SESIÓN 19
Métricas de Calidad: Complejidad Ciclomática y CRAP Score

COMPLEJIDAD CICLOMÁTICA (CC)

La complejidad ciclomática mide el número de caminos independientes que puede seguir el código.

Más caminos
- más pruebas necesarias
    - más riesgo.

Regla

CC = E - N + 2
(E = aristas, N = nodos del grafo de flujo)

Pero en la práctica profesional:

CC = 1 + número de decisiones

Decisiones:
- if
- else if
- for
- while
- case
- catch
- operadores && y ||

EJEMPLOS PROGRESIVOS

1.Código limpio

```
public int suma(int a, int b) {
   return a + b;
}
```

Decisiones: 0
CC = 1
Fácil de probar.

2.Un condicional

```
public boolean esMayor(int edad) {
   if (edad >= 18) {
      return true;
   }
   return false;
}
```

Decisiones: 1
CC = 2
Necesito mínimo 2 tests.

3. Dos decisiones

```
public String evaluar(int nota, boolean recuperacion) {

   if (nota >= 5) {
      return "Aprobado";
   }

   if (recuperacion) {
      return "Aprobado por recuperación";
   }

   return "Suspenso";
}
```

Decisiones: 2
CC = 3
Ya necesitamos 3 caminos mínimos.

MÉTRICA CRAP

La complejidad sola no es suficiente.
Un código complejo puede ser seguro si está bien cubierto por tests.
CRAP Score
(Change Risk Anti-Patterns)

Qué mide realmente
Complejidad × Falta de cobertura

CC	Nivel	Qué significa en empresa
1–10	Sana	Código profesional
11–20	Moderada	Revisar
21–30	Alta	Difícil de probar
>30	Crítica	Refactor urgente

Fórmula

$CRAP = CC^2 \times (1 - Coverage)^3 + CC$

Donde:

- **CC** = *Complejidad Ciclomática*
- **Coverage** = Cobertura de tests (en decimal, no en porcentaje)

¿Qué significa cada parte?

CC^2
Se eleva al cuadrado la complejidad ciclomática.
Esto hace que la penalización crezca mucho si el código es complejo.

$(1 - Coverage)^3$

- Si la cobertura es alta: (1 - Coverage) es pequeño, el resultado baja mucho.
- Si la cobertura es baja: (1 - Coverage) es grande, el resultado se dispara.

Se eleva al cubo para penalizar fuertemente la falta de tests.

+ CC
Se suma la complejidad original al final.

Ejemplo
Supongamos:

- CC = 10
- Coverage = 80%, en decimal es 0.8

Paso 1
$CC^2=10^2=100$

Paso 2
1-0.8=0.21 - 0.8 = 0.2

Paso 3
$0.2^3=0.0080$

Paso 4
$100\times0.008=0.8$

Paso 5
CRAP=0.8+10=10.8
Resultado: **10.8 (valor bajo, aceptable)**

Para calcular **CRAP**, necesitas **dos datos**:

1. **CC (Complejidad Ciclomática)**
2. **Coverage (Cobertura de tests)**

¿De dónde sale la CC?

La **Complejidad Ciclomática** mide cuántos caminos distintos tiene un método.

Regla básica para calcularla manualmente:

Empiezas en **1** y sumas:

- +1 por cada if
- +1 por cada else if
- +1 por cada for
- +1 por cada while
- +1 por cada case
- +1 por cada catch
- +1 por cada operador && o ||

```
public int ejemplo(int x) {
   if (x > 0) {         // +1
      for (int i = 0; i < x; i++) {  // +1
         if (i % 2 == 0) {  // +1
            System.out.println(i);
         }
      }
   }
   return x;
}
```

Cálculo:
Base = 1
+1 (if)
+1 (for)
+1 (if interno)
CC = 4

Cómo ver la Complejidad Ciclomática (CC)
IntelliJ por defecto NO la muestra directamente.

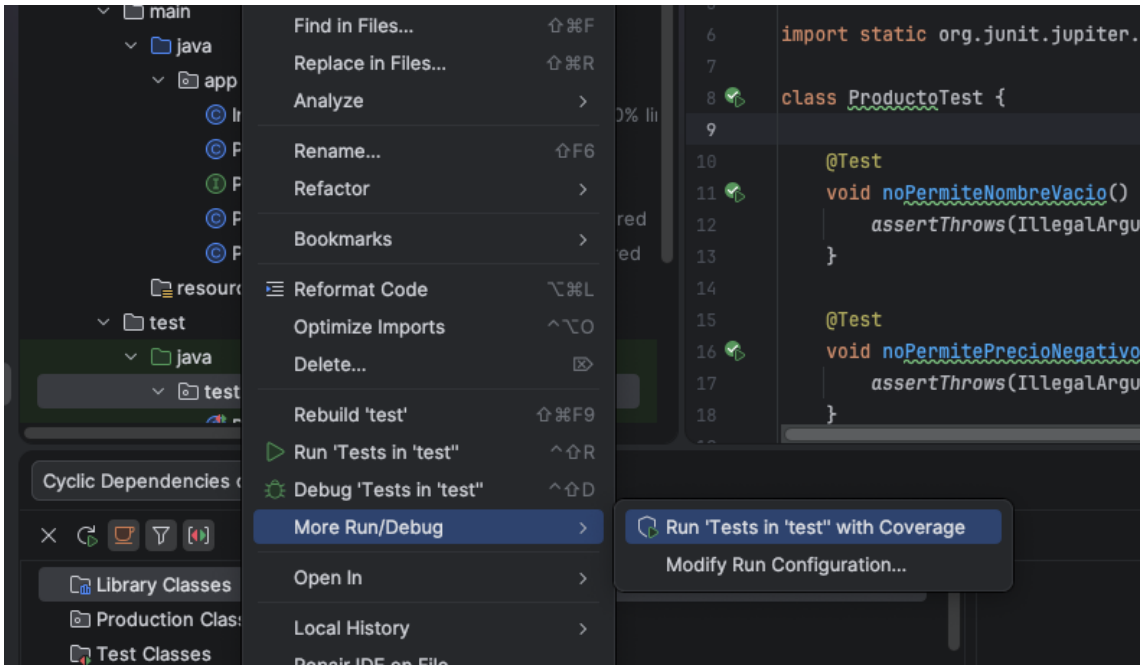
Instalar SonarQube

Paso 1
Instalar.

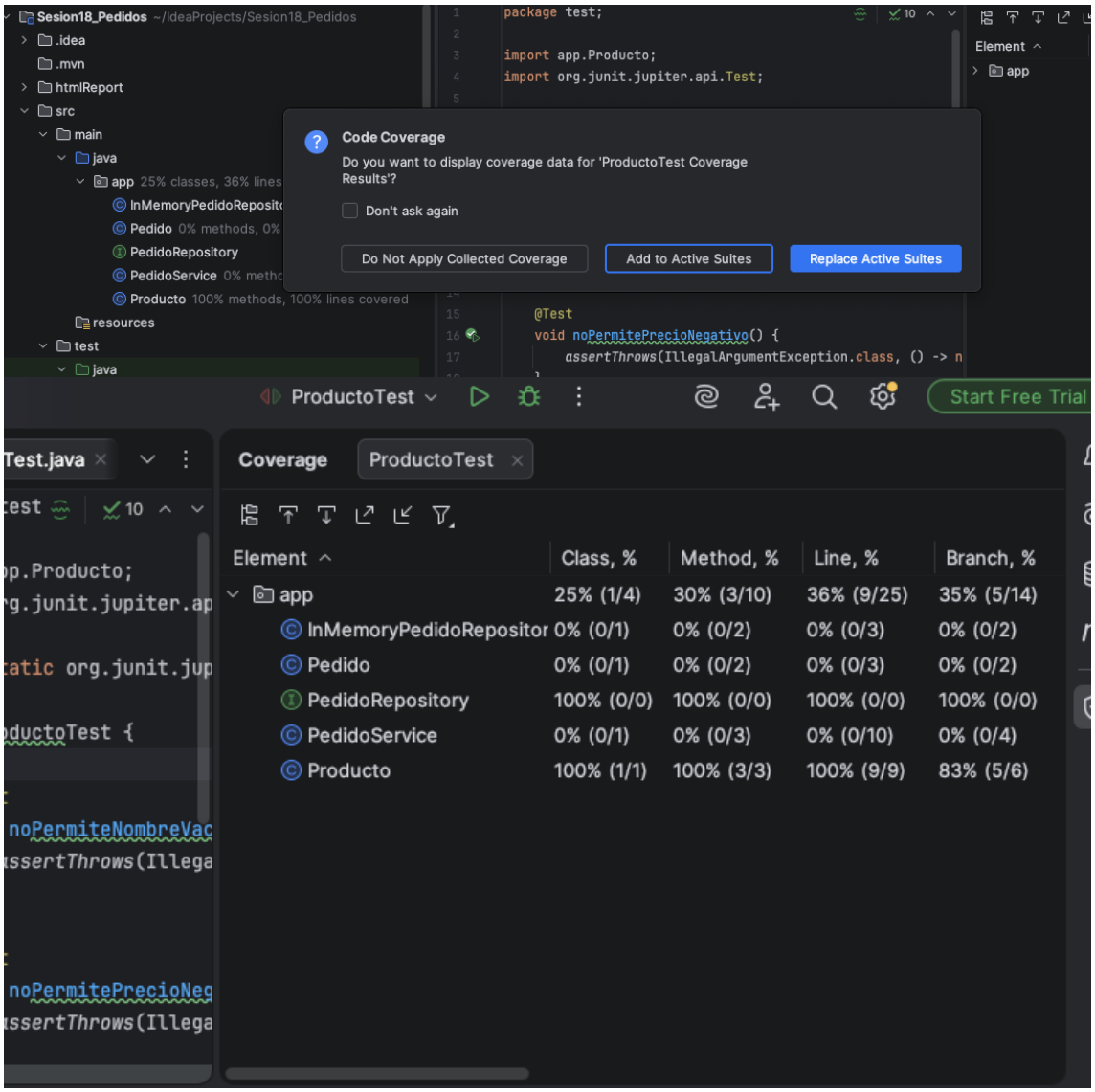
Paso 2
Busca:
SonarQube
Instálalo y reinicia.

Paso 3

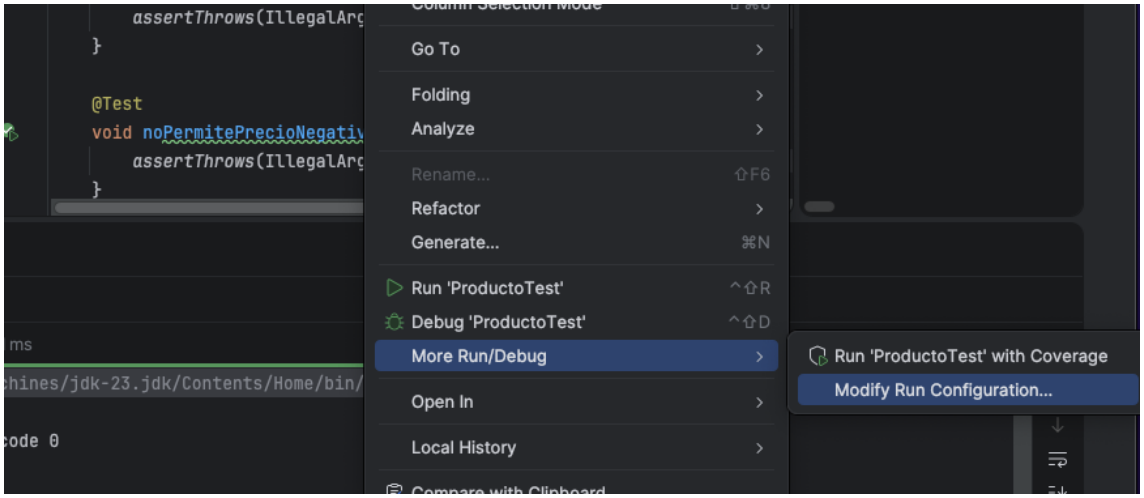
Botón derecho Analyze → Analyze with SonarQube/Coverage



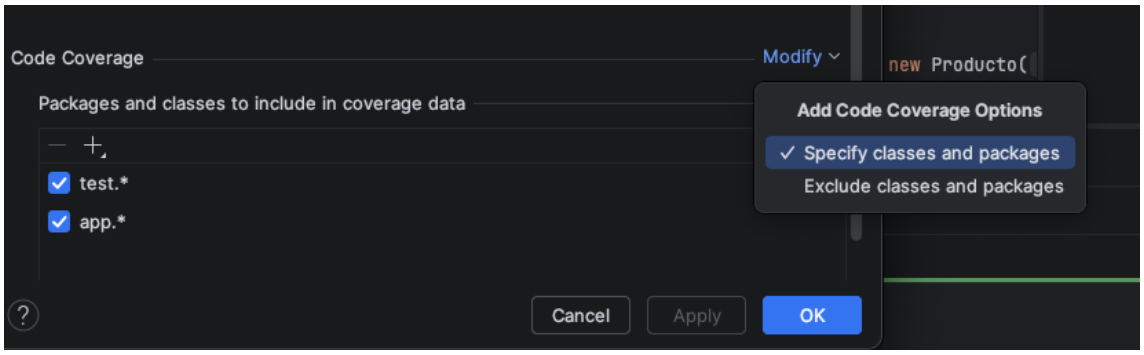
Damos a Replace



Si no os sale nada,



En Modify añadir vuestro paquete, el mío es app.



Os dejo mi pom.xml

Sin JaCoCo puede que no funcione.
JaCoCo (Java Code Coverage) es una herramienta que sirve para medir la cobertura de código en proyectos Java.
Te dice qué porcentaje de tu código está siendo ejecutado por tus tests.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Sesion18_Pedidos</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>23</maven.compiler.source>
        <maven.compiler.target>23</maven.compiler.target>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
```

```
      <version>5.10.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.2.5</version>
      </plugin>
        <plugin>
          <groupId>org.jacoco</groupId>
          <artifactId>jacoco-maven-plugin</artifactId>
          <version>0.8.8</version>
          <executions>
            <execution>
              <goals>
                <goal>prepare-agent</goal>
              </goals>
            </execution>
            <execution>
              <id>report</id>
              <phase>test</phase>
              <goals>
                <goal>report</goal>
              </goals>
            </execution>
          </executions>
        </plugin>


    </plugins>
  </build>

</project>
```

Ejemplo

```
public boolean esPar(int numero) {
   if (numero % 2 == 0) {
      return true;
   }
   return false;
}
```

Paso 1: Contar decisiones

- Hay **1** if = **1 decisión**

Paso 2: Calcular CC

Regla: **CC = 1 + nº decisiones**

CC = 1 + 1 = **2**

Paso 3: Tests mínimos

Mínimo = CC = **2 tests**

- Test 1: número par (ej. 2) = true
- Test 2: número impar (ej. 3) = false

Resultado:

- decisiones=1
- CC=2
- tests mínimos=2