UNIVERSIDADE LUTERANA DO BRASIL (ULBRA)
ANDRÉ RAMOS KARNAS

PROGRAMAÇÃO ORIENTADA A OBJETOS NO DESENVOLVIMENTO DE UM SISTEMA DE LOJA

TORRES
2024

ANDRÉ RAMOS KARNAS

PROGRAMAÇÃO ORIENTADA A OBJETOS NO DESENVOLVIMENTO DE UM SISTEMA DE LOJA

Trabalho apresentado a disciplina de Laboratório de Programação da Universidade Luterana do Brasil (ULBRA).

TORRES
2024

INTRODUÇÃO

A programação orientada a objetos (POO) é um dos paradigmas de programação mais utilizados na atualidade, permitindo que os desenvolvedores construam sistemas de forma modular e reutilizável. O trabalho descreve a implementação de um sistema de loja que gerencia produtos, clientes e pedidos, utilizando os conceitos fundamentais da POO. A escolha por esse paradigma justifica-se pela sua capacidade de facilitar a manutenção do código e promover a organização das informações, aspectos cruciais para o desenvolvimento de sistemas complexos.

Neste projeto, serão abordados os quatro pilares da POO: encapsulamento, herança, polimorfismo e abstração. Cada um desses conceitos será detalhado com exemplos práticos retirados do código desenvolvido, demonstrando sua aplicabilidade e a contribuição que oferecem para a estrutura e funcionalidade do sistema.

Encapsulamento

O encapsulamento é um dos princípios mais importantes da programação orientada a objetos. Ele se refere à prática de restringir o acesso direto a certos componentes de um objeto, permitindo que apenas métodos da própria classe possam manipular os dados. Essa abordagem protege a integridade do objeto e melhora a manutenção do código, uma vez que mudanças em um componente podem ser feitas sem afetar o restante do sistema.

No sistema de loja, o encapsulamento é implementado nas classes que representam entidades como Cliente, Produto e Pedido. Por exemplo, a classe Cliente possui atributos que são acessíveis apenas através de métodos públicos, garantindo que a manipulação de dados siga regras específicas. O código abaixo ilustra esse conceito:

```
public class Cliente
    {
    public string NomeCliente { get; private set; }
    public string Cpf { get; private set; }
    // Outros atributos...

    public Cliente(string nomeCliente, string cpf)
    {
        NomeCliente = nomeCliente;
        Cpf = cpf;
    }

    public void ExibirInformacoes()
```

```
    {
        Console.WriteLine($"Nome: {NomeCliente}, CPF: {Cpf}");
    }
}
```

Dessa forma, ao restringir o acesso aos atributos NomeCliente e Cpf, garantimos que eles não possam ser alterados diretamente, evitando estados inválidos e mantendo a integridade do objeto.

Herança

A herança é um mecanismo que permite a criação de novas classes a partir de classes existentes, facilitando a reutilização de código. No nosso projeto, utilizamos a herança para criar classes específicas de produtos, como ProdutoDigital e ProdutoFisico, que herdam da classe base Produto. Isso nos permite compartilhar atributos e métodos comuns entre diferentes tipos de produtos.

```
public abstract class Produto
{
    public string Nome { get; set; }
    public string Codigo { get; set; }
    public decimal Preco { get; set; }

    public abstract decimal CalcularPrecoFinal();
```

```
}
```

As classes ProdutoDigital e ProdutoFisico implementam o método CalcularPrecoFinal, cada uma com sua lógica específica. O uso da herança reduz a redundância de código e promove uma organização mais clara:

```
public class ProdutoDigital : Produto
{
    public override decimal CalcularPrecoFinal()
    {
        return Preco - (Preco * 0.10m); // Exemplo de desconto
    }
}

public class ProdutoFisico : Produto
{
    public override decimal CalcularPrecoFinal()
    {
        // Cálculo com impostos e frete
    }
}
```

Essa estrutura hierárquica facilita a expansão do sistema, permitindo a adição de novos tipos de produtos sem modificar o código existente.

Polimorfismo

O polimorfismo permite que um mesmo método tenha diferentes comportamentos dependendo do contexto em que é utilizado. Esse conceito é fundamental para a flexibilidade do sistema, permitindo que coleções de objetos de diferentes classes compartilhem a mesma interface ou classe base.

No nosso sistema, o método CalcularPrecoFinal é um exemplo claro de polimorfismo. Embora o método tenha o mesmo nome, sua implementação varia conforme a classe do produto. Isso significa que podemos chamar o método em uma lista de produtos e o sistema determinará automaticamente qual versão deve ser executada:

```
public decimal CalcularTotal(List<Produto> produtos)
{
    decimal total = 0;
    foreach (var produto in produtos)
    {
        total += produto.CalcularPrecoFinal();
    }
    return total;
}
```

Esse exemplo ilustra como o polimorfismo simplifica a lógica de cálculo, permitindo que diferentes tipos de produtos sejam tratados de forma uniforme.

Abstração

A abstração é o processo de ocultar a complexidade de um sistema, expondo apenas as informações relevantes. No nosso projeto, a classe abstrata Produto representa essa ideia ao definir uma interface comum para todos os tipos de produtos, sem especificar os detalhes de cada implementação.

```
public abstract class Produto
{
    public abstract decimal CalcularPrecoFinal();
}
```

Dessa maneira, os desenvolvedores podem trabalhar com objetos do tipo Produto sem se preocupar com as particularidades de cada subclasse, facilitando a implementação de novas funcionalidades e a manutenção do código. A abstração promove um design mais limpo e intuitivo, permitindo que novas classes sejam adicionadas facilmente.

CONCLUSÃO

O desenvolvimento do sistema de loja utilizando a programação orientada a objetos proporcionou um aprendizado significativo sobre os conceitos fundamentais desse paradigma. A aplicação de encapsulamento, herança, polimorfismo e abstração não apenas facilitou a organização e a manutenção do código, mas também possibilitou a criação de um sistema flexível e extensível.

Este projeto demonstrou a importância da POO como uma ferramenta eficaz para o desenvolvimento de software. A experiência adquirida com a implementação prática desses conceitos reforçou a compreensão de como eles podem ser utilizados para resolver problemas complexos de forma estruturada e eficiente.

REFERÊNCIAS

https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos

https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264