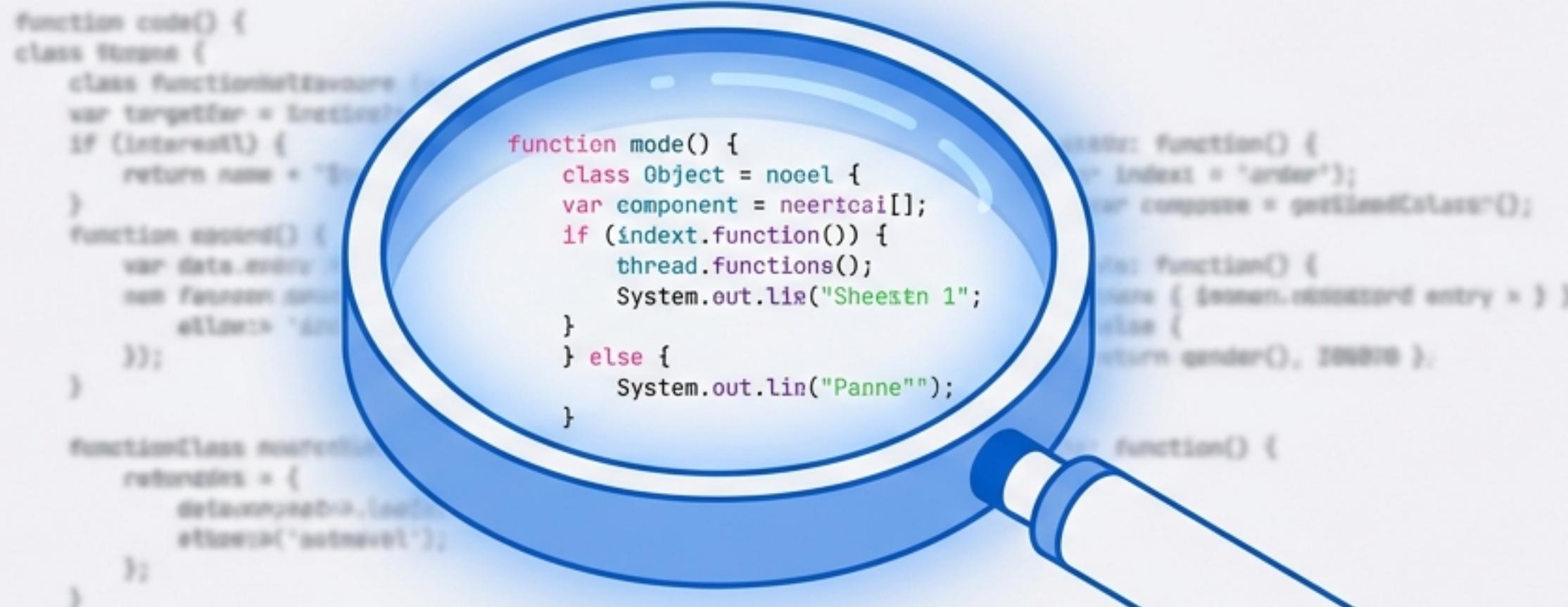


Analizadores de Código: Revisión Estática y Configuración

Cómo detectar errores, mejorar el estilo y asegurar la calidad antes de ejecutar una sola línea.



Mirando el futuro del código sin ejecutarlo

Un analizador de código realiza una revisión estática. Lee las estructuras (clases, métodos, variables) para detectar problemas sin necesidad de compilar ni ejecutar el programa.

Te ayuda a 'ver' problemas antes de compilar y a mantener una calidad consistente.



Errores probables

Bugs típicos que rompen la ejecución.



Malas prácticas

Código que funciona pero es frágil.



Estilo y Mantenibilidad

Reglas de formato y limpieza.

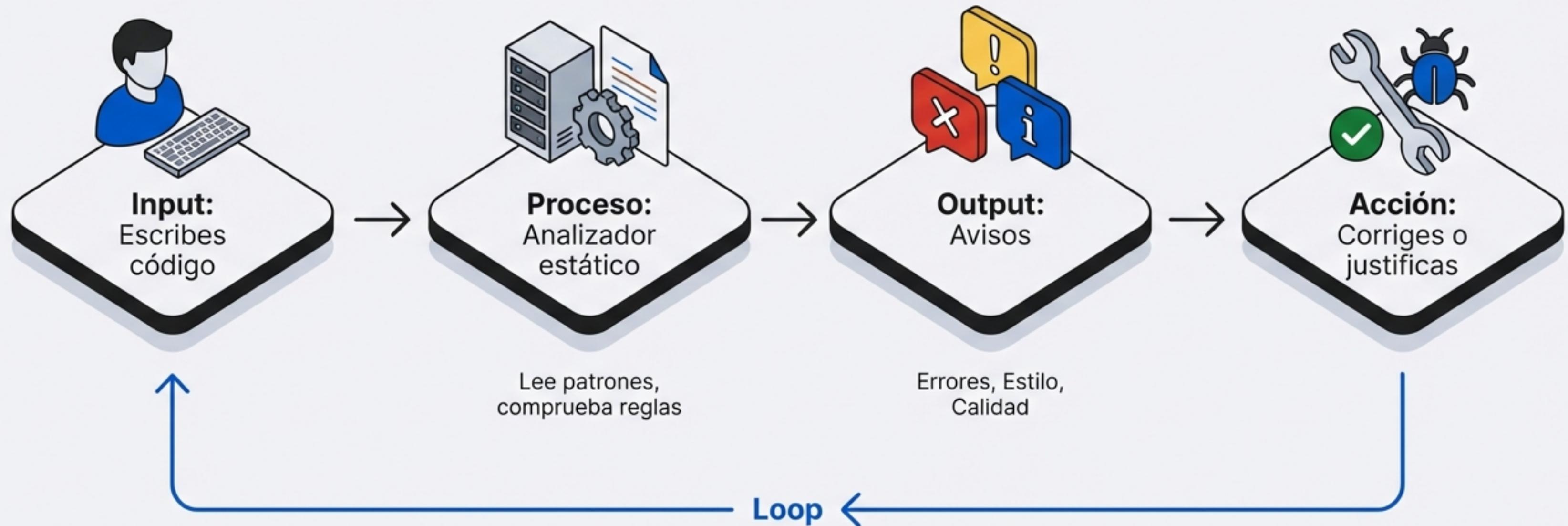


Rendimiento/Seguridad

Eficiencia y vulnerabilidades.

El Ciclo de Feedback Inmediato

Una conversación constante entre el desarrollador y la herramienta.



Por qué son indispensables en proyectos reales



Prevención de bugs

Detectan fallos típicos antes de que lleguen a producción (Escudo).



Estilo uniforme

Todo el equipo sigue las mismas reglas, independientemente de quién escriba.



Mantenibilidad

Código más fácil de leer, modificar y escalar a largo plazo.



Trabajo en equipo

Reduce discusiones subjetivas en las Code Reviews. “Lo decide la regla”.

Detectando Bugs: El caso del NullPointerException

Los analizadores detectan variables que pueden ser null o comparaciones peligrosas.

PROPENSO A ERROR

```
String rol = null;  
  
// ❌ Puede lanzar NullPointerException  
if (rol.equals("ADMIN")) { ... }
```

FORMA SEGURA

```
// ✅ Forma segura (evita NPE)  
if ("ADMIN".equals(rol)) { ... }
```

Seguridad defensiva aplicada automáticamente.

Limpieza de Código (Dead Code)

Si el código no aporta valor, estorba. El análisis estático detecta asignaciones redundantes, variables sin uso y código inalcanzable.

```
int x = 10;  
x = 10;
```

⚠ Warning: Asignación redundante

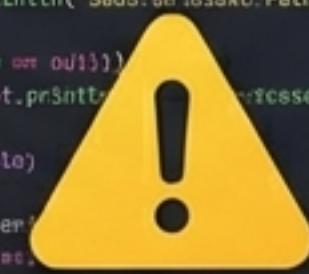
```
System.out.println(x);
```

Estilo, Complejidad y Mantenibilidad

Esto no rompe el programa, pero afecta su lectura futura.

- **Estilo:** Nombres mal formados, indentación, imports innecesarios.
- **Complejidad:** Métodos demasiado largos, exceso de if/else.

```
public setmias. AcohethBeotEheetVroove.wciOReçfreoid erite, ueers, Nunbsi-Vdatodote:Esdiegthoe.xioeeeagae, LaotSancValue-Enney> lot, StartGetGetance:loweecs) {  
    String Moredons < now: soerAccounter();  
    Set. cardelsandaaro < nolt;  
    IF (urnoredSct:oddrbnntencareolsegunotets >= 0) {  
        cenot:caonby = evaator.ceoRecetodate( <now 10PL2_P0221);  
        IF (saacouitcsbteXcaetisse == null) {  
            System.out.println("Sads: se iasaiko. Pathvilonootbz900+<bleavtaet/dsbbtse");  
        } else {  
            if (to:second == od15) {  
                System.out.println("Sads: se iasaiko. Pathvilonootbz900+<bleavtaet/dsbbtse());  
            } etae {  
                return Role;  
            }  
            else IF (laeven1) {  
                IF (lx > sc1)  
                    return seAsoem();  
                }  
            }  
            etae IF (caerent&scotce == eederdatewertedlofserou(1) {  
                System.out.println(Ecenent1);  
            }  
        } etae {  
            SF (Ssmtabandtacecate == S-noste@westeosEmp()) {  
                System.out.println(Flecoot);  
            }  
        }  
    }  
}
```



SEÑAL TÍPICA: Si un método tiene 80-100 líneas, probablemente necesita refactorización.

El Ecosistema de Herramientas Java



Inspections del IDE

Warnings y sugerencias integradas (IntelliJ).



Checkstyle

Especialista en reglas de estilo y convenciones visuales.



PMD

Enfocado en malas prácticas y calidad de código.



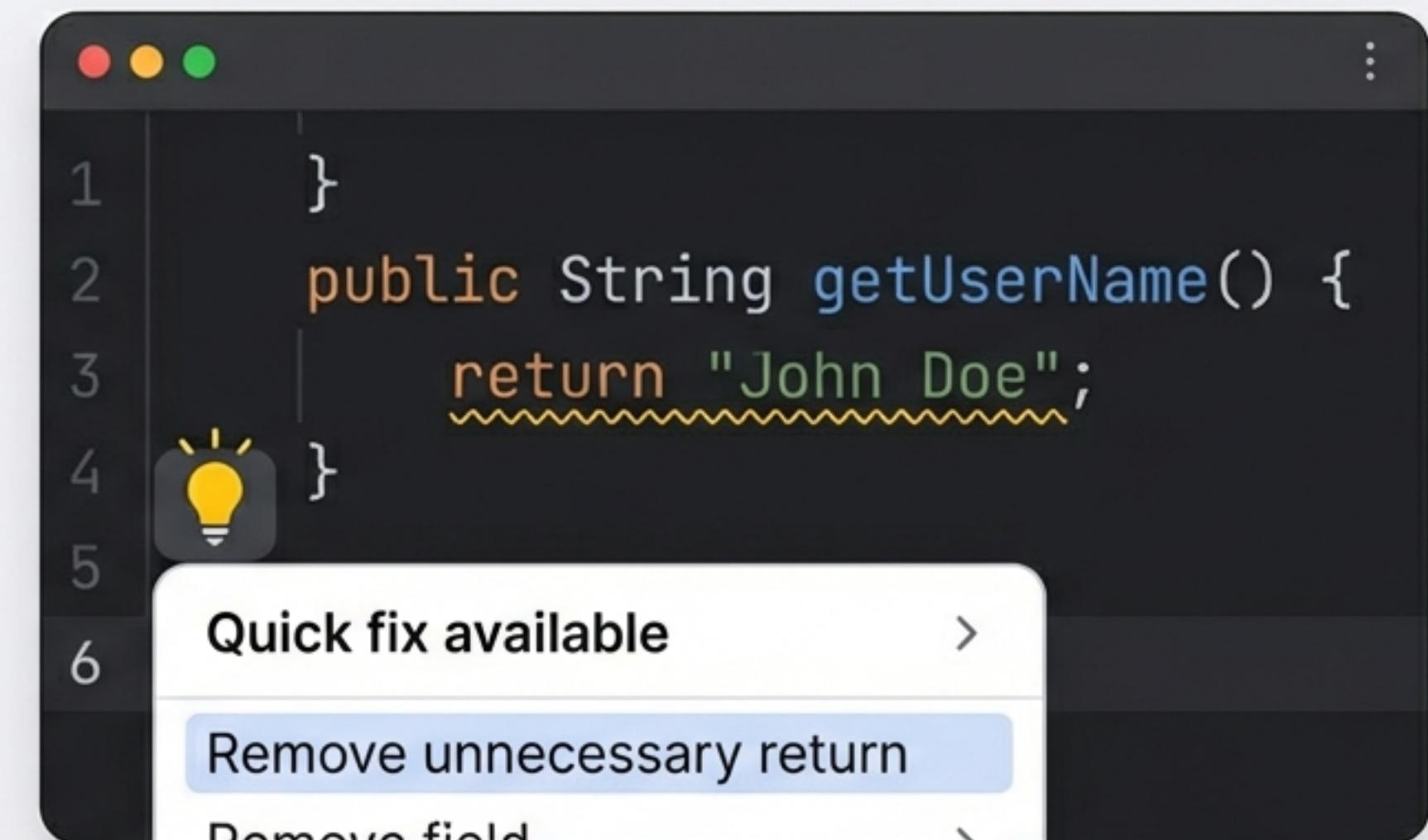
SpotBugs

Detecta bugs típicos basándose en patrones.

Las herramientas externas suelen integrarse en el proyecto y en los procesos de Integración Continua (CI).

Integración en el IDE: Tu Copiloto Diario

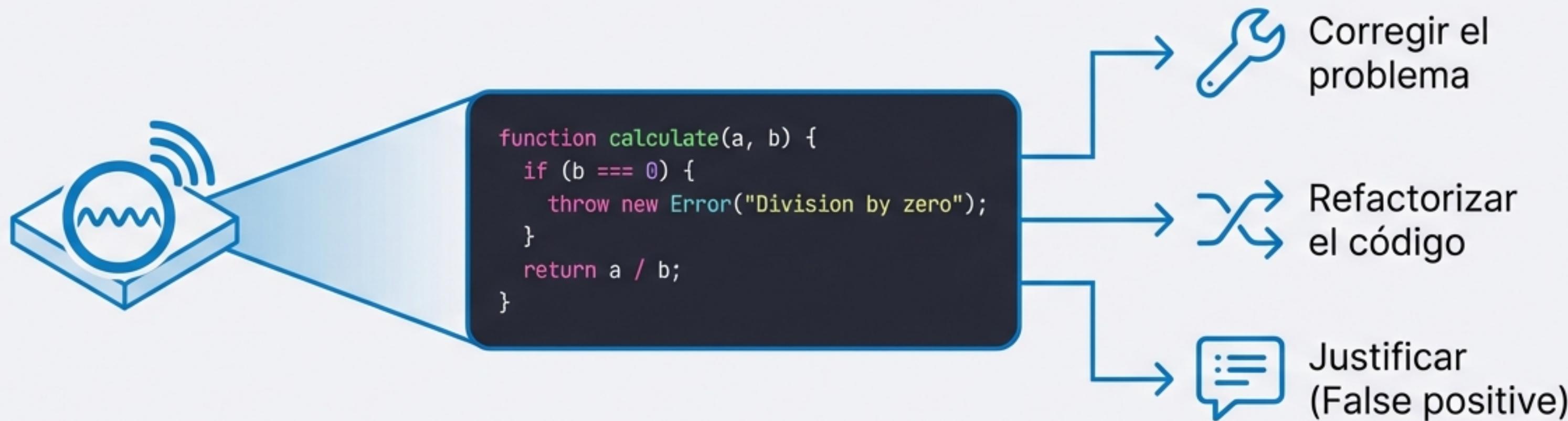
- Avisos visuales (subrayado, bombilla).
- Inspecciones por archivo o proyecto completo.
- Sugerencias de refactorización automática.



¡CUIDADO! No aceptes quick fixes a ciegas.
Entiende qué cambia y por qué antes de aplicarlo.

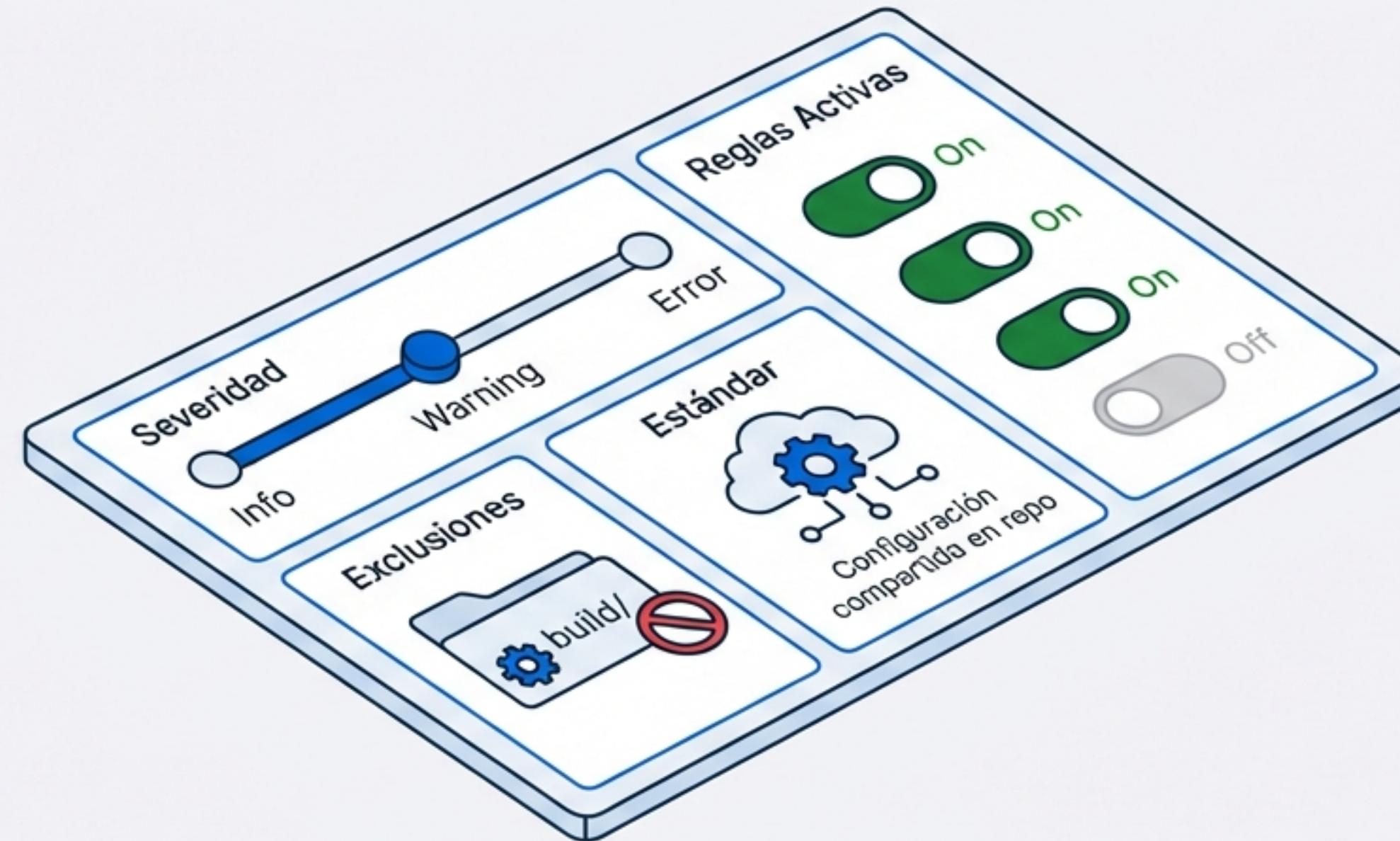
Spotlight: SonarLint

Señales para decidir, no magia negra.



SonarLint no arregla el código por ti; te da señales para que tú tomes la decisión.

Configuración Inteligente



Evita el ruido configurando solo lo necesario.

Ejemplo de Regla: "Evitar Números Mágicos"

Transformando números oscuros en intenciones claras.

ANTES (Confuso)

```
// Nadie sabe qué significa 8
if (password.length() < 8) {
    ...
}
```

DESPUÉS (Claro)

```
final int MIN_PASSWORD_LENGTH = 8;
if (password.length() <
    MIN_PASSWORD_LENGTH) { ... }
```

Gestión de Avisos: Salud Mental y Calidad



- **Reduce Ruido:** Excluye carpetas irrelevantes y desactiva reglas que no apliquen.
- **Consenso:** Acordad reglas en equipo y guardadlas en el repositorio.



Antipatrón: Ignorar todos los avisos porque “molestan”. Es mejor tener pocos avisos bien elegidos que cien inútiles.

Tu Red de Seguridad Invisible



- El análisis estático ofrece “visión de rayos X” para prevenir bugs y deuda técnica.
- Las herramientas (IntelliJ, SonarLint) son asistentes, no sustitutos del criterio humano.
- Una buena configuración elimina el ruido y centra al equipo en lo importante.