Comparative Analysis of Patterns: Distinctions and Applications of Behavioral, Creational, and Structural Patterns

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/fb7252
set 16 2024

Abstract:
Comparative analysis between behavioral, creational, and structural patterns, highlighting their distinctions and applications in software development. Behavioral patterns are explored for their ability to facilitate interaction between objects. Creational patterns are evaluated for their capability to abstract the object creation process. Finally, structural patterns are analyzed by the way they organize classes and objects in larger and more complex systems.

Key words:
comparative analysis, behavioral patterns, creative patterns, structural patterns, distinctions, applications, design, organizational structure, analysis methodologies, creation, innovation, modeling, systems, comparison.

Introduction to Design Patterns

Design patterns are reusable solutions to recurring problems in software development. They provide a standardized way to address architectural and coding challenges, promoting code maintainability and scalability. Patterns are commonly classified into three major categories: behavioral, creational, and structural. Each category addresses different aspects of software development, from object creation to interaction between different components. The appropriate use of these patterns can result in more flexible, reusable, and maintainable systems. In this article, we will detail the characteristics and applications of each of these types, with practical examples and case studies that demonstrate their importance in modern development practice (Gamma et al., 1994).

Creational Patterns

Creational patterns are directly related to the process of object instantiation. They help abstract the way objects are created, allowing the code to be more flexible and less coupled to specific implementations. Among the most well-known patterns in this category are the Factory Method, Abstract Factory, Singleton, Builder, and Prototype. These patterns are often used in architectures that require independence in object creation, such as projects that implement principles of inversion of control or dependency injection (Freeman et al., 2020).

Example: Factory Method

The Factory Method pattern defines an interface for creating an object but allows subclasses to decide which class to instantiate. It promotes the single responsibility principle by allowing the logic for object creation to be centralized in a single class or method. Below is an example of the Factory Method implemented in 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();
    }
}
```

A recent use case of this pattern is in microservices systems, where the creation of different types of services can be done in a more modular and flexible way through factories, allowing new functionalities to be added without modifying existing code (Gamma et al., 1994). Furthermore, the Factory Method can be combined with other patterns, such as the Abstract Factory, to create families of related objects.

Another widely used creational pattern in distributed systems is the Singleton. The Singleton ensures that a class has only one instance and provides a global access point to that instance. This is useful in scenarios where it is necessary to have a single source of truth, such as in global application configurations or in cache managers shared among different components of a system (Gamma et al., 1994).

Example: Singleton

The example below illustrates a typical implementation of the Singleton pattern in C#. This pattern is widely used in large-scale applications, such as cloud services, where it is crucial to ensure that shared resources, such as configurations and cache, are accessed consistently by all components of the system.

```
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"));
    }
}
```

Structural Patterns

Structural patterns deal with the composition of classes and objects. They help ensure that different parts of a system work together effectively and efficiently. Among the most common structural patterns are the Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy (Freeman et al., 2020).

Example: 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());
    }
}
```

Recently, the Adapter pattern has been used in e-commerce applications, especially in payment system integrations. Platforms that offer multiple payment methods, such as PayPal, Stripe, and PagSeguro, can use the Adapter to unify the way these systems are integrated, providing a single interface for the main application (Freeman et al., 2020).

Behavioral Patterns

Behavioral patterns deal with communication and responsibility between objects. They help define how objects interact with each other and how responsibilities are distributed within the system. Some of the most common patterns in this category are: Observer, Strategy, Command, Mediator, Memento, and Chain of Responsibility (Gamma et al., 1994).

Example: 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";
```

```
    }
}
```

Conclusion

Design patterns provide developers with a common language to solve recurring problems effectively and reusably. By applying patterns such as Factory Method, Singleton, Adapter, and Observer, it is possible to create systems that are easier to maintain, extend, and modify. Each pattern has its appropriate place and timing for use, and understanding these situations is essential for applying patterns effectively. Understanding and mastering these concepts can significantly improve the quality and flexibility of software, as well as promote more agile and sustainable development.

References

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

Nagib is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He has a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs at MIT and the University of Chicago. Other achievements include the authorship of a peer-reviewed article on chatbots, presented at the University of Barcelona.