Composição em Programação Orientada a Objetos

Introdução à Composição

A composição é um conceito fundamental na programação orientada a objetos (POO). Ela permite que os objetos sejam compostos de outros objetos, promovendo a reutilização de código e a flexibilidade. Em vez de criar hierarquias complexas de classes através da herança, a composição permite que os objetos utilizem outros objetos para realizar tarefas específicas.

O Que É Composição?

A composição é um princípio de design onde uma classe é composta de uma ou mais instâncias de outras classes. Isso é conhecido como "tem um" relacionamento, em contraste com a herança que é um "é um" relacionamento. Com a composição, uma classe pode reutilizar funcionalidades de outras classes sem precisar herdar delas.

Quando Usar Composição?

Promover a Reutilização de Código Use composição quando quiser reutilizar funcionalidades de várias classes sem precisar criar hierarquias complexas.

Exemplo Real: Sistema de Gestão de Projetos

Em um sistema de gestão de projetos, um projeto pode ter várias tarefas, e cada tarefa pode ter vários membros da equipe trabalhando nela. Em vez de criar uma hierarquia complexa de herança, podemos usar composição para modelar essa relação.

```
public class Tarefa
{
    public string Nome { get; set; }
    public DateTime DataInicio { get; set; }
    public DateTime DataFim { get; set; }

    public Tarefa(string nome, DateTime dataInicio, DateTime dataFim)
    {
        Nome = nome;
        DataInicio = dataInicio;
        DataFim = dataFim;
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Tarefa: {Nome}, Início: {DataInicio}, Fim: {DataFim}");
    }
}

public class MembroEquipe
{
    public string Nome { get; set; }
    public string Funcao { get; set; }
```

```
    public MembroEquipe(string nome, string funcao)
    {
        Nome = nome;
        Funcao = funcao;
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Membro: {Nome}, Função: {Funcao}");
    }
}

public class Projeto
{
    public string Nome { get; set; }
    public List<Tarefa> Tarefas { get; set; }
    public List<MembroEquipe> Membros { get; set; }

    public Projeto(string nome)
    {
        Nome = nome;
        Tarefas = new List<Tarefa>();
        Membros = new List<MembroEquipe>();
    }

    public void AdicionarTarefa(Tarefa tarefa)
    {
        Tarefas.Add(tarefa);
    }

    public void AdicionarMembro(MembroEquipe membro)
    {
        Membros.Add(membro);
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Projeto: {Nome}");
        Console.WriteLine("Tarefas:");
        foreach (var tarefa in Tarefas)
        {
            tarefa.MostrarDetalhes();
        }
        Console.WriteLine("Membros:");
        foreach (var membro in Membros)
        {
            membro.MostrarDetalhes();
        }
    }
}
```

```
// Exemplo de uso
public class Program
{
    public static void Main()
    {
        Projeto projeto = new Projeto("Projeto ABC");

        Tarefa tarefa1 = new Tarefa("Desenvolvimento", new DateTime(2024, 1, 1), new
DateTime(2024, 6, 30));
        Tarefa tarefa2 = new Tarefa("Testes", new DateTime(2024, 7, 1), new
DateTime(2024, 9, 30));

        MembroEquipe membro1 = new MembroEquipe("João", "Desenvolvedor");
        MembroEquipe membro2 = new MembroEquipe("Maria", "Testadora");

        projeto.AdicionarTarefa(tarefa1);
        projeto.AdicionarTarefa(tarefa2);
        projeto.AdicionarMembro(membro1);
        projeto.AdicionarMembro(membro2);

        projeto.MostrarDetalhes();
    }
}
```

Neste exemplo, a classe Projeto é composta de instâncias das classes Tarefa e MembroEquipe, permitindo uma estrutura flexível e reutilizável.

Reduzir o Acoplamento

Use composição para reduzir o acoplamento entre classes. Com a composição, as classes podem ser modificadas de forma independente, facilitando a manutenção e a evolução do código.

Exemplo Real: Sistema de Vendas

Em um sistema de vendas, um pedido pode ter vários itens e um cliente associado a ele. Usar composição ajuda a manter o sistema flexível e modular.

```
public class Item
{
    public string Nome { get; set; }
    public decimal Preco { get; set; }

    public Item(string nome, decimal preco)
    {
        Nome = nome;
        Preco = preco;
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Item: {Nome}, Preço: {Preco}");
    }
}
```

```
public class Cliente
{
    public string Nome { get; set; }
    public string Email { get; set; }

    public Cliente(string nome, string email)
    {
        Nome = nome;
        Email = email;
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Cliente: {Nome}, Email: {Email}");
    }
}

public class Pedido
{
    public int Numero { get; set; }
    public Cliente Cliente { get; set; }
    public List<Item> Itens { get; set; }

    public Pedido(int numero, Cliente cliente)
    {
        Numero = numero;
        Cliente = cliente;
        Itens = new List<Item>();
    }

    public void AdicionarItem(Item item)
    {
        Itens.Add(item);
    }

    public void MostrarDetalhes()
    {
        Console.WriteLine($"Pedido Número: {Numero}");
        Cliente.MostrarDetalhes();
        Console.WriteLine("Itens:");
        foreach (var item in Itens)
        {
            item.MostrarDetalhes();
        }
    }
}

// Exemplo de uso
public class Program
{
    public static void Main()
```

```
    {
        Cliente cliente = new Cliente("Ana", "ana@example.com");
        Pedido pedido = new Pedido(12345, cliente);

        Item item1 = new Item("Laptop", 1500m);
        Item item2 = new Item("Mouse", 50m);

        pedido.AdicionarItem(item1);
        pedido.AdicionarItem(item2);

        pedido.MostrarDetalhes();
    }
}
```

Neste exemplo, a classe Pedido é composta de instâncias das classes Item e Cliente, facilitando a reutilização e a manutenção do código.

Benefícios da Composição

Vamos refletir sobre os benefícios que essa abordagem traz:

1. **Flexibilidade**: A composição permite criar objetos complexos a partir de outros objetos mais simples, proporcionando maior flexibilidade no design do sistema.
2. **Reutilização de Código**: Facilita a reutilização de código, pois classes compostas podem usar funcionalidades de várias classes sem precisar herdar delas.
3. **Manutenção e Evolução**: Reduz o acoplamento entre classes, facilitando a manutenção e a evolução do sistema.

Desvantagens e Considerações

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

1. **Gerenciamento de Objetos**: A composição pode aumentar a complexidade no gerenciamento de objetos, especialmente em sistemas grandes com muitos componentes.
2. **Desempenho**: Em alguns casos, a composição pode introduzir overhead adicional, afetando o desempenho do sistema.

Conclusão

A composição é uma técnica poderosa na programação orientada a objetos, permitindo criar sistemas flexíveis e modulares. Utilizando composição, podemos reduzir a duplicação de código, aumentar a reutilização e facilitar a manutenção do sistema. O que você acha, está pronto para aplicar esses conceitos?

Exercícios de Fixação

Exercícios Teóricos

1. Explique como a composição ajuda a reduzir a duplicação de código em sistemas empresariais.
2. Descreva um cenário em que o uso de composição pode ser desfavorável devido à complexidade no gerenciamento de objetos.

Exercícios Práticos

1. Implemente uma hierarquia de classes para um sistema de gerenciamento de bibliotecas, incluindo classes para Livro, Autor e Categoria, utilizando composição.
2. Crie uma aplicação que gerencie diferentes tipos de eventos em uma agenda, utilizando composição para especializar classes como EventoPessoal, EventoProfissional e EventoSocial.