Patrones de Diseño en el Desarrollo de Software y QA

De la teoría a la práctica profesional

Creado por "Roberto Arce"

Este libro está basado en experiencias reales y contiene opiniones sobre el ejercicio profesional de la calidad en proyectos de software.
Nombres de productos, empresas o situaciones reales se mencionan únicamente con fines educativos.

Primera edición: 2025
Diseño y estrategia editorial: **QA sin filtros**

Publicado por el autor a través de Amazon Kindle Direct Publishing (KDP)
www.amazon.com/kdp

Prólogo

Como desarrollador, seguro te has enfrentado a problemas de diseño similares en más de una ocasión. Los **patrones de diseño** son colecciones de soluciones reutilizables para esos problemas comunes, pensadas para simplificar la vida del programador. Imagina que al diseñar tu aplicación de chat, reconoces que lo que necesitas es un mecanismo que notifique los cambios a muchos objetos (el clásico **Observer**). En lugar de reinventar la rueda, empleas el patrón adecuado: ganas flexibilidad, claridad y tiempo. Los patrones no son código rígido sino un *lenguaje común*: simplemente con nombrarlos, todos en el equipo entienden la intención. Este libro está pensado para todo tipo de desarrolladores –desde juniors que buscan fundamentos sólidos hasta seniors, arquitectos y líderes técnicos–, porque entender los patrones hará tu código más limpio y tu trabajo más eficaz.

El origen de los patrones de diseño (Gang of Four y más allá)

La idea de «patrones» se originó en la arquitectura: el arquitecto Christopher Alexander describió por primera vez un *lenguaje de patrones* para diseño urbano. Los desarrolladores de software tomaron esa inspiración y la adaptaron al código. En 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides –la famosa "Gang of Four"– publicaron su influyente libro *Design Patterns: Elements of Reusable Object-Oriented Software*, formalizando 23 patrones clásicos. Estos patrones proporcionan soluciones comprobadas que mejoran la estructura y la flexibilidad del código, facilitando que las aplicaciones sean más fáciles de mantener y ampliar. Desde entonces se han descubierto docenas de patrones adicionales, incluso en nuevos paradigmas más allá de la programación orientada a objetos. En este libro repasaremos tanto los patrones clásicos como los más modernos, mostrando cómo cada uno surgió para resolver necesidades reales.

¿Por qué los patrones siguen siendo relevantes en 2025?

Aunque los lenguajes y tecnologías evolucionan, los patrones siguen siendo esenciales **como concepto y vocabulario común**, más que como código literal. Por ejemplo, hoy muchos lenguajes incorporan capacidades internas (los *iteradores* en Java o los *módulos* en Python pueden jugar el papel de un Singleton), pero saber que existe el patrón Singleton es como conocer teoría musical para un músico de jazz: te permite improvisar sobre una estructura. Además, los patrones facilitan la comunicación en un equipo. Imagina decir «implementemos el patrón Strategy aquí»: no es necesario detallar el funcionamiento, porque todos captan la idea de separar comportamientos dinámicos. En resumen, **"los patrones son atemporales; su código no lo es"**.

En la práctica actual, los patrones permanecen vigentes en muchos contextos:

- **Sistemas heredados (legado empresarial):** Muchas aplicaciones corporativas en C++, Java o C# todavía utilizan patrones clásicos de GoF (por ejemplo, fábricas de objetos, proxys u observadores). En entornos así, reconocer un patrón permite modificar o extender la funcionalidad sin romper la estructura existente.
- **Equipos centrados en POO:** En proyectos estrictamente orientados a objetos, patrones como Strategy o Command siguen siendo herramientas clave para separar responsabilidades y encapsular comportamientos. Usarlos ayuda a mantener el código limpio y escalable.
- **Entrevistas y diseño arquitectónico:** Aun si no los programas día a día, los patrones aparecen con frecuencia en entrevistas técnicas y revisiones de arquitectura. Son un lenguaje compartido que permite a desarrolladores y arquitectos discutir soluciones complejas de forma concisa.

En todos estos escenarios, el verdadero valor de los patrones está en el *modelo mental* que aportan. No es que debas memorizar cada fragmento de código, sino entender las **ideas de diseño** que representan. Dominar estos conceptos te convierte en un mejor ingeniero y líder, pues facilitan pensar y comunicar soluciones de forma estructurada.

Conexión entre patrones, arquitectura y calidad de software

Los patrones de diseño se integran naturalmente con la arquitectura del sistema y mejoran la calidad del software. La arquitectura define el esqueleto general del sistema, mientras que los patrones guían la construcción de sus componentes internos. Entender ambos niveles es «vital para software escalable y mantenible. Por ejemplo, el patrón Modelo-Vista-Controlador (MVC) es tanto arquitectónico como de diseño, separando claramente presentación, lógica y datos.

Al aplicar patrones probados, se fomenta la **modularidad y el desacoplo**, lo que redunda en sistemas más robustos. Estudios señalan que el uso adecuado de patrones contribuye a crear software *"robusto, eficiente y confiable"*. Siguiendo patrones estandarizados, el código resulta más limpio y fácil de entender; esto se traduce en menos errores y mayor facilidad de mantenimiento. Un recurso reciente destaca que adoptar buenas prácticas mejora la *"organización del código, la modularidad y la escalabilidad"*, lo que a su vez simplifica la depuración y reduce la deuda técnica. En conjunto, patrones y arquitectura forman la base de software de alta calidad: sistemas que podemos escalar, modificar y extender con confianza, sin desequilibrios ni sobrecargas innecesarias.

Estructura del libro: teoría + implementación + QA

Para aprovechar al máximo cada patrón, este libro combina tres enfoques:

- **Teoría:** Se presenta cada patrón explicando su propósito, motivación y estructura conceptual. Discutimos cuándo usarlo y qué problema soluciona, apoyándonos en analogías y diagramas para hacer la teoría clara.

- **Implementación:** Ofrecemos ejemplos prácticos en código real (por ejemplo, en Java o Python) que ilustran cómo aplicar el patrón paso a paso. Verás casos concretos y buenas prácticas que podrás adaptar a tus proyectos.
- **QA (Garantía de Calidad):** Finalmente, cada sección incluye recomendaciones de calidad: cómo probar que la implementación funciona correctamente, qué métricas o pruebas unitarias usar, y consejos para mantener el patrón sin introducir defectos. De este modo aprenderás no solo a escribir el patrón, sino también a validarlo y documentarlo.

Con esta estructura –conceptos claros, código ejemplar y enfoque en calidad–, el libro te guía de lo abstracto a lo práctico, asegurando que interiorices los patrones y sepas aplicarlos con confianza en el mundo real.

ÍNDICE GENERAL

Una guía avanzada que conecta los patrones de diseño clásicos con la ingeniería de software moderna, el QA automatizado y las arquitecturas escalables de 2025.

  - Propósito y aplicación

- **Abstract Factory**
  - Implementación y ejemplos prácticos

- Patrones creacionales en frameworks de testing
  - *Mocks* e inyección de dependencias
  - Ejemplo en Java (JUnit + Mockito)
  - Ejemplo en Python (pytest + unittest.mock)
  - Ejemplo en C# (NUnit + Moq)

- Cobertura y pruebas unitarias/integración
- **Conclusión:** modularidad y consistencia en la creación de objetos

CAPÍTULO 4: Builder y Prototype

- **Patrón Builder**
  - Creación controlada de objetos complejos

- **Patrón Prototype**
  - Clonación y reutilización de instancias
  - Datos de prueba en entornos QA
  - Buenas prácticas en *test data*
  - Errores comunes y cómo evitarlos

- **Ejemplos prácticos:** generación de datos de prueba dinámicos
- **Conclusión:** patrones para control y flexibilidad en test data

CAPÍTULO 5: Patrones Estructurales — Adapter, Bridge y Composite

- **Adapter:** adaptación de interfaces y compatibilidad entre sistemas
- **Bridge:** separación entre abstracción e implementación
- **Composite:** estructuras jerárquicas reutilizables
- **Ejemplos QA:** integración de diferentes frameworks y entornos de prueba
- **Conclusión:** cómo simplificar sistemas complejos con patrones estructurales

CAPÍTULO 6: Decorator, Facade y Proxy — Aplicaciones Técnicas y QA Automatizada

- **Decorator (Decorador)**
  - Extender comportamiento sin modificar el código base

- **Facade (Fachada)**
  - Simplificar interacciones complejas

- **Proxy**
  - Control y mediación de acceso a recursos

- **Aplicaciones en QA:**
  - *Facade* en el patrón Page Object Model
  - *Proxy* en entornos distribuidos y pruebas de microservicios

- **Conclusión:** patrones para aislar la complejidad en pruebas

CAPÍTULO 7: Patrones de Comportamiento — Observer, Strategy y State

- **Observer (Observador):** notificaciones reactivas
- **Strategy (Estrategia):** selección dinámica de algoritmos
- **State (Estado):** modelar flujos dinámicos
- Aplicaciones en QA:
  - *Strategy* para validaciones dinámicas
  - *State* para simular flujos de usuario y pruebas complejas

- **Conclusión:** patrones para flexibilidad y adaptación en automatización

CAPÍTULO 8: Command, Chain of Responsibility y Template Method

- **Command:** encapsular acciones y permitir reversión
- **Chain of Responsibility:** cadenas de validación
- **Template Method:** reutilización de lógica en tests
- **Aplicaciones QA:**
  - Validaciones JSON y flujos automatizados
  - Reutilización de lógica de pruebas

- **Conclusión:** diseño limpio y extensible para frameworks de QA

CAPÍTULO 9: Mediator, Memento e Interpreter

- **Mediator:** coordinación entre múltiples objetos
- **Memento:** guardar y restaurar estados del sistema
- **Interpreter:** analizar y ejecutar reglas o scripts personalizados
- **Aplicaciones en QA:** orquestación de microservicios, rollback y validaciones DSL
- **Conclusión:** patrones para comunicación, trazabilidad y control

CAPÍTULO 10: Patrones Avanzados y QA

- Patrones modernos e inyección de dependencias
- Patrones de frameworks de testing:
  - **Page Object Model (POM)**
  - **Screenplay Pattern**

- Patrones aplicados a microservicios y testing de APIs
- Antipatrones de diseño y su impacto en DevOps y CI/CD
- **Conclusión:** arquitectura, QA y patrones convergen en la era moderna

ANEXOS – Ejercicios Prácticos con Soluciones

1. Singleton para configuración de pruebas
2. Factory Method para selección de navegadores
3. Abstract Factory para test doubles
4. Builder para datos de prueba
5. Prototype para clonar datos
6. Composite para escenarios BDD
7. Adapter para clientes REST
8. Strategy para validaciones dinámicas
9. State para flujos de login
10. Template Method en tests automatizados
11. Proxy para simular microservicios
12. Decorator para logs de pruebas
13. Observer para notificaciones de test
14. Chain of Responsibility para validaciones JSON
15. Strategy para algoritmos de *retry*
16. State para simulación de pipeline CI/CD
17. Template Method para API testing
18. Composite para escenarios de prueba complejos
19. Adapter para Selenium y Playwright
20. Command para acciones reversibles

Glosario de Patrones y Terminología QA

- **Patrones de Diseño:** definiciones y clasificaciones
- **Terminología QA y Testing:** mocks, fixtures, pipelines, integraciones

BONUS: Recursos Profesionales

- Libros y referencias del *Gang of Four*
- Ejemplos de patrones aplicados en frameworks QA (Selenium, Playwright, Pytest)
- Plantillas de código en Python y Java
- Comunidades y repositorios para seguir practicando

Capítulo 1: Introducción a los patrones de diseño

Definición y propósito

Cuando hablamos de patrones de diseño, no nos referimos a recetas mágicas ni a fórmulas que deban aplicarse de manera ciega. Un patrón es, ante todo, **una solución probada a un problema recurrente en el desarrollo de software**. Imagina que cada proyecto es un camino nuevo, con bifurcaciones y obstáculos. Los patrones son como las señales que dejaron otros viajeros antes que nosotros: no te dicen exactamente cómo caminar, pero te muestran la ruta más segura para no perderte. Su propósito es claro: ayudarte a comunicar, a razonar y a construir software con un lenguaje compartido que trasciende tecnologías y generaciones de programadores.

Beneficios: mantenibilidad, escalabilidad, testabilidad.

El primer beneficio tangible de aplicar patrones es la **mantenibilidad**. Un código estructurado siguiendo patrones se entiende más fácilmente, no solo por quien lo escribió, sino también por quienes se unan al proyecto meses o años después. En vez de tener que descifrar líneas caóticas, los nuevos integrantes encuentran un diseño familiar, casi como llegar a una ciudad bien organizada, con calles señalizadas y barrios definidos.

La **escalabilidad** es otro de los grandes aportes. Cuando tu aplicación crece en usuarios, funcionalidades o integración con otros sistemas, un código sin patrones tiende a quebrarse como una casa sin cimientos sólidos. En cambio, los patrones permiten añadir nuevas piezas sin derribar lo que ya existe, igual que una construcción modular en la que cada bloque encaja con armonía.

Y no menos importante, la **testabilidad**. En tiempos de metodologías ágiles y DevOps, escribir pruebas automatizadas ya no es opcional, es parte esencial del ciclo de vida. Los patrones, al promover separación de responsabilidades y bajo acoplamiento, facilitan que cada módulo pueda ser probado de forma independiente. Esto reduce el riesgo de errores en producción y aumenta la confianza en cada entrega.

Principios SOLID como base de los patrones

Si los patrones son caminos que guían al desarrollador, los principios **SOLID** son el mapa que les da coherencia. Son cinco ideas sencillas, pero profundas, que actúan como cimientos para muchos de los patrones que estudiaremos:

- **S**ingle Responsibility Principle (Responsabilidad Única): cada clase debería tener un solo motivo de cambio. Este principio inspira patrones como *Strategy* o *Observer*, que dividen comportamientos para evitar mezclar lógicas distintas.
- **O**pen/Closed Principle (Abierto/Cerrado): las entidades deben estar abiertas a la extensión, pero cerradas a la modificación. Aquí nacen patrones como *Decorator* o *Factory Method*, que permiten extender funcionalidades sin romper lo existente.
- **L**iskov Substitution Principle: una subclase debe poder reemplazar a su superclase sin alterar el correcto funcionamiento. Los patrones de herencia, como *Template Method*, beben directamente de esta idea.
- **I**nterface Segregation Principle: es mejor muchas interfaces pequeñas que una interfaz gigantesca. Los patrones que promueven desacoplo, como *Adapter* o *Proxy*, están en línea con este principio.
- **D**ependency Inversion Principle: los módulos de alto nivel no deben depender de los de bajo nivel, ambos deben depender de abstracciones. El patrón *Dependency Injection* es la encarnación práctica de este concepto.

En conjunto, SOLID y los patrones se complementan: los principios son las **leyes físicas** que definen el terreno, y los patrones son los **puentes, túneles y carreteras** que construimos sobre ese terreno para llegar más lejos con seguridad.

Relación con arquitecturas modernas (DDD, Clean Architecture, Microservicios)

En arquitecturas modernas como DDD, *Clean Architecture* o microservicios, los patrones clásicos de diseño GoF siguen siendo fundamentales. De hecho, conforman un vocabulario común y ofrecen soluciones probadas que facilitan el desacoplamiento y la flexibilidad del sistema. Por ejemplo, en sistemas event-driven los conceptos del patrón **Observer** son críticos (los componentes reaccionan a eventos), mientras que en integraciones o APIs complejas los patrones **Adapter** y **Facade** sirven de capa de compatibilidad o simplifican subsistemas complejos. Patrones como **Strategy** (algoritmos intercambiables) o **Decorator** (añadir responsabilidades dinámicas) permiten construir sistemas configurables y extensibles, y estructuras como **Chain of Responsibility** ayudan a encadenar procesamientos (por ejemplo, en filtros o pipelines de validación). Incluso en arquitecturas distribuidas y microservicios se aplican patrones de mensajería como **Mediator** o **Command** (por ejemplo, en buses de mensajes o colas) para coordinar operaciones sin acoplar las partes. En resumen, los patrones GoF proveen principios de bajo acoplamiento y alta cohesión que ayudan a diseñar arquitecturas robustas y mantenibles.

En **Domain-Driven Design**, los patrones clásicos se emplean dentro de los *contextos acotados* para modelar el dominio correctamente y separar las capas. A nivel táctico DDD define como bloques de construcción las *Entidades*, *Value Objects*, *Agregados*, *Servicios de Dominio*, *Fábricas* y *Repositorios*. Por ejemplo, **Repositorios** abstraen la persistencia: se define una interfaz de repositorio en el dominio (un repositorio por raíz de agregado) y su implementación concreta en la capa de infraestructura. De este modo el modelo de dominio no conoce detalles de la base de datos, cumpliendo el Principio de Inversión de Dependencias. Microsoft destaca que "el patrón Repositorio es un patrón de DDD destinado a mantener las preocupaciones de persistencia fuera del modelo de dominio". En la práctica, frameworks ORM lo ejemplifican: p.ej. en .NET el DbContext de Entity Framework implementa tanto Repository como Unit of Work.

Las **Fábricas (Factory)** en DDD se utilizan cuando la creación de un objeto de dominio (entidad o agregado) es compleja o requiere datos externos. Por ejemplo, si un agregado incluye datos de otros microservicios (otro contexto), la fábrica encapsula esa lógica de construcción. Radhakrishnan Periyasamy ilustra esto en un microservicio de "carrito de compras": una fábrica *CartFactory* ensambla el agregado `Cart` obteniendo información de un servicio de catálogo sin que el código llamador necesite conocer esos detalles internos. El patrón Factory (junto con Builder) abstrae los detalles internos de cómo se construye un objeto complejo, permitiendo preservar invariantes de dominio.

Otros patrones GoF clásicos también aparecen en DDD. Por ejemplo, **Strategy** se usa para encapsular algoritmos o reglas de negocio variables (p.ej. diferentes estrategias de cálculo de descuentos), **Decorator** permite agregar comportamientos (p. ej. validaciones) a objetos de dominio de forma dinámica, y **Facade** puede emplearse en servicios de dominio para ofrecer interfaces simplificadas a casos de uso complejos. En general, el uso correcto de estos patrones en DDD garantiza separación de responsabilidades: el dominio expone solo lo esencial y delega detalles a capas externas o componentes especializados.

La **Clean Architecture** o arquitecturas hexagonales/onion llevan esta idea al siguiente nivel mediante capas bien separadas y dependencias invertidas. El núcleo del negocio (dominio) es independiente de tecnologías externas y se comunica con el exterior a través de "puertos" (interfaces). Por ejemplo, el patrón **Adapter** encarna esta separación: el dominio define un puerto (p. ej. `IUserRepository`) y se implementa un adaptador concreto en la capa de infraestructura (p.ej. una clase que usa JDBC o un ORM). Como explica el caso de Ports & Adapters, "dentro del core definimos algo llamado *Puerto*, que es un contrato, y los adaptadores implementan ese contrato". Esto aplica también al patrón Repository: el repositorio es el puerto en la capa de dominio, y su implementación (acceso a BD, API externa, etc.) es el adaptador en infraestructura. Clean Architecture alienta a usar patrones creacionales y estructurales para mantener independientes las capas: por ejemplo, **Factory Method** o **Abstract Factory** para crear objetos del dominio sin acoplarse a clases concretas, **Bridge** para desacoplar abstracción e implementación cambiantes, o **Facade** para exponer APIs limpias a la capa de presentación. En esta arquitectura, las dependencias fluyen hacia adentro: el dominio no conoce frameworks externos ni detalles de infraestructura. De este modo se respeta DIP e ISP: cada capa se comunica por medio de contratos mínimos, y el núcleo permanece agnóstico a cambios tecnológicos.

Frameworks modernos reflejan esto: por ejemplo, en Java *Spring Boot* se usa inyección de dependencias y anotaciones (como `@Repository`) para implementar repositorios genéricos, en TypeScript el framework *NestJS* promueve controladores y servicios desacoplados, y en Python librerías como *Django ORM* o *SQLAlchemy* incentivan modelos de dominio independientes del motor de BD. En resumen, Clean Architecture extiende los patrones GoF al diseño de la aplicación completa, asegurando alta cohesión en cada capa y bajo acoplamiento entre capas.

En **microservicios** se emplean tanto patrones de diseño de software clásicos (a nivel de cada servicio) como patrones de arquitectura distribuida. A nivel macro, cada microservicio es una pequeña aplicación independiente (un solo contexto de dominio) con su propia base de datos. Se suele usar un patrón **Database per Service** para evitar acoplamientos por esquema, manteniendo consistencia eventual coordinada vía **Saga** u orquestación. Además, un **API Gateway** actúa como fachada unificada: es un punto de entrada único para todas las llamadas de clientes, lo que permite evolucionar los límites de servicios sin impactar al cliente. Así, el cliente no necesita conocer cuántos servicios hay ni sus endpoints internos. Patrones de resiliencia como **Circuit Breaker** (p.ej. implementados por Netflix Hystrix o Resilience4j) se utilizan para aislar fallos cuando un servicio no responde.

Para la comunicación entre servicios, son comunes patrones asincrónicos: el estilo *event-driven* con colas o brokers (Kafka, RabbitMQ) implica usar el patrón **Publisher-Subscriber/Observer** para notificar cambios de estado. El artículo de Capital One destaca que la mensajería asíncrona desconecta temporalmente los servicios, mejorando la escalabilidad y evitando acoplamiento estricto. Internamente, cada microservicio recurre a patrones GoF clásicos para su lógica local: por ejemplo, un microservicio podría usar **Strategy** para elegir dinámicamente un algoritmo (como distintos proveedores de geolocalización), **Repository** para acceso a datos locales, o **Factory** para crear agregados a partir de datos de otros sistemas. Es importante destacar que cada microservicio sigue el Principio de Responsabilidad Única: "cada microservicio implementa una única responsabilidad de negocio del contexto", lo que hace que sean fáciles de entender y probar. En este sentido, los patrones clásicos dentro de cada servicio colaboran con los patrones de arquitectura (Saga, Gateway, CQRS, etc.) para lograr sistemas modulares y escalables.

La implementación de estos patrones varía según el lenguaje. En **Java** o **C#**, lenguajes estáticos, los patrones se expresan naturalmente con clases e interfaces. Por ejemplo, Spring Data genera repositorios mediante interfaces (`IRepository`) y anotaciones; Entity Framework usa la clase `DbContext` que internamente implementa Repository y Unit of Work. En **TypeScript/JavaScript**, con ES6 se emplean clases o funciones de primer orden: se puede definir una interfaz de estrategia y varias clases (o funciones) que la implementan. En **Python**, al ser dinámico, muchos patrones "pesados" se simplifican: un **Singleton** a menudo se implementa usando una variable de módulo (los módulos en Python son *singleton* por naturaleza), y el *duck typing* permite que cualquier objeto con los métodos apropiados sirva de estrategia. En **C** (lenguaje procedural), los patrones se construyen con `struct` y punteros a función: p.ej. un Strategy puede ser una estructura con un puntero a la función concreta, y un Adapter puede ser un conjunto de funciones wrapper que implementan la interfaz. Cada lenguaje ofrece herramientas propias (frameworks de inyección de dependencias,

ORMs, librerías de concurrencia, etc.) que facilitan o incluso incorporan estos patrones de diseño según las buenas prácticas actuales.

El uso correcto de patrones favorece los principios SOLID y la separación de responsabilidades. Gracias a DIP, componentes de alto nivel (dominio) dependen solo de abstracciones (interfaces) y no de implementaciones concretas. El patrón Repository desacopla la capa de dominio de la persistencia, y Adapter separa la lógica de negocio de detalles tecnológicos, cumpliendo ISP y DIP. Patrones como Strategy o Decorator permiten extender el sistema sin modificar código existente (cumpliendo OCP). La coherencia de contexto (SRP) se mantiene: cada agregado o microservicio tiene una única responsabilidad principal. El resultado es un código modular, fácil de probar y de extender: como señala DigitalOcean, aplicar consistentemente patrones promueve "código más modular, fácil de probar, menos propenso a errores y más sencillo de modificar o extender". En Clean Architecture, por ejemplo, el hecho de que "el núcleo de negocio no conozca nada del mundo exterior" garantiza que las capas permanezcan independientes.

En conclusión, los patrones de diseño GoF no solo siguen siendo útiles, sino esenciales en arquitecturas modernas. En DDD permiten modelar el dominio con claridad (Repositorios, Entidades, Fábricas) y respetar sus invariantes. En Clean Architecture/Hexagonal facilitan la inversión de dependencias y la independencia de capas (puertos/adaptadores, interfaces limpias). En microservicios, se combinan patrones de código (Strategy, Repository, Factory) con patrones distribuidos (Saga, API Gateway, Circuit Breaker, mensajería) para alcanzar un sistema desacoplado y escalable. En cada caso, estos patrones apoyan principios SOLID y la separación de responsabilidades: ayudan a organizar el código en componentes cohesivos con mínimas dependencias externas. Por ello, en la práctica moderna de diseño de software se consideran indispensables tanto en la capa de aplicación como en la de dominio.

Capítulo 2: Clasificación de los patrones de diseño.

Clasificación de Patrones de Diseño

Los patrones de diseño son soluciones probadas a problemas comunes en el diseño de software. Se clasifican en tres grandes grupos según su propósito:

- **Patrones Creacionales:** Abordan la creación de objetos de forma flexible y reutilizable. Permiten abstraer la instanciación (p. ej., para evitar acoplar el código a clases concretas) y controlan situaciones como la creación de objetos complejos o únicos. Ejemplos típicos incluyen *Factory Method*, *Abstract Factory*, *Builder*, *Prototype* y *Singleton*. Estos patrones son útiles cuando el sistema debe ser independiente de cómo se crean, componen y representan sus objetos.
- **Patrones Estructurales:** Se centran en cómo ensamblar clases y objetos para formar estructuras más grandes manteniendo flexibilidad y eficiencia. Resuelven problemas de compatibilidad de interfaces y composición de objetos. Algunos patrones estructurales clásicos son *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* y *Proxy*. Se emplean para desacoplar módulos (p. ej. adaptando una interfaz) o para optimizar recursos compartiendo estado (Flyweight).
- **Patrones de Comportamiento:** Gestionan la interacción y asignación de responsabilidades entre objetos. Organizan algoritmos y flujos de control (por ejemplo, definir cómo un objeto realiza una tarea o notifica cambios). Entre los más usados figuran *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template Method* y *Visitor*. Sirven para reducir dependencias directas entre objetos (p. ej. con un mediador) o para permitir la extensión del comportamiento en tiempo de ejecución (p. ej. estrategia intercambiable).

Estos patrones aportan ventajas como la reutilización de soluciones contrastadas y un lenguaje común que mejora la comunicación del equipo. En general **ahorran tiempo** al evitar reinventar algoritmos básicos, hacen el código más **fácil de entender** y **modificar**, y promueven arquitecturas limpias y modulares.

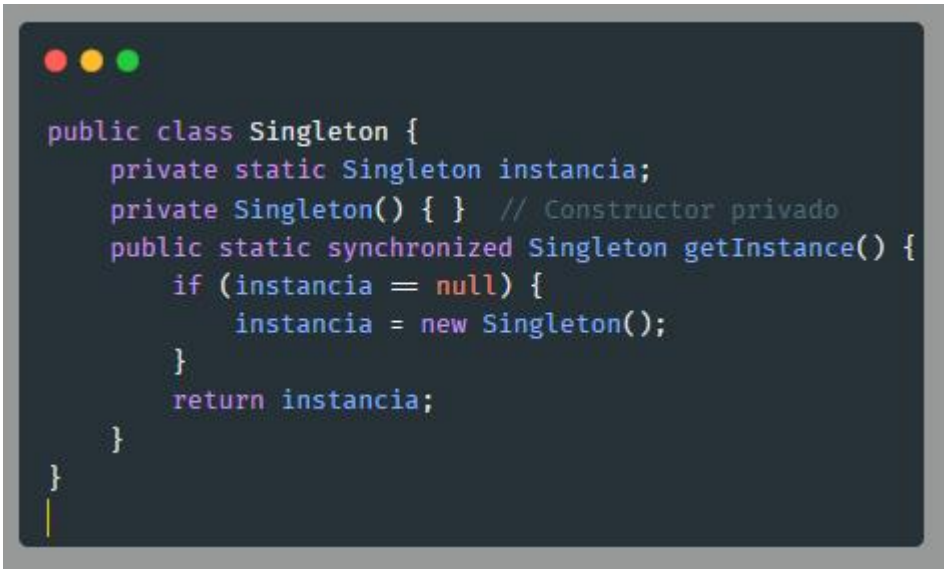
Patrones Creacionales

Los patrones creacionales abstraen la creación de objetos para **incrementar la flexibilidad**. Por ejemplo, un *Factory Method* define una interfaz para crear objetos pero permite que las subclases elijan la clase concreta. Un *Builder* separa la construcción de un objeto complejo en pasos independientes. El *Singleton* garantiza una única instancia global (útil para gestores de recursos como conexiones de BD).

Patrones principales: Factory Method, Abstract Factory, Builder, Prototype, Singleton.

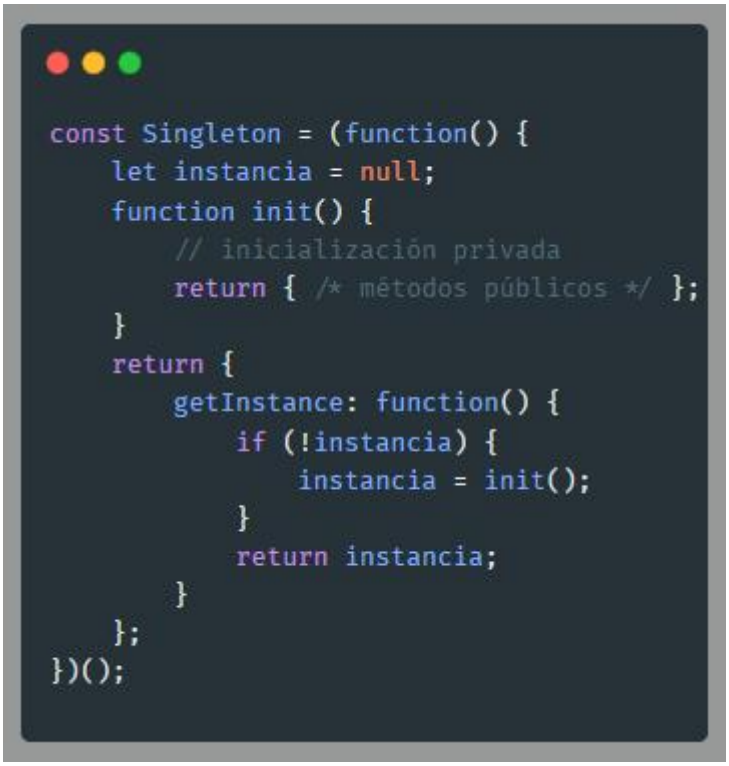
Ejemplo (Singleton): Asegura un único objeto global.

Java



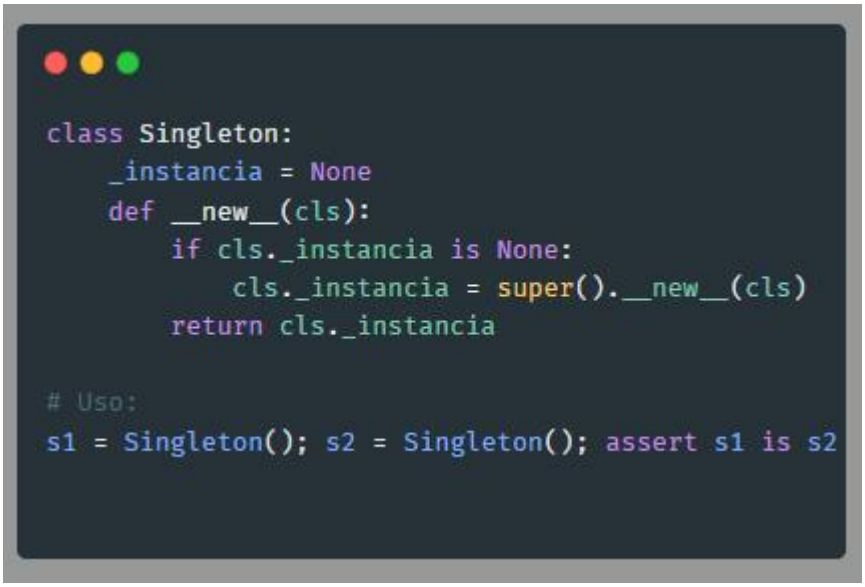
```java
public class Singleton {
    private static Singleton instancia;
    private Singleton() { }  // Constructor privado
    public static synchronized Singleton getInstance() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }
}
```

JavaScript



```javascript
const Singleton = (function() {
    let instancia = null;
    function init() {
        // inicialización privada
        return { /* métodos públicos */ };
    }
    return {
        getInstance: function() {
            if (!instancia) {
                instancia = init();
            }
            return instancia;
        }
    };
})();
```

Python:



```python
class Singleton:
    _instancia = None
    def __new__(cls):
        if cls._instancia is None:
            cls._instancia = super().__new__(cls)
        return cls._instancia

# Uso:
s1 = Singleton(); s2 = Singleton(); assert s1 is s2
```

C:

```c
#include <stdlib.h>
typedef struct { /* datos */ } Singleton;
Singleton* getInstance() {
    static Singleton* instancia = NULL;
    if (instancia == NULL) {
        instancia = malloc(sizeof(Singleton));
        // inicializar datos
    }
    return instancia;
}
// Uso:
// Singleton* s = getInstance();
```

Ventajas y casos de uso: Los patrones creacionales permiten cumplir el Principio de Abierto/Cerrado (OCP) al poder ampliar la creación de objetos sin modificar código existente. Facilitan la inyección de dependencias (mediante fábricas abstractas) cumpliendo el Principio de Inversión de Dependencias. Se emplean en la implementación de frameworks, gestores de recursos (singleton), creación de familias de objetos relacionados (abstract factory) o en la generación paso a paso de instancias complejas (builder).

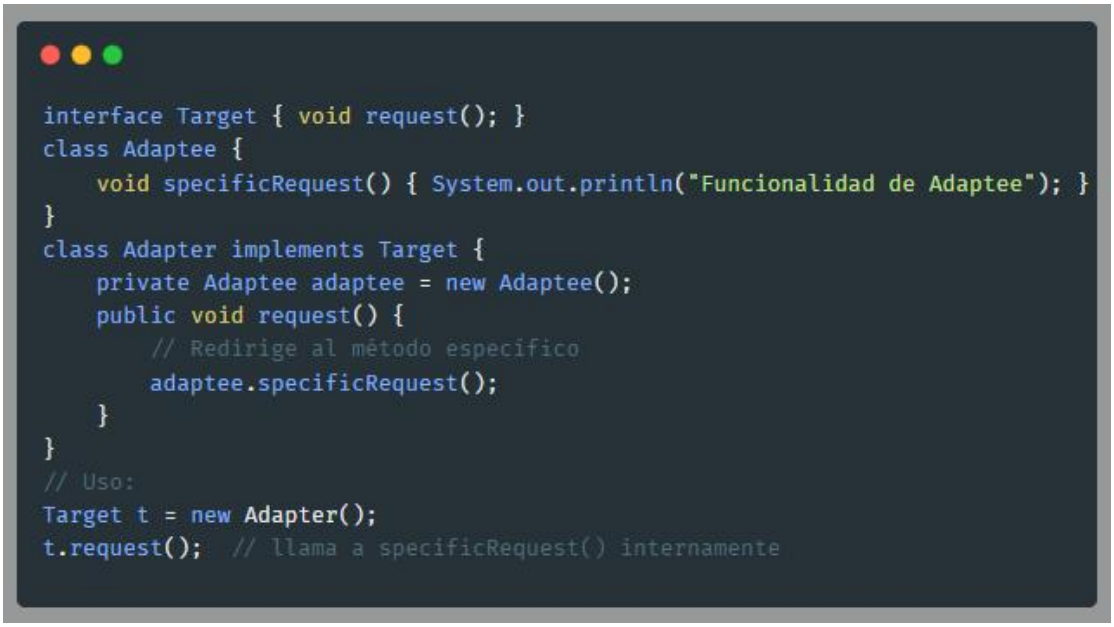
Patrones Estructurales

Los patrones estructurales **organizan y unen objetos** para crear sistemas más grandes, manteniendo bajo acoplamiento. Por ejemplo, el *Adapter* permite que dos clases con interfaces incompatibles trabajen juntas: adapta la interfaz de un objeto existente para que el cliente lo utilice sin cambios en el cliente. Un *Facade* ofrece una interfaz simplificada a un subsistema complejo. Un *Decorator* agrega responsabilidades adicionales a un objeto de forma dinámica. Un *Flyweight* comparte estado común entre muchos objetos para ahorrar memoria.

Patrones principales: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

Ejemplo (Adapter): Traductor entre interfaces distintas.

Java



```java
interface Target { void request(); }
class Adaptee {
    void specificRequest() { System.out.println("Funcionalidad de Adaptee"); }
}
class Adapter implements Target {
    private Adaptee adaptee = new Adaptee();
    public void request() {
        // Redirige al método especifico
        adaptee.specificRequest();
    }
}
// Uso:
Target t = new Adapter();
t.request();  // llama a specificRequest() internamente
```

JavaScript

```javascript
class Adaptee {
    specificRequest() {
        console.log("Funcionalidad específica de Adaptee");
    }
}
class Adapter {
    constructor(adaptee) { this.adaptee = adaptee; }
    request() { this.adaptee.specificRequest(); }
}
// Uso:
const adaptee = new Adaptee();
const adapter = new Adapter(adaptee);
adapter.request();
```

Python

```python
class Adaptee:
    def specific_request(self):
        print("Función específica del adaptee")
class Adapter:
    def __init__(self, adaptee):
        self.adaptee = adaptee
    def request(self):
        return self.adaptee.specific_request()
# Uso:
adaptee = Adaptee()
adapter = Adapter(adaptee)
adapter.request()
```

C

```c
#include <stdio.h>
typedef struct { /* datos */ } Adaptee;
void specificRequest(Adaptee* a) {
    printf("Función específica del Adaptee\n");
}
typedef struct { Adaptee* adaptee; } Adapter;
void request(Adapter* obj) {
    specificRequest(obj->adaptee);
}
// Uso:
Adaptee a;
Adapter adapter = { &a };
request(&adapter);
```

Ventajas y casos de uso: Los patrones estructurales reducen el acoplamiento y favorecen la **reutilización**. El *Adapter* y el *Facade* permiten integrar librerías o módulos externos sin alterar el código cliente, cumpliendo el Principio de Inversión de Dependencias. El *Decorator* y *Bridge* siguen el Principio de Responsabilidad Única al agregar funcionalidades de manera separada. El *Composite* facilita el procesamiento recursivo de estructuras tipo árbol (p. ej. sistemas de archivos, interfaces gráficas). En arquitecturas modernas, estos patrones ayudan a montar módulos independientes (por ejemplo, adaptando servicios en microservicios) respetando la **modularidad** y la **escala** del sistema.

Patrones de Comportamiento

Los patrones de comportamiento definen **cómo interactúan los objetos**, organizando flujos de control y colaboración. Por ejemplo, el *Observer* implementa un mecanismo de suscripción para notificar a varios objetos (observadores) cuando cambia el estado de otro (sujeto). El *Strategy* encapsula algoritmos intercambiables, cumpliendo OCP al permitir cambiar el comportamiento en tiempo de ejecución. El *Command* convierte acciones en objetos, facilitando la parametrización y manejo de ejecuciones diferidas. El *Iterator* separa la iteración de una colección de su representación interna.

Patrones principales: Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Ejemplo (Observer): Varios “observadores” reciben notificaciones de un cambio.

Java

```java
import java.util.*;
// Sujeto que notifica
class Subject {
    private List<Observer> obs = new ArrayList<>();
    void attach(Observer o) { obs.add(o); }
    void notifyAllObservers() {
        for(Observer o : obs) o.update();
    }
}
interface Observer { void update(); }
// Uso:
Subject s = new Subject();
s.attach(() -> System.out.println("Notificado!"));
s.notifyAllObservers();
```

JavaScript

```javascript
class Subject {
    constructor() { this.observers = []; }
    subscribe(o) { this.observers.push(o); }
    notify() { this.observers.forEach(o => o.update()); }
}
class Observer {
    update() { console.log("Notificado"); }
}
// Uso:
const sub = new Subject();
sub.subscribe(new Observer());
sub.notify();
```

Python

```python
class Subject:
    def __init__(self): self._observers = []
    def subscribe(self, obs): self._observers.append(obs)
    def notify(self):
        for o in self._observers: o.update()
class Observer:
    def update(self): print("Notificado")
# Uso:
subj = Subject()
subj.subscribe(Observer())
subj.notify()
```

C

```c
#include <stdio.h>
typedef void (*Observer)(void*);
typedef struct {
    Observer observers[10];
    int count;
} Subject;
void subscribe(Subject* s, Observer o) {
    s->observers[s->count++] = o;
}
void notifyAll(Subject* s) {
    for(int i=0; i<s->count; i++) {
        s->observers[i](s);
    }
}
void onNotify(void* s) {
    printf("Notificado\n");
}
// Uso:
Subject subj = { .count = 0 };
subscribe(&subj, onNotify);
notifyAll(&subj);
```

Ventajas y casos de uso: Los patrones de comportamiento fomentan la **baja dependencia** y la **extensibilidad**. Por ejemplo, *Observer* desacopla el emisor y los receptores de eventos (ideal en interfaces gráficas o sistemas reactivos). *State* y *Strategy* permiten añadir nuevos comportamientos sin modificar clases existentes (principio OCP). *Command* y *Mediator* ayudan a implementar inyección de dependencias y coordinar subsistemas complejos. En arquitecturas contemporáneas (microservicios, DDD, etc.), estos patrones refuerzan principios SOLID. En especial, el principio de inversión de dependencias (DIP) se logra implementando abstracciones (por ejemplo, mediante adaptadores o fachadas) y luego inyectando implementaciones concretas.

En conjunto, la clasificación de patrones de diseño ayuda a aplicar buenas prácticas en arquitecturas modernas. Los patrones brindan soluciones reutilizables y un vocabulario común, y su uso consciente refuerza los principios SOLID que rigen el diseño modular. Por ejemplo, el *Open/Closed Principle* se apoya en patrones que permiten extender funcionalidad sin cambiar código existente, y el *Dependency Inversion* se logra con patrones como la Fábrica abstracta, el Adaptador o la Fachada. En resumen, emplear patrones creacionales, estructurales y de comportamiento mejora la **flexibilidad**, **mantenibilidad** y **escalabilidad** de sistemas complejos en entornos modernos.