Lusíada
Repositório das Universidades Lusíada

Universidades Lusíada

Silveira, Paulo Enes da

Fundamentos da POO (programação orientada por objectos) : programar computadores com objectos

http://hdl.handle.net/11067/1720

Metadados

Data de Publicação 2015

Resumo A Programação Orientada por Objectos, apesar de ter surgido na década de 60, só começou a ser mais reconhecida e utilizada a partir dos anos 80, tendo o conceito de Objecto ultrapassado a programação, passando pelas bases de dados e chegando até aos modelos de concepção de aplicações informáticas. Ganhou reconhecimento na comunidade científica e é hoje largamente utilizada na indústria de desenvolvimento de software. O presente artigo pretende explicar este paradigma de programação, de uma forma...

Palavras Chave Programação orientada por objectos (Informática)

Tipo article

Revisão de Pares Não

Coleções [ULL-FCEE] LEE, n. 19 (2015)

Esta página foi gerada automaticamente em 2024-04-26T05:58:02Z com informação proveniente do Repositório

FUNDAMENTOS DA POO
PROGRAMAÇÃO ORIENTADA POR OBJECTOS.
Programar Computadores com Objectos

Paulo Enes da Silveira
Universidade Lusíada de Lisboa
Universidade Autónoma de Lisboa
paulosilveira@edu.ulusiada.pt

Resumo: A Programação Orientada por Objectos, apesar de ter surgido na década de 60, só começou a ser mais reconhecida e utilizada a partir dos anos 80, tendo o conceito de Objecto ultrapassado a programação, passando pelas bases de dados e chegando até aos modelos de concepção de aplicações informáticas. Ganhou reconhecimento na comunidade científica e é hoje largamente utilizada na indústria de desenvolvimento de *software*. O presente artigo pretende explicar este paradigma de programação, de uma forma simples e clara, abordando as suas origens e evolução, expondo as suas principais componentes e características que permitem compreender como programar computadores com Objectos.

Palavras-Chave: Programação Orientada por Objectos; Objecto; Classe; Mensagem; Método; Encapsulamento; Herança; Polimorfismo.

Abstract: The Object-Oriented Programming (OOP), despite having appeared in the sixties, only began to be more consistently recognized and used from the eighties on, when the concept of Object exceeded the programming, covering databases and going up to the design of software for computer applications. It then gained recognition in the scientific community and is now widely used in the software development industry. This paper aims to explain the OOP paradigm, as simply and clearly as possible, addressing its origins and evolution, setting out its main components and characteristics which allow us in understanding how to program computers with Objects.

Key-words: Object-Oriented Programming; Object; Class; Message; Method; Encapsulation; Inheritance; Polymorphism.

1. Introdução

Nos últimos trinta e cinco anos a Programação Orientada por Objectos (POO) teve uma enorme expansão, fundada na inovação dos seus princípios, componentes e características, cujos benefícios para a programação dos computadores têm sido reconhecidos pela comunidade científica, pelos programadores e pela indústria.

A utilidade da POO acabou por ser reconhecida pela grande maioria das linguagens de programação actuais, que a integram, possibilitando a programação com Objectos.

No seu artigo “Good Ideas, through the Looking Glass” [1], Niklaus Wirth, com a sua imensa experiência, aborda várias notas sobre a programação, e afirma sobre a POO: “Este paradigma reflecte de perto a estrutura dos sistemas no mundo real, e é então adequado, de modo a modelizar sistemas complexos com comportamentos complexos”. Tendo presente a complexidade dos sistemas a levar em linha de conta e os problemas a solucionar, esta será, certamente, uma das razões do seu sucesso.

Os conceitos Orientados por Objectos, emergentes da Programação, chegaram às bases de dados pela necessidade de se tornar persistente o Objecto (consultar “Objecto persistente” em 4.1). Estão também presentes nos modelos de análise de sistemas, contribuindo para representá-los, dando suporte à programação necessária dos problemas neles identificados.

Neste contexto, o objectivo das presentes notas é o de desmistificar este conceito Orientado por Objectos na Programação, seja aos olhos dos estudantes, seja aos dos que desejam compreendê-lo, mesmo sem serem especialistas de programação, explicando-o simples e claramente.

Nas suas experiências de ensino da POO [2], Zhu e Zhou referem que primeiro se deve abordar a metodologia e só depois a linguagem. Após cerca de trinta anos leccionando POO, não poderia estar mais de acordo com essa afirmação. Para melhor se entenderem os Objectos em Programação, devemos antes compreender bem o seu conceito e a forma de os usar ao representar sistemas e resolver problemas, para somente depois se começar a programar.

Assim, na secção 2 começaremos por indicar as origens, a evolução e impacto da POO. Na secção 3 apontaremos os benefícios do modelo OO. Os componentes e as características deste modelo serão apresentados nas secções 4 e 5. Terminaremos estas notas com as necessárias conclusões.

2. POO – Programação Orientada por Objectos: origens, evolução e impacto

A Programação Orientada por Objectos surge no ano 1961/2, na Noruega, no Norwegian Computing Center (NCC), com o contributo de Ole-Johan Dahl e Kristen Nygaard, criando Simula I (1962-65) e Simula'67 (1967) [3], as primeiras linguagens de programação com conceitos Orientados por Objectos. Simula I, apresentada como linguagem para a simulação de eventos discretos, aparece como uma extensão de Algol'60, uma linguagem de grande notoriedade na época, com a necessidade de programar a simulação de processos. Simula 67 introduz, segundo as palavras de Nygaard e Dahl, os componentes-chave da POO: Objectos, Classes e Subclasses com herança.

É curioso observar-se que, segundo Holmevik no seu excelente estudo sobre a história do Simula [4], quando esta linguagem foi apresentada no NCC, a direcção deste centro que, apesar de moralmente suportar o seu desenvolvimento, não lhe conferiu importância, tendo considerado que não seria uma linguagem que "tivesse futuro", que existia falta de competência para completar o seu desenvolvimento e que a Noruega seria um país demasiadamente pequeno para suportar tal tarefa. Mas Ole-Johan Dahl e Kristen Nygaard não desistiram e - ironia do destino - passados quarenta anos receberam ambos as maiores distinções, a nível mundial, pelos seus contributos à Programação Orientada por Objectos: ACM Turing Award'2001 e IEEE's von Neumann Medal'2002.

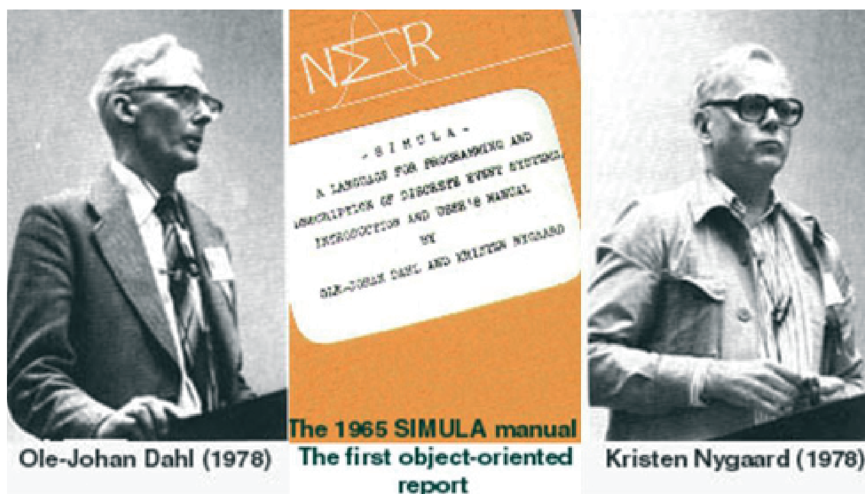

Fig. 1 Ole-Johan Dahl e Kristen Nygaard (1978) e a capa do seu manual de introdução e de utilizador de Simula em 1965, *in* [3]

Tive a oportunidade e o prazer pessoal de privar com Ole-Johan Dahl, em diversas ocasiões, aquando das suas vindas a Portugal para leccionar em seminários, nos quais participei durante os anos 90. Pude constatar a sua alta

competência nos domínios da matemática e da programação, em particular nos compiladores, matérias que tão bem exerceu na concepção do Simula. Deu-nos a conhecer os seus trabalhos de então, que partilhava com Olaf Owe no projecto ABEL (Abstraction Building, Experimental Language) [6]: conceber uma linguagem de programação que pudesse verificar a semântica do código que fosse escrito nessa linguagem. Uma competência curiosa que também evidenciava era a de ser pianista amador, apaixonado por música de câmara. Detinha o sonho de ser quem tivesse o maior reportório executado daquele género musical, com piano.

O Simula I começou a ser utilizado para simulação e ensino, tendo a sua primeira implementação em computadores da UNIVAC e o Simula 67, a partir de 1969, além da UNIVAC também teve implementação através da Control Data, da IBM e da DEC 10, permitindo a sua distribuição e o seu sucesso, como testemunha Dahl [7]. A partir de então, assiste-se a um "impacto cultural" dos conceitos apresentados no Simula 67, na programação e nas linguagens de programação, que se passaram a designar como *Object-Oriented (OO)*.

A INTEL utilizou o Simula 67, no desenho de circuitos *VLSI (Very Large Scale Integration)* e muitas foram as linguagens de programação que incorporaram ou se inspiraram nos Objectos.

Alan Kay, no *Learning Research Group* do *Xerox's Palo Alto Research Center* criou a linguagem Smalltalk'71 [8] e futuras versões Smalltalk'76 e Smalltalk'80, com uma equipa, da qual se destacou o contributo de Dan Ingalls, que desenvolveu muitas arquitecturas de várias gerações de Smalltalk, com a base dos conceitos OO, Objecto, Classe e Subclasses com herança. Criaram as janelas gráficas como Objectos, ensaiaram o dispositivo rato, ambos mais tarde utilizados pelos sistemas operativos da Macintosh e do Windows, tendo também cuidado da interface gráfica, de música e de pintura.

Eiffel, criada por Bertrand Meyer em 1985 [9], também tem as suas raízes no Simula. Introduziu a programação por contractos, com os conceitos de pré-condição e pós-condição onde um bloco de código é executado, antes ou depois de outro, e ainda o conceito de invariante, como um bloco de código que numa classe se executa sempre que o código expresso nessa classe seja chamado a executar-se, permitindo uma programação mais robusta e verificável. Meyer abordou também os aspectos da concorrência em POO [15].

Refiro, agora, algumas das principais linguagens de programação que integraram conceitos OO, sejam as oriundas de outros paradigmas da programação - CLOS (Common Lisp Object System), C++, ObjectPascal/, ObjectOberon, VisualBasic, Ada 95, Fortran 90 e ObjectCobol, PHP, VisualProlog, Perl - sejam outras linguagens designadas de POO "puras", pelos seus componentes serem todos Objectos, como as referidas Smalltalk e Eiffel¸ sejam ainda outras que não são "puras" por terem alguns tipos que não são Objectos, mas que são consideradas OO, como Java e C#. Outras ainda integram os paradigmas imperativo e funcional, para além do OO, Phyton e Ruby.

Com o desenvolvimento das linguagens OO nos anos 80, começaram a aparecer as Bases de Dados OO, que permitem tornar persistentes os Objectos. Alguns exemplos são: GemStone, Objectivity/BD, Versant, ObjectStore, Matisse e Db4o.

Consequentemente, com a aceitação e utilização da POO, apareceram ferramentas para desenho e desenvolvimento de sistemas informáticos, tais como o UML (Unified Modeling Language).

A distribuição de Objectos por vários sistemas numa rede de computadores e a necessidade de comunicação entre os mesmos, tem padrão reconhecido: o CORBA (Common Object Request Broker Architecture).

O COM (*Component Object Model*) da Microsoft permite utilizar código de várias linguagens de POO, permitindo aos vários componentes a comunicação entre si, por exemplo, na plataforma .NET.

Alguns outros paradigmas da programação assentam na base da POO. É o caso da Programação por Padrões (*Patterns*) [10] que permite a utilização de padrões já existentes para o desenvolvimento de *software*, optimizando a sua produção, ou ainda da Programação Orientada por Aspectos (*Aspect-Oriented Programming*) [11] na programação de grandes aplicações, ao separar no conceito de Aspecto, código não funcional, anteriormente espalhado de forma repetida ao longo da aplicação funcional (relativa ao domínio de aplicação), a fim de, em uma só unidade - esse Aspecto - se optimizar a correcção e a manutenção do referido código.

Andrew Black apresenta-nos uma curiosa especulação sobre o futuro dos Objectos [5], em temas como "*multicore and manycore*", "*mobile computing*" e "*cloud*", entre outros.

Perante este impacto que a POO obteve, tendo começado com uma linguagem dedicada à simulação e acabando por estar presente na grande maioria das linguagens hoje utilizadas, na programação distribuída, nas bases de dados, na concepção e desenho de soluções informáticas e em desenvolvimentos futuros da programação, é importante que se conheçam e compreendam bem as características e os componentes principais deste paradigma OO.

3. Benefícios do modelo OO

3.1 Correspondência próxima ao domínio dos problemas

À medida que os problemas a tratar com a programação de computadores foram sendo mais complexos, tendo em conta a informação a considerar nos seus domínios, números e caracteres já não se revelavam suficientes, com os seus tipos numéricos e alfanuméricos, representados em estruturas de dados, desde o género escalar até aos dados compostos. Durante cerca de vinte anos, as linguagens de programação não dispunham de mais de uma dezena de tipos possíveis para

representar a informação consubstanciada em dados de um problema: os de género escalar, como os numéricos inteiros, os numéricos que simulam os reais, os Booleanos; as cadeias de caracteres (*strings*); os de género composto, como as tabelas *(arrays)* - vectores ou matrizes - os registos ou estruturas *(records ou structures)* descritas pelos seus campos, e os ficheiros *(files)*.

Com o advento da POO, o objecto do mundo real ou imaginário passa a ser representado pelo conceito de Objecto da POO, sendo os Objectos de igual descrição e comportamento agrupados em Classes da POO, as quais podem ser criadas sem limite, com a designação que se entender necessária. Nasce, assim, a possibilidade do programador poder criar os tipos que entender para melhor representar as partes do problema, bastando criar as correspondentes Classes: o nome de cada Classe passa a designar um novo Tipo.

Os Objectos do programa correspondem, consequentemente, aos do problema a tratar com a sua descrição (atributos) e com as suas actividades (comportamento).

3.2 A reutilização do *software*

A POO introduziu a possibilidade de reutilizar código já escrito, o que constitui uma vantagem, pois evita-se a reescrita do mesmo e minimizam-se eventuais erros daí resultantes.

Esta reutilização é conseguida através de uma ou mais Subclasses poderem herdar código da sua Classe-mãe ou, em certos casos, de várias Classes-mãe, apenas por declaração desta condição de Subclasse.

Habitualmente, as linguagens de POO dispõem, intrinsecamente, de muito código, centenas ou milhares de módulos de código, distribuídos pelas Classes do sistema das linguagens. Um programador, além de utilizar as Classes do sistema das linguagens, também implementa as Classes que completam a solução do seu problema. Em ambos os casos, as Classes já existentes constituem uma mais-valia, à disposição do programador que, através da herança, poderá reutilizar *software* nessas Classes contido.

3.3 Abstracção e Encapsulamento

Segundo Ole-Johan Dahl [7], foi Tony Hoare quem formalizou matematicamente a visão do Objecto como uma abstracção, a qual veio a ser implementada no Simula 67.

Com efeito, um Objecto encapsulando as suas propriedades ou atributos descritivos juntamente com módulos de código representando o seu comportamento, tem uma formulação abstracta que se pode aplicar universalmente, sendo um modelo de representação dos objectos do mundo real ou imaginário que se deseje considerar em POO.

Barbara Liskov também utilizou o mecanismo da abstração na linguagem

CLU [12], ao definir tipos abstractos de dados: "um dado abstracto consiste num conjunto de objectos e um conjunto de operações caracterizando o comportamento dos objectos".

A ideia de encapsulamento está implícita no facto do Objecto poder ser visto como uma cápsula contendo atributos descritivos e operações de comportamento permitidas sobre o próprio Objecto.

Até ao aparecimento deste conceito de Objecto abstracto na programação, os dados e as operações estavam dispersos, separados. Com a abstracção e o encapsulamento do Objecto, surge o benefício de se reunir a descrição e o comportamento de uma coisa em uma só unidade de programação – o Objecto da POO.

4. Componentes do modelo POO

4.1 Objecto

Um Objecto, em POO, é uma entidade abstracta que engloba Atributos descritivos e operações de comportamento, as quais também se designam como Métodos. Consubstancia-se como instância de uma Classe.

Coisas do mundo real, tais como uma factura, uma janela, um automóvel, um condutor, uma máquina fotográfica ou um gato, podem ser criados como Objectos de uma linguagem de programação num programa que tenha de os considerar. Será possível descrever os seus atributos, como o montante total da factura, a matrícula da viatura, o tipo de combustível do automóvel, a idade do condutor, a marca da máquina fotográfica ou a raça do gato. Mas também será conveniente indicar métodos de comportamento, como sejam, modificar a factura, abrir a janela, acelerar o automóvel, alterar a idade do condutor, focar a máquina fotográfica, ou ouvir a "fala" do gato.

A fim de melhor se compreender um Objecto na POO, comparemo-lo com um Dado (Tipo de Dado) que o MI - modelo imperativo (que engloba linguagens como C, Fortran e Ada), utiliza para representar as coisas.

O Dado (MI):

- é passivo, por si só;
- é activado por agentes exteriores, por exemplo, uma função cuja acção pode resultar no cálculo ou alteração de um seu valor;
- tem um Tipo.

O Objecto (POO):

- é activo, ou mesmo reactivo, por reagir às mensagens que lhe enviam;
- recebe mensagens que originam acções, tais como mudar o todo ou parte do seu estado, ou seja, do seu conjunto de atributos;

- pertence a uma Classe (equivalente ao Tipo do MI).

Os Objectos, ao serem criados, passam a ser instâncias de uma Classe, assumindo os Atributos e Métodos que são comuns aos Objectos dessa mesma Classe (exemplos em 4.4: Smalltalk - instâncias da classe do sistema "*Point*" (5); C# - instâncias da nova Classe "Ponto" (6)).

O ciclo de vida de um Objecto é geralmente limitado ao tempo de vida do programa em que é instanciado, se não for, entretanto, destruído através de uma instrução expressa no programa ou do processo de *Garbage Collection* (quando existente no compilador da linguagem) que gere a memória do computador, destruindo Objectos que já não estejam em uso. Mas se desejamos guardar o Objecto para além da vida do programa, promove-se a condição deste a Objecto persistente - de existência permanente. A persistência do Objecto passa por se respeitar a sua integridade transacional, mantendo intacta a sua estrutura interna de Atributos com valores e as suas relações com outros Objectos, seja através de "serialização" no arquivo em ficheiro (automática na imagem do compilador Smalltalk, ou serializável em Java ou C#), ou seja por se integrar em base de dados, por simples formatação de Objecto persistente.

4.2 Classe

A Classe é um módulo onde, em particular, se declaram os Atributos e os Métodos dos potenciais Objectos que se vierem a instanciar:

- Atributos que descrevem um Objecto e que constituem a estrutura do seu estado interno, os quais serão validados aquando da instanciação desse Objecto, de forma semelhante às propriedades de um conjunto que, por sua vez, serão validadas nas propriedades dos seus elementos, quando criados;
- Métodos que um Objecto possa executar, expressando os seus possíveis comportamentos vários. Cada Método é um módulo de código que implementa as acções que descrevem um comportamento do Objecto.

Todos os Objectos que são criados como instâncias de uma Classe têm os mesmos Atributos e os mesmos Métodos.

Uma Classe em POO é um corpo declarativo, onde se indica o que um seu Objecto é efectivamente e o que pode fazer. Uma nova Classe é declarada com um <nome de Classe> que passa a constituir em POO o equivalente a um novo Tipo no modelo imperativo (exemplo em 4.4: C# - Classe "Ponto" (6)).

Ao construir um programa OO o programador pode criar quantas novas Classes necessitar, ou utilizar as Classes-sistema (*built-in*) da linguagem OO, reutilizando o *sotfware* já existente, seja criando Subclasses das Classes-sistema, seja instanciando Objectos destas Classes-sistema que existem em elevado número. Exemplos destas Classes-sistema são a Classe "*Collection*" no Smalltalk,

com mais de trinta Classes e Subclasses (*Bag, MappedCollection, Array, ByteArray, WordArray, SortedCollection, LinkedList, String, HashedCollection, Dictionary, Set, …*), ou ainda as mais de cinquenta Classes incluídas em vários *namespaces* "*Collections*" do C# (*ArrayList, BitArray, Hashtable, Queue, List, SortedList, Stack, Dictionary, SortedDictionary, HashSet, SortedSet, …*).

4.3 Mensagem

Poderia dizer-se que um Programa Orientado por Objectos é constituído por Objectos que enviam Mensagens a outros Objectos. Simples mas eficaz. Os Objectos comunicam entre si através de Mensagens.

Um Objecto receptor de uma Mensagem faz executar o Método do mesmo nome desta, desde que este exista na Classe a que o Objecto pertence.

A Mensagem é a invocação do Método do mesmo nome, com a sintaxe:

<Objecto receptor> <Mensagem> <Objecto emissor>

a qual é representada graficamente na Figura 2.

Fig. 2 Representação gráfica da Mensagem

Existem Mensagens em que o Objecto emissor está ausente. Vejamos um exemplo deste caso em Smalltalk e o equivalente em C#.

Smalltalk:

```
4 factorial.
```
(1)

Sendo Smalltalk uma linguagem OO pura, os números inteiros, em particular, são Objectos. Por isso, na instrução (1), o Objecto 4 recebe a Mensagem factorial, a qual invoca o Método *factorial* que está presente na Classe *Integer* a que o 4 pertence, executando-se o respectivo código que calcula o factorial de 4 (devolvendo 24, resultado de 4*3*2*1). Em Smalltalk, esta Mensagem designa-se como unária, por só ter o objecto receptor e o selector ou nome de Mensagem.

C#:

```
factorial (4);
```
(2)

A linguagem C# não é OO pura, sendo os numéricos, como os *Integer*, tipos primitivos. Por este motivo, o cálculo do factorial é feito por invocação da função do mesmo nome (2) com o parâmetro 4, de forma idêntica à do modelo imperativo.

4.4 Método

Um Método é um conjunto de acções que podem ser executadas sobre o Objecto receptor da Mensagem com mesmo nome desse Método. O Objecto receptor é sujeito a esse conjunto de acções - a operação ou Método - como resultado da Mensagem recebida.

Uma correspondência biunívoca entre a Mensagem e o Método ocorre quando uma Mensagem é recebida por um Objecto, pois esta provoca execução de um Método com o mesmo nome.

Podem existir vários Métodos com o mesmo nome, no mesmo programa, com a condição de que, em cada Classe, exista uma e apenas uma vez, ou seja, Métodos com o mesmo nome são permitidos, desde que se situem em Classes diferentes.

Vejamos o que acontece quando uma Mensagem recebida por um Objecto invoca um Método, com o exemplo do factorial em Smalltalk.

```
Smalltalk:
          4 factorial. “correcto”
          ‘4’ factorial. “incorrecto”
```
(3)

Na Figura 3 são apresentadas as representações gráficas de Objecto, Classe e Método necessárias à interpretação destas duas instruções, segundo o modelo HBDS (Hypergraph-Based Data Structure) [14] e modelo UML [13].

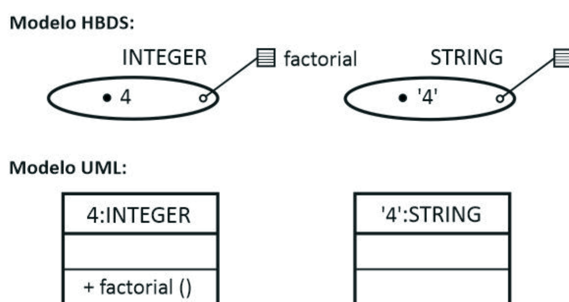

Fig. 3 Representação de Objecto, Classe e Método, em HBSD e UML, para interpretação das instruções (3) em Smalltalk

Na primeira instrução de (3), 4 factorial, 4 recebe a Mensagem factorial. O Smalltalk verifica a que Classe pertence o Objecto 4 e conclui ser à INTEGER;

verifica se nesta Classe existe o Método factorial e encontra-o. Feitas estas verificações, pode então executar o Método factorial, presente na Classe INTEGER, com a seguinte codificação recursiva:

```
Smalltalk:
            factorial
              self = 0 ifTrue: [^ 1].
              self > 0 ifTrue: [^ self * (self - 1) factorial].          (4)
```

Em Smalltalk, a palavra reservada *self* é a representação abstracta do Objecto. Neste caso, a Mensagem 4 factorial. faz executar o Método factorial com o Objecto receptor 4 no lugar de self. O símbolo ^ devolve o valor do Método. Esta implementação segue a definição recursiva não terminal de factorial(n) = n*factorial(n-1), com a condição de paragem n=0 a devolver 1.

Na segunda instrução de (3), '4' factorial, o Objecto '4' recebe a Mensagem factorial. De forma idêntica à descrita para o Objecto inteiro 4 da instrução anterior, o Smalltalk conclui que o Objecto '4' pertence à Classe STRING e que o Método factorial não existe nesta Classe, verificando a incorrecção dessa instrução.

Com a análise destas duas instruções (3), observou-se que um Método descrito numa Classe é invocado através de Mensagens do mesmo nome, enviadas a um Objecto que pertença a essa Classe. Se o Método invocado não se encontrar nessa Classe, não poderá ser executado (exceptuando o caso deste Método poder existir em Classes hierarquicamente superiores a essa Classe, facto que melhor se compreenderá ao estudarmos a característica - Herança).

O Método corresponde ao procedimento ou à função do modelo imperativo.

Observemos ainda a implementação do Método "distância" entre dois pontos, em Smalltalk e C#.

```
Smalltalk:

            "Método distância entre dois pontos"
            Point»distancia: umPonto
                | dx dy |
               dx := umPonto x - x.
               dy :=  umPonto y - y.
               ^ ((dx * dx) + (dy * dy)) sqrt.

            "Mensagem invocando o método distância com mensagem @ que
            instância pontos"
            4@2 distancia: 8@4.                                          (5)
```

Em Smalltalk existe uma Classe Point, já disponível para o programador. Em (5) apenas se apresenta a implementação do Método distancia, com duas

variáveis temporárias dx e dy, as quais recebem as diferenças das coordenadas x de cada um dos dois pontos, e das coordenadas y de cada um dos dois pontos. A instrução 4@2 distancia: 8@4. é uma Mensagem binária em Smalltalk, porque tem o Objecto receptor 4@2, seguido do nome de Mensagem distancia, e ainda do Objecto emissor 8@4. Esta mensagem invoca o método distancia, que foi colocado na classe Point, o qual, por sua vez, é executado devolvendo o valor da distância. O Objecto emissor 8@4 passa a ser umPonto, com os seus atributos, as coordenadas umPonto x com o valor 8 e umPonto y com o valor 4. O Objecto receptor 4@2 passa a ser implicitamente considerado através das suas coordenadas x e y. A distância é então calculada pela raiz quadrada da soma dos quadrados das diferenças das coordenadas x e das coordenadas y dos dois pontos, através das sucessivas mensagens de multiplicação (*) soma (+) e raiz quadrada (sqrt).

C#:

```
class Ponto
{
  double x;
  double y;
  public Ponto(double x, double y)
  {
    this.x = x;
    this.y = y;
  }
  public double Distancia(Ponto umPonto)
  {
    double dx,dy;
    dx := umPonto x – x;
    dy := umPonto y – y;
    return Math.Sqrt((dx * dx) + (dy * dy)) ;
  }
}

Class Programa_Pontos
{
  static void Main( )
  {
    Ponto p1 = new Ponto(4.0, 2.0);
    Ponto p2 = new Ponto (8.0,4.0);
    Console.WriteLine(p1.Distancia(p2));
  }
}
```

(6)

Em C# (6), não existindo a Classe Ponto, esta é criada com: os seus Atributos x e y; o seu construtor, um Método do mesmo da Classe que permite a instanciação dos Objectos (em cujo código, this representa o presente Objecto, tal como self em Smalltalk); o Método Distancia. No Método Main( ), necessário para início do programa, instanciam-se os pontos p1 e p2, afixando o resultado da distância, através da Mensagem p1.Distancia(p2). O mecanismo seguido é semelhante ao descrito para o Smalltalk em (5).

Neste caso, os dados usados pelo Método Distancia foram os do Objecto receptor (o seu estado interno, as coordenadas x e y) da Mensagem, mais os do parâmetro Objecto emissor, que integrava a Mensagem.

5. Características do modelo POO

5.1 Encapsulamento

As noções de Objecto e de Classe emanam uma característica fundamental da POO, o encapsulamento, que vamos analisar em dois aspectos.

1.º aspecto: A cápsula

O Objecto visto como uma cápsula, contém os seus atributos e o acesso às suas operações ou métodos de comportamento.

A Classe oferece um contexto para o encapsulamento, reunindo os Atributos e Métodos dos Objectos que lhe pertencem, proporcionando-lhes uma protecção. Na Figura 4, a Classe CÍRCULO reúne o Atributo raio e o Método área. Ao instanciar-se o Objecto umCírculo, o raio tomou o valor de 2 e a área é implicitamente uma operação possível sobre este Objecto.

Fig. 4 Encapsulamento reunindo Atributos e Métodos

Esta característica - encapsulamento - separa claramente a implementação do uso. Na Figura 5, a implementação do Método desenhar na Classe CÍRCULO, pode ser reescrita enquanto outro programador pode estar a implementar um código para o uso dessa implementação. Com o mesmo código de uso em que o Objecto umCírculo recebe a mensagem desenhar, a consequente invocação do

Método desenhar pode dar origem ao desenho de umCírculo com uma textura lisa, num momento de execução em que o método desenhar contenha instruções para afixar essa textura lisa, ou rugosa, em outro momento de execução em que o Método desenhar contenha instruções para afixar essa textura rugosa, como consequência da reescrita do Método.

Fig. 5 Separação da implementação do uso do código

2.º aspecto: A acessibilidade

O encapsulamento permite "esconder" o estado interno do objecto (Atributos com os seus valores) e os seus Métodos ou operações do "mundo exterior", permitindo um controlo de acessibilidade.

A Classe constitui uma delimitação entre o seu interior (com Objectos descritos por Atributos e Métodos de comportamento) e o exterior. Pode declarar-se uma acessibilidade aos Atributos ou aos Métodos, como sendo públicos (*public*) ou privados (*private*), entre outras variantes, resultando o efeito de se permitir ou não o acesso proveniente de outras Classes. Um Atributo qualificado como privado pode ser consultado ou modificado no seio da Classe em que é declarado, mas não pode sê-lo no âmbito de outra Classe, a menos que este Atributo seja qualificado como público. O mesmo se passa com um Método que, ao ser qualificado como público, pode ser invocado por uma Mensagem do mesmo nome existente no âmbito de uma outra Classe que não declara este Método. Um Método qualificado como privado, apenas pode ser invocado por Mensagens enviadas a Objectos pertencentes à mesma Classe onde este Método é declarado.

Por exemplo, o Smalltalk determinou que todos os Atributos sejam privados e todos os Métodos sejam públicos, assegurando assim a privacidade dos Atributos que só podem ser acedidos via Métodos da mesma Classe. Já o C# entendeu que os Atributos e os Métodos sejam, por defeito, privados, devendo ser expressamente declarados públicos, quando o programador assim o entender.

5.2 Herança

A herança em POO é uma relação hierárquica específica entre Classes, em que a classe "mãe" dá a herdar o conjunto de atributos e de métodos nela declarados, às suas subclasses "filhas".

Teoricamente não há limite nos níveis de hierarquia da herança. Uma Subclasse herda todos os Atributos e Métodos das classes que lhe são hierarquicamente superiores.

O exemplo de herança, na Figura 6, evidencia a Classe MAMÍFERO e as Subclasses CÃO e GATO, mostrando que estas herdam implicitamente o Atributo peso e o Método alteraPeso, os quais passam a coexistir com os locais Atributo raça e Método fala.

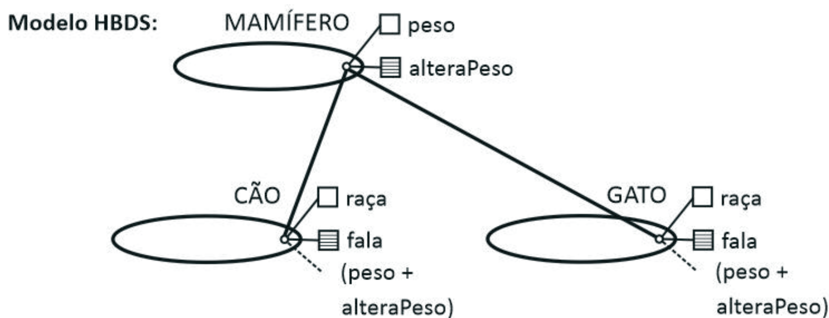

Fig. 6 A herança entre Classes

A Figura 7 ilustra vários níveis de herança, mostrando a acumulação sucessiva dos Atributos e Métodos herdados, da Classe hierarquicamente superior POLÍGONO até ao RECTÂNGULO.

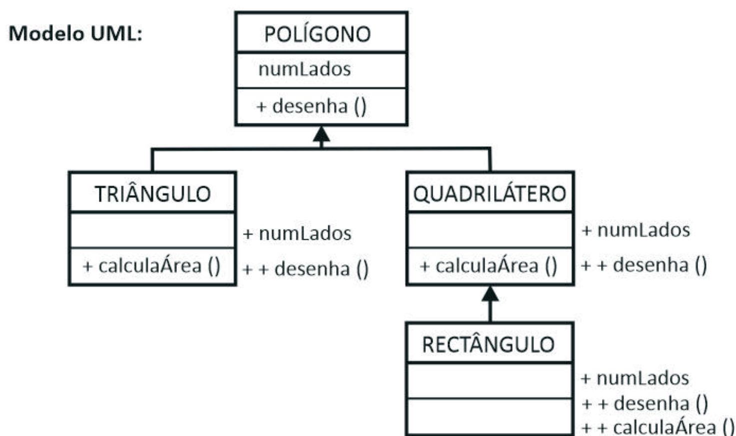

Fig. 7 A herança entre Classes, com mais de um nível de herança

A herança pode ser simples ou múltipla. É simples quando cada subclasse tem uma única classe “mãe”, como são as linguagens Smalltalk, Object Pascal, Modula 3, Java, C# e Ruby. É múltipla quando cada subclasse pode ter mais do que uma classe “mãe”, sendo o caso das linguagens Eiffel, C++, CLOS, Perl e Python.

A herança múltipla é uma característica poderosa, nomeadamente na reutilização de *software*, quando utilizada criteriosamente, mas levanta problemas de semântica e de complexidade.

As Classes são estruturadas hierarquicamente, verificando-se a existência de uma classe “mãe” de todas, embora nem sempre as linguagens de POO tenham esta estrutura. Na raiz da árvore da hierarquia das Classes, a Classe “mãe” de todas é designada por *Object* (nas linguagens Smalltalk, Java e C#) ou por *ANY* (em Eiffel).

Observemos a sintaxe para declaração de herança:

```
Smalltalk:
          <classeMãe> subclass: #<classeFilha>
                ...
C#:
          class  <classeFilha | classeDerivada> : <classeMãe | classeBase>
               { ... }                                                    (7)
```

Na herança, podem escolher-se membros das Classes já existentes, eliminando-os ou modificando-os.

5.3 Polimorfismo

Literalmente, o polimorfismo significa “muitas formas”, ou seja, a capacidade da mesma coisa apresentar diferentes formas.

Em POO o polimorfismo é a capacidade do mesmo nome de Mensagem poder ser interpretado de formas diversas, por diferentes Objectos provocando diferentes invocações de Métodos de diferentes Classes. É o Objecto receptor da Mensagem, com a sua relação de pertença à Classe, que associa o Método que deve ser invocado.

Esta capacidade só é possível pela estrutura das Classes, pela relação de pertença dos Objectos às suas Classes e pela diferença entre Mensagem e Método. Quando se observa uma Mensagem enviada a um Objecto, não é possível conhecer o seu efeito, enquanto não se souber o código do Método que é invocado.

Na Figura 8 e código (8) apresenta-se um exemplo de polimorfismo. As Classes CÃO e GATO apresentam Métodos fala, cada um com a respectiva fala. A fala do CÃO ladra e a fala do GATO mia. Quando se criam dois Objectos, o cão piloto e o gato tareco, e se lhes envia a mesma mensagem fala, a Mensagem fala enviada ao piloto que pertence à Classe CÃO invoca o Método fala com o código 1 que ladra. Por outro lado, a Mensagem fala enviada ao tareco que pertence à Classe GATO invoca o Método fala com o código 2 que mia.

Fig. 8 Evidência do Polimorfismo

Smalltalk:

```
fala ...<código 1 ladra> “na classe CÃO”
fala ...<código 2 mia> “na classe GATO”
“Uso:”
piloto fala. “piloto ladra”
tareco fala. “tareco mia”
```

C#:

```
public fala() {<código 1 ladra>} //na classe CÃO
public fala() {<código 2 mia>}  //na classe GATO
//Uso:
piloto.fala(); // piloto ladra
tareco.fala(); // tareco mia
```

(8)

Assim, observámos que para estarmos na presença de um polimorfismo, há que haver pelo menos duas Mensagens do mesmo nome enviadas a Objectos diferentes que pertençam a Classes diferentes, nelas contendo Métodos com o mesmo nome das Mensagens. Deste modo duas Mensagens do mesmo nome invocam Métodos com diferentes efeitos.

O polimorfismo permite que em várias Classes possam existir Métodos com o mesmo nome, tendo estes, em princípio, códigos diferentes.

6. Conclusões

Não estaria inicialmente nos objectivos destas notas sobre os fundamentos da Programação Orientada por Objectos, proporcionar ao leitor os conhecimentos necessários para começar a programar neste paradigma. Ou estaria? Na

realidade, entender os seus fundamentos é compreender a sua filosofia, conhecer as suas raízes, evolução e impacto, os seus princípios, os seus componentes e características. Eis a razão pela qual abordámos estas notas.

Depois de observarmos a evolução e impacto da POO desde a sua criação nos anos 60, salientámos os seus benefícios: a proximidade dos problemas a resolver, a reutilização do *software*, a abstracção e o encapsulamento. Expusemos os componentes fundamentais da POO: o Objecto, a Classe, a Mensagem e o Método, exemplificando alguns casos com código nas linguagens Smalltalk e C#. As características do paradigma foram abordadas de forma a compreender-se como devem relacionar-se os componentes: o encapsulamento, a herança e o polimorfismo.

Existem na POO outros conceitos que não foram discutidos nestas notas. No entanto, o conhecimento dos fundamentos expostos, permitirão com facilidade a sua compreensão.

Conhecer os fundamentos da POO é uma base essencial para se entenderem os vários domínios de aplicação Orientados por Objectos, em particular, como programar computadores com Objectos.

Agradecimento

O autor agradece ao Professor Rui Agonia Pereira as prolíferas discussões que com ele manteve sobre a Programação Orientada por Objectos, seja na fase de *working paper* do presente artigo, seja em outros momentos da vida académica e científica.

Referências

1. Niklaus Wirth. 2006. Good Ideas, through the Looking Glass. *IEEE Computer* 39, 1 January 2006, 28-39.
2. Haibin Zhu, MengChu Zhou. 2003. Methodology first and language second: a way to teach object-oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA ‘03). ACM, New York, NY, USA, 140-147.
3. Ole-Johan Dahl, Kristen Nygaard (sem data). How Object-Oriented Programming Started. Dept. of Informatics, University of Oslo. Os autores descrevem estas notas como: “an abbreviated version of an article requested by an encyclopedia some years ago”. [Consultado a 2015-05-27 https://web.archive.org/web/20021210082312/http://www.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html]
4. Jan Rune Holmevik. 1994. Educating the Machine: A Study in the History of Computing and the Construction of the SIMULA Programming Languages.

Trondheim: STS Report 22/94.

5. Andrew P. Black. 2013. Object-oriented programming: Some history, and challenges for the next fifty years. *Inf. Comput.* 231, October 2013, 3-20.
6. Ole-Johan Dahl, Olaf Owe. 1991. Formal Development with ABEL. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2: Tutorials* (VDM '91), Søren Prehn and W. J. Toetenel (Eds.). Springer-Verlag, London, UK, UK, 320-362.
7. Ole-Johan Dahl. 2004. The Birth of Object Orientation: the Simula Languages. *In: From Object- Orientation to Formal Methods: Essays in memory of Ole-Johan Dahl*. O. Owe, S. Krogdahl, T. Lyche (eds.), LNCS 2635, Springer Verlag, 15-25.
8. Alan C. Kay. 1993. The early history of Smalltalk, in: *The second ACM SIGPLAN conference on History of programming languages, HOPL-II*, ACM, New York, NY, USA, 511-598.
9. Bertrand Meyer. 1992. *Eiffel: the Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
10. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
11. Filman, Elrad, Clarke, Aksit, editors. 2004. *Aspect-Oriented Software Development*, Addison-Wesley.
12. Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert. 1977. Abstraction Mechanisms in CLU. *Communications ofthe ACM*, *Volume 20 Issue 8*, 564-576.
13. James Rumbaugh, Ivar Jacobson, Grady Booch. 1998. *The Unified Modeling Language User Guide*. Addison-Wesley.
14. Paulo Enes da Silveira. 2015. A Structural Prototype for Planning and Controlling a Manufacturing System, *New Contributions in Information Systems and Technologies, Advances in Intelligent Systems and Computing, volume 353*, Springer International Publishing, DOI 10.1007/978-3-319-16486-1_8, 79-90.
15. Bertrand Meyer. 1993. Systematic Concurrent Object-Oriented Programming, *Communications of the ACM, 36, 9*, September 1993, 56-80.