Polimorfismo em Programação Orientada a Objetos

Introdução ao Polimorfismo

Polimorfismo é um dos pilares da programação orientada a objetos (POO). Ele permite que objetos de diferentes classes sejam tratados como objetos de uma classe comum. É um conceito que permite que uma interface seja usada para representar diferentes tipos de objetos.

Tipos de Polimorfismo

1. **Polimorfismo de Sobrecarga (Overloading)**: Permite criar várias versões de um método na mesma classe, diferenciando-os pelo número ou tipo de parâmetros.
2. **Polimorfismo de Sobrescrita (Overriding)**: Permite que uma subclasse forneça uma implementação específica de um método que já é definido na sua superclasse.

Exemplo de Polimorfismo de Sobrecarga

Introdução ao Problema

Imagine que você está desenvolvendo um sistema para calcular os pagamentos de diferentes tipos de freelancers e funcionários em uma empresa. Dependendo do tipo de trabalhador (freelancer, funcionário horário, funcionário mensal), o cálculo do pagamento pode variar. Vamos ver como uma implementação inicial pode ser feita usando sobrecarga de métodos para resolver esse problema.

Implementação Inicial (Incorreta)

```
public class PagamentoCalculadora
{
    public decimal CalcularPagamento(decimal taxaPorHora, int horasTrabalhadas)
    {
        return taxaPorHora * horasTrabalhadas;
    }

    public decimal CalcularPagamento(decimal salarioMensal)
    {
        return salarioMensal;
    }

    public decimal CalcularPagamento(decimal taxaPorProjeto, int projetosCompletados,
decimal bonusPorProjeto)
    {
        return (taxaPorProjeto * projetosCompletados) + (bonusPorProjeto *
projetosCompletados);
    }
}
```

Problemas com a Implementação Inicial

Você consegue identificar os problemas nesta abordagem? Vamos explorar alguns pontos críticos:

1. **Claridade do Código**: O método `CalcularPagamento` está sobrecarregado com diferentes assinaturas, o que pode ser confuso e dificultar a manutenção do código.
2. **Flexibilidade**: Esta abordagem não é flexível para adicionar novos tipos de cálculos de pagamento no futuro sem modificar a classe existente.

Solução Usando Polimorfismo com Herança e Interfaces

Para resolver esses problemas, podemos usar polimorfismo com interfaces. Que tal criar uma interface que defina o método de cálculo de pagamento e implementar essa interface em classes específicas para cada tipo de trabalhador? Vamos ver como isso funciona.

Implementação Melhorada Usando Polimorfismo

```
// Interface para cálculo de pagamento
public interface IPagamento
{
    decimal CalcularPagamento();
}

// Classe para freelancers
public class Freelancer : IPagamento
{
    public decimal TaxaPorHora { get; set; }
    public int HorasTrabalhadas { get; set; }

    public decimal CalcularPagamento()
    {
        return TaxaPorHora * HorasTrabalhadas;
    }
}

// Classe para funcionários horistas
public class FuncionarioHorista : IPagamento
{
    public decimal TaxaPorHora { get; set; }
    public int HorasTrabalhadas { get; set; }

    public decimal CalcularPagamento()
    {
        return TaxaPorHora * HorasTrabalhadas;
    }
}

// Classe para funcionários mensais
public class FuncionarioMensal : IPagamento
{
    public decimal SalarioMensal { get; set; }

    public decimal CalcularPagamento()
    {
        return SalarioMensal;
    }
}
```

```
// Classe para freelancers por projeto
public class FreelancerPorProjeto : IPagamento
{
    public decimal TaxaPorProjeto { get; set; }
    public int ProjetosCompletados { get; set; }
    public decimal BonusPorProjeto { get; set; }

    public decimal CalcularPagamento()
    {
        return (TaxaPorProjeto * ProjetosCompletados) + (BonusPorProjeto *
ProjetosCompletados);
    }
}

// Exemplo de uso
public class Program
{
    public static void Main()
    {
        List<IPagamento> trabalhadores = new List<IPagamento>
        {
            new Freelancer { TaxaPorHora = 50m, HorasTrabalhadas = 100 },
            new FuncionarioHorista { TaxaPorHora = 30m, HorasTrabalhadas = 160 },
            new FuncionarioMensal { SalarioMensal = 4000m },
            new FreelancerPorProjeto { TaxaPorProjeto = 1000m, ProjetosCompletados =
5, BonusPorProjeto = 200m }
        };

        foreach (var trabalhador in trabalhadores)
        {
            Console.WriteLine($"Pagamento calculado:
{trabalhador.CalcularPagamento()}");
        }
    }
}
```

Exemplo de Polimorfismo de Sobrescrita

Introdução ao Problema

Imagine que você está desenvolvendo um sistema para gerenciar notificações em uma plataforma online. Diferentes tipos de notificações (e.g., email, SMS, push notification) precisam ser enviadas de maneiras diferentes. Inicialmente, pode parecer fácil criar classes separadas para cada tipo de notificação, mas será que essa abordagem é a mais eficiente? Vamos ver como uma implementação inicial pode se tornar problemática.

Implementação Inicial (Incorreta)

```
public class NotificacaoEmail
{
    public string Destinatario { get; set; }
    public string Mensagem { get; set; }
```

```
    public void Enviar()
    {
        Console.WriteLine($"Enviando email para: {Destinatario} com a mensagem:
{Mensagem}");
    }
}

public class NotificacaoSMS
{
    public string Numero { get; set; }
    public string Mensagem { get; set; }

    public void Enviar()
    {
        Console.WriteLine($"Enviando SMS para: {Numero} com a mensagem: {Mensagem}");
    }
}

public class NotificacaoPush
{
    public string Dispositivo { get; set; }
    public string Mensagem { get; set; }

    public void Enviar()
    {
        Console.WriteLine($"Enviando notificação push para: {Dispositivo} com a
mensagem: {Mensagem}");
    }
}
```

Problemas com a Implementação Inicial

Você consegue identificar os problemas nesta abordagem? Vamos explorar alguns pontos críticos:

1. `Duplicação de Código`: Note que as classes `NotificacaoEmail`, `NotificacaoSMS` e `NotificacaoPush` possuem métodos idênticos, como Enviar. Isso não parece ineficiente?
2. `Manutenção Difícil`: Imagine que você precise adicionar um novo tipo de notificação ou mudar a forma de envio das notificações. Terá que fazer isso em várias classes, o que aumenta a chance de erro. Isso soa prático para você?

Solução Usando Polimorfismo

Para resolver esses problemas, podemos usar polimorfismo. Que tal criar uma classe base ou uma interface que defina o método de envio e implementar essa interface em classes específicas para cada tipo de notificação? Vamos ver como isso funciona.

Implementação Melhorada Usando Polimorfismo

```
// Classe base para notificações
public abstract class Notificacao
{
    public string Mensagem { get; set; }
```

```
    public abstract void Enviar();
}

// Classe para notificações por email
public class NotificacaoEmail : Notificacao
{
    public string Destinatario { get; set; }

    public override void Enviar()
    {
        Console.WriteLine($"Enviando email para: {Destinatario} com a mensagem:
{Mensagem}");
    }
}

// Classe para notificações por SMS
public class NotificacaoSMS : Notificacao
{
    public string Numero { get; set; }

    public override void Enviar()
    {
        Console.WriteLine($"Enviando SMS para: {Numero} com a mensagem: {Mensagem}");
    }
}

// Classe para notificações push
public class NotificacaoPush : Notificacao
{
    public string Dispositivo { get; set; }

    public override void Enviar()
    {
        Console.WriteLine($"Enviando notificação push para: {Dispositivo} com a
mensagem: {Mensagem}");
    }
}

// Exemplo de uso
public class Program
{
    public static void Main()
    {
        List<Notificacao> notificacoes = new List<Notificacao>
        {
            new NotificacaoEmail { Destinatario = "email@example.com", Mensagem =
"Olá, esta é uma notificação por email!" },
            new NotificacaoSMS { Numero = "123456789", Mensagem = "Olá, esta é uma
notificação por SMS!" },
            new NotificacaoPush { Dispositivo = "Dispositivo1", Mensagem = "Olá, esta
é uma notificação push!" }
        };
```

```
        foreach (var notificacao in notificacoes)
        {
            notificacao.Enviar();
        }
    }
}
```

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 notificações. Isso facilita a manutenção. Você percebe como isso pode tornar o código mais limpo?
- **Organização e Estrutura**: O polimorfismo permite uma organização melhor do código, refletindo a estrutura lógica das notificações. Isso faz sentido para você?
- **Flexibilidade**: O polimorfismo facilita a criação de novos tipos de notificações com comportamentos específicos. Consegue ver como isso torna 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. Será que isso pode ser um problema no futuro?
- **Manutenção das Classes Base**: Mudanças na classe base podem exigir atualizações em todas as classes que a herdam. Você acha que isso pode aumentar a complexidade do código?

Conclusão

O polimorfismo é uma ferramenta poderosa e prática para o desenvolvimento de sistemas empresariais 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. O que você acha, está pronto para aplicar esses conceitos?

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 desenho, incluindo classes para Circulo, Retangulo e Triangulo, utilizando polimorfismo para calcular a área de diferentes formas.

2. Crie uma aplicação que gerencie diferentes tipos de relatórios (e.g., Relatório Financeiro, Relatório de Vendas), utilizando uma classe base Relatorio e classes específicas para cada tipo de relatório.