

PROMETEO

Unidad 3: Ampliación en Estructuras de Control

El switch en Java ha sido, durante años, una de las estructuras de control más utilizadas para simplificar decisiones múltiples.

Sesión 9: Uso avanzado de switch (expresiones, enums).

Solo admitía tipos primitivos (int, char, byte, short) y cadenas (String desde Java 7), requería el uso obligatorio de break; para evitar la ejecución en cascada y no podía devolver valores directamente.

Con la llegada de **Java 12** y su consolidación en **Java 17**, el switch experimentó una transformación profunda, convirtiéndose en una **expresión** más flexible, segura y expresiva. Este nuevo enfoque permite **evaluar y devolver un valor**, simplificando el código y eliminando errores comunes como el *fall-through* (cuando se omite un break; por descuido).

Veamos la diferencia entre ambas versiones:

Versión clásica:

```
String dia = "LUNES";
int numero;
switch (dia) {
    case "LUNES":
    case "MARTES":
    case "MIÉRCOLES":
        numero = 1;
        break;
    case "JUEVES":
    case "VIERNES":
        numero = 2;
        break;
    default:
        numero = 3;
}
```

Versión moderna con switch expression:

```
String dia = "LUNES";
int numero = switch (dia) {
    case "LUNES", "MARTES", "MIÉRCOLES" -> 1;
    case "JUEVES", "VIERNES" -> 2;
    default -> 3;
};
```

El cambio es evidente: el código es más **compacto, legible y seguro**. Las flechas (->) sustituyen los bloques case tradicionales y cada caso devuelve directamente un valor.

Otra mejora significativa es el uso de **enums** (enumeraciones). Un enum define un conjunto cerrado de constantes, proporcionando mayor seguridad y legibilidad frente a valores literales (como cadenas). Por ejemplo:

```
enum DiaSemana {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO  
}
```

```
DiaSemana dia = DiaSemana.MARTES;  
int tipo = switch (dia) {  
    case LUNES, MARTES, MIERCOLES -> 1;  
    case JUEVES, VIERNES -> 2;  
    default -> 3;  
};
```

El uso de enum en switch evita errores de escritura ("MIERCOLES" vs "MIÉRCOLES"), facilita el autocompletado y permite al compilador advertir si se omite algún caso, mejorando la robustez del código.

En resumen, el nuevo switch:

Admite tipos avanzados

int, String, enum y, desde Java 17, algunos tipos de patrones (Pattern Matching).

Devuelve valores

Directamente, sin necesidad de variables auxiliares.

Evita errores

Por falta de break y ejecución en cascada.

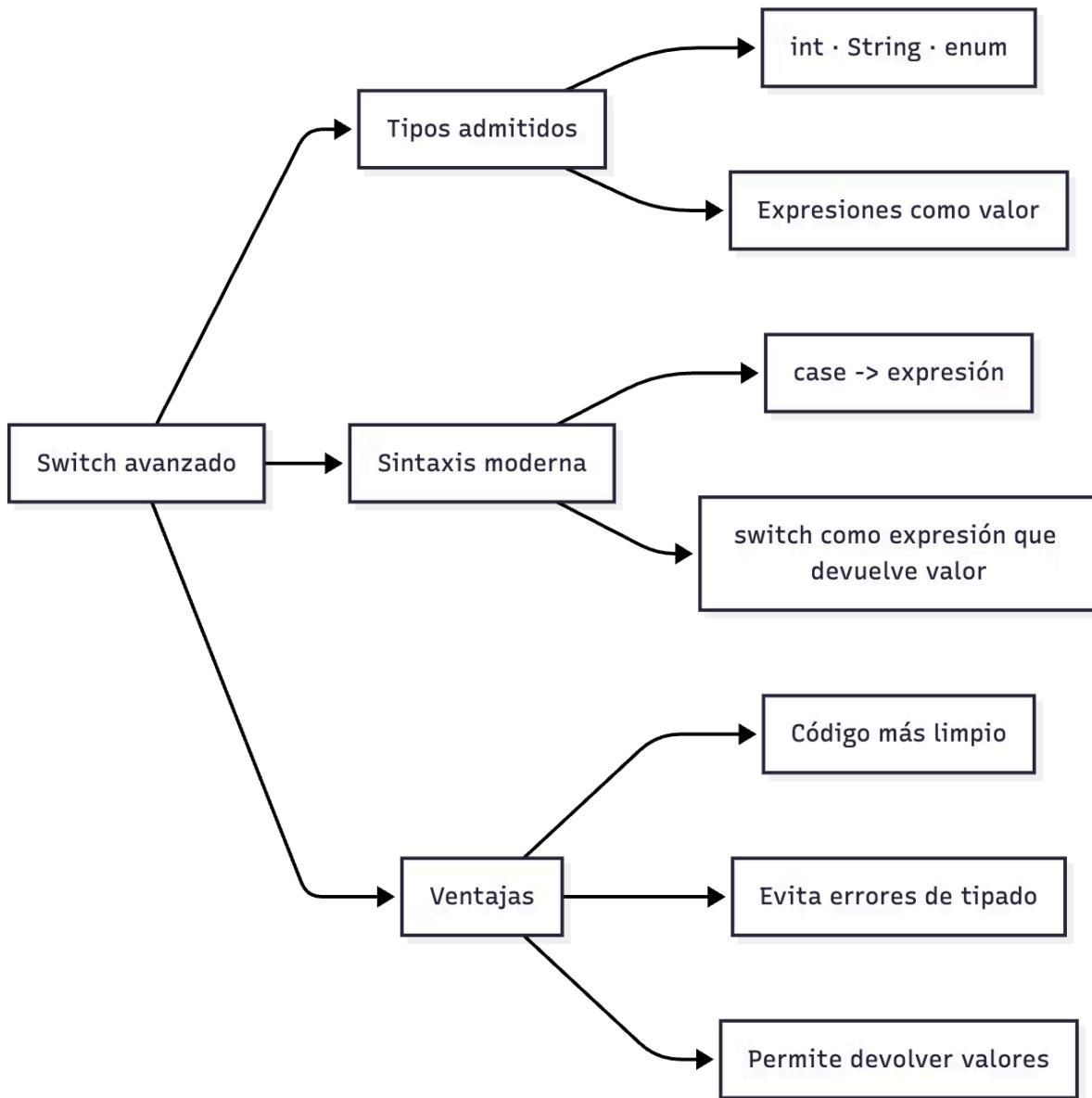
Mejora legibilidad

Y la mantenibilidad del código profesional.

En entornos profesionales —desde fintech hasta sistemas logísticos o e-commerce— estas mejoras impactan directamente en la productividad del equipo y la calidad del software, ya que reducen el número de líneas, errores de control y complejidad cognitiva.

Esquema Visual

A continuación se muestra el esquema conceptual del **switch avanzado** y sus elementos:



ⓘ Explicación del esquema:

- **Tipos admitidos:** representan los valores que el switch puede evaluar. A diferencia de versiones antiguas, ahora se incluyen tipos más expresivos como String y enum.
- **Sintaxis moderna:** introduce el uso del operador `->`, eliminando la necesidad de `break` y permitiendo que el switch sea una expresión que **retorna valores**.
- **Ventajas:** las mejoras sintácticas tienen efectos prácticos: menos líneas de código, mayor seguridad en la comparación de tipos y una lógica más declarativa.

Caso de Estudio: Airbnb y la gestión de estados de reserva



Contexto

Airbnb gestiona millones de reservas en tiempo real en todo el mundo. Cada reserva pasa por distintos **estados**: PENDIENTE, CONFIRMADA, CANCELADA, COMPLETADA, etc. Antes de adoptar enums y switch expressions, parte del código que gestionaba los flujos de reserva utilizaba cadenas y largas estructuras if-else, generando errores por diferencias mínimas en los textos o duplicidades.



Estrategia

El equipo de desarrollo decidió refactorizar su lógica de negocio definiendo un enum llamado EstadoReserva:

```
enum EstadoReserva {  
    PENDIENTE, CONFIRMADA, CANCELADA, COMPLETADA  
}
```

Y lo aplicó con un switch expression para calcular comisiones y penalizaciones:

```
double calcularComision(EstadoReserva estado) {  
    return switch (estado) {  
        case PENDIENTE -> 0.0;  
        case CONFIRMADA -> 0.15;  
        case CANCELADA -> 0.05;  
        case COMPLETADA -> 0.20;  
    };  
}
```

Además, el uso de enums permitió detectar **en tiempo de compilación** los casos no manejados. Si el equipo añadía un nuevo estado (por ejemplo, REPROGRAMADA), el compilador mostraba una advertencia indicando que el switch debía actualizarse. Esto redujo significativamente los fallos en producción.



Resultado

25%

Reducción de código

En flujos de reserva relacionados con estados.

0

Errores de comparación

Eliminación total de errores por texto.

100%

Mantenimiento

Más rápido ante nuevas reglas de negocio.

100%

Claridad

Mayor en cálculo de comisiones y penalizaciones.

El caso de Airbnb ilustra cómo una simple mejora en la sintaxis del lenguaje puede traducirse en **eficiencia operativa, robustez del sistema y mayor calidad del código** en proyectos reales de gran escala.



Herramientas y Consejos

Consejos prácticos

01

Usa enums para conjuntos fijos de valores

Cada vez que tengas una lista limitada de opciones (días de la semana, estados, tipos de usuario), define un enum. Evitarás errores por escritura y mejorarás la mantenibilidad.

02

Prefiere switch expressions con ->

La sintaxis moderna no solo es más limpia, sino que evita el clásico error de olvidarse un break entre casos, lo que antes generaba ejecuciones no deseadas en cascada.

03

Evita cadenas literales ("magic strings")

Si un valor representa una categoría predefinida, conviértelo en enum. Las cadenas son propensas a errores tipográficos y no ofrecen autocompletado.

04

Refactoriza estructuras de decisión largas

Si tu código tiene múltiples if-else anidados para comparar una misma variable, conviértelo en switch. Herramientas modernas pueden automatizar este cambio.

05

Combina switch con métodos funcionales

En Java moderno, puedes combinar switch con lambdas o streams para lógica declarativa más potente.

Herramientas recomendadas



IntelliJ IDEA

Refactor > "Convert if/else to switch": transforma automáticamente estructuras de decisión anidadas en un switch más legible.



Replit o JDoodle

Plataformas online ideales para probar la sintaxis moderna del switch sin necesidad de configurar un entorno completo.



JDK 17+

Las mejoras del switch están disponibles desde Java 12, pero alcanzan estabilidad y compatibilidad plena en JDK 17 (versión LTS).



SonarLint o PMD

Detectan patrones redundantes y recomiendan el uso de switch expressions cuando es más apropiado.

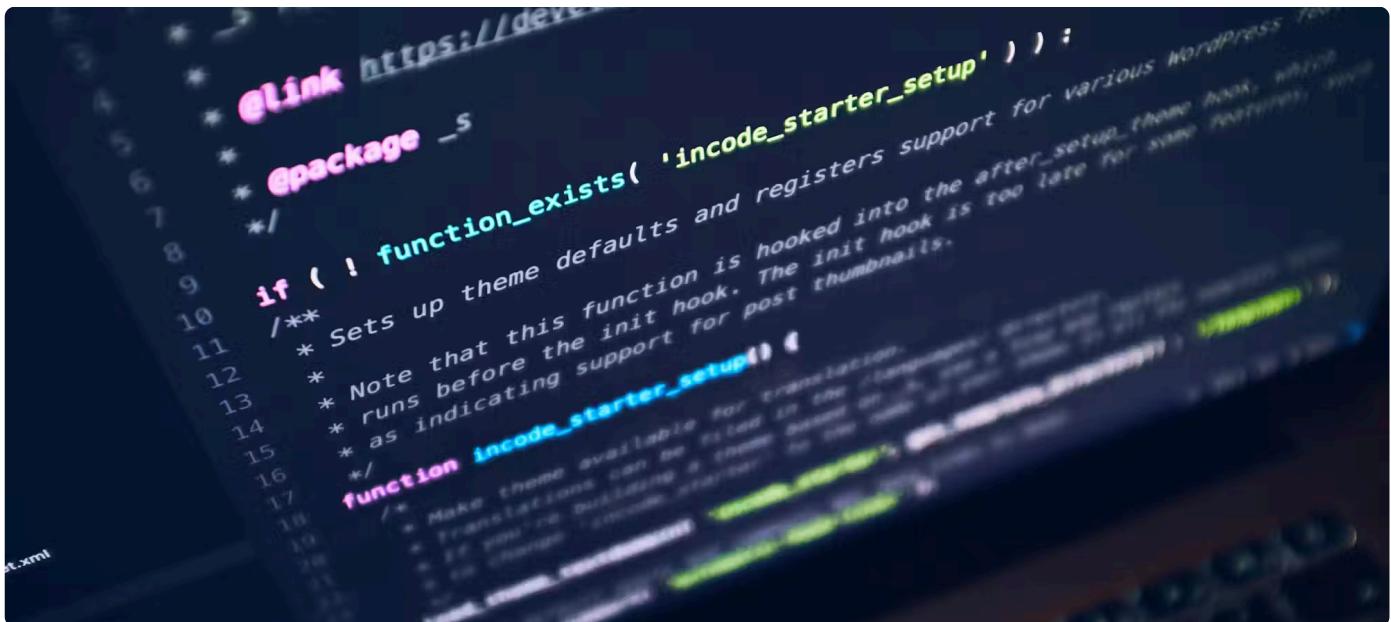
Mitos y Realidades

⊗ **X Mito:** "El switch solo sirve con enteros."

→ **FALSO.** Desde Java 7 admite `String`, y desde Java 12 admite `enum` y expresiones que devuelven valores. Incluso puede integrarse con *pattern matching* en versiones recientes del lenguaje.

⊗ **X Mito:** "El nuevo switch es solo una mejora estética."

→ **FALSO.** No se trata de estilo, sino de funcionalidad. El switch moderno devuelve valores, mejora la seguridad de tipos, evita errores de flujo y facilita el código declarativo. Su uso puede reducir la complejidad ciclomática de un método en hasta un 40%.



□ Resumen Final

- switch moderno = **expresión** que **devuelve valores**.
- Admite **int, String y enum** como tipos evaluables.
- Nueva sintaxis con case -> elimina el uso de break.
- Los **enums** mejoran la **seguridad, claridad y mantenimiento** del código.
- Ideal para **refactorizar** estructuras largas de if/else y manejar **casos fijos o categorizados**.

Sesión 10: Control de flujo con etiquetas (break, continue, return)

Las **estructuras de control de flujo** en Java son las responsables de determinar cómo se ejecutan las instrucciones dentro de un programa. Gracias a ellas, el código puede **repetir acciones, tomar decisiones y detener procesos** según condiciones específicas. Dentro de este conjunto, tres palabras clave desempeñan un papel esencial: break, continue y return.

Aunque todas interrumpen la secuencia normal de ejecución, cada una lo hace con un propósito y alcance diferente:

break

Interrumpe el bucle actual.

continue

Salta a la siguiente iteración del bucle.

return

Finaliza por completo la ejecución del método.

Estas tres instrucciones existen desde las primeras versiones de Java, pero un elemento menos conocido potencia su uso en situaciones más complejas: las **etiquetas** (*labels*).

Las etiquetas permiten **nombrar un bucle** o bloque de código para referirse a él de manera explícita cuando se usa break o continue. Esto resulta especialmente útil en **bucles anidados**, donde es necesario controlar la salida o el salto entre niveles.

Imagina el siguiente ejemplo sin etiquetas:

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        if (i == j) break; // Sale solo del bucle interno  
    }  
}
```

Este código detiene únicamente el bucle interno cuando i y j son iguales, pero el bucle externo (i) continúa. Ahora observa lo que ocurre con una **etiqueta**:

```
outer:  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        if (i == j) break outer; // Sale de ambos bucles  
    }  
}
```

Aquí, break outer; **rompe ambos bucles a la vez**, controlando el flujo con precisión y evitando el uso de banderas booleanas o condiciones adicionales.

En un nivel conceptual:

break	continue	return
Actúa como un interruptor que detiene el bucle actual.	Actúa como un acelerador que salta la iteración actual y pasa a la siguiente.	Es la salida definitiva del método, independiente mente del número de bucles anidados.

Por ejemplo:

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) continue; // salta cuando i vale 5  
    if (i == 8) break; // termina el bucle cuando i vale 8  
}
```

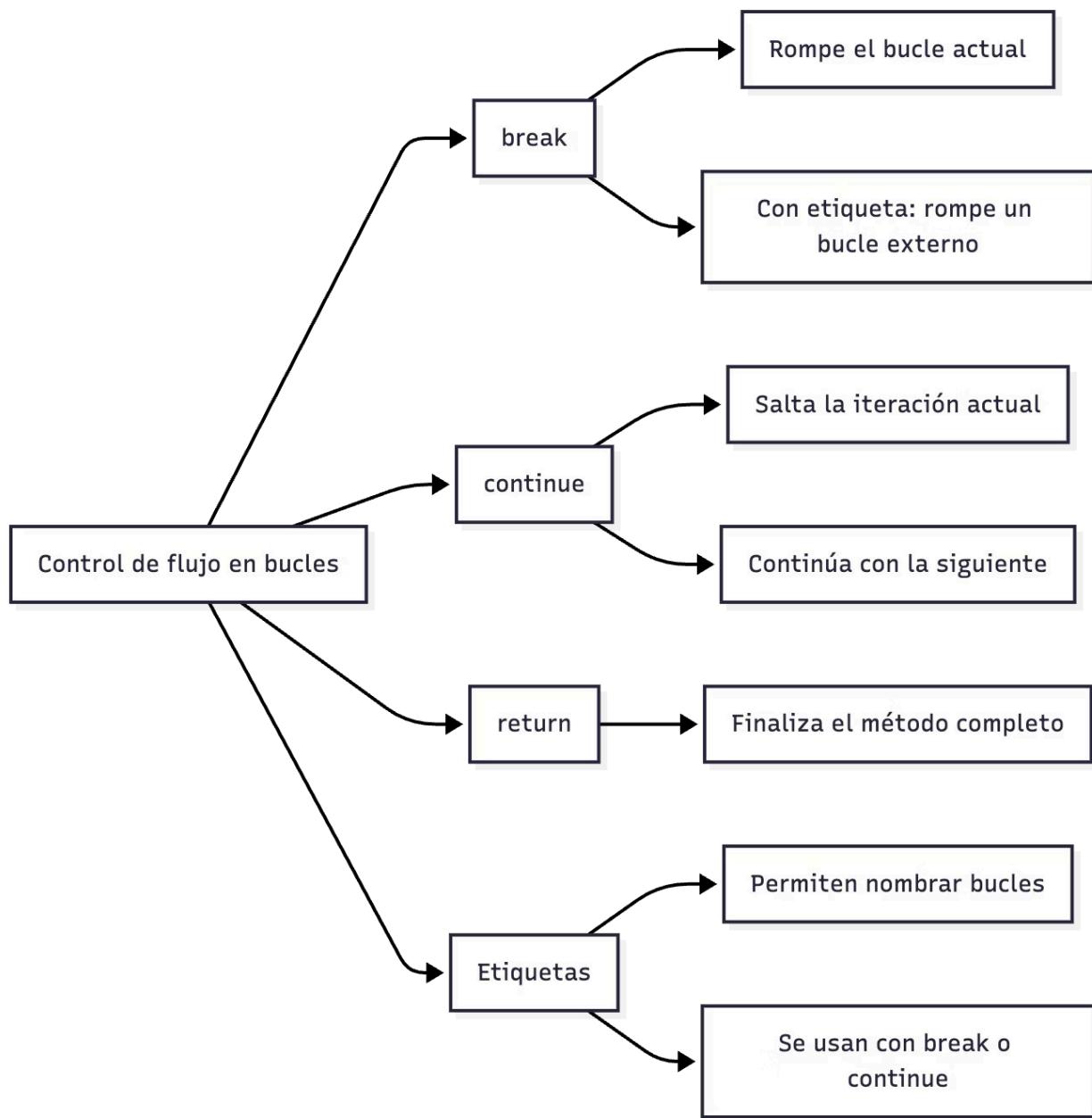
Y si dentro del bucle hay una condición que invalida el propósito del método completo:

```
for (int n : numeros) {  
    if (n < 0) return; // termina el método completamente  
}
```

El dominio de estas instrucciones —y especialmente de las **etiquetas**— es fundamental en entornos donde la eficiencia y el control preciso son prioritarios, como la manipulación de datos, la navegación por estructuras de matrices o la gestión de flujos en sistemas de inventario o logística.

Esquema Visual

El siguiente diagrama resume las tres instrucciones clave y su interacción con las etiquetas:



ⓘ Interpretación del esquema:

- **break** tiene un alcance limitado al bucle donde se ejecuta, salvo que se use con etiqueta, en cuyo caso puede salir de múltiples niveles de anidación.
- **continue** afecta solo al bucle actual y permite evitar código innecesario cuando una condición no se cumple.
- **return** rompe todo el flujo del método, sin importar en qué nivel esté.
- Las **etiquetas** amplían el control de break y continue, permitiendo acciones dirigidas sobre bucles externos.

Caso de Estudio: Amazon y la optimización de búsquedas internas



Contexto

En los sistemas de inventario de Amazon, millones de productos se buscan, clasifican y filtran constantemente. En este tipo de procesos, recorrer estructuras de datos anidadas es habitual. Imagina un escenario donde una matriz representa el inventario por almacén y ubicación, y se necesita **detener la búsqueda** en cuanto se encuentra un producto específico.



Estrategia

El equipo de desarrollo implementó una solución basada en break con etiquetas para optimizar la búsqueda, evitando recorrer innecesariamente toda la estructura.

Ejemplo simplificado del patrón utilizado:

```
search:  
for (int i = 0; i < almacenes.length; i++) {  
    for (int j = 0; j < estanterias[i].length; j++) {  
        if (estanterias[i][j].equals("PRODUCTO-XYZ")) {  
            System.out.println("Producto encontrado en almacén " + i);  
            break search; // Sale de ambos bucles  
        }  
    }  
}
```



Resultado

- Reducción del tiempo de búsqueda en matrices grandes hasta un **40%**.
- Código más claro y sin necesidad de variables auxiliares como boolean encontrado.
- Mayor legibilidad en los procesos de búsqueda y menor riesgo de errores en la lógica de bucles.

El caso demuestra cómo un uso inteligente de **break etiquetado** puede tener un **impacto real en la eficiencia** de sistemas complejos. En entornos de alta demanda como Amazon, optimizar la cantidad de iteraciones puede representar **miles de operaciones por segundo** ahorraditas.

Herramientas y Consejos

Consejos prácticos

01

Usa etiquetas solo cuando sea necesario

En la mayoría de los casos, un break normal o una función auxiliar bastan. Sin embargo, en bucles anidados donde el flujo depende de varios niveles, las etiquetas simplifican el control.

02

Comenta siempre las etiquetas

Al nombrar una etiqueta (outer, loop1, search), añade un comentario que explique su propósito. Un código con etiquetas sin contexto puede volverse confuso rápidamente.

03

Evita anidar más de dos bucles

Si necesitas tres o más niveles de bucles anidados, probablemente el diseño de tu algoritmo pueda optimizarse. Considera extraer partes en métodos o usar estructuras de datos más eficientes.

04

Combina return con validaciones tempranas

En métodos complejos, puedes usar return para salir de la ejecución cuando una condición no se cumple. Este patrón, conocido como "**early return**", mejora la legibilidad.

05

Prueba el flujo paso a paso

Usa un **depurador (debugger)** para seguir la ejecución de cada bucle. Esto ayuda a comprender exactamente cómo se comporta break, continue o return en diferentes contextos.

Herramientas



IntelliJ IDEA / Eclipse Debugger

Permiten ejecutar el código paso a paso y observar cuándo y cómo se activan las instrucciones de control.



Pythontutor.com (modo Java)

Excelente herramienta visual para ver el recorrido de bucles y el salto de flujo en tiempo real.



SonarLint

Detecta bucles excesivamente anidados o flujos de control innecesariamente complejos.



VisualVM o JProfiler

Para analizar rendimiento cuando el número de iteraciones afecta al tiempo de ejecución.

Mitos y Realidades

- ⊗ **✗ Mito:** "Las etiquetas son una mala práctica y deben evitarse siempre."

→ **FALSO.** Las etiquetas mal utilizadas pueden complicar el código, pero cuando se aplican en estructuras de búsqueda o validación de datos anidados, **mejoran la eficiencia y reducen la complejidad.** Como toda herramienta, su valor depende del contexto y del criterio del programador.

- ⊗ **✗ Mito:** "break, continue y return son equivalentes."

→ **FALSO.** Cada una tiene un propósito distinto y un alcance diferente:

- `break` detiene el bucle actual (o el etiquetado).
- `continue` omite la iteración actual.
- `return` termina el método completo.

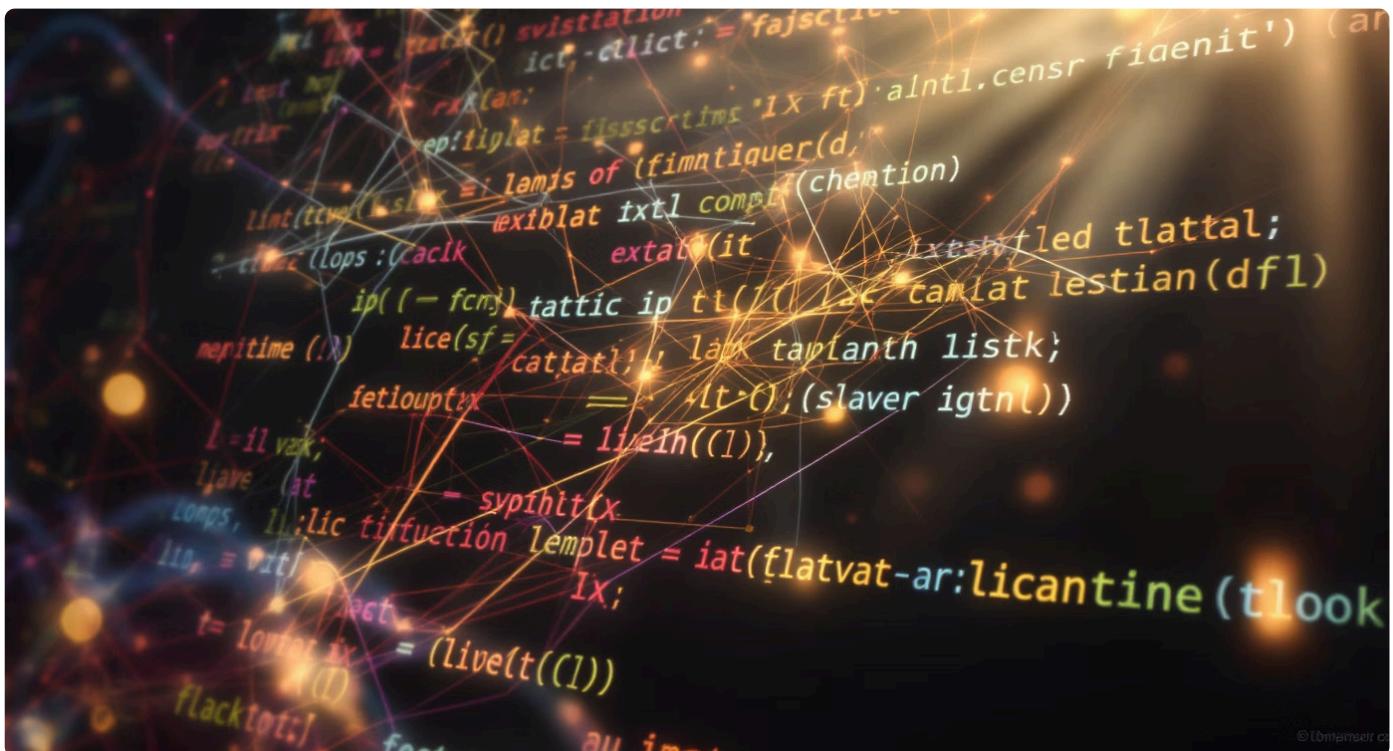
Confundirlas puede provocar errores lógicos graves o comportamientos inesperados.

▢ Resumen Final

- `break` **interrumpe** el bucle actual o el etiquetado.
- `continue` **salta** la iteración actual y pasa a la siguiente.
- `return` **finaliza el método completo.**
- Las **etiquetas** permiten controlar bucles externos en anidaciones múltiples.
- Deben usarse **con moderación** y siempre **documentadas**.
- Son especialmente útiles en **búsquedas, validaciones y algoritmos de optimización.**

Sesión 11: Buenas prácticas en estructuras anidadas (cómo evitar el "código espagueti").

Cuando comienzas a programar, es natural utilizar estructuras anidadas —bucles dentro de bucles, condicionales dentro de condicionales— para resolver problemas complejos. Sin embargo, con el crecimiento del código, esa forma de estructuración puede volverse peligrosa. Surge entonces lo que los desarrolladores llaman "**código espagueti**", un término que describe un programa con tantas dependencias y niveles de anidación que se asemeja a un plato de fideos enredados: imposible de seguir con la vista.



El "código espagueti" no es un error sintáctico; **es un error de diseño**. Suele aparecer cuando los programadores buscan soluciones rápidas sin pensar en la escalabilidad o la legibilidad. En un equipo profesional, este tipo de código genera tres grandes problemas:

Dificultad de lectura

Entender la lógica requiere saltar entre múltiples niveles de if, else, for o while.

Errores lógicos

Las condiciones se vuelven confusas y es fácil introducir errores que no se detectan hasta tiempo después.

Baja mantenibilidad

Cambiar un bloque puede afectar otro sin que sea evidente.

Por ejemplo, un código típico de un programador principiante puede verse así:

```
if (usuario != null) {  
    if (usuario.activo) {  
        if (usuario.tienePermisos()) {  
            if (pedido != null) {  
                if (pedido.total > 0) {  
                    procesarPedido(pedido);  
                } else {  
                    System.out.println("El pedido está vacío.");  
                }  
            } else {  
                System.out.println("Pedido no encontrado.");  
            }  
        }  
    }  
}
```

Aunque el código funciona, tiene **cinco niveles de anidación**, lo que lo hace difícil de mantener. Ahora veamos cómo aplicar **buenas prácticas** para simplificarlo:

- **Evita más de tres niveles de anidación.**
- **Usa retornos tempranos (return)** para salir de condiciones innecesarias.
- **Divide la lógica en métodos más pequeños**, cada uno con una sola responsabilidad.
- **Utiliza polimorfismo o switch modernos** en lugar de largas cadenas de if/else.

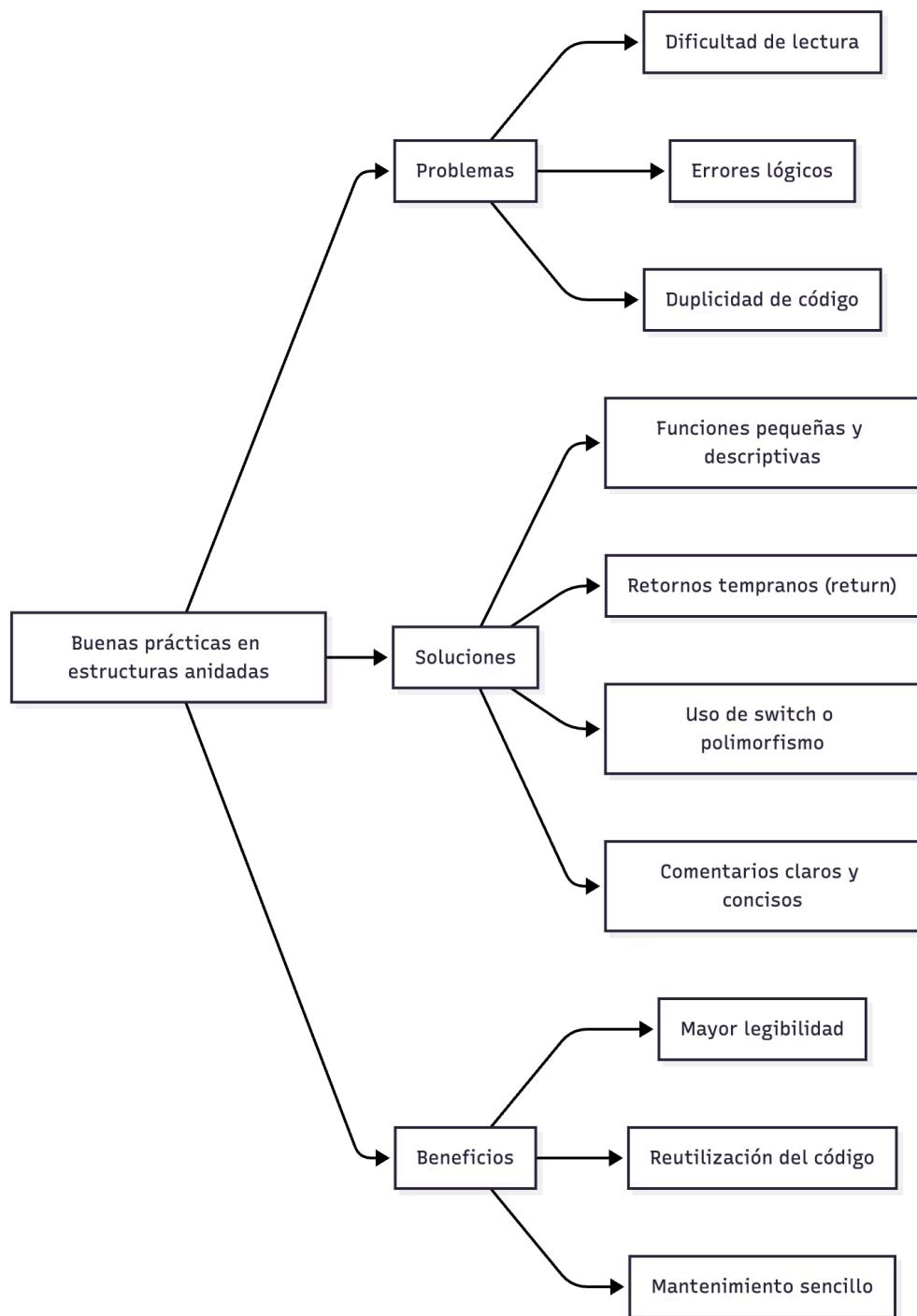
Un ejemplo refactorizado sería:

```
void procesarPedidoSeguro(Usuario usuario, Pedido pedido) {  
    if (usuario == null || !usuario.activo || !usuario.tienePermisos()) return;  
    if (pedido == null || pedido.total <= 0) {  
        System.out.println("Pedido no válido.");  
        return;  
    }  
    procesarPedido(pedido);  
}
```

Este código hace exactamente lo mismo que el anterior, pero con **la mitad de las líneas** y una lógica mucho más clara. La clave está en adoptar el principio de diseño de **código limpio (Clean Code)**: *cada función debe hacer una sola cosa y hacerla bien*.

En un entorno profesional, mantener la claridad en estructuras de control es esencial. Equipos de desarrollo, QA y operaciones deben entender la lógica sin necesidad de "descifrar" bucles o condicionales. Un código limpio no solo es más rápido de escribir: es más rápido de entender, probar y mantener.

Esquema Visual



ⓘ Explicación del esquema:

- En la parte superior se encuentran los **problemas** que genera el exceso de anidación.
- En el centro se muestran las **soluciones prácticas**, basadas en principios de refactorización y diseño modular.
- En la parte inferior se resumen los **beneficios concretos**: código limpio, claro y sostenible en el tiempo.

Caso de Estudio: Refactorización en Spotify



Contexto

Spotify maneja uno de los sistemas de recomendación más complejos del mundo, basado en miles de variables por usuario: hábitos de escucha, ubicación, dispositivos, hora del día, entre otros. Con el crecimiento del servicio, el código original del motor de recomendaciones había caído en un patrón clásico de **condicionales anidados**. Cada nueva funcionalidad añadía un nuevo "nivel" de decisión.



Estrategia

El equipo de ingeniería decidió emprender una **refactorización progresiva**. El objetivo era pasar de una lógica basada en `if/else` a una arquitectura modular con métodos específicos para cada tipo de recomendación.

Por ejemplo, antes existía una lógica similar a esta (simplificada):

```
if (usuario.premium) {  
    if (usuario.escuchaFrecuente("rock")) {  
        if (horaDelDia < 12) {  
            recomendarPlaylist("Morning Rock");  
        } else {  
            recomendarPlaylist("Rock Hits");  
        }  
    } else {  
        if (usuario.escuchaFrecuente("pop")) {  
            recomendarPlaylist("Pop Daily");  
        }  
    }  
}
```

La refactorización dividió la lógica en funciones independientes, aplicando **el principio de responsabilidad única (SRP)**:

```
void recomendar(Usuario usuario) {  
    if (!usuario.premium) return;  
    if (esFanDeRock(usuario))  
        recomendarRock(usuario);  
    else if (esFanDePop(usuario))  
        recomendarPop(usuario);  
}  
  
void recomendarRock(Usuario usuario) {  
    String playlist = horaDelDia < 12 ?  
        "Morning Rock" : "Rock Hits";  
    recomendarPlaylist(playlist);  
}
```



Resultado

30%

Reducción de código

El código final fue más corto y eficiente.

15%

Mejora de rendimiento

Los tiempos de ejecución mejoraron gracias a la eliminación de redundancias.

50%

Reducción de errores

Se redujo la tasa de errores de regresión (errores que reaparecen tras cambios).

Este caso muestra que reducir la anidación no solo mejora la legibilidad, sino también la **eficiencia, escalabilidad y capacidad de prueba** del software



Herramientas y Consejos

Consejos prácticos

Divide funciones largas en submétodos con nombres descriptivos

Si un método supera las 20 líneas o contiene más de tres niveles de indentación, es señal de que debe dividirse. Nombres claros como validarUsuario(), procesarPedido(), o generarReporte() facilitan la lectura.

Usa retornos tempranos (return) para evitar bloques anidados

En lugar de anidar condiciones negativas, sal del método tan pronto como una condición no se cumpla. Esto mejora la claridad y reduce errores.

Refactoriza condicionales con switch o polimorfismo

Cuando tengas muchos if/else que dependen de un tipo o estado, considera usar switch moderno o implementar clases específicas para cada comportamiento (patrón *State* o *Strategy*).

Aplica el principio "Clean Code"

Cada método debe tener **una única responsabilidad**. Si un método "hace demasiadas cosas", es un candidato claro a refactorización.

Documenta de forma mínima pero útil

Los comentarios deben explicar el "por qué", no el "cómo". Un código bien estructurado debería ser casi autoexplicativo.

Herramientas recomendadas

• SonarLint

Analiza el código en tiempo real y alerta sobre anidaciones excesivas, complejidad ciclomática y funciones demasiado largas.

• IntelliJ Code Smells Detector

Identifica patrones de código que indican necesidad de refactorización.

• ReSharper

Para IntelliJ o Visual Studio Code: ofrece sugerencias automáticas para dividir funciones complejas.

• PMD

Herramienta de análisis estático que detecta malas prácticas y complejidad innecesaria en el código Java.

Estas herramientas se integran con CI/CD para asegurar código de calidad en entornos profesionales.

Mitos y Realidades

- ⊗ **✗ Mito:** "Más anidación significa mayor control del programa."

→ **FALSO.** En realidad, **más anidación indica falta de diseño modular**. Un código con demasiados niveles de decisión suele ser más frágil y difícil de modificar. El verdadero control proviene de dividir el problema en unidades pequeñas y bien delimitadas.

- ⊗ **✗ Mito:** "La legibilidad es menos importante que la eficiencia."

→ **FALSO.** En la mayoría de los proyectos empresariales, **el mantenimiento representa hasta el 70 % del coste total del software**. La eficiencia puede optimizarse, pero un código ilegible ralentiza el trabajo de todo el equipo y multiplica los errores.

```
38     self.file.seek(0)
39     self.fingerprints.append(fp + os.linesep)
40
41     @classmethod
42     def from_settings(cls, settings):
43         debug = settings.getbool('superuser.debug')
44         return cls(job_dir(settings), debug)
45
46     def request_seen(self, request):
47         fp = self.request_fingerprint(request)
48         if fp in self.fingerprints:
49             return True
50         self.fingerprints.add(fp)
51         if self.file:
52             self.file.write(fp + os.linesep)
53             self.fingerprint(self, request)
```

□ Resumen Final

- Evita **más de tres niveles de anidación**.
- Divide el código en **funciones pequeñas y descriptivas**.
- Utiliza **retornos tempranos (return)** para simplificar condiciones.
- Refactoriza con **switch, enums o polimorfismo**.
- Menos anidación = más legibilidad, menos errores, más productividad.

Sesión 12: Patrones de control de flujo: "state machine"

En el desarrollo de software, es común que un programa deba **cambiar de comportamiento según su situación actual**. Piensa, por ejemplo, en una aplicación que gestiona sesiones de usuario: puede estar en estado *inicial*, *logueado* o *cerrado*. Cada estado determina qué acciones son válidas y cuáles no. Intentar gestionar esa lógica únicamente con condicionales (*if/else*) o *switch* puede funcionar al principio, pero a medida que los estados y las transiciones crecen, el código se vuelve confuso y difícil de mantener.

Ahí entra en juego el **patrón de control de flujo "State Machine" o máquina de estados**, una herramienta esencial para **estructurar flujos complejos de manera predecible, ordenada y extensible**.



- ⓘ Una **máquina de estados** (FSM, *Finite State Machine*) es un modelo conceptual donde un sistema puede encontrarse en uno de varios estados posibles, y las acciones que puede realizar dependen del estado actual. Cada transición entre estados está definida explícitamente, lo que evita inconsistencias y permite modelar flujos reales de forma intuitiva.

En Java, una máquina de estados puede implementarse de varias formas, desde la más simple hasta la más modular:

Con un enum y un switch:

```
enum EstadoSesion { INICIO,  
LOGUEADO, SALIDA }
```

```
EstadoSesion estado =  
EstadoSesion.INICIO;
```

```
switch (estado) {  
    case INICIO -> estado =  
EstadoSesion.LOGUEADO;  
    case LOGUEADO -> estado =  
EstadoSesion.SALIDA;  
    case SALIDA ->  
System.out.println("Sesión  
finalizada");  
}
```

Este enfoque es excelente para flujos simples, donde hay pocos estados y las transiciones son lineales.

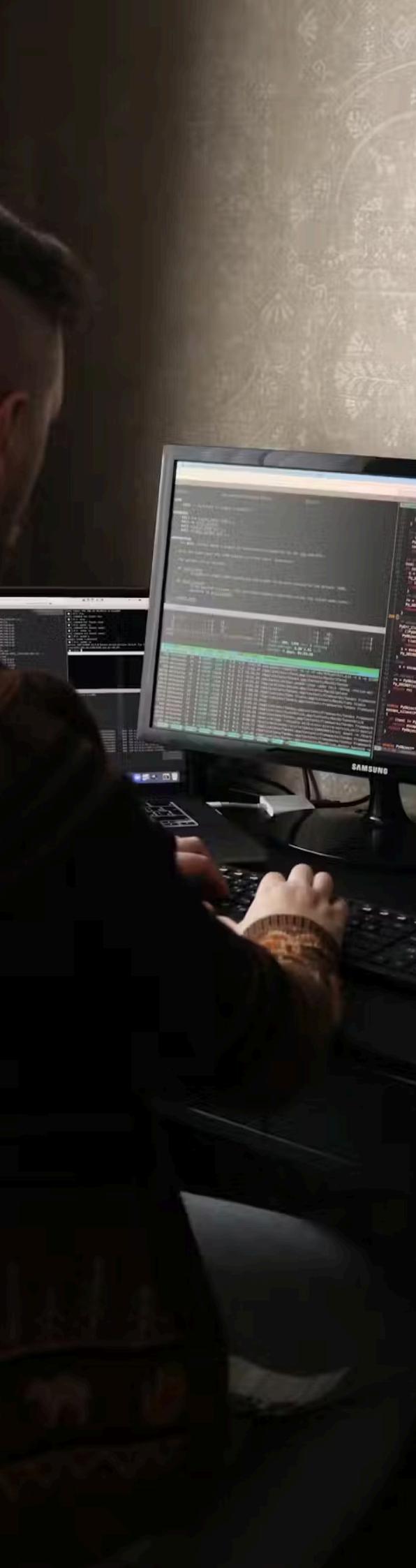
Con clases separadas (patrón State):

Cada estado se convierte en una clase que implementa un comportamiento común:

```
interface Estado {  
    void ejecutarAccion();  
}  
  
class EstadoInicio implements Estado {  
    public void ejecutarAccion() {  
        System.out.println("Iniciando sesión...");  
    }  
}  
  
class EstadoLogueado implements Estado {  
    public void ejecutarAccion() {  
        System.out.println("Usuario logueado.");  
    }  
}  
  
class EstadoSalida implements Estado {  
    public void ejecutarAccion() {  
        System.out.println("Cerrando sesión...");  
    }  
}
```

Este enfoque es más escalable y permite añadir nuevos estados sin modificar el código existente, siguiendo el **Principio Abierto/Cerrado (OCP)** de la programación orientada a objetos.

En sistemas grandes (como una aplicación de pagos, un flujo de pedidos o un videojuego), las máquinas de estados permiten **controlar el flujo sin depender de largas cadenas de condicionales**, mejorando la claridad y la capacidad de depuración.



Ventajas de usar State Machine



Claridad estructural

Cada estado está bien definido y las transiciones son explícitas.



Mantenibilidad

Añadir o modificar un estado no rompe la lógica general.



Reutilización

Los estados pueden implementarse como objetos reutilizables.

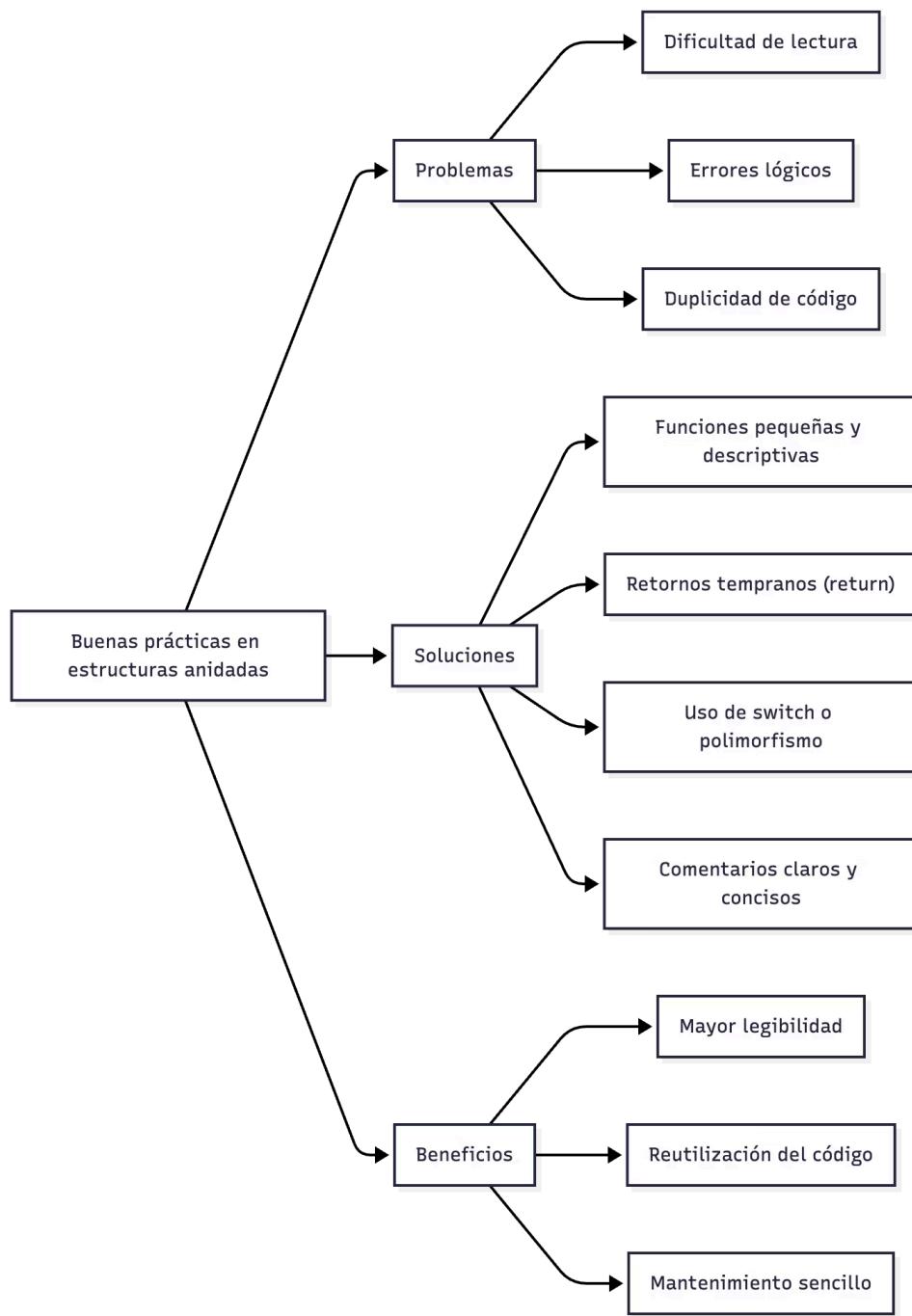


Seguridad

Se evitan combinaciones de condiciones imposibles o incoherentes.

Una máquina de estados bien diseñada actúa como un **mapa del comportamiento del sistema**, algo que todo desarrollador puede seguir visualmente y mantener con facilidad.

Esquema Visual



ⓘ Explicación del esquema:

- La parte superior explica la **naturaleza conceptual** del patrón: un sistema finito con estados delimitados y transiciones claras.
- En el centro, se ilustran las **formas de implementación** más comunes en Java, desde la sintaxis sencilla con switch hasta el enfoque orientado a objetos.
- En la parte inferior, se muestran los **contextos profesionales** donde más se aplica: videojuegos, sistemas de autenticación, procesos automáticos o incluso robots industriales.

Caso de Estudio: Netflix y las fases de reproducción de video

Contexto

Netflix reproduce más de mil millones de horas de video cada semana. Cada vez que un usuario interactúa con el reproductor (pausa, reanuda, cambia de capítulo, salta la intro...), el sistema debe **reaccionar correctamente según el estado actual** del video. En los inicios, estas acciones se manejaban mediante múltiples if/else, lo que generaba errores difíciles de rastrear, especialmente cuando se añadían nuevas funcionalidades.



Estrategia

El equipo de ingeniería adoptó una **máquina de estados para gestionar las fases de reproducción**, definiendo estados como:

- BUFFERING
- PLAYING
- PAUSED
- STOPPED
- ERROR

Cada estado conoce cuáles transiciones son válidas:

- De BUFFERING → se puede pasar solo a PLAYING o ERROR.
- De PLAYING → se puede pasar a PAUSED o STOPPED.
- De PAUSED → se puede volver a PLAYING o terminar (STOPPED).

La implementación inicial usó un enum y un switch:

```
enum EstadoVideo { BUFFERING, PLAYING, PAUSED,  
STOPPED, ERROR }

void cambiarEstado(EstadoVideo estado) {  
    switch (estado) {  
        case BUFFERING -> System.out.println("Cargando  
contenido...");  
        case PLAYING -> System.out.println("Reproduciendo  
video...");  
        case PAUSED -> System.out.println("Video pausado.");  
        case STOPPED -> System.out.println("Reproducción  
detenida.");  
        case ERROR -> System.out.println("Error en la  
conexión.");  
    }  
}
```

Con el tiempo, escalaron a una **arquitectura basada en clases**, donde cada estado gestionaba su propio comportamiento:

```
interface Estado {  
    void reproducir();  
    void pausar();  
    void detener();  
}

class EstadoPlaying implements  
Estado {  
    public void reproducir() { /* ya  
en reproducción */ }  
    public void pausar() { /*  
transición a pausa */ }  
    public void detener() { /*  
transición a stop */ }  
}
```

Resultado

60%

Reducción de errores

En transiciones inválidas.

100%

Facilidad de extensión

Para añadir nuevas funciones
sin modificar el código base.

100%

Mejora en trazabilidad

Los logs de transición
permitieron depurar
incidencias en tiempo real.

El caso de Netflix es un ejemplo claro de cómo el uso de **máquinas de estados** permite convertir un flujo complejo y dinámico en un sistema ordenado, mantenible y escalable



Herramientas y Consejos

Consejos prácticos



Define claramente los estados posibles

Antes de escribir una sola línea de código, anota o dibuja todos los estados que puede tener tu sistema y qué transiciones son válidas entre ellos. Esto previene ambigüedades y errores lógicos.

Evita usar múltiples variables booleanas

En lugar de `isLogged`, `isPaused`, `isFinished`, usa un único enum Estado. Es más legible y previene combinaciones imposibles (por ejemplo, estar "pausado y terminado" a la vez).

Aplica el patrón State cuando el flujo sea complejo

Si tu aplicación tiene muchos estados o transiciones, implementa clases separadas para cada estado. Esto evita condicionales gigantes y favorece la extensión del sistema sin modificar el código existente.

Dibuja el diagrama de estados antes de programar

Las herramientas visuales ayudan a entender cómo fluye el sistema. Si no puedes dibujarlo, probablemente tu flujo sea demasiado complejo y necesite simplificación.

Usa logs o mensajes de depuración

Registrar cada cambio de estado (`System.out.println` o `Logger.info`) facilita el seguimiento de errores en tiempo real y mejora la trazabilidad del comportamiento del sistema.

Herramientas recomendadas



draw.io / Lucidchart

Permiten crear diagramas de estados y flujos de transición visuales antes de codificar.



PlantUML

Genera automáticamente diagramas UML a partir de texto o comentarios en el código.



IntelliJ UML Viewer

Visualiza dependencias y relaciones entre clases si usas el patrón State orientado a objetos.



Visual Paradigm

Herramienta profesional para modelar sistemas basados en máquinas de estados, útil en proyectos empresariales o de ingeniería de software.

Estas herramientas son ampliamente utilizadas por ingenieros de software en diseño de arquitecturas, especialmente cuando se trabaja en sistemas distribuidos o aplicaciones con múltiples procesos simultáneos.

```
length, c=11) } a.memo
    return n.each(a)
function() { return e
nla){var b=[["res
tion(){return e.de
live).fail(c.reject
2].disable,b[2]
function(a,b,c)
fail(g.reject):-
n--n.readyWait>
"onstatechange",K
addEventListener("
etimeout(f,50)}]()
me "div"),e.style.
3),c.remove
^(?:\{[\w\W]*\}
data(a,b,c)
if(k&&j[k]&&
void 0!==d&&(g
ranner(case)):b.in
0,"embed":!0
removeData:function(
camelCase(d,s)
removeData,removeD
function(_removeD
removeData,dequ
```

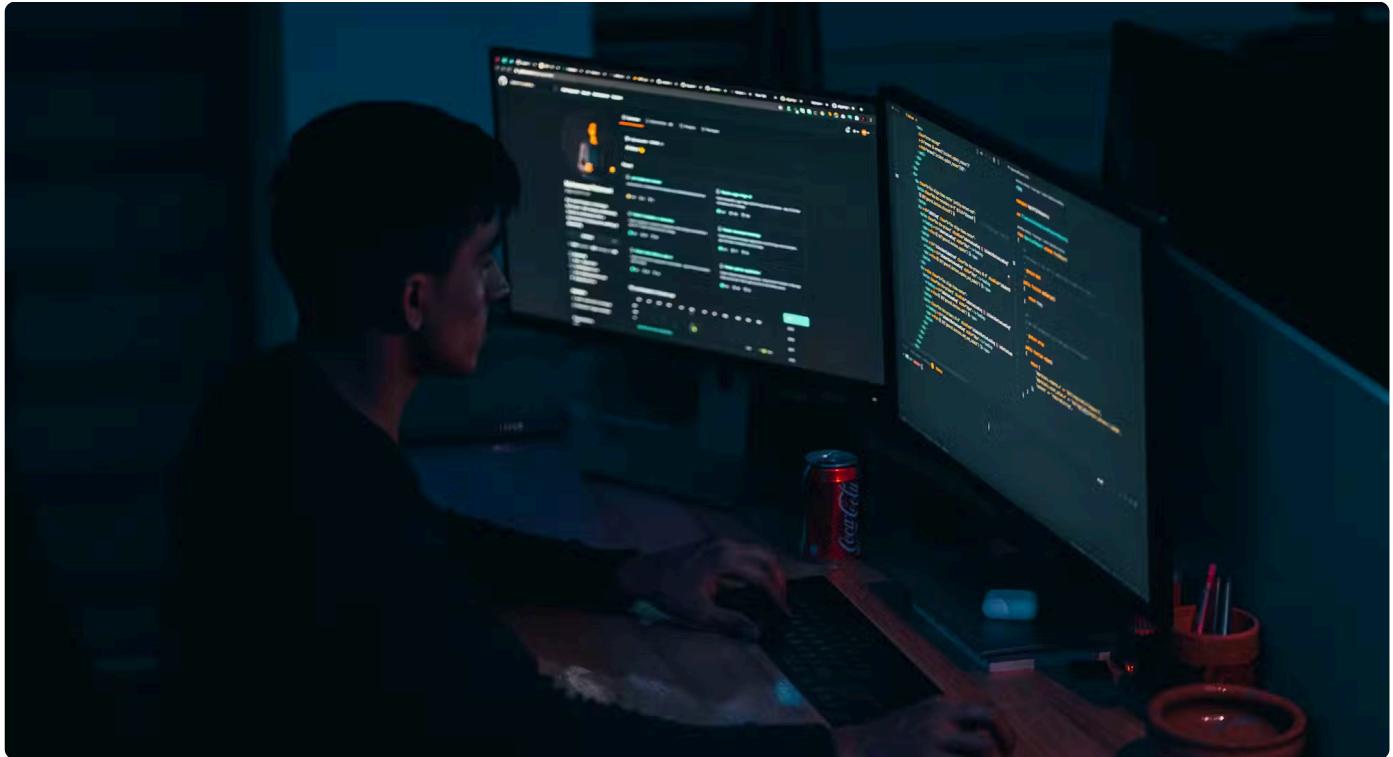
Mitos y Realidades

⊗ **✗ Mito:** "Las máquinas de estados son solo para videojuegos."

→ **FALSO.** Aunque se usan en motores de juego, su aplicación abarca **procesos industriales, sistemas de autenticación, asistentes virtuales, e-commerce** o cualquier flujo que dependa de una secuencia lógica de etapas.

⊗ **✗ Mito:** "Las máquinas de estados complican el código."

→ **FALSO.** Al contrario, **lo simplifican**. Organizan la lógica, evitan condicionales anidados y facilitan la extensión del sistema. Lo que puede parecer más código al principio, en realidad es **más claridad y menos mantenimiento futuro**.



▢ Resumen Final

- Una **máquina de estados** organiza el flujo del programa en **etapas definidas**.
- Puede implementarse con **enum + switch** o con **clases específicas por estado**.
- **Evita** usar múltiples booleanos: usa un solo estado global.
- Es ideal para **flujos secuenciales** o procesos con transiciones claras.
- **Mejora la legibilidad**, reduce errores y simplifica el mantenimiento del código.

Sesión 13: Estrategias de depuración en estructuras de control

La **depuración** (o *debugging*) es una de las habilidades más valiosas que puede desarrollar un programador profesional. No se trata solo de "encontrar errores", sino de **entender cómo y por qué el código no se comporta como esperas**. En estructuras de control complejas —como bucles anidados, condicionales o switch— los errores más frecuentes no son de sintaxis, sino **de lógica**: el código compila correctamente, pero produce resultados incorrectos.

Depurar no es un signo de debilidad; es una parte esencial del proceso de desarrollo. Ningún programa importante funciona a la primera, y los mejores desarrolladores no son los que nunca fallan, sino los que **detectan y corrigen errores con rapidez y método**.

En estructuras de control, los errores más comunes incluyen:

- Condiciones booleanas incorrectas (if ($x = 5$) en lugar de if ($x == 5$)).
- Variables no inicializadas o modificadas dentro del bucle en orden incorrecto.
- Bucles infinitos por condiciones mal formuladas (while(true) sin control de salida).
- Saltos de flujo inesperados en break, continue o return.

Para enfrentarte a ellos, necesitas **estrategias de depuración estructuradas**, no simples intuiciones. Las principales son:



Uso de puntos de interrupción (breakpoints)

Permiten detener el programa en una línea concreta y observar el estado de las variables en ese momento.



Inspección de variables

Consiste en examinar los valores actuales de las variables en tiempo de ejecución para entender cómo cambia el estado interno del programa.



Trazas o logs

System.out.println o frameworks de logging: añadir mensajes temporales ayuda a visualizar el flujo lógico del programa.

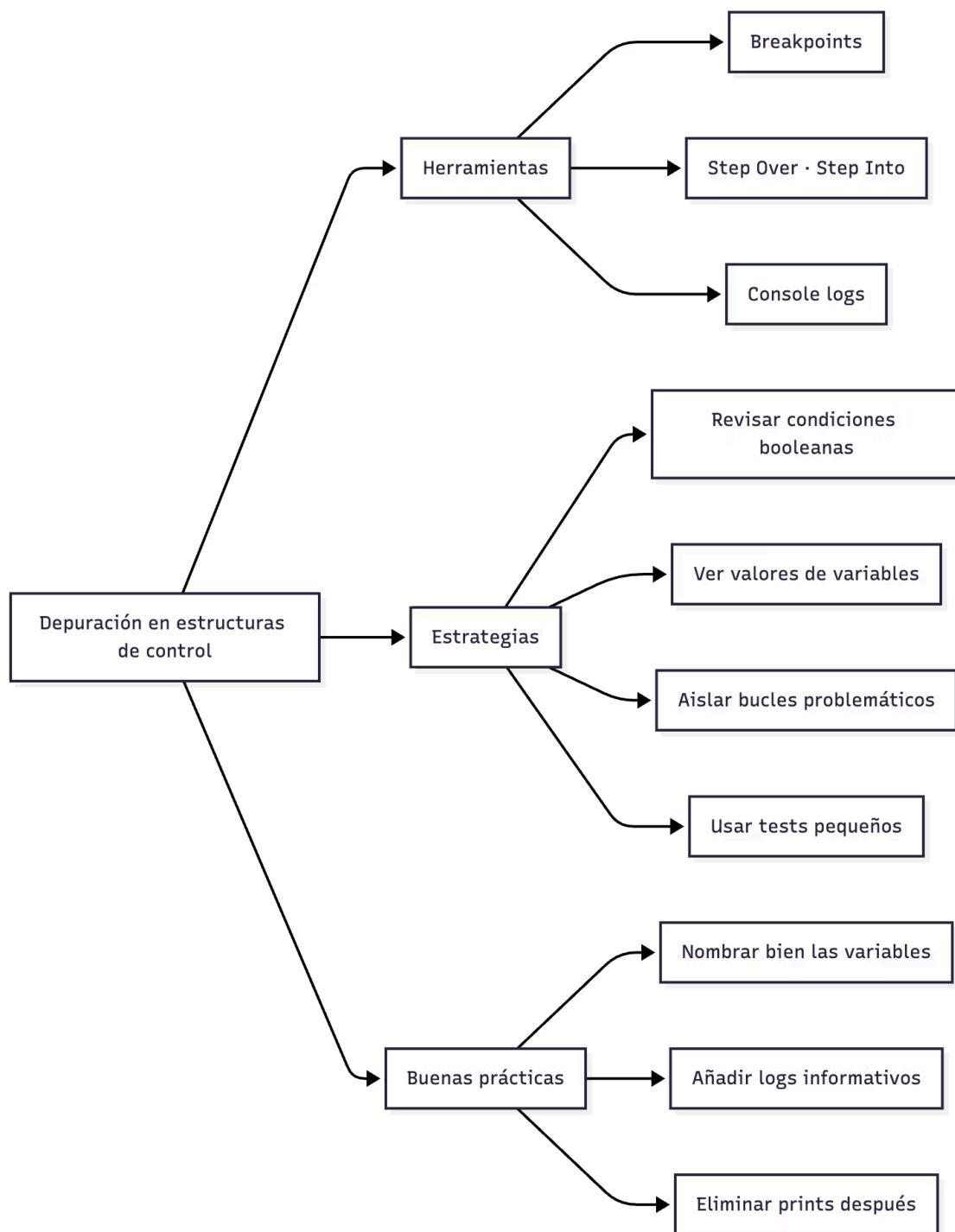


Revisión del flujo de ejecución

Analizar qué condiciones se cumplen y cuáles no, verificando la lógica de cada decisión.

Depurar es, en esencia, un proceso de **razonamiento estructurado**: observar, formular hipótesis, comprobar y corregir. Un buen programador no evita los errores: **aprende a rastrearlos con precisión quirúrgica**.

Esquema Visual



ⓘ Interpretación del esquema:

- **Herramientas:** representan los recursos técnicos que facilitan la depuración, especialmente los IDE modernos (IntelliJ, Eclipse, VS Code).
- **Estrategias:** explican los pasos mentales del proceso: analizar, aislar y comprobar.
- **Buenas prácticas:** refuerzan los hábitos profesionales para mantener el código limpio y legible incluso durante la depuración.

Caso de Estudio: Twitter y la depuración automática en sistemas de timeline

Contexto

En Twitter, millones de publicaciones por segundo deben ordenarse y mostrarse en el *timeline* de cada usuario según relevancia, idioma, ubicación y comportamiento previo. Detrás de ese proceso hay estructuras de control muy complejas: bucles anidados para recorrer usuarios y tuits, condicionales para aplicar filtros, y *switches* para priorizar contenidos según categoría.

En un entorno tan dinámico, los errores más comunes no son fallos de compilación, sino **fallos lógicos en tiempo real**, como tweets que no se muestran o aparecen duplicados.

Estrategia

El equipo de ingeniería de Twitter desarrolló un sistema de **depuración automática** basado en trazas de ejecución y *breakpoints virtuales*. En lugar de detener la producción (algo imposible en sistemas en vivo), el sistema detecta automáticamente cuándo una condición no se cumple y **registra los valores de las variables implicadas** en ese instante.

Por ejemplo:

- Si un tweet no aparece en el timeline, el sistema registra:
 - ID del usuario.
 - Estado de visibilidad.
 - Resultado de cada condición lógica evaluada.
 - Timestamp del evento.

Estos registros se almacenan en un sistema de análisis posterior (basado en *ELK Stack* – Elasticsearch, Logstash y Kibana), permitiendo reproducir el error fuera del entorno de producción.



Resultado

5x

Velocidad de detección

Detección de errores lógicos hasta 5 veces más rápida.

40%

Reducción de errores

De errores de presentación en timeline.

100%

Disponibilidad

Mantenimiento del servicio sin interrumpir el flujo de usuarios.

El caso de Twitter demuestra que una **buenas estrategia de depuración no es reactiva, sino preventiva**. Automatizar la observación de variables críticas permite mantener sistemas complejos con



Herramientas y Consejos

Consejos prácticos

Usa el debugger del IDE antes de añadir `System.out.println`.

1. Los prints son útiles, pero limitados. Un depurador te permite detener el código, inspeccionar el valor exacto de las variables y ejecutar línea por línea sin modificar el programa.

Aísla el problema.

1. Si el error ocurre en un bucle anidado o una condición compleja, extrae ese bloque en un método independiente y pruébalo por separado. Es más fácil detectar un error cuando el contexto es pequeño.

Verifica condiciones booleanas.

1. Muchos errores provienen de operadores mal empleados (`&&` vs `||`, o el uso incorrecto del `!`). Añade trazas como:
2. `System.out.println("Entrando en bucle, condición=" + (x < y));`
3. para confirmar qué partes del código se ejecutan realmente.

Añade logs informativos y temporales.

1. Evita mensajes genéricos como "Error aquí". Prefiere logs claros:
2. `log.info("Iteración {}: valor actual de total = {}", i, total);`
3. Si usas frameworks como Log4J o SLF4J, podrás ajustar los niveles de detalle (INFO, DEBUG, ERROR).

Usa tests pequeños y específicos.

1. Cada bloque lógico debería tener su propio test unitario. Si algo falla, sabrás exactamente qué parte del código lo causa.

Elimina prints después de depurar.

1. Los mensajes de depuración deben retirarse o sustituirse por logs controlados antes de entregar el código. Dejar `System.out.println` en producción puede afectar al rendimiento.

Herramientas recomendadas



IntelliJ IDEA / Eclipse Debugger:

permiten añadir breakpoints, inspeccionar variables y ejecutar paso a paso (Step Into, Step Over, Step Out).



JUnit 5: framework estándar para crear tests unitarios que verifiquen condiciones y resultados esperados.



Log4J / SLF4J: librerías de logging profesionales para registrar información estructurada durante la ejecución.

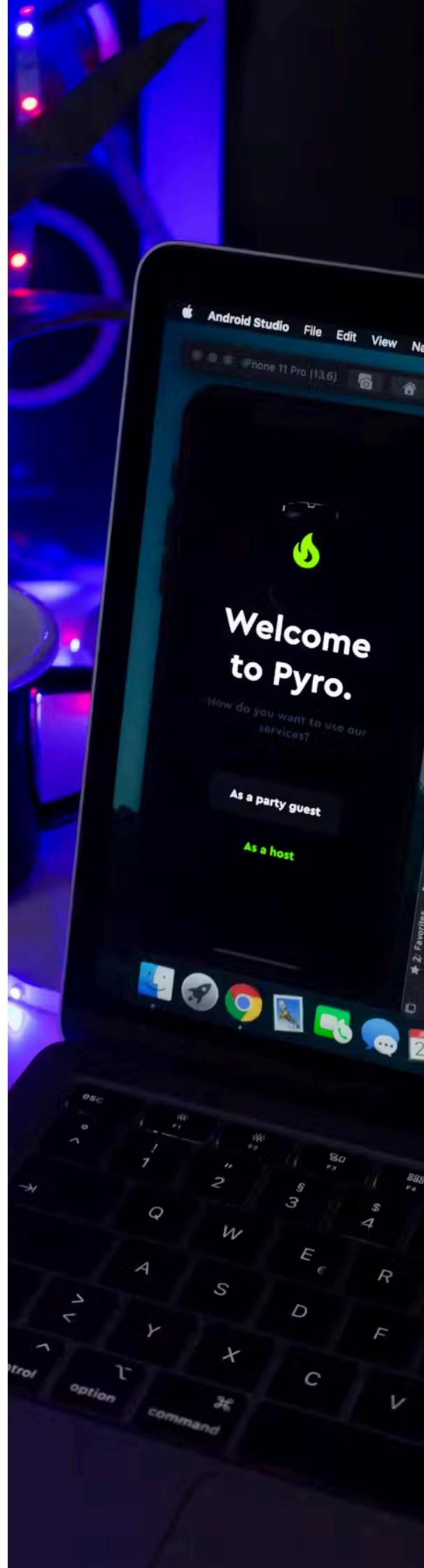


JVisualVM: monitoriza en tiempo real la ejecución y consumo de recursos, útil para detectar bucles infinitos o cuellos de botella.



Mockito: útil para depurar interacciones entre objetos simulados (en pruebas unitarias).

Estas herramientas son esenciales en entornos empresariales donde los errores deben detectarse antes de llegar a producción.



Mitos y Realidades

⊗ **✗ Mito:** “Depurar es perder tiempo.”

→ **FALSO.** La depuración no es una pérdida de tiempo, es una inversión. Encontrar y corregir un error en la etapa de desarrollo cuesta 10 veces menos que hacerlo en producción. Un buen proceso de debugging **ahorra horas de correcciones y evita incidentes costosos.**

⊗ **✗ Mito:** “Los mejores programadores no necesitan depurar.”

→ **FALSO.** Todos los desarrolladores, incluso los más experimentados, depuran constantemente. La diferencia es que **usan las herramientas adecuadas** y tienen un método. Depurar no demuestra debilidad, demuestra profesionalismo.



□ **Resumen Final**

- La **depuración** localiza y corrige errores lógicos.
- Usa **breakpoints** para detener la ejecución y analizar el estado del programa.
- Añade **logs informativos** o trazas temporales para seguir el flujo.
- **Aísla bucles o condicionales problemáticos** en métodos separados.
- Usa **JUnit** y **asserts** para validar resultados y prevenir errores futuros.