



# UNIVERSIDAD DE LA FRONTERA

FACULTAD DE INGENIERÍA Y CIENCIAS

Departamento de Ciencias de la Computación e Informática

## RT-POO-03 Maven y JUnit 5

**Profesor**

*Dr. Samuel Sepúlveda Cuevas*

**Ayudante**

*Jesús Tapia Martin*

**Código**

*ICC490*

Temuco, Chile.

Abril, 2025

# Índice

1	Problemática Actual	3
1.1	Sistema integrado de IntelliJ IDEA . . . . .	3
1.1.1	Gestión limitada de dependencias externas . . . . .	3
1.1.2	Falta de automatización en el ciclo de desarrollo . . . . .	4
1.1.3	Problemas de escalabilidad y colaboración en equipo . . . . .	4
1.2	Consecuencias en el flujo de trabajo . . . . .	5
2	Maven	6
2.1	Coordinador del ciclo de desarrollo . . . . .	6
2.2	Gestión automática de dependencias externas . . . . .	7
2.2.1	Project Object Model (POM) . . . . .	7
2.2.2	Ejemplo . . . . .	7
2.3	Creación de un proyecto Maven en IntelliJ IDEA . . . . .	8
3	JUnit	9
3.1	Pruebas Unitarias . . . . .	9
3.2	Arquitectura del framework . . . . .	9
3.3	Anotaciones . . . . .	10
3.3.1	Definición . . . . .	10
3.4	Integración de JUnit en un proyecto con Maven . . . . .	10
3.4.1	Buscar la dependencia . . . . .	10
3.4.2	Agregar dependencia al archivo <b>pom.xml</b> . . . . .	12
3.4.3	Generar archivo de pruebas . . . . .	13

# 1 Problemática Actual

---

Llegados hasta este momento del curso, se optado por utilizar el sistema predeterminado que proporciona *IntelliJ IDEA* para la creación y desarrollo de proyectos Java. Este entorno integra un sistema que permite crear, configurar y ejecutar proyectos sin requerir herramientas externas ni configuraciones manuales adicionales.

Con este sistema, hasta este momento, se han simplificado tareas que tradicionalmente serían complejas manualmente, como la generación y configuración inicial de un proyecto, ya que por medio de una interfaz gráfica intuitiva y múltiples opciones se logra reducir significativamente el tiempo de configuración y la complejidad técnica para los desarrolladores.

Esta ventaja que entrega el sistema integrado en el *IDE* es particularmente útil en etapas iniciales del aprendizaje o en el desarrollo de aplicaciones simples. Sin embargo, a medida que los proyectos crecen en complejidad o se requiere trabajo colaborativo, este enfoque presenta limitaciones, especialmente en la gestión de dependencias externas y la estandarización del entorno de desarrollo.

## 1.1. Sistema integrado de IntelliJ IDEA

El sistema integrado de IntelliJ IDEA se refiere a la capacidad del entorno para gestionar internamente el ciclo de vida básico de un proyecto Java. Esto incluye:

- Creación de la estructura del proyecto.
- Configuración de compiladores y rutas.
- Ejecución de código.
- Inclusión manual de bibliotecas externas.

A pesar de su utilidad para usuarios principiantes su diseño imposibilita resolver las necesidades de proyectos profesionales, como la consistencia en entornos de trabajo y automatización de tareas repetitivas.

### 1.1.1. Gestión limitada de dependencias externas

En el contexto actual del desarrollo de software, la mayoría de los proyectos requieren el uso de bibliotecas externas, también conocidas como dependencias, para incorporar

funcionalidades específicas ya preconfiguradas y listas para su uso. Un ejemplo común es el caso de *JUnit* para validar el correcto funcionamiento del código mediante la ejecución de pruebas automatizadas.

Con el sistema integrado de *IntelliJ IDEA*, esta gestión de dependencias debe realizarse de forma manual:

- Buscar y descargar archivos *.jar* desde fuentes externas.
- Incorporar dichos archivos al proyecto mediante rutas específicas.
- Repetir el proceso cada vez que se requiera una nueva versión o se añadan nuevas bibliotecas.

Este enfoque, además de consumir tiempo, dificulta trazar las dependencias utilizadas junto a sus versiones, dejando vulnerable el proyecto a errores por incompatibilidades de versiones, duplicación de bibliotecas o pérdidas de configuración al compartir el proyecto.

#### **1.1.2. Falta de automatización en el ciclo de desarrollo**

La automatización en proyectos de software permite reducir errores humanos, aumentar la eficiencia y asegurar una mayor consistencia en las entregas. Sin embargo, el sistema integrado de *IntelliJ IDEA* ofrece capacidades limitadas para automatizar procesos esenciales como:

- Ejecución automatizada de pruebas unitarias.
- Generación de artefactos empaquetados listos para distribución (por ejemplo, archivos *.jar* o *.war*).

Al deber gestionar cada uno de estos pasos manualmente por el desarrollador, se introduce una carga operativa innecesaria, aumentando el riesgo de omitir tareas clave.

#### **1.1.3. Problemas de escalabilidad y colaboración en equipo**

Uno de los mayores desafíos al escalar un proyecto o al trabajar en equipos distribuidos es garantizar la coherencia del entorno de desarrollo. El sistema integrado de *IntelliJ IDEA* no proporciona un mecanismo estandarizado para definir de manera explícita las configuraciones del proyecto, lo que conlleva los siguientes inconvenientes:

- Cada integrante del equipo puede configurar su entorno de forma distinta.
- Las bibliotecas utilizadas por un desarrollador pueden no estar presentes en el entorno de otro, lo que genera errores al compilar o ejecutar el proyecto.
- No existe un archivo centralizado que defina y gestione las dependencias de forma declarativa y reproducible.

Esta falta de estandarización impide una integración continua eficaz, dificulta la incorporación de nuevos miembros al equipo y aumenta los costos de mantenimiento del proyecto a largo plazo.

## 1.2. Consecuencias en el flujo de trabajo

Si bien el sistema integrado de *IntelliJ IDEA* funciona bien para entornos de desarrollo individuales o académicos, no ofrece las capacidades necesarias para sostener un flujo de trabajo profesional en proyectos colaborativos, escalables y mantenibles. Las limitaciones en cuanto a gestión de dependencias, automatización y estandarización provocan:

- Mayor probabilidad de errores humanos.
- Reducción en la calidad del software entregado.
- Dificultades en la integración continua y el despliegue automatizado.

## 2 Maven

Frente a las limitaciones del sistema integrado de *IntelliJ IDEA* aparece **Apache Maven** como una solución robusta para estas.

*Maven* automatiza tareas rutinarias del ciclo de vida de un proyecto, como la gestión de dependencias, la compilación del código fuente, la ejecución de pruebas unitarias y la generación de artefactos finales, tareas que demandan tiempo y son propensas a errores al realizarse manualmente, en cambio, con *Maven* estas tareas se ejecutan de manera automática y controlada mediante un archivo de configuración centralizado.

### 2.1. Coordinador del ciclo de desarrollo

Desde manera conceptual se puede concebir a Maven como un coordinador del proyecto: orquesta las distintas etapas del desarrollo para que todas las piezas (dependencias, compilación, pruebas y empaquetado) trabajen de manera sincronizada y estandarizada. Esto reduce significativamente la intervención manual del desarrollador, minimiza errores humanos y permite establecer flujos de trabajo repetibles y confiables.

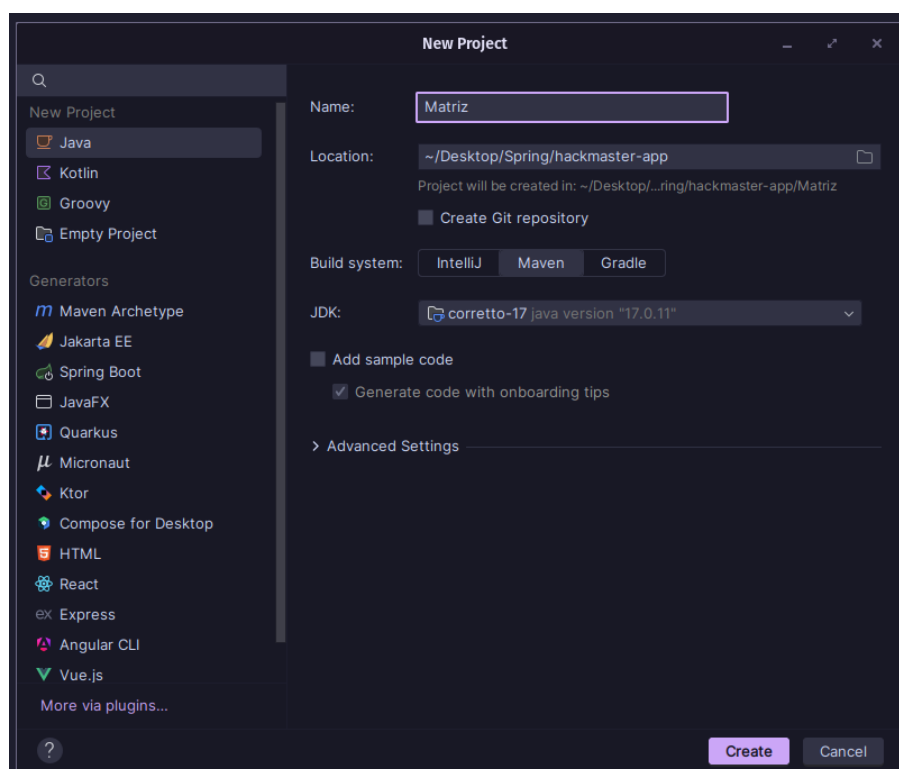


Figura 1: Interfaz de creación de un proyecto Maven en IntelliJ IDEA

## 2.2. Gestión automática de dependencias externas

A diferencia del enfoque manual del sistema integrado de *IntelliJ IDEA*, *Maven* permite declarar las dependencias necesarias del proyecto en un archivo denominado **pom.xml**, eliminando así la necesidad de descargar e incorporar manualmente archivos *.jar*.

### 2.2.1 Project Object Model (POM)

Archivo de configuración principal de Maven, escrito en formato XML, donde se describe

- Las dependencias externas necesarias para el proyecto.
- Los plugins requeridos para tareas adicionales.
- El identificador del grupo y del artefacto.
- La versión del proyecto.
- Los perfiles de ejecución y configuraciones adicionales.

Este archivo actúa como una documentación técnica fácilmente compartible, asegurando que todos los integrantes de un equipo trabajen con las mismas bibliotecas, versiones y configuraciones.

### 2.2.2 Ejemplo

Si el proyecto requiere la biblioteca *JUnit* para realizar pruebas unitarias, basta con incluir la siguiente entrada en el archivo **pom.xml**:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

Maven se encargará automáticamente de descargar esta biblioteca, junto con sus dependencias desde un repositorio central, asegurando compatibilidad y eliminando conflictos de versión.

## 2.3. Creación de un proyecto Maven en IntelliJ IDEA

Para utilizar Maven como sistema de construcción en IntelliJ IDEA, es necesario modificar la configuración por defecto durante la creación del proyecto. En la sección *Build System*, se debe seleccionar **Maven** en lugar del sistema integrado de IntelliJ.

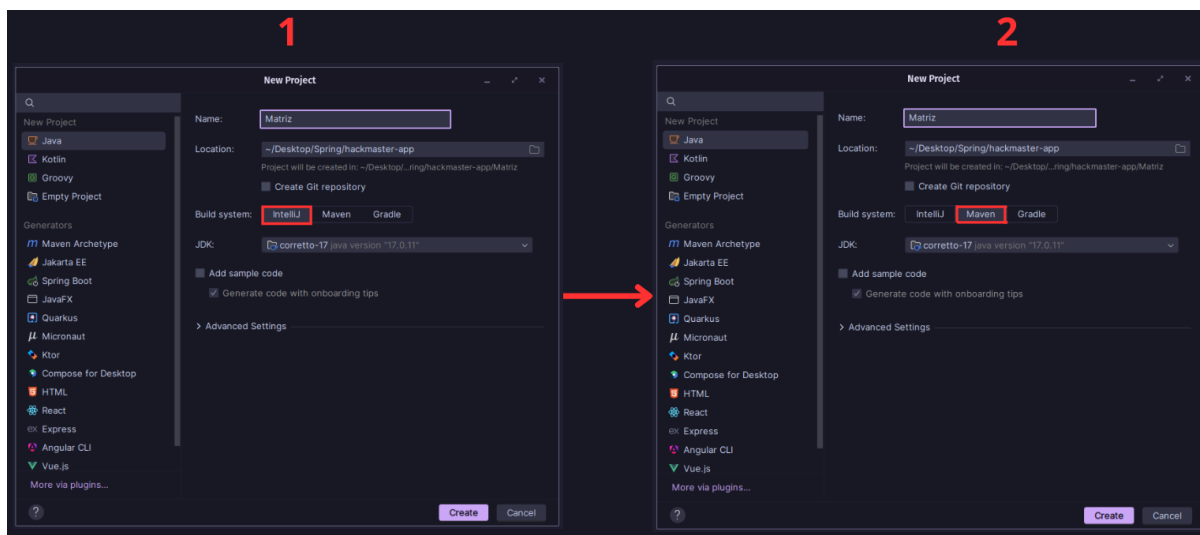


Figura 2: Selección de sistema de construcción Maven en IntelliJ IDEA

Una vez seleccionado *Maven*, *IntelliJ* habilita la configuración avanzada del proyecto, donde deben definirse los siguientes parámetros:

- **GroupId**: Identificador del grupo o entidad responsable del proyecto.
- **ArtifactId**: Nombre del artefacto que será usado como nombre del módulo y del archivo empaquetado.
- **Version**: Versión del project, por defecto se asigna *1.0-SNAPSHOT*.

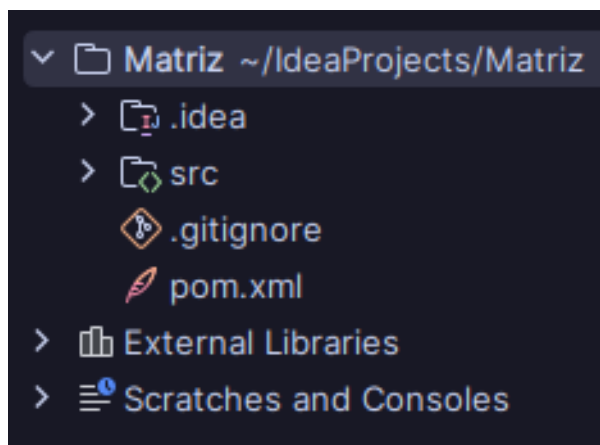


Figura 3: Estructura típica de un proyecto Maven



*JUnit* es un *framework* diseñado para la escritura y ejecución de pruebas unitarias en proyectos desarrollados en Java. Su objetivo principal es verificar que módulos individuales de código, como métodos o clases, se comporten de la manera que se espera y de forma aislada, permitiendo una detección temprana de errores y un software de confiable.

### 3.1. Pruebas Unitarias

Una prueba unitaria evalúa como se comporta una unidad de código, verificando que sus resultados coincidan con un valor esperados bajo ciertas condiciones.

#### Nota

*No debe confundirse la ejecución de pruebas unitarias con una garantía automática de calidad en el software, ya que la presencia de pruebas no implican por si mismas que estas aporten una cobertura real al sistema si estas no están bien diseñadas. En particular, las pruebas triviales —como verificar si una variable contiene un valor arbitrario sin relación directa con la lógica del sistema— no entregan información relevante sobre la estabilidad, corrección o robustez del código.*

*Para que una prueba unitaria tenga valor debe estar basada en un **caso de prueba significativo**, es decir, una situación diseñada para validar un aspecto importante del comportamiento del sistema, considerando tanto condiciones esperadas como posibles casos límite o excepciones. Solo a través de estas pruebas bien diseñadas es posible asegurar que el sistema responde correctamente ante distintos escenarios de uso real.*

### 3.2. Arquitectura del framework

Desde su versión 5, JUnit presenta una arquitectura modular compuesta por los siguientes componentes:

- **JUnit Platform:** Proporciona la infraestructura base para el lanzamiento de pruebas. Soporta la integración de múltiples motores de prueba y herramientas externas.
- **JUnit Jupiter:** Incluye la API moderna de JUnit 5 junto con las nuevas anotaciones, diseñada para pruebas unitarias modernas.
- **JUnit Vintage:** Permite ejecutar pruebas escritas con versiones anteriores, como JUnit 3 y 4 asegurando la retrocompatibilidad.

### 3.3. Anotaciones

*JUnit* proporciona un conjunto de anotaciones que facilitan la escritura de pruebas estructuradas y comprensibles.

#### 3.3.1 Definición

En *JUnit* las anotaciones es una sintaxis que permite definir el ciclo de vida y el comportamiento de las pruebas unitarias. Por medio del uso de anotaciones, se indica qué métodos deben ser ejecutados como pruebas, cuáles deben ejecutarse antes o después de cada prueba y cómo deben interpretarse los resultados esperados.

- **@Test**: Indica que un método representa una prueba unitaria.
- **@BeforeEach**: Ejecuta un método antes de cada prueba individual, útil para inicializar datos o instancias reutilizables en múltiples pruebas.
- **@AfterEach**: Se ejecuta después de cada prueba, ideal para liberar recursos o limpiar estados.
- **@BeforeAll** y **@AfterAll**: Ejecutan métodos una sola vez antes o después de todas las pruebas del conjunto.
- **@DisplayName**: Permite asignar nombres legibles y descriptivos a los métodos de prueba, mejorando la comprensión de los resultados.

Adicionalmente, *JUnit* ofrece métodos de aserción como **assertEquals**, **assertTrue** o **assertThrows**, los cuales permiten verificar que los resultados obtenidos durante la ejecución coincidan con los esperados.

### 3.4. Integración de JUnit en un proyecto con Maven

Para integrar JUnit en un proyecto Java gestionado con Maven, es necesario agregar la dependencia correspondiente en el archivo `pom.xml`. Maven descargará automáticamente los artefactos necesarios desde su repositorio central.

#### 3.4.1. Buscar la dependencia

Se debe acceder al sitio oficial de Maven Central: <https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api>

Home » org.junit.jupiter » junit-jupiter-api

**JUnit Jupiter API**  
JUnit Jupiter is the API for writing tests using JUnit 5.

License	EPL 2.0
Categories	Testing Frameworks & Tools
Tags	quality junit testing api
HomePage	https://junit.org/junit5/
Ranking	#21 in MvnRepository (See Top Artifacts) #3 in Testing Frameworks & Tools
Used By	18,044 artifacts

Central (85) Redhat GA (4) ICM (2)

Version	Vulnerabilities	Repository	Usages	Date
5.13.x		Central	15	Mar 24, 2025
5.13.0-M2		Central	4	Mar 21, 2025
5.12.2		Central	56	Apr 11, 2025
5.12.1		Central	554	Mar 14, 2025
5.12.0		Central	479	Feb 21, 2025
5.12.x		Central	16	Feb 12, 2025
5.12.0-RC2		Central	20	Feb 07, 2025
5.12.0-RC1		Central	9	Jan 31, 2025
5.12.0-M1		Central		
5.11.4		Central	1,487	Dec 16, 2024
5.11.3		Central	1,095	Oct 21, 2024
5.11.2		Central	539	Oct 04, 2024

Figura 4: Página de búsqueda de dependencias JUnit Jupiter API

Seleccionar una versión y copiar el bloque XML correspondiente a la dependencia.

Home » org.junit.jupiter » junit-jupiter-api » 5.13.0-M2

**JUnit Jupiter API » 5.13.0-M2**  
JUnit Jupiter is the API for writing tests using JUnit 5.

License	EPL 2.0
Categories	Testing Frameworks & Tools
Tags	quality junit testing api
HomePage	https://junit.org/junit5/
Date	Mar 24, 2025
Files	pom (3 KB) jar (233 KB) View All
Repositories	Central
Ranking	#21 in MvnRepository (See Top Artifacts) #3 in Testing Frameworks & Tools
Used By	18,044 artifacts

Maven Gradle SBT Mill Ivy Grape Leiningen Buildr

Scope: Test

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.13.0-M2</version>
  <scope>test</scope>
</dependency>
```

☒ Include backlinks

Figura 5: Dependencia de JUnit Jupiter API en formato XML

### 3.4.2. Agregar dependencia al archivo pom.xml

En la sección `<dependencies>` del archivo `pom.xml`, se debe incluir el siguiente bloque:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Luego de agregar este trozo de código, IntelliJ IDEA mostrará advertencias temporales si aún no reconoce los cambios realizados. Para solucionar esto se debe hacer clic en el botón **Load Maven Changes** en la parte superior derecha del editor.

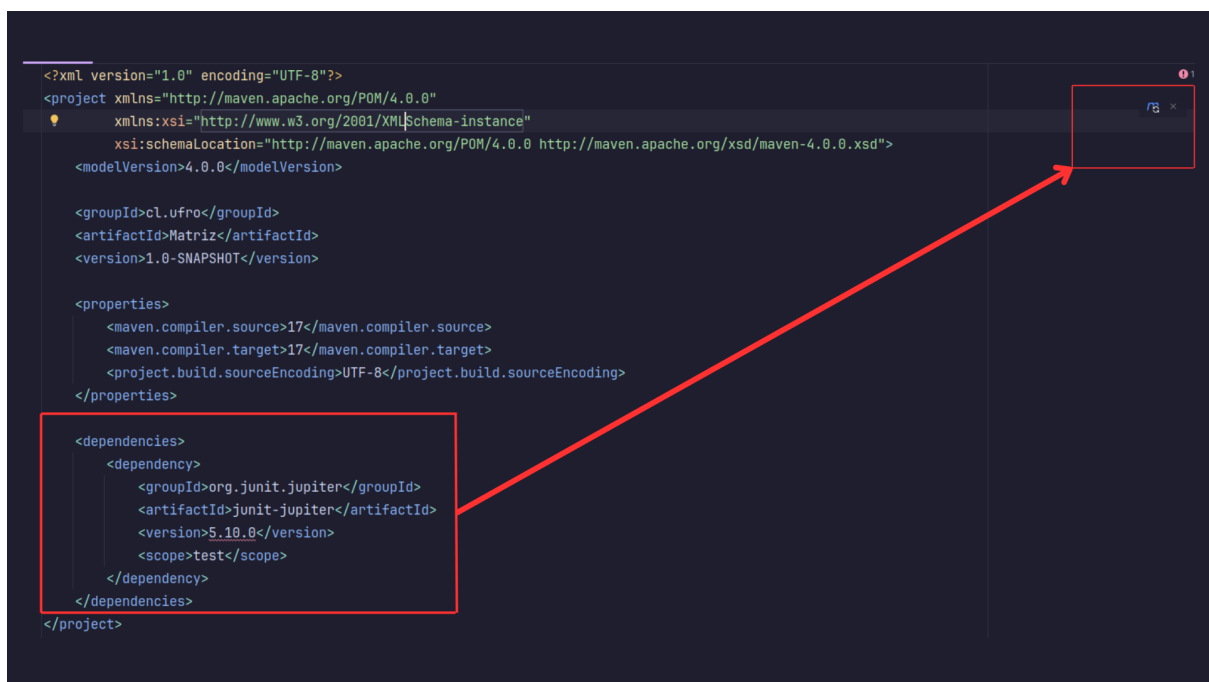


Figura 6: Sincronización del archivo `pom.xml` con Maven

```

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

Figura 7: Carga automática de dependencias tras modificar el archivo POM

### 3.4.3. Generar archivo de pruebas

IntelliJ IDEA permite generar automáticamente clases de prueba a clases ya existentes en el proyecto:

- Hacer clic derecho sobre el nombre de la clase que se desea probar.
- Seleccionar: **Generate** → **Test**.



```

public class Matriz {
    public static void main(String[] args) {
    }

    public static boolean validarDimensiones(int filas, int cols) { no usages
        return filas > 0 && cols > 0;
    }

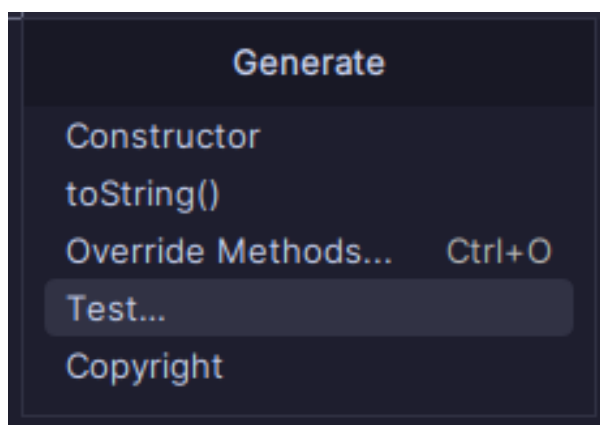
    public static int[][] crearMatriz(int filas, int cols) { no usages
        return new int[filas][cols];
    }
}

```

Context menu options:

- Show Context Actions (Alt+Enter)
- Paste (Ctrl+V)
- Copy / Paste Special
- Column Selection Mode (Alt+Shift+Insert)
- Go To
- Folding
- Analyze
- Refactor
- Generate... (Alt+Insert)

Figura 8: Generate



**Generate**

- Constructor
- toString()
- Override Methods... (Ctrl+O)
- Test...**
- Copyright

Figura 9: Generación de test para la clase

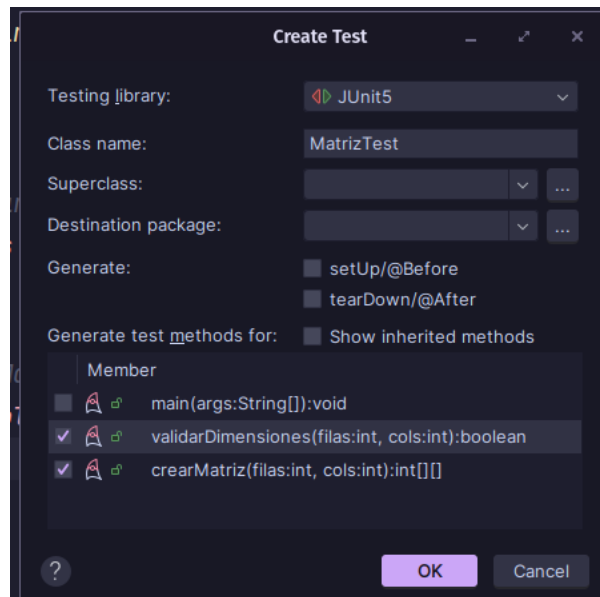


Figura 10: Configuración de la clase de prueba generada automáticamente

Se abrirá una ventana donde se podrá:

- Asignar un nombre a la clase de prueba (por defecto: **NombreClaseTest**).
- Seleccionar los métodos que se desean incluir como pruebas.
- Habilitar la creación automática de anotaciones como **@BeforeEach** o **@AfterEach**.

Una vez confirmado *IntelliJ IDEA* generará automáticamente la clase de prueba dentro del directorio *src/test/java/*, siguiendo la estructura estándar de *Maven*.

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class MatrizTest {

    @Test
    void validarDimensiones() {
    }

    @Test
    void crearMatriz() {
    }

}
```

Figura 11: Archivo de prueba generado con estructura base para pruebas unitarias