Investigación sobre Patrones de Diseño

PSWE-04 Diseño de Software

Daniel Canessa Valverde Stephanie Delgado Brenes

2025-07-18

Tabla de contenidos

Abstract

Este artículo ofrece un análisis actualizado sobre patrones de diseño en el desarrollo de software, abordando su definición, clasificación, propósito y aplicaciones en distintos paradigmas de programación. Además, examina anti-patrones comunes que afectan la calidad del software y explora tendencias emergentes, con énfasis en la programación orientada a objetos y su evolución frente a tecnologías modernas como la nube, la inteligencia artificial y "DevOps".

1 Introducción

En el desarrollo de software, los problemas recurrentes y las demandas de calidad han llevado a la búsqueda de soluciones reutilizables y probadas. Los patrones de diseño surgieron como respuesta a esta necesidad, ofreciendo esquemas y estrategias para estructurar y organizar sistemas de manera eficiente y mantenible.

Si bien la programación orientada a objetos ha sido el contexto principal para la sistematización y aplicación de patrones, la realidad es que todos los paradigmas de programación presentan problemáticas recurrentes con soluciones estándar. Por ello, el estudio de los patrones trasciende el dominio de programación orientada a objetos y se extiende a paradigmas estructurados, funcionales, reactivos, lógicos, entre otros.

Este trabajo explora el universo de los patrones de diseño, su evolución y su impacto en la calidad del software, conectando su aplicación cotidiana con tendencias arquitectónicas, idiomáticas y tecnológicas modernas. El análisis va más allá de la simple enumeración de patrones clásicos, abordando su relación con principios de diseño, anti-patrones, y los nuevos retos que plantea la ingeniería de software contemporánea.

2 Objetivos

2.1 Objetivo general

Analizar el papel de los patrones de diseño en la ingeniería de software, explorando su evolución, clasificación, aplicación y relevancia para la construcción de sistemas robustos, mantenibles y alineados con los desafíos actuales de la industria.

2.2 Objetivos específicos

- Describir los fundamentos teóricos y los principales tipos de patrones de diseño en software, identificando sus características, ventajas y limitaciones en distintos paradigmas de programación.
- Examinar los patrones de diseño clásicos y modernos más relevantes, ilustrando su aplicación a través de ejemplos prácticos y casos de uso en contextos reales de desarrollo.

- Analizar la relación entre los patrones de diseño, los principios fundamentales del diseño orientado a objetos (como SOLID y GRASP), y su impacto en la calidad, mantenibilidad y evolución del software.

- Identificar los principales anti-patrones, riesgos y desafíos asociados al uso inapropiado o excesivo de patrones de diseño, así como las tendencias emergentes en la disciplina.

3 ¿Qué es un patrón de diseño?

Un patrón de diseño en ingeniería de software es una solución reutilizable a un problema común que ocurre frecuentemente en un contexto particular durante el proceso de desarrollo de software. El concepto de patrones surge originalmente en el campo de la arquitectura, donde Christopher Alexander y sus colaboradores los describieron como soluciones recurrentes a problemas de diseño en edificaciones y urbanismo [1]. Esta idea fue posteriormente adaptada al desarrollo de software por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, conocidos como la "Banda de los Cuatro" ("Gang of Four" o "GoF"), quienes en 1994 publicaron *Design Patterns: Elements of Reusable Object-Oriented Software*, esta publicación formalizó el uso de patrones en el ámbito del software [2].

A diferencia de un fragmento de código reutilizable, un patrón de diseño es una descripción abstracta de cómo resolver un tipo de problema específico de diseño, sin imponer una implementación única. Esto permite a los desarrolladores aplicar soluciones probadas y documentadas, facilitando la construcción de sistemas más robustos y mantenibles [3].

Típicamente, un patrón de diseño incluye:

- El nombre del patrón.
- El problema que resuelve.
- La solución propuesta (estructura y dinámica).
- Las consecuencias de su aplicación.

4 ¿Por qué se usan los patrones de diseño?

Los patrones de diseño previenen "reinventar la rueda" y proporcionan un lenguaje común para discutir problemas y estrategias en el desarrollo de software [4]. Utilizar patrones contribuye significativamente a la calidad del software, algunos beneficios clave son:

- **Reutilización de soluciones:** los patrones condensan la experiencia y el conocimiento de la industria, evitando la necesidad de reinventar soluciones ante los mismos problemas en distintos contextos [2].
- **Mejora de la comunicación:** el uso de patrones proporciona un vocabulario común y preciso para que los equipos de desarrollo puedan discutir, documentar y transferir ideas de manera clara y eficiente [2].
- **Buenas prácticas:** orientan hacia la creación de sistemas robustos, flexibles y mantenibles, ayudando a evitar errores comunes y facilitando la evolución del software [5].
- **Consistencia y estandarización:** el uso de patrones contribuye a establecer criterios uniformes en la estructura del software, facilitando la comprensión y colaboración entre equipos [2].
- **Mantenibilidad y escalabilidad:** facilitan la adaptación del sistema a nuevos requerimientos y su mantenimiento a lo largo del tiempo, permitiendo cambios controlados y predecibles [3].
- **Ahorro de tiempo y aumento de la productividad:** al ofrecer soluciones estructuradas a problemas conocidos, los patrones agilizan el proceso de diseño y toma de decisiones [2].

5 Patrones de diseño según paradigma de programación

El concepto de patrón de diseño no está limitado a la programación orientada a objetos ("OOP"). Aunque la formalización más conocida y documentada de los patrones provienen de OOP, la realidad es que todos los paradigmas de programación presentan problemas recurrentes para los cuales se han desarrollado soluciones reutilizables y reconocidas. Así, lenguajes como "C", que no implementan orientación a objetos, también hacen uso de patrones, aunque estos estén enfocados en el control del flujo, la gestión modular y el manejo eficiente de recursos [3].

Tal como señala Buschmann et al. [3], "los patrones de diseño representan soluciones probadas para problemas recurrentes en el diseño de software, independientemente del paradigma o lenguaje." La diferencia radica en el nivel de abstracción, el tipo de problemas que resuelven y la forma en que se expresan en el lenguaje o el paradigma en cuestión.

5.1 Particularidades de los patrones según el paradigma

- **Orientado a objetos:** La mayoría de los patrones de diseño clásicos y principios como "SOLID" y "GRASP" se definieron para este paradigma, aprovechando conceptos como clases, objetos, herencia y polimorfismo.
- **Estructurado:** Lenguajes como "C" o Pascal emplean patrones enfocados en la organización del flujo de control (como "Table-Driven Methods" o "State Machines"), manejo de recursos y modularidad, esto incluso sin contar con una sintaxis para objetos.
- **Funcional:** Los patrones se centran en la composición de funciones, inmutabilidad y transformación de datos. Patrones como "Monad" o "Pipe" son característicos en Haskell, Scala o F#.
- **Lógico:** Lenguajes como Prolog emplean patrones para resolución de problemas mediante reglas, inferencia y backtracking.

  Nota: Si bien existen patrones de diseño en todos los paradigmas de programación, su identificación es valiosa en cada contexto y se utilizan mucho en la práctica profesional, el enfoque principal de este documento estará en los patrones desarrollados y aplicados en la programación orientada a objetos ("OOP"), ya que es el ámbito donde estos han sido formalizados, documentados y utilizados más extensamente en la literatura.

6 Principios de diseño: SOLID y GRASP

El diseño de software en el paradigma orientado a objetos se apoya en principios fundamentales que guían la toma de decisiones para lograr sistemas de alta calidad. Entre los más reconocidos y aplicados en la industria destacan los principios "SOLID" y los patrones "GRASP". Estos proporcionan criterios claros para la asignación de responsabilidades, la gestión de dependencias y la evolución sostenible del software, facilitando la correcta aplicación de patrones de diseño [2].

6.1 Principios SOLID

El acrónimo "SOLID" agrupa cinco principios esenciales, formulados por Robert C. Martin ("Uncle Bob") para guiar el diseño y la implementación de clases y objetos en "OOP". Estos principios fomentan la cohesión y reducen el acoplamiento entre los componentes del software, facilitando su evolución, extensión y mantenimiento a largo plazo [2].

- **Principio de Responsabilidad Única (SRP):** una clase debe tener una única responsabilidad, es decir, solo una razón para cambiar. *Ejemplo:* Una clase "GestorDeUsuarios" que solo administra usuarios, y no reportes ni validaciones externas.
- **Principio Open/Close (OCP):** las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación. *Ejemplo:* Implementar nuevas formas de pago añadiendo nuevas clases que implementan una interfaz "Pago", sin modificar el código existente.
- **Principio de Sustitución de Liskov (LSP):** los objetos de una subclase deben poder sustituir a los de la clase base sin alterar el funcionamiento correcto del sistema. *Ejemplo:* Una clase "PatoDeGoma" que implementa a "Pato" debe poder usarse donde se espera un "Pato", sin romper la lógica.
- **Principio de Segregación de Interfaces (ISP):** es preferible definir varias interfaces específicas y pequeñas antes que una interfaz general y amplia. *Ejemplo:* Interfaces separadas "Imprimible" y "Escaneable" en vez de una única interfaz "DispositivoMultifunción".

- **Principio de Inversión de Dependencias (DIP):** los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. *Ejemplo:* Una clase "ControladorDeLuz" depende de la interfaz "Interruptor", no de una implementación concreta.

6.2 Patrones GRASP

Los patrones "GRASP" ("General Responsibility Assignment Software Patterns"), propuestos por Craig Larman, ofrecen un marco para asignar responsabilidades en "OOP" [4]. Cada patrón proporciona una respuesta estructurada a preguntas clave sobre cómo distribuir tareas y relaciones entre clases y objetos:

- **Creador (Creator):** indica qué clase debe ser responsable de crear instancias de otras clases, basándose en la posesión o uso de los objetos creados. *Ejemplo:* Una clase "Pedido" crea instancias de "DetalleDePedido" porque "Pedido" está compuesto por varios detalles, y administra el ciclo de vida de esos objetos.
- **Experto en información (Information Expert):** asigna la responsabilidad a la clase que tiene la información necesaria para cumplirla de manera eficiente. *Ejemplo:* Una clase "CuentaBancaria" calcula su saldo porque posee la información de los movimientos.
- **Controlador (Controller):** sugiere la introducción de un objeto intermediario que reciba y coordine solicitudes provenientes de la interfaz de usuario u otros sistemas externos. *Ejemplo:* Un objeto "ControladorDeReservas" recibe las solicitudes de la interfaz de usuario para crear, modificar o cancelar reservas, y coordina las operaciones necesarias sobre los objetos "Reserva", "Cliente" y "Habitación".
- **Bajo acoplamiento (Low Coupling):** favorece el diseño de clases y módulos con dependencias mínimas, promoviendo la flexibilidad y facilidad de cambio. *Ejemplo:* Una clase "Reporte" que recibe sus dependencias vía interfaces, no crea instancias concretas internamente.
- **Alta cohesión (High Cohesion):** promueve que las clases agrupen responsabilidades altamente relacionadas, evitando clases que gestionen tareas que no le corresponden. *Ejemplo:* Una clase "GestorDeClientes" se ocupa solo de operaciones sobre clientes, no de facturación ni inventarios.
- **Polimorfismo (Polymorphism):** permite que diferentes tipos de objetos respondan a la misma interfaz o mensaje, haciendo posible que una misma operación se ejecute de distintas formas según el contexto. El polimorfismo se puede dar a nivel de clases (mediante herencia e interfaces) y a nivel de funciones (mediante sobrecarga o uso de genéricos). *Ejemplo:* Una interfaz "Notificación" define el método "enviar()". Tanto "NotificaciónPorCorreo" como "NotificaciónPorSMS" implementan este método con

lógica diferente. El sistema puede usar una referencia de tipo "Notificación" para invocar "enviar()", la implementación concreta dependerá del tipo real del objeto.

- **Fabricación pura (Pure Fabrication):** plantea la creación de clases que no representan conceptos del dominio real, sino que existen únicamente para lograr un diseño más flexible, reutilizable o desacoplado. *Ejemplo:* Una clase "ServicioDeAutenticación" que gestiona la verificación de credenciales y el manejo de sesiones de usuario, pero no representa un concepto real del dominio, sino que existe para separar responsabilidades y mejorar la reutilización del código.
- **Indirección (Indirection):** introduce un objeto intermediario para desacoplar dos elementos, facilitando la extensibilidad y permitiendo cambios en uno de los extremos sin afectar al otro. *Ejemplo:* El patrón "Adapter", donde una clase adapta la interfaz de un componente externo para que pueda ser utilizado por el sistema.
- **Variaciones protegidas (Protected Variations):** identifica los puntos donde es probable que ocurran cambios o variaciones y los protege mediante abstracciones, minimizando el impacto de los cambios futuros. *Ejemplo:* Definir una interfaz "Almacenamiento" y tener varias implementaciones ("AlmacenamientoEnMemoria", "AlmacenamientoEnDisco") para proteger el resto del sistema de cambios en la forma de almacenamiento.

7 Clasificación clásica de patrones de diseño

La clasificación clásica de los patrones de diseño en software fue establecida por la llamada "Banda de los Cuatro" ("Gang of Four" o "GoF") en su libro *Design Patterns: Elements of Reusable Object-Oriented Software* [2]. En este libro, los autores describen 23 patrones fundamentales para la programación orientada a objetos, agrupando los patrones en creacionales, estructurales y de comportamiento.

7.1 1. Patrones creacionales

Estos patrones abordan la forma en que los objetos se crean, permitiendo mayor flexibilidad y control sobre la instancia de clases.

7.1.1 Singleton

- **Propósito:** garantizar que una clase tenga solo una instancia a lo largo del ciclo de vida de la aplicación y ofrecer un punto global de acceso a ella.
- **Beneficios:** asegura el control centralizado de recursos y evita inconsistencias por múltiples instancias.
- **Contexto de uso:** ideal para gestores de configuración, controladores de acceso a base de datos, registros de logs o recursos compartidos.
- **Ejemplo:** Un `GestorDeConfiguración` único para toda la aplicación.

7.1.2 Factory Method

- **Propósito:** definir una interfaz para crear un objeto, pero delegar a las subclases la decisión sobre qué clase concreta instanciar.
- **Beneficios:** permite que el sistema sea flexible y extensible, evitando acoplarse a clases concretas y facilitando la sustitución o extensión de creadores.
- **Contexto de uso:** cuando la creación de objetos depende de la lógica o datos que sólo se conocen en tiempo de ejecución, o para desacoplar código cliente de implementaciones concretas.
- **Ejemplo:** Un método `crearNotificador()` que devuelve instancias de `NotificadorCorreo` o `NotificadorSMS` según configuración.

7.1.3 Abstract Factory

- **Propósito:** proveer una interfaz para crear familias de objetos relacionados (por ejemplo, botones y menús) sin especificar las clases concretas que se van a utilizar.
- **Beneficios:** asegura la coherencia entre productos de la misma familia y permite cambiar toda la familia de objetos fácilmente.
- **Contexto de uso:** sistemas multiplataforma, o cuando el sistema debe ser independiente de cómo se crean, componen y representan los productos.
- **Ejemplo:** Una fábrica de interfaces gráficas que puede crear componentes (botón, menú, ventana) para Windows o para Linux.

7.1.4 Builder

- **Propósito:** separar la construcción de un objeto complejo de su representación final, permitiendo crear diferentes variantes de un mismo objeto mediante un proceso de construcción paso a paso.
- **Beneficios:** facilita la creación de objetos complejos con múltiples opciones, mejorando la legibilidad y la gestión de dependencias internas.
- **Contexto de uso:** objetos que requieren configuración detallada o composición de partes, como documentos, reportes o automóviles.
- **Ejemplo:** Un `ConstructorDeReporte` que permite armar reportes personalizados agregando secciones a voluntad.

7.1.5 Prototype

- **Propósito:** crear nuevos objetos a partir de la clonación de una instancia prototipo existente, en vez de crear nuevos desde cero.
- **Beneficios:** agiliza la creación de objetos complejos o costosos, y permite copiar configuraciones fácilmente.
- **Contexto de uso:** juegos, editores gráficos, o cuando la creación de un objeto es costosa en tiempo o recursos y conviene copiar un modelo existente.
- **Ejemplo:** Una instancia de `Figura` (círculo, rectángulo) que se clona para crear nuevas figuras similares.

7.2 2. Patrones estructurales

Estos patrones facilitan la composición de clases y objetos para formar estructuras más grandes y flexibles.

7.2.1 Adapter

- **Propósito:** permitir la colaboración entre clases con interfaces incompatibles, transformando la interfaz de una clase en otra esperada por el cliente.
- **Beneficios:** posibilita el uso de clases existentes en nuevos contextos sin modificar su código.
- **Contexto de uso:** integración de sistemas antiguos con sistemas nuevos, o integración de librerías externas con la aplicación.
- **Ejemplo:** Un adaptador que convierte la salida de una API de pagos para que sea compatible con la interfaz interna del sistema.

7.2.2 Bridge

- **Propósito:** separar una abstracción de su implementación, permitiendo que ambas evolucionen de forma independiente.
- **Beneficios:** reduce el acoplamiento entre jerarquías de abstracción e implementación, facilita la extensión y el mantenimiento.
- **Contexto de uso:** sistemas que requieren combinar múltiples variantes de una abstracción y múltiples variantes de implementación, sin multiplicar clases.
- **Ejemplo:** Un sistema de documentos (`Documento`) que puede combinarse con diferentes formas de exportar (`ExportadorPDF`, `ExportadorHTML`), permitiendo nuevas combinaciones sin modificar el resto del sistema.

7.2.3 Composite

- **Propósito:** componer objetos en estructuras de árbol para representar jerarquías parte-todo, permitiendo que los clientes traten objetos simples y compuestos de manera uniforme.
- **Beneficios:** simplifica el código cliente, permite operaciones recursivas y jerarquías complejas.
- **Contexto de uso:** menús de aplicaciones, sistemas de archivos, estructuras gráficas o de organización.
- **Ejemplo:** Un menú de sistema con opciones simples y submenús, tratados todos como `ElementoMenu`.

7.2.4 Decorator

- **Propósito:** añadir responsabilidades adicionales a un objeto de manera dinámica, sin modificar la clase original.
- **Beneficios:** permite agregar funcionalidades a objetos individuales de forma flexible, favoreciendo la composición sobre la herencia.

- **Contexto de uso:** cuando se requieren combinaciones dinámicas de funcionalidades, como agregar cifrado, compresión o logs a una clase base.
- **Ejemplo:** Un `NotificadorConEncriptación` que extiende un `Notificador` agregando cifrado antes de enviar mensajes.

7.2.5 Facade

- **Propósito:** proporcionar una interfaz unificada y simplificada a un conjunto de interfaces de un subsistema complejo.
- **Beneficios:** reduce la complejidad y el acoplamiento del sistema, facilitando el uso de subsistemas.
- **Contexto de uso:** aplicaciones con muchos módulos o APIs complejas, donde conviene simplificar la interacción para el cliente.
- **Ejemplo:** Una `FachadaBancaria` que expone métodos simples para operaciones bancarias y oculta la lógica interna de cuentas, tarjetas y préstamos.

7.2.6 Flyweight

- **Propósito:** reducir el consumo de memoria compartiendo partes de objetos entre muchas instancias, cuando estas comparten datos en común.
- **Beneficios:** eficiencia y escalabilidad en sistemas con gran cantidad de objetos similares.
- **Contexto de uso:** edición de texto, juegos, gráficos vectoriales donde se crean muchos objetos ligeros.
- **Ejemplo:** Objetos `Caracter` que comparten la misma representación de letra en diferentes posiciones de un documento.

7.2.7 Proxy

- **Propósito:** proporcionar un representante o intermediario para controlar el acceso a otro objeto, gestionando su creación, acceso o protección.
- **Beneficios:** controla acceso remoto, carga diferida, protección y registro.
- **Contexto de uso:** control de acceso a recursos, optimización de cargas pesadas, proxies de red.
- **Ejemplo:** Un `ProxyImagen` que gestiona la carga de una imagen desde disco sólo cuando se necesita mostrarla.

7.3 3. Patrones de comportamiento

Estos patrones se enfocan en cómo los objetos interactúan y distribuyen responsabilidades.

7.3.1 Chain of Responsibility

- **Propósito:** evitar el acoplamiento entre el emisor y el receptor de una petición, pasando la solicitud por una cadena de objetos hasta que uno la maneje.
- **Beneficios:** flexibilidad para agregar, quitar o reordenar manejadores en tiempo de ejecución.
- **Contexto de uso:** validaciones en formularios, procesamiento de eventos, sistemas de soporte o autorización.
- **Ejemplo:** Una cadena de objetos de validación de entrada para un formulario de usuario.

7.3.2 Command

- **Propósito:** encapsular una solicitud como un objeto, permitiendo parametrizar clientes con diferentes operaciones, almacenar operaciones y soportar deshacer/rehacer.
- **Beneficios:** desacopla quien solicita la acción de quien la ejecuta, permite colas de comandos y operaciones deshacer.
- **Contexto de uso:** menús, barras de herramientas, sistemas con operaciones que pueden programarse o deshacerse.
- **Ejemplo:** Un comando `GuardarDocumento` ejecutable y reversible en un editor de texto.

7.3.3 Interpreter

- **Propósito:** definir una representación y un intérprete para el lenguaje de un dominio específico ("DSL").
- **Beneficios:** facilita la construcción de analizadores o ejecutores de lenguajes simples.
- **Contexto de uso:** interpretación de comandos, procesamiento de expresiones matemáticas, reglas de negocio.
- **Ejemplo:** Un intérprete que evalúa expresiones aritméticas ingresadas por el usuario.

7.3.4 Iterator

- **Propósito:** proporcionar una manera estándar de recorrer una colección de objetos sin exponer su estructura interna.
- **Beneficios:** separa el recorrido de la lógica interna, permite múltiples tipos de recorridos.
- **Contexto de uso:** listas, árboles, colecciones complejas.
- **Ejemplo:** Un `IteradorDeLista` que permite recorrer los elementos de una lista de clientes.

7.3.5 Mediator

- **Propósito:** centralizar la comunicación entre objetos para reducir dependencias directas y simplificar la interacción.
- **Beneficios:** disminuye el acoplamiento y mejora la modularidad del sistema.
- **Contexto de uso:** interfaces gráficas, sistemas donde muchos objetos se comunican entre sí.
- **Ejemplo:** Un `MediadorChat` que coordina el envío de mensajes entre usuarios en una sala de chat.

7.3.6 Memento

- **Propósito:** capturar y almacenar el estado interno de un objeto para poder restaurarlo más tarde, sin violar el encapsulamiento.
- **Beneficios:** permite deshacer cambios y restaurar estados anteriores de forma segura.
- **Contexto de uso:** editores, sistemas de juego, historial de cambios en documentos.
- **Ejemplo:** Un `MementoDocumento` que almacena versiones previas del documento.

7.3.7 Observer

- **Propósito:** establecer una relación de dependencia uno-a-muchos, de modo que cuando el objeto observado cambia, notifica automáticamente a sus observadores.
- **Beneficios:** promueve el bajo acoplamiento y facilita la extensión de funcionalidad.
- **Contexto de uso:** sistemas de eventos, interfaces de usuario, modelos "pub/sub".
- **Ejemplo:** Un `SensorDeTemperatura` que notifica a varias pantallas cuando la temperatura cambia.

7.3.8 State

- **Propósito:** permitir que un objeto cambie su comportamiento cuando su estado interno cambia, comportándose como si fuera de otra clase.
- **Beneficios:** organiza los comportamientos dependientes del estado, evitando condicionales complejos.
- **Contexto de uso:** máquinas de estados, flujos de trabajo, sistemas con diferentes modos operativos.
- **Ejemplo:** Un objeto `Semáforo` que alterna entre los estados de `verde`, `amarillo` y `rojo`, cambiando su comportamiento según su estado actual.

7.3.9 Strategy

- **Propósito:** definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables, permitiendo que el algoritmo varíe independientemente del cliente.
- **Beneficios:** favorece el código flexible y la selección dinámica de algoritmos.
- **Contexto de uso:** algoritmos de ordenamiento, cálculos tributarios, estrategias de juego.
- **Ejemplo:** Un `CalculadorDeDescuento` que selecciona la estrategia de cálculo adecuada según el tipo de cliente.

7.3.10 Template Method

- **Propósito:** definir el esqueleto de un algoritmo, dejando que algunos pasos sean implementados por subclases.
- **Beneficios:** promueve la reutilización de código y la personalización controlada.
- **Contexto de uso:** procesos que tienen una estructura fija con pasos variables.
- **Ejemplo:** Una clase `ProcesadorDeArchivo` que define el proceso general de lectura y procesamiento, permitiendo que subclases definan cómo procesar cada línea.

7.3.11 Visitor

- **Propósito:** separar una operación de la estructura de objetos sobre la que opera, permitiendo añadir nuevas operaciones sin modificar las clases de esos objetos.
- **Beneficios:** facilita la adición de funcionalidades y el mantenimiento del sistema.
- **Contexto de uso:** recorridos de estructuras de datos, análisis sintáctico, operaciones sobre árboles o colecciones heterogéneas.
- **Ejemplo:** Un visitante que recorre un sistema de archivos y calcula el tamaño total de todos los archivos.

8 Otros patrones de diseño reconocidos

Además de los patrones clásicos de "GoF", la literatura y la práctica profesional han documentado otros patrones de diseño ampliamente utilizados, que resuelven problemas comunes en la construcción y mantenimiento de sistemas de software.

8.0.1 Null Object [3]

- **Propósito:** eliminar la necesidad de comprobaciones explícitas de valores nulos, implementando un objeto "vacío" que cumple la interfaz requerida sin realizar ninguna acción.
- **Beneficios:** simplifica el código cliente, evita errores por referencias nulas ("null pointer exceptions"), y mejora la robustez y legibilidad.
- **Contexto de uso:** cuando es común que un objeto pueda estar ausente, y se desea evitar condicionales repetidos o comportamientos especiales para el caso nulo.
- **Ejemplo:** En un sistema de notificaciones, si no hay un notificador activo, se puede usar un objeto "NotificadorNulo" que implementa la interfaz de notificación pero no realiza ninguna acción al llamar al método "enviar()".

8.0.2 Object Pool [3]

- **Propósito:** gestionar y reutilizar un conjunto de objetos costosos de crear, manteniéndolos en un "pool" y sirviéndolos a los clientes según demanda.
- **Beneficios:** reduce la sobrecarga de instanciación y destrucción repetitiva de objetos, mejora el rendimiento y la utilización de recursos.
- **Contexto de uso:** conexiones a bases de datos, hilos, buffers, o cualquier recurso cuyo costo de creación sea significativo.
- **Ejemplo:** Un pool de conexiones a la base de datos que entrega instancias disponibles y las reutiliza cuando se liberan.

8.0.3 Dependency Injection [6]

- **Propósito:** desacoplar la creación y la gestión de dependencias de los objetos, delegando su provisión a un contenedor externo o al propio framework.

- **Beneficios:** favorece la inyección de diferentes implementaciones, mejora la testabilidad, facilita la configuración y la extensión del sistema.
- **Contexto de uso:** sistemas grandes con múltiples dependencias, frameworks modernos como “Spring”, “Angular” o “.NET Core”.
- **Ejemplo:** Un servicio de autenticación recibe un repositorio de usuarios por inyección, lo que permite sustituirlo fácilmente por un repositorio de pruebas en los tests.

8.0.4 Specification [7]

- **Propósito:** encapsular criterios de selección, validación o filtrado en objetos “especificación”, que pueden combinarse y reutilizarse.
- **Beneficios:** mejora la claridad y flexibilidad de las reglas de negocio, favorece la reutilización y la composición de criterios complejos.
- **Contexto de uso:** validación de reglas de negocio, filtros en consultas, construcción de criterios dinámicos.
- **Ejemplo:** Un objeto `EspecificacionClienteVIP` define el criterio para identificar clientes VIP; puede combinarse con otras especificaciones para consultas más complejas.

9 Patrones de diseño más allá de “GoF”

Si bien los patrones de diseño clásicos definidos por Gamma et al. (“GoF”) [2] han sido fundamentales para el desarrollo orientado a objetos, el avance de la tecnología, la complejidad de los sistemas y la diversidad de paradigmas han impulsado la aparición de nuevos patrones que atienden desafíos actuales como la escalabilidad, integración, resiliencia y adaptabilidad.

9.1 1. Patrones arquitectónicos

Estos patrones resuelven problemas recurrentes en la estructura global y la organización de sistemas de software, operando en un nivel de abstracción superior al de los patrones de diseño clásicos. Mientras estos últimos se centran en la colaboración entre clases y objetos dentro de componentes individuales, los arquitectónicos abordan la disposición de módulos principales, su comunicación y la facilitación de escalabilidad y mantenimiento a nivel de todo el sistema [3].

A continuación, se describen algunos de los patrones arquitectónicos más relevantes en el desarrollo de software moderno:

9.1.1 MVC (“Modelo-Vista-Controlador”) [3]

- **Propósito:** separar la lógica de negocio, la interfaz de usuario y el control de eventos/acciones, facilitando la gestión modular de aplicaciones.
- **Beneficios:** promueve el mantenimiento, la reutilización de componentes y el desarrollo en paralelo de distintas partes del sistema.
- **Contexto de uso:** aplicaciones web y de escritorio que requieren una clara separación entre presentación y lógica.
- **Ejemplo:** En una aplicación de reservas, el modelo gestiona los datos de las reservas, la vista muestra las fechas y horarios, y el controlador procesa las acciones del usuario.

9.1.2 Microservicios [8]

- **Propósito:** organizar una aplicación como un conjunto de servicios pequeños, independientes y desplegables de manera autónoma.

- **Beneficios:** facilita la escalabilidad horizontal, la resiliencia y la evolución independiente de los distintos servicios; favorece equipos autónomos.
- **Contexto de uso:** sistemas con alta demanda de escalabilidad y despliegue frecuente, como plataformas de streaming, banca digital o comercio electrónico.
- **Ejemplo:** Un sistema de compras online donde la gestión de productos, pagos y envíos se realiza mediante servicios independientes que se comunican vía API.

9.1.3 Microkernel ("Plug-in") [3]

- **Propósito:** permitir la extensión dinámica de funcionalidades mediante módulos o "plug-ins" sobre un núcleo central.
- **Beneficios:** posibilita adaptar o ampliar el sistema sin modificar el núcleo ni detener su operación; fomenta la extensibilidad y personalización.
- **Contexto de uso:** aplicaciones extensibles como editores de texto, IDEs o servidores de aplicaciones.
- **Ejemplo:** Un editor de texto al que se pueden agregar "plug-ins" para soporte de nuevos lenguajes o funciones avanzadas.

9.1.4 Layered ("Capas") [3]

- **Propósito:** organizar el sistema en capas jerárquicas, donde cada capa proporciona servicios a la superior y utiliza servicios de la inferior.
- **Beneficios:** facilita la separación de responsabilidades, el mantenimiento y la reutilización; permite modificar o reemplazar capas sin afectar el resto del sistema.
- **Contexto de uso:** aplicaciones empresariales, sistemas operativos, aplicaciones web con capas de presentación, lógica de negocio y acceso a datos.
- **Ejemplo:** Una aplicación web estructurada en capas de presentación, lógica de negocio y persistencia de datos.

9.1.5 Pipes and Filters ("Tuberías y Filtros") [3]

- **Propósito:** dividir el procesamiento en una secuencia de pasos independientes (filtros) conectados por canales (tuberías), donde cada filtro transforma los datos.
- **Beneficios:** promueve la reutilización y composición flexible de componentes. Además facilita el procesamiento incremental y la paralelización.
- **Contexto de uso:** procesamiento de datos, compiladores, sistemas de procesamiento multimedia.
- **Ejemplo:** Un sistema de procesamiento de imágenes donde cada filtro aplica una transformación (escala, color, rotación) y los datos fluyen entre filtros.

9.1.6 Blackboard [3]

- **Propósito:** resolver problemas complejos mediante la colaboración de varios componentes especializados que acceden y actualizan una estructura común llamada "blackboard".
- **Beneficios:** permite la integración de diferentes estrategias de solución y la cooperación entre módulos independientes.
- **Contexto de uso:** sistemas de inteligencia artificial, reconocimiento de voz, sistemas expertos.
- **Ejemplo:** Un sistema de reconocimiento de voz donde distintos módulos aportan hipótesis y resultados al "blackboard" central para construir la interpretación final.

9.1.7 Broker [3]

- **Propósito:** gestionar la comunicación entre componentes distribuidos mediante un intermediario ("broker") que desacopla emisores y receptores.
- **Beneficios:** facilita la escalabilidad, la interoperabilidad y la integración de sistemas heterogéneos; simplifica la gestión de mensajes y servicios.
- **Contexto de uso:** middleware empresarial, integración de aplicaciones distribuidas, sistemas orientados a servicios.
- **Ejemplo:** Un sistema de mensajería empresarial donde los servicios de inventario, facturación y envíos interactúan a través de un broker de mensajes.

9.2 2. Patrones de integración

En sistemas distribuidos y arquitecturas modernas, los patrones de integración abordan problemas de comunicación, coordinación y sincronización entre componentes heterogéneos, frecuentemente implementados en distintas plataformas o tecnologías. Estos patrones facilitan la interoperabilidad, la escalabilidad y la consistencia de los datos a través de soluciones probadas para la transferencia y procesamiento de mensajes, comandos y eventos [9].

9.2.1 Event Sourcing [10]

- **Propósito:** almacenar todos los cambios de estado de un sistema como eventos inmutables, en vez de solo guardar el estado actual.
- **Beneficios:** permite trazabilidad completa, deshacer acciones, reconstruir el estado histórico y facilita la auditoría.
- **Contexto de uso:** aplicaciones con requerimientos de auditoría, historial o reversión, como sistemas de órdenes, logística o control de inventario.

- **Ejemplo:** Un sistema de pedidos que registra cada creación, modificación y cancelación como eventos, permitiendo reconstruir el historial completo de un pedido.

9.2.2 CQRS ("Command Query Responsibility Segregation") [10]

- **Propósito:** separar los modelos para comandos (escritura) y consultas (lectura) para optimizar el rendimiento y la escalabilidad.
- **Beneficios:** especialización y optimización de los modelos, mejoras en el rendimiento y la seguridad, escalabilidad independiente de lectura y escritura.
- **Contexto de uso:** sistemas donde las operaciones de consulta y actualización tienen diferentes requisitos de rendimiento o escalabilidad, como plataformas de comercio electrónico.
- **Ejemplo:** Un sistema de ventas con una base de datos para registrar compras y otra, optimizada para reportes y consultas rápidas.

9.3 3. Patrones para la nube y sistemas distribuidos

Las arquitecturas "cloud" y los sistemas distribuidos modernos presentan nuevos retos en cuanto a resiliencia, descubrimiento de servicios, balanceo de carga y tolerancia a fallos. Los siguientes patrones permiten abordar estos desafíos, facilitando la operación confiable y escalable de aplicaciones en la nube y entornos distribuidos [8].

9.3.1 Circuit Breaker [11]

- **Propósito:** proteger a un sistema de llamadas repetidas a servicios fallidos, evitando la sobrecarga y permitiendo una recuperación controlada.
- **Beneficios:** mejora la resiliencia y estabilidad general, evita cascadas de fallos en sistemas distribuidos.
- **Contexto de uso:** arquitecturas de microservicios, servicios que dependen de recursos externos o APIs poco confiables.
- **Ejemplo:** Una aplicación que detecta que el sistema de pagos externo está caído y, temporalmente, deja de enviar solicitudes para evitar saturar el sistema.

9.3.2 API Gateway [8]

- **Propósito:** centralizar el acceso a múltiples servicios de backend a través de un punto de entrada único.
- **Beneficios:** simplifica la gestión de autenticación, control de tráfico, transformación de mensajes y monitoreo de servicios; mejora la experiencia de clientes.

- **Contexto de uso:** arquitecturas de microservicios y aplicaciones móviles/web que consumen múltiples servicios internos.
- **Ejemplo:** Un "API Gateway" que recibe solicitudes de apps móviles y las distribuye a servicios internos de usuarios, productos y pedidos.

9.3.3 Service Discovery [8]

- **Propósito:** permitir que los servicios encuentren y se comuniquen automáticamente entre sí en entornos dinámicos y escalables.
- **Beneficios:** facilita el escalado, la actualización de servicios y la tolerancia a fallos sin intervención manual.
- **Contexto de uso:** orquestadores de contenedores ("Kubernetes", "Docker Swarm"), sistemas con servicios que cambian de ubicación frecuentemente.
- **Ejemplo:** En "Kubernetes", los servicios se descubren dinámicamente a través de DNS interno y etiquetas, sin configuración manual de direcciones IP.

9.4 4. Patrones idiomáticos de lenguajes modernos

Algunos lenguajes y plataformas han dado lugar a patrones específicos que explotan sus capacidades particulares, promoviendo soluciones idiomáticas que aprovechan las ventajas del lenguaje y su ecosistema [12].

9.4.1 Channels ("Go") [12]

- **Propósito:** facilitar la comunicación segura y concurrencia entre procesos ("goroutines") mediante el paso de mensajes.
- **Beneficios:** elimina muchos problemas asociados a "locks" y condiciones, promoviendo un modelo seguro y fácil de entender.
- **Contexto de uso:** aplicaciones concurrentes y paralelas escritas en "Go", procesamiento en tiempo real.
- **Ejemplo:** Una aplicación web en "Go" usa canales para enviar tareas desde el servidor HTTP a los workers que las procesan en paralelo.

9.4.2 Option/Result ("Rust") [13]

- **Propósito:** gestionar el manejo seguro de valores opcionales (que pueden o no estar presentes) y el control explícito de errores sin usar excepciones.
- **Beneficios:** promueve código seguro y robusto, obliga al programador a manejar casos de error o ausencia de valores.

- **Contexto de uso:** desarrollo en “Rust”, especialmente en funciones que pueden fallar o devolver valores nulos.
- **Ejemplo:** En “Rust”, una función que puede devolver un valor o no (“Option”) o que puede fallar (“Result<T, E>”), forzando al programador a manejar ambos casos.

9.4.3 Promise/Async Await (“JavaScript”, “Python”, “C#”) [14]

- **Propósito:** manejar operaciones asincrónicas de forma legible y controlada, evitando el “callback hell”.
- **Beneficios:** facilita la programación reactiva y el trabajo con operaciones I/O, redes o temporizadores.
- **Contexto de uso:** desarrollo web frontend y backend, aplicaciones que requieren concurrencia y manejo de múltiples eventos.
- **Ejemplo:** Una función en “JavaScript” utiliza “async” y “await” para esperar la respuesta de una API antes de continuar con la ejecución.

10 Tendencias futuras en patrones de diseño

El universo de los patrones de diseño está lejos de ser estático; su evolución refleja tanto la madurez de la ingeniería de software como los desafíos emergentes de las nuevas tecnologías y modelos de desarrollo. Los nuevos patrones trascienden la orientación a objetos y responden a problemáticas de escala, automatización, inteligencia artificial y operación continua.

Entre las tendencias más relevantes destacan:

- **Patrones para "Inteligencia Artificial" y "Machine Learning":** El diseño de sistemas inteligentes requiere arquitecturas para gestionar flujos de datos complejos, "pipelines" de entrenamiento y despliegue de modelos, reproducibilidad de experimentos y manejo ético de la inteligencia artificial. Se han propuesto patrones específicos para estos retos, como "ML Patterns" y frameworks para reproducibilidad y mantenimiento de modelos [15].
- **Patrones para arquitecturas "serverless":** Con la popularización del cómputo sin servidor, emergen estrategias y patrones para orquestar funciones efímeras, gestionar el estado y la comunicación en plataformas desacopladas, así como optimizar costos y tiempos de respuesta [16].
- **Patrones para "DevOps" y automatización:** La integración entre desarrollo y operación está impulsando patrones para infraestructura como código, despliegues progresivos ("blue-green deployment", "canary releases"), monitoreo automatizado y autoescalado [17].
- **Patrones para sistemas distribuidos y resilientes:** El auge de arquitecturas "cloud-native" y microservicios exige patrones avanzados para tolerancia a fallos, recuperación automática y operación global [8].
- **Patrones para aplicaciones reactivas y en tiempo real:** La creciente demanda de experiencias interactivas y procesamiento masivo de eventos impulsa la creación de patrones para manejar grandes volúmenes de datos, flujos de eventos y baja latencia [18].

11 Anti-patrones

Así como existen patrones de diseño reconocidos por aportar soluciones efectivas a problemas recurrentes, la literatura también ha documentado los **anti-patrones**. Estos son prácticas comunes que, aunque parezcan resolver un problema a corto plazo, suelen conducir a sistemas difíciles de mantener, poco escalables o propensos a errores [19].

Identificar y evitar anti-patrones es fundamental para garantizar la calidad del software y aprender de errores recurrentes en la industria.

11.1 Ejemplos de anti-patrones:

- **God Object** ("Objeto Dios"): concentración excesiva de responsabilidades en una sola clase u objeto, violando el principio de responsabilidad única y dificultando el mantenimiento.
- **Spaghetti Code:** código sin estructura clara, con dependencias y saltos de control desordenados, lo que dificulta su comprensión y modificación.
- **Golden Hammer:** aplicar siempre la misma solución conocida (patrón, tecnología o enfoque) a cualquier problema, incluso cuando no es adecuado.
- **Lava Flow:** acumulación de código obsoleto o experimental que permanece en el sistema, generando complejidad innecesaria y dificultando el mantenimiento.
- **Cargo Cult Programming:** imitar prácticas, patrones o fragmentos de código sin comprender su propósito, esperando que simplemente "funcionen".

Un caso particularmente relevante en el contexto de este trabajo es el uso inapropiado o forzado de patrones de diseño. Aplicar patrones por moda, sin un análisis real del problema, o tratar de incorporar demasiados patrones en una solución sencilla, puede convertir el diseño en un sistema innecesariamente complejo, rígido y difícil de entender. Este fenómeno es conocido como **Pattern Overuse** o **Pattern Abuse**, y se considera un anti-patrón moderno, pues contradice el objetivo central de los patrones: simplificar el diseño y mejorar la mantenibilidad [20].

12 Conclusiones

A partir del análisis realizado en este documento, se pueden establecer las siguientes conclusiones basadas en los objetivos específicos planteados:

- El valor de los patrones de diseño reside en la capacidad de abstraer y documentar soluciones reutilizables a problemas recurrentes, demostrando su universalidad al trascender paradigmas y lenguajes. Si bien su formalización más influyente ha sido en el contexto de la programación orientada a objetos, existen adaptaciones efectivas para paradigmas estructurado, funcional, lógico y nuevos lenguajes que combinan paradigmas o plantean nuevos paradigmas, lo que amplía su relevancia en la ingeniería de software.
- La inclusión de ejemplos prácticos y casos reales evidencia la aplicabilidad y vigencia de los patrones de diseño para resolver desafíos concretos en el desarrollo de sistemas. La variedad de enfoques, desde los patrones clásicos hasta los arquitectónicos y de integración, permite abordar problemas en distintos niveles de abstracción.
- La interacción entre patrones de diseño y principios como SOLID y GRASP materializa buenas prácticas de diseño orientado a objetos, aportando cohesión, bajo acoplamiento y facilitando el mantenimiento y la evolución sostenible del software. Esta relación favorece la construcción de sistemas preparados para el cambio y reduce la ocurrencia de errores y deuda técnica.
- La identificación de anti-patrones y la reflexión sobre los riesgos del uso inadecuado de patrones son clave para un ejercicio profesional responsable. Reconocer los peligros de su aplicación mecánica o excesiva, así como la importancia de adaptarlos al contexto, previene la complejidad innecesaria y el deterioro de la calidad del software. Además, atender a las tendencias emergentes como patrones para IA, sistemas distribuidos y arquitecturas “cloud-native” es fundamental ante los nuevos desafíos de la disciplina.

Referencias

[1] C. Alexander, S. Ishikawa, y M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[2] E. Gamma, R. Helm, R. Johnson, y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, y M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[4] D. C. Schmidt, «Using design patterns to develop reusable object-oriented communication software», *Communications of the ACM*, vol. 38, n.º 10, pp. 65-74, 1995.

[5] S. Freeman y N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, 2009.

[6] M. Fowler, «Inversion of Control Containers and the Dependency Injection pattern». [En línea]. Disponible en: https://martinfowler.com/articles/injection.html

[7] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[8] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[9] G. Hohpe y B. Woolf, *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[10] M. Fowler, «Event Sourcing». [En línea]. Disponible en: https://martinfowler.com/eaaDev/EventSourcing.html

[11] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[12] A. A. A. Donovan y B. W. Kernighan, *The Go Programming Language*. Addison-Wesley, 2016.

[13] S. Klabnik y C. Nichols, *The Rust Programming Language*. No Starch Press, 2018.

[14] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[15] S. Amershi *et al.*, «Software Engineering for Machine Learning: A Case Study», en *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, IEEE, 2019.

[16] M. Roberts, *Serverless Architectures on AWS: With examples using AWS Lambda*. Manning, 2018.

[17] M. Fowler, «Blue-Green Deployment». [En línea]. Disponible en: https://martinfowler.com/bliki/BlueGreenDeployment.html

[18] R. Kuhn y B. Hanappi, «Reactive Design Patterns for Distributed Systems», en *Reactive Design Patterns*, Manning, 2017.

[19] W. J. Brown, R. C. Malveau, H. W. M. III, y T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[20] J. O. Coplien, «Why patterns fail», *C++ Report*, vol. 9, n.º 5, pp. 38-42, 1997.