Módulo Profesional Optativo I

PROMETEO

Unidad 1. Ampliación en Introducción a la Programación

Un paradigma de programación es una forma de pensar y estructurar soluciones en código. No es solo "estilo"; define cómo modelas el estado, cómo compones operaciones y cómo razonas sobre la corrección.

Sesión 1 – Paradigmas de programación (imperativo, declarativo, funcional, lógico)

Un paradigma de programación es una forma de pensar y estructurar soluciones en código. No es solo "estilo"; define cómo modelas el estado, cómo compones operaciones y cómo razonas sobre la corrección. En la práctica profesional actual, rara vez trabajas con un único paradigma: la mayoría de lenguajes y plataformas son multiparadigma y te conviene elegir el enfoque que mejor resuelva cada parte del problema.

1

Imperativo.

Te centras en *cómo* lograr el resultado: secuencias de instrucciones que modifican un estado (variables, estructuras). Es el enfoque natural para lógica de orquestación, automatización de tareas, E/S y flujos que dependen del tiempo.

Lenguajes habituales: C, C++, Java, Python (en su uso más común), Go.

Ventajas: claridad paso a paso, control fino de recursos. Riesgos: mayor probabilidad de errores por estado compartido y efectos secundarios.

2

Declarativo.

Describes *qué* quieres sin fijar el *cómo*. SQL es el ejemplo icónico: "SELECT nombre FROM alumnos" y el motor decide el plan de ejecución. También lo ves en HTML/CSS, infra como código (Terraform), o en consultas sobre datos (Spark SQL).

Ventajas: menos código para el mismo objetivo, optimización delegada al motor. Riesgos: curva de aprendizaje para pensar en términos de resultados y no de pasos.

3

Funcional.

Basado en funciones puras (misma entrada → misma salida, sin efectos secundarios) y composición. Lenguajes: Haskell, Clojure, Scala, F#, Elixir; muchos lenguajes mainstream incorporan funciones de orden superior (map, filter, reduce) y *inmutabilidad preferente*.

Ventajas: código más fácil de testear, paralelizar y razonar (menos "estado escondido"). Riesgos: puede resultar menos intuitivo al inicio y requiere cambiar hábitos (evitar mutaciones).

4

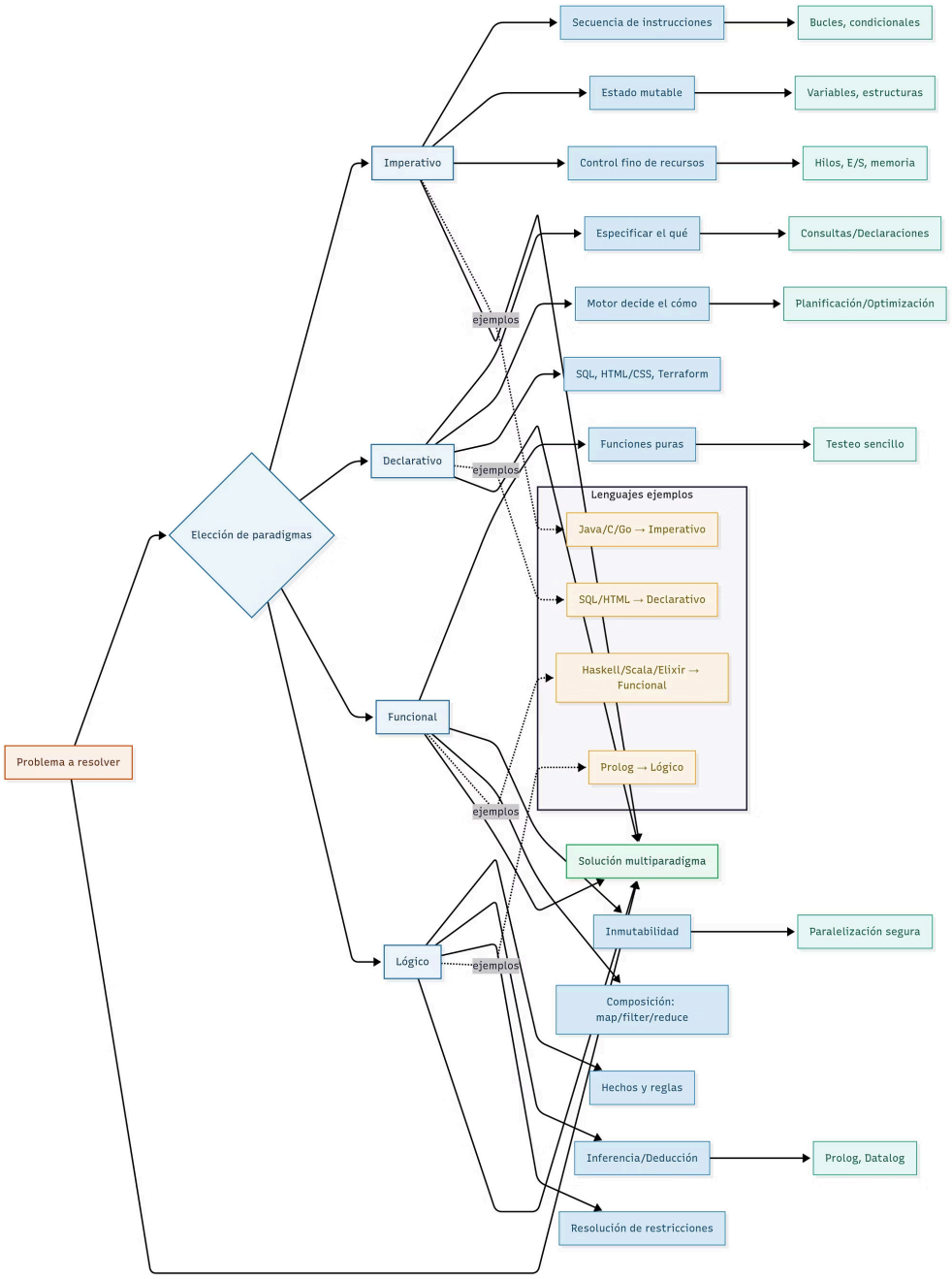
Lógico.

Programa mediante hechos y reglas; el motor de inferencia deduce respuestas. Prolog es el ejemplo clásico. Se usa en resolución de restricciones, reglas de negocio complejas, verificación y prototipado de sistemas expertos.

Ventajas: expresividad para problemas combinatorios y de razonamiento. Riesgos: rendimiento si no se modela bien, ecosistema más reducido.

Conclusión práctica. En un backend típico puedes orquestar flujos (imperativo), consultar datos (declarativo/SQL), procesar grandes volúmenes con pipelines inmutables (funcional) y, si procede, encapsular reglas con un motor lógico. Pensar "en plural" (multiparadigma) te da herramientas para escribir soluciones más robustas, concisas y mantenibles.

Esquema Visual



Leyenda y lectura:

- **A → B:** Partes de un problema y decides el paradigma (o mezcla).
- **Imperativo (I):** Bloques I1–I6 representan pasos, estado y control de recursos.
- **Declarativo (D):** D1–D5 indican que expresas el resultado y el motor planifica.
- **Funcional (F):** F1–F5 enfatiza funciones puras, inmutabilidad y paralelización.
- **Lógico (L):** L1–L4 muestran hechos/reglas y deducción automática.
- **Solución multiparadigma (C):** Confluencia práctica de enfoques en un sistema real.

Caso de Estudio – Spotify: recomendaciones a escala con mezcla de paradigmas

Contexto. Spotify procesa actividad de más de 500 millones de usuarios para generar recomendaciones (playlists personalizadas como Discover Weekly o Daily Mix) y rankings. El reto: combinar *ingestión masiva de datos*, *procesamiento eficiente*, *reglas de negocio* y *exposición rápida* en producto.

Estrategia (paradigmas aplicados).

Declarativo para consultas y agregaciones.

Gran parte del análisis intermedio se modela con SQL (o dialectos como Spark SQL/Trino), definiendo *qué* métricas/joins se necesitan sin codificar el plan físico. Esto permite que el motor optimice los planes de ejecución (pushdown, particionamiento, broadcast joins) y cambie estrategias sin tocar la declaración.

Funcional en pipelines de datos.

Los jobs de procesamiento por lotes/streaming suelen escribirse en marcos que favorecen el estilo funcional (p. ej., *map*, *flatMap*, *filter*, *reduceByKey*, *window*), a menudo con lenguajes como Scala o Python. La inmutabilidad y la composición facilitan la paralelización y reducen errores por estado compartido.

Imperativo para orquestación y servicios.

Los microservicios que sirven recomendaciones y las capas de orquestación/ETL usan estilo imperativo (en Python, Java o Go) para controlar flujos, reintentos, timeouts, cachés y E/S de red. Aquí el control explícito del *cómo* es valioso.

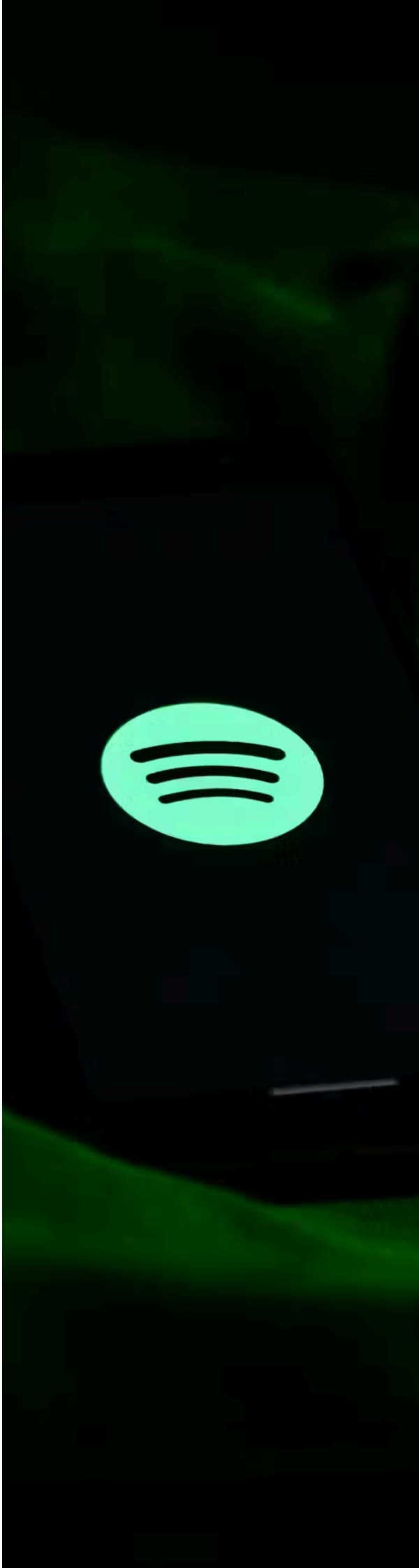
Lógico/reglas para afinado editorial.

Además de modelos de ML, hay reglas de negocio (por ejemplo, limitar repeticiones, evitar contenido inapropiado o respetar licencias por región). Estas reglas pueden declararse en motores de reglas (lógico/declarativo) o en DSLs, separando *política* de *implementación* para mantener claridad y auditablez.

Resultado. La combinación multiparadigma permite:

- Escalar el cálculo (funcional + declarativo) con costes controlados.
- Cambiar estrategias de ejecución sin reescribir lógica (declarativo).
- Exponer recomendaciones de baja latencia y controlar fallos (imperativo).
- Ajustar reglas de negocio sin redeploy completo (lógico/DSL).

En conjunto, se consigue personalización consistente y rápida, mejorando *engagement* y retención, sin bloquearse en un único estilo de programación.



Herramientas y Consejos para Paradigmas

Piensa en capas, no en dogmas

- **Capa de datos:** SQL/Spark SQL (declarativo) para expresar métricas y joins.
- **Capa de procesamiento:** Apache Spark/Beam/Flink con APIs funcionales.
- **Capa de servicio:** Python/Java/Go para orquestación imperativa, control de E/S y resiliencia.
- **Capa de reglas:** Motores como **Drools**, **Open Policy Agent (OPA)** o DSLs propias para políticas.

En tu día a día con Python (laboratorio multiparadigma)

- Imperativo: bucles, manejo de archivos, requests.
- Declarativo: **SQLModel**/**SQLAlchemy** para expresar consultas; **pandas.query** para filtros declarativos.
- Funcional: **itertools**, **functools**, funciones puras, **map/filter/reduce** (o **pandas**/**Polars** para transformaciones vectorizadas).
- Reglas: mini-DSLs con expresiones o uso de **jsonlogic**.

Prueba un mismo problema en estilos distintos

"Primos hasta N":

- Imperativo: bucles y marcadores.
- Funcional: filter con un predicado puro y comprensión inmutable.
- Declarativo: generar tabla de números y filtrar con una vista/CTE en SQL.
- Lógico: reglas en Prolog para divisibilidad y búsqueda.

Control de calidad y mantenibilidad por paradigma

- Imperativo: **pytest/JUnit**, cobertura, *linters* (**flake8**, **Pylint**, **Checkstyle**).
- Declarativo: verifica resultados esperados con *golden tests* y snapshots; monitoriza planes de ejecución.
- Funcional: prioriza funciones puras y *property-based testing* (**Hypothesis**, **ScalaCheck**).
- Lógico: tests de reglas y trazas de inferencia.

Estrategia de adopción gradual en equipos

- Introduce primero *patrones funcionales* en código imperativo (inmutabilidad local, map/filter).
- Aísla reglas en módulos declarativos o lógicos para facilitar cambios normativos o de negocio.
- Ofrece *katas* internas alternando paradigmas para que el equipo gane soltura.

Mitos y Realidades

❌ Mito: "Un lenguaje solo puede pertenecer a un paradigma" → FALSO.

La mayoría de lenguajes modernos son multiparadigma: Python, JavaScript, Kotlin o C# permiten mezclar imperativo, funcional y, con librerías, estilos declarativos. Esto te da flexibilidad para elegir la herramienta mental adecuada a cada parte del sistema.

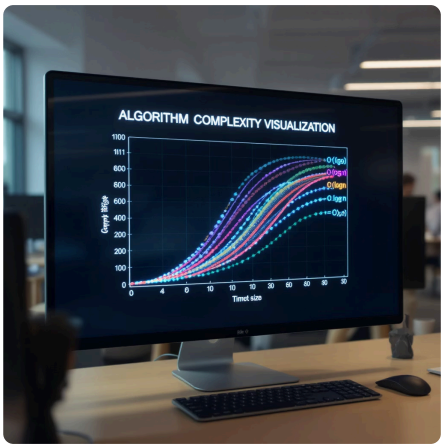
❌ Mito: "El funcional es solo para la academia y no escala en la industria" → FALSO.

Empresas como WhatsApp (Erlang/Elixir en el ecosistema BEAM), Twitter (Scala en su día), o muchas plataformas de *data engineering* usan intensamente conceptos funcionales para concurrencia, tolerancia a fallos y procesamiento distribuido. Lo funcional resuelve problemas reales, especialmente donde la inmutabilidad y la paralelización son valiosas.

Resumen final

- Paradigma = forma de pensar y estructurar un programa.
- Imperativo: pasos y estado; Declarativo: describes el "qué"; Funcional: funciones puras e inmutabilidad; Lógico: hechos y reglas.
- En práctica profesional, mezcla de paradigmas según capa y objetivo.
- Multiparadigma = soluciones más claras, escalables y mantenibles.

Sesión 2 – Eficiencia algorítmica (tiempo y espacio). Big-O



Saber si un programa funciona no es suficiente: en entornos profesionales también se mide *cómo de bien* lo hace. Aquí entra en juego la eficiencia algorítmica, que analiza dos recursos principales: **tiempo** y **espacio**.

- **Tiempo de ejecución.** Indica el número de operaciones que un algoritmo requiere en función del tamaño de los datos de entrada (*n*). No se mide en segundos exactos, sino en *orden de magnitud*. Por ejemplo, un algoritmo que procesa una lista de 1.000 elementos en 1.000 pasos es O(n). Si duplicas el tamaño, duplica también el trabajo.
- **Espacio en memoria.** Cuánto almacenamiento adicional necesita. Un algoritmo puede ser muy rápido pero consumir demasiada memoria, lo que lo hace poco práctico en sistemas limitados.

Para comparar algoritmos se utiliza la **notación Big-O**, que clasifica el crecimiento de recursos en función de *n*. Se centra en el "peor caso", ya que en producción importa saber qué ocurre con la carga máxima.

Ejemplos típicos:

O(1) - Tiempo constante

Sin importar el tamaño. Ejemplo: acceder a un elemento en un array por índice.

O(log n) - Crecimiento lento

Incluso con datos enormes. Ejemplo: búsqueda binaria en una lista ordenada.

O(n) - Tiempo lineal

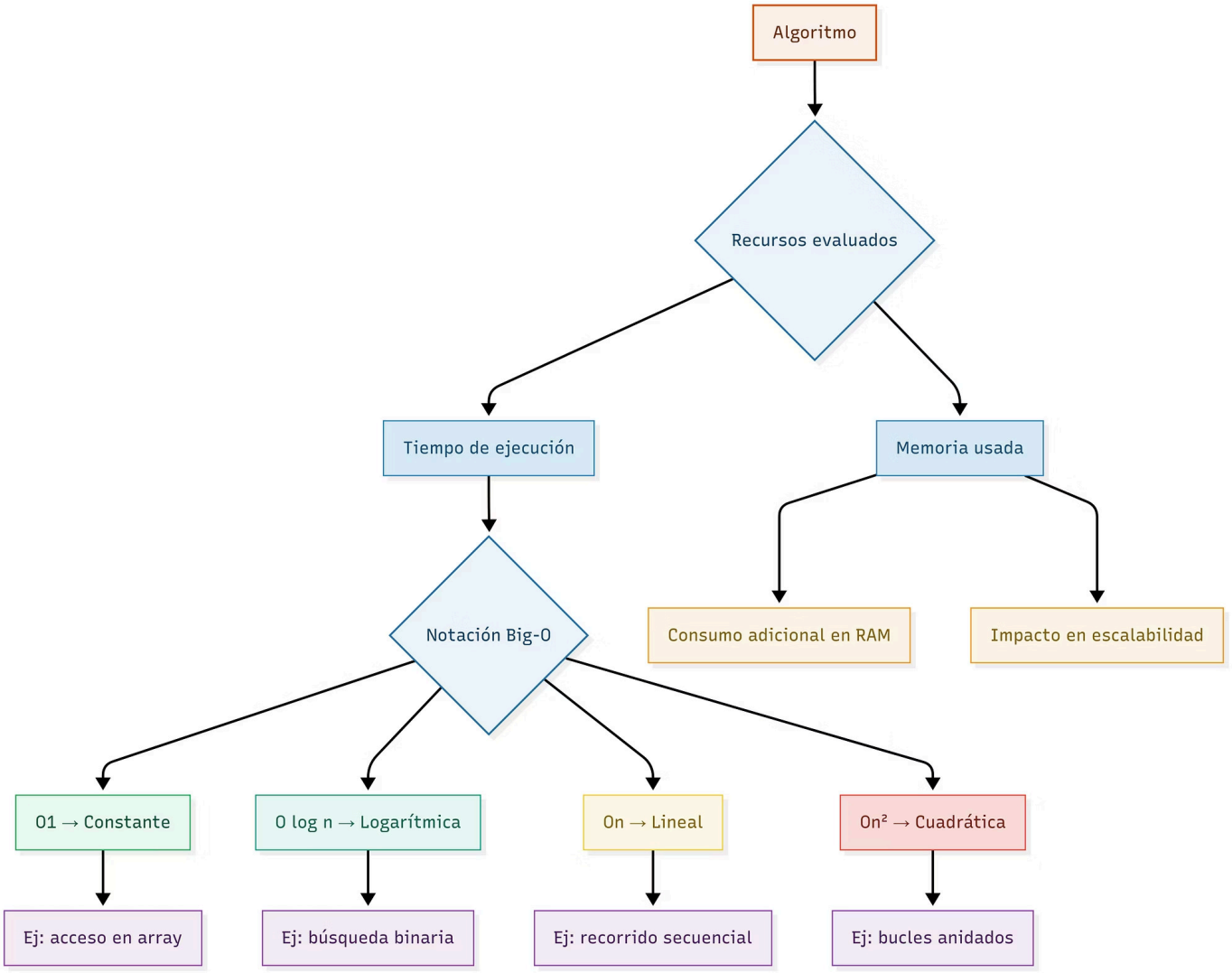
Proporcional al tamaño de la entrada. Ejemplo: recorrer una lista con un bucle.

O(n²) - Tiempo cuadrático

Típico de algoritmos con bucles anidados. Ejemplo: comparar cada par de elementos en una lista.

En la práctica profesional, elegir entre algoritmos no es cuestión académica: puede significar la diferencia entre un sistema que responde en milisegundos o uno que bloquea servidores.

Esquema Visual



Cómo leerlo:

- Todo algoritmo (A) se evalúa por recursos (B).
- Tiempo (C) se mide con Big-O (E), que clasifica casos (F–I).
- Cada caso tiene un ejemplo claro (J–M).
- Espacio (D) refleja memoria adicional y su efecto en la escalabilidad (N–O).

Caso de Estudio – Amazon y la búsqueda en catálogos

Contexto. Amazon gestiona catálogos con cientos de millones de productos. Un usuario espera resultados inmediatos al escribir en el buscador. La eficiencia algorítmica es crítica: incluso un pequeño retraso afecta a las ventas.

Búsqueda lineal (O(n)):

Comprobar uno por uno los productos sería impracticable. Con 100 millones de ítems, el tiempo de respuesta se dispararía.

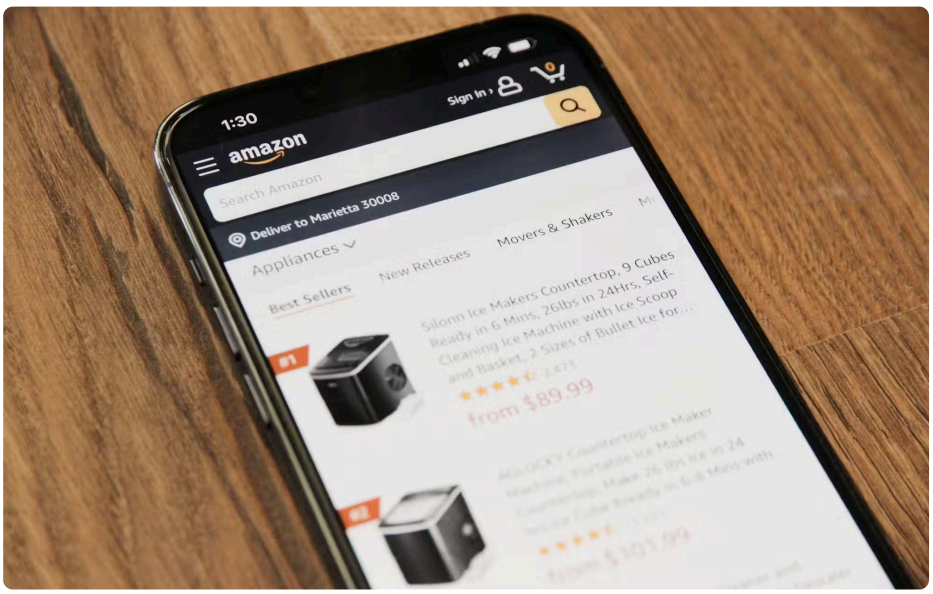
Búsqueda binaria (O(log n)):

En catálogos ordenados, cada comparación descarta la mitad. Con 100 millones de productos, se necesitan unas 27 comparaciones en lugar de 100 millones.

Estructuras avanzadas:

Amazon combina índices invertidos y árboles B (eficiencia cercana a O(log n)) para consultas rápidas, junto con cachés para respuestas en milisegundos.

Resultado. Este enfoque permite que millones de búsquedas concurrentes se resuelvan en tiempos subsegundo, mejorando la conversión y optimizando el uso de servidores.



Herramientas y Consejos

Perfilado en tu lenguaje

- En Python: cProfile, line_profiler.
- En Java: **VisualVM** o **YourKit**.
- En JavaScript: pestaña *Performance* en Chrome DevTools.

Analiza estructuras de datos

- Arrays: acceso O(1), inserciones O(n).
- Listas enlazadas: inserciones rápidas, accesos lentos.
- Árboles binarios balanceados: O(log n) en búsqueda e inserción.
- Tablas hash: acceso promedio O(1), pero depende del *hashing*.

No ignores la memoria

Un algoritmo con baja complejidad temporal puede ser inviable si requiere gigabytes de memoria. Usa herramientas como **Valgrind** (C/C++) o **memory_profiler** (Python).

Mide con datos reales

Antes de elegir un algoritmo, prueba con el volumen de datos esperado. A veces un algoritmo "menos eficiente" teórico rinde mejor en entradas pequeñas por la sobrecarga de otro más complejo.

Mitos y Realidades

❌ Mito: "El algoritmo más rápido en una prueba pequeña es siempre el mejor."
✔️ Realidad: Lo que importa es el comportamiento al crecer los datos. Por eso Big-O se centra en el peor caso y grandes volúmenes.

❌ Mito: "Big-O mide el tiempo exacto de ejecución."
✔️ Realidad: Big-O no mide segundos, sino el orden de crecimiento. Dos algoritmos O(n) pueden tardar tiempos distintos; lo importante es cómo escalan con *n*.

Resumen final

- Eficiencia = tiempo y memoria de un algoritmo.
- Big-O clasifica crecimiento: O(1), O(log n), O(n), O(n^2).
- Escalabilidad se mide por el peor caso, no por ejemplos aislados.
- Herramientas de perfilado ayudan a confirmar rendimiento real.
- Un algoritmo eficiente mejora experiencia y reduce costes de infraestructura.

Sesión 3 – Diseño incremental y metodologías ágiles en pequeños proyectos

Cuando empiezas a programar, la tentación es resolverlo todo en un único bloque de código. Eso funciona en ejercicios breves, pero en proyectos con varias funciones, usuarios o requisitos cambiantes, la complejidad se dispara. Aquí surge el **diseño incremental**, que consiste en construir un sistema en pasos pequeños, manteniendo siempre una versión funcional.

Este enfoque conecta directamente con las **metodologías ágiles**, que priorizan entregas frecuentes, retroalimentación continua y capacidad de adaptación. La clave es el ciclo:

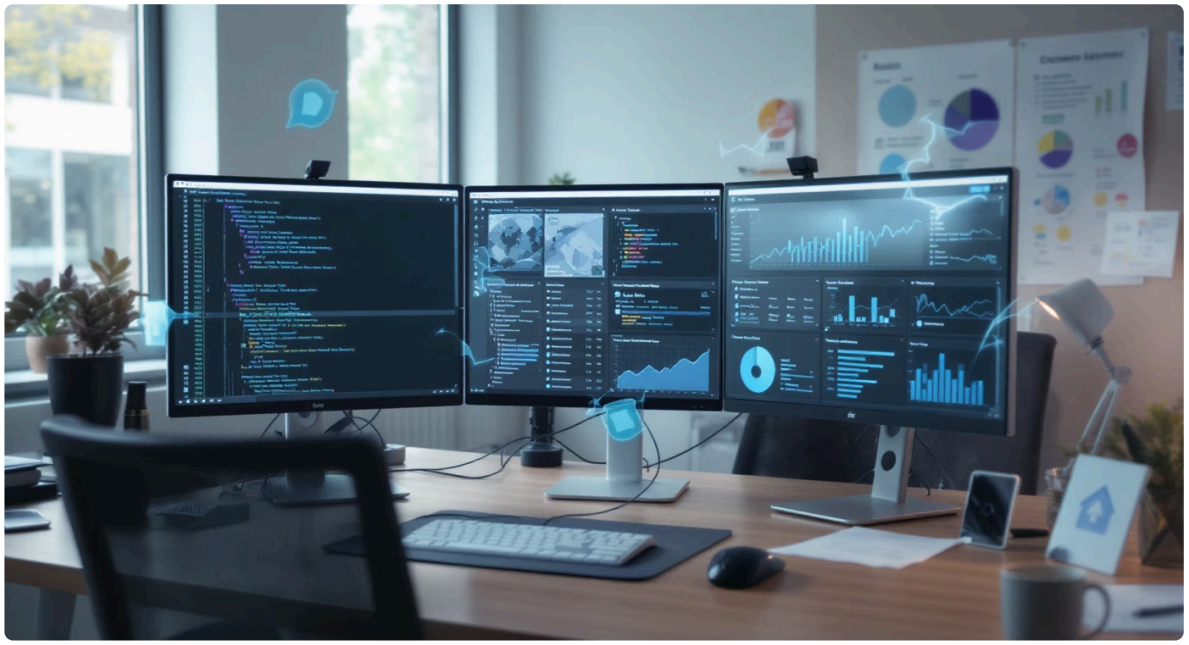
1	2
Implementar una funcionalidad mínima	Probarla con usuarios o casos reales
3	4
Recibir feedback y ajustar	Iterar con nuevas funciones sin romper lo anterior

El concepto de **MVP (Producto Mínimo Viable)** resume esta idea: construir lo esencial que permite validar si la solución es útil, antes de invertir tiempo en extras.

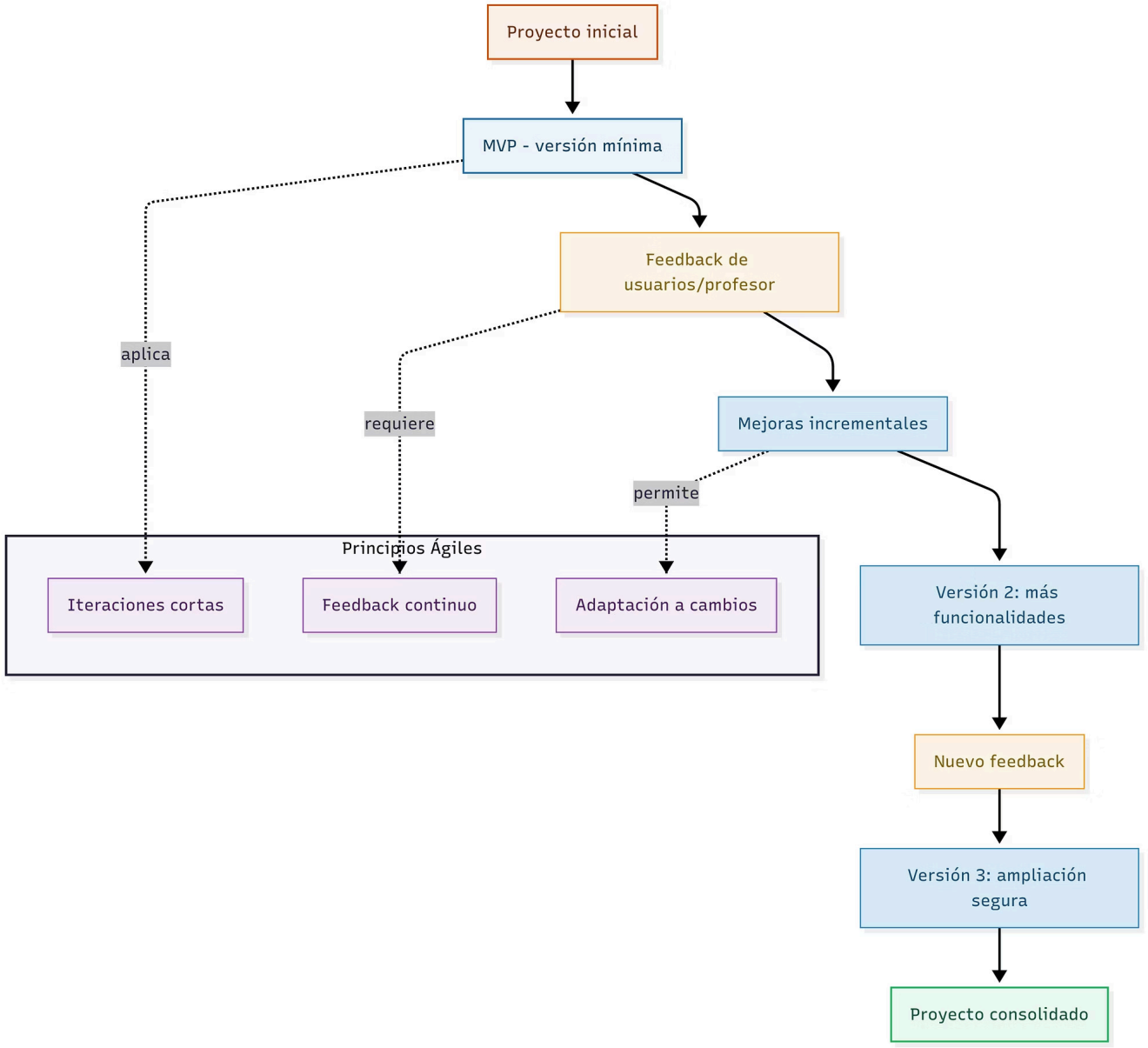
En proyectos de programación, esta forma de trabajo tiene beneficios claros:

- Siempre dispones de una versión que funciona, aunque sea básica.
- Detectas errores temprano y evitas acumular fallos difíciles de depurar.
- Te adaptas mejor a cambios de requisitos, algo muy común tanto en clase como en empresas.

En definitiva, el diseño incremental no es un lujo de grandes compañías, sino una práctica imprescindible incluso en pequeños proyectos personales o académicos.

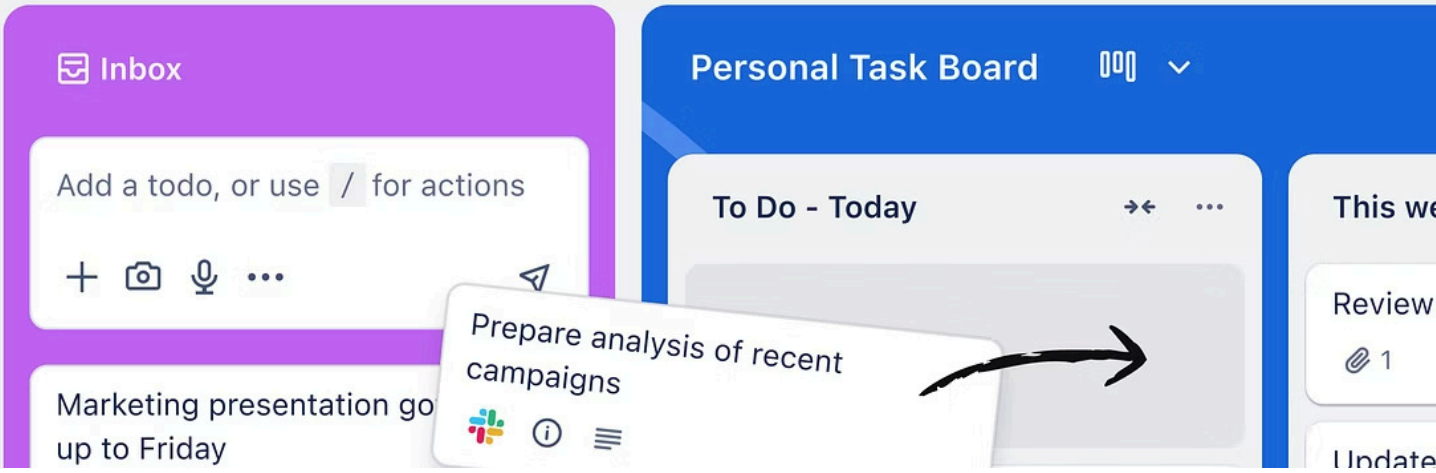


Esquema visual



Lectura:

- El ciclo arranca en un MVP (B).
- Cada iteración añade mejoras (D, E, G) siempre respaldadas por feedback.
- Los principios ágiles (I, J, K) son el marco que guía cada paso.



Caso de Estudio – Trello: de prototipo a producto global

> **Contexto.** Trello nació como una herramienta interna en Fog Creek Software. El objetivo inicial: visualizar tareas de manera simple, sin la complejidad de sistemas de gestión tradicionales.

Estrategia.

1	2
MVP: un tablero con tarjetas que podían moverse entre columnas (pendiente, en progreso, hecho).	**Iteraciones:** tras probarlo internamente, añadieron gradualmente funciones: invitación de usuarios, comentarios en tarjetas, checklists.
3	**4**
Feedback: liberaron el producto en beta pública y recogieron sugerencias directas de la comunidad.	**Agilidad:** priorizaron funcionalidades según la demanda, en lugar de desarrollar un plan cerrado a largo plazo.

Resultado. En pocos meses tenían un producto adoptado masivamente, que luego fue adquirido por Atlassian. El secreto: empezar con lo mínimo y mejorar paso a paso, aplicando principios ágiles incluso siendo un proyecto pequeño.

Herramientas y Consejos

Divide y vencerás

Descompón tu proyecto en tareas pequeñas. Un backlog en **Trello**, **Jira** o incluso una hoja de Google Sheets ayuda a priorizar.

Control de versiones

Usa **Git + GitHub/GitLab/Bitbucket**. Permite avanzar sin miedo a romper el proyecto, ya que siempre puedes volver atrás.

MVP primero

No pierdas semanas diseñando extras; construye lo esencial y valida. Ejemplo: un gestor de tareas solo necesita añadir y mostrar tareas en su primera versión.

Iteraciones temporizadas

Trabaja en ciclos cortos (sprints de 1-2 semanas en proyectos académicos o de empresa). Así limitas riesgos y mantienes el foco.

Automatiza pruebas

Incluso en proyectos pequeños, escribe tests simples (con pytest en Python o JUnit en Java) para asegurar que las versiones incrementales no rompan lo anterior.

Mitos y Realidades

❌ Mito: "Agile es solo para grandes corporaciones."
✔️ Realidad:

Los principios ágiles se aplican perfectamente en proyectos individuales o en equipos pequeños; de hecho, ahí es donde muestran más impacto inmediato.

❌ Mito: "Diseño incremental significa falta de planificación."
✔️ Realidad:

Hay planificación, pero flexible. Se establece una dirección general y se ajusta con cada iteración según el feedback.

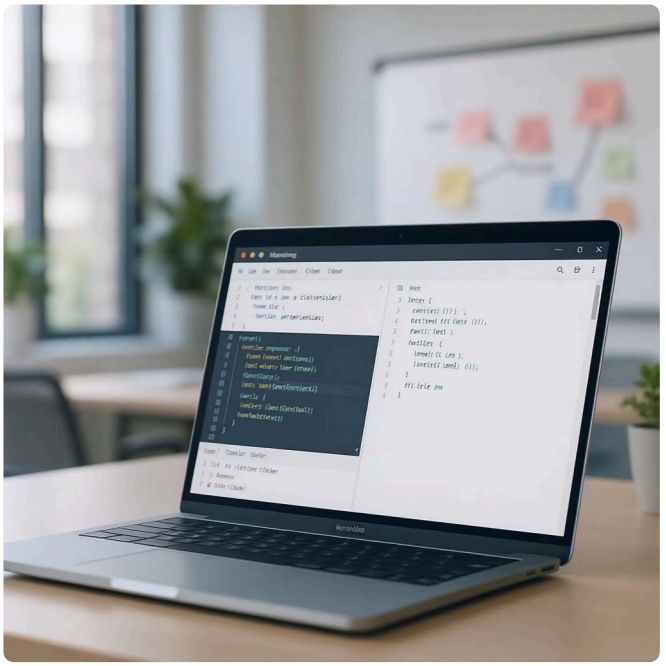
Resumen final

- Diseño incremental: construir software en pasos pequeños, siempre con una versión funcional.
- Metodologías ágiles: iteraciones cortas, retroalimentación constante y adaptación rápida.
- MVP: pieza mínima que valida la idea antes de invertir más recursos.
- Herramientas clave: Trello, Git, tableros de tareas, automatización de pruebas.
- Aplicable tanto a proyectos pequeños como a productos globales.

Sesión 4 – Refactorización y Código Limpio (Clean Code)

Un programa que "funciona" no siempre es un programa profesional. El código que se entiende fácilmente, se prueba sin complicaciones y se mantiene con confianza es el que perdura. Aquí entran en juego dos conceptos clave: **refactorización** y **Clean Code**.

- **Refactorización.** Es el proceso de mejorar la estructura interna del código sin alterar su comportamiento observable. No implica reescribir desde cero, sino introducir pequeños ajustes graduales: extraer funciones, renombrar variables, eliminar duplicados. El objetivo es hacer que el código sea más claro y fácil de extender.
- **Clean Code.** Popularizado por Robert C. Martin, propone principios y hábitos de escritura para lograr código comprensible y sostenible.



Los pilates del Clean Code incluyen:

Nombres significativos

Variables, funciones y clases deben explicar su propósito por sí mismas.

Funciones cortas y de un solo propósito

Evita bloques que hacen "de todo".

Evitar duplicación

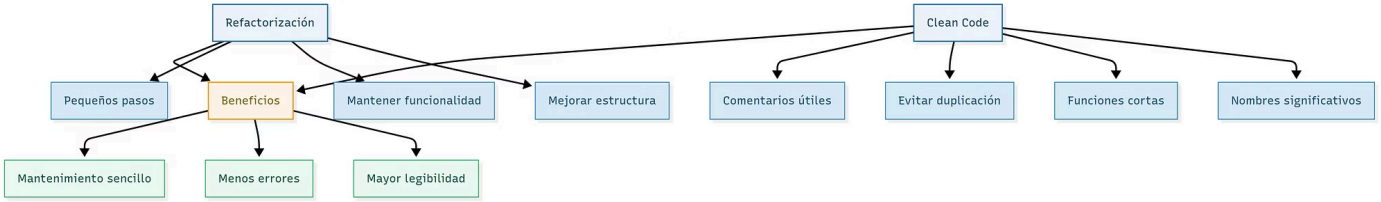
Centraliza la lógica común para prevenir inconsistencias.

Comentarios útiles

El código debe explicarse por sí mismo; los comentarios sirven para aclarar lo excepcional.

En proyectos reales, estas prácticas evitan el "código espagueti", reducen los errores al modificar y permiten que distintos desarrolladores trabajen sobre la misma base sin fricciones. En entornos empresariales, la calidad interna del código es un activo tan valioso como la funcionalidad externa.

Esquema visual



- La refactorización (A) se basa en ajustes pequeños sin alterar comportamiento (B–D).
- Clean Code (E) aporta principios claros (F–I).
- Ambos confluyen en beneficios prácticos (J–M).

Caso de Estudio – BBVA y la modernización de sistemas internos

Contexto.

BBVA contaba con aplicaciones internas críticas desarrolladas en Java con más de una década de antigüedad. El código funcionaba, pero la acumulación de parches lo había convertido en difícil de mantener y arriesgado de modificar.

Estrategia.

Refactorización progresiva. En lugar de reescribir sistemas completos, se aplicó la técnica *boy scout rule*: dejar cada módulo mejor de lo que se encontró.

Principios Clean Code. Renombraron variables genéricas como val1, objX a nombres expresivos (saldoCuenta, transaccionPendiente). Extrajeron funciones largas en métodos más pequeños como calcularIntereses() o validarCliente().

Automatización. Se reforzó la cobertura de tests unitarios y de integración para asegurar que la refactorización no rompiera funcionalidades críticas.

Herramientas de análisis. Utilizaron SonarQube para detectar duplicaciones, métodos demasiado largos y dependencias cíclicas.

Resultado.

El código pasó a ser más comprensible y fácil de auditar. Los tiempos de incorporación de nuevas funcionalidades se redujeron, y los errores en producción relacionados con regresiones disminuyeron en más de un 30% tras la refactorización continua.

Herramientas y Consejos

Refactoriza en pasos pequeños

Cambia un nombre, extrae una función, ejecuta tests, y confirma que nada se rompió. Herramientas como **IntelliJ IDEA** o **Visual Studio Code** ofrecen atajos para renombrar y extraer métodos de forma segura.

Automatiza la verificación

Usa frameworks de testing como **JUnit** (Java), **pytest** (Python) o **Jest** (JavaScript) para comprobar que tu refactorización mantiene el comportamiento.

Analiza tu código

Herramientas como **SonarQube**, **Checkstyle** (Java), **Pylint** (Python) o **ESLint** (JavaScript) identifican duplicaciones y malas prácticas automáticamente.

Adopta estándares de estilo

Configura *linters* y formateadores como **Prettier**, **Black** o **clang-format** para mantener uniformidad en el equipo.

Incorpora Clean Code como hábito

No esperes a que el proyecto "madure": aplicar principios desde el inicio reduce deuda técnica y facilita la colaboración.

Mitos y Realidades

❌ Mito: "Refactorizar significa rehacer todo el programa desde cero."

✔️ Realidad: La refactorización son ajustes controlados y graduales que mejoran la estructura manteniendo el mismo comportamiento externo.

❌ Mito: "El código limpio solo es importante para programadores novatos."

✔️ Realidad: Clean Code es esencial en entornos profesionales: reduce costes de mantenimiento, facilita la incorporación de nuevos miembros y disminuye errores en producción.

Resumen final

- Refactorización = mejorar estructura interna sin cambiar funcionalidad.
- Clean Code = principios de claridad: nombres expresivos, funciones cortas, sin duplicación, comentarios útiles.
- Beneficios: legibilidad, menos errores, mantenimiento más simple.
- Apóyate en herramientas de análisis, linters y tests automáticos.