Herança

Introdução

O relacionamento de herança está presente em linguagens de programação orientadas a objetos. Corresponde ao relacionamento de Generalização / Especialização presente no diagrama de classes da UML. O objetivo dessa associação é representar a ocorrência de herança entre classes, identificando as classes-mãe (ou superclasses), e classes filhas (ou subclasses).

Uma classe filha "herda" todas as características da classe-mãe, permitindo que características que são gerais entre várias classes possam ser inseridas na classe mãe e herdadas pelas classes filha sem a necessidade de reescrever o código.

A generalização / especialização ocorre quando existem duas ou mais classes com características muito semelhantes. Assim, para evitar ter de declarar atributos e / ou métodos idênticos cria-se uma classe geral em que são declarados os atributos e métodos comuns a todas as classes envolvidas no processo e, então, declaram-se classes especializadas ligadas à classe geral, que herdam todas as suas características (atributos e métodos).

A generalização / especialização pode também se vista como uma forma de reuso de código, mas esse não é o foco principal de sua aplicação. Devemos utilizá-la para modelar uma associação específica entre classes próximas (ou parecidas) que herdam características umas das outras.

Exemplo

Vamos começar com um exemplo bem simples para entender a utilidade da herança. Suponha que estamos construindo um sistema bancário, e que temos que manipular dois tipos de conta diferentes: ContaComum e ContaEspecial.

```
public class ContaComum {
    private Cliente cliente;
    private double saldo;
    public ContaComum(Cliente cliente) {
        this.cliente = cliente;
    }
    public Cliente getCliente() {
        return cliente;
    }
    public void setCliente(Cliente cliente) {
        this.cliente = cliente;
    }
    public double getSaldo() {
        return saldo;
    }
    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
    public String toString(){
        return new String("Cliente: "+cliente.getNome()+", saldo= "+saldo);
    }
}
```

Figura 1 - Classe ContaComum sem Hierarquia.

A ContaComum tem como atributos o cliente (um objeto da classe Cliente) e o saldo (double) – vamos manter o exemplo bem simples. Além disso possui os métodos Getters & Setters para cliente e saldo e um método toString() para imprimir os dados daquela conta.

A ContaEspecial tem os mesmos atributos e métodos da ContaComum, mas tem também o atributo taxaDeJuros e o método aplicaCorrecao() para permitir que mensalmente a taxa de juros seja aplicada sobre o saldo da conta.

```
public class ContaEspecial {
    private Cliente cliente;
    private double saldo;
    private double taxaDeJuros;
    public ContaEspecial(Cliente cliente) {
        this.cliente = cliente;
    }
    public Cliente getCliente() {
        return cliente;
    }
    public void setCliente(Cliente cliente) {
        this.cliente = cliente;
    }
    public double getSaldo() {
        return saldo;
    }
    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
    public double getTaxaDeJuros() {
        return taxaDeJuros;
    }
    public void setTaxaDeJuros(double taxaDeJuros) {
        this.taxaDeJuros = taxaDeJuros;
    }
    public void aplicaCorrecao(){
        saldo = saldo * (1+taxaDeJuros);
    }
    public String toString(){
        return new String("Cliente: "+cliente.getNome()+", saldo= "+saldo);
    }
}
```

Figura 2 - Classe ContaEspecial sem Hierarquia.

Observe que para os dois casos o código é muito parecido. Nesse caso, quando identificamos a necessidade das classes, podemos também identificar que as classes possuem um relacionamento especial entre si, ou seja, as duas são tipos específicos de Conta Corrente. Poderíamos desta forma criar uma classe mais geral chamada ContaCorrente que contivesse o que é comum entre ContaComum e ContaEspecial e, em seguida, fazer com que essas duas classes se tornassem classes derivadas (filhas) da classe ContaCorrente.

É comum quando definimos relacionamentos entre classes do tipo Hierarquia que a classe mais geral seja definida como uma classe abstrata. Classes abstratas não podem ser instanciadas diretamente em objetos. A instanciação deve ocorrer necessariamente por meio de uma de suas classes especializadas (classes filha).

Quando uma classes possui métodos definidos como abstratos, ela precisa necessariamente ser também definida como abstrata. Um método abstrato não possui implementação, a implementação desse tipo de método deve obrigatoriamente ser fornecida pela classe filha se esta for concreta.

A presença desse tipo de método na classe mais geral serve para definir um tipo de dados. Ou seja, a classe mais geral é utilizada para definir o tipo de parâmetros, por exemplo, enquanto instâncias das classes filhas é que são efetivamente passadas como parâmetros.

Exercícios

1. Defina na linguagem Dart a classe abstrata “ContaCorrente" e suas classes derivadas “ContaComum" e “ContaEspecial”. O comportamento destas duas últimas deve ser semelhante ao código em Java mostrado neste documento, mas lembre que os códigos que ambas possuem em comum devem estar na classe "ContaCorrente".
2. Defina uma classe Banco que armazene várias contas corrente. Essas contas podem ser contas comuns ou especiais.
3. Dentro da classe Banco defina o método "transferencia", que recebe uma conta de origem, uma conta de destino e o valor a ser transferido entre as contas. Não permita que a transferência ocorra caso a conta origem não tenha saldo suficiente.
4. Modifique a classe ContaEspecial para que ela tenha um limite de crédito, ou seja, a conta pode ficar negativa até um certo limite. Essa modificação deve ficar transparente para o resto do sistema, ou seja, o código implementado no exercício 3 deve continuar funcionando e o sistema deve agora considerar a existência de um limite de crédito quando forem feitas transferências a partir de uma ContaEspecial.