type.
- There are other anonymous objects of the same type and giving them names is the only way to differentiate them.

When a target sends a message to another target, it is shown as an arrow between their lifelines. The arrow originates at the sender and ends at the receiver. Near the arrow, the name and parameters of the message are shown.

The white rectangles on a lifeline are called activations and indicate that an object is responding to a message. It starts when the message is received and ends when the object is done handling the message.

Targets that exist at the start of an interaction are placed at the top of the diagram. Any targets that are created during the interaction are placed further down the diagram, at their time of creation. A target's lifeline extends as long as the target exists. If the target is destroyed during the interaction, the lifeline ends at that point in time with a big cross.

From book: Design Patterns, Gamma, Helm, Johnson, Vlissides

An interaction diagram shows the order in which requests between objects get executed. Figure B.3 is an interaction diagram that shows how a shape gets added to a drawing. Time flows from top to bottom in an interaction diagram. A solid vertical line indicates the lifetime of a particular object. The naming convention for objects is the same as for object diagrams — the class name prefixed by the letter "a"(e.g., aShape). If the object doesn't get instantiated until after the beginning of time as recorded in the diagram, then its vertical line appears dashed until the point of creation. A vertical rectangle shows that an object is active; that is, it is handling a request. The operation can send requests to other objects; these are indicated with a horizontal arrow pointing to the receiving object. The name of the request is shown above the arrow. A request to create an object is shown with a dashed arrowheaded line. A request to the sending object itself points back to the sender.

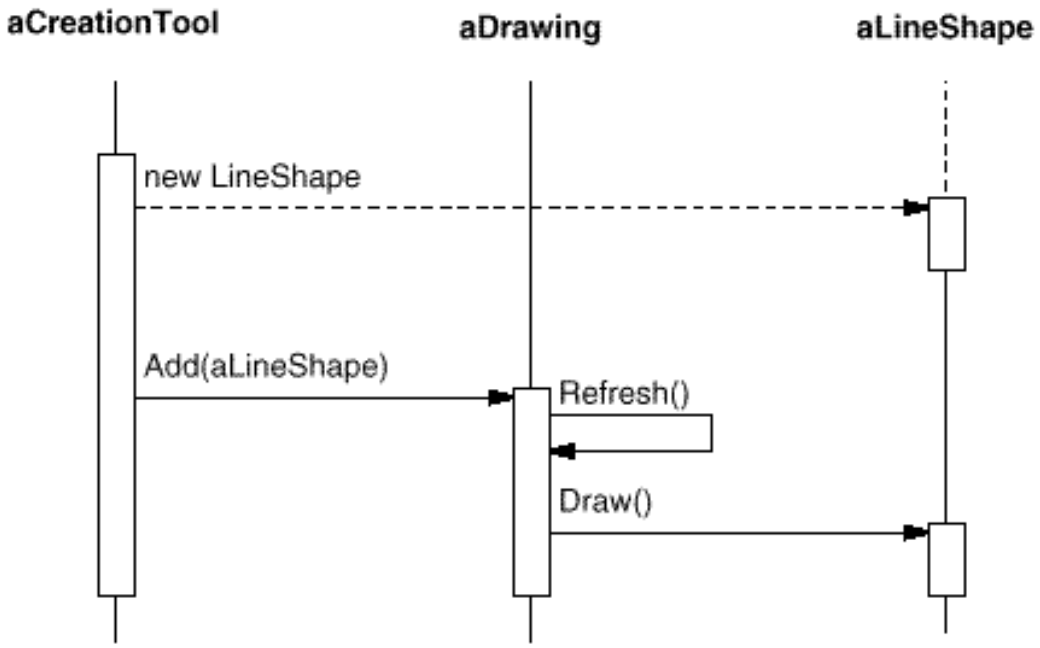


Figure B.3: Interaction diagram notation

Figure B.3 shows that the first request is from `aCreationTool` to create `aLineShape`. Later, `aLineShape` is `Added` to `aDrawing`, which prompts `aDrawing` to send a `Refresh` request to itself. Note that `aDrawing` sends a `Draw` request to `aLineShape` as part of the `Refresh` operation.

UML

- Os objectos são rectângulos com o nome sublinhado, as classes são rectângulos com o nome.