Sistema de Gerenciamento de Relatórios

Introdução ao Problema

Vamos imaginar um novo contexto. Suponha que você está desenvolvendo um sistema para gerenciar diferentes tipos de relatórios em uma empresa. O objetivo é criar um sistema que possa gerenciar relatórios financeiros, de vendas e de recursos humanos. Inicialmente, pode parecer uma boa ideia criar classes separadas para cada tipo de relatório, onde cada classe possui seus próprios atributos e métodos. No entanto, essa abordagem pode se tornar ineficiente e difícil de manter conforme a empresa 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 relatório é representado por uma classe separada:

```
public class RelatorioFinanceiro
{
    public string Titulo { get; set; }
    public DateTime Data { get; set; }
    public decimal TotalReceitas { get; set; }
    public decimal TotalDespesas { get; set; }

    public void GerarRelatorio()
    {
        Console.WriteLine($"Relatório Financeiro: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Receitas: {TotalReceitas}, Total de Despesas:
{TotalDespesas}");
    }
}

public class RelatorioDeVendas
{
    public string Titulo { get; set; }
    public DateTime Data { get; set; }
    public int TotalVendas { get; set; }
    public decimal ReceitaTotal { get; set; }

    public void GerarRelatorio()
    {
        Console.WriteLine($"Relatório de Vendas: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Vendas: {TotalVendas}, Receita Total:
{ReceitaTotal}");
    }
}

public class RelatorioDeRH
{
    public string Titulo { get; set; }
```

```
    public DateTime Data { get; set; }
    public int TotalFuncionarios { get; set; }
    public int NovasContratacoes { get; set; }

    public void GerarRelatorio()
    {
        Console.WriteLine($"Relatório de RH: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Funcionários: {TotalFuncionarios}, Novas
Contratações: {NovasContratacoes}");
    }
}
```

Problemas com a Implementação Inicial

Vamos refletir sobre os problemas dessa abordagem:

1. **Duplicação de Código**: Notem que as classes RelatorioFinanceiro, RelatorioDeVendas e RelatorioDeRH possuem atributos comuns, como Titulo e Data. Isso é ineficiente porque estamos repetindo o mesmo código em várias classes.
   - **Pergunta**: O que acontece se precisarmos adicionar um novo atributo comum a todos os relatórios, como Autor? Teríamos que modificar cada classe individualmente. Isso parece prático?
2. **Manutenção Difícil**: Se você precisar adicionar ou alterar um atributo comum, terá que fazer isso em várias classes, aumentando a chance de erro.
   - **Pergunta**: Se houver um erro em um atributo comum a todos os relatórios, como garantiríamos que ele fosse corrigido em todas as classes? Isso é eficiente para um sistema grande?

Solução Usando Classes Abstratas

Para resolver esses problemas, podemos usar classes abstratas. Vamos criar uma classe base que contém os atributos e métodos comuns e derivar classes específicas dela.

Implementação Melhorada Usando Classes Abstratas

```
public abstract class Relatorio
{
    public string Titulo { get; set; }
    public DateTime Data { get; set; }

    public Relatorio(string titulo, DateTime data)
    {
        Titulo = titulo;
        Data = data;
    }

    public abstract void GerarRelatorio();
}

public class RelatorioFinanceiro : Relatorio
{
    public decimal TotalReceitas { get; set; }
    public decimal TotalDespesas { get; set; }
```

```
    public RelatorioFinanceiro(string titulo, DateTime data, decimal totalReceitas,
decimal totalDespesas)
        : base(titulo, data)
    {
        TotalReceitas = totalReceitas;
        TotalDespesas = totalDespesas;
    }

    public override void GerarRelatorio()
    {
        Console.WriteLine($"Relatório Financeiro: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Receitas: {TotalReceitas}, Total de Despesas:
{TotalDespesas}");
    }
}

public class RelatorioDeVendas : Relatorio
{
    public int TotalVendas { get; set; }
    public decimal ReceitaTotal { get; set; }

    public RelatorioDeVendas(string titulo, DateTime data, int totalVendas, decimal
receitaTotal)
        : base(titulo, data)
    {
        TotalVendas = totalVendas;
        ReceitaTotal = receitaTotal;
    }

    public override void GerarRelatorio()
    {
        Console.WriteLine($"Relatório de Vendas: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Vendas: {TotalVendas}, Receita Total:
{ReceitaTotal}");
    }
}

public class RelatorioDeRH : Relatorio
{
    public int TotalFuncionarios { get; set; }
    public int NovasContratacoes { get; set; }

    public RelatorioDeRH(string titulo, DateTime data, int totalFuncionarios, int
novasContratacoes)
        : base(titulo, data)
    {
        TotalFuncionarios = totalFuncionarios;
        NovasContratacoes = novasContratacoes;
    }

    public override void GerarRelatorio()
```

```
    {
        Console.WriteLine($"Relatório de RH: {Titulo} - Data: {Data}");
        Console.WriteLine($"Total de Funcionários: {TotalFuncionarios}, Novas
Contratações: {NovasContratacoes}");
    }
}

// Exemplo de uso
public class Program
{
    public static void Main()
    {
        RelatorioFinanceiro relatorioFinanceiro = new RelatorioFinanceiro("Financeiro
- Janeiro", new DateTime(2024, 1, 31), 50000m, 20000m);
        RelatorioDeVendas relatorioDeVendas = new RelatorioDeVendas("Vendas -
Janeiro", new DateTime(2024, 1, 31), 300, 15000m);
        RelatorioDeRH relatorioDeRH = new RelatorioDeRH("RH - Janeiro", new
DateTime(2024, 1, 31), 50, 5);

        relatorioFinanceiro.GerarRelatorio();
        relatorioDeVendas.GerarRelatorio();
        relatorioDeRH.GerarRelatorio();
    }
}
```

Benefícios do Uso de Classes Abstratas

Vamos refletir sobre os benefícios que essa abordagem traz:

1.**Reutilização de Código**: Reduzimos a duplicação de código comum entre diferentes tipos de relatórios, facilitando a manutenção. 2.**Organização e Estrutura**: A herança permite uma melhor organização do código, refletindo a estrutura lógica do sistema de relatórios. 3.**Especialização**: As classes abstratas facilitam a criação de comportamentos especializados para diferentes tipos de relatórios, tornando o sistema mais flexível.

Desvantagens e Considerações

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

1. **Acoplamento**: A classe derivada depende fortemente da classe base, o que pode dificultar mudanças na classe base.
2. **Sobrescrita de Métodos**: Precisamos ser cuidadosos ao sobrescrever métodos para garantir que o comportamento desejado seja implementado corretamente.
3. **Complexidade**: O uso de classes abstratas pode aumentar a complexidade do sistema, exigindo um bom entendimento dos conceitos de orientação a objetos.

Conclusão

O uso de classes abstratas é uma ferramenta poderosa e prática para o desenvolvimento de sistemas empresariais complexos. Elas permitem que diferentes tipos de objetos compartilhem características comuns, enquanto possuem comportamentos específicos.

Utilizando-as 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 como o uso de classes abstratas ajuda a reduzir a duplicação de código em sistemas empresariais.
2. Descreva um cenário em que o uso de classes abstratas pode ser desfavorável devido à complexidade.

Exercícios Práticos

1. Implemente uma hierarquia de classes para um sistema de gerenciamento de contas bancárias, incluindo classes para ContaCorrente, ContaPoupanca e ContaInvestimento, utilizando uma classe abstrata Conta.
2. Crie uma aplicação que gerencie diferentes tipos de veículos em uma locadora, utilizando uma classe abstrata Veiculo para especializar classes como Carro, Moto e Caminhao.