

Unidad 4:

Optimización, patrones y buenas prácticas

Entornos de desarrollo

Técnico Superior de DAM / DAW



Sesión 11: Patrones de diseño (creacionales y estructurales). Soluciones reutilizables para un código más sólido

En el desarrollo de software, uno de los mayores retos es cómo escribir código que pueda crecer, mantenerse y adaptarse sin romperse. Con el tiempo, los programadores observaron que muchos problemas de diseño se repetían una y otra vez. La respuesta a esos problemas recurrentes fueron los patrones de diseño: soluciones probadas y estructuradas que no dictan código exacto, sino modelos conceptuales que guían cómo construir arquitecturas limpias y escalables.

Un patrón de diseño no es una receta cerrada, sino un enfoque reutilizable que puedes adaptar al contexto de tu proyecto. Por ejemplo, cuando necesitas que solo exista una instancia de una clase (como una conexión de base de datos), o cuando quieres crear objetos sin acoplar el código a clases específicas, los patrones te ofrecen una forma estandarizada de resolverlo.

Creacionales

Se centran en el cómo se crean los objetos. Su objetivo es abstraer y controlar el proceso de creación para evitar dependencias rígidas. Entre los más conocidos encontramos:

- **Singleton:** garantiza que exista solo una instancia de una clase y proporciona un punto de acceso global a ella.
- **Factory Method / Abstract Factory:** encapsula la creación de objetos, permitiendo cambiar las clases concretas sin modificar el código cliente.
- **Builder:** separa la construcción compleja de un objeto de su representación final, facilitando la creación paso a paso.

Estructurales

Tratan sobre cómo se relacionan las clases y objetos entre sí para formar estructuras más grandes y flexibles.

- **Adapter:** actúa como un traductor entre clases con interfaces incompatibles.
- **Composite:** organiza objetos en jerarquías (por ejemplo, menús, carpetas, árboles) de forma que el cliente pueda tratarlos de manera uniforme.
- **Decorator:** permite añadir responsabilidades a los objetos de forma dinámica sin modificar su código original.

La utilidad de los patrones radica en tres principios clave de la ingeniería del software moderna:

Desacoplamiento

El código no debe depender de implementaciones concretas, sino de abstracciones.

Reutilización

Un diseño bien pensado se puede aplicar en múltiples contextos sin reinventar la rueda.

Mantenibilidad

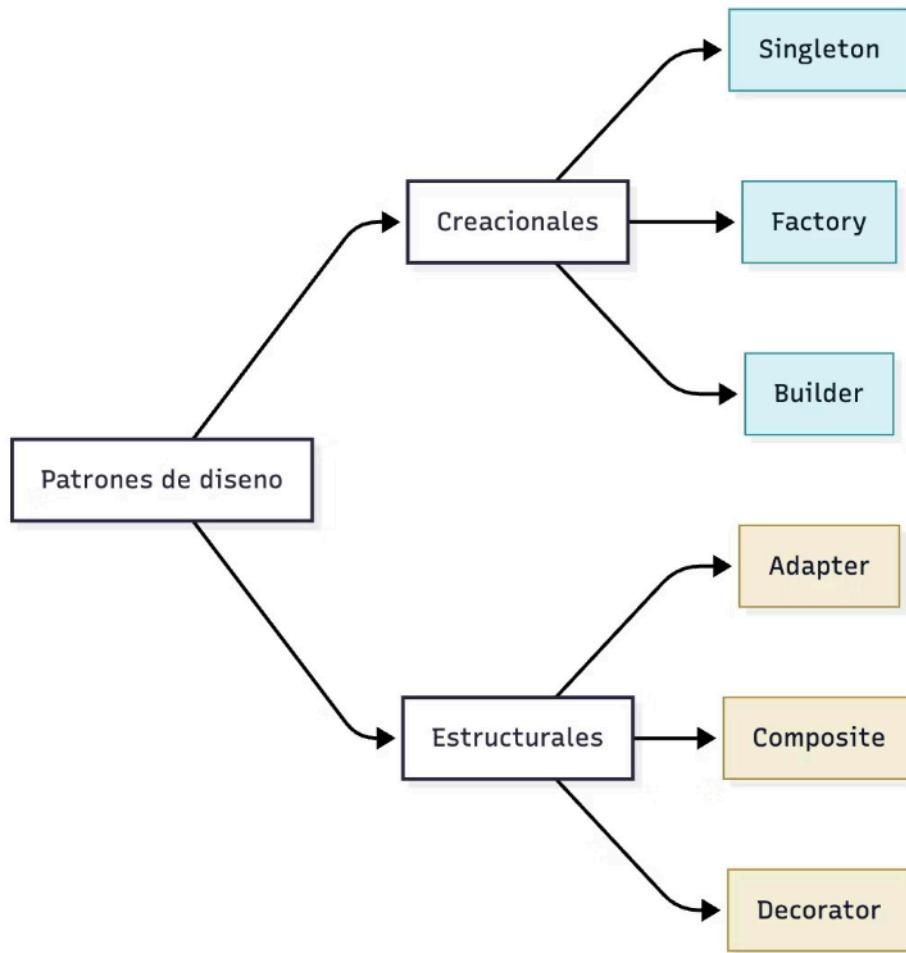
Los patrones facilitan que otros desarrolladores entiendan y amplíen el código sin romper su estructura.

En definitiva, los patrones de diseño son el vocabulario común entre desarrolladores. Usarlos no es "saberse los nombres", sino pensar en términos de arquitectura, anticipando cómo el software crecerá con el tiempo. Aplicar patrones creacionales y estructurales es esencial para diseñar sistemas robustos, modulares y preparados para el cambio.



Esquema Visual: Clasificación de Patrones Creacionales y Estructurales

A continuación se muestra el esquema conceptual que representa las dos familias de patrones que estudiamos en esta sesión.



Interpretación del esquema:

- El nodo central representa el concepto general de patrones de diseño.
- Se divide en dos ramas principales: los Creacionales, que controlan la creación de objetos, y los Estructurales, que definen las relaciones entre ellos.

- Los patrones creacionales (Singleton, Factory, Builder) se enfocan en cómo instanciar objetos de forma flexible.
- Los patrones estructurales (Adapter, Composite, Decorator) se centran en cómo organizar e interconectar clases y objetos para aumentar la modularidad y reutilización.

Este mapa visual permite comprender de un vistazo cómo los distintos patrones se relacionan jerárquicamente y cuál es su propósito dentro de la arquitectura de software.



Caso de Estudio: Netflix y el uso de patrones para escalar su arquitectura

Contexto

Netflix gestiona una de las plataformas tecnológicas más complejas del mundo: millones de usuarios acceden cada segundo desde miles de dispositivos diferentes (smart TVs, móviles, navegadores, consolas). Para ofrecer una experiencia consistente, su arquitectura debe ser altamente modular y adaptable, permitiendo reemplazar o modificar servicios sin afectar el sistema completo.

Estrategia: Aplicación de patrones creacionales y estructurales

Factory Pattern para instancias de servicios

Netflix utiliza el patrón Factory para crear instancias de distintos servicios (video streaming, subtítulos, recomendaciones, autenticación, etc.) según el contexto del usuario o dispositivo.

En lugar de codificar la creación de cada servicio de forma rígida, la fábrica selecciona automáticamente la implementación adecuada. Por ejemplo:

- Un usuario en España puede recibir una versión de recomendación con metadatos regionales.
- Un televisor Samsung y un iPhone usan distintos módulos de reproducción, pero el cliente los invoca del mismo modo.

Esto reduce el acoplamiento y permite añadir nuevos servicios sin alterar el código del cliente.

Adapter Pattern para compatibilidad con dispositivos

Cada fabricante (Sony, LG, Apple, Amazon) tiene APIs diferentes para integrar Netflix. Para evitar reescribir todo, se aplicó el patrón Adapter, que actúa como traductor entre la API de Netflix y la del dispositivo.

El resultado es una capa intermedia que convierte llamadas genéricas (play, pause, resume) en comandos específicos del hardware.

Decorator Pattern para funcionalidades opcionales

Algunas versiones de la app ofrecen funciones adicionales (descargas, perfiles infantiles, etc.). En lugar de crear una clase nueva para cada combinación, Netflix aplica el patrón Decorator, que añade funcionalidades envolviendo objetos existentes, sin modificar su código base.

Resultado

Gracias a estos patrones:

- Netflix mantiene una arquitectura escalable, desacoplada y adaptable.
- Puede desplegar nuevas versiones de la app o servicios en distintas regiones con un mínimo de riesgo.
- La modularidad le permite realizar pruebas A/B sin afectar al sistema principal.

En palabras de un arquitecto de Netflix: "No se trata de escribir más código, sino de escribir menos código mejor organizado." Los patrones creacionales y estructurales son la base invisible que permite esa organización.



Herramientas y Consejos para tu Futuro Profesional



Usa Singleton solo cuando sea estrictamente necesario

Es útil para objetos que deben ser únicos (como un registro de logs, un gestor de configuración o una conexión a base de datos). Pero úsalo con precaución: un exceso de Singeltons puede generar dependencias ocultas y dificultar las pruebas unitarias.



Implementa Factory o Builder para separar la creación de objetos

Factory: cuando necesitas delegar la decisión de qué clase instanciar.

👉 Ejemplo: `ShapeFactory.create("circle")` devuelve un objeto Circle sin que el cliente sepa su implementación.

Builder: cuando los objetos tienen muchos parámetros opcionales.

👉 Ejemplo: `CarBuilder.setEngine("diesel").setColor("blue").build()`.



Adapter para integración con APIs externas o sistemas heredados

Si trabajas en una empresa que usa distintos proveedores (pagos, mapas, CRM), el patrón Adapter permite "traducir" sus interfaces a la estructura interna sin reescribir todo el sistema.



Decorator para ampliar funcionalidades sin herencia

Es ideal cuando necesitas añadir características dinámicas (por ejemplo, validaciones extra en un formulario o log de auditoría) sin duplicar clases.

Herramientas recomendadas para practicar patrones:

- **Refactoring.Guru** → una de las mejores webs para visualizar patrones con ejemplos en múltiples lenguajes.
- **PlantUML** → genera diagramas UML de patrones (muy útil para documentar).
- **Visual Paradigm** → software profesional para diseñar arquitecturas con patrones.
- **GitHub repos "Design Patterns in Java/Python/C#"** → código abierto para aprender implementaciones reales.

❑ Consejo final

No memorices patrones: entiende los problemas que resuelven. Pregúntate siempre: "¿Qué pasaría si mañana cambio esta clase? ¿Tendría que reescribir todo?" Si la respuesta es sí, probablemente necesites aplicar un patrón de diseño.



Mitos y Realidades

 Mito 1: "Los patrones de diseño son solo para expertos."

→ **FALSO.** Los patrones no son un lujo teórico para arquitectos senior, sino herramientas prácticas que cualquier programador puede aprender. De hecho, aplicarlos desde etapas tempranas evita errores costosos. Un junior que entiende un patrón Factory o Adapter ya está escribiendo código más profesional y sostenible.

 Mito 2: "Usar muchos patrones hace el código más elegante."

→ **FALSO.** El abuso de patrones genera sobreingeniería. La clave está en la simplicidad: usar el patrón cuando resuelve un problema real, no para presumir de complejidad. Un buen desarrollador conoce los patrones, pero mejor aún, sabe cuándo no aplicarlos.

Resumen Final

- Patrones de diseño = soluciones reutilizables a problemas comunes de arquitectura.
- Creacionales → controlan la creación de objetos (ej. Singleton, Factory, Builder).
- Estructurales → organizan clases y dependencias (ej. Adapter, Composite, Decorator).
- Mejoran la modularidad, la escalabilidad y el mantenimiento del software.
- Netflix es un ejemplo real de cómo los patrones permiten construir un sistema global adaptable.

Sesión 12: Patrones de comportamiento y ejemplos en apps modernas

Cómo los objetos cooperan para construir software inteligente

Los patrones de comportamiento son uno de los pilares fundamentales del diseño orientado a objetos. Mientras que los patrones estructurales definen cómo se organizan las clases y los patrones creacionales se centran en cómo se generan los objetos, los patrones de comportamiento responden a una pregunta clave: ¿cómo interactúan los objetos entre sí para lograr un objetivo común sin depender rígidamente unos de otros?

En la práctica, estos patrones buscan lograr una comunicación flexible, escalable y mantenible entre los componentes de una aplicación. En lugar de que cada clase conozca los detalles internos de las demás, se establecen relaciones bien definidas que permiten que el sistema evolucione sin romper su estructura. Esto es lo que diferencia a una aplicación "de laboratorio" de una aplicación moderna, como Instagram o Netflix, que deben responder a miles de eventos por segundo sin colapsar.

Observer (Observador)

Permite que un objeto (el "sujeto") notifique automáticamente a otros objetos ("observadores") cuando ocurre un cambio. Este patrón se usa constantemente en interfaces gráficas, sistemas de eventos o aplicaciones con notificaciones. Su gran ventaja es que el sujeto no necesita saber cuántos observadores tiene ni cómo funcionan: solo emite el aviso.

Ejemplo: en una app de mensajería, cuando un usuario cambia su estado, todos sus contactos ven el cambio instantáneamente.

Strategy (Estrategia)

Encapsula distintos algoritmos o comportamientos intercambiables, de forma que un objeto pueda cambiar su manera de actuar en tiempo de ejecución sin modificar su estructura interna.

Ejemplo: un comparador de precios que puede ordenar resultados por precio, popularidad o distancia según la preferencia del usuario.

Command (Comando)

Encapsula una acción o petición como un objeto independiente. Así, las operaciones pueden ejecutarse, deshacerse o almacenarse fácilmente.

Ejemplo: en un editor de texto, cada acción (copiar, pegar, deshacer, rehacer) se guarda como un comando ejecutable o reversible.

State (Estado)

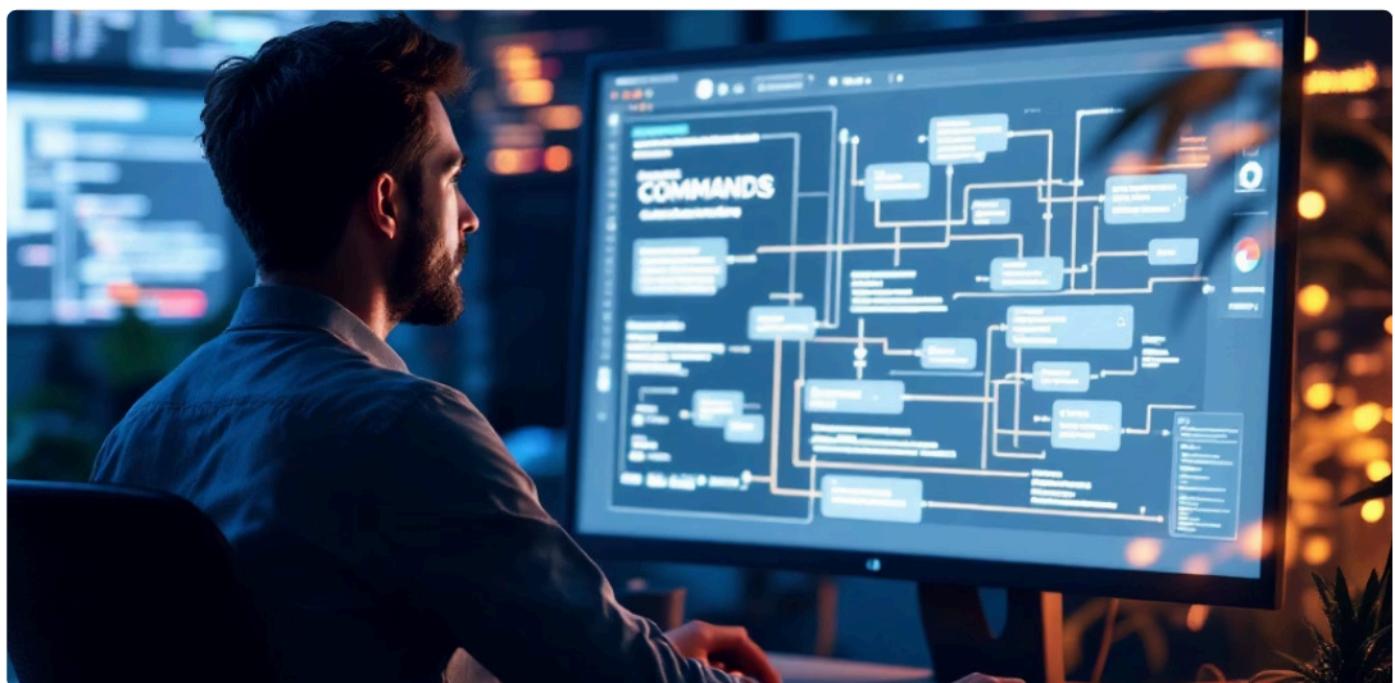
Permite que un objeto cambie su comportamiento según su estado interno sin recurrir a múltiples condicionales.

Ejemplo: un pedido en una app de delivery no se comporta igual si está "en preparación", "en camino" o "entregado".

El valor de estos patrones no está en su complejidad, sino en su capacidad para reducir el acoplamiento entre clases y aumentar la cohesión interna. Cada objeto se concentra en su responsabilidad y delega la coordinación en estructuras bien definidas. Gracias a ello, las aplicaciones modernas —especialmente las basadas en eventos, notificaciones o interfaces gráficas— pueden crecer sin convertirse en un caos de dependencias.

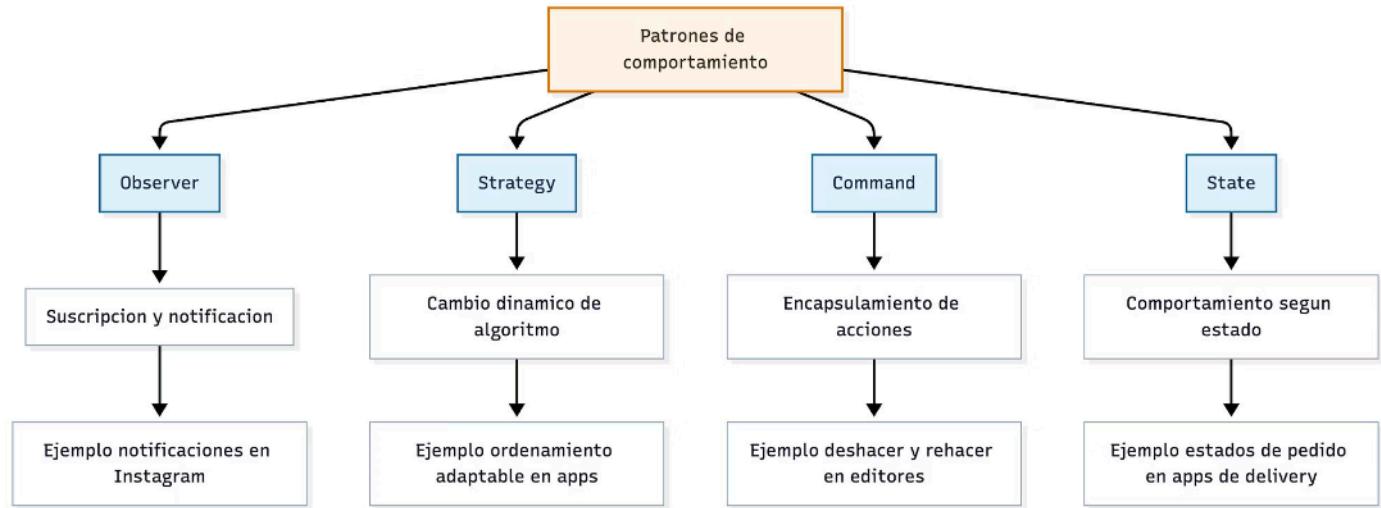
En el contexto actual del desarrollo profesional, dominar estos patrones es esencial. Frameworks populares como React, Angular, SwiftUI o Django los implementan de forma implícita: el data binding o la actualización automática de vistas son ejemplos directos del patrón Observer; los controladores que gestionan acciones de usuario funcionan con Command; y los motores de decisión que cambian la lógica según un modo (por ejemplo, "tema oscuro" o "tema claro") se basan en State.

En resumen, los patrones de comportamiento son la gramática invisible que organiza la comunicación entre objetos, garantizando que el software no solo funcione, sino que sea sostenible, ampliable y preparado para el cambio constante del entorno tecnológico.



Esquema Visual: Los cuatro pilares del comportamiento dinámico

A continuación se describe un esquema conceptual que representa cómo interactúan los principales patrones de comportamiento en un sistema orientado a objetos.



Descripción del esquema:

En el nodo central se encuentran los Patrones de Comportamiento, que conectan con cuatro ramas principales:

- **Observer:** gestiona relaciones de tipo "uno a muchos". Un objeto (sujeto) notifica a múltiples observadores. Es el núcleo de los sistemas de notificaciones.
- **Strategy:** permite intercambiar algoritmos sin modificar las clases que los usan, ideal para aplicaciones que personalizan su comportamiento.

- **Command:** encapsula una acción para ejecutarla, almacenarla o deshacerla. Es el patrón detrás de cualquier menú o historial de operaciones.
- **State:** representa comportamientos cambiantes según el estado interno de un objeto, simplificando la lógica condicional.

Cada rama conecta con un ejemplo práctico de aplicación moderna que ilustra cómo los patrones están presentes en el software cotidiano.



Caso de Estudio: Instagram y el poder del patrón Observer

Contexto:

Instagram es una red social que gestiona millones de interacciones por minuto: fotos publicadas, likes, comentarios, notificaciones y mensajes. Para que la experiencia sea fluida y los usuarios reciban actualizaciones en tiempo real, la aplicación necesita un sistema que observe los cambios y los difunda sin que el servidor tenga que conocer a cada usuario concreto.

Estrategia:

La solución se basa en el patrón Observer. Cuando un usuario publica una foto, ese evento desencadena una notificación automática a todos los seguidores suscritos a su cuenta. Instagram actúa como el "sujeto" (publisher) y los seguidores son los "observadores" (subscribers).

Publicación del contenido

El usuario A (sujeto) publica una nueva foto.

Disparo del evento

El sistema dispara un evento: "nueva publicación".

Notificación a observadores

Los observadores (seguidores) que están suscritos reciben la notificación, sin que A tenga que saber quiénes son.

Reacción personalizada

Cada observador decide cómo reaccionar (mostrar la publicación, generar una alerta, o ignorarla).

Este patrón permite que Instagram escale de forma masiva. Si mañana un usuario tiene 10 millones de seguidores, el sistema no se colapsa porque el código no cambia: simplemente hay más observadores en la lista. Además, el patrón facilita la integración con otros servicios, como el envío de notificaciones push o la sincronización con la API de Facebook.

Resultado:

Gracias al uso del patrón Observer, Instagram mantiene un sistema de actualizaciones en tiempo real, eficiente y escalable. El mismo modelo se aplica en sus notificaciones de likes, comentarios y mensajes directos, lo que demuestra cómo un patrón clásico de diseño sigue siendo el corazón de una de las plataformas más modernas del mundo.



Herramientas y Consejos para tu Futuro Profesional



Aplica el patrón Observer en sistemas de eventos y mensajería

En cualquier aplicación donde haya que reaccionar a cambios (por ejemplo, actualizaciones en una base de datos o eventos de usuario), Observer es la mejor opción. En frameworks como React o Vue, los sistemas de binding de datos siguen este principio.



Usa Strategy para intercambiar algoritmos sin romper el código

Si tu programa necesita realizar una tarea con varios métodos posibles (por ejemplo, diferentes algoritmos de recomendación o de cálculo de precios), encapsula cada uno como una estrategia. Puedes implementarlo fácilmente en Python, Java o C# mediante interfaces o funciones de orden superior.



Implementa Command para manejar acciones reversibles o programables

Ideal en interfaces de usuario: menús, botones o atajos. En frameworks como Swing, Qt o Electron, cada acción del usuario puede representarse como un comando independiente que luego puede deshacerse (undo) o repetirse (redo).



State: el patrón de los flujos dinámicos

Si desarrollas una app con múltiples estados —por ejemplo, un pedido o una máquina de estados en videojuegos—, implementa este patrón para evitar estructuras de control complejas. Cada estado se convierte en una clase que define su propio comportamiento.



Visualiza las relaciones con UML

Herramientas como Lucidchart, Draw.io, PlantUML o incluso Mermaid te ayudarán a representar gráficamente las relaciones entre objetos y las dependencias de cada patrón. Comprenderlo visualmente te hará más eficaz al implementarlo.



Practica en entornos reales

Simula la arquitectura de una red social o un gestor de tareas con notificaciones en tiempo real. Es una excelente forma de consolidar estos patrones y demostrar habilidades de arquitectura de software en tu portfolio profesional.

Mitos y Realidades: Patrones de Comportamiento

 Mito: "Los patrones de comportamiento son difíciles de implementar."

 **Realidad:** En realidad, su poder reside en la simplicidad. Una vez comprendes la relación entre clases (quién observa, quién ejecuta, quién cambia de estado), su implementación resulta intuitiva. La complejidad viene más de su comprensión conceptual que de su codificación.

 Mito: "Solo se usan en proyectos grandes o empresariales."

 **Realidad:** Los patrones de comportamiento están presentes incluso en aplicaciones pequeñas. Por ejemplo, una app de notas que guarda el historial de cambios usa Command; un temporizador que cambia su interfaz cuando está en pausa aplica State; y un formulario web que notifica cambios al backend sigue el modelo Observer.

Resumen Final

- Patrones de comportamiento: definen cómo colaboran los objetos en un sistema.
- Observer: notifica automáticamente a los suscriptores ante cambios.
- Strategy: intercambia algoritmos sin modificar el código base.
- Command: encapsula acciones como objetos, permitiendo deshacer o repetir.
- State: cambia el comportamiento según el estado interno del objeto.
- Aplicación práctica: Instagram usa Observer para notificar a sus seguidores de nuevas publicaciones.
- Clave profesional: promueven código flexible, escalable y fácil de mantener.



Sesión 13: Principios SOLID, cohesión y acoplamiento

La base del código limpio y mantenible

Cuando un proyecto de software crece, mantenerlo se vuelve un desafío. Las clases empiezan a hacer más de lo que deberían, las dependencias se entrelazan y modificar una parte rompe otras. En ese punto, lo que al principio era un código "que funciona" se convierte en un código difícil de escalar, probar o entender. Para evitar esto, el ingeniero Robert C. Martin (conocido como "Uncle Bob") formuló un conjunto de principios que guían la escritura de software limpio, modular y sostenible: los principios SOLID.

Cada letra de SOLID representa una regla fundamental que, aplicada correctamente, mejora la arquitectura y reduce el acoplamiento:

S – Single Responsibility Principle (Responsabilidad Única)

Cada clase o módulo debe tener una sola razón para cambiar. Esto significa que cada componente del sistema debe encargarse de una única tarea concreta. Por ejemplo, si una clase gestiona usuarios, no debería también manejar la base de datos o enviar correos. Separar responsabilidades mejora la claridad y facilita la reutilización.

O – Open/Closed Principle (Abierto/Cerrado)

El código debe estar abierto a la extensión pero cerrado a la modificación. Es decir, si queremos añadir una nueva funcionalidad, debemos hacerlo sin tocar el código existente. Por ejemplo, en lugar de modificar una clase base, se puede extender mediante herencia o composición. Este principio previene errores al alterar funcionalidades ya probadas.

L – Liskov Substitution Principle (Sustitución de Liskov)

Las clases hijas deben poder sustituir a sus clases padre sin alterar el comportamiento del programa. Si una subclase no cumple con las expectativas de la clase base, rompe la lógica del sistema. Este principio promueve una jerarquía coherente y el uso correcto de la herencia.

I – Interface Segregation Principle (Segregación de Interfaces)

Es mejor tener interfaces pequeñas y específicas que una grande y genérica. Obligar a una clase a implementar métodos que no necesita es una mala práctica. Por ejemplo, en lugar de una interfaz "Animal" con métodos volar(), nadar() y correr(), es mejor tener interfaces más precisas como "Volador", "Nadador" o "Corredor".

D – Dependency Inversion Principle (Inversión de Dependencias)

Las clases deben depender de abstracciones y no de concreciones. Esto significa que el código debe comunicarse a través de interfaces o clases abstractas, no de implementaciones concretas. Así, si se cambia una dependencia (por ejemplo, un motor de base de datos), no es necesario reescribir todo el sistema.

Cohesión y Acoplamiento: los dos termómetros de la calidad del diseño

Estos principios se conectan directamente con dos conceptos clave:

Cohesión

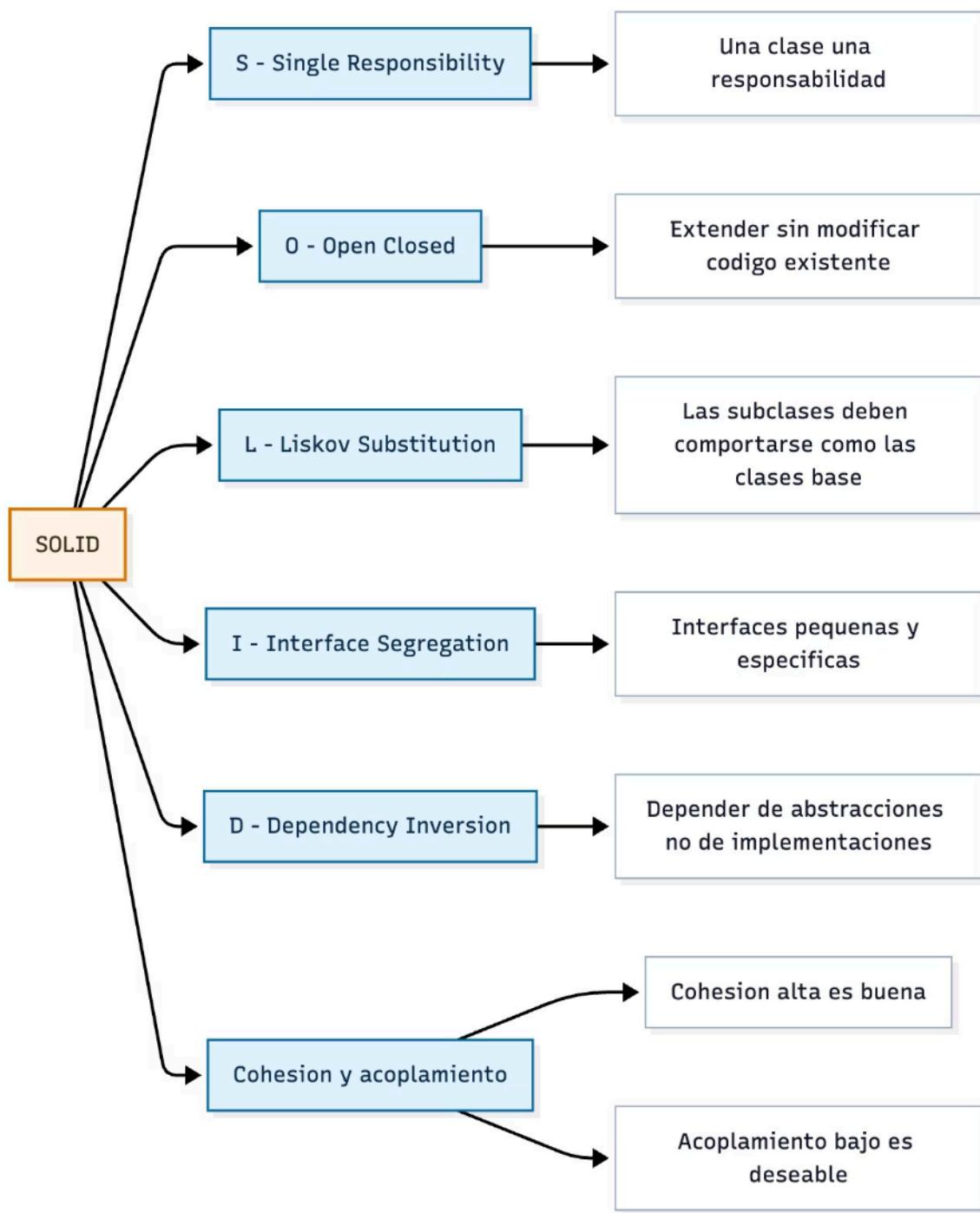
Mide qué tan enfocada está una clase o módulo en una sola tarea. La cohesión alta significa que el código tiene propósito y coherencia.

Acoplamiento

Mide cuánto depende una parte del sistema de otra. El acoplamiento bajo indica independencia y flexibilidad, permitiendo modificar una parte sin afectar el resto.

Un diseño sólido combina alta cohesión y bajo acoplamiento. Esa es la meta que los principios SOLID ayudan a alcanzar. En el contexto profesional, aplicar SOLID no solo mejora el código: reduce el coste de mantenimiento, facilita la incorporación de nuevos desarrolladores y minimiza los errores cuando el proyecto crece.

Esquema Visual: estructura conceptual de SOLID



Descripción:

El esquema muestra los cinco principios como ramas principales que confluyen en un mismo objetivo: lograr código limpio, flexible y estable. Cada rama representa un principio con su enunciado esencial, y al final se relaciona con los conceptos de cohesión (unidad funcional clara) y acoplamiento (baja dependencia entre módulos). Esta estructura visual permite ver a SOLID no como un conjunto de reglas aisladas, sino como una filosofía de diseño coherente.



Caso de Estudio: Shopify y la aplicación práctica de SOLID

Contexto:

Shopify, una de las plataformas de comercio electrónico más utilizadas del mundo, maneja millones de transacciones diarias. Su éxito se basa en una arquitectura de microservicios que necesita ser flexible, estable y fácil de escalar.

Estrategia:

Para mantener esa estabilidad, Shopify adoptó los principios SOLID como estándar de desarrollo interno.

Q

Single Responsibility (S)

Cada microservicio (por ejemplo, pagos, pedidos, usuarios) se diseñó aplicando el principio: cada uno hace una sola cosa y la hace bien.

O

Open/Closed (O)

El sistema está abierto a extensión, permitiendo agregar nuevos métodos de pago o integraciones sin modificar el código base.

I

Interface Segregation (I)

Cada módulo se comunica mediante interfaces ligeras y bien definidas. Por ejemplo, un servicio de inventario no necesita conocer los detalles del servicio de facturación, solo su interfaz pública.

D

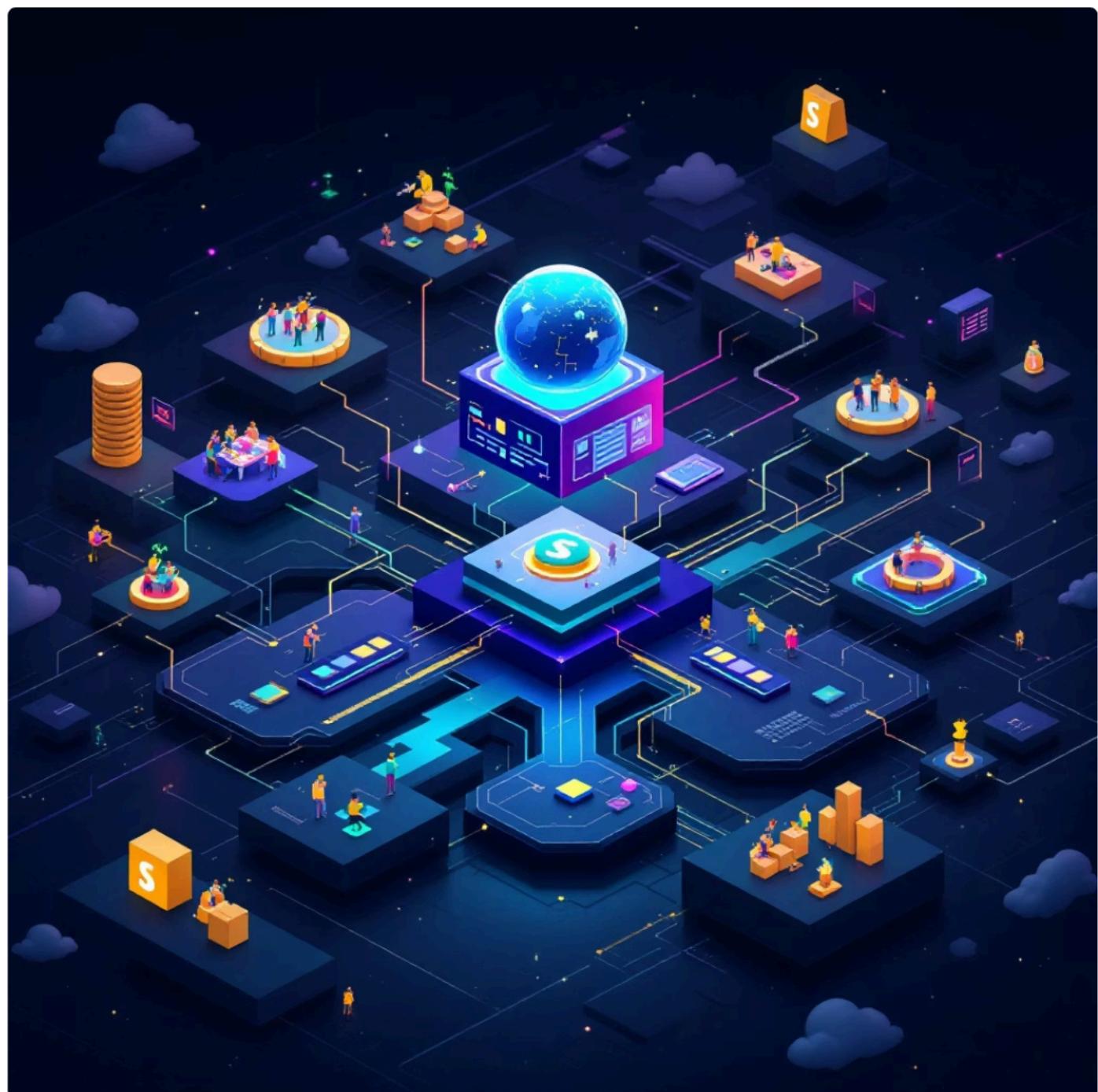
Dependency Inversion (D)

Se usa para desacoplar dependencias. Los servicios no dependen directamente unos de otros, sino de contratos abstractos que pueden reemplazarse o ampliarse.

Resultado:

- Shopify redujo drásticamente la deuda técnica.
- Los equipos de desarrollo pueden trabajar en paralelo sin interferirse.
- El sistema se volvió más resistente a cambios: agregar un nuevo método de pago o actualizar una API no implica reescribir el núcleo.

En palabras de uno de sus arquitectos: "Aplicar SOLID no hizo nuestro código más grande, lo hizo más confiable y más fácil de entender."



Herramientas y Consejos

Mantén las clases pequeñas y bien nombradas

Si una clase tiene más de una responsabilidad, crea otra. Usa nombres que indiquen claramente su propósito: GestorPedidos, ServicioPagos, RepositorioUsuarios.

Inyecta dependencias en lugar de crearlas directamente

Usa inyección de dependencias (DI) con frameworks como Spring (Java), .NET Core (C#) o NestJS (Node.js). Así las clases reciben sus dependencias a través del constructor, en lugar de instanciarlas internamente.

Aplica pruebas unitarias

SOLID y las pruebas se refuerzan mutuamente. Un código con alta cohesión y bajo acoplamiento es más fácil de testear porque cada componente puede probarse de forma independiente.

Evalúa el acoplamiento con herramientas de análisis

Plataformas como SonarQube, CodeScene o Visual Studio Metrics pueden medir automáticamente la dependencia entre módulos y sugerir refactorizaciones.

Refactoriza con frecuencia

La aplicación de SOLID no es un evento puntual. A medida que el sistema evoluciona, revisa y adapta el diseño. Recuerda: "código limpio hoy, proyecto vivo mañana."

Prefiere composición sobre herencia

Aunque SOLID no prohíbe la herencia, abusar de ella puede romper el principio de Liskov. Crear objetos combinando comportamientos (composición) suele ser más flexible que depender de jerarquías rígidas.

Mitos y Realidades

 Mito: "Aplicar SOLID hace el código más complejo."

 **Realidad:** En apariencia puede aumentar el número de clases o interfaces, pero el resultado final es más ordenado, comprensible y fácil de mantener. Lo que se gana en claridad compensa con creces la complejidad inicial.

 Mito: "SOLID solo sirve para proyectos grandes."

 **Realidad:** Los principios son universales. En proyectos pequeños, ayudan a sentar bases limpias; en grandes, evitan el caos. No se trata del tamaño, sino de la calidad del diseño.

Resumen Final

- SOLID = 5 principios para escribir código limpio y extensible.
- S (Single Responsibility): una clase, una tarea.
- O (Open/Closed): extender sin modificar.
- L (Liskov Substitution): las subclases deben comportarse como las clases base.
- I (Interface Segregation): interfaces pequeñas y específicas.
- D (Dependency Inversion): depender de abstracciones, no de implementaciones.
- Cohesión alta = código con propósito claro.
- Acoplamiento bajo = sistemas flexibles y sostenibles.
- Clave profesional: aplicar SOLID no es solo escribir buen código, sino diseñar sistemas preparados para cambiar sin romperse.



Sesión 14: Code Smells y análisis estático con SonarQube / ESLint

Detectar el "mal olor" del código antes de que cause problemas

En el desarrollo de software profesional, escribir código funcional no basta. Un programa puede "funcionar" y, sin embargo, estar plagado de defectos estructurales que dificultan su mantenimiento, comprensión o evolución. A estos defectos sutiles se les llama code smells o "olerías de código".

El término fue popularizado por Martin Fowler en su obra *Refactoring: Improving the Design of Existing Code*, y no se refiere a errores de compilación ni a fallos de ejecución, sino a síntomas de un diseño pobre o descuidado. Un code smell no impide que el programa corra, pero indica que algo podría romperse o volverse inmanejable en el futuro. Es como una grieta en una pared: aún sostiene el peso, pero si no se atiende, acabará provocando un colapso.

Los smells suelen emerger en proyectos que crecen rápidamente, con múltiples desarrolladores y sin controles de calidad automatizados.

Tipos de "mal olor" más comunes:

	Código duplicado El mismo fragmento repetido en varios lugares. Si hay que modificarlo, se debe hacer en todos los sitios, lo que multiplica los errores.		Métodos o funciones excesivamente largos Hacen demasiadas cosas, violando el principio de responsabilidad única.		Clases "Dios" Clases que controlan todo, con cientos de líneas y responsabilidades múltiples.
---	---	---	--	---	---



Nombres confusos

Variables o funciones mal nombradas que requieren leer el código para entender su propósito.



Complejidad ciclomática alta

Demasiadas decisiones (if, while, switch) que dificultan las pruebas y el mantenimiento.

La solución: el análisis estático de código

Para detectar estos code smells de manera temprana, existen herramientas que analizan el código sin ejecutarlo. A esta práctica se le llama análisis estático, y su propósito es inspeccionar el código fuente en busca de patrones problemáticos, inconsistencias y vulnerabilidades.

SonarQube

Para Java, C#, Python, Kotlin y más.

ESLint

Para JavaScript y TypeScript.

PMD o Checkstyle

Complementos enfocados en estilo y convenciones.

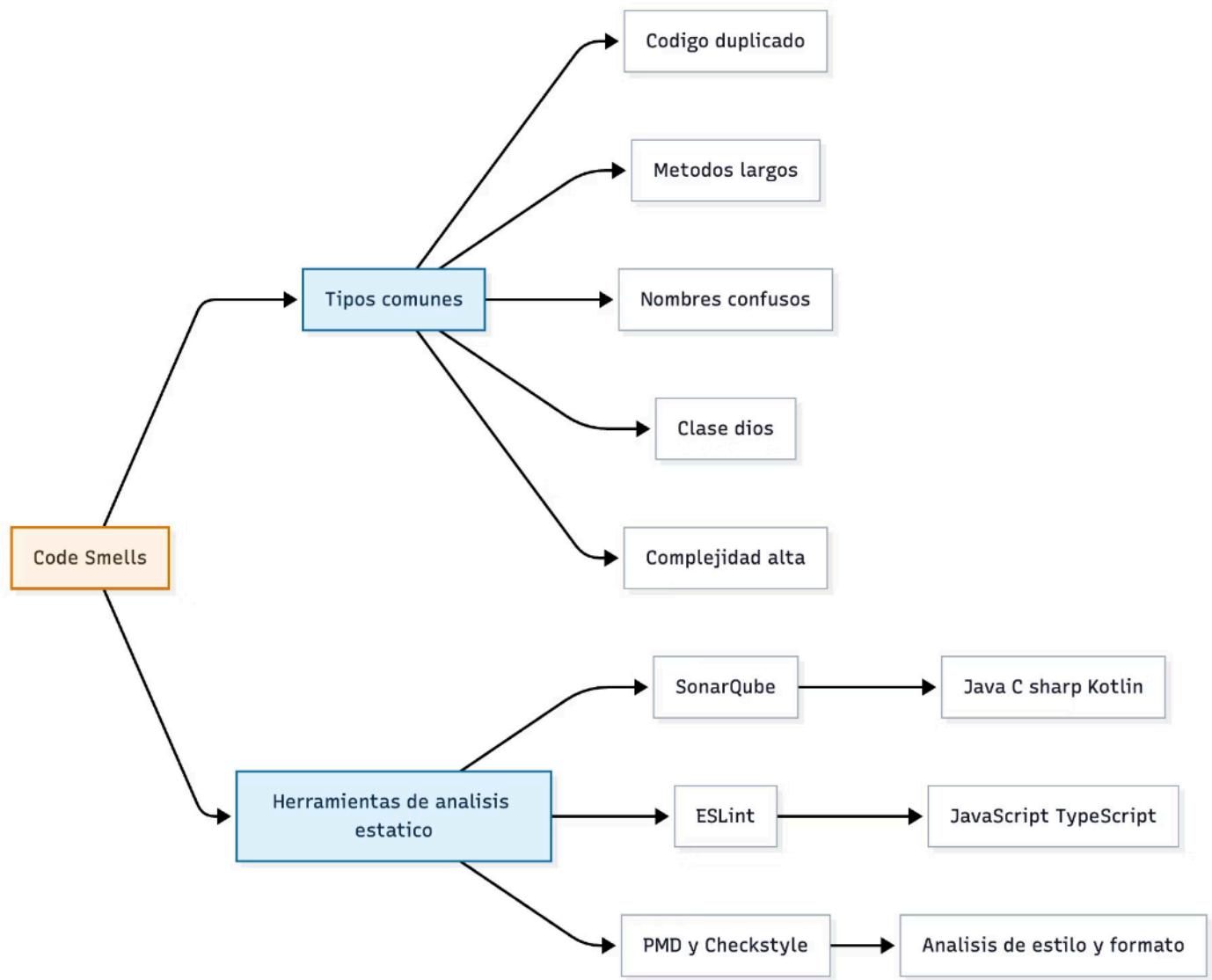
Estas herramientas escanean el código y generan informes detallados con métricas de calidad, complejidad ciclomática, duplicación de código, errores potenciales y cumplimiento de buenas prácticas. Los resultados suelen presentarse visualmente, con colores y puntuaciones (por ejemplo, "A", "B", "C") que indican la salud del proyecto.

Un punto clave del análisis estático es su integración con el pipeline de CI/CD (Integración Continua / Entrega Continua). Esto significa que, cada vez que un desarrollador sube un cambio al repositorio (por ejemplo, a GitHub o GitLab), las herramientas analizan automáticamente el código y bloquean la entrega si se detectan smells críticos o vulnerabilidades.

El beneficio es enorme: los errores se detectan antes de llegar a producción, reduciendo los costes de corrección y manteniendo la confianza en el código. Como regla general, cuanto antes se detecta un problema, más barato es resolverlo.

En definitiva, los code smells no son un signo de incompetencia, sino un recordatorio de que todo código tiende al desorden si no se vigila. Las herramientas de análisis estático funcionan como un espejo: muestran al equipo las áreas que necesitan atención, ayudando a crear software limpio, estable y preparado para escalar.

Esquema Visual: Mapa conceptual de los code smells y su detección



Descripción del diagrama:

El gráfico se estructura en dos grandes bloques conectados:

- A la izquierda, los tipos de code smells más frecuentes, representando las señales de alerta en el código.
- A la derecha, las herramientas de análisis estático que permiten detectarlos automáticamente. Cada herramienta se asocia con su ámbito principal: SonarQube en entornos backend, ESLint en frontend, y PMD/Checkstyle como soporte para normas de estilo.
- El conjunto representa un flujo natural: Detección → Diagnóstico → Mejora continua.



Caso de Estudio: cómo una fintech mejoró su calidad con SonarQube

Contexto:

Una empresa fintech con sede en Barcelona desarrollaba una plataforma de préstamos digitales. Tras tres años de crecimiento acelerado, el equipo notó que cada nueva funcionalidad requería más tiempo de desarrollo y generaba más errores en producción. Las revisiones manuales no eran suficientes: el código base superaba las 500.000 líneas y empezaban a aparecer dependencias circulares y clases con más de 400 líneas.

Estrategia:

El CTO decidió integrar SonarQube en el proceso de build del proyecto. Configuraron la herramienta para que analizara todo el repositorio en cada commit.

150

Code smells
detectados

Problemas de diseño
identificados

30%

Código duplicado
Fragmentos repetidos
innecesariamente

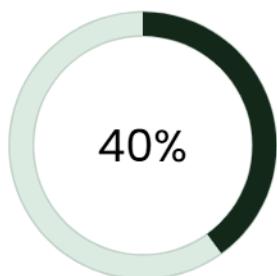
25+

Complejidad
ciclomática
Clases con lógica demasiado
compleja

El equipo priorizó las alertas "críticas" (por ejemplo, duplicaciones y métodos largos) y comenzó una fase intensiva de refactorización. Se dividieron módulos en clases más pequeñas, se renombraron variables confusas y se eliminaron bloques de código repetido.

Resultado:

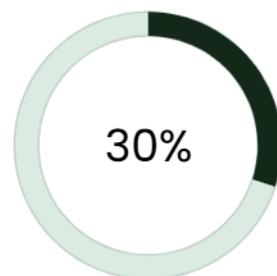
Tras una semana de trabajo:



La legibilidad del código aumentó (según la métrica de SonarQube)



Los errores en producción se redujeron a la mitad



Los tiempos de revisión por pares bajaron

Y, sobre todo, se instauró una cultura de calidad continua, en la que cada nuevo commit debía aprobar los criterios de análisis antes de fusionarse con la rama principal.

La empresa comprobó que invertir en análisis estático no solo mejora el código, sino también la confianza del equipo y la estabilidad del producto.



Herramientas y Consejos

Evita métodos o funciones con más de 20 líneas

Si una función hace demasiadas cosas, es difícil de probar y de entender. Aplica el principio "una función, una responsabilidad".

Usa nombres descriptivos

Un buen nombre reduce la necesidad de comentarios. Prefiere calcularPromedio() frente a calcP(). La claridad es mejor que la brevedad.

Ejecuta análisis estático antes de cada entrega

Configura SonarQube o ESLint para analizar el código automáticamente antes de cada merge request. Así evitas introducir deuda técnica.

Integra el análisis en tu pipeline de CI/CD

Usa GitHub Actions, Jenkins o GitLab CI para automatizar los análisis en cada push. Si el código no cumple con los estándares, el sistema lo bloquea.

Monitorea la complejidad ciclomática

Este indicador mide cuántos caminos lógicos existen en una función. Un valor alto significa que hay demasiados condicionales. Lo ideal es mantenerlo por debajo de 10.

Combina herramientas

SonarQube y ESLint no son excluyentes: pueden usarse juntos en proyectos híbridos (backend + frontend). Mientras SonarQube evalúa calidad global, ESLint se centra en la sintaxis y el estilo del código JavaScript.

Educa al equipo

No basta con usar herramientas; todos deben entender qué significan los reportes. Un smell no es una crítica, es una oportunidad de mejora.

Mitos y Realidades

X Mito: "Los code smells no afectan al rendimiento."

✓ Realidad: Aunque no ralentizan directamente el programa, afectan la mantenibilidad y aumentan el riesgo de errores futuros. Cuanto más difícil es entender el código, más fácil es romperlo accidentalmente.

X Mito: "El análisis estático es solo para grandes empresas."

✓ Realidad: Hoy existen versiones gratuitas y ligeras de SonarQube y ESLint que cualquier equipo puede integrar. Incluso en proyectos pequeños, detectar smells temprano evita reescrituras costosas.

□ Resumen Final

- Code smells: señales de que el código puede mejorarse, aunque funcione.
- Ejemplos: duplicación, métodos largos, clases gigantes, nombres confusos.
- Análisis estático: inspección del código sin ejecutarlo, detectando errores y malas prácticas.
- Herramientas clave: SonarQube (Java, C#), ESLint (JavaScript), PMD y Checkstyle (análisis de estilo).
- Buenas prácticas: ejecutar el análisis antes de cada entrega e integrarlo en CI/CD.
- Beneficio principal: mayor mantenibilidad, menor deuda técnica y equipos más productivos.





Sesión 15: Técnicas de refactorización + documentación con Javadoc

Mejorar el código sin romperlo

En el ciclo de vida de un proyecto de software, el código no permanece estático: evoluciona, se amplía y se adapta a nuevas necesidades. Con el paso del tiempo, esta evolución genera fragmentos redundantes, estructuras poco claras o dependencias innecesarias que, si no se gestionan, degradan la calidad del sistema. Aquí es donde entra en juego la refactorización, una práctica fundamental en el desarrollo profesional.

Refactorizar significa modificar la estructura interna del código sin alterar su comportamiento externo. Es decir, el programa sigue haciendo exactamente lo mismo, pero su código queda más limpio, más legible y más fácil de mantener. Este proceso no se centra en añadir nuevas funcionalidades, sino en mejorar la base existente, eliminando duplicaciones, simplificando la lógica o renombrando elementos para hacer el código más comprensible.

Cuándo refactorizar

↓	⌚	⇨
Después de detectar code smells	Antes de añadir nuevas funciones	Tras una revisión de código o entrega de sprint
Cuando el código muestra "oler" (duplicaciones, clases gigantes, métodos confusos).	Limpiar antes de construir evita que los nuevos cambios se asienten sobre una base frágil.	Mantener la higiene del código de forma continua previene la acumulación de deuda técnica.

Técnicas más comunes

Extraer métodos o clases

Dividir un bloque grande en partes pequeñas y coherentes. Por ejemplo, separar la lógica de validación de la de cálculo.

Renombrar variables y funciones

Usar nombres significativos mejora la lectura del código sin necesidad de comentarios innecesarios.

Eliminar duplicados

Trasladar código repetido a una función o clase reutilizable reduce los errores y facilita el mantenimiento.

Simplificar condicionales complejas

Reemplazar cadenas de if-else por estructuras más claras o por polimorfismo.

Refactorizar con propósito

El objetivo de la refactorización no es la estética, sino la sostenibilidad. Un código bien estructurado:

- Reduce el riesgo de errores al modificarlo.
- Facilita las pruebas unitarias.
- Mejora la colaboración entre equipos.
- Permite incorporar nuevas funcionalidades sin romper lo existente.

La importancia de la documentación

La refactorización y la documentación son dos caras de la misma moneda. Limpiar el código mejora su estructura, pero documentarlo comunica el conocimiento técnico al resto del equipo. En entornos Java, la herramienta más usada para este propósito es Javadoc, que genera documentación HTML automáticamente a partir de comentarios especiales dentro del código.

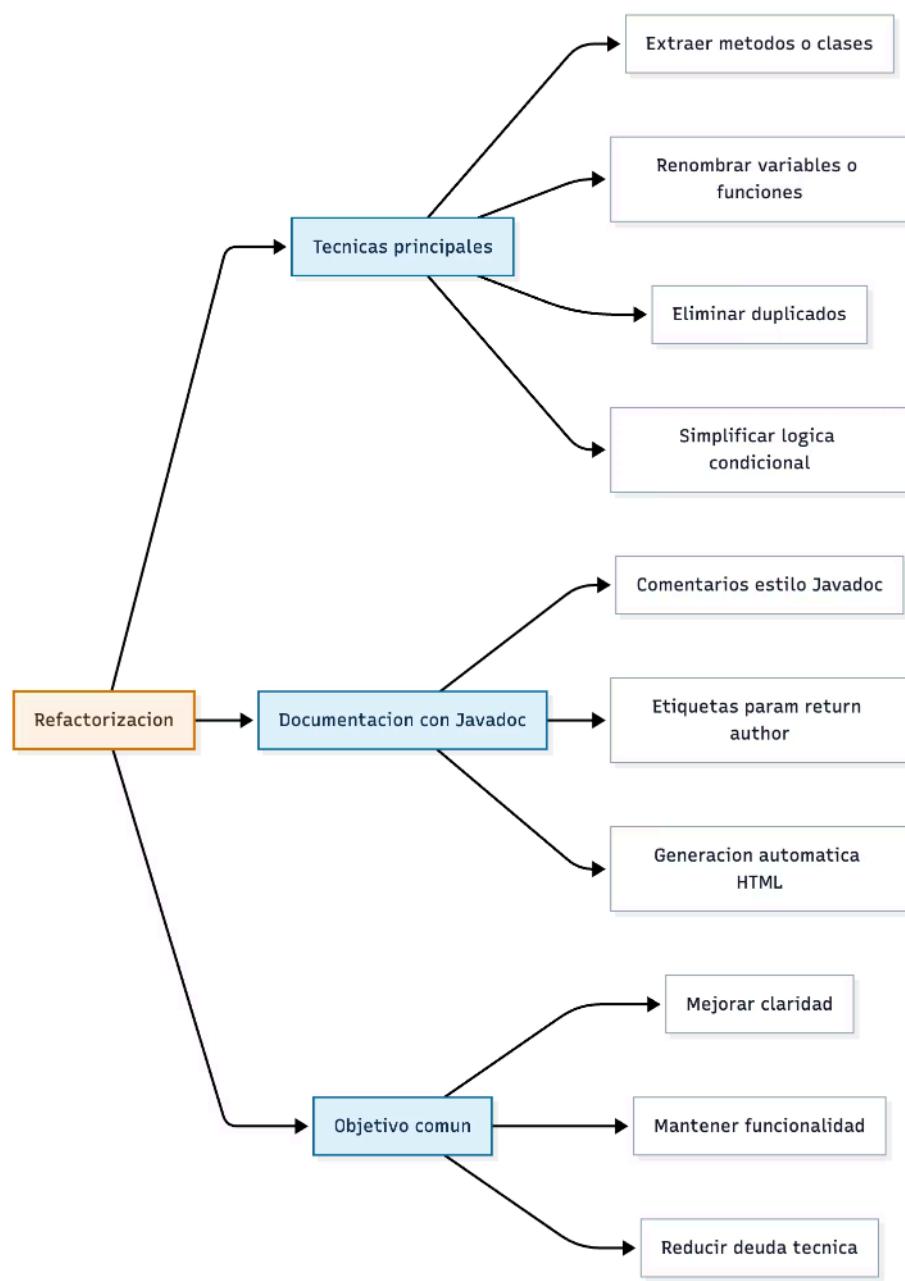
Por ejemplo:

```
/**  
 * Calcula el promedio de ventas mensuales.  
 * @param ventas Lista de importes diarios.  
 * @return Promedio mensual en formato double.  
 * @author Carlos Pérez  
 */  
public double calcularPromedio(List<Double> ventas) { ... }
```

Estos comentarios, al procesarse con el comando javadoc, se transforman en páginas navegables que describen las clases, métodos, parámetros y autores, convirtiéndose en una guía viva del sistema.

En resumen, refactorizar es limpiar; documentar es comunicar. Ambas prácticas forman el núcleo del código limpio y profesional.

Esquema Visual: mapa conceptual de refactorización y documentación



Descripción del esquema:

El diagrama muestra la relación entre refactorización y documentación como dos procesos complementarios:

- Las técnicas de refactorización mejoran la estructura interna del código.
- La documentación con Javadoc explica su comportamiento externo.
- Ambas convergen en un mismo fin: lograr código claro, funcional y mantenible.
- El nodo "Objetivo Común" resalta los tres pilares de estas prácticas: claridad, estabilidad y sostenibilidad.



Caso de Estudio: IntelliJ IDEA y la refactorización continua

Contexto:

IntelliJ IDEA, el popular entorno de desarrollo de JetBrains, es un software con millones de líneas de código y actualizaciones frecuentes. Gestionar una base tan extensa exige mantener un nivel excepcional de calidad.

Estrategia:

El equipo de JetBrains practica la refactorización continua. Cada semana, dedican tiempo específico a revisar y limpiar partes del código sin alterar las funcionalidades del producto. Utilizan las potentes herramientas de refactorización automática de su propio IDE —IntelliJ IDEA—, que detecta duplicaciones, variables no utilizadas y dependencias innecesarias.

Además, cada módulo y clase pública está documentada mediante Javadoc, lo que permite que nuevos ingenieros puedan integrarse rápidamente y comprender el funcionamiento sin tener que analizar el código completo.



Refactorización programada

Tiempo semanal dedicado específicamente a limpiar código



Herramientas automáticas

Uso de funciones de refactorización del propio IDE



Documentación completa

Javadoc en cada módulo y clase pública



Integración rápida

Nuevos desarrolladores comprenden el código fácilmente

Resultados:

- **Deuda técnica mínima:** el código se mantiene estable pese al tamaño del proyecto.
- **Mayor productividad:** los desarrolladores entienden rápidamente el contexto de cada clase gracias a la documentación generada.
- **Actualizaciones seguras:** los cambios no rompen funcionalidades existentes, gracias al bajo acoplamiento y las pruebas automatizadas.

Conclusión:

El caso de IntelliJ demuestra que refactorizar de forma continua y documentar de manera sistemática es una inversión en estabilidad. No se trata de corregir errores, sino de prevenirlos antes de que aparezcan.



Herramientas y Consejos



Refactoriza después de cada entrega o sprint

No esperes a tener "tiempo libre". La refactorización debe integrarse como una práctica regular del flujo de trabajo, no como una tarea excepcional.



Usa IDEs con refactorización automática

Herramientas como IntelliJ IDEA, Eclipse y Visual Studio Code ofrecen funciones seguras para renombrar variables, extraer métodos o mover clases sin romper referencias.



Aplica pruebas automatizadas tras cada cambio

La refactorización debe ir acompañada de pruebas unitarias. Así se garantiza que, aunque se haya modificado la estructura, la funcionalidad sigue siendo la misma.



Documenta cada clase pública y método con Javadoc

Utiliza etiquetas estándar como:

- `@param` para describir parámetros.
- `@return` para el valor de retorno.
- `@throws` para excepciones.
- `@author` y `@version` para metadatos del autor y la versión.



Genera documentación automáticamente

Usa el comando:

```
javadoc -d ./docs *.java
```

Esto creará una carpeta "docs" con páginas HTML naveables, perfectas para compartir dentro del equipo o con otros desarrolladores.

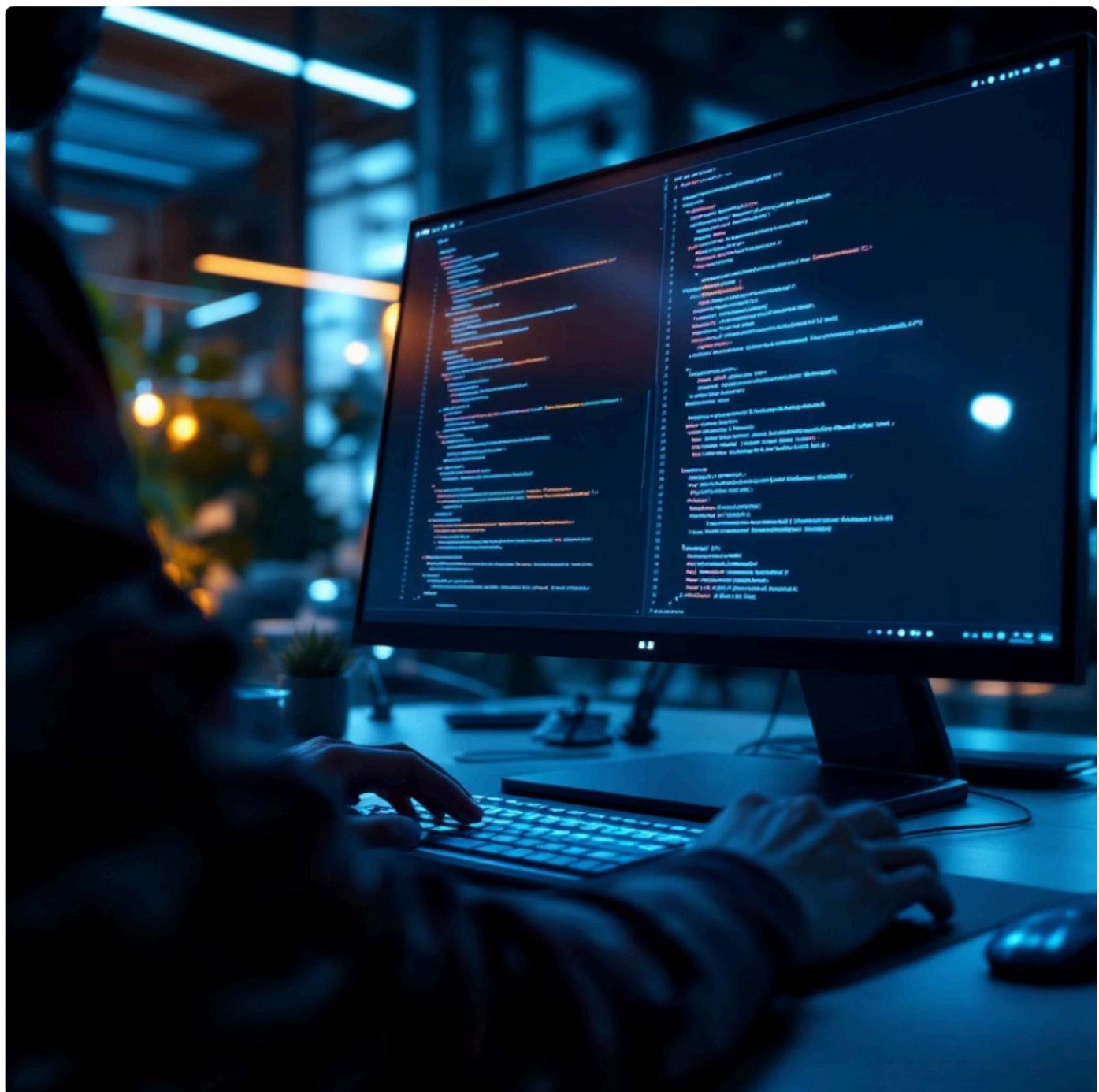


Integra la documentación en el pipeline CI/CD

Configura tu entorno de integración continua (Jenkins, GitHub Actions) para generar documentación automáticamente en cada versión.

Establece revisiones de código centradas en claridad

Durante las code reviews, no solo se debe revisar la funcionalidad, sino también la legibilidad y la calidad de los nombres. La claridad del código es un criterio de calidad tan importante como su rendimiento.



Mitos y Realidades

✗ Mito: "Refactorizar es perder el tiempo."

✓ **Realidad:** Es una inversión en la salud del proyecto. Cada hora dedicada a limpiar código ahorra muchas más en corrección de errores y mantenimiento futuro.

✗ Mito: "La documentación se hace al final del proyecto."

✓ **Realidad:** Si se pospone, rara vez se completa. Documentar durante el desarrollo asegura precisión y coherencia. Además, herramientas como Javadoc automatizan el proceso, reduciendo el esfuerzo manual.

✗ Mito: "Solo los grandes proyectos necesitan refactorización."

✓ **Realidad:** La refactorización es útil desde el primer día. Cuanto antes se aplique, más limpio crece el proyecto y menos costosa resulta a largo plazo.

□ Resumen Final

- Refactorización: proceso de mejorar la estructura del código sin alterar su funcionalidad.
- Objetivo: claridad, eficiencia y mantenibilidad.
- Técnicas clave: extraer métodos/clases, renombrar variables, eliminar duplicados y simplificar lógica.
- Documentación: Javadoc genera documentación HTML a partir de comentarios estructurados.
- Herramientas: IntelliJ IDEA, Eclipse, VS Code.
- Buenas prácticas: refactorizar de forma continua y documentar cada componente público.
- Resultado profesional: código limpio, legible y sostenible a largo plazo.