Análise Comparativa de Padrões: Distinções e Aplicações dos Padrões Comportamentais, Criacionais e Estruturais

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/analise-comparativa-de-padroes-distincoes-e-aplicacoes-dos-padroes-comportamentais-criacionais-e-estruturais
set 16 2024

Abstract:
Análise comparativa entre os padrões comportamentais, criacionais e estruturais, destacando suas distinções e aplicações no desenvolvimento de software. Os padrões comportamentais são explorados por sua capacidade de facilitar a interação entre objetos. Já os padrões criacionais são avaliados quanto à sua habilidade de abstrair o processo de criação de objetos. Por fim, os padrões estruturais são analisados pela forma como organizam classes e objetos em sistemas maiores e mais complexos.

Key words:
análise comparativa, padrões comportamentais, padrões criacionais, padrões estruturais, distinções, aplicações, design, estrutura organizacional, metodologias de análise, criação, inovação, modelagem, sistemas, comparação.

Introdução aos Padrões de Design

Os padrões de design são soluções reutilizáveis para problemas recorrentes no desenvolvimento de software. Eles oferecem uma forma padronizada de resolver desafios arquiteturais e de codificação, promovendo a manutenção e escalabilidade do código. Os padrões são comumente classificados em três grandes categorias: comportamentais, criacionais e estruturais. Cada categoria aborda diferentes aspectos do desenvolvimento de software, desde a criação de objetos até a interação entre diferentes componentes. O uso apropriado desses padrões pode resultar em sistemas mais flexíveis, reutilizáveis e fáceis de manter. Neste artigo, iremos detalhar as características e aplicações de cada um desses tipos, com exemplos práticos e estudos de caso que demonstram sua importância na prática do desenvolvimento moderno (Gamma et al., 1994).

Padrões Criacionais

Os padrões criacionais estão diretamente relacionados com o processo de instanciação de objetos. Eles ajudam a abstrair a forma como os objetos são criados, permitindo que o código seja mais flexível e menos acoplado a implementações específicas. Entre os padrões mais conhecidos dessa categoria, temos o Factory Method, Abstract Factory, Singleton, Builder e Prototype. Esses padrões são frequentemente usados em arquiteturas que exigem independência na criação de objetos, como em projetos que implementam princípios de inversão de controle ou injeção de dependência (Freeman et al., 2020).

Exemplo: Factory Method

O padrão Factory Method define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar. Ele promove o princípio de responsabilidade única ao permitir que a lógica de criação de objetos seja centralizada em uma única classe ou método. Abaixo, um exemplo de Factory Method implementado em C#:

```
public abstract class Document
{
    public abstract void Print();
}

public class Report : Document
{
    public override void Print() => Console.WriteLine("Printing Report...");
}

public class Invoice : Document
{
    public override void Print() => Console.WriteLine("Printing Invoice...");
}

public abstract class DocumentCreator
{
    public abstract Document CreateDocument();
}

public class ReportCreator : DocumentCreator
{
    public override Document CreateDocument() => new Report();
}

public class InvoiceCreator : DocumentCreator
{
    public override Document CreateDocument() => new Invoice();
}

public class Program
{
    public static void Main()
    {
        DocumentCreator creator = new ReportCreator();
        Document document = creator.CreateDocument();
        document.Print();
    }
}
```

Um caso de uso recente desse padrão é em sistemas de microserviços, onde a criação de diferentes tipos de serviços pode ser feita de forma mais modular e flexível através de fábricas, permitindo que novas funcionalidades sejam adicionadas sem modificar o código existente (Gamma et al., 1994). Além disso, o Factory Method pode ser combinado com outros padrões, como o Abstract Factory, para criar famílias de objetos relacionados.

Outro padrão criacional muito utilizado em sistemas distribuídos é o Singleton. O Singleton garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a essa instância. Isso é útil em cenários onde é necessário ter uma única fonte de verdade, como em configurações globais de uma aplicação ou em gerenciadores de cache compartilhados entre diferentes componentes de um sistema (Gamma et al., 1994).

Exemplo: Singleton

O exemplo abaixo ilustra uma implementação típica do padrão Singleton em C#. Esse padrão é bastante utilizado em aplicações de grande escala, como serviços em nuvem, onde é crucial garantir que recursos compartilhados, como configurações e cache, sejam acessados de maneira consistente por todos os componentes do sistema.

```
public class ConfigurationManager
{
    private static ConfigurationManager _instance;
    private static readonly object _lock = new object();

    private ConfigurationManager() { }

    public static ConfigurationManager Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new ConfigurationManager();
                    }
                }
            }
            return _instance;
        }
    }

    public string GetSetting(string key) => "SomeValue";
}

public class Program
{
    public static void Main()
    {
        var config = ConfigurationManager.Instance;
        Console.WriteLine(config.GetSetting("MySetting"));
    }
}
```

Padrões Estruturais

Os padrões estruturais dizem respeito à composição de classes e objetos. Eles ajudam a garantir que diferentes partes de um sistema trabalhem juntas de maneira eficaz e eficiente. Entre os padrões estruturais mais comuns estão o Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy (Freeman et al., 2020).

Exemplo: Adapter

```
public interface ITarget
{
    string Request();
```

```
}

public class Adaptee
{
    public string SpecificRequest() => "Specific Request";
}

public class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        _adaptee = adaptee;
    }

    public string Request() => _adaptee.SpecificRequest();
}

public class Program
{
    public static void Main()
    {
        Adaptee adaptee = new Adaptee();
        ITarget target = new Adapter(adaptee);
        Console.WriteLine(target.Request());
    }
}
```

Recentemente, o padrão Adapter tem sido utilizado em aplicações de comércio eletrônico, especialmente em integrações de sistemas de pagamento. Plataformas que oferecem múltiplas formas de pagamento, como PayPal, Stripe e PagSeguro, podem usar o Adapter para unificar a forma como esses sistemas são integrados, oferecendo uma interface única para a aplicação principal (Freeman et al., 2020).

Padrões Comportamentais

Os padrões comportamentais lidam com a comunicação e a responsabilidade entre os objetos. Eles ajudam a definir como os objetos interagem entre si e como as responsabilidades são distribuídas no sistema. Alguns dos padrões mais comuns nessa categoria são: Observer, Strategy, Command, Mediator, Memento e Chain of Responsibility (Gamma et al., 1994).

Exemplo: Observer

```
using System;
using System.Collections.Generic;

public interface IObserver
{
    void Update(string message);
}

public class Subject
```

```
{
    private readonly List _observers = new List();

    public void Attach(IObserver observer) => _observers.Add(observer);

    public void Detach(IObserver observer) => _observers.Remove(observer);

    protected void Notify(string message)
    {
        foreach (var observer in _observers)
        {
            observer.Update(message);
        }
    }
}

public class ConcreteSubject : Subject
{
    private string _state;

    public string State
    {
        get => _state;
        set
        {
            _state = value;
            Notify(_state);
        }
    }
}

public class ConcreteObserver : IObserver
{
    private readonly string _name;

    public ConcreteObserver(string name)
    {
        _name = name;
    }

    public void Update(string message) => Console.WriteLine($"{_name} received:
{message}");
}

public class Program
{
    public static void Main()
    {
        ConcreteSubject subject = new ConcreteSubject();
        ConcreteObserver observer1 = new ConcreteObserver("Observer 1");
        ConcreteObserver observer2 = new ConcreteObserver("Observer 2");
```

```
        subject.Attach(observer1);
        subject.Attach(observer2);

        subject.State = "New State";
    }
}
```

Conclusão

Os padrões de design fornecem aos desenvolvedores uma linguagem comum para resolver problemas recorrentes de maneira eficaz e reutilizável. Ao aplicar padrões como Factory Method, Singleton, Adapter e Observer, é possível criar sistemas que são mais fáceis de manter, estender e modificar. Cada padrão tem seu lugar e momento apropriado para ser utilizado, e entender essas situações é fundamental para aplicar os padrões de forma eficaz. Compreender e dominar esses conceitos pode melhorar significativamente a qualidade e a flexibilidade do software, além de promover um desenvolvimento mais ágil e sustentável.

Referências

GAMMA, Erich et al. Design patterns: elements of reusable object-oriented software. 1. ed. Addison-Wesley, 1994.
FREEMAN, Eric et al. Head First Design Patterns. 2. ed. O'Reilly Media, 2020.

Nagib é Professor Universitário e Tech Manager. Possui uma trajetória de conquistas em certificações técnicas e ágeis, incluindo MCSD, MCSA e PSM1. PG em Gestão de TI pelo SENAC e MBA em Tecnologia de Software pela USP, Nagib cursou programas de extensão do MIT e Universidade de Chicago. Outras conquistas incluem a autoria de um artigo sobre chatbots, revisado por pares e apresentado na Universidade de Barcelona.