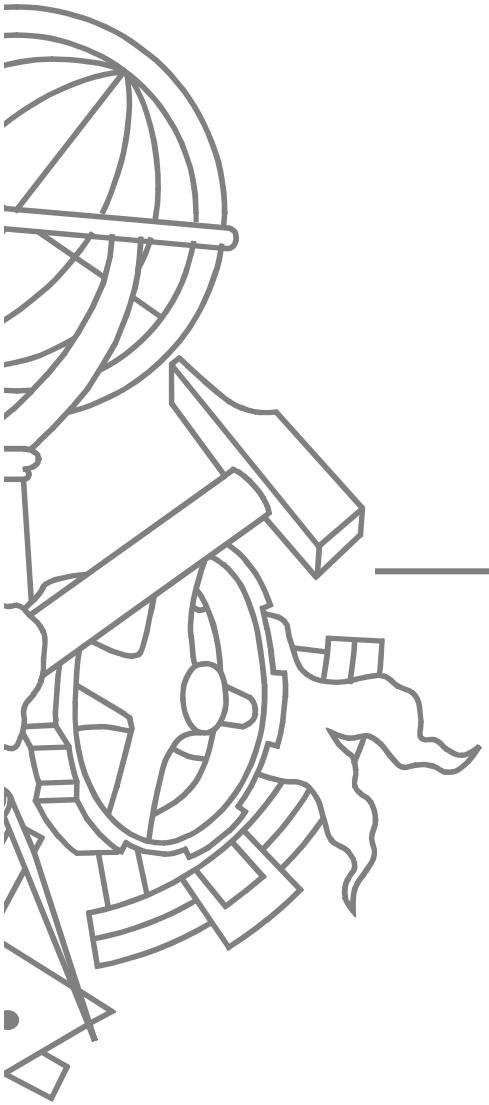
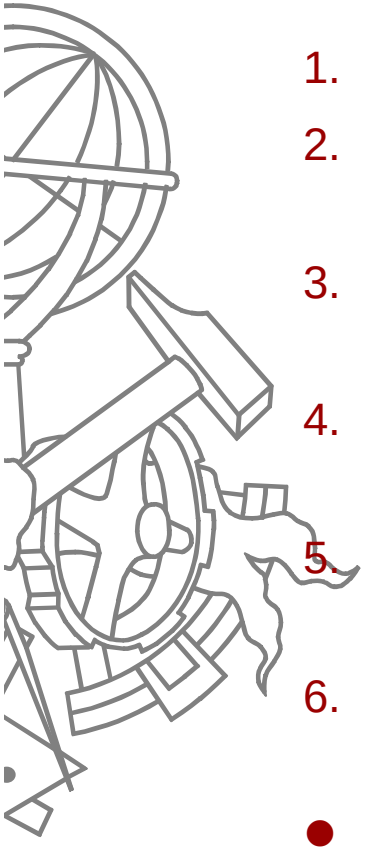


Relembrar os conceitos OO

Parte 2

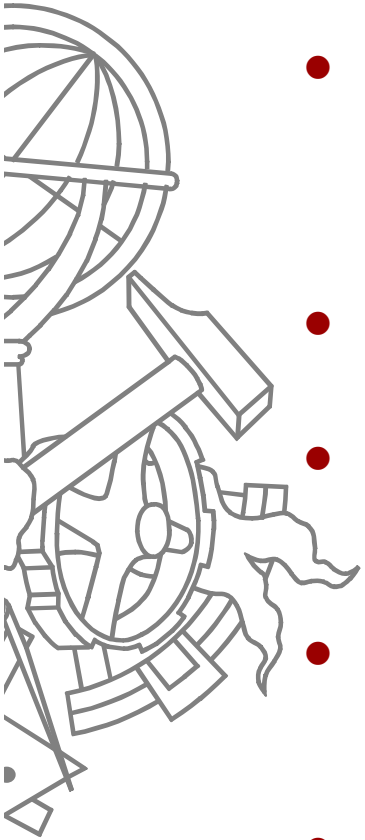
Princípios OOP de Alan Kay



1. Tudo é um objecto.
2. Os objectos efectuam a sua computação trocando pedidos entre si recorrendo a mensagens.
3. Cada objecto tem a sua própria memória que consiste noutros objectos.
4. Cada objecto é uma instância de uma classe. A classe agrupa objectos similares.
5. A classe é um repositório do comportamento associado com um objecto
6. As classes estão organizadas numa árvore (de raíz única) chamada hierarquia de herança.

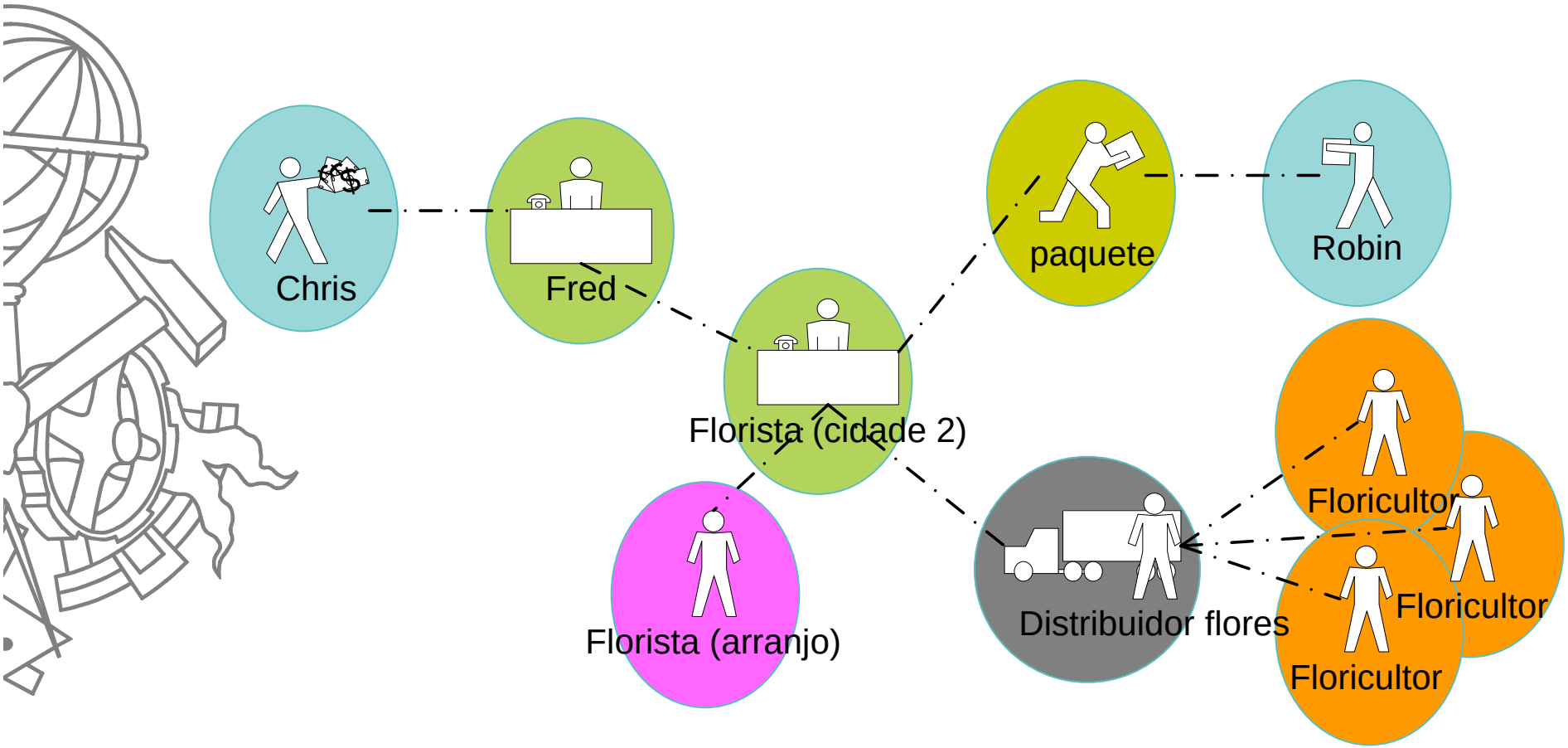
- Programação orientada aos objectos é baseada no princípio do desenho recursivo.

Exemplo

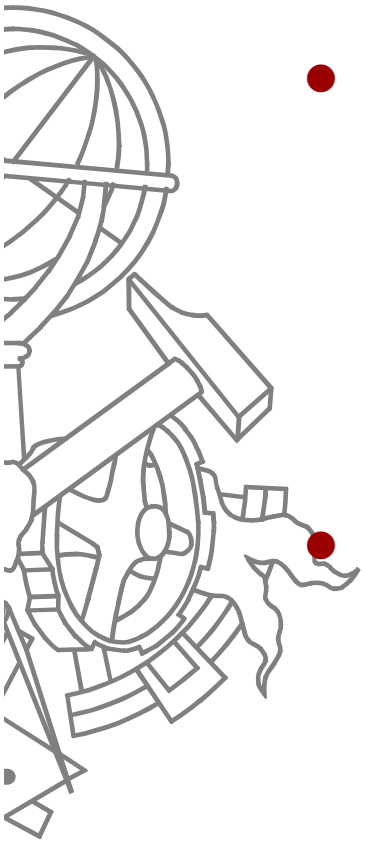


- Para ilustrar os conceitos OO vamos considerar um cenário exemplo de envio de flores para uma pessoa amiga que vive noutra cidade, o Chris vai enviar flores à Robin.
- O Chris não pode entregar pessoalmente as flores, por isso recorre aos serviços da sua loja de flores local.
- O Chris diz ao florista (chamado Fred) a morada da Robin, quanto quer gastar, e o tipo de flores e arranjo que pretende.
- O Fred contacta a florista na cidade onde mora a Robin, que faz o arranjo de flores e em seguida contacta um paquete para a entrega das flores.
- É natural que neste cenário estejam envolvidas ainda outras pessoas: os floricultores, as empresas de distribuição, etc..

Comunidade de objectos no exemplo

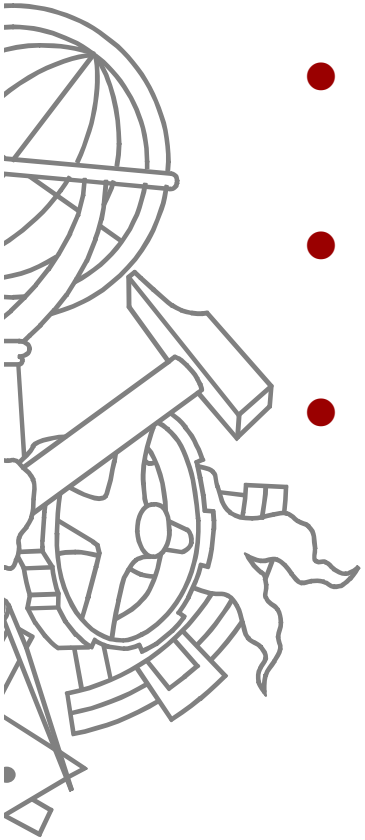


Comunidades de objectos



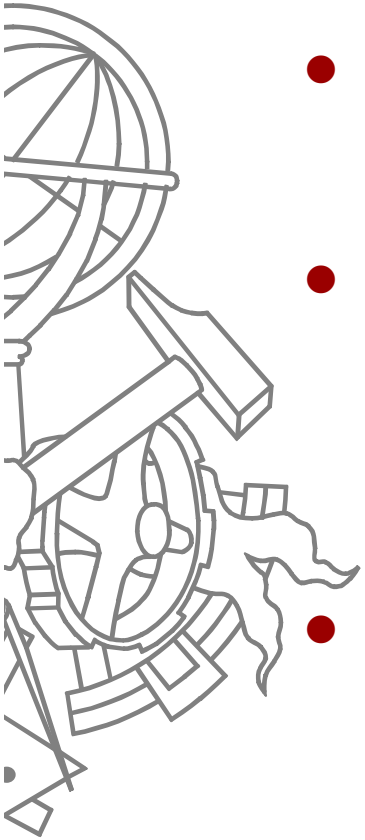
- A primeira observação é que os resultados são obtidos através da **interacção** de diversos agentes, aos quais chamamos objectos. Adicionalmente, qualquer actividade não trivial exige a interacção de uma **comunidade** de objectos trabalhando em conjunto.
- Cada objecto tem um papel a desempenhar, um serviço que pode fornecer aos outros membros da comunidade.

Conceito OO: Objectos



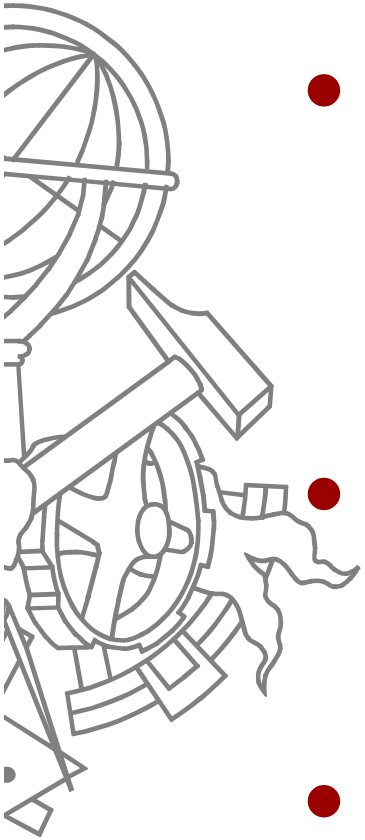
- O primeiro princípio de Alan Kay.
  - Tudo é um objecto.
- As acções em POO são executadas por agentes, denominados *instâncias* ou *objectos*
- Há vários agentes trabalhando cooperativamente neste cenário: o Chris, a Robin, Fred o florista, a florista na cidade da Robin, o paquete, e o floricultor. Cada agente tem um papel a representar e o resultado final é consequência do trabalho de todos para encontrar uma solução para um problema.

Conceito OO: Mensagens



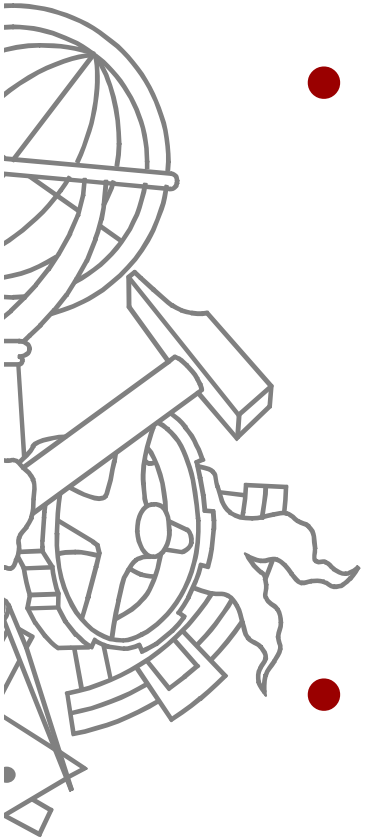
- Princípio número dois:
  - Os objectos efectuam a sua computação trocando pedidos entre si recorrendo a mensagens
- As acções em POO são produzidas em resposta a pedidos, chamados *mensagens*. Uma instância pode aceitar uma mensagem e em seguida executar uma acção e retornar um resultado.
- Para iniciar o processo de envio de flores, o Chris envia uma mensagem ao Fred. O Fred por sua vez envia uma mensagem à florista na cidade da Robin, que envia outra mensagem ao paquete e por aí fora.

Information Hiding



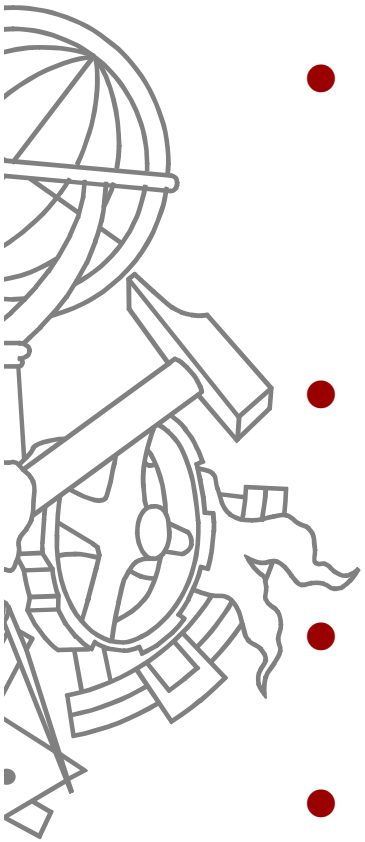
- É de reparar que um utilizador de um serviço fornecido por um objecto, necessita apenas de saber **quais** as mensagens aceites por esse objecto.
- Não necessita de ter qualquer tipo de ideia sobre **como** as acções que serão executadas em resposta a um pedido.
- Após aceitação de uma mensagem, um objecto é **responsável** pela sua execução.

Conceito OO: Receptores



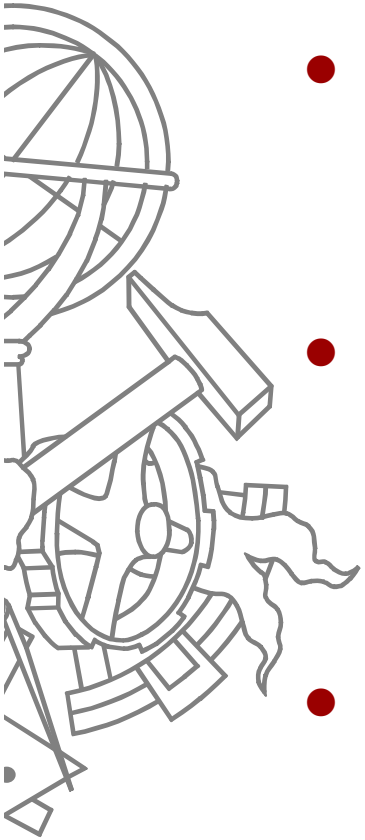
- As mensagens diferem das chamadas de funções tradicionais em dois aspectos muito importantes:
  - Numa mensagem há um receptor designado que aceita a mensagem
  - A interpretação da mensagem pode ser diferente dependendo do receptor
- A mesma mensagem resultará em diferentes acções dependendo de quem a recebe.

Comportamento e interpretação



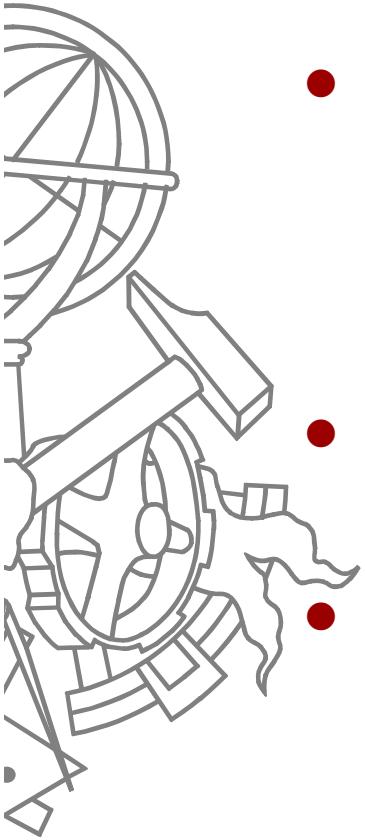
- Apesar de diferentes objectos aceitarem a mesma mensagem, as acções (**comportamento**) que cada objecto desencadeará serão provavelmente diferentes.
- A escolha de qual o comportamento a efectuar pode ser apenas decidido em *run-time* (*late binding*)
- Cada objecto é livre de **interpretar** uma mensagem da maneira que melhor entender.
- O facto de um mesmo nome poder ter significados completamente diferentes é uma forma de *polimorfismo*.

Não interferência (responsabilidade)



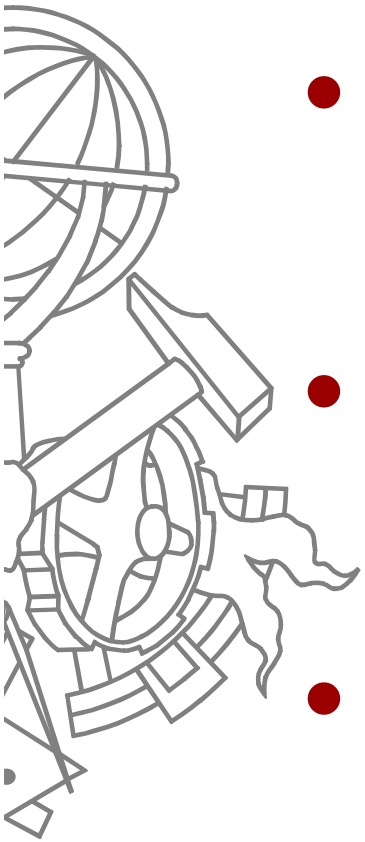
- É importante permitir que os objectos executem as suas tarefas da forma que entenderem adequada sem interacções desnecessárias nem interferências de outros objectos.
- "*Instead of a bit-grinding processor ... plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires*" -- Dan Ingalls.
- "*Ask not what you can do to your data structures, but ask what your data structures can do for you*" -- anónimo

Conceito OO: *overloading*



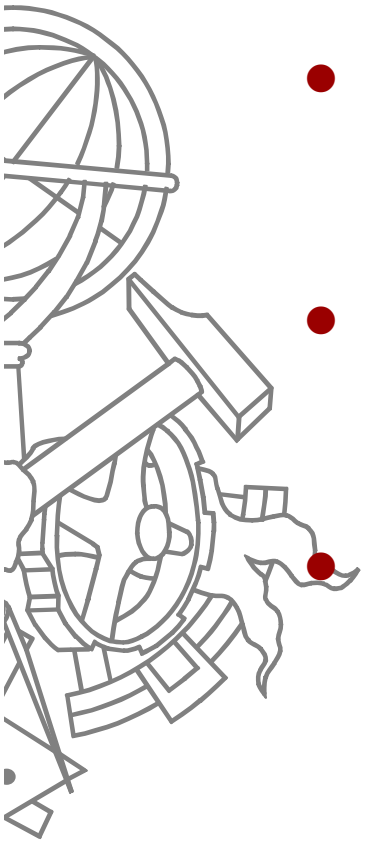
- É um mecanismo que permite a definição de diferentes métodos para uma mesma mensagem num objecto, baseado-se no número e tipo de parâmetros da mensagem
- Tipicamente todas as mensagens estão associadas com o mesmo comportamento
- Exemplo (Java):
  - `Complexo.setValue(float r, float i);`
  - `Complexo.setValue(Complexo o);`

Conceito OO: Desenho recursivo



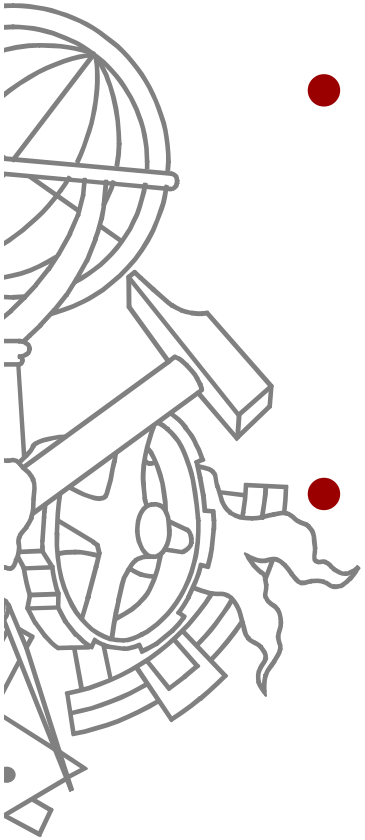
- O terceiro princípio de Alan Kay
  - Cada objecto tem a sua própria memória que consiste noutros objectos.
- Cada objecto é como um computador em miniatura, um processador especializado numa determinada tarefa.
- Composição de um objecto à custa de outros objectos existentes -- reutilização

Conceito OO: Composição (*Has-a*)



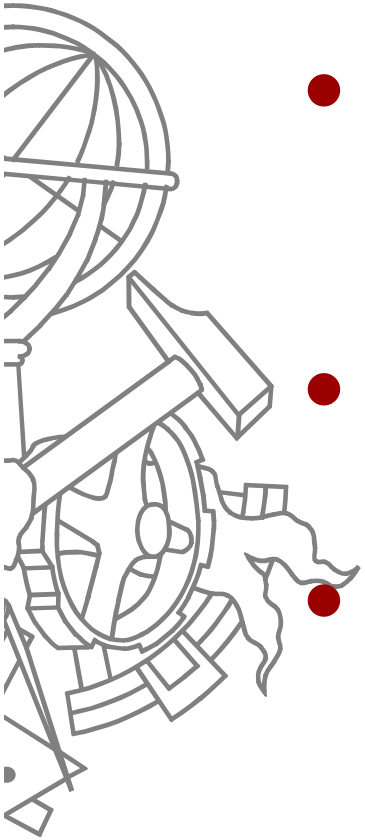
- A divisão em partes particiona um sistema complexo dividindo-o em partes (componentes) que podem ser consideradas isoladamente.
- Caracterizada por frases com o verbo “ter”
  - Um carro *tem* um motor, e *tem* uma transmissão
  - Uma bicicleta *tem* duas rodas
- Permite diminuir a complexidade do sistema ao considerar componentes isolados.

Conceito OO: Encapsulamento



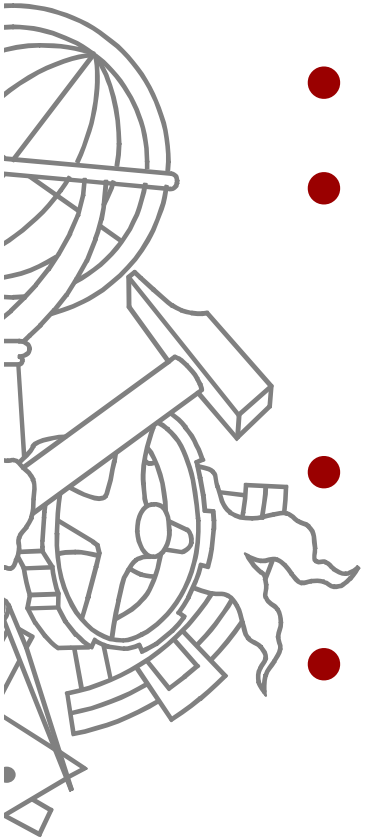
- O encapsulamento consiste na agregação dos dados e seus comportamentos numa única unidade organizacional - **classe**.
- O Encapsulamento permite
  - Reforçar a integridade dos tipos de dados, não permitindo aos programadores o acesso aos campos de dados individuais de um modo inapropriado.

Vantagens do encapsulamento



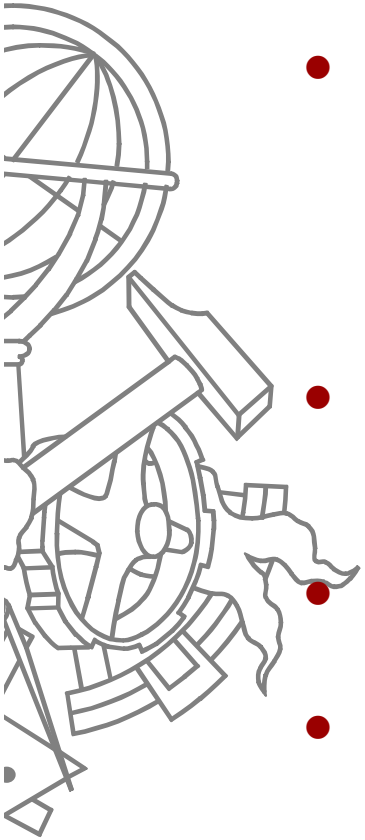
- Modularidade
  - agregando dados e comportamentos numa única unidade.
- Privilégios idênticos aos dos tipos de dados primitivos.
- Protecção dos dados
  - garantindo a sua consistência e segurança.

Vantagens do encapsulamento



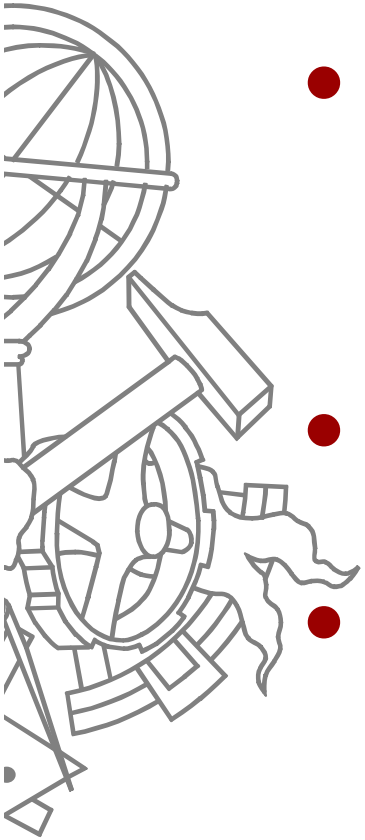
- Separa a implementação da interface.
- Simplifica a *percepção* que os utilizadores têm da classe
  - nível de abstracção.
- Facilita a possibilidade de *modificação da implementação* da classe.
- Independência do contexto
  - promove o desacoplamento.
- Melhor qualidade na produção de software.

Conceito OO: Classes e Instâncias



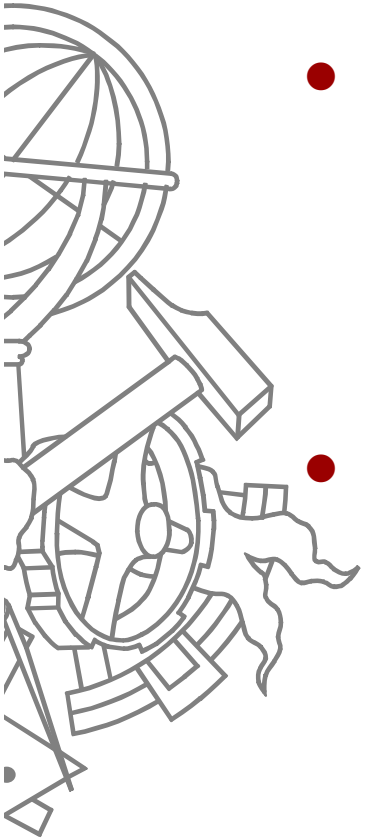
- Princípio n.º 4 e 5
  - Cada objecto é uma instância de uma classe. A classe agrupa objectos similares.
  - A classe é um repositório do comportamento associado com um objecto
- O comportamento expectável do Fred é determinado com base na ideia generalizada que cada um de nós tem dos floristas.
- Diz-se então que o Fred é uma **instância** da **classe** Florista
- O comportamento está associado com as classes e não com os objectos. Todos os objectos que são instâncias de uma classe usam o mesmo método em resposta a uma mesma mensagem (*mas cada um tem a sua própria memória!!! – princípio n.º 3*).

Conceito OO: Herança



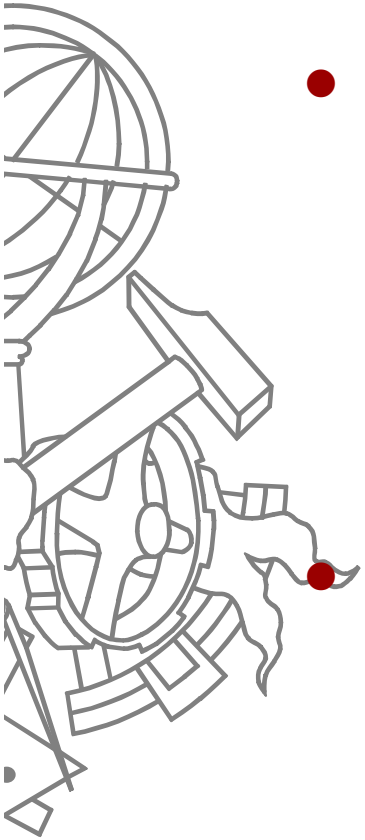
- Princípio n.º 6
  - As classes estão organizadas numa árvore (de raiz única) chamada hierarquia de herança
- Esta hierarquia fornece diferentes níveis de abstracção
- Estados e comportamentos associados a um nível da hierarquia são automaticamente associados aos seus descendentes.

Conceito OO: Herança (*Is-a*)



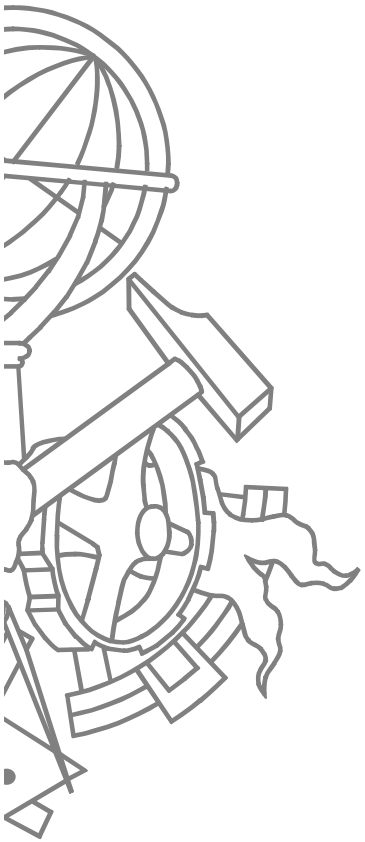
- As abstracções “Is-a” (divisão por especialização) particionam um sistema complexo criando abstracções mais generalizadas/especializadas e relacionando essas abstracções por herança.
- Caracterizada for frases com o verbo “ser”
  - Um carro *é* um veículo que por sua vez *é* um meio de transporte
  - Uma bicicleta *é* um veículo com rodas
  - Uma carruagem a cavalo *é* um meio de transporte
- Permite categorizar artefactos.

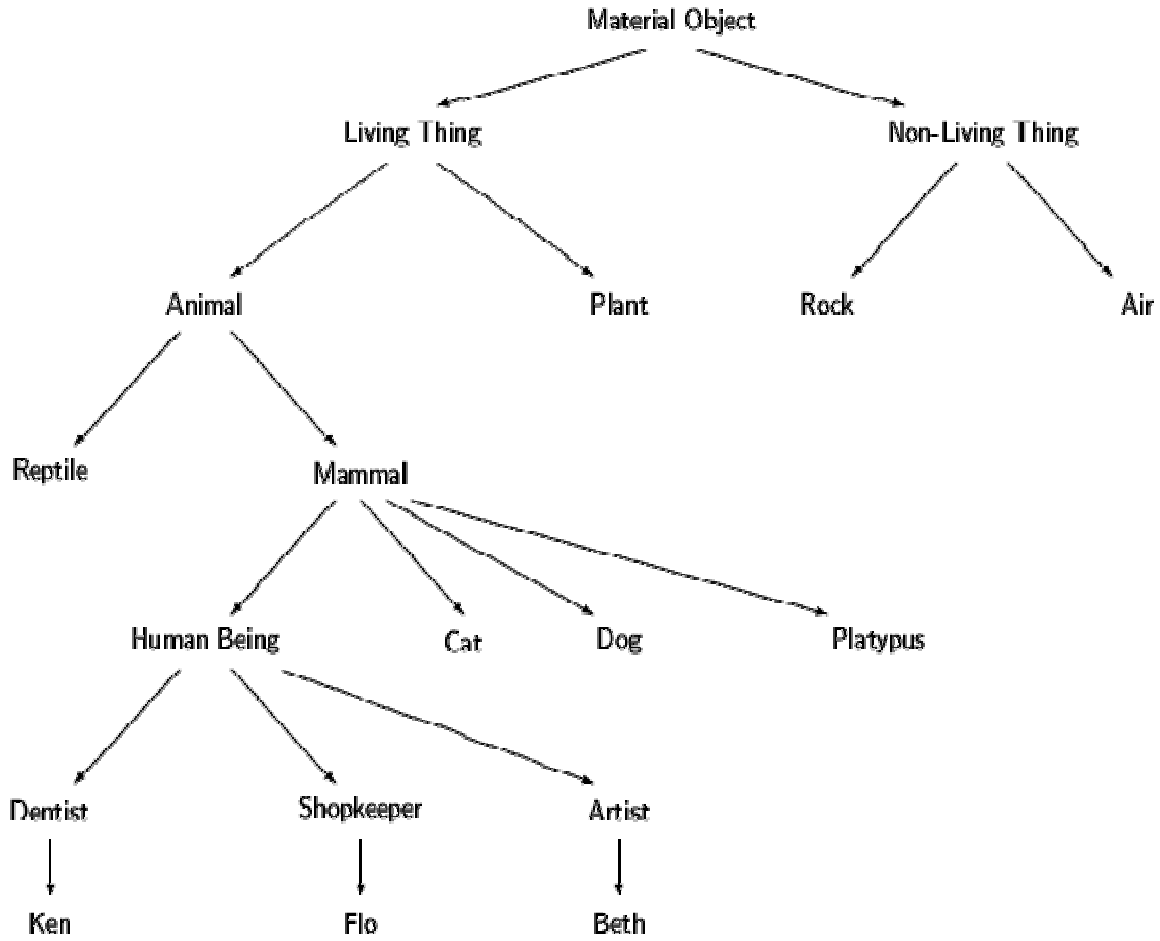
Herança



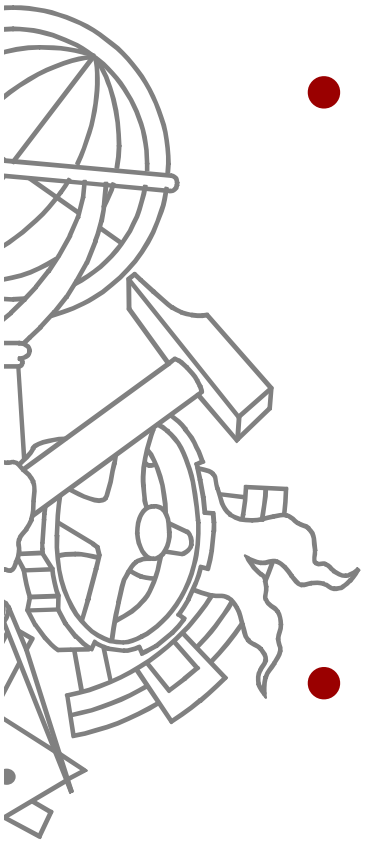
- Herança é uma forma de reutilizar software
  - Novas classes são criadas a partir de classes existentes,
  - Herdando os atributos e comportamentos, e
  - Acrescentando novos atributos e comportamentos que necessitem.
- Herança é utilizada para especialização
  - Todo o objecto da subclasse é também um objecto da superclasse.
  - O inverso não é verdade.

Exemplo de hierarquia de classes



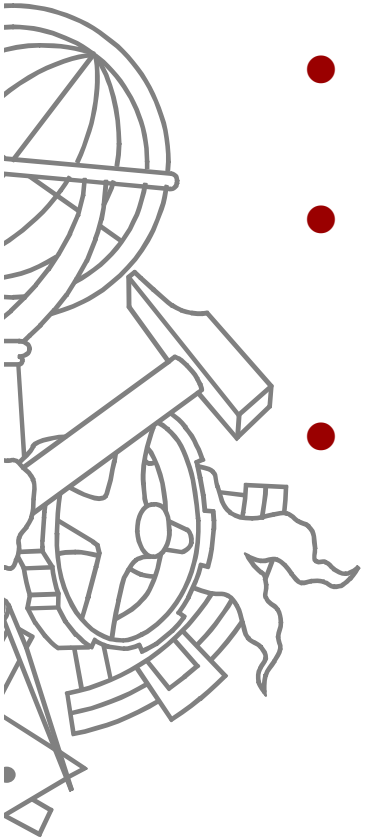


Conceito OO: *overriding* (sobreposição)



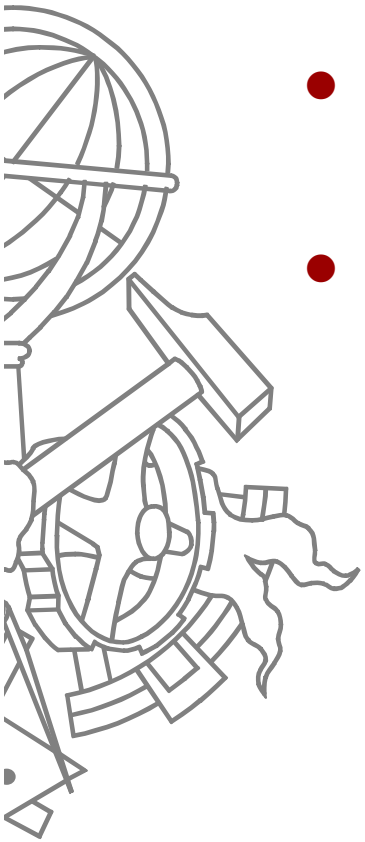
- As Subclasses podem alterar ou sobrepor o comportamento herdado das superclasses:
  - Todos os mamíferos “dão à luz” as suas crias
  - O ornitorrinco é um mamífero que põe ovos
- O mecanismo de herança combinado com a sobreposição é que permitem grande parte do “poder” do conceito OO.

Conceito OO: Polimorfismo



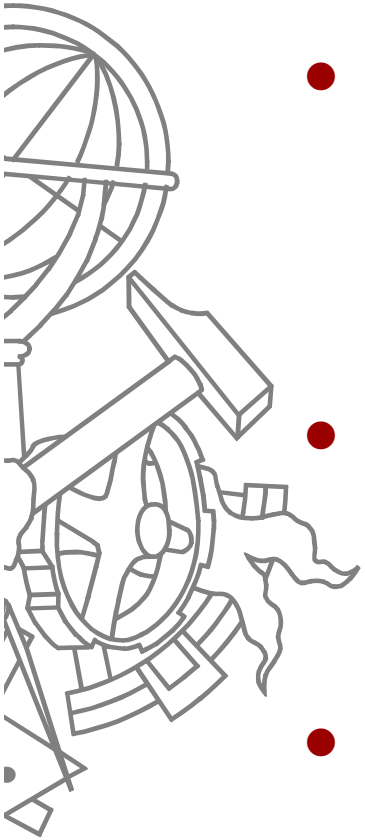
- Mecanismo ligado à herança e à sobreposição de métodos (overriding)
- Consiste na utilização de variáveis de um tipo que na realidade são objectos de outro tipo derivado
- Exemplo:
  - **`Veiculo m;`**
  - Na realidade esta variável pode conter qualquer objecto desde que este seja derivado da classe **`Veiculo`**, por exemplo, **`Carro`** ou **`Bicicleta`**.
  - As mensagens enviadas serão interpretadas pelo objecto receptor e executadas de maneira apropriada. Por exemplo, **`mover`**.

Polimorfismo



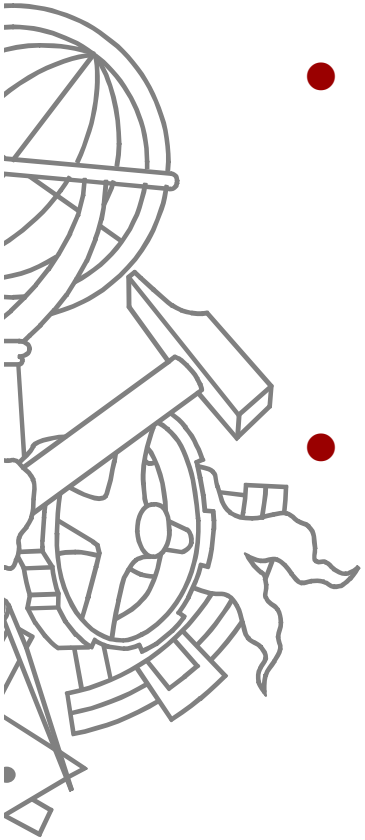
- Significa que através do mesmo nome se pode invocar diferentes métodos.
- O mecanismo de polimorfismo permite que:
  - usando uma referência de uma superclasse para referenciar um objecto (instância de uma subclasse) e
  - invocando um método que exista na superclasse e que também exista (*overriden*) na subclasse do objecto,
  - o programa escolherá dinamicamente (isto é, em tempo de execução) o método correcto da subclasse.

Benefícios do polimorfismo



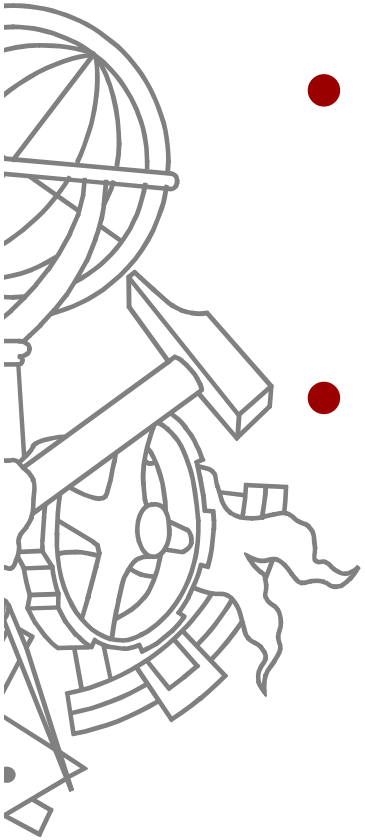
- Tratamento genérico de código
  - numa hierarquia de tipos com o mesmo interface (os mesmos métodos públicos), código que funciona com um tipo genérico também funciona com qualquer objecto que seja subtipo desse tipo.
- Permite uma fácil extensão de um programa com um mínimo de modificações
  - podem-se adicionar novos tipos de objectos que respondem a mensagens existentes.
- Simplifica a escrita de código, a leitura e compreensão e a manutenção - nível de abstracção.

Métodos e classes abstractas



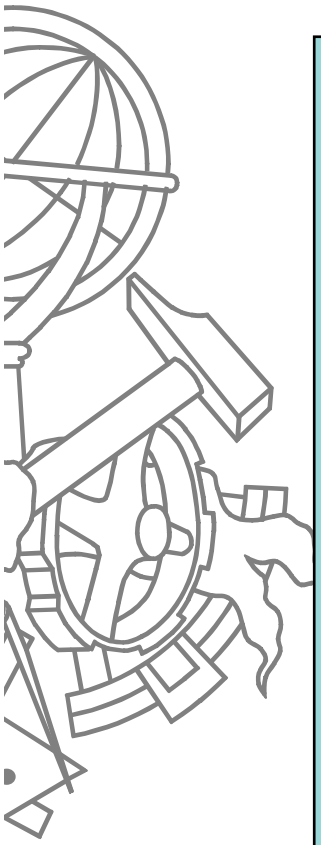
- Diz-se de um método que é definido em termos de “assinatura” mas não é implementado
  - Apenas se define o nome do método, os argumentos e o tipo de retorno sem fornecer a sua implementação
- Uma classe abstracta é aquela que tem pelo menos um método abstracto
  - Não permite a criação de objectos desta classe pois nem todos os comportamentos estão definidos
  - Obriga à implementação dos métodos abstractos em classes derivadas

Membros de classe e de instância



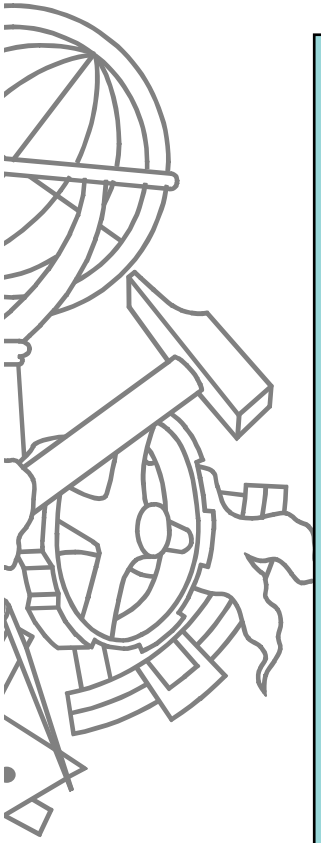
- Uma classe define os atributos e as operações de todas as instâncias dessa classe
- É no entanto possível definir atributos e operações da própria classe
  - Não dependem de nenhuma instância em particular
  - São partilhados por todas as instâncias

Membros de classe e de instância



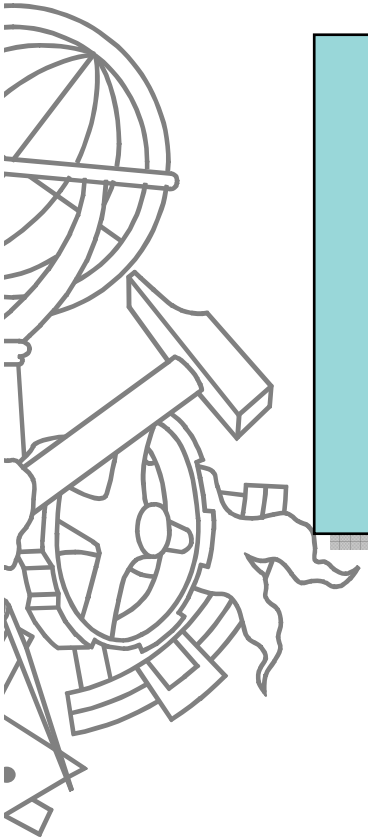
```
class Entidade
{
   static int lastKey = 0;
   string nome;
   int key;
   ...
   private static void NovoCodigo() {
     return ++lastKey;
   }
   public Entidade(string nome) {
     this.nome = nome;
     key = Entidade.NovoCodigo();
   }
   public string GetNome() {
     return nome;
   }
   ...
}
```

Membros de classe e de instância



```
class Pessoa
{
   string nome;
   string morada;
   int id;
   ...
   public static Pessoa Load(int id) {
     // [ código para ler da BD ]
     Pessoa x = new Pessoa(nm, mr);
     x.id = id;
     return x;
   }
   public Pessoa(string nome, string morada) {
     ...
   }
   public string GetNome() {
     return nome;
   }
   ...
}
```

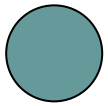
Membros de classe e de instância



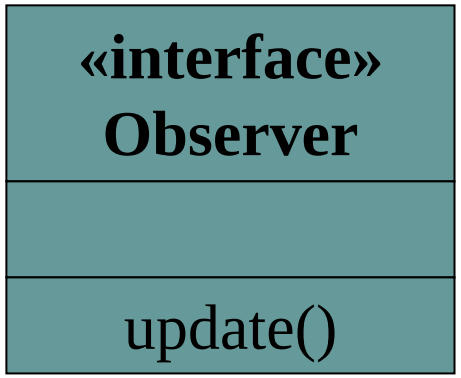
```
// criação de objecto
Pessoa p1 = new Pessoa("António", "r. direita");


// utilização de método de classe para criar objecto
Pessoa p2 = Pessoa.Load(34);
p2.GetNome();
```

Interfaces

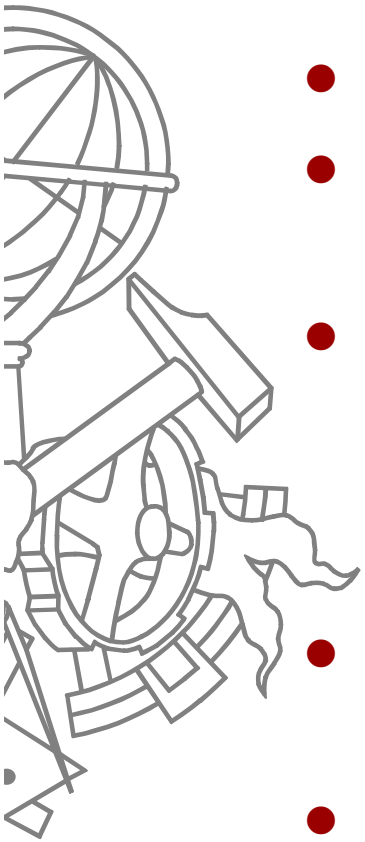
An **interface** is a named collection of operations used to specify a service of a class without dictating its implementation.



Observer

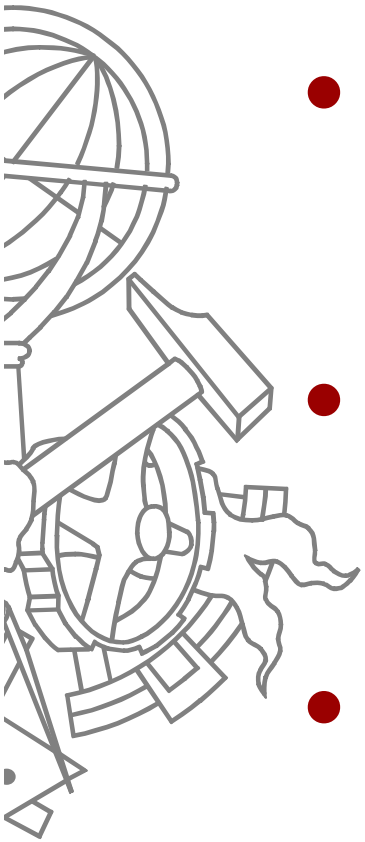


Design by contract



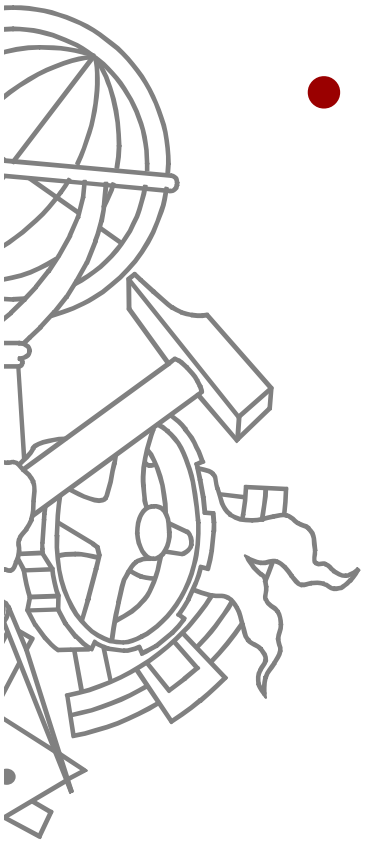
- Visão como serviços
- A interface descreve o serviço fornecido pelo objecto.
- A interface funciona como um contrato para esse serviço: se o serviço é fornecido, então será fornecido de acordo com o descrito no contrato
- Permite a captura de semelhança entre classes sem impor relações artificiais entre elas
- Evidencia o serviço prestado sem forçar qualquer tipo de implementação

Permutabilidade

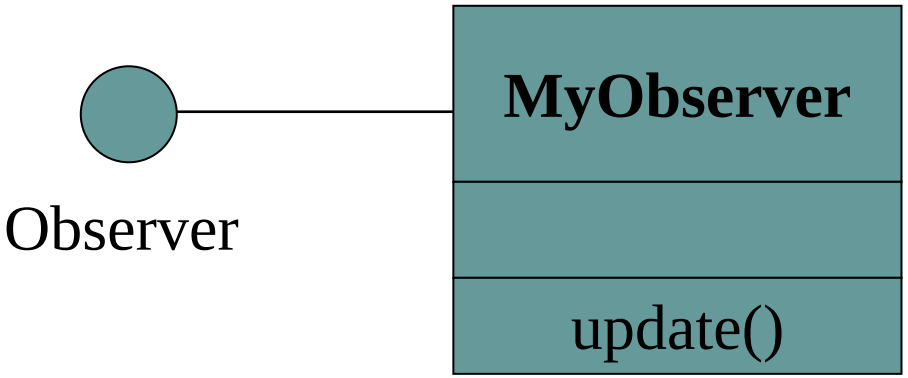


- Um aspecto importante das interfaces é a definição clara da conexão, *interface*, entre componentes.
- Permite considerar várias implementações para uma mesma interface.
- Por exemplo, um carro pode ter diferentes tipos de motor e de transmissão.

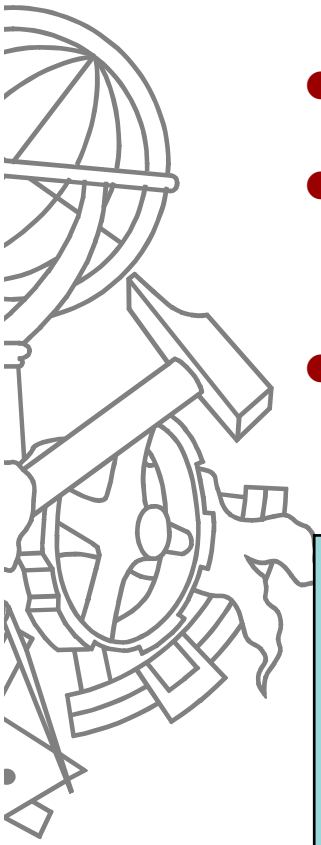
Implementação de Interfaces



- As classes podem “realizar” uma ou mais interfaces



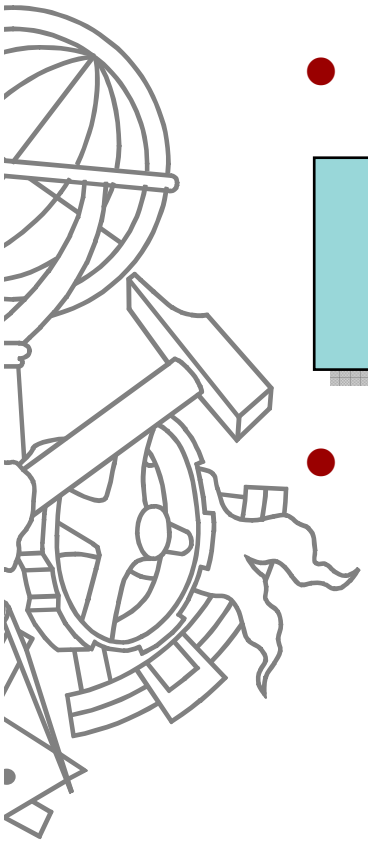
Interface



- Uma interface define um contrato entre partes.
- Apenas define as operações e não a implementação
- Deve ser **imutável**, i.e., as interfaces não podem ser modificadas após "publicação"

```
public interface IMovimentavel {
    public void Mover();
    public void MoverPara(Posicao p);
}
```

Interface

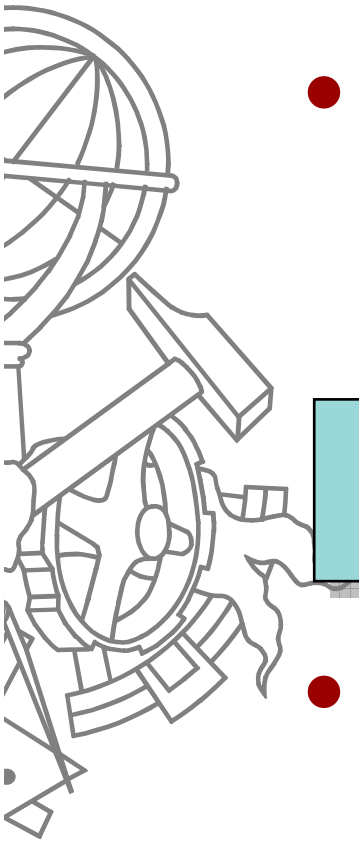


- Pode ser feita herança entre interfaces

```
public interface IMovimentavel2 : IMovimentavel {
    public void Percorrer(Posicao[] p);
}
```

- Esta é a maneira correcta de actualizar (*upgrade*) uma interface
  - A interface `IMovimentavel` já tinha sido publicada e como tal é imutável
  - Novas funcionalidades devem ser colocadas noutra interface derivada
  - Permite o funcionamento correcto de “clientes” da versão anterior

Variáveis do tipo Interface

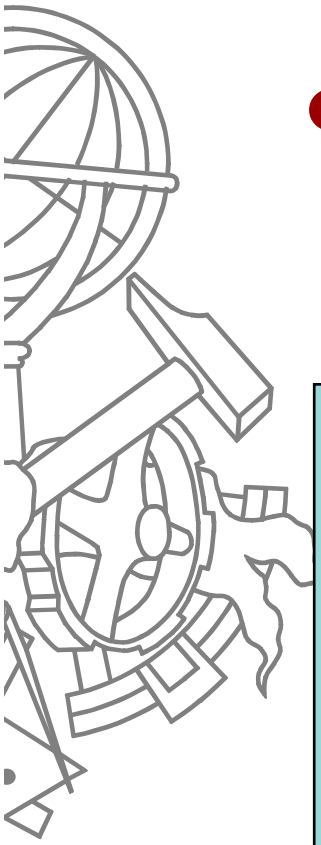


- Funciona como um tipo de dados embora não seja possível criar objectos desse tipo

```
IMovimentavel m; //declara variavel
IMovimentavel x = new IMovimentavel (); // ERRO!!!
```

- A interface não define a implementação logo não se podem criar objectos desse tipo

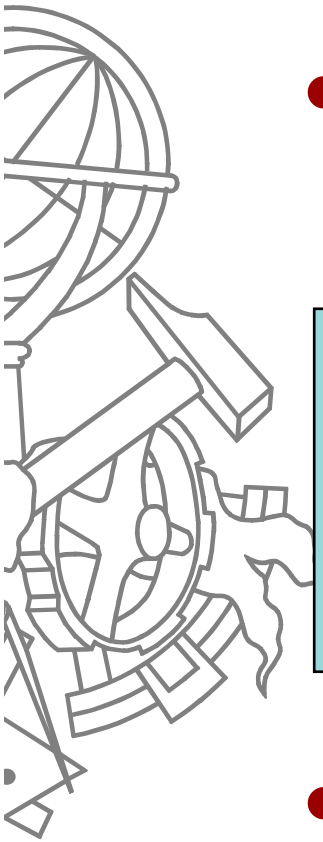
Interface



- Uma classe pode implementar várias interfaces

```
public class Pessoa : ICloneable, IMovimentavel2
{
        ...
        public object Clone() { ... }
        public void Mover() { ... }
        public void MoverPara(Posicao p) { ... }
        public void Percorrer(Posicao[] p) { ... }
}
```

Interface

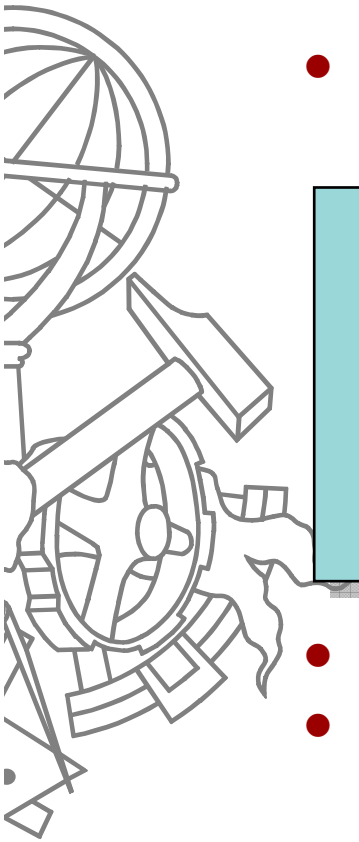


- Várias classes podem implementar a mesma interface

```
public class Carro : Veiculo, IMovimentavel {
        ...
        public void mover() { ... }
        public void moverPara(Posicao p) { ... }
}
```

- Permite relacionar objectos de hierarquias de herança não relacionadas

Interface

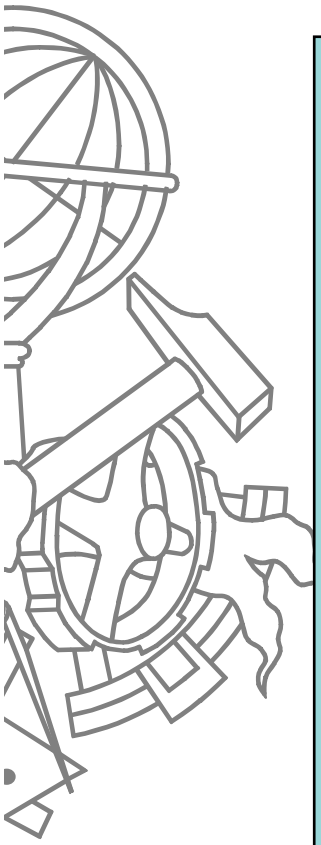


- Uma variável do tipo interface pode referenciar qualquer objecto que implementa essa interface

```
IMovimentavel m;
Pessoa p = new Pessoa();
Carro c = new Carro();
m = p;
m.mover(); //Pessoa.mover()
m = c;
m.mover(); //Carro.mover();
```

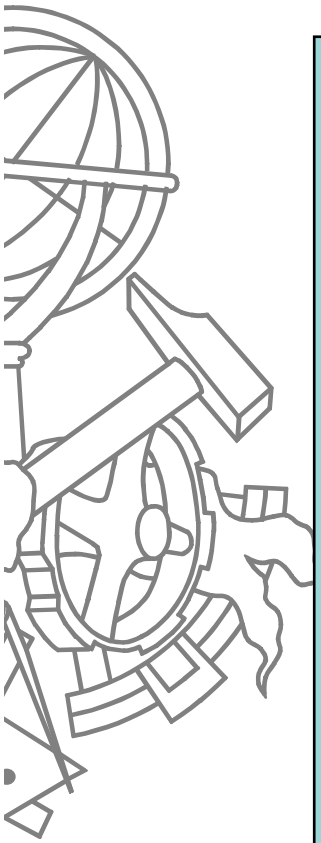
- Muito útil para criar código genérico (framework)
- Apenas necessita que os objectos implementem as interfaces

Exemplo de utilização



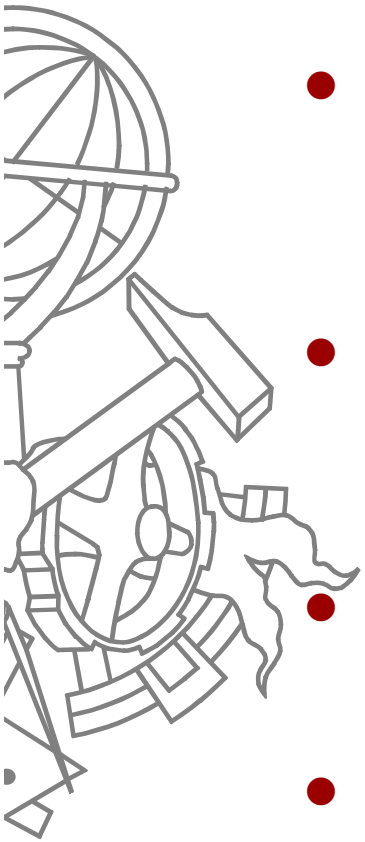
```
public class Cidade {
        protected IList moviles = new ArrayList();
        ...
        public void AddObjMovel(IMovimentavel o) {
                moviles.Add(o);
        }
        public void Movimento() {
                IEnumerator i = moviles.GetEnumerator();
                while (i.MoveNext()) {
                        IMovimentavel m =
                                (IMovimentavel)i.Current;
                        m.mover();
                }
        }
}
```

Exemplo de utilização



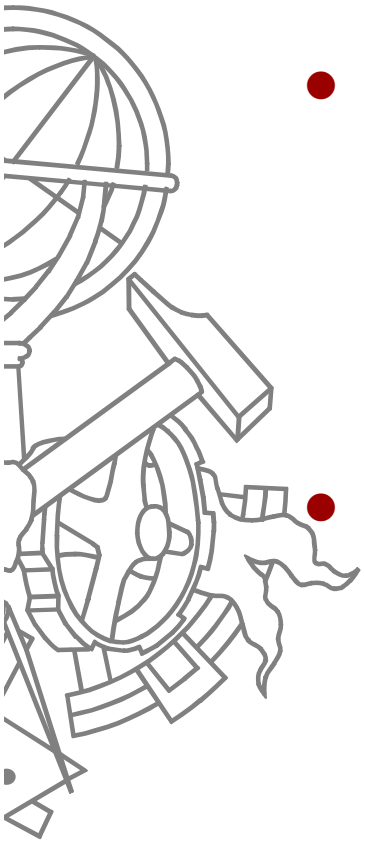
```
public class TestCidadeApp
{
        public static void Main()
        {
                Cidade c = new Cidade();

                c.addMovel(new Pessoa());
                c.addMovel(new Carro());
                c.addMovel(new Carro());

                c.movimento();
        }
}
```

Rules of thumb



- Programar para uma interface e não para uma implementação
  - Diminuir o acoplamento/dependências entre classes
- Favorecer a composição em vez da herança
  - Herança apenas no conceito de especialização conceptual e não funcional
- Separação de responsabilidades
  - “cada macaco no seu galho”
- Separação do código de criação de instâncias

Rules of thumb



- Diminuir as dependências entre objectos/componentes ao nível dos métodos
  - Evitar ligações estruturais entre objectos a não ser quando necessário
  - Favorecer a criação de variáveis locais
- Optimizar apenas no final
  - Não sacrificar a leitura e estrutura do código para optimizações que na prática não fazem diferença
  - Usar um *profiler* para identificar onde vale a pena optimizar