Sistema de Gestão de Transportes com Polimorfismo

Introdução ao Problema

Imagine que você está desenvolvendo um sistema para gerenciar diferentes tipos de veículos de transporte em uma empresa de logística. A empresa possui caminhões, vans e motocicletas, cada um com diferentes métodos de cálculo de custo operacional, mas todos eles têm características comuns, como placa e modelo. Inicialmente, você pode pensar em criar classes separadas para cada tipo de veículo com seus próprios métodos de cálculo de custo operacional. No entanto, essa abordagem pode se tornar ineficiente e difícil de manter à medida que a frota cresce. Vamos explorar uma implementação inicial e identificar os problemas antes de propor uma solução ideal.

Implementação Inicial (Incorreta)

Aqui está uma implementação inicial onde cada tipo de veículo é representado por uma classe separada:

```
public class Caminhao
{
    public string Placa { get; set; }
    public string Modelo { get; set; }
    public decimal CustoPorKm { get; set; }
    public int Quilometragem { get; set; }

    public decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}

public class Van
{
    public string Placa { get; set; }
    public string Modelo { get; set; }
    public decimal CustoPorKm { get; set; }
    public int Quilometragem { get; set; }

    public decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}

public class Motocicleta
{
    public string Placa { get; set; }
    public string Modelo { get; set; }
    public decimal CustoPorKm { get; set; }
```

```
    public int Quilometragem { get; set; }

    public decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}
```

Problemas com a Implementação Inicial

Vamos refletir sobre os problemas dessa abordagem:

1. **Duplicação de Código**: As classes `Caminhao`, `Van` e `Motocicleta` possuem atributos comuns, como `Placa` e `Modelo`, e métodos, como `CalcularCustoOperacional`. Isso é ineficiente porque estamos repetindo o mesmo código em várias classes.
2. **Manutenção Difícil**: Se precisar adicionar ou alterar um atributo ou método comum, terá que fazer isso em várias classes, aumentando a chance de erro.

Solução Usando Polimorfismo

Para resolver esses problemas, podemos usar polimorfismo. Vamos criar uma classe base ou uma interface que defina os métodos e atributos comuns e implementar essa interface em classes específicas para cada tipo de veículo.

Implementação Melhorada Usando Polimorfismo

```
// Classe base para veículos
public abstract class Veiculo
{
    public string Placa { get; set; }
    public string Modelo { get; set; }

    public abstract decimal CalcularCustoOperacional();
}

// Classe para caminhões
public class Caminhao : Veiculo
{
    public decimal CustoPorKm { get; set; }
    public int Quilometragem { get; set; }

    public override decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}

// Classe para vans
public class Van : Veiculo
{
    public decimal CustoPorKm { get; set; }
    public int Quilometragem { get; set; }
```

```
    public override decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}

// Classe para motocicletas
public class Motocicleta : Veiculo
{
    public decimal CustoPorKm { get; set; }
    public int Quilometragem { get; set; }

    public override decimal CalcularCustoOperacional()
    {
        return CustoPorKm * Quilometragem;
    }
}

// Exemplo de uso
public class Program
{
    public static void Main()
    {
        List<Veiculo> veiculos = new List<Veiculo>
        {
            new Caminhao { Placa = "ABC-1234", Modelo = "Caminhão X", CustoPorKm =
5.0m, Quilometragem = 1000 },
            new Van { Placa = "DEF-5678", Modelo = "Van Y", CustoPorKm = 3.0m,
Quilometragem = 2000 },
            new Motocicleta { Placa = "GHI-9101", Modelo = "Moto Z", CustoPorKm =
1.0m, Quilometragem = 500 }
        };

        foreach (var veiculo in veiculos)
        {
            Console.WriteLine($"Custo operacional do veículo ({veiculo.Placa} -
{veiculo.Modelo}): {veiculo.CalcularCustoOperacional()}");
        }
    }
}
```

Benefícios do Polimorfismo em Cenários Reais

Vamos refletir sobre os benefícios que essa abordagem traz:

- **Reutilização de Código**: Reduzimos a duplicação de código comum entre diferentes tipos de veículos, facilitando a manutenção.
- **Organização e Estrutura**: O polimorfismo permite uma organização melhor do código, refletindo a estrutura lógica da frota.
- **Flexibilidade**: O polimorfismo facilita a criação de novos tipos de veículos com comportamentos específicos, tornando o sistema mais flexível.

Desvantagens e Considerações

Mas nem tudo são flores. Precisamos estar atentos a alguns pontos:

- **Complexidade**: O uso de classes abstratas e polimorfismo pode adicionar complexidade ao código, tornando-o mais difícil de entender.
- **Manutenção das Classes Base**: Mudanças na classe base podem exigir atualizações em todas as classes que a herdam.

Conclusão

O polimorfismo é uma ferramenta poderosa e prática para o desenvolvimento de sistemas complexos. Ele permite que diferentes tipos de objetos compartilhem características comuns, enquanto possuem comportamentos específicos. Utilizando-o corretamente, podemos obter um código mais limpo, organizado e fácil de manter.

Exercícios de Fixação

Exercícios Teóricos

1. Explique a diferença entre polimorfismo de sobrecarga e polimorfismo de sobrescrita.
2. Descreva um cenário em que o polimorfismo pode ser usado para simplificar o código.

Exercícios Práticos

1. Implemente uma hierarquia de classes para um sistema de gerenciamento de estoque, incluindo classes para `ProdutoPerecivel`, `ProdutoNaoPerecivel`, utilizando polimorfismo para calcular a validade de diferentes produtos.
2. Crie uma aplicação que gerencie diferentes tipos de contas bancárias (e.g., `ContaCorrente`, `ContaPoupanca`), utilizando uma classe base `ContaBancaria` e classes específicas para cada tipo de conta.