Artículo - Patrones de diseño del software

Presentado por:

Jeffry Varela Barrantes

Carlos Villafuerte Díaz

Duvan Vásquez López

Universidad Cenfotec

Fecha: 1 de Agosto de 2025

Table of contents

Objetivo general y específicos

Objetivo General: Comprender y comunicar conceptos fundamentales del diseño de software, enfocándose en el análisis y descripción de patrones, su propósito y su aplicación, como parte del proceso de diseño orientado a la calidad del software.

Objetivos Específicos:

- Identificar y describir patrones de diseño comunes utilizados en el desarrollo de software.
- Analizar el propósito, los beneficios y el contexto de uso de diferentes patrones de diseño.
- Investigar y comunicar el estado del arte sobre patrones de diseño, utilizando fuentes actualizadas y confiables.
- Estructurar un documento técnico claro y coherente, empleando herramientas como Quarto para su presentación formal.

Estado del arte

Conceptos clave de patrones de diseño

Los patrones de diseño de software son soluciones generales y reutilizables para problemas comunes que ocurren en el proceso de diseño de software. No son fragmentos de código listos para usar, sino pautas o descripciones estructurales de una solución óptima, obtenidas por experiencia acumulada. Por ejemplo, (Soler, 2024) los define como “soluciones generales y reutilizables que se aplican a problemas comunes en el diseño de software”. Estas soluciones estandarizadas buscan promover la modularidad, la escalabilidad y el mantenimiento del sistema. La aplicación de patrones “ahorra tiempo y esfuerzo” en el desarrollo, facilita la comunicación entre desarrolladores (al disponer de un lenguaje común) y contribuye a que el código sea más mantenible y escalable. Los patrones se basan en principios de diseño (como SOLID) y en la experiencia histórica de la ingeniería de software. Su uso sistemático permite evitar reinventar la rueda, ya que ofrecen estrategias que han sido comprobadas en múltiples proyectos. En palabras de Refactoring.Guru, “los patrones son un juego de herramientas que brindan soluciones a problemas habituales en el diseño de software” y definen un lenguaje compartido que mejora la eficiencia comunicativa del equipo. Sin embargo, es importante recordar que los patrones no son algoritmos específicos, sino esquemas de solución que deben adaptarse al contexto de cada proyecto.

Los patrones de diseño fueron popularizados por (Gamma et al., 1995), quienes definieron 23 patrones clásicos en el contexto de la programación orientada a objetos, agrupándolos en patrones creacionales, estructurales

y de comportamiento.

Clasificación de patrones de diseño

La literatura coincide en clasificar los patrones de diseño en tres categorías generales, según el problema que abordan:

- Patrones creacionales: Se enfocan en el proceso de creación de objetos, encapsulando mecanismos de instanciación complejos. Promueven la flexibilidad y la reutilización del código al ocultar la lógica de creación. Ejemplos típicos son ***Abstract Factory, Builder y Singleton***. Por ejemplo, (Soler, 2024) menciona que Abstract Factory permite crear familias de objetos sin especificar las clases concretas, mientras que Builder facilita la construcción de objetos complejos paso a paso. El patrón Singleton garantiza que una clase tenga una única instancia global, controlando su acceso. Estas técnicas ayudan a desacoplar el código de cómo se crean sus objetos internos.

-Patrones estructurales: Definen cómo componer clases y objetos para formar estructuras más grandes y eficaces. Buscan facilitar la extensión y mantener la eficiencia al enlazar componentes. Ejemplos conocidos son ***Adapter, Facade y Proxy***. Por ejemplo, el patrón Adapter permite que interfaces incompatibles cooperen, y el patrón Facade proporciona una interfaz simplificada para un conjunto complejo de clases. De esta forma se logra organizar el sistema en capas de abstracción o envolturas que ocultan la complejidad del subsistema.

-Patrones de comportamiento: Se centran en la interacción entre objetos y la distribución de responsabilidades. Estos patrones optimizan la

forma en que los objetos se comunican y colaboran. Ejemplos representativos incluyen ***Observer, Strategy y Command***. Por ejemplo, Observer permite notificar a múltiples objetos ante cambios en otro objeto observado, y Strategy define una familia de algoritmos intercambiables dentro de un mismo objeto. En el contexto de C#, los eventos y delegados suelen implementar el patrón Observer (por ejemplo, ***INotifyPropertyChanged*** en WPF), mientras que mecanismos como interfaces y funciones anónimas facilitan estrategias intercambiables (Strategy).

Estos grupos de patrones provienen del catálogo clásico de (Gamma et al. 1995) –los "Gang of Four"– que recopiló 23 patrones de diseño ampliamente reconocidos. Desde entonces, dichos patrones se han mantenido vigentes y se siguen enseñando como principios fundamentales del diseño orientado a objetos. Ejemplos de patrones creacionales, estructurales y de comportamiento: Los patrones creacionales más comunes incluyen Factory Method, Abstract Factory, Builder, Prototype y Singleton. Por ejemplo, el patrón Factory Method delega la creación de objetos a subclases concretas (reduciendo dependencias directas), y Abstract Factory agrupa fábricas de familias relacionadas. En C#, la clase StringBuilder implementa el patrón Builder al proporcionar pasos seguros para construir cadenas complejas. En el ámbito estructural, destacan Adapter, Bridge, Composite, Decorator, Facade, Flyweight y Proxy. El patrón Composite permite tratar grupos de objetos como elementos individuales, mientras que Proxy controla el acceso a un objeto real. En cuanto a comportamiento, además de Observer, Strategy y Command, existen Iterator, Mediator, Memento, State, Template Method, Visitor, entre otros. Por ejemplo, Observer en C# se ve en los eventos de Windows Forms o WPF, y Strategy puede verse al pasar distintas

implementaciones de algoritmos (por ejemplo, clasificadores) a través de interfaces.

Patrones GOF comunes (patrones de diseño clásicos)

El catálogo GOF sigue siendo una referencia obligada. A continuación se destacan algunos patrones clave y su aplicación:

-Factory Method (Patrón Factoría). Define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase concreta instanciar. Esto desacopla el código cliente de las clases específicas. Por ejemplo, en C# un método factory puede devolver distintas clases derivadas según parámetros de entrada. Figura: Diagrama UML del patrón Factory Method. En el diagrama UML anterior se observa cómo una clase creadora (Creator) delega la instanciación de Product a métodos concretos (FactoryMethod) que devuelven objetos concretos (ConcreteProduct). Este esquema facilita la extensión para nuevos tipos de productos sin modificar el código cliente. En la plataforma .NET, patrones similares se usan en abstracciones como DbProviderFactory o en la fábrica de objetos de HttpClient.

-Singleton (Patrón Único). Asegura que una clase tenga exactamente una instancia y proporciona un punto de acceso global. En C#, esto se implementa mediante una clase que controla la creación de la instancia (por ejemplo, usando propiedades estáticas y constructor privado). El siguiente diagrama UML ilustra el patrón: Figura: Diagrama UML del patrón Singleton. En C# comúnmente se emplea el modificador static y un constructor privado para lograrlo, o bien la propiedad Lazy para inicialización perezosa, lo que garantiza creación única incluso en entornos multihilo. El patrón Sin-

gleton se usa, por ejemplo, para administrar conexiones compartidas (un único pool de conexiones a base de datos) o configuraciones globales de la aplicación.

Observer (Observador). Define una relación uno a muchos entre objetos, de modo que al cambiar el estado de uno, se notifica automáticamente a los demás suscritos. Esto implementa mecanismos de suscripción/publicación. En C#, el sistema de eventos (delegados EventHandler) es una implementación típica de Observer. (Refactoring.Guru, n.d.) describe este patrón como "un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento". Se emplea, por ejemplo, en interfaces gráficas (un modelo notifica a varias vistas) o en comunicaciones de mensajería interna.

Strategy (Estrategia). Permite definir una familia de algoritmos intercambiables y encapsularlos en clases separadas, de forma que el objeto que los utiliza pueda cambiar de algoritmo en tiempo de ejecución. Esto se logra típicamente mediante interfaces. En C#, se puede implementar pasando distintas implementaciones de una interfaz o incluso funciones (delegados) al objeto que las ejecuta. (Soler, 2024) menciona Strategy como patrón de comportamiento, e ilustra su uso para algoritmos alternativos. Es útil, por ejemplo, en sistemas de pago donde la estrategia de cálculo de comisión puede variar sin modificar el cliente. Otros patrones como Decorator, Facade o Template Method también se aplican frecuentemente en C#. Por ejemplo, Decorator permite añadir responsabilidades a objetos existentes al envolverlos, lo que puede verse en clases que añaden funcionalidad adicional a colecciones o flujos de datos.

Ejemplos comunes en C#

Factory Method

```
public abstract class DocumentFactory
{
    public abstract IDocument CreateDocument();
}

public class PdfDocumentFactory : DocumentFactory
{
    public override IDocument CreateDocument() => new PdfDocument();
}
```

Singleton

```
public sealed class ConfigurationManager
{
    private static readonly Lazy<ConfigurationManager> instance = new(() => new Confi
    public static ConfigurationManager Instance => instance.Value;
    private ConfigurationManager() {}
}
```

Observer (eventos en C#)

```
public class Stock
{
    public event Action<decimal> PriceChanged;
```

```
    private decimal price;
    public decimal Price
    {
        get => price;
        set {
            price = value;
            PriceChanged?.Invoke(price);
        }
    }
}
```

Strategy: ejemplo completo

```
public interface ICommissionStrategy
{
    decimal CalculateCommission(decimal amount);
}

public class StandardCommission : ICommissionStrategy
{
    public decimal CalculateCommission(decimal amount) => amount * 0.05m;
}

public class PremiumCommission : ICommissionStrategy
{
    public decimal CalculateCommission(decimal amount) => amount * 0.025m;
}
```

```
public class CorporateCommission : ICommissionStrategy
{
    public decimal CalculateCommission(decimal amount) => 0m;
}

public class TransactionProcessor
{
    private readonly ICommissionStrategy _strategy;

    public TransactionProcessor(ICommissionStrategy strategy)
    {
        _strategy = strategy;
    }

    public void Process(decimal amount)
    {
        var commission = _strategy.CalculateCommission(amount);
        Console.WriteLine($"Processing ${amount} with commission: ${commission}");
    }
}

// Ejemplo de uso
var processor = new TransactionProcessor(new PremiumCommission());
processor.Process(1000); // Output: Processing $1000 with commission: $25
```

Patrones POSA (Pattern-Oriented Software Architecture) y ejemplos estructurales

Más allá de los patrones GOF, los patrones de arquitectura (o POSA, Pattern-Oriented Software Architecture) abordan la estructura global del sistema.

Un patrón arquitectónico es “una solución general y reutilizable a un problema común en la arquitectura de software dentro de un contexto dado”. Tienen un alcance mayor que los de diseño de bajo nivel, definiendo cómo se organizan los subsistemas de alto nivel. (Buschmann et al., 1996) introdujeron patrones arquitectónicos como Microkernel, Layers y Pipes and Filters, aplicables a sistemas complejos y distribuidos.

-Patrón en capas (Layered Architecture). Una de las arquitecturas más comunes es la de múltiples capas. Por ejemplo, se suele dividir el sistema en capa de presentación (interfaz), capa de lógica de negocio, capa de servicios/aplicación y capa de acceso a datos. El artículo de Ccori (Ccori Huaman, 2018) ilustra esto indicando cuatro capas típicas: presentación, aplicación, negocio y persistencia. Esta separación mejora la modularidad y facilita tareas como pruebas unitarias, ya que cada capa puede desarrollarse y probarse de forma independiente.

-Modelo-Vista-Controlador (MVC). Es un patrón arquitectónico que divide la aplicación en tres componentes: Modelo (datos y lógica), Vista (presentación) y Controlador (gestión de la interacción del usuario). Este desacoplamiento permite cambios independientes: por ejemplo, modificar la interfaz de usuario (Vista) sin alterar la lógica del Modelo. La figura siguiente ilustra el flujo básico de MVC: Figura: Arquitectura Modelo-Vista-

Controlador (MVC). En C#, MVC se emplea ampliamente en frameworks web (ASP.NET MVC) y de escritorio. Existen variantes como MVVM en WPF que derivan de este concepto. El patrón MVC mejora la mantenibilidad de aplicaciones interactivas al separar las responsabilidades internas de la forma en que se presenta y acepta la información. Otros patrones arquitectónicos comunes: Entre ellos figuran Cliente-Servidor (un servidor atiende a múltiples clientes).

-Maestro-Esclavo (un coordinador delega tareas a subsistemas) o Pipeline (canal de procesamiento). El patrón pipe-filter (filtros y tuberías) organiza el flujo de datos en etapas secuenciales (muy usado en compiladores y pipelines de procesamiento de datos).

-El patrón Broker (Agente intermediario). Gestiona la comunicación entre componentes distribuidos, mediando peticiones (por ejemplo, middleware de mensajería como RabbitMQ). En la era moderna también es relevante mencionar la arquitectura de microservicios, que puede considerarse un estilo donde cada servicio actúa como un “cliente-servidor” independiente comunicándose mediante APIs ligeras.

La investigación reciente destaca patrones específicos para microservicios: uso de un “registro de servicios” (por ejemplo, Netflix Eureka) para el descubrimiento dinámico de instancias, pasarela API centralizada (API Gateway) y el uso de circuit breaker (e.g. Hystrix) para tolerancia a fallos.

Microservicios y CQRS

La arquitectura de microservicios ha emergido como una solución efectiva para abordar los desafíos de escalabilidad, mantenimiento y despliegue

en aplicaciones modernas distribuidas. A diferencia de las arquitecturas monolíticas tradicionales, en las que toda la lógica de negocio se encuentra contenida en una única unidad desplegable, los microservicios promueven la descomposición del sistema en servicios independientes, cada uno con su propia responsabilidad y base de datos. Esta separación permite desarrollar, desplegar y escalar cada componente de forma autónoma (Newman, 2015).

Uno de los retos principales en arquitecturas distribuidas es la coherencia de los datos y el diseño de la comunicación entre servicios. Para abordar estos problemas, se han propuesto diversos patrones arquitectónicos, entre los que destaca CQRS (Command and Query Responsibility Segregation). Este patrón sugiere una separación explícita entre los modelos que procesan comandos (acciones que modifican el estado del sistema) y aquellos que resuelven consultas (lecturas del sistema), lo cual facilita el diseño de sistemas altamente escalables y mantenibles (Fowler, 2011).

La combinación de CQRS con Event Sourcing ha demostrado ser especialmente potente en sistemas distribuidos. Event Sourcing consiste en almacenar el estado del sistema como una secuencia de eventos inmutables, permitiendo reconstruir el estado a partir de estos eventos y facilitando capacidades como el auditado, el debugging y la replicación (Vernon, 2013). Aunque esta aproximación agrega complejidad, ha sido utilizada con éxito en entornos de alta concurrencia y dominio complejo.

Varios autores recomiendan una aplicación pragmática de CQRS y microservicios. Fowler (2015) y Richardson (2018) advierten que no todos los contextos se benefician igualmente de estos patrones, y que deben aplicarse de forma gradual y guiada por el dominio del negocio. En sis-

temas pequeños o con requisitos simples, una arquitectura monolítica bien estructurada puede ser más efectiva (Richardson, 2018).

En entornos .NET, CQRS se implementa frecuentemente con el patrón MediatR para desacoplar comandos y queries mediante un bus de mensajes interno. Combinado con DDD (Domain-Driven Design), este enfoque permite modelar las reglas del negocio con alta fidelidad y claridad (Lowy, 2013).

En resumen, la integración de microservicios y CQRS representa una estrategia avanzada para el diseño de software moderno, pero requiere una evaluación cuidadosa del contexto y la experiencia del equipo para evitar una complejidad innecesaria.

Beneficios y desafíos actuales

En resumen, los patrones de diseño contribuyen a la calidad del software en aspectos como reusabilidad, mantenibilidad y escalabilidad. Facilitan la comunicación técnica (describiendo soluciones con nombres estándar) y proveen una base común de conocimientos. Según (Freeman et al., 2004), el uso de patrones no solo mejora la mantenibilidad, sino también la comunicación entre desarrolladores mediante un lenguaje común.

Sin embargo, la aplicación de patrones no está exenta de retos. (Chippagiri, 2025) advierte que implementar patrones de forma incorrecta puede introducir anti-patrones (malas prácticas que imitan patrones válidos) y complejidad innecesaria. Además, automatizar la detección de patrones en código es difícil, lo que motiva la investigación en herramientas basadas en inteligencia artificial para identificación de patrones y anti-patrones. Por

otro lado, los avances tecnológicos han ampliado el alcance de los patrones tradicionales. Además de los patrones clásicos, se estudian patrones emergentes adaptados a la computación en nube, big data e inteligencia artificial. La revisión de Chippagiri destaca patrones avanzados para sistemas basados en IA y aprendizaje automático (como canalizaciones de procesamiento de datos, optimización de hiperparámetros), así como patrones de seguridad específicos (autenticación proxy, enmascarado de datos) para cumplir con regulaciones actuales. Igualmente se reconocen patrones en arquitecturas de nube: por ejemplo, caché distribuido y segmentación por compartimentos (bulkhead) para mejorar rendimiento y tolerancia a fallos en servicios distribuidos. En conclusión, el estado del arte muestra que los patrones de diseño siguen siendo un pilar del desarrollo de software de calidad, adaptándose a nuevos paradigmas. Su estudio actual no solo reafirma los beneficios clásicos (comunicación, mantenimiento y reutilización), sino que también aborda nuevas áreas (microservicios, IA, seguridad) y reconoce desafíos asociados a su uso.

Conclusiones

Los patrones de diseño sin lugar a dudas son soluciones a problemas conocidos dentro del desarrollo de software. Ya sea un patrón GoF o POSA nos permiten resolver distintos retos al momento de crear una herramienta de software.

Los patrones de arquitectura, nos permiten decidir cual de estos se ajusta más a la necesidad de la aplicación que vamos a desarrollar y con esto no tener problemas de mantenibilidad y escalabilidad en el futuro. Por su parte, los GoF nos permiten resolver distintos problemas que se pueden presentar en el desarrollo y la escritura del código de nuestro aplicativo; con esto podemos evitar problemas de redundancia, del llamado código espagueti y muchos otros que pueden llegar a ser un verdadero dolor de cabeza.

En este informe se ha revisado en profundidad la definición, clasificación y utilidad de los patrones de diseño en ingeniería de software. Como mencionamos, los patrones proporcionan un lenguaje común y un conjunto de soluciones maduras para problemas recurrentes, lo que mejora la calidad del diseño y acelera el desarrollo.

La clasificación tradicional (creacionales, estructurales y comportamentales) sigue siendo válida, y los patrones GOF ejemplifican cada categoría. Además, los patrones de arquitectura (como capas y MVC) orientan la estructura macro del sistema.

La investigación reciente indica que los patrones se aplican ampliamente en dominios modernos (sistemas basados en IA, microservicios y arquitecturas en nube). Sin embargo, también advierte sobre la

necesidad de aplicar los patrones con criterio, evitando anti-patrones y aprovechando herramientas avanzadas de soporte. Para la práctica en C#, muchos patrones se reflejan en características del lenguaje: por ejemplo, eventos/delegados implementan Observer, y clases como StringBuilder ejemplifican Builder. Concluimos que el conocimiento profundo de patrones de diseño es esencial para diseñar software robusto y de calidad. Se recomienda a los desarrolladores y arquitectos familiarizarse con estos patrones clásicos y emergentes, adaptándolos según contexto, siempre basándose en evidencia y fuentes confiables.

La incorporación de patrones, tal como se expone en obras fundamentales como las de (Gamma et al., 1995) y (Fowler, 2002), no solo permite abordar problemas comunes de forma estructurada, sino que también facilita el diseño sostenible de soluciones escalables.

Referencias

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture, volume 1: A system of patterns*. Wiley.

Ccori Huaman, W. (2018, September 7). *Los 10 patrones comunes de arquitectura de software*. https://medium.com/@maniakhitoccori/los-10-patrones-comunes-de-arquitectura-de-software-d8b9047edf0b

Chippagiri, S. (2025). A comprehensive review of software design patterns: Applications and future direction. *The Review of Contemporary Scientific and Academic Studies*, *5*(2), 1–12. https://doi.org/10.55454/rcsas.5.02.2025.001

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley.

Fowler, M. (2011). *CQRS*. https://martinfowler.com/bliki/CQRS.html

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head first design patterns*. O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Lowy, U. (2013). *Implementing domain-driven design*. Blue Spire Consulting.

Newman, S. (2015). *Building microservices*. O'Reilly Media.

Refactoring.Guru. (n.d., n.d.). *Patrones de diseño (design patterns)*. https://refactoring.guru/es/design-patterns

Richardson, C. (2018). *Microservices patterns: With examples in java*. Manning Publications.

Soler, D. (2024, May 10). *¿Qué son los patrones de diseño de software?*

https://keepcoding.io/blog/patrones-de-diseno-de-software/

Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.