**1. PROGRAMAÇÃO ORIENTADA A OBJETOS**

Segundo Moreira Neto (2009), a programação orientada a objetos (POO) pode ser definida como um modelo que tem como base a execução de métodos (pequenas funções que atuam diretamente sobre os dados de um objeto), levando em consideração a maneira como o usuário enxerga o sistema e suas funções.

O termo Programação Orientada a Objetos foi criado por Alan Kay, criador da linguagem de programação Smalltalk-80, mais conhecida com Smalltalk. Mas, antes disso foi criado por Ole Johan Dahl e Kristen Nygaard a linguagem Simula 67, em 1967, onde já se aplicava o conceito de POO. Com o passar dos anos, foi-se descobrindo que a POO tornava a codificação mais simples de fácil manutenção, tornando assim os softwares mais duradouros, precisos e cada vez mais livres de erros. Segundo um estudo realizado por Deitel (2009), com o passar dos anos houve uma mudança significativa por parte das grandes empresas em relação ao desenvolvimento usando esse modelo.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Através desse relacionamento, o programador consegue estabelecer uma relação de herança e polimorfismo entre sua classes e assim tornando o seu código mais limpo, de fácil manutenção e com menos redundâncias.

Para Moreira Neto (2009), as principais definições da orientação a objetos são:

**1.1 Classe**

“Uma classe é composta de métodos (funções) e propriedades (variáveis). Ela nada mais é que o modelo de um objeto, sendo assim é a partir dela que é criado, instanciado, um objeto.”

Abaixo temos um exemplo de classe:

|  |
| --- |
| class Pessoa {  private String nome;  public void setSobrenome(String sobrenome) { nome += “ ” + sobrenome; }  } |

**1.2 Objeto**

**“**O objeto é a representação de uma classe na memória. Sua existência só ocorre quando em tempo de execução ocorre a instanciação de uma classe, dando assim a sua origem.”

Abaixo temo um exemplo de objeto:

|  |
| --- |
| Pessoa pessoa = new Pessoa(); |

*Observação: Não devemos confundir variável de referência com objeto. No exemplo acima o objeto é* ***Pessoa*** *e pessoa é apenas a variável que o referencia, para que assim o programador possa manipular seu estado em memória.*

Sendo assim podemos concluir que o modelo conceitual de Programação Orientada a Objetos (POO), consegue expressar de uma maneira mais natural, mais próximo da realidade humana e muito mais simples a simulação/solução dos problemas humanos em um computador.

**2. MODIFICADORES DE ACESSO**

Modificadores de acesso, são palavras reservadas que utilizamos para restringir o acesso à propriedades, métodos ou classes, Sierra (2009). Esse conceito é bastante utilizado no Unified Modeling Language (UML), onde pode ser criado o diagrama de classe do sistema. Esse diagrama é apenas um esboço das relações entre as classes e seus modificadores de acesso.

**2.1 Público**

Esse modificador concede acesso total para que outros objetos ou classes possam utilizá-lo. Quando utilizado em métodos ou propriedades, indica que qualquer objeto pode ter acesso a ele. Quando utilizado em classes ele indica que classes de pacotes diferentes podem ter acesso a ela. Esse modificador de acesso é representado no UML com o símbolo de “+”.

**2.2 Privado**

Ele torna o acesso restrito apenas à métodos ou propriedades. Apenas à classe que o contém pode acessá-lo. Quando instanciado um objeto que possui algum de seus métodos ou propriedades privados, apenas ele, de maneira interna, consegue os utilizar. Sua representação no UML é o símbolo “-”.

**2.3 Protegido**

É considerado um meio termo entre o público e o privado. Quando utilizado em métodos ou em propriedades ele concede o acesso à classe que o contém e à classes que possam estabelecer uma relação de herança com ela. O mesmo conceito do privado é dado à objetos instanciados que possuam propriedades ou métodos protegidos. Sua representação no UML com o símbolo “#”.

**2.4 Estático**

É o modificador de acesso que pode ser utilizado em conjunto com os outros citados acima. Quando utilizado em uma propriedade, representa que seu valor será a mesmo para qualquer instância da classe que o pertence. Em outras palavras, mesmo que sejam feitas várias instâncias daquela classe, todas elas irão compartilhar a mesma região em memória daquela propriedade, sendo assim, uma vez aquela propriedade tendo seu valor alterado, em todas as outras instâncias também será alterado. Já quando utilizado em métodos, costuma-se dizer que o método não mais pertence a instância e sim à classe. Isso quer dizer que não é mais necessário criar uma instância da classe para chamar o método bastando apenas colocar o nome da classe em logo seguida o nome do método (Exemplo: **NomeDaClasse.*metodoEstatico();*** ). Sua representação no UML é com o nome da propriedade ou método todo sublinhado (Exemplo: propriedade ou metodo).

**2.5 Abstrato**

Esse modificador é utilizado em conjunto com o público e pode ser apenas utilizado em métodos ou classes. Quando utilizado em métodos deverão ser desenvolvidos/implementados nas subclasses. Quando a classe que contiver métodos abstratos for herdada, os referidos métodos deverão ser implementados, caso contrário, a classe que estabelece a relação de herança deverá ser declarada como abstrata. Uma vez utilizado em um método, a classe também desverá ser assinada como abstrata e consequentemente não podendo ser instanciadas. Métodos e classes abstratas são escritos em itálico no UML.

**3. PRINCIPAIS CARACTERÍSTICAS**

Segundo Sierra (2009), podemos citar como a principal característica na POO a herança entre as classes que permite ao código definido em uma classe ser reutilizado em outras. A partir do conceito de herança podemos extrair outros, abaixo irei listar os principais citados pela autora:

**3.1 Encapsulamento**

É a premissa de que apenas o objeto consegue acessar diretamente determinadas propriedades. Para que possa existir uma interação entre os objetos, eles devem utilizar os métodos que concedam o acesso à essas propriedades (os “get”s e “set”s). São esses métodos que permitem o acesso de agentes externos ao estado do objeto e também é uma forma de proteger o objeto. Uma propriedade sendo acessada por um método “get”, permite que caso ela não esteja formatada corretamente ocorra antes algum tipo de validação. Isso serve também para métodos “set”, onde caso o valor que queira ser atribuído à uma propriedade não esteja de acordo com a regra de negócio ele seja descartado. Assim podemos garantir uma maior integridade no estado do objeto e também garantir um maior funcionamento do software de acordo com a regra de negócios. Abaixo temos um diagrama em de classes do UML que exemplifica isso:

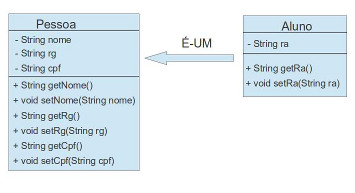


*Figura 01: representação de uma classe pelo diagrama de classes do UML.*

*Fonte: Elaborado pelo autor do trabalho de conclusão de curso.*

**3.2 Herança**

O conceito é igual ao do mundo real, onde um ser vivo herda características de seu descendente e possui as próprias. A herança lhe permite abstrair características comuns e centralizá - las em uma superclasse, para que subclasses possam usá – las. Uma superclasse nada mais é que uma classe que centraliza métodos e propriedades comuns a mais de uma classe, e a subclasse é a classe que herda de uma superclasse e assim possui características próprias mais as da superclasse. Um bom exemplo é a relação Pessoa – Aluno, onde um Aluno antes de mais nada é uma pessoa, gerando assim uma relação de herança. Abaixo temos um exemplo dessa relação:



*Figura 02: representação de uma classe herança entre classe pelo diagrama de classes do UML.*

*Fonte: Elaborado pelo autor do trabalho de conclusão de curso.*

**3.3 Polimorfismo**

No caso do Polimorfismo eu irei citar a definição de Moreira Neto (2009), pois sua definição está mais completa:

“- Polimorfismo é o princípio pelo qual, a partir de uma subclasse, criam-se (instanciam - se) objetos utilizando referência do tipo de uma superclasse existente na hierarquia. O uso do polimorfismo introduz conceitos relacionados a herança trazendo a motivação para a definição de classes abstratas, métodos abstratos, melhorando a modelagem para garantir a evolução da aplicação.”

Tendo isso como base e nos utilizando dos exemplos citados anteriormente, na relação

Pessoa – Aluno, podemos extrair outros dois conceitos abordados por Kathy Sierra, o upcasting e o downcasting.

**3.3.1 Upcasting:** é quando uma subclasse tornar-se uma superclasse, exemplo:

|  |
| --- |
| 1 Aluno aluno = new Aluno();  2 Pessoa pessoa = (Pessoa)aluno; //upcasting explícito  ou |
| 2 Pessoa pessoa = aluno; //upcasting implícito |
|  |

No exemplo acima, na linha um criamos uma variável de referência denominada *aluno* que é tipo **Aluno**, e logo após instanciamos um objeto **Aluno**. Sendo assim, *aluno* aponta para um região de memória onde existe um objeto do tipo **Aluno**. Logo após, na linha dois, criamos uma variável de referência denominada *pessoa* que é do tipo **Pessoa**, em seguida atribuímos a mesma referência em memória que *aluno* têm. Isso é possível pois existe uma relação de herança entre **Aluno** e **Pessoa**. Percebemos então que *aluno* conseguirá acessar todos os métodos e propriedades da classe **Aluno** e **Pessoa**, porém *pessoa* mesmo que esteja apontando para um região de memória que possui um objeto do tipo **Aluno** ela irá conseguir apenas acessar os métodos e propriedades que sejam da classe **Pessoa**, isso porque *pessoa* é uma variável de referência do tipo **Pessoa**, sendo assim ela consegue apenas acessar métodos e propriedades de **Pessoa**.

**3.3.2 Downcasting:** é quando uma superclasse torna-se uma subclasses, exemplo:

|  |
| --- |
| 1 Pessoa pessoa = new Aluno();  2 Aluno aluno = (Aluno)pessoa; //downcasting explícito |

No exemplo acima ocorre o mesmo conceito do upcasting, onde podemos alternar as referências para um objeto em memória, desde que haja uma relação de herança entre ambos.

**4. CONCLUSÃO**

Conclui-se que a programação orientada a objetos, mesmo que seja um conceito antigo, foi amplamente difundido nos anos 90 e que ainda hoje é utilizado por várias empresas. Seu modelo é considerado o mais próximo de exemplificar a realidade humana em um ambiente computacional, o que torna assim a vida de um software mais duradouro e de fácil manutenção.