Polimorfismo (informática)

Origem: Wikipédia, a enciclopédia livre.

Na programação orientada a objetos, o **polimorfismo** permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato). O termo ***polimorfismo*** é originário do grego e significa "muitas formas" (*poli* = muitas, *morphos* = formas).

O polimorfismo é caracterizado quando duas ou mais classes distintas têm métodos de mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto.[1]

Uma das formas de implementar o polimorfismo é através de uma classe abstrata, cujos métodos são declarados mas não são definidos, e através de classes que herdam os métodos desta classe abstrata.[2]

Índice

Tipos de polimorfismo

Existem quatro tipos de polimorfismo que a linguagem pode ter (atente para o fato de que nem toda linguagem orientada a objeto tem implementado todos os tipos de polimorfismo):

- Universal
  - Inclusão - um ponteiro para classe mãe pode apontar para uma instância de uma classe filha (exemplo em Java: `List lista = new LinkedList();` (tipo de polimorfismo mais básico que existe)
  - Paramétrico - se restringe ao uso de templates (C++, por exemplo) e generics (C#/Java)
- Ad-Hoc
  - Sobrecarga - duas funções/métodos com o mesmo nome mas assinaturas diferentes
  - Coerção - conversão implícita de tipos sobre os parâmetros de uma função

Exemplos

Suponha a seguinte classe escrita em Java:

```
public abstract class OperacaoMatematica {
    public abstract double calcular(double x, double y);
}
```

Esta é uma classe abstrata que representa qualquer operação matemática. Podemos imaginar diversas operações que se encaixam na sua interface, como soma, subtração, multiplicação ou divisão, entre outras. Note que, mesmo que a natureza do cálculo mude, a semântica do método calcular não muda, ou seja, ele sempre calculará o resultado da operação matemática que está sendo

trabalhada. Definamos então, duas subclasses, Soma e Subtracao, que extendem a classe `OperacaoMatematica`:

```
public class Soma extends OperacaoMatematica {
    public double calcular(double x, double y) {
        return x + y;
    }
}

public class Subtracao extends OperacaoMatematica {
    public double calcular(double x, double y) {
        return x - y;
    }
}
```

O seguinte trecho de código demonstra o uso do polimorfismo:

```
public class Contas {
    public static void mostrarCalculo(OperacaoMatematica operacao, double x, double y) {
        System.out.println("O resultado é: " + operacao.calcular(x, y));
    }

    public static void main(String args[]) {
        // Primeiro calculamos uma soma
        Contas.mostrarCalculo(new Soma(), 5, 5); // Imprime o resultado é: 10

        // Depois uma subtração
        Contas.mostrarCalculo(new Subtracao(), 5, 5); // Imprime o resultado é: 0
    }
}
```

Embora o método *calcular* tenha sido chamado duas vezes no interior de *mostrarCalculo*, os tipos (isto é, as classes das instâncias) utilizados como parâmetros eram diferentes. De fato, o comportamento de cada tipo era exatamente oposto. É comum definir sobrecarga de métodos ou simplesmente sobrecarga como uma forma de polimorfismo (chamado de polimorfismo ad-hoc). Nesse caso, implementa-se métodos com um mesmo nome, mudando apenas a lista de parâmetros que ele recebe. Digamos

```
    public static void mostrarCalculo(Soma soma, double x, double y) {
        System.out.println("O resultado é: " + soma.calcular(x, y));
    }

    public static void mostrarCalculo(Subtracao subtracao, double x, double y) {
        System.out.println("O resultado é: " + subtracao.calcular(x, y));
    }
```

Embora nesse caso o resultado possa ser o mesmo que aquele obtido com o uso de herança, no polimorfismo ad-hoc não existem garantias que os métodos sobrecarregados tenham o mesmo comportamento.

Benefícios do polimorfismo

Clareza e manutenção do código

Em linguagens de programação não-polimórficas, para implementar o método `mostrarCalculo`, seria necessário recorrer a uma enumeração com o tipo de operação e, dentro do método, testar o valor da enumeração, como no exemplo abaixo:

```
public void mostrarCalculo(String operacao, double x, double y) {
    System.out.print("O resultado é: ");

    switch (operacao) {
        case SOMA:
            System.out.print(x + y);
            break;
        case SUBTRACAO:
            System.out.print(x - y);
            break;

        // Outras operações...
        default:
            throw new UnsupportedOperationException();
            break;
    }
}
```

Além do código ser maior e mais difícil de ler, essa implementação tem outros problemas. Provavelmente esse não será o único método a utilizar operações matemáticas e, portanto, pode-se esperar não um, mas vários switchs como esse pelo código. Se uma nova operação for adicionada ao sistema, será necessário que todos os switchs sejam encontrados e substituídos, por exemplo. Com o polimorfismo, a modificação restringiria-se apenas a criação de uma nova classe!

Polimorfismo e padrões de projeto

Boa parte dos padrões de projeto de software baseia-se no uso de polimorfismo, por exemplo: Abstract Factory, Composite, Observer, Strategy, Template Method etc;

O polimorfismo também é usado em uma série de refatorações, como substituir condicional por polimorfismo.[3]

Referências

1. Steven F. Lott, *A Programmer's Introduction to Python*, *Building Skills in Python*, *Part III. Data + Processing = Objects*, *Chapter 22. Advanced Class Definition*, *Polymorphism* [em linha] (http://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch22s02.html)
2. Documentação do software livre Qt, *Academic Solutions to Academic Problems* [em linha] (http://doc.trolltech.com/qq/qq15-academic.html)
3. «Refactoring: Replace Conditional with Polymorphism» (http://www.refactoring.com/catalog/replaceConditionalWithPolymorphism.html). www.refactoring.com. Consultado em 10 de Março de 2011

Ver também

- Programação orientada a objetos
- UML
- Arquitetura de dados
- Administração de dados
- Modelagem de dados