Pedro Inácio Rodrigues Pontes

Design Patterns: Sistema Eccomerce

Belo Horizonte, Brasil

2025

1 Introdução

O objetivo do presente trabalho foi a implementação de um sistema de e-commerce simplificado em C que demonstrasse a aplicação prática de cinco padrões de projeto fundamentais da engenharia de software. O sistema original precisava gerenciar múltiplas funcionalidades essenciais para o comércio eletrônico - incluindo criação de produtos, processamento de pagamentos, notificações e configurações globais -, onde o desafio principal estava em criar uma arquitetura flexível e de fácil manutenção. O desafio consistia em transformar requisitos funcionais complexos em uma implementação modular utilizando padrões de projeto (*Design Patterns*), com o objetivo de criar componentes independentes e reutilizáveis. Para isso, foi necessário implementar cinco padrões específicos: *Singleton* para gerenciamento de configurações, *Factory Method* para criação de produtos, *Observer* para sistema de notificações, *Strategy* para métodos de pagamento e *Decorator* para funcionalidades extras dos produtos. A principal motivação para esta abordagem era resolver problemas comuns em sistemas de e-commerce, como rigidez arquitetural, dificuldade de manutenção e baixa capacidade de extensão, criando um sistema onde novas funcionalidades pudessem ser adicionadas sem modificar o código existente.

2 Desenvolvimento

2.1 Arquitetura Base do Sistema

O sistema de e-commerce foi desenvolvido seguindo uma arquitetura modular onde cada padrão de projeto resolve um problema específico. A implementação base conta com classes abstratas e interfaces que definem contratos claros para extensibilidade futura, permitindo que novas funcionalidades sejam adicionadas sem modificar o código existente.

2.1.1 Estrutura Fundamental dos Produtos

A hierarquia de produtos foi implementada através de uma classe abstrata base que define o comportamento comum a todos os produtos:

```
public abstract class Produto
{
public string Nome { get; set; }
public decimal Preco { get; set; }
public abstract string ObterCategoria();
public abstract decimal CalcularFrete();
}
```

Cada tipo de produto concreto implementa suas próprias regras de negócio. A classe *Eletronico* define sua categoria como "Eletrônicos"e calcula o frete como 5

2.2 Implementação dos Padrões de Projeto

2.2.1 Padrão Singleton - Gerenciamento de Configurações

O padrão Singleton foi implementado para garantir uma única instância do gerenciador de configurações em todo o sistema. A implementação utiliza *double-checked locking* para thread-safety:

```
public sealed class GerenciadorConfiguracao
{
private static GerenciadorConfiguracao _instancia;
private static readonly object _bloqueio = new object();
private GerenciadorConfiguracao() { }

public static GerenciadorConfiguracao Instancia
{
    get
    {
        if (_instancia == null)
        {
            lock (_bloqueio)
            {
                if (_instancia == null)
                    _instancia = new GerenciadorConfiguracao();
            }
        }
        return _instancia;
    }
}

public string ConexaoBancoDados { get; set; } = "ConexaoPadrao";
public decimal TaxaImposto { get; set; } = 0.08m;
}
```

2.2.2 Padrão Factory Method - Criação de Produtos

O Factory Method foi implementado para encapsular a lógica de criação de diferentes tipos de produtos. Cada fábrica concreta é responsável por criar produtos específicos com suas características particulares:

```
public class FabricaEletronicos : FabricaProduto
{
```

```
public override Produto CriarProduto(string nome, decimal preco)
{
return new Eletronico { Nome = nome, Preco = preco };
}
}
public class FabricaLivro : FabricaProduto
{
public override Produto CriarProduto(string nome, decimal preco)
{
return CriarProduto(nome, preco, "Autor Desconhecido", 100);
}
public Produto CriarProduto(string nome, decimal preco, string autor, int
{
    return new Livro
    {
        Nome = nome,
        Preco = preco,
        Autor = autor,
        NumeroPaginas = numeroPaginas,
    };
}
}
```

2.2.3 Padrão Observer - Sistema de Notificações

O padrão Observer foi implementado para criar um sistema de notificações reativo que informa automaticamente sobre mudanças no status dos pedidos:

```
public class Pedido : IObservavel<IObservadorPedido>
{
private List<IObservadorPedido> _observadores = new List<IObservadorPedid
private string _status;
public string Status
{
    get => _status;
    set
    {
        _status = value;
        NotificarObservadores();
    }
```

```
}

public void Inscrever(IObservadorPedido observador)
{
    _observadores.Add(observador);
}

private void NotificarObservadores()
{
    foreach (var observador in _observadores)
    {
        observador.AoMudarStatusPedido(this, _status);
    }
}
}
```

2.2.4 Padrão Strategy - Métodos de Pagamento

O padrão Strategy foi implementado para suportar múltiplas formas de pagamento com validações específicas para cada método. Cada estratégia implementa sua própria lógica de processamento:

```
public class PagamentoCartaoCredito : IEstrategiaPagamento
{
public string NumeroCartao { get; set; }
public string NomeTitular { get; set; }
public bool ProcessarPagamento(decimal valor)
{
    return valor > 0 && valor < 5000;
}

public string ObterDetalhesPagamento()
{
    string Ultimos4Digitos = NumeroCartao.Substring(NumeroCartao.Length -
    return $" ltimos 4 D gitos Cart o de Cr dito: {Ultimos4Digitos}";
}
}
```

2.2.5 Padrão Decorator - Funcionalidades Extras

O padrão Decorator foi implementado para permitir a adição dinâmica de funcionalidades extras aos produtos sem modificar suas classes base. Cada decorador adiciona comportamentos específicos:

```
public class DecoradorGarantia : DecoradorProduto
{
private int _mesesGarantia;
public DecoradorGarantia(Produto produto, int mesesGarantia) : base(produ
{
    _mesesGarantia = mesesGarantia;
    Preco = produto.Preco + (mesesGarantia * 10); // R$10 por m s
}

public override string ObterCategoria() => base.ObterCategoria() + $"␣Ga
}
```

2.3 Integração e Fluxo Principal

O sistema integra todos os padrões através da classe principal *SistemaECommerce*, que demonstra o fluxo completo desde a configuração inicial até o processamento final do pagamento. A implementação garante que cada componente funcione independentemente, mas se integre harmoniosamente com os demais, proporcionando uma experiência de uso fluida e extensível.

3 Resultados

```
Configuração: Conexão=ServidorLocal;Banco=EcommerceDB, Imposto=10.00 %
[EMAIL] O pedido mudou para o status: Processando
[SMS] O pedido mudou para o status: Processando
[EMAIL] O pedido mudou para o status: Enviado
[SMS] O pedido mudou para o status: Enviado
[EMAIL] O pedido mudou para o status: Entregue
[SMS] O pedido mudou para o status: Entregue
1134.99
True
Pagamento Cartão: Aprovado - Últimos 4 Dígitos Cartao de Crédito: 5678
Pagamento PayPal: Aprovado - Pagamento com PayPal
 Email: usuario@paypal.com
```

```
Pagamento Pix: Aprovado - PIX: 99999999999
```

4 Conclusão

A implementação do sistema de e-commerce com padrões de projeto foi realizada com sucesso. O sistema demonstrou como a aplicação adequada de design patterns pode resolver problemas arquiteturais complexos, proporcionando uma base sólida para futuras extensões e manutenções. Cada um dos cinco padrões implementados atendeu aos requisitos específicos propostos, criando uma arquitetura modular e flexível. Os resultados demonstraram que os padrões de projeto são mais eficazes quando aplicados em conjunto, formando uma arquitetura coesa. O padrão Singleton garantiu o gerenciamento centralizado de configurações, o Factory Method proporcionou flexibilidade na criação de produtos, o Observer implementou um sistema de notificações reativo, o Strategy permitiu múltiplos métodos de pagamento, e o Decorator adicionou funcionalidades extras de forma dinâmica. A integração entre os padrões mostrou-se harmoniosa, com cada componente funcionando independentemente mas contribuindo para o sistema como um todo. A implementação do fluxo principal na classe *SistemaECommerce* demonstrou como produtos podem ser criados via Factory, decorados com funcionalidades extras, processados através de diferentes estratégias de pagamento, e ter suas mudanças de status notificadas automaticamente aos observadores interessados. A solução implementada para a separação de responsabilidades através dos padrões foi fundamental para criar um sistema extensível e de fácil manutenção. Esta abordagem permitiu que cada funcionalidade fosse implementada de forma independente, facilitando testes unitários e futuras modificações, o que é essencial para a evolução contínua de sistemas comerciais. A arquitetura baseada em padrões de projeto proporcionou benefícios substanciais em termos de organização do código, reutilização de componentes e facilidade de extensão. O sistema se mostrou preparado para crescer e adaptar-se a novos requisitos, onde novas formas de pagamento, tipos de produtos ou métodos de notificação podem ser adicionados sem impactar o código existente. Os resultados confirmam que, embora os padrões de projeto introduzam complexidade inicial, eles são viáveis e vantajosos para aplicações comerciais que precisam de flexibilidade e manutenibilidade a longo prazo.