Programação Orientada a Objetos

Resumo do Professor
Antonio Carlos Sobieranski

1 – Conceitualização

O conceito de **POO** teve início na linguagem **Simula 67**, que foi a primeira a **implementar** o **conceito** de **objeto**. Diferentemente dos paradigmas orientados a procedimentos, onde o principal foco são os sub-programas e bibliotecas, funções e suas passagens de parâmetros, o principal enfoque da OO são os dados sob a ótica de um objeto. Os conceitos foram aproveitados e refinados na Linguagem **SmallTalk** em **80**, que continua servir como **protótipo** de **modelos** de **OO**. Embora existam outras linguagens, tal como Eiffel e Java, muitos ainda consideram **SmallTalk** como a **única** LP **puramente orientada** a **objetos**. A capacidade de trabalhar com OO é frequentemente adicionada às linguagens **imperativas**, tal como **C++**, ditas **Híbridas**.

No entanto, o paradigma de POO tornou-se popular apenas na década de 80. A **programação** que ajudou no **desenvolvimento** da **POO** é baseada em **dados**, tendo como foco os **tipos abstratos** de **dados**. Neste paradigma, o processamento de um objeto de dado é especificado através da **chamada** dos **sub-programas**, **associados** com o **objeto**. Exemplo:

- a ordenação de um <**objeto**> vetor é definida no TAD (Tipo Abstrato de Dados) para esse vetor, e o processo de ordenação é realizado através da **chamada desta operação**, **contida** no **objeto** vetor específico.
- na programação estruturada a ordenação de um vetor/array <**variável**> de valores inteiros é enviado como parâmetro para um outro sub-programa, que o ordena.

Uma linguagem OO deve fornecer suporte a **4 características chave**: **TAD/abstração**, **encapsulamento**, **herança** e **polimorfismo,** vistos mais adiante. POO tem como **entidade** fundamental o **OBJETO**, que **recebe** e **envia mensagens**, **executa processamento**, e possui um **estado local** que ele pode **modificar**. Problemas são resolvidos através **objetos** que **enviam mensagens uns para os outros**. Assim, pode-se também afirmar que o modelo OO é formado por 4 componentes básicos: **objetos, mensagens, métodos e atributos, e classes**, descritos a seguir.

2 – Componentes da POO

2.1. Objeto

Consiste em um **conjunto** de **operações** encapsuladas (**métodos**) e um **estado** (determinado pelo valor dos **atributos**) que recupera e grava os efeitos destas operações. Em outras palavras: o objeto possui tudo o que é necessário para conhecer a si próprio. O objeto executa uma operação em resposta a uma *msg* recebida, e o resultado da operação depende tanto do conteúdo da mensagem recebida, quanto do estado interno do objeto ao receber a mensagem. Um objeto pode também enviar mensagens para outros objetos, assim como para si mesmo. Pode-se representar que objetos consistem em uma coleção de dados relacionados com um tema em comum. Exemplo de um objeto, é um círculo, que contém como atributos *raio* e posicionamento *x,y,* ou um veículo, que contém como atributos nome, marca, ano, modelo, e como métodos um conjunto de ações: ligar, acelerar, freiar, etc.

2.2. Mensagens

São **requisições enviadas** de um **objeto** para **outro**, envolvendo um **objeto emissor** da **mensagem** e outro **receptor**. Trata-se de um ciclo completo onde uma **msg** é **enviada** a um **objeto**, operações são **executadas** dentro deste **objeto**, e uma mensagem contendo o **resultado** da operação é enviada ao **objeto solicitante.** Ex.: uma fábrica, que recebe uma ordem de produção (mensagem de

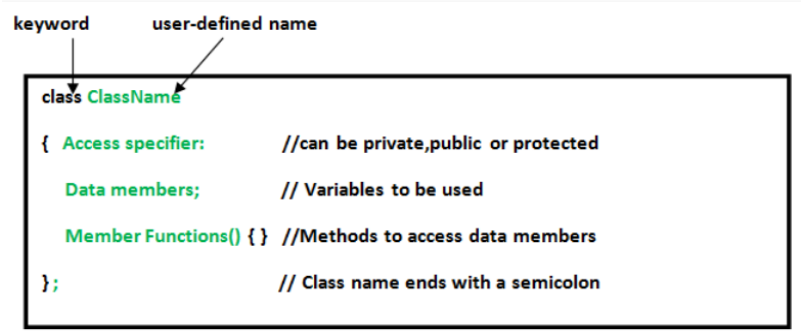
solicitação), processa esta ordem (operações) utilizando matéria-prima (atributos), e gera um produto final (mensagem de resposta).

2.3. Métodos e Atributos

- **Métodos** são similares a **procedimentos** e **funções**. Descrições das operações que um objeto executa quando recebe uma mensagem. **Correspondência 1-para-1 entre mensagens e métodos** que são executados quando a mensagem é recebida através de um dado objeto.
- **Atributos** consistem na **informação** de **estado** ou **dado**, para o qual cada objeto de uma classe tem seu próprio valor. Existem 2 tipos de atributos em **OO**: **atributos de objetos e atributos de classes**. Atributos de objetos descrevem valores (estados) mantidos pelo objeto, e diferentes objetos não compartilham os atributos, ou seja, cada um possui sua própria configuração de atributos. Atributos de classe são aqueles cujos objetos devem compartilhar em comum.
- Exemplo: O círculo pode ter os métodos de computação, que realizam os cálculos necessários para alterar as propriedades do objeto.

2.4. Classes

Classes definem as **características** de uma **coleção** de **objetos**. Consistem em descrições de **métodos** e **atributos** que **objetos** que pertencem à classe irão possuir. Uma classe é similar a um **TAD** no sentido que define a **estrutura interna** e um **conjunto** de **operações** que todos os objetos instâncias dessa classe devem possuir. Desta forma, é **desnecessário redefinir** cada **objeto** que possui as **mesmas propriedades**. Em resumo, uma classe representa uma ideia ou conceito simples categoriza objetos que possuem propriedades similares.



3 – Características da OO

Para entendermos exatamente do que se trata a **orientação** a **objetos**, vamos entender quais são os requerimentos de uma linguagem para ser considerada nesse paradigma. Para isso, a linguagem precisa atender a quatro **conceitos bastante importantes**:

3.1. Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem **Orientada a Objetos**. Como **estamos lidando** com uma **representação** de um **objeto real** (o que dá nome ao paradigma), temos que **imaginar o que** esse **objeto** irá **realizar** dentro de **nosso sistema**. São três pontos que devem ser levados em consideração nessa abstração: **Identidade, propriedades e métodos.**

- **Identidade:** O primeiro ponto é darmos uma ***identidade*** ao objeto que iremos criar. Essa identidade deve ser única dentro do sistema para que não haja conflito. Na maior parte das linguagens, há o conceito de pacotes (*ou namespaces*). Nessas linguagens, a identidade do objeto não pode ser repetida dentro do pacote, e não necessariamente no sistema inteiro.
- **Propriedades:** A segunda parte diz respeito a **características do objeto**. Como sabemos, no mundo real qualquer objeto **possui elementos que o definem**. Dentro da programação

orientada a objetos, essas características são nomeadas ***propriedades***. Por exemplo, as propriedades de um objeto "Cachorro" poderiam ser "Tamanho", "Raça" e "Idade".

- **Métodos:** Por fim, a terceira parte é definirmos as ações que o objeto irá executar. Essas **ações**, ou **eventos**, são chamados ***métodos***. Esses métodos podem ser extremamente variáveis, desde "Acender()" em um objeto lâmpada até "Latir()" em um objeto cachorro.

3.2. Encapsulamento

O *encapsulamento* é uma das **principais** técnicas que define a **programação orientada** a **objetos**. Se trata de um dos elementos que **adicionam segurança** à aplicação em uma programação orientada a objetos pelo fato de **esconder** as **propriedades**, criando uma **espécie** de **caixa preta**.

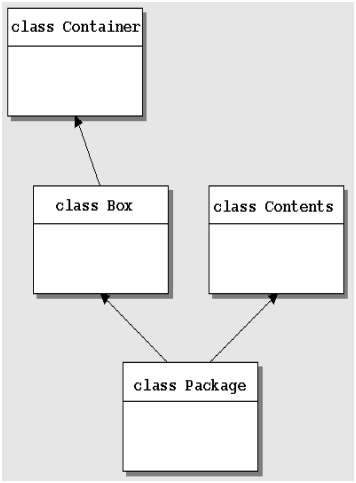
A maior parte das **linguagens orientadas** a **objetos** implementam o encapsulamento baseado em **propriedades privadas**, ligadas a métodos especiais chamados ***getters*** e ***setters***, que irão retornar e setar o **valor** da **propriedade**, respectivamente. Essa atitude **evita** o **acesso direto** a **propriedade** do objeto, adicionando uma outra camada de segurança à aplicação.

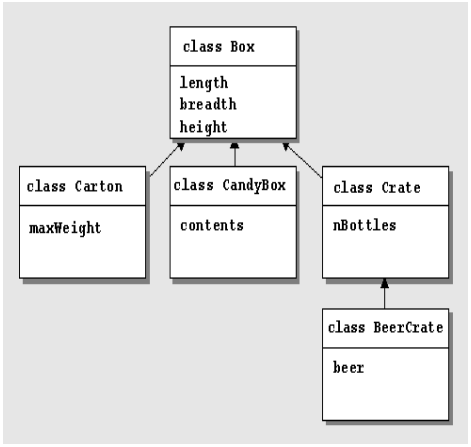
Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da **televisão**, **não sabemos** o que está **acontecendo internamente**. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

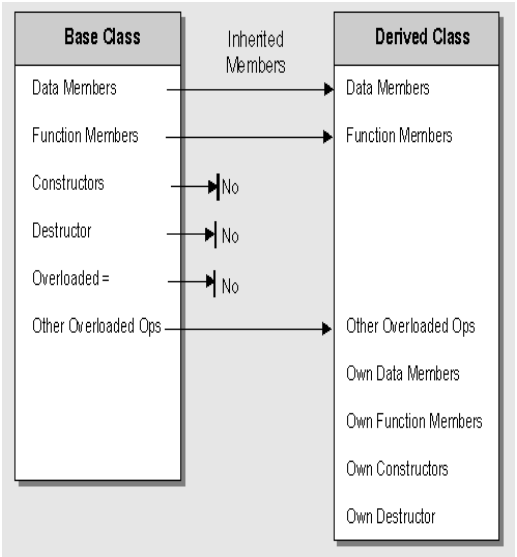
3.3. Herança

O **reuso** de **código** é uma das grandes **vantagens** da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como ***herança***. Essa característica **otimiza** a **produção** da **aplicação** em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus "ancestrais". A questão da **herança varia** bastante de **linguagem** para **linguagem**. Em algumas delas, como **C++**, há a questão da **herança múltipla**. Isso, essencialmente, significa que o objeto pode herdar características de **vários** "**ancestrais**" ao mesmo tempo diretamente. Em outras palavras, cada objeto pode possuir quantos pais for necessário. Devido a problemas, essa prática não foi difundida em linguagens mais modernas, que utilizam outras artimanhas para criar uma espécie de herança múltipla. Outras linguagens orientadas a objetos, como C#, trazem um objeto base para todos os demais. A classe *object* fornece características para todos os objetos em C#, sejam criados pelo usuário ou não.







3.4. Polimorfismo

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus "ancestrais". Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o *polimorfismo* consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Como um exemplo, temos um objeto genérico "Eletrodoméstico". Esse objeto possui um método, ou ação, "Ligar()". Temos dois objetos, "Televisão" e "Geladeira", herdados de Eletrodoméstico, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método "Ligar()".

Com relação ao polimorfismo, valem algumas observações. Como se trata de um assunto que está intimamente conectado à herança, entender os dois juntamente é uma boa ideia. Outro ponto é o fato de que as linguagens de programação implementam o polimorfismo de maneiras diferentes. O C#, por exemplo, faz uso de método virtuais (com a palavra-chave *virtual*) que podem ser reimplementados (com a palavra-chave *override*) nas classes filhas. Já em Java, apenas o atributo "@Override" é necessário.

Foo.h

```
#pragma once

class Foo {
public:
    Foo(void);
    ~Foo(void);
    void print(void);
};
```

Foo.cpp

```
#include <iostream>
#include "Foo.h"

using namespace std;

Foo::Foo(void) {
}


Foo::~Foo(void) {
}


void Foo::print(void) {
    cout << "Foo!" << endl;
}
```

Bar.h

```
#pragma once
#include "Foo.h"

class Bar : public Foo {
public:
    Bar(void);
    ~Bar(void);
    void print(void);
};
```

Bar.cpp

```
#include <iostream>
#include "Bar.h"

using namespace std;

Bar::Bar(void) {
}


Bar::~Bar(void) {
}


void Bar::print(void) {
    cout << "Bar!" << endl;
}
```

Função main:

```
#include "stdafx.h"
#include "Foo.h"
#include "Bar.h"


int main() {

    Foo* f = new Foo();
    Foo* b = new Bar();

    f->print();
    b->print();

    return 0;

}
```

```
#pragma once

class Foo {
public:
    Foo(void);
    ~Foo(void);
    virtual void print(void);
};
```

Esses quatro pilares são essenciais no entendimento de qualquer linguagem orientada a objetos e da orientação a objetos como um todo. Cada linguagem irá implementar esses pilares de uma forma, mas essencialmente, deve ser observada o conceto envolvido. Apenas a questão da herança, como comentado, que pode trazer variações mais bruscas, como a presença de herança múltipla. Além disso, o encapsulamento também é feito de maneiras distintas nas diversas linguagens, embora os *getters* e *setters* sejam praticamente onipresentes.

4 – Considerações Finais

A programação orientada a objetos traz uma ideia muito interessante: a **representação** de **cada elemento** em **termos** de um **objeto**, ou **classe**. Esse tipo de representação procura aproximar o sistema que está sendo criado ao que é observado no mundo real, e um objeto contém características e ações, assim como vemos na realidade. Esse tipo de representação traz algumas vantagens muito interessantes para os desenvolvedores e também para o usuário da aplicação.

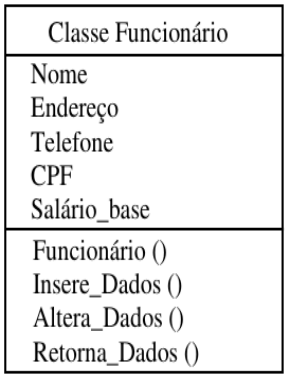
A **reutilização** de **código** é um dos **principais requisitos** no desenvolvimento de software atual. Com a **complexidade** dos **sistemas** cada vez maior, o **tempo** de desenvolvimento iria **aumentar** consideravelmente caso **não** fosse **possível** a **reutilização**. A orientação a objetos permite que haja uma reutilização do código criado, **diminuindo** o **tempo** de desenvolvimento, bem como o **número** de **linhas** de **código**. Isso é possível devido ao fato de que as linguagens de programação orientadas a objetos trazem **representações muito claras** de cada um dos **elementos**, e esses elementos normalmente **não** são **interdependentes**. Essa independência entre as partes do software é o que permite que esse código seja reutilizado em outros sistemas no futuro.

Outra grande vantagem que o desenvolvimento orientado a objetos traz diz respeito a **leitura** e **manutenção** de **código**. Como a representação do sistema se aproxima muito do que vemos na vida real, o entendimento do sistema como um todo e de cada parte individualmente fica muito mais simples. Isso permite que a equipe de desenvolvimento não fique dependente de uma pessoa apenas, como acontecia com frequência em linguagens estruturadas como o C, por exemplo.

A **criação** de **bibliotecas** é outro ponto que é muito mais simples com a orientação a objetos. No caso das linguagens estruturadas, como o C, temos que as bibliotecas são coleções de procedimentos (ou funções) que podem ser reutilizadas. No caso da **POO**, entretanto, as bibliotecas trazem representações de classes, que são muito mais claras para permitirem a reutilização.

Quanto a **interfaces**, estas são úteis porque elas estabelecem contratos. Se uma classe implementar uma interface você vai poder referenciar instâncias da classe pela interface tendo somente acesso aos membros definidos na interface. Isso significa basicamente, que você garante que a classe apresentará um certo comportamento sem que se saiba a priori como esse comportamento é implementado. Por exemplo, pense na interface IComparable do C#, ela determina que classes que a implementam possuam um método de comparação. Essa ideia pode ser usada por exemplo para construir uma classe de árvore binária como mostrado no livro Microsoft Visual C# 2010 passo a passo, pois assinando essa interface você garante que existe alguma forma de comparar objetos daquela classe. **Em resumo: Interfaces são contratos que te ajudam a escrever códigos com baixo acoplamento e alta coesão.**

É interessante comentar que, para uma melhor visualização e entendimento da hierarquia de classes, normalmente utiliza-se uma notação gráfica para a sua representação. Entre várias notações podemos citar *Booch*, *OMT* e *UML*. *OMT* por exemplo – *Object Modeling Technique*, descreve os objetos no sistema e seus relacionamentos da seguinte forma:



Entretanto, **nem tudo** é **perfeição** na **programação orientada** a **objetos**. A execução de uma aplicação orientada a objetos é **mais lenta** do que o que vemos na programação estruturada, por exemplo. Isso acontece **devido** à **complexidade** do **modelo**, que traz representações na forma de classes. Essas representações irão fazer com que a execução do programa tenha muitos desvios, diferente da execução sequencial da programação estruturada. Esse é o grande motivo por trás da preferência pela linguagem C em hardware limitado, como sistemas embarcados. Também é o motivo pelo qual a programação para sistemas móveis como o Google Android, embora em Java (linguagem orientada a objetos), seja feita o menos orientada a objetos possível.

No momento atual em que estamos, tecnologicamente essa execução mais lenta não é sentida. Isso significa que, em termos de desenvolvimento de sistemas modernos, a programação orientada a objetos é a mais recomendada devido as vantagens que foram apresentadas. Essas vantagens são derivadas do modelo de programação, que busca uma representação baseada no que vemos no mundo real.

Referencial Bibliográfico

- LARMAN, Graig. Utilizando UML e padrões: uma introdução à análise e ao projeto orientado a objetos e ao desenvolvimento interativo. 3. ed. Porto Alegre: Bookman, 2007.
- DEITEL, H. M.; DEITEL, P.J. C++ Como Programar. 5ª. edição. Pearson, 2006.
- Sítio na WEB: https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264
- BOOCH, Grady. Object-Oriented Analysis and Design with Applications (3rd Edition), Addison Wesly, 2007.
- GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. (18)