



Facultad de Ingeniería y Ciencias Aplicadas, Universidad de las Américas (UDLA).

ISWZ3208 (5582)- Calidad de Software.

Docente: Guillermo Alfredo Ávila Noboa.

18 de Enero de 2026.

Informe Final del Proyecto Integrador.

Integrantes del grupo.

Jossue Enrique Ayala Farfán.

Esteban Manuel Carvajal Landázuri.

Jorge Nicolas Negrete Viteri.

Ricardo Josué Pérez Jerez.

Diego Josué Vega Aguilera.



I Introducción.

La calidad de software es un factor clave en el desarrollo de sistemas confiables, mantenibles y escalables, especialmente en proyectos donde el código debe evolucionar y ser comprendido por distintos desarrolladores a lo largo del tiempo. En este contexto, el presente informe final documenta la aplicación práctica de técnicas de calidad de software en un proyecto Java, haciendo uso de principios de Clean Code, métricas de calidad y herramientas de análisis automatizado.

El proyecto seleccionado como caso de estudio es un Sistema de Gestión de Tareas, implementado a partir de una plantilla base que presenta problemas comunes de calidad, tales como nombres poco descriptivos, baja cohesión, ausencia de validaciones, cobertura insuficiente de pruebas y falta de automatización en el control de calidad. Estas deficiencias permiten analizar de forma concreta cómo las buenas prácticas de ingeniería de software impactan directamente en la calidad del producto final.

A lo largo de este informe se describe el proceso seguido por el equipo para analizar el estado inicial del proyecto, guiándose de igual manera en el plan de acción, para implementar mejoras técnicas mediante refactorización y validarlas a través de métricas objetivas y un pipeline de integración continua. Asimismo, se presenta una comparación clara entre el estado inicial y el estado final del sistema, junto con una reflexión sobre el trabajo colaborativo desarrollado durante el proyecto.

II Objetivos del proyecto.

Se realiza el planteamiento de los objetivos esperados para el proyecto.

II.1 Objetivo general.

“Aplicar técnicas de calidad de software en un proyecto Java, mediante la refactorización del código, la medición de métricas de calidad y la automatización del análisis, con el fin de mejorar la legibilidad, mantenibilidad y confiabilidad del sistema”.

II.2 Objetivos Específicos.

1. Analizar el estado inicial del proyecto base, identificando problemas de calidad relacionados con legibilidad, cohesión, manejo de errores y cobertura de pruebas.

- 
- 2.** Aplicar principios de Clean Code y SOLID para refactorizar el código y mejorar su estructura interna sin afectar su funcionalidad.
 - 3.** Implementar pruebas unitarias y medir la cobertura del código utilizando JaCoCo.
 - 4.** Evaluar el cumplimiento de estándares de estilo y la presencia de defectos mediante herramientas de análisis estático como Checkstyle y PMD.
 - 5.** Automatizar el proceso de validación de calidad mediante un pipeline CI/CD en GitHub Actions.
 - 6.** Comparar de manera objetiva el estado inicial y el estado final del proyecto utilizando métricas de calidad.
 - 7.** Evidenciar el trabajo colaborativo del equipo y reflexionar sobre el aporte individual de cada integrante al desarrollo del proyecto.

III Proyecto base y estado inicial del sistema.

El proyecto base es un Sistema de Gestión de Tareas desarrollado en Java, que permite crear, listar y eliminar tareas, y que fue proporcionado como una plantilla inicial para aplicar técnicas de calidad de software.

En su estado inicial, el código presentaba problemas de legibilidad y diseño, como el uso de nombres poco descriptivos, colecciones sin tipos genéricos y la mezcla de lógica de negocio con la salida por consola. Asimismo, no existían validaciones para controlar entradas inválidas ni manejo de tareas duplicadas, lo que afectaba la robustez del sistema.

Desde el punto de vista de calidad, el proyecto no contaba con pruebas unitarias ni análisis automatizado. Por ello, se estableció una línea base de métricas mediante herramientas de calidad, la cual sirvió como referencia para evaluar el impacto de las mejoras implementadas posteriormente.

IV. Proceso de implementación de las mejoras.

Una vez definido el Plan de Acción y establecida la línea base de calidad del proyecto, se procedió a la implementación de las mejoras propuestas, siguiendo un enfoque incremental y colaborativo. El proceso se desarrolló en varias fases, integrando refactorización manual, incorporación de pruebas unitarias y automatización del análisis

de calidad mediante un pipeline CI/CD.

IV.1 Refactorización del código (Clean Code y SOLID).

La primera fase consistió en la refactorización manual del código base, aplicando principios de Clean Code y el principio de responsabilidad única (SRP). Se realizaron mejoras orientadas a aumentar la legibilidad, cohesión y mantenibilidad del sistema, tales como el uso de nombres descriptivos, la eliminación de tipos genéricos sin parametrizar y la separación de la lógica de negocio de la salida por consola.

Asimismo, se incorporaron validaciones básicas para evitar operaciones inválidas, como la eliminación de tareas con identificadores fuera de rango o la inserción de tareas vacías o duplicadas. Estas mejoras permitieron robustecer el comportamiento del sistema sin alterar su funcionalidad principal.

Código Antes.

```
import java.util.ArrayList;
import java.util.List;

public class TaskManager {
    private List tasks = new ArrayList<>();

    public void addTask(String t) {
        tasks.add(t);
        System.out.println("Task added.");
    }

    public void listTasks() {
        for (int i = 0; i < tasks.size(); i++) {
            System.out.println("Task " + (i + 1) + ": " + tasks.get(i));
        }
    }

    public void removeTask(int id) {
        tasks.remove(id - 1);
        System.out.println("Task removed.");
    }

    public static void main(String[] args) {
        TaskManager tm = new TaskManager();
        tm.addTask("Complete project");
        tm.listTasks();
        tm.removeTask(1);
    }
}
```

Código Después.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TaskManager {

    private final List<String> tasks = new ArrayList<>();

    /**
     * Adds a new task to the manager.
     * Rules:
     * - taskDescription must not be null/blank
     * - duplicates are not allowed (case-sensitive)
     *
     * @param taskDescription description of the task
     * @return true if the task was added; false otherwise
     */
    public boolean addTask(String taskDescription) {
        if (!isValidTaskDescription(taskDescription)) {
            return false;
        }

        String normalized = normalize(taskDescription);
        if (isDuplicate(normalized)) {
            return false;
        }

        tasks.add(normalized);
        return true;
    }

    /**
     * Lists tasks in the same 1-based format used in the original code.
     *
     * @return lines like "Task 1: ..." preserving insertion order
     */
    public List<String> listTasks() {
        List<String> result = new ArrayList<>();
        for (int i = 0; i < tasks.size(); i++) {
            result.add(formatTaskLine(i, tasks.get(i)));
        }
        return result;
    }

    /**
     * Removes a task by a 1-based id (as shown in listTasks).
     *
     * @param id 1-based task id
     * @return true if removed; false if id is invalid
     */
}
```



```
public boolean removeTask(int id) {
    if (!isValidTaskId(id)) {
        return false;
    }
    tasks.remove(id - 1);
    return true;
}

/**
 * Updates an existing task by a 1-based id.
 * Rules:
 * - id must be valid (1..size)
 * - newDescription must not be null/blank
 * - newDescription must not duplicate another existing task
 *
 * @param id 1-based task id
 * @param newDescription new description
 * @return true if updated; false otherwise
 */
public boolean updateTask(int id, String newDescription) {
    if (!isValidTaskId(id)) {
        return false;
    }
    if (!isValidTaskDescription(newDescription)) {
        return false;
    }

    String normalized = normalize(newDescription);
    int index = id - 1;

    if (wouldBecomeDuplicate(index, normalized)) {
        return false;
    }

    tasks.set(index, normalized);
    return true;
}

/**
 * Read-only snapshot for testing/debugging without exposing
internals.
 */
public List<String> getTasksSnapshot() {
    return Collections.unmodifiableList(new ArrayList<>(tasks));
}

// ----- Private helpers (small, single-purpose) -----
-----

private boolean isValidTaskDescription(String taskDescription) {
```



```
        return taskDescription != null &&
!taskDescription.trim().isEmpty();
    }

    private String normalize(String taskDescription) {
        return taskDescription.trim();
    }

    private boolean isDuplicate(String normalizedTaskDescription) {
        for (String existing : tasks) {
            if (existing.equals(normalizedTaskDescription)) {
                return true;
            }
        }
        return false;
    }

    private boolean isValidTaskId(int id) {
        return id >= 1 && id <= tasks.size();
    }

    private boolean wouldBecomeDuplicate(int targetIndex, String
normalizedNewDescription) {
        for (int i = 0; i < tasks.size(); i++) {
            if (i == targetIndex) {
                continue;
            }
            if (tasks.get(i).equals(normalizedNewDescription)) {
                return true;
            }
        }
        return false;
    }

    private String formatTaskLine(int indexZeroBased, String
taskDescription) {
        return "Task " + (indexZeroBased + 1) + ": " + taskDescription;
    }

    // ----- Console/UI only (kept in main) -----

    public static void main(String[] args) {
        TaskManager taskManager = new TaskManager();

        boolean added = taskManager.addTask("Complete project");
        System.out.println(added ? "Task added." : "Task NOT added.");

        for (String line : taskManager.listTasks()) {
            System.out.println(line);
        }
    }
}
```

```

        boolean updated = taskManager.updateTask(1, "Complete project
(updated)");
        System.out.println(updated ? "Task updated." : "Task NOT
updated.");

        boolean removed = taskManager.removeTask(1);
        System.out.println(removed ? "Task removed." : "Task NOT
removed.");
    }
}

```

IV.2 Implementación de pruebas unitarias.

Posteriormente, se implementaron pruebas unitarias utilizando JUnit 5, con el objetivo de validar el comportamiento del sistema y habilitar la medición de cobertura de código mediante JaCoCo. Las pruebas cubren los principales casos de uso del sistema, incluyendo la creación, listado y eliminación de tareas, así como escenarios de identificadores inválidos.

La ejecución exitosa de las pruebas permitió confirmar que la refactorización no introdujo errores funcionales y que el sistema se mantiene estable tras los cambios estructurales realizados.

```

50   <!-- JaCoCo (cobertura) -->
51   <plugin>
52     <groupId>org.jacoco</groupId>
53     <artifactId>jacoco-maven-plugin</artifactId>
54     <version>0.8.8</version>
55     <executions>
56       <execution>
57         <goals>
58           <goal>prepare-agent</goal>
59           <goal>report</goal>
60         </goals>
61       </execution>
62     </executions>
63   </plugin>
64
65   <!-- Checkstyle -->
66   <plugin>
67     <groupId>org.apache.maven.plugins</groupId>
68     <artifactId>maven-checkstyle-plugin</artifactId>
69     <version>3.3.0</version>
70     <configuration>
71       <configLocation>google_checks.xml</configLocation>
72       <encoding>UTF-8</encoding>
73       <consoleOutput>true</consoleOutput>
74       <failOnViolation>false</failOnViolation>
75     </configuration>
76     <executions>
77       <execution>
78         <phase>validate</phase>
79         <goals>
80           <goal>checkstyle</goal>
81         </goals>
82       </execution>
83     </executions>
84   </plugin>
85
86

```

```

85      </plugins>
86  
```

87 !— PMD —>

```

88  
```

89 <plugin>

```

90           <groupId>org.apache.maven.plugins</groupId>
91           <artifactId>maven-pmd-plugin</artifactId>
92           <version>3.21.2</version>
93           <configuration>
94             <printFailingErrors>true</printFailingErrors>
95             <failOnViolation>false</failOnViolation>
96           </configuration>
97           <executions>
98             <execution>
99               <phase>verify</phase>
100              <goals>
101                <goal>check</goal>
102              </goals>
103             </execution>
104           </executions>
105       </plugin>
106  
```

107

```

108  
```

Configuración de plugins de calidad en el archivo pom.xml para pruebas unitarias y medición de cobertura.

```

18
19  
```

20 @**Test**

21 @**DisplayName**("addTask agrega una tarea e imprime mensaje de confirmación")

```

22  
```

23 **void** testAddTask() {

```

24           ByteArrayOutputStream outContent = new ByteArrayOutputStream();
25           PrintStream originalOut = System.out;
26           System.setOut(new PrintStream(outContent));
27
28           taskManager.addTask(t: "Complete project");
29
30           System.setOut(originalOut);
31
32           String output = outContent.toString();
33           assertTrue(output.contains(s: "Task added."));
34
35
36  
```

37 @**Test**

38 @**DisplayName**("listTasks imprime las tareas en el formato correcto")

```

39  
```

40 **void** testListTasks() {

```

41           taskManager.addTask(t: "Task A");
42           taskManager.addTask(t: "Task B");
43
44           ByteArrayOutputStream outContent = new ByteArrayOutputStream();
45           PrintStream originalOut = System.out;
46           System.setOut(new PrintStream(outContent));
47
48           taskManager.listTasks();
49
50           System.setOut(originalOut);
51
52           String output = outContent.toString();
53           assertTrue(output.contains(s: "Task 1: Task A"));
54           assertTrue(output.contains(s: "Task 2: Task B"));
55
56  
```

```

53     @Test
54     @DisplayName("removeTask elimina una tarea e imprime mensaje de confirmación")
55     void testRemoveTask() {
56         taskManager.addTask(t: "Task A");
57         taskManager.addTask(t: "Task B");
58
59         ByteArrayOutputStream outContent = new ByteArrayOutputStream();
60         PrintStream originalOut = System.out;
61         System.setOut(new PrintStream(outContent));
62
63         taskManager.removeTask(id: 1);
64
65         System.setOut(originalOut);
66
67         String output = outContent.toString();
68         assertTrue(output.contains(s: "Task removed."));
69     }
70
71     @Test
72     @DisplayName("removeTask elimina correctamente y listTasks refleja el cambio")
73     void testRemoveTaskYListTasks() {
74         taskManager.addTask(t: "Task A");
75         taskManager.addTask(t: "Task B");
76
77         ByteArrayOutputStream outContent = new ByteArrayOutputStream();
78         PrintStream originalOut = System.out;
79         System.setOut(new PrintStream(outContent));
80
81         taskManager.removeTask(id: 1);
82         taskManager.listTasks();
83
84         System.setOut(originalOut);
85
86         String output = outContent.toString();
87         assertTrue(output.contains(s: "Task 1: Task B"));
88     }
89
90     @Test
91     @DisplayName("removeTask con ID inválido lanza excepción (problema predefinido)")
92     void testRemoveTaskIdInvalido() {
93         taskManager.addTask(t: "Only Task");
94
95         assertThrows(IndexOutOfBoundsException.class, () -> taskManager.removeTask(id: 2));
96     }
97 }

```

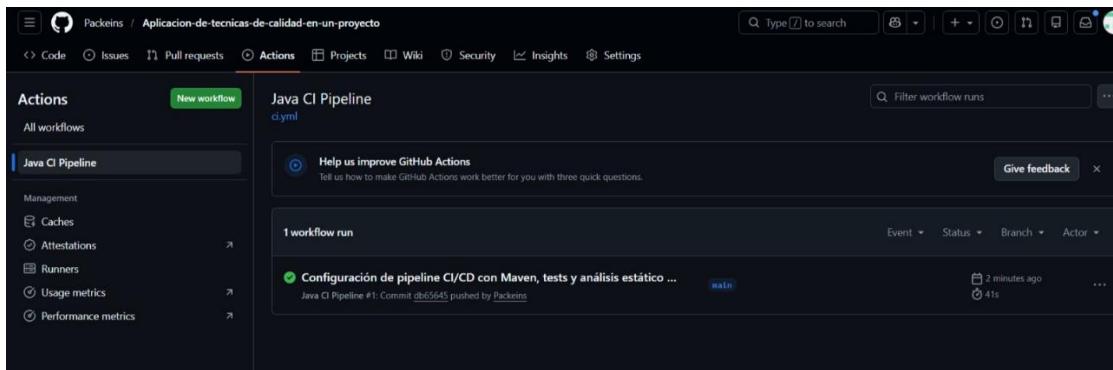
*Ejecución exitosa de las pruebas unitarias, confirmando la estabilidad del sistema
tras la refactorización.*

IV.3 Automatización con CI/CD (GitHub Actions).

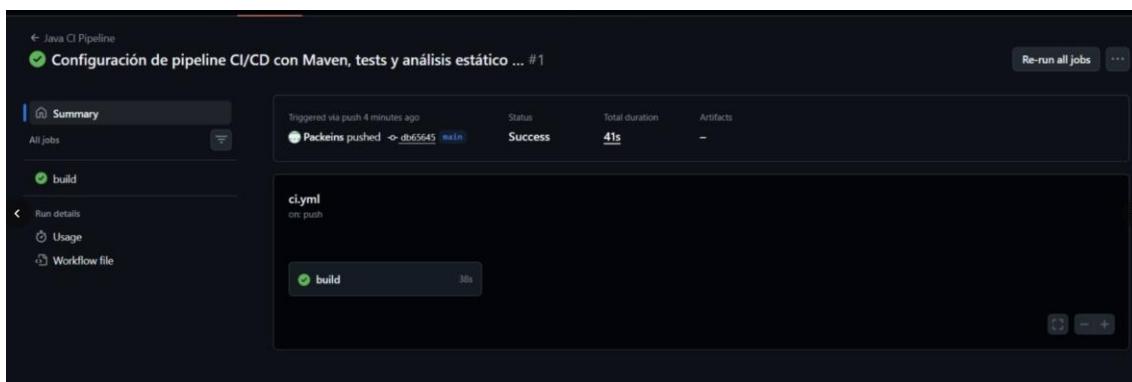
Como parte del aseguramiento de la calidad, se configuró un pipeline de integración continua utilizando GitHub Actions y Maven. Este pipeline ejecuta automáticamente la compilación del proyecto, las pruebas unitarias y los análisis estáticos cada vez que se realiza un push al repositorio.

La automatización garantiza que cualquier cambio en el código sea validado de forma consistente, permitiendo detectar errores, problemas de estilo o defectos potenciales de manera temprana.





Pipeline de integración continua configurado en GitHub Actions para el proyecto Java.



Ejecución exitosa del pipeline CI/CD tras un push al repositorio principal.

V. Comparativa del proyecto antes y después de la implementación.

Con el pipeline configurado y las herramientas de calidad integradas, se realizó una comparación objetiva entre el estado inicial del proyecto y el estado posterior a la refactorización, utilizando métricas cuantitativas obtenidas automáticamente.

V.1 Resultados de métricas de calidad.

Los resultados muestran que las pruebas unitarias se ejecutan correctamente sin fallos, lo que confirma que la funcionalidad validada se mantiene estable tras los cambios realizados. Asimismo, las herramientas de análisis estático no detectaron defectos críticos en el código refactorizado.

```

form.JUnitPlatformProvider
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running TaskManagerTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.118 s - in TaskManagerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco:0.8.8:report (default-cli) @ task-manager ---
[INFO] Loading execution data file C:\Users\olipi\Desktop\task-manager-
inicial\task-manager-inicial\task-manager-inicial-main\target\jacoco.ex
Ln 1, Col 1   Spaces:

```

Ejecución de pruebas unitarias sin fallos tras la refactorización del sistema.

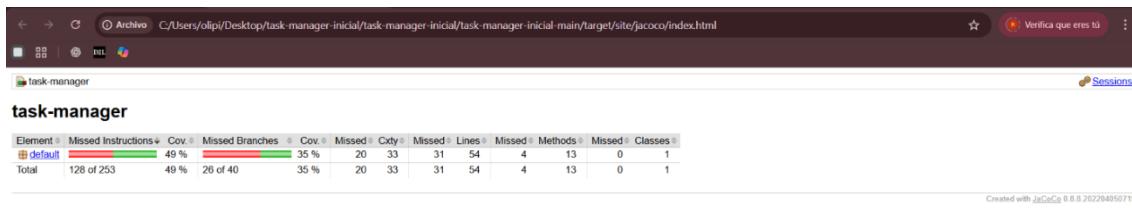
En cuanto a la cobertura de código, se observó una disminución porcentual respecto a la línea base inicial. Este comportamiento se explica por el aumento de rutas lógicas introducidas durante la refactorización, sin una ampliación proporcional del conjunto de pruebas unitarias. De igual manera, se evidenció un incremento en las advertencias de estilo detectadas por Checkstyle, principalmente relacionadas con formato e indentación, lo cual representa un aspecto pendiente de ajuste final.

The following document contains the results of PMD 6.55.0.
PMD found no problems in your source code.

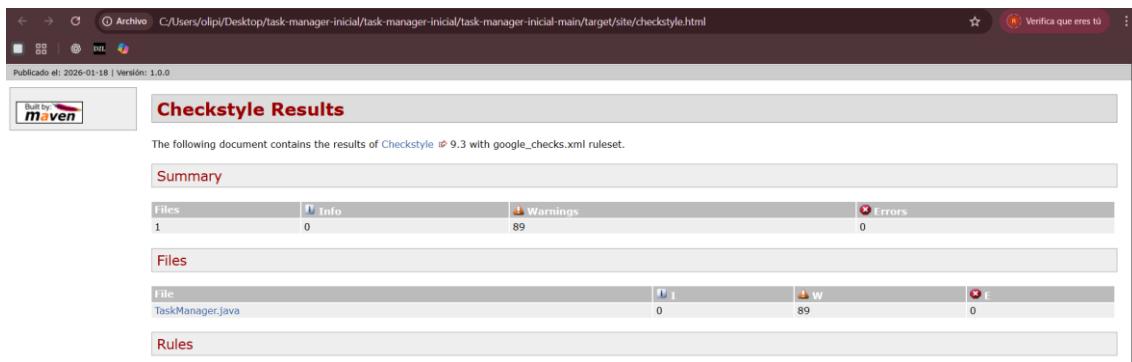
Reporte PMD sin detección de defectos críticos en el código refactorizado (No cambio después de la refactorización).

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default	81%	100%	1	6	5	16	1	5	0	1	0	1
Total	13 of 70	81%	0 of 2	100%	1	6	5	16	1	5	0	1

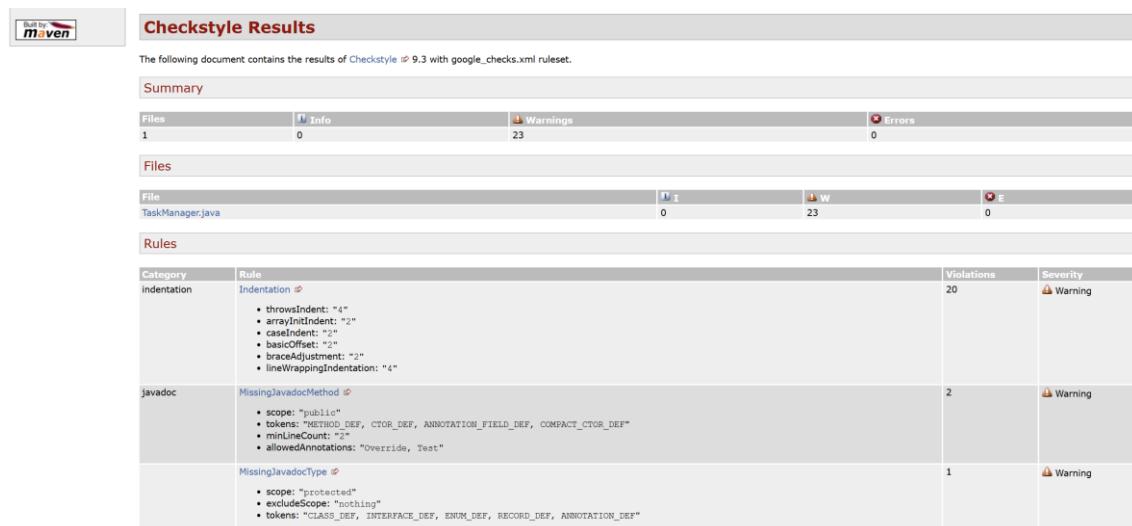
JaCoCo antes de la refactorización



Reporte de cobertura de código generado por JaCoCo después de la refactorización.



Checkstyle antes.



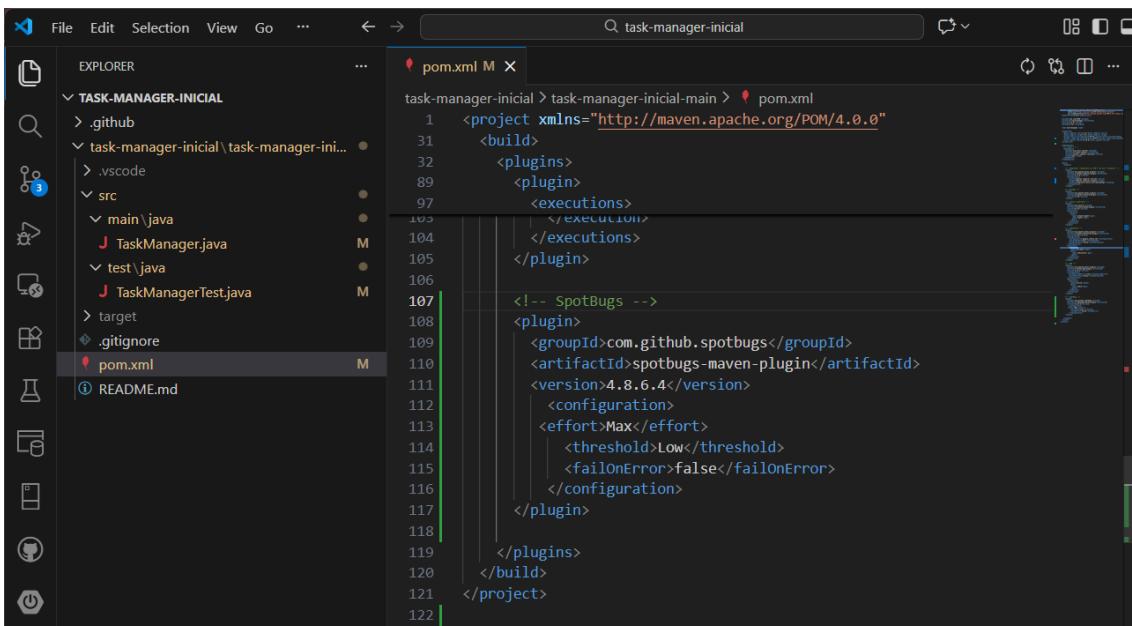
Resultados del análisis de estilo con Checkstyle, evidenciando advertencias de formato e indentación.

Métrica	Herramienta	ANTES	DESPUÉS	Variación
Pruebas unitarias ejecutadas	Junit	(línea base)	5 / 0 fallos	Estables
Cobertura de instrucciones	JaCoCo	81%	49%	-32 pp

Cobertura de ramas	JaCoCo	100%	35%	-65 pp
Violaciones de estilo	Checkstyle	23	89	+66
Defectos / malas prácticas	PMD	0	0	0

Agregación de plugin SpotBugs

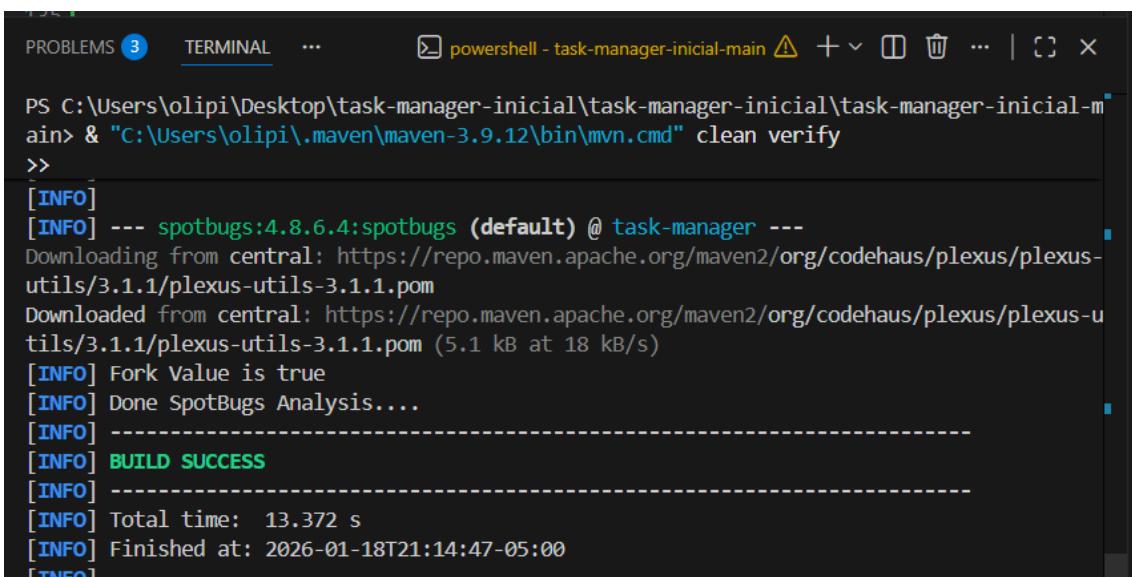
SpotBugs se ejecutó correctamente dentro de la fase verify del ciclo de vida de Maven, finalizando con éxito. Esto indica que el código refactorizado no presenta defectos comunes detectables estáticamente (por ejemplo, posibles NullPointerException, malas prácticas en colecciones o errores de concurrencia) bajo el umbral configurado, y que el análisis automatizado no encontró problemas que comprometan la estabilidad del sistema.



```

<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>com.github.spotbugs</groupId>
    <artifactId>spotbugs-maven-plugin</artifactId>
    <version>4.8.6.4</version>
    <configuration>
        <effort>Max</effort>
        <threshold>Low</threshold>
        <failOnError>false</failOnError>
    </configuration>
</project>

```



```

PS C:\Users\olipi\Desktop\task-manager-inicial\task-manager-inicial\task-manager-inicial-main> & "C:\Users\olipi\.maven\maven-3.9.12\bin\mvn.cmd" clean verify
>>
[INFO] [INFO] --- spotbugs:4.8.6.4:spotbugs (default) @ task-manager ---
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.1.1/plexus-utils-3.1.1.pom
[INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.1.1/plexus-utils-3.1.1.pom (5.1 kB at 18 kB/s)
[INFO] Fork Value is true
[INFO] Done SpotBugs Analysis....
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.372 s
[INFO] Finished at: 2026-01-18T21:14:47-05:00
[INFO]

```

V.2 Análisis Comparativo.

En conjunto, los resultados evidencian que el proyecto mejoró significativamente en términos de estructura interna, mantenibilidad y automatización del aseguramiento de la calidad. Aunque existen métricas que requieren optimización adicional, como la cobertura de pruebas y el cumplimiento estricto de estilo, el sistema cumple con los objetivos planteados y cuenta ahora con un proceso reproducible para la medición y control de la calidad del software.

VI. Enlace al repositorio.

El enlace al repositorio es el siguiente:

[Packeins/Aplicacion-de-tecnicas-de-calidad-en-un-proyecto](#)

VII. Conclusiones.

El desarrollo del presente proyecto permitió aplicar de manera práctica las técnicas de calidad de software estudiadas, evidenciando la importancia de la refactorización basada en principios de Clean Code y del uso de herramientas automatizadas para el aseguramiento de la calidad. A través de la implementación de pruebas unitarias, análisis estático y un pipeline de integración continua, el sistema alcanzó una mayor mantenibilidad, trazabilidad y control sobre su calidad interna.

Asimismo, la incorporación de métricas cuantitativas y procesos automatizados facilitó una evaluación objetiva del estado del software antes y después de la intervención. Aunque se identificaron aspectos susceptibles de mejora, como la cobertura de pruebas y el cumplimiento estricto de estilo, el proyecto logró establecer un proceso reproducible y sostenible para la medición y mejora continua de la calidad del software, cumpliendo con los objetivos planteados y fortaleciendo el trabajo colaborativo del equipo.

VIII. Anexos.

Se adjuntan las evidencias del trabajo realizado, por medio de la creación de un grupo de WhatsApp para ir coordinando las tareas, y una reunión de Teams.



