



EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Tratamiento de la Deuda Técnica por la definición de Entidades Lógicas Aisladas en Sistemas de Software existentes

presentada por

Ing. Octavio Tonatiuh Flores Millán

como requisito para la obtención del grado de

Maestro en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Codirector de tesis

Dr. Juan Carlos Rojas Pérez



Centro Nacional de Investigación y Desarrollo tecnológico
Departamento de Ciencias Computacionales

Cuernavaca, Mor., 11/septiembre/2025

OFICIO No. DCC/251/2025

Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de **Octavio Tonatíuh Flores Millán** con número de control **M22CE049**, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Tratamiento de la deuda técnica por la definición de entidades lógicas aisladas en sistemas de software existentes"** y hemos encontrado que se han atendido todas las observaciones que se le indicaron, por lo que hemos acordado aceptar el documento de tesis y le solicitamos la autorización de impresión definitiva.

ATENTAMENTE

Excelencia en Educación Tecnológica®
"Conocimiento y Tecnología al Servicio de México"

Dr. René Santaolaya Salgado
Director de tesis

Dr. Juan Carlos Rojas Pérez
Codirector de tesis

Dra. Blanca Dina Valenzuela Robles
Revisor 1

M.C. Humberto Hernández García
Revisor 2

C.c.p. Depto. Servicios Escolares.
Expediente / Estudiante



2025
Año de
La Mujer
Indígena

Interior Internado Palmira S/N, Col. Palmira,
C. P. 62490, Cuernavaca, Morelos Tel. 01 (777) 3627770, ext. 3201,
e-mail: dcc_cenidet@tecnm.mx tecnm.mx | cenidet.tecnm.mx





Centro Nacional de Investigación y Desarrollo tecnológico
Subdirección Académica

Cuernavaca Mor, 23/septiembre/2025

Oficio No. SAC/227/2025

Asunto: Autorización de impresión de tesis

**OCTAVIO TONATIUH FLORES MILLÁN
CANDIDATO AL GRADO DE MAESTRO
EN CIENCIAS DE LA COMPUTACIÓN
P R E S E N T E**

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"Tratamiento de la deuda técnica por la definición de entidades lógicas aisladas en sistemas de software existentes"**, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

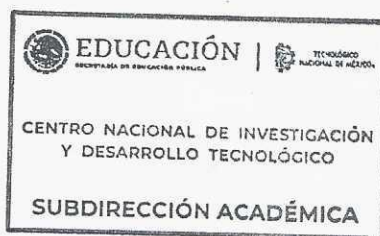
Esperando que el logro del mismo sea acorde con sus aspiraciones profesionales, reciba un cordial saludo.

ATENTAMENTE

Excelencia en Educación Tecnológica®

"Conocimiento y Tecnología al Servicio de México"

**CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO**



c.c.p. Departamento de Ciencias Computacionales
Departamento de Servicios Escolares

CMAZ/imz



2025
Año de
La Mujer
Indígena

Interior Internado Palmira S/N, Col. Palmira,
C. P. 62490, Cuernavaca, Morelos Tel. 01 (777) 3627770, ext. 4104,
e-mail: acad_cenidet@tecnm.mx tecnm.mx | cenidet.tecnm.mx

cenidet
Centro Nacional de Investigación
y Desarrollo Tecnológico



DEDICATORIAS

A mi Dios, mi Castillo Fuerte, por mostrar siempre su fidelidad, caminando conmigo en cada paso de mi formación académica y darme la fuerza para seguir adelante.

*A mis padres **Octavio Flores y Elizabeth Millán**, por ayudarme a alcanzar cada meta, su presencia en mi vida es un regalo invaluable, ¡este logro es nuestro!*

*A mis abuelitos **Mario y Amparo**, por su amor y cariño, mi gratitud hacia ustedes es imposible de expresar completamente.*

*A mis hermanos **Iván y Gerardo**, por su apoyo y cariño.*

*Con cariño a mis tíos **Miriam y Martín** y a mi prima **Roxana**, en recuerdo de **Uriel**, cuyo amor y luz siempre perdurarán en nuestros corazones.*

*Encomienda a **Jehová** tus obras,
Y tus pensamientos serán afirmados.
Proverbios 16:3*

IN MEMORIAM

*En honor a **Uriel Zúñiga Millán**, por ser un gran ejemplo de fortaleza y valentía.*

“Nos veremos allá, al final del camino... en la presencia de nuestro Salvador.”

AGRADECIMIENTOS

Al Tecnológico Nacional de México por ser una institución de excelencia, formación académica y profesional al servicio de México.

Al Centro Nacional de Investigación y Desarrollo Tecnológico por brindarme el espacio y el tiempo necesario para concluir el programa de Maestría en Ciencias de la Computación en dicha Institución.

Al Consejo Nacional de Humanidades, Ciencias y Tecnologías (CONAHCYT) por la beca económica brindada durante la realización de mis estudios de Maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por compartir su conocimiento y dirigirme en el desarrollo de esta investigación y sobre todo por brindarme su amistad.

A mis revisores, M. en C. Humberto Hernández García y Dra. Blanca Dina Valenzuela Robles, por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

A mis profesores en general por su enseñanza profesional y académica.

A mis amigos y colegas de maestría: Sinaí, Verenice, Alexis, Nelida, Antonio, Ariel, Elías, Arelia, Andros, Esteban y Eyzel gracias por su apoyo y amistad durante este periodo.

RESUMEN

El concepto de "Entropía" se aplica en una amplia gama de disciplinas, incluyendo física, química, matemáticas, astrofísica, lingüística, computación y ecología, y describe la tendencia natural de los sistemas hacia el desorden. En teorías de la información, la entropía se define como el grado de incertidumbre asociado a un conjunto de datos, lo que refleja la falta de predictibilidad y el potencial de información contenida en un mensaje. Este concepto es especialmente relevante en el desarrollo de software, donde la entropía se manifiesta como código dispensable y redundante que incrementa la complejidad y disminuye la eficiencia del sistema, lo que contribuye a la acumulación de deuda técnica y afecta negativamente la calidad y mantenibilidad del software.

Para desarrollar sistemas de software de calidad, es esencial que los desarrolladores posean conocimientos sólidos sobre los principios de diseño, técnicas adecuadas de programación y la toma de decisiones que se adapten a las necesidades del proyecto. La falta de habilidades como la documentación, el análisis de requerimientos, la creatividad y la capacidad de imaginar soluciones puede resultar en sistemas que no cumplen con los estándares de calidad, generando elementos de código dispensable y obsoleto. Este tipo de código da lugar a la acumulación de deuda técnica, lo que incrementa el costo de mantenimiento y hace que las arquitecturas de software sean menos reutilizables y más difíciles de gestionar.

Uno de los principales factores que generan deuda técnica es la presencia de elementos de código dispensable, como variables no utilizadas, clases vacías y funciones no implementadas, que incrementan la complejidad y fragilidad del sistema al violar principios fundamentales como el Principio de Sustitución.

Con el fin de abordar esta problemática, en este trabajo de investigación se propone un Método de Refactorización de Código Dispensable. El objetivo principal de este método es eliminar los elementos innecesarios del código en arquitecturas modulares, mejorando la calidad del sistema sin modificar su comportamiento. Este proceso, aplicado a código escrito en Java, tiene como propósito limpiar el código, hacerlo más eficiente y facilitar su mantenimiento a largo plazo.

Se presentan cinco casos de estudio en los que se aplica el Método de Refactorización, realizando un análisis cuantitativo del impacto de la refactorización en cada caso. Los resultados muestran una reducción significativa en la cantidad de código dispensable, así como una optimización en el uso de memoria y recursos, demostrando los beneficios de la aplicación del método de refactorización propuesto.

ABSTRACT

The concept of "Entropy" is applied across a wide range of disciplines, including physics, chemistry, mathematics, astrophysics, linguistics, computing, and ecology, and it describes the natural tendency of systems towards disorder. In information theory, entropy is defined as the degree of uncertainty associated with a set of data, reflecting the unpredictability and the potential information contained in a message. This concept is particularly relevant in software development, where entropy manifests as dispensable and redundant code that increases system complexity and reduces efficiency. This, in turn, contributes to the accumulation of technical debt, negatively affecting the quality and maintainability of the software.

To develop high-quality software systems, it is essential that developers possess solid knowledge of design principles, appropriate programming techniques, and decision-making skills that are adaptable to project needs. A lack of skills such as documentation, requirements analysis, creativity, and the ability to envision solutions can result in systems that fail to meet quality standards, generating dispensable and obsolete code elements. This type of code leads to the accumulation of technical debt, which increases maintenance costs and makes software architectures less reusable and harder to manage.

One of the main factors contributing to technical debt is the presence of dispensable code elements, such as unused variables, empty classes, and unimplemented functions, which increase the system's complexity and fragility by violating fundamental principles such as the Substitution Principle.

To address this issue, this research proposes a Method for Refactoring Dispensable Code. The main goal of this method is to remove unnecessary code elements from modular architectures, improving the system's quality without altering its behavior. This process, applied to Java-written code, aims to clean up the code, make it more efficient, and facilitate long-term maintenance.

Five case studies are presented in which the Refactoring Method is applied, conducting a quantitative analysis of the impact of refactoring in each case. The results show a significant reduction in the amount of dispensable code, as well as an optimization in memory and resource usage, demonstrating the benefits of applying the proposed refactoring method.

CONTENIDO

GLOSARIO.....	1
CÓDIGO OLOROSO / CODE SMELL	1
CÓDIGO DISPENSABLE	1
DEUDA TÉCNICA	1
ENTROPÍA	1
FRAGILIDAD.....	1
LEGIBILIDAD.....	2
META DE VALOR	2
PRINCIPIO ABIERTO/CERRADO	2
PRINCIPIO DE SUSTITUCIÓN.....	2
REFACTORIZACIÓN.....	2
RIGIDEZ	3
SISTEMAS DE SOFTWARE / UNIDADES DE SOFTWARE	3
SISTEMAS LEGADOS.....	3
ANTLR	3
CAPÍTULO 1 INTRODUCCIÓN.....	4
CONTENIDO DE ESTE DOCUMENTO DE TESIS	6
CAPÍTULO 2 ANTECEDENTES.....	7
2.1 ANTECEDENTES.....	7
2.2 PLANTEAMIENTO DEL PROBLEMA	7
2.3 SOLUCIÓN PROPUESTA	8
2.4 JUSTIFICACIÓN	9
2.5 OBJETIVOS	9
2.6 ALCANCES	9
2.7 LIMITACIONES	9
2.8 ESTADO DEL ARTE.....	10
2.9 TRABAJOS RELACIONADOS	13
CAPÍTULO 3 MARCO TEÓRICO.....	18
3.1 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)	18
3.2 CLASE	18
3.3 MÉTODO.....	18
3.4 OBJETO	19
3.5 ATRIBUTOS DE CLASE	19
3.6 CÓDIGO COMENTADO.....	19
3.7 ANTLR (ANOTHER TOOL FOR LANGUAGE RECOGNITION)	20
3.8 CÓDIGO DISPENSABLE	20
3.9 TEORÍA DE LA MEDICIÓN.....	21
3.10 VARIABLES DE OBJETO	22
3.11 PAQUETES (PACKAGES).....	22
CAPÍTULO 4 MATERIALES Y MÉTODO DE SOLUCIÓN.....	23

4.1 SISTEMA DE MEDICIÓN DE ENTROPÍA DE SOFTWARE	23
4.1 MÉTRICA DE LEGIBILIDAD DE SOFTWARE	25
4.2. INDICADOR DE ESPACIO DE MEMORIA.....	26
4.3 PROCESO DE REFACTORIZACIÓN DE CÓDIGO DISPENSABLE	27
CAPÍTULO 5 DESARROLLO DEL SISTEMA	30
DIAGRAMA CASO DE USO REFACTORIZAR CÓDIGO DISPENSABLE	30
5.1 DESCRIPCIÓN DE ESCENARIOS.....	32
5.3 DIAGRAMA DE CLASES DEL MÉTODO DE REFACTORIZACIÓN DE CÓDIGO DISPENSABLE	42
CAPÍTULO 6 PRUEBAS	44
6.1 ANÁLISIS DE ESCENARIOS	44
6.2 CONVENCION DE NOMBRES	48
6.3 PLAN DE PRUEBAS	49
6.4 CARACTERÍSTICAS A SER PROBADAS.	50
6.5 ESPECIFICACIÓN DEL DISEÑO DE PRUEBAS	52
6.6 CASOS DE PRUEBA.....	53
CAPÍTULO 7 CONCLUSIONES Y TRABAJO FUTURO.....	75
7.1 CONCLUSIONES	75
7.2 APORTACIONES DE LA TESIS	76
7.3 TRABAJOS FUTUROS.....	76
REFERENCIAS.....	77

ÍNDICE DE TABLAS

Tabla 1	Comparativa entre trabajos relacionados.....	16
Tabla 2	Descripción del Caso de Uso “Sistema de Medición de Entropía de Software”	24
Tabla 3	Descripción del Caso de Uso "Refactorización de Código Dispensable"	30
Tabla 4	Seleccionar Archivos.....	32
Tabla 5	Descripción Refactorización de código	34
Tabla 6	Convención de nombres.....	48
Tabla 7	Módulos de programa	49
Tabla 8	Control de tareas	49
Tabla 9	Características a probar	50

Índice de Figuras

Figura 1 Diagrama UML	24
Figura 2 Diagrama BPMN.....	28
Figura 3 Diagrama C.U. Refactorizar Código Dispensable	30
Figura 4 Diagrama de Secuencia del Sistema de Refactorización de Código Dispensable	35
Figura 5 Diagrama C1 Arquitectura general del sistema.....	37
Figura 6 Diagrama C2 Contenedores	38
Figura 7 Diagrama C3 Componentes	40
Figura 8 Diagrama de Clases Método Refactorización Código Dispensable.....	42
Figura 9 Variables no utilizadas.....	45
Figura 10 Diagrama de Clases, con Clase no utilizada	45
Figura 11 Funciones no Utilizadas	46
Figura 12 Función sin comportamiento.....	46
Figura 13 Código comentado	47
Figura 14 Paquetes vacíos	47
Figura 15 Convención de nombres.....	48

GLOSARIO

Código Oloroso / Code Smell

Presencia de elementos de código innecesarios o mal gestionados, se produce en sistemas o unidades de software donde existen variables, atributos o métodos, tanto de instancia como de clase, que han sido declarados, pero no son utilizados en la lógica del sistema. Estos elementos incluyen clases y métodos nulos o sin comportamiento definido, lo que no solo provoca un uso ineficiente del espacio de memoria, sino también incrementa la complejidad y dificulta la lectura y comprensión del código. Técnicamente, el código indeseable no es incorrecto ni constituye un defecto, y no impide el funcionamiento adecuado del programa. Sin embargo, señala debilidades en el diseño que pueden dificultar su evolución y aumentar el riesgo de fallos a futuro. (Fowler, 1999)

Código dispensable

Es cualquier fragmento de código que no contribuye al funcionamiento del sistema; esto incluye variables, métodos o clases que no se utilizan en el flujo lógico del programa, pero que permanecen en el código por falta de limpieza o refactorización. El código dispensable es una forma de deuda técnica, ya que aumenta la complejidad y dificulta el mantenimiento y evolución del software. Además, puede causar confusión entre los desarrolladores, generar errores imprevistos y consumir recursos innecesarios, como espacio de memoria o tiempo de procesamiento. (Fowler, 1999)

Deuda Técnica

Es el costo acumulado por decisiones de diseño o implementación subóptimas, tomadas intencionalmente o no, para acelerar el desarrollo a corto plazo. Este término se refiere a las concesiones realizadas en términos de calidad, que generan un "interés" en forma de mayor complejidad, errores futuros y esfuerzo adicional para su corrección o refactorización. Al igual que una deuda financiera, la deuda técnica debe ser "pagada" eventualmente, lo que implica un aumento en el costo de mantenimiento y la disminución de la agilidad del sistema a largo plazo. Si no se gestiona adecuadamente, puede comprometer la sostenibilidad y la evolución del software. (Cunningham, 1992)

Entropía

Deterioro gradual de la calidad y el orden en el código de un sistema a lo largo del tiempo, a medida que se introducen cambios sin un diseño cuidadoso o una refactorización adecuada. Este fenómeno es una medida cuantitativa del grado de incertidumbre o desorden asociado a un sistema o conjunto de datos. Cuando se habla de entropía negativa o neguentropía, se hace referencia a los mecanismos mediante los cuales un sistema reduce o exporta su propia entropía para mantener un estado de orden o baja incertidumbre. En el caso del software, esto se puede lograr mediante la refactorización y el mantenimiento constante del sistema. Estos mecanismos son esenciales en sistemas abiertos, donde el intercambio con otros subsistemas permite compensar la tendencia natural al desorden, conservando así un equilibrio o estado ordenado frente al inevitable proceso de degradación temporal. (Pressman, 2005)

Fragilidad

Es una debilidad que tiene un sistema de software para adaptarse a cambios. Implica la generación de fallas en módulos que aparentemente funcionan bien cuando se introducen cambios en otra parte del sistema, incluso si no hay una relación directa. Un cambio en un módulo puede propagarse a otros, afectando el funcionamiento global

del sistema. Un sistema frágil es aquel en el que las modificaciones en un bloque de código pueden provocar fallas en otros bloques que aparentemente funcionaban correctamente antes del cambio. La fragilidad aumenta la probabilidad de errores inesperados y reduce la flexibilidad del software para adaptarse a nuevos requerimientos. (C. Martin & Martin, 2006)

Legibilidad

Se refiere a lo fácil que es para un ser humano leer y comprender el código fuente de un programa. Un código legible está bien estructurado, con un uso claro de variables, funciones, y comentarios que facilitan su entendimiento por parte de otros desarrolladores. La legibilidad es un aspecto crítico en el desarrollo de software, ya que facilita la corrección de errores, la implementación de mejoras y la reutilización del código por otros miembros del equipo de desarrollo, promoviendo la mantenibilidad y eficiencia a largo plazo. (McConnell, 2004)

Meta de valor

Propósito final o al objetivo que el sistema debe alcanzar, cumpliendo con los requisitos y expectativas para su funcionamiento efectivo. Esto implica que el sistema de software debe estar diseñado y desarrollado de manera que logre su razón de ser, proporcionando las funcionalidades necesarias y alcanzando los objetivos establecidos para satisfacer las necesidades del usuario, la meta de valor alinea el software con los resultados esperados. (Leffingwell, 2011)

Principio Abierto/Cerrado

Establece que las entidades de software, como clases y módulos, deben estar abiertas para ser extendidas, pero cerradas para su modificación. Esto significa que el comportamiento de una entidad debe poder extenderse sin modificar su código fuente, favoreciendo la mantenibilidad y reducción de errores. Este principio favorece la reutilización del software y facilita la adaptación a nuevos requerimientos sin introducir errores en el sistema existente. (Meyer, 1997)

Principio de Sustitución

El Principio de Sustitución de Liskov establece que, en un programa, las instancias de una clase base deben poder ser reemplazadas por instancias de sus subclasses sin alterar el comportamiento correcto del programa. Esto garantiza que las subclasses mantengan la integridad de los contratos definidos por la clase base. (Liskov, 1994)

Refactorización

Es el proceso de modificar la estructura interna de un programa sin alterar su comportamiento externo observable, con el fin de mejorar su comprensión y facilitar futuras modificaciones. Preserva el comportamiento del software, garantizando que se mantenga consistente antes y después de realizar los cambios estructurales. (Fowler et al., 1999)

Rigidez

Es un defecto que describe la dificultad para adaptarse a cambios, incluso en las modificaciones más sencillas. Un sistema rígido tiende a propagar los efectos de un cambio en un módulo a muchos otros módulos, lo que hace que cualquier ajuste en el código sea costoso y propenso a introducir errores. Cuanto más difícil sea realizar cambios sin afectar otras partes del sistema, más rígido es dicho sistema. (C. Martin & Martin, 2006)

Sistemas de Software / Unidades de Software

Son una colección de programas, procedimientos, datos y documentación que trabajan en conjunto para realizar tareas específicas en un entorno computacional. Estos sistemas están compuestos por varios componentes o unidades de software, que son módulos o subsistemas individuales que pueden ser independientes o interdependientes entre sí. Las unidades de software pueden ser bibliotecas, servicios o componentes que cumplen con una funcionalidad específica y que interactúan con otras unidades dentro del sistema para lograr un objetivo más amplio. (Sommerville, 2011)

Sistemas Legados

Son sistemas de software antiguos que siguen siendo esenciales para el funcionamiento de una organización, pero que están basados en tecnologías obsoletas o difíciles de mantener. Estos sistemas presentan desafíos significativos, ya que suelen ser costosos de actualizar o integrar con nuevas tecnologías debido a su antigüedad y la falta de documentación o soporte. (Bisbal et al., 1999)

ANTLR

Another Tool for Language Recognition es una herramienta para reconocimiento de lenguajes que está escrito en lenguaje Java, proporciona la capacidad para construir reconocedores, intérpretes, compiladores y traductores de lenguajes (Parr, 2013).

CAPÍTULO 1 INTRODUCCIÓN

En cualquier sistema complejo, ya sea físico, biológico o informático, el concepto de entropía desempeña un papel importante para entender su evolución y comportamiento a lo largo del tiempo. En el ámbito de la física, la entropía está relacionada con la segunda ley de la termodinámica, que establece que en un sistema cerrado la entropía tiende a aumentar, lo que lleva al sistema hacia un estado de mayor desorden. Este mismo concepto ha sido adoptado en diversas disciplinas, como la informática y el desarrollo de software, para describir cómo la complejidad y el desorden pueden acumularse con el tiempo si no se gestionan adecuadamente. Para los sistemas de software, este desorden progresivo se manifiesta en dificultades crecientes para mantener, actualizar y evolucionar el sistema, lo que se conoce como "entropía del software". Dado que muchos sistemas informáticos no son cerrados, sino abiertos e interactúan con su entorno, es posible contrarrestar el aumento de entropía mediante la introducción de factores que promuevan el orden y la estabilidad del sistema, usando métodos como la refactorización de software.

En los sistemas abiertos, el incremento constante de la entropía se contrarresta mediante la importación de entropía negativa. Esto se logra a través de flujos de información, energía, y modificaciones en los objetivos o estructuras del sistema, permitiendo así reducir su entropía. Aunque estos flujos pueden generar inestabilidad, una adecuada asimilación de los mismos facilita que el sistema continúe operando eficientemente (Lara, 2004).

En el contexto del software, Ivar Jacobson (1992) explora el concepto de "entropía del software" como una medida del desorden interno, el cual refleja la creciente complejidad en su mantenimiento. Según la segunda ley de la termodinámica, en un sistema cerrado, el desorden no puede reducirse, solo mantenerse o aumentar. Este principio se aplica también a los sistemas de software, donde cada modificación tiende a incrementar su desorden o entropía, lo que complica su mantenimiento y evolución.

Las restricciones de tiempo en el desarrollo de software suelen dar lugar a la creación de sistemas difíciles de manejar y mantener a largo plazo, lo que se conoce como deuda técnica. Este término se refiere a la postergación de compromisos de calidad en favor de necesidades inmediatas, lo cual a largo plazo resulta en mayor desorden y complejidad. La acumulación de entropía en estos proyectos suele ser consecuencia de la falta de habilidades técnicas, malas prácticas, o el uso de enfoques de desarrollo obsoletos. Para mitigar este problema, es necesario inyectar neguentropía a los sistemas mediante refactorizaciones profundas y la adopción de nuevas tecnologías que reduzcan la complejidad y faciliten el mantenimiento y reuso del software (Software Guru, 2019).

Debido a la entropía de software, las empresas de desarrollo de Sistemas de Software y las empresas con activos de software están preocupados por remover de sus sistemas actuales el código oloroso. En un evento denominado "REAL" organizado por Software Guru en México en noviembre del año 2019, varias empresas entre ellas: "Bunsan", "Provident México", "Creditas Digital Plataforma", "Bitso", "Konfio", "Entropy", "Fundary", "Círculo de crédito" y "Propiedades"; mostraron casos reales sobre arquitecturas de software con deuda técnica, y mencionaron su esfuerzo de aplicar manualmente la remoción de código desagradable, así como para transformar sistemas de software monolítico en arquitecturas de Microservicios. (Software Guru, 2019)

Diversos lenguajes de programación han propuesto soluciones para mitigar las causas comunes de la entropía en el software, como el acoplamiento excesivo, la falta de encapsulamiento y la rigidez ante cambios. En los sistemas con código legado, algunas de las anomalías más frecuentes incluyen módulos y clases con responsabilidades excesivas, estructuras condicionales profundamente anidadas, o atributos y métodos que no cumplen ninguna función relevante dentro del sistema. Estas prácticas incrementan innecesariamente la cantidad de líneas de código, afectando tanto su legibilidad como su mantenibilidad.

Una anomalía recurrente se presenta cuando el código contiene atributos, variables de referencia y métodos, ya sean de instancia o de clase, que fueron declarados, pero no contribuyen al valor del sistema. Además, también es habitual encontrar clases y métodos nulos, es decir, aquellos que no tienen ningún comportamiento. Este tipo de código no solo ocupa espacio de memoria de manera innecesaria, sino que también incrementa la complejidad del sistema y dificulta su mantenimiento.

Cuando las clases o métodos no cumplen con el comportamiento esperado, se viola el principio de sustitución de Bárbara Liskov, comprometiendo el principio abierto/cerrado, que establece que todas las clases derivadas deben cumplir con los contratos definidos por sus clases base (Liskov, 1988). Los sistemas legados que presentan estas anomalías no solo desperdician recursos, sino que también incrementan la deuda técnica, lo que repercute directamente en la legibilidad y en la dificultad de su mantenimiento.

Además de las anomalías mencionadas, otro problema frecuente en los sistemas legados es la acumulación de líneas de código comentadas y la presencia de paquetes nulos o vacíos. Las líneas de código comentadas, en muchos casos, representan fragmentos de código obsoleto o descartado que ya no tiene utilidad en el sistema, pero que sigue ocupando espacio en el archivo fuente. Aunque los comentarios pueden tener la intención de conservar historial o facilitar la comprensión, su uso excesivo y desorganizado genera confusión y afecta la legibilidad del código. Esto dificulta la identificación del código activo y útil, incrementando la carga cognitiva para los desarrolladores.

Por otro lado, los paquetes nulos o vacíos que fueron creados, pero no contienen ninguna clase, método o funcionalidad activa añaden innecesaria complejidad a la estructura del proyecto. Estos paquetes pueden haber sido creados con la intención de futuras expansiones que nunca se materializaron o simplemente quedaron olvidados durante la evolución del software. La presencia de estos elementos innecesarios contribuye a la entropía del sistema, ocupando espacio y reduciendo la eficiencia sin ofrecer ningún valor funcional. Mantener estos paquetes y líneas comentadas incrementa la deuda técnica y complica tanto el mantenimiento como la evolución del sistema.

La aportación de este trabajo de tesis consiste en proporcionar a la comunidad de empresas de desarrollo de software un método y un servicio de refactorización adicional, implementado en una herramienta automatizada que aborde el problema de la entropía de software generada por código dispensable. Esta herramienta elimina y/o disminuye las anomalías descritas en la introducción, contribuyendo a la reducción de la deuda técnica mediante la eliminación de atributos y funciones aisladas, así como de clases de objetos y métodos nulos presentes en el código de sistemas de software existentes en Java. El propósito es mitigar los efectos negativos del código dispensable y mejorar la mantenibilidad y eficiencia de los sistemas.

Contenido de este documento de Tesis

El presente documento de tesis se contiene siete capítulos organizados de la siguiente manera:

En el Capítulo 2 se presenta el problema de investigación, se propone una solución al problema identificado, se establecen los objetivos de este estudio y se ofrece un resumen de los trabajos relacionados en la misma línea de investigación.

El Capítulo 3 presenta el marco teórico, donde se fundamentan los conceptos esenciales para comprender esta tesis.

En el Capítulo 4 se describen los instrumentos de medición que son: métrica de legibilidad y medición de espacio de memoria.

El Capítulo 5 detalla el proceso completo para el desarrollo de la solución propuesta, que consiste en un método de refactorización de código dispensable. Este proceso incluye el modelado del sistema mediante diagramas de casos de uso, diagramas de secuencia, diagramas C4 y diagramas de clases.

El Capítulo 6 aborda el plan de pruebas, definiendo los objetivos y el diseño necesarios para verificar el correcto funcionamiento del método.

El Capítulo 7 presenta las conclusiones, las aportaciones de esta tesis y las propuestas para trabajo futuro.

CAPÍTULO 2 ANTECEDENTES

2.1 Antecedentes

En respaldo de este enfoque, en el laboratorio de Ingeniería de Software del TecNM/cenidet, se han desarrollado diversos proyectos de reingeniería y refactorización con el objetivo de extender la vida útil y mejorar el reúso del software legado, mediante métodos de refactorización tanto de alta como de baja escala, así como técnicas de transformación para código escrito en lenguajes de programación C++ y Java. Estos proyectos incluyen el desarrollo de herramientas orientadas a:

- Adaptar interfaces lógicas que no cumplen con las necesidades de los clientes.
- Medir y refactorizar el problema de interfaces no utilizadas en marcos de aplicaciones orientadas a objetos.
- Evaluar y reducir el acoplamiento excesivo mediante refactorización, minimizando las dependencias innecesarias.
- Refactorizar marcos orientados a objetos con abstracciones incorrectas, equilibrando métricas como DIT y NOC.
- Reducir el acoplamiento entre clases relacionadas por herencia en arquitecturas orientadas a objetos.
- Refactorizar arquitecturas con funciones atómicas globalmente visibles que violan el criterio de protección modular.
- Mejorar la modularidad y reducir las dependencias entre clases en código Java.
- Refactorizar arquitecturas de software que carecen de abstracciones adecuadas.

Algunos de estos métodos de refactorización están diseñados para código fuente en C++, mientras que otros se aplican a código Java. Actualmente, se está trabajando para que todos los métodos sean compatibles con ambos lenguajes de programación.

2.2 Planteamiento del Problema

El problema se encuentra en sistemas de software que presentan desperdicio de memoria y falta de legibilidad debido a la presencia de código dispensable, como atributos, variables y métodos declarados, pero no utilizados, así como clases y métodos nulos. Esto no solo genera fragilidad a corto plazo, sino que también infringe el principio de sustitución de Liskov, al no cumplir con los contratos subyacentes en métodos y clases superfluos.

La acumulación de este código innecesario aumenta la complejidad del sistema, dificulta su mantenimiento y la realización de pruebas efectivas, y puede introducir errores imprevistos al modificar o extender el software. Además, este tipo de entropía impacta en el rendimiento del sistema y aumentan la deuda técnica, lo que lleva a costos significativos a largo plazo en términos de tiempo y recursos invertidos para corregir los problemas, afectando el tiempo de vida del software. A continuación, se enlistan algunos ejemplos:

1. Complejidad innecesaria: El código dispensable agrega complejidad innecesaria al sistema. Puede incluir funciones, clases o módulos que no se utilizan en absoluto. Esto dificulta la comprensión del sistema en su conjunto y puede llevar a errores y entropía.
2. Dificultad en el mantenimiento: El código dispensable complica el proceso de mantenimiento del software. Si hay partes del código que no se utilizan o que son obsoletas, los desarrolladores pueden perder tiempo y esfuerzo en entender y depurar esas secciones. Además, si el código dispensable no se actualiza junto con el resto del sistema, puede causar problemas de compatibilidad y vulnerabilidades de seguridad.

3. Aumento de la carga de trabajo: El código dispensable ocupa espacio en los archivos de código fuente y agrega complejidad a la base de código. Esto aumenta la carga de trabajo para los desarrolladores, ya que deben revisar y mantener código que no aporta valor o que podría eliminarse.
4. Dificultad en la detección de errores: Si el código dispensable contiene errores o comportamientos inesperados, puede dificultar la detección de esos problemas. Los desarrolladores pueden pasar por alto estos errores al concentrarse en las partes utilizadas y relevantes del código, lo que puede conducir a problemas de calidad y rendimiento en el software.
5. Impacto negativo en el rendimiento: El código dispensable puede afectar negativamente el rendimiento del software. Incluir operaciones o cálculos innecesarios consume recursos del sistema, como tiempo de procesamiento y memoria, lo que puede ralentizar el rendimiento general de la aplicación.
6. Dificultad en la escalabilidad y reutilización del código: El código dispensable puede dificultar la escalabilidad del software y su capacidad para reutilizar componentes. Si hay partes innecesarias en el código, puede ser difícil separar y utilizar esas partes en otros proyectos o ampliar el sistema de manera eficiente.

2.3 Solución Propuesta

Los objetivos de lograr sistemas y unidades de programa mantenibles y reutilizables se alcanzan mediante el uso adecuado de relaciones entre clases, un nivel adecuado de granularidad y el correcto encapsulamiento. Además, es fundamental cumplir con los criterios de modularización, que incluyen descomposición, composición, legibilidad, continuidad y protección modular, junto con los principios de diseño tanto modular como orientado a objetos. Estos principios abarcan aspectos como el uso de unidades lingüísticas, interfaces mínimas y explícitas (bajo acoplamiento y alta cohesión), ocultamiento de datos, y principios orientados a objetos como el principio abierto/cerrado, inversión de dependencias, sustitución, responsabilidad única, no-repetición y separación de interfaces.

La solución propuesta para abordar el problema planteado es el diseño de un método automatizado de refactorización de gran escala, capaz de detectar, evaluar y corregir el código innecesario presente en los activos de software desarrollados en Java. Este método tiene como objetivo eliminar las anomalías descritas previamente, como el código dispensable que contribuye a la entropía del sistema.

El sistema se basa en un analizador estático con acciones semánticas, el cual identificará las entidades de software no utilizadas y recopilará la información necesaria para cuantificar el nivel de deuda técnica asociado a este código innecesario. Además, permitirá la regeneración de código limpio, libre de elementos no utilizados, reduciendo o eliminando este tipo de deuda técnica.

El usuario seleccionará las unidades de programa escritas en Java que requieran refactorización. Tras generar el código refactorizado, el sistema medirá el nivel de deuda técnica restante para evaluar la mejora lograda, asegurando la preservación total del comportamiento funcional del código original tras el proceso de refactorización.

2.4 Justificación

Cuando una unidad de programa contiene código dispensable, se desperdician recursos de memoria y se compromete la legibilidad del sistema, lo que aumenta su fragilidad. El método de refactorización propuesto busca ofrecer a la comunidad de desarrollo de software una solución automatizada que aborde este problema, eliminando la entropía generada por las anomalías descritas en la introducción. Esta herramienta tiene como objetivo reducir la deuda técnica mediante la eliminación de código prescindible, como clases y métodos nulos, en sistemas de software desarrollados en Java. Al hacerlo, se pretende mitigar los efectos negativos del código indeseable, mejorando la mantenibilidad y eficiencia del software.

2.5 Objetivos

2.5.1 Objetivo general

Reducir la deuda técnica de sistemas de software existentes que es producida por código dispensable, por violación del principio de sustitución, derivado de arquitecturas de clases que contienen Atributos, Variables de Referencia, Métodos y Clases que no participan en el sistema para alcanzar su meta de valor, así como Clases y Métodos nulos.

2.5.2 Objetivos específicos

- Eliminación de líneas de código presentes en los sistemas legados en java que no participan en el sistema.
- Disminución del espacio de memoria ocupado para albergar atributos, variables de referencia, métodos y clases que no participan en el sistema para alcanzar su meta de valor.
- Mejorar la legibilidad del código del sistema existente.

2.6 Alcances

- Crear un método y una herramienta de refactorización que detecte el problema presentado y lo solucione de manera automática.
- Integrar las reglas semánticas y sintácticas que expresadas en lenguaje Java.
- Análisis léxico-sintáctico de la gramática del lenguaje de las aplicaciones con código legado que se explorarán, mediante el metalenguaje ANTLR.
- Con la información que el método recabe, se analizara el código para localizar, valorar y reparar estas anomalías presentes en el código, para posteriormente generar un código refactorizado.
- Los nuevos archivos funcionarán exactamente igual que los archivos originales, pero la arquitectura será de mayor legibilidad, reduciendo o logrando eliminar el código prescindible.

2.7 Limitaciones

- La refactorización que se lleva a cabo es bottom-up, que parte del análisis del código fuente.
- El código debe estar en el lenguaje Java versión 8.1 orientado a objetos.

- El código debe estar en backend.
- Se debe considerar que el código original a refactorizar posee cierto comportamiento funcional el cual debe ser preservado totalmente después del proceso de refactorización.
- La refactorización llevada a cabo por este método sólo resuelve el problema de dependencia código prescindible; otro tipo de problemas de diseño, estructura o codificación que presenten los sistemas de código legado, no serán resueltos, ya que existen o existirán otros métodos que se encarguen de esos otros problemas.

2.8 Estado del Arte

Previo a este trabajo, en el Centro Nacional de Investigación y Desarrollo Tecnológico se han llevado a cabo diversos proyectos de reingeniería con el objetivo de optimizar las arquitecturas de software desarrolladas en los lenguajes C++ y Java, los cuales se detallan a continuación.

2.8.1 SR2-Refactoring

La herramienta principal que sirve como antecedente es SR2-Refactoring (Sistema de Reingeniería para Reúso). Este sistema implementa varios métodos de refactorización, que en conjunto mejoran la arquitectura de marcos de aplicaciones orientadas a objetos. De estos métodos, tres están dirigidos a código escrito en C++ y cinco a código en Java.

Actualmente, SR2-Refactoring utiliza diversas métricas para identificar problemas específicos de diseño en marcos de aplicaciones orientadas a objetos. El sistema cuenta con un menú que facilita la ejecución de las acciones de refactorización, el cálculo de métricas y otras funcionalidades adicionales, como la selección y comparación de archivos y la gestión de usuarios.

La mayoría de las opciones del menú llevan a pantallas o cuadros de diálogo, cada uno de ellos con elementos gráficos como botones y cuadros de texto. Además, se ha desarrollado un proyecto basado en un modelo SaaS desplegado en la nube que ofrece diferentes servicios de refactorización, tanto de alta como de baja escala.

SR2-Refactoring fue implementado en lenguaje Java, utilizando el entorno de desarrollo Eclipse, y emplea MySQL como gestor de bases de datos.

2.8.2 “Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces” (Valdés, 2004)

Este proyecto de investigación desarrolló el método de refactorización denominado "Separación de Interfaces", implementado en la herramienta SR2 Refactoring. El método refactoriza automáticamente el código de marcos orientados a objetos escritos en C++, reduciendo las dependencias de herencia de interfaz entre clases, causadas por la implementación nula de interfaces no utilizadas en clases derivadas. Este problema viola el principio de sustitución de Barbara Liskov, al debilitar las postcondiciones en las clases derivadas. Además, se creó la métrica “V-DINO” para medir la dependencia causada por interfaces no utilizadas.

2.8.3 “Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator” (Cárdenas, 2004)

Este proyecto planteó un método de refactorización enfocado en reducir el acoplamiento entre clases, derivado de numerosas relaciones de asociación y dependencia en sistemas de software escritos en C++. La refactorización se basa en el patrón de diseño "Mediator", el cual reorganiza la arquitectura para reducir la cantidad de canales de comunicación entre clases. El método fue automatizado e incorporado en la herramienta SR2 Refactoring. También se implementó la métrica "COF" para calcular el nivel de acoplamiento.

2.8.4 “Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter” (Santos, 2005)

Este proyecto desarrolló un algoritmo para adaptar interfaces entre el código cliente y el código servidor cuando sus parámetros no coinciden en tipo o número. El método genera adaptadores de interfaces de forma semiautomática, ajustando las interfaces a las necesidades del código cliente. Este algoritmo fue integrado en la herramienta SR2 Refactoring y es aplicable al código existente escrito en C++.

2.8.5 “Método de Refactorización de Código Java con Interfaces y Abstracciones Incorrectas” (Padilla, 2019)

Este trabajo implementa el "principio de separación de interfaces" para automatizar la división de responsabilidades entre clases. El método detecta estructuras de código problemáticas, como abstracciones incorrectas que no respetan los principios de "sustitución" y "única responsabilidad", y las corrige automáticamente en aplicaciones escritas en Java. El método también elimina implementaciones nulas en clases derivadas y equilibra jerarquías de herencia tanto vertical como horizontalmente para distribuir correctamente las responsabilidades.

2.8.6 “Refactorización de Código para Reducir el Acoplamiento entre Clases Relacionadas por Herencia de Implementación en Arquitecturas Orientadas a Objetos” (Ortiz, 2020)

Este proyecto desarrolló un método de refactorización que reduce el acoplamiento entre clases relacionadas por herencia de implementación en sistemas de software escritos en Java. El método busca disminuir la interdependencia entre objetos y clases causadas por la herencia de implementación. Se diseñaron cinco métricas de calidad, tres de las cuales miden el factor de herencia de implementación y dos el factor de flexibilidad de las aplicaciones orientadas a objetos.

2.8.7 “Método de refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles” (Barón, 2019)

Esta tesis aplica el criterio de protección modular, asegurando que solo las funciones que actúan como iniciadores de Casos de Uso, funciones abstractas en clases abstractas o funciones de interfaces todas con acceso externo, tengan el alcance "Public" en el código de marcos de aplicaciones orientadas a objetos (MAOO's).

El proyecto incluye el cálculo del nivel de protección modular de los servicios reutilizables obtenidos a partir del software legado de MAOO's, así como una comparación del número de servicios antes y después de la refactorización. Aunque actualmente no existe una métrica estándar para medir la protección modular, esta tesis propone una nueva métrica basada en la teoría de la medición para abordar esta necesidad.

2.8.8 “Reducción de la fragilidad del software legado originado por el acceso a datos compartidos” (Meza, 2022)

Este trabajo de investigación aborda la necesidad de fortalecer la protección de los atributos de clase en aplicaciones Java, complementando los esfuerzos anteriores enfocados en la protección de funciones de clase. Se desarrolla un método de refactorización para identificar y asignar correctamente los calificadores de alcance (como private, protected, friendly) a los atributos de las clases. Para evaluar la efectividad del método, se introducen cinco métricas de calidad: PMAP, PMAPr, PMAF, PMA y TPMA, las cuales miden el grado de protección modular en diferentes niveles de visibilidad.

2.8.9 “Refactorización de Módulos Colaborativos con Carencia de Abstracción” (Sánchez, 2023)

Este trabajo aborda la carencia de abstracción en arquitecturas de software orientadas a objetos, lo que limita el uso del polimorfismo y genera rigidez frente a cambios funcionales, impidiendo la extensión a nuevas funcionalidades. Para solucionar esta falta de flexibilidad, se propone el un método de refactorización, implementado en Java, que automatiza la refactorización de arquitecturas que no utilizan adecuadamente clases abstractas o interfaces. El método se apoya en métricas de Abstracción y Factor de Polimorfismo para evaluar el nivel de abstracción y polimorfismo en sistemas existentes. Su objetivo principal es identificar y corregir estas carencias mediante un análisis estático que transforma clases base y abstractas en interfaces, mejorando la flexibilidad, reutilización y robustez del diseño.

2.8.10 “Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado” (Ramírez, 2023)

Este trabajo tiene como objetivo es reducir la deuda técnica en sistemas de software existentes, ocasionada por el código innecesario y mal estructurado, resultado de una deficiencia en la protección modular de los métodos dentro del patrón de diseño plantilla. Esta falta de protección modular puede llevar a problemas de accesibilidad indebida a métodos que deberían estar encapsulados, lo que genera vulnerabilidades y reduce la mantenibilidad del software. Además, la ausencia de un adecuado control en la visibilidad de los métodos puede incrementar el acoplamiento entre clases, dificultando la evolución del sistema y aumentando el riesgo de introducir errores durante modificaciones o expansiones. El enfoque propuesto busca optimizar la arquitectura, asegurando una correcta modularidad y encapsulamiento para mejorar la robustez y flexibilidad del software.

2.8.11 “Disminución de deuda técnica producida por arquitectura de clases con más de una responsabilidad” (Díaz, 2023)

El trabajo aborda la problemática de la deuda técnica generada por la falta de separación de responsabilidades en arquitecturas modulares, lo que aumenta la fragilidad del sistema y los costos de mantenimiento. Para resolver esto, se propone un método de refactorización, cuyo objetivo es garantizar que cada módulo en una arquitectura de software escrita en Java tenga una única responsabilidad. Además, se introduce una nueva métrica, el "Número de Responsabilidades (NR)", para medir la correcta asignación de responsabilidades en los módulos, permitiendo evaluar la calidad del software antes y después de la refactorización.

2.9 Trabajos Relacionados

2.9.1 Exploring the Impact of Code Smells on Fine-Grained Structural Change-Prone

(Liu et al., 2018)

Este trabajo tiene como objetivo desarrollar una técnica eficaz para priorizar y clasificar los olores de código (code smells) más importantes en un sistema, lo que permite a los desarrolladores enfocarse primero en resolver los olores que más afectan la estructura del sistema durante la fase de refactorización. A diferencia de otras técnicas que se limitan a detectar olores de código, esta técnica propone un enfoque que también tiene en cuenta la relevancia contextual de los olores para el desarrollador, dándole prioridad a aquellos con baja gravedad pero alta relevancia. El sistema utiliza datos sobre los cambios en el código y una herramienta de detección de olores de código para identificar módulos críticos. Fue evaluado en cuatro proyectos de código abierto, y los resultados mostraron que la técnica prioriza más eficientemente los olores de grano grueso (nivel de clase) frente a los de grano fino (nivel de método).

2.9.2 Manual Refactoring Changes with Automated Refactoring Validation (Murphy-hill, 2014)

El objetivo de la investigación presentada es detectar refactorizaciones realizadas manualmente y comprobar automáticamente su corrección. En la investigación se desarrolló una técnica automática de análisis estático llamada Ghost-Factor que separa la transformación de la verificación de corrección: permite al desarrollador transformar el código manualmente, y automáticamente comprueba la corrección de su transformación. Esta técnica fue implementada en un plug-in de código abierto para Visual Studio. Este complemento, también llamado Ghost-Factor, notifica de inmediato a los desarrolladores cuando refactorizan incorrectamente y sugiere maneras de corregir el error.

2.9.3 Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort

(Arif, A., & Rana, Z. A., 2021)

Este trabajo de investigación tiene como objetivo estudiar el impacto que tiene la eliminación de la deuda técnica (DT) en el esfuerzo de mantenimiento y en la implementación de nuevos requisitos. Se planteó la hipótesis de que eliminar los malos olores de código, identificados como deuda técnica, reduce el esfuerzo necesario para agregar nuevas funcionalidades y corregir errores. Se realizó un experimento en cinco proyectos desarrollados en C#, usando herramientas como ReSharper para identificar y limpiar los malos olores. Los resultados mostraron que, al eliminar los olores de código antes de agregar un nuevo requerimiento, se requiere un 7% menos de esfuerzo en comparación con mantener el código sin refactorizar.

2.9.4 Refactoring Code Clone Detection (Othman, Z. S., & Kaya, M., 2019)

Este trabajo de investigación se centra en la detección y refactorización de clones de código, proponiendo un algoritmo que detecta fragmentos de código similares en el sistema y los reemplaza con una única llamada a función. Basado en la estructura del árbol de alcance y el análisis de variables, el algoritmo identifica los clones y los presenta al usuario para su refactorización. Se realizó una prueba con archivos prediseñados que contenían clones, y el algoritmo demostró ser efectivo al identificar candidatos para la refactorización.

2.9.5 Proactive Clone Recommendation System for Extract Method Refactoring (Yoshida, N., Numata, S., Choi, E., & Inoue, K., 2019)

Este trabajo propone un sistema proactivo para la refactorización de clones de código, centrado en la actividad del desarrollador. El sistema monitoriza la edición del código y, al detectar que el desarrollador está realizando una refactorización del tipo "Extract Method", recomienda candidatos para refactorizar clones. El sistema fue implementado como un complemento para Eclipse y se comparó con GemX, una herramienta de extracción de clones basada en tokens. En un experimento con desarrolladores, el sistema mostró ser más eficaz que GemX, permitiendo a los usuarios refactorizar más clones en menos tiempo.

2.9.6 Code Smell Analyzer: A Tool To Teaching Support Of Refactoring Techniques Source Code (Sirqueira, T. F. M., & Brandl, A. H. M., 2016)

Esta investigación presenta una herramienta llamada Code Smell Analyzer, diseñada para ayudar en la enseñanza y evaluación de técnicas de refactorización. La herramienta identifica cinco tipos de malos olores, como código duplicado y métodos largos, y ofrece explicaciones detalladas sobre cómo corregirlos. Fue evaluada en 20 trabajos académicos de estudiantes de informática, demostrando que identifica correctamente los malos olores, siendo los métodos largos los más comunes.

2.9.7 How do software development teams manage technical debt? – An empirical study (Yli-Huumo, J., Maglyas, A., & Smolander, K., 2016)

Este estudio de caso exploratorio busca entender cómo los equipos de desarrollo de software gestionan la deuda técnica (DT), ya sea intencional o involuntaria, que surge de código mal estructurado (code smells) o que no sigue pautas de codificación. Se realizaron entrevistas a ocho equipos de desarrollo de una organización de software, y se examinaron las actividades de gestión de la deuda técnica (GDT), que incluyen la identificación, medición, monitoreo y priorización de la DT. Los resultados mostraron que la mayoría de los equipos no implementa un enfoque sistemático para la gestión de la DT, y solo unos pocos la gestionan como parte regular de su proceso de desarrollo. Se propuso un marco técnico dividido en cinco secciones para mejorar las actividades de GDT.

2.9.8 Automated refactoring to the “NULL OBJECT” design pattern (Singh, S., & Kaur, S., 2016)

El objetivo de la investigación es hacer un nuevo método para la refactorización automatizada a “NULL OBJECT” que elimina los condicionales de verificación nula asociados con los campos de clase opcionales. Es decir, campos que no se inicializan en todas las instancias de clase y, por lo tanto, su uso debe ser protegido para evitar referencias nulas.

En la investigación se desarrolló un algoritmo para el descubrimiento automático de oportunidades de refactorización para “NULL OBJECT”. Además, especifican el procedimiento de transformación del código fuente y un amplio conjunto de condiciones previas de actualización para refactorizar de forma segura un campo opcional y sus condiciones de verificación nula asociadas al patrón de diseño “NULL OBJECT”. El método se implementa como un complemento de Eclipse y se evalúa en un conjunto de proyectos de código abierto de Java.

2.9.9 Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality

Attributes: A Systematic Literature Review (Al Dallal, J., & Abdin, A., 2016)

El objetivo de este artículo es presentar los resultados de una revisión sistemática (RS) sobre el impacto de la refactorización en la calidad del software. La RS se llevó a cabo siguiendo las pautas de Kitchenham y Charters, y se enfocó exclusivamente en estudios empíricos que investigaran el impacto de la refactorización en sistemas orientados a objetos con problemas de malos olores. De un total de 2,259 artículos potenciales, se seleccionaron 76 estudios primarios que cumplieran con los criterios de inclusión. Los resultados revelan que los efectos de la refactorización sobre los atributos de calidad del software no siempre son consistentes: algunos aspectos de calidad mejoran, mientras que otros pueden verse perjudicados. Se concluye que la refactorización no garantiza mejoras globales en la calidad del software, sino que su impacto es más específico a características particulares del código.

2.9.10 Automated refactoring of legacy Java software to enumerated types (Khatchadourian, 2016)

El objetivo de la investigación presenta un enfoque de preservación de la semántica aumenta la seguridad del tipo, produce un código que es más fácil de comprender, elimina la complejidad innecesaria y elimina los problemas de fragilidad debido a la composición por separado.

En la investigación se desarrolló un algoritmo de inferencia de tipo inter- procedural que rastrea el flujo de valores enumerados. El algoritmo se implementó como un plug-in de Eclipse de código abierto, disponible para el público y se evaluó experimentalmente en 17 grandes puntos de referencia de Java.

Como resultado de este artículo le indica que el costo del análisis es práctico y el algoritmo puede refactorizar con éxito una cantidad sustancial de campos a los tipos enumerados. Este trabajo es un paso importante para proporcionar soporte automatizado de herramientas para migrar el software heredado de Java a las tecnologías Java modernas.

2.9.11 Green software: Refactoring approach (Sehgal, Mehrotra, et al, 2020)

Este estudio explora el impacto de la refactorización en el consumo de energía en aplicaciones basadas en Java mediante la identificación y eliminación de "códigos mal formados". El objetivo del estudio es mejorar la eficiencia del código y reducir las pérdidas de energía. La metodología implica seleccionar aplicaciones Java de GitHub, identificar "códigos mal formados", refactorizar manualmente el código y medir el consumo de energía utilizando Microsoft Joulemeter. Los resultados muestran una correlación entre la refactorización y la reducción del consumo de energía. El estudio investiga el impacto de los "códigos mal formados" en el consumo de energía en aplicaciones basadas en Java. Los "códigos mal formados" se detectan utilizando Jdeodorant y se aplican técnicas de refactorización para reducirlos. El estudio muestra una mejora significativa en el consumo de energía del código refactorizado.

La diferencia clave entre esta revisión de trabajos relacionados y lo propuesto en esta tesis radica en que, según la revisión de la literatura, aún no existe un método que refactorice automáticamente el código dispensable. En la Tabla 1 se presenta un resumen comparativo de los grupos de trabajo y herramientas mencionados anteriormente, destacando los resultados, si dan un tratamiento al código dispensable si las herramientas de refactorización operan de manera automatizada, los resultados, las métricas usadas, además de su comparación con los objetivos de esta tesis.

Tabla 1
Comparativa entre trabajos relacionados

Trabajo de Investigación	Código Dispensable	Resultados	Tipo de Proceso	Métricas usadas	Tipo de Herramienta/Lenguaje
<i>Liu et al., 2018</i>	Sí	Priorización eficiente de olores de grano grueso (nivel de clase).	Automática	Grano grueso vs grano fino	Herramienta de detección de olores de código, evaluada en 4 proyectos de código abierto.
<i>Murphy-hill, 2014</i>	No	Valida automáticamente las transformaciones manuales, corrigiendo errores.	Semi-Automática	Ghost-Factor	Ghost-Factor (plug-in para Visual Studio)
<i>Arif & Rana, 2021</i>	Sí	Reducir los malos olores de código disminuye el esfuerzo en un 7%.	Semi-Automática	7% de reducción en esfuerzo	ReSharper, C#
<i>Othman & Kaya, 2019</i>	Sí	El algoritmo detecta y refactoriza efectivamente clones de código.	Automática	Detección de clones	Algoritmo de detección de clones, probado en archivos prediseñados.
<i>Yoshida et al., 2019</i>	Sí	Refactorización de clones más rápida que con GemX.	Automática	Comparación con GemX	Complemento para Eclipse, comparado con GemX.
<i>Sirqueira & Brandl, 2016</i>	Sí	Identifica correctamente olores de código como métodos largos.	Automática	Cinco olores de código	Code Smell Analyzer (plug-in para Eclipse), evaluado en trabajos académicos.
<i>Yli-Huumo et al., 2016</i>	No	Propuesta de un marco técnico para la gestión de la deuda técnica.	No Aplica	No Aplica	Entrevistas a equipos de desarrollo de software.
<i>Singh & Kaur, 2016</i>	No	Algoritmo de refactorización implementado y probado en proyectos Java.	Automática	NULL OBJECT pattern	Complemento para Eclipse, aplicado en proyectos de código abierto de Java.
<i>Al Dallal & Abdin, 2016</i>	Sí	Resultados mixtos: algunos aspectos de calidad mejoran, otros empeoran.	No Aplica	Calidad de software	Revisión sistemática de 76 estudios primarios.

<i>Khatchadourian, 2016</i>	No	Mejora la seguridad del tipo y la comprensión del código.	Automática	Seguridad de tipo, complejidad	Plug-in de Eclipse para refactorizar tipos enumerados en Java.
<i>Sehgal, Mehrotra, et al., 2020</i>	Sí	Refactorización reduce significativamente el consumo de energía.	Manual	Consumo de energía con Microsoft Joulemeter	Medición del consumo de energía con Microsoft Joulemeter en aplicaciones Java de GitHub.
Esta Tesis	Sí	Refactorización para reducir la deuda técnica en sistemas de software existentes causada por código prescindible, violaciones del principio de sustitución, arquitecturas de clases que contienen atributos, variables de referencia, métodos y clases que no contribuyen al objetivo del sistema, además de clases y métodos nulos.	Automática	Legibilidad, Espacio de Memoria	Análisis Léxico-Sintáctico del código mediante ANTLR, Aplicación desarrollada en Eclipse

CAPÍTULO 3 MARCO TEÓRICO

3.1 Programación Orientada a Objetos (POO)

Es un modelo de programación que se basa en la idea de que el software se puede estructurar a partir de objetos, que son entidades que agrupan datos (atributos) y comportamientos (métodos). Este paradigma facilita la organización del código de una manera modular, permitiendo la reutilización, el mantenimiento y la escalabilidad de las aplicaciones (Booch, 1991).

La POO se basa en cuatro principios fundamentales que son abordados en la mayoría de las fuentes teóricas:

1. **Encapsulamiento:** Los datos y métodos que manipulan dichos datos se agrupan en una unidad llamada **clase** u **objeto**, protegiendo los datos internos del acceso directo desde otras partes del programa. Esto garantiza que la información se gestione de forma segura y controlada (Meyer, 1988).
2. **Abstracción:** La abstracción permite que los detalles internos de una implementación se oculten y solo se muestren las funcionalidades esenciales del objeto. Los programadores trabajan con **interfaces** claras, sin necesidad de conocer todos los detalles de cómo están implementados internamente los objetos (Booch, 1991).
3. **Herencia:** La herencia es el mecanismo por el cual una **clase** puede derivarse de otra, heredando atributos y métodos. Esto permite la creación de jerarquías y promueve la reutilización del código, ya que las clases hijas pueden extender o modificar el comportamiento de las clases padre (Meyer, 1988).
4. **Polimorfismo:** El polimorfismo permite que un objeto pueda tomar muchas formas. Esto significa que diferentes clases pueden ser tratadas de manera uniforme a través de una interfaz común, permitiendo que se utilicen métodos de manera flexible según el tipo de objeto que lo implemente (Meyer, 1988).

Uno de los objetivos principales de la Programación Orientada a Objetos es promover la reutilización de código. Al desarrollar un sistema utilizando este paradigma, las clases actúan como elementos clave, proporcionando mecanismos como el encapsulamiento, la abstracción y la ocultación de información, lo que facilita la reutilización de componentes en diferentes contextos (García Peñalvo et al., 1997).

3.2 Clase

Una clase se define como una plantilla o modelo que describe un conjunto de objetos con características y comportamientos comunes. La clase especifica tanto los atributos (propiedades o datos) como los métodos (acciones o funciones) que los objetos de ese tipo pueden realizar (Booch, 1991). En otras palabras, una clase es la estructura básica a partir de la cual se crean instancias o objetos. Las clases permiten la organización y modularidad del código, ya que encapsulan tanto los datos como las funciones en un solo lugar, facilitando la reutilización y el mantenimiento del software (Meyer, 1988).

3.3 Método

Un método es una función o procedimiento asociado a una clase que define las operaciones que los objetos de esa clase pueden realizar. Los métodos permiten que los objetos interactúen con sus propios datos (atributos) y con

otros objetos del sistema (García Peñalvo et al., 1997). Estos métodos son llamados sobre las instancias u objetos de la clase para realizar tareas específicas. Además de realizar acciones, los métodos pueden devolver valores y recibir parámetros. La correcta definición de métodos es fundamental para lograr una buena abstracción y modularidad, permitiendo que el código sea más legible y mantenible (Meyer, 1988).

3.4 Objeto

En la **Programación Orientada a Objetos**, un **objeto** es una instancia concreta de una clase. Mientras que la clase define la estructura y el comportamiento general (como un blueprint o plano), el objeto es la realización de esa estructura en un contexto específico. Cada objeto tiene su propio estado (valores de sus atributos) y puede realizar acciones (a través de sus métodos) basadas en la definición de la clase (Meyer, 1988).

Por ejemplo, si tienes una clase llamada **Coche**, cada **objeto** creado a partir de esa clase representará un coche específico. Cada uno tendrá sus propios valores para los atributos (como el color, modelo, marca, etc.) y podrá ejecutar las acciones definidas por los métodos (como acelerar o frenar).

3.5 Atributos de Clase

Los atributos de una clase son las variables que pertenecen a la clase y que representan las características o propiedades de los objetos que se instancian a partir de esa clase. Estos atributos almacenan el estado de cada objeto. (Booch, 1991).

Los atributos pueden ser de dos tipos:

1. **Atributos de instancia:** Cada objeto tiene su propia copia de estos atributos.
2. **Atributos de clase:** Son compartidos por todas las instancias de la clase.

3.6 Código Comentado

El código comentado se refiere a anotaciones dentro del código fuente que no son ejecutadas por el programa, pero que sirven para proporcionar información adicional sobre el propósito, la lógica o el funcionamiento del código. Los comentarios son cruciales para la mantenibilidad del software, ya que facilitan la comprensión del código, tanto para el autor como para otros desarrolladores que puedan trabajar en el proyecto (García Peñalvo et al., 1997).

Existen dos tipos comunes de comentarios:

- Comentarios de línea, que generalmente se escriben en una sola línea del código.
- Comentarios de bloque, que abarcan varias líneas y se utilizan para explicar en detalle una porción más extensa de la lógica.

Aunque el código comentado no afecta la ejecución del programa, su uso excesivo o insuficiente puede ser contraproducente. Es importante mantener los comentarios claros, concisos y relevantes, explicando el porqué del código en lugar del cómo, ya que el propio código bien estructurado debería expresar cómo funciona (Booch, 1991).

3.7 ANTLR (ANother Tool for Language Recognition)

ANTLR es una herramienta utilizada para la creación de analizadores léxicos y sintácticos a partir de gramáticas definidas formalmente. Es ampliamente utilizada para procesar lenguajes de programación, así como para traducir o interpretar lenguajes específicos de dominio (Domain-Specific Languages o DSL). ANTLR permite definir reglas gramaticales y genera automáticamente el código que analiza y comprende el lenguaje de entrada (Parr, 1995).

Características Principales de ANTLR

1. Generación de Analizadores Léxicos y Sintácticos:

ANTLR se destaca por la capacidad de generar analizadores (parsers) que pueden procesar tanto la estructura léxica (identificación de tokens) como la estructura sintáctica (jerarquía y relaciones entre los tokens) de un lenguaje. Esto lo hace ideal para crear compiladores, interpretes, o procesadores de lenguajes formales (Parr, 2013).

2. Gramáticas LL(*):

ANTLR utiliza una variante del análisis sintáctico LL(*), que es un método de análisis top-down. A diferencia de los analizadores LL tradicionales que se limitan a mirar un número fijo de tokens hacia adelante, ANTLR permite un número indefinido de tokens lookahead cuando sea necesario, haciéndolo más flexible y adecuado para gramáticas más complejas (Parr, 2013).

3. Compatibilidad Multilenguaje:

Una de las grandes ventajas de ANTLR es su capacidad para generar analizadores en varios lenguajes de programación, como Java, Python, C++, JavaScript, y más. Esto facilita la integración de ANTLR en diferentes entornos de desarrollo (Parr, 2013).

4. Integración con Árboles Sintácticos Abstractos (AST):

ANTLR permite generar Árboles Sintácticos Abstractos (AST) de manera automática, que representan la estructura jerárquica de un código fuente. Estos ASTs pueden ser manipulados y utilizados para análisis semántico, optimizaciones, o generación de código (Parr, 2013).

3.8 Código Dispensable

El concepto de código dispensable se refiere a partes del código que no son necesarias para el funcionamiento óptimo de un programa o que generan una complejidad innecesaria, dificultando el mantenimiento y comprensión del software. Este tipo de código es común en sistemas que han evolucionado a lo largo del tiempo sin una limpieza o refactorización adecuada.

Martin Fowler (1999), uno de los referentes en ingeniería de software, define el código dispensable como todo aquel código que puede ser eliminado sin afectar la funcionalidad del programa. El código dispensable es un problema común en sistemas de software que carecen de mantenimiento adecuado, y su presencia puede llevar a

una disminución en la calidad del código, afectando tanto el rendimiento como la capacidad de evolución del sistema.

3.9 Teoría de la Medición

La teoría de la medición es un campo dentro de la matemática aplicada y la ciencia que trata de cómo cuantificar propiedades y fenómenos mediante el uso de números y escalas. Este campo se ocupa de asignar valores numéricos o categorías a objetos o eventos de manera que reflejen una propiedad o relación particular de esos objetos o eventos. Es fundamental en muchas disciplinas, desde las ciencias naturales y la ingeniería hasta las ciencias sociales y la informática (Krantz et al., 1971).

3.9.1 Conceptos Fundamentales en la Teoría de la Medición

1. **Medición:** La medición es el proceso de asignar números o símbolos a objetos o eventos de acuerdo con reglas específicas para representar magnitudes, cantidades o categorías. Este proceso es esencial para analizar fenómenos de forma objetiva y reproducible (Stevens, 1946).
2. **Escalas de Medición:** Uno de los pilares fundamentales de la teoría de la medición son las escalas de medición, que determinan cómo se asignan los números y qué operaciones matemáticas tienen sentido en esos números. Según Stanley Smith Stevens (1946), hay cuatro tipos principales de escalas:
 - **Escala Nominal:** Utilizada para clasificar o categorizar sin establecer un orden entre las categorías. Un ejemplo sería asignar números a tipos de sangre (A, B, O). En esta escala, las operaciones matemáticas, como sumas o restas, no tienen sentido.
 - **Escala Ordinal:** Ordena los elementos en función de una característica, pero no mide la distancia entre ellos. Las operaciones matemáticas son limitadas, pero se puede establecer un orden (Stevens, 1946).
 - **Escala de Intervalo:** Además de ordenar los elementos, las diferencias entre los valores son significativas y constantes. Un ejemplo es la temperatura en grados Celsius. Sin embargo, no hay un cero absoluto, por lo que no se pueden realizar comparaciones de proporciones (Stevens, 1946).
 - **Escala de Razón:** Esta escala tiene todas las propiedades de la escala de intervalo, pero incluye un cero absoluto, lo que permite hacer comparaciones de proporciones. Ejemplos incluyen medidas de longitud, peso o tiempo (Stevens, 1946).
3. **Error de Medición:** El error de medición es una parte crucial de la teoría de la medición. Ninguna medición es perfecta, por lo que siempre hay una discrepancia entre el valor medido y el valor real. El error puede ser de dos tipos:
 - **Error sistemático:** Error constante o predecible en todas las mediciones, como un instrumento que siempre mide 1 gramo más (Krantz et al., 1971).
 - **Error aleatorio:** Varía de manera impredecible en cada medición, como pequeñas fluctuaciones debidas a las condiciones ambientales (Stevens, 1946).
4. **Validez y Fiabilidad en la Medición:**
 - **Validez:** Se refiere a si una medición mide realmente lo que pretende medir. Un instrumento de medición debe ser válido para ofrecer resultados significativos. (Krantz et al., 1971).

- **Fiabilidad:** La fiabilidad es la consistencia de una medición a lo largo del tiempo o entre diferentes observadores. Un instrumento es fiable si produce los mismos resultados bajo las mismas condiciones repetidas veces (Stevens, 1946).
- 5. **Instrumentos de Medición:** La teoría de la medición también trata sobre los instrumentos de medición que se utilizan para realizar las mediciones. Estos pueden ser físicos, como una balanza, o abstractos, como un cuestionario en ciencias sociales. En ambos casos, es importante que los instrumentos sean válidos y fiables (Krantz et al., 1971).
- 6. **Representación de la Medición:** En la teoría de la medición, las relaciones observadas entre objetos o eventos se representan numéricamente. (Krantz et al., 1971).
- 7. **Escalabilidad:** Se refiere a la capacidad de un sistema de medición para medir en diferentes rangos de valores sin perder precisión o exactitud. Esto es crucial en campos como la física o la ingeniería, donde las mediciones deben ser consistentes en un amplio rango de valores (Stevens, 1946).

3.10 Variables de Objeto

Las variables de objeto, también conocidas como variables de instancia, son atributos que pertenecen a cada instancia de una clase en la Programación Orientada a Objetos (POO). Estas variables almacenan el estado de un objeto en particular y permiten que cada objeto creado a partir de una clase mantenga sus propios valores independientes de los demás objetos (Booch, 1991).

Una característica clave de las variables de objeto es que cada instancia de una clase posee su propia copia de estas variables, lo que significa que pueden tomar diferentes valores en diferentes objetos, aunque todos pertenezcan a la misma clase. Esto proporciona a cada objeto su propia identidad y comportamiento en función de los datos almacenados en sus variables de objeto (Meyer, 1988).

Además, el acceso a estas variables está restringido al contexto del objeto en cuestión. Es decir, estas variables solo pueden ser manipuladas por los métodos del objeto que las contiene, promoviendo la encapsulación, uno de los principios fundamentales de la POO (Larman, 2004).

3.11 Paquetes (Packages)

Los paquetes son una unidad de organización fundamental en la Programación Orientada a Objetos, que agrupan clases e interfaces relacionadas en módulos coherentes y estructurados. Su propósito es proporcionar una jerarquía lógica para el código, facilitando su modularidad y reutilización, así como evitando conflictos de nombres entre clases con la misma denominación (Gamma et al., 1994). Una de las principales virtudes de los paquetes es que permiten controlar el acceso y la visibilidad de las clases y sus miembros (atributos y métodos). Esto mejora el encapsulamiento, ya que los programadores pueden especificar qué elementos son accesibles fuera del paquete y cuáles permanecen ocultos, restringiendo así el acceso directo a las implementaciones internas (Meyer, 1997).

Además, los paquetes son importantes para gestionar grandes proyectos de software. Al dividir el código en módulos bien definidos, no solo se facilita su mantenimiento y evolución, sino que también se promueve la reutilización de código a lo largo de diferentes partes de una aplicación o incluso en diferentes proyectos (Fowler, 2004).

CAPÍTULO 4 MATERIALES Y MÉTODO DE SOLUCIÓN

Los objetivos de lograr sistemas y unidades de programa mantenibles y reutilizables se alcanzan mediante la adecuada gestión de las relaciones entre clases, un nivel óptimo de granularidad, y un encapsulamiento efectivo. Además, es necesario cumplir con los principios fundamentales de modularización, los cuales incluyen la Descomposición, Composición, Legibilidad, Continuidad y Protección modular. Estos principios no solo facilitan la evolución del software, sino que aseguran que las unidades de código puedan ser comprendidas y modificadas con menor esfuerzo.

Asimismo, los principios de diseño, tanto modular como orientado a objetos, se requieren para garantizar la escalabilidad y sostenibilidad del software a largo plazo. Estos principios incluyen el uso de Unidades Lingüísticas bien definidas, el empleo de Pocas Interfaces (promoviendo baja cohesión), Pequeñas Interfaces (acoplamiento débil), Interfaces Explícitas, Ocultamiento de Datos y principios clave de la programación orientada a objetos como el Principio de Abierto-Cerrado, Inversión de Dependencias, Sustitución de Liskov, Única Responsabilidad, No-Repetición y Separación de Interfaces. Todos estos conceptos refuerzan la creación de sistemas flexibles y robustos frente a los cambios y la expansión funcional.

La solución propuesta para abordar el problema identificado consiste en el diseño de un método automatizado de refactorización a gran escala. Este método permitirá la detección, evaluación y corrección de código innecesario presente en los activos de software escritos en Java, eliminando así las anomalías descritas en el planteamiento del problema.

El sistema está basado en un analizador estático avanzado, con acciones semánticas que permite identificar entidades de software no utilizadas y recopilar la información necesaria para cuantificar el nivel de deuda técnica generada por este código innecesario. El sistema no solo detecta el código no utilizado, sino que también permite la regeneración del código, eliminando o reduciendo significativamente este tipo de deuda técnica.

El proceso será iniciado por el usuario introduciendo la ubicación de los archivos Java, donde podrá seleccionar una o varias unidades de programa escritas en Java que requieran refactorización. Al finalizar la generación del código refactorizado, el sistema medirá el nivel de deuda técnica prevaleciente para evaluar la mejora lograda, asegurando además que la funcionalidad original del código sea completamente preservada tras el proceso de refactorización. Este aspecto es importante, ya que el comportamiento funcional del código original debe mantenerse intacto después de la refactorización.

4.1 Sistema de Medición de Entropía de Software

Como primer paso, es necesario evaluar el grado de entropía presente en sistemas de software legados derivado del código dispensable. Este tipo de código se refiere a arquitecturas de clases que contienen atributos, variables de referencia, métodos y clases que no contribuyen a los objetivos finales del sistema, incluyendo la presencia de clases y métodos nulos, presentes en los sistemas de software legados. El método contempla dos fases: detección y valoración.

1) Por medio de un analizador estático con capacidades semánticas avanzadas se identifican los diferentes factores de código dispensable y se recaba la información necesaria y suficiente para cuantificar el nivel de entropía generada por dicho código.

2) La entropía es cuantificada para efectos de priorizar las acciones semánticas de refactorización según su impacto potencial en la calidad del software.

En la Figura 1 se muestra el diagrama de caso de uso del Sistema de Evaluación de Entropía.

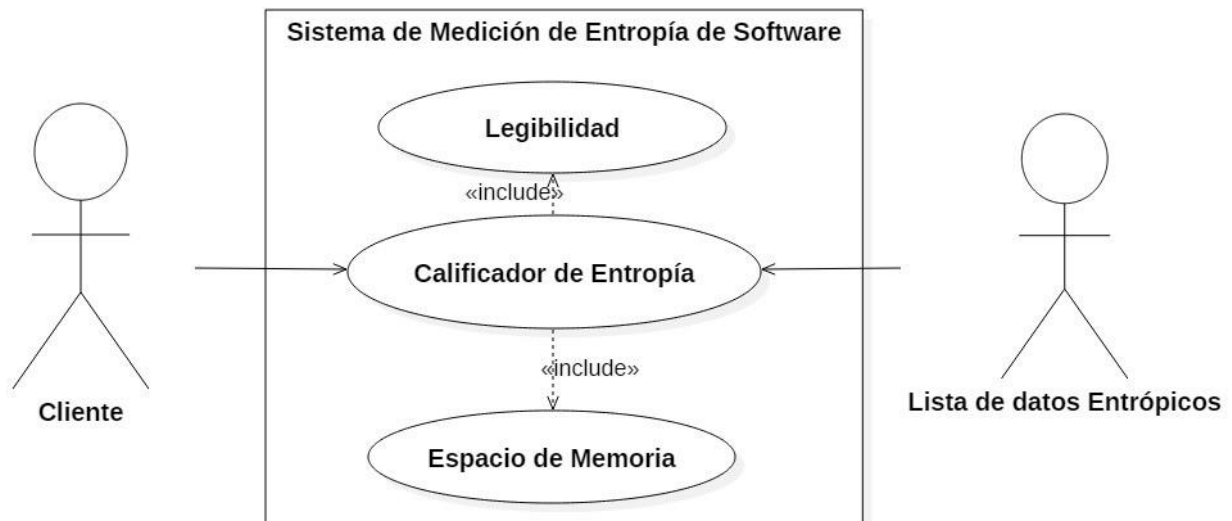


Figura 1 Diagrama UML

Tabla 2

Descripción del Caso de Uso “Sistema de Medición de Entropía de Software”

Nombre:	D.C.U: 01 Sistema de Medición de Entropía de Software
Descripción	El sistema permite realizar la medición de entropía generada por código dispensable con el fin de cuantificar las declaraciones de métodos, funciones, variables, clases, llamadas a métodos, valores predeterminados, código comentado, los cuales han sido declarados en el sistema, pero que no tiene comportamiento alguno.
Actores:	Usuario físico o lógico
Precondiciones:	1.- El código a medir deberá estar escrito en lenguaje JAVA en su versión 8.5. 2.- Para poder llevar a cabo la medición, el código debe respetar el paradigma de programación orientada a objetos. 3. El código debe estar limpio de errores de sintaxis. 4.- El código debe contener algún tipo de código dispensable, ya que si no existiera se entendería que el código no presenta problemas en cuestión a código dispensable.
Flujo de Eventos:	1.- Seleccionar los archivos con código escrito en lenguaje java de un sistema a medir. 2.- Se llevará a cabo la medición de entropía del Código.

Excepciones:	1.- Archivo No Encontrado: Se intenta seleccionar un archivo que no existe o no está accesible en el momento de la ejecución. 2.- Formato de Archivo Incorrecto: El archivo seleccionado no es un archivo java válido o está corrompido. 3.- Fallo en Análisis: Errores durante el análisis debido a incompatibilidades o errores en el código fuente que impiden su procesamiento correcto. 4.-Código Fuente Sin Elementos Dispensables: El análisis identifica que no hay elementos dispensables en el código fuente, lo que impide la medición. 5.-Interrupción del Proceso: El proceso de medición es interrumpido por el usuario o por fallos del sistema.
Postcondiciones:	1.- Resultados de Entropía Presentados: Al finalizar la medición, el sistema presenta un reporte detallado de la entropía del código. Este reporte incluye la cuantificación de métodos, funciones, variables, clases, llamadas a métodos, valores predeterminados y código comentado que son considerados dispensables. 2.- Notificaciones al Usuario: Si el sistema identifica que no existen elementos dispensables en el código fuente, notifica al usuario que el código está libre de este tipo de entropía, lo que podría indicar un código limpio y eficiente en este aspecto. 3.- Estado del Sistema: El sistema vuelve a su estado inicial listo para realizar una nueva medición, asegurando que no hay cambios persistentes que afecten futuras operaciones. 4.- Resolución de Excepciones: Cualquier excepción que ocurra durante el proceso es manejada adecuadamente y se informa al usuario sobre el problema, incluyendo recomendaciones sobre cómo proceder para resolverlo.

4.1 Métrica de Legibilidad de Software

Para determinar los mejores indicadores de legibilidad del código fuente, llevamos a cabo un experimento detallado. Iniciamos con una muestra de código en Java, seguido de una encuesta para evaluar las percepciones de los participantes sobre la legibilidad de este código. Posteriormente, aplicamos un análisis de correlación regresión para explorar las relaciones entre los indicadores propuestos y la legibilidad. Este análisis permitió formular una ecuación preliminar con pesos específicos para cada indicador. Finalmente, en la fase de optimización, buscamos identificar los indicadores más eficaces para evaluar la legibilidad del código, minimizando el número de éstos y manteniendo un nivel de explicación adecuado. A continuación, se detallan los indicadores seleccionados para esta métrica:

- 1) Número de Comentarios (numOfComments): La presencia de comentarios en el código ha sido correlacionada positivamente con la legibilidad. Los comentarios ayudan a explicar la lógica y las decisiones de diseño detrás del código, facilitando la comprensión para otros desarrolladores (Buse & Weimer, 2008).
- 2) Longitud Promedio de las Líneas (AverageLineLength): La longitud de las líneas de código influye directamente en la legibilidad. Códigos con líneas más cortas y concisas tienden a ser más legibles que aquellos con líneas extensas y complejas (Posnett et al., 2011).

3) Complejidad de Control (maxNestedControl): Este indicador mide el nivel máximo de anidamiento en estructuras de control como bucles y condicionales. La literatura sugiere que una menor complejidad en estas estructuras está asociada con una mayor legibilidad (Halstead, 1977).

4) Volumen del Programa (ProgramVolume): Basado en la métrica de Halstead, este indicador refleja el tamaño y la complejidad del programa. Un mayor volumen puede ser indicativo de un código más complejo y potencialmente menos legible (Halstead, 1977).

5) Entropía (Entropy): La entropía se usa para cuantificar la complejidad y el desorden dentro de un código. Un nivel alto de entropía puede indicar que el código es difícil de entender y mantener, afectando negativamente su legibilidad, dada por la siguiente fórmula de la investigación de (Buse & Weimer, 2008).

$$H(X) = \sum_{i=1}^n P(x_i) \log_2 P(x_i)$$

Donde X es un documento y xi es un término en X. P (xi) es la probabilidad de un cambio que ocurre en un documento, n es el número total de documentos, en nuestro caso, archivos fuente.

Se utiliza la función:

$$1/(1+e^{-z})$$

para calcular la puntuación de legibilidad real. Esto normaliza los resultados están en el rango [0, 1].

6) Número de Operadores Lógicos y Aritméticos (numOfLogicalOperators, numOfArithmeticOperators): La densidad de operaciones en un segmento de código puede complicar su comprensión. Se ha encontrado que una menor densidad de estos operadores está asociada con una mejor legibilidad (Buse & Weimer, 2008).

4.2. Indicador de Espacio de Memoria

La Máquina Virtual de Java ofrece herramientas integradas como **MemoryMXBean** para obtener estadísticas de uso de memoria RAM, utilizando comandos como *jmap* y *jcmd*, que proporcionan un histograma de la memoria ocupada. Estos histogramas son esenciales para analizar el consumo de memoria por clase, permitiendo identificar potenciales fugas de memoria o ineficiencias.

El proceso comienza con la creación de un objeto “Histogramer” que al ejecutarse produce un “MemoryHistogram”. Este histograma muestra el número de instancias y el tamaño total de memoria usado por cada clase. Además, permite obtener diferencias entre dos histogramas tomados en distintos tiempos, lo cual es esencial para identificar fugas de memoria o para optimizar el uso de la memoria antes y después de realizar ciertas operaciones críticas en el código como por ejemplo la Refactorización de código.

Es importante considerar que previo a la ejecución de estos comandos, se requiere una recolección completa de basura antes de ejecutarse. Por esta razón, no se recomienda su uso en ambientes de producción. Idealmente, estas mediciones deberían llevarse a cabo en un entorno de prueba o durante el desarrollo para evitar un impacto negativo en el rendimiento en vivo.

Cada una de las métricas presentadas fue sustentada con base en la teoría de la medición, específicamente en el uso de escalas ordinales. Esto significa que se utilizaron escalas que permiten ordenar los elementos en función de su valor relativo, sin necesidad de especificar distancias exactas entre ellos. Para sustentar el uso de estas métricas, se evaluaron tres diferentes arquitecturas de clases, demostrando que cada una cumple con los requisitos del axioma de orden débil. Este axioma garantiza que las métricas pueden ser interpretadas como un orden relativo de complejidad o calidad, y además cumplen con la propiedad de homomorfismo, que establece que existe una relación consistente entre la estructura de los datos y los valores medidos (Zuse, 1995).

La integración de este enfoque basado en la teoría de la medición añade un sustento teórico sólido para el uso de las métricas de legibilidad y el indicador de espacio de memoria en el proceso de refactorización de código dispensable. Al considerar estas métricas como escalas ordinales, es posible establecer un orden de prioridad para la refactorización de las áreas de código más problemáticas. De esta manera, no solo se identifican los componentes del código que presentan mayor deuda técnica, sino que se establece un marco teórico que asegura que las métricas utilizadas están correctamente estructuradas y cumplen con las propiedades matemáticas necesarias para una evaluación confiable.

Las métricas no solo sirven como indicadores empíricos de la deuda técnica, sino que también se sustentan desde una perspectiva formal, garantizando que sus resultados sean consistentes y comparables entre diferentes arquitecturas de clases. Esto refuerza la justificación para su uso en la refactorización, ya que se asegura que la interpretación de los resultados sigue principios teóricos bien fundamentados. Así, la combinación de estos indicadores permite cuantificar de manera robusta la deuda técnica por código dispensable.

4.3 Proceso de refactorización de Código Dispensable

En la Figura 2 se ilustra el modelo BPMN (Business Process Model and Notation) del proceso general del método de refactorización de código dispensable. A continuación, se describen los pasos que conforman el algoritmo que implementa este método de refactorización.

1. Analizar el proyecto de entrada.
 - 1.1. Obtener todas las clases del proyecto.
 - 1.2. Analizar las clases que pertenecen a los módulos.
 - 1.3. Buscar dentro de las clases si existe código dispensable.
2. Analizar cada una de las clases, para obtener las variables, expresiones booleanas, librerías o módulos que no se utilizan.
3. Almacenar cada uno de los métodos públicos que están siendo utilizados desde fuera del módulo.
4. Generar una nueva arquitectura dependiendo del código dispensable encontrado.
 - 4.1. Cada línea de código que sea considerada dispensable, será eliminada

4.3.1 Precondiciones del Método de Refactorización de Código Dispensable

- El código de la arquitectura a analizar no debe contar con errores léxicos ni sintácticos.
- El código deberá estar escrito en lenguaje JAVA y estar funcionando.
- El código debe presentar algún tipo de deuda técnica presentada en los escenarios.

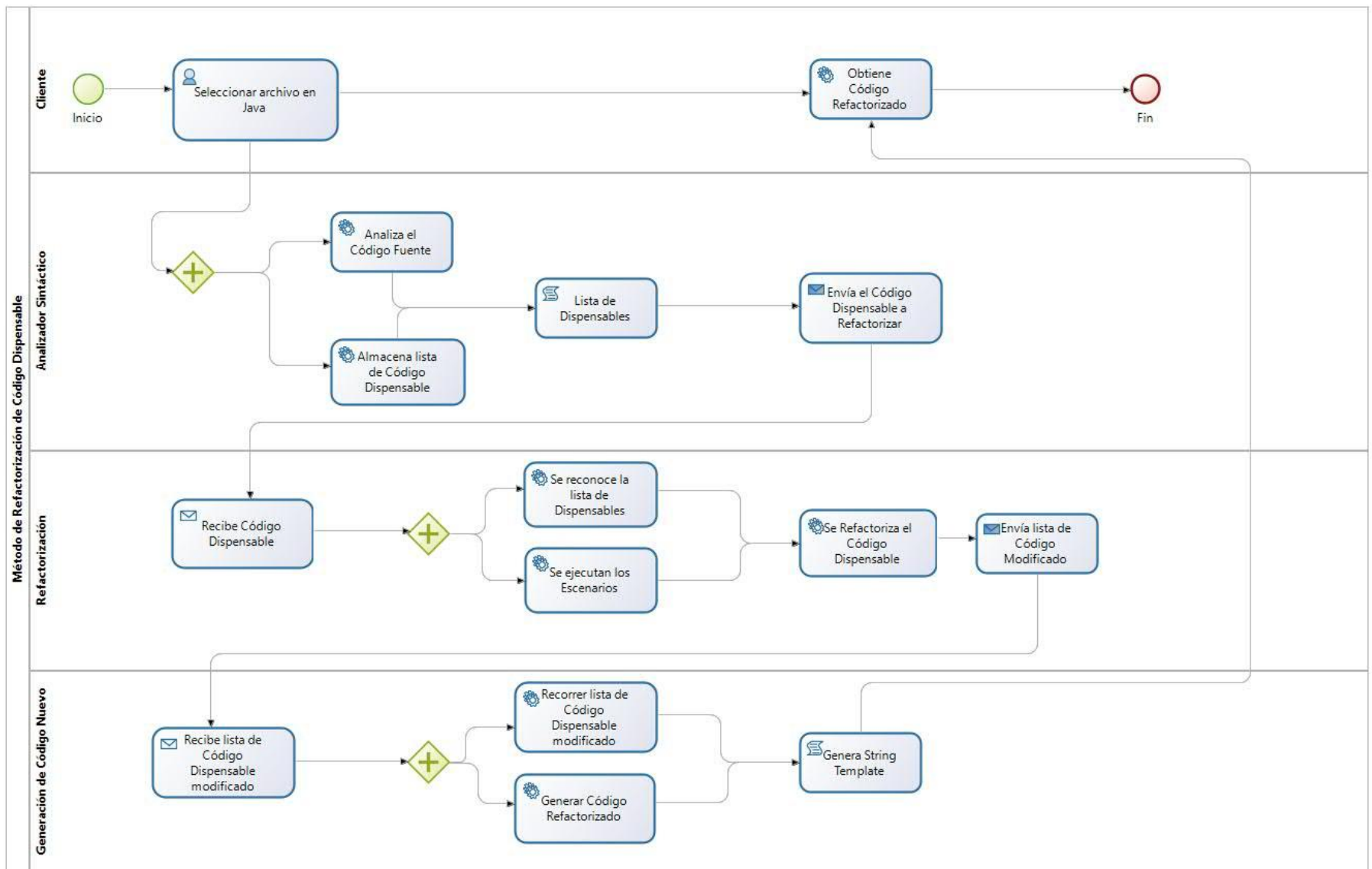


Figura 2 Diagrama BPMN

4.3.2 Descripción del Diagrama BPMN

Este diagrama BPMN (*Business Process Model and Notation*) representa un flujo de trabajo del método de refactorización de código dispensable en Java, el proceso está dividido en cuatro carriles que indican las distintas entidades involucradas, cada una con un rol definido:

1. Cliente:

- El flujo comienza cuando el cliente selecciona un archivo en Java para ser procesado.
- El objetivo final es obtener un código refactorizado, lo que cierra el ciclo del proceso en este carril.

2. Analizador Sintáctico:

- Una vez que el archivo en Java es seleccionado, el analizador sintáctico procede a analizar el código fuente para identificar el Código Dispensable.
- Este código dispensable es almacenado en una lista, y luego esa lista es enviada para el proceso de refactorización.

3. Método de Refactorización del Código Dispensable Refactorización:

- El proceso de refactorización comienza al recibir el código dispensable.
- El sistema reconoce la lista de código dispensable.
- El código refactorizado se envía a la siguiente fase tras ser modificado.

4. Generación de Código Nuevo:

- Esta fase recibe la lista de código dispensable modificado y la recorre.
- Luego, se utiliza la plantilla *String Template* para generar el código refactorizado.
- Finalmente, el código nuevo generado cierra el ciclo.

CAPÍTULO 5 DESARROLLO DEL SISTEMA

Durante el proceso de desarrollo, se utilizaron tanto diagramas del estándar UML como diagramas C4 para modelar y visualizar la arquitectura del sistema. Esta combinación facilitó la identificación y refactorización del código dispensable.

Diagrama Caso de Uso Refactorizar Código Dispensable

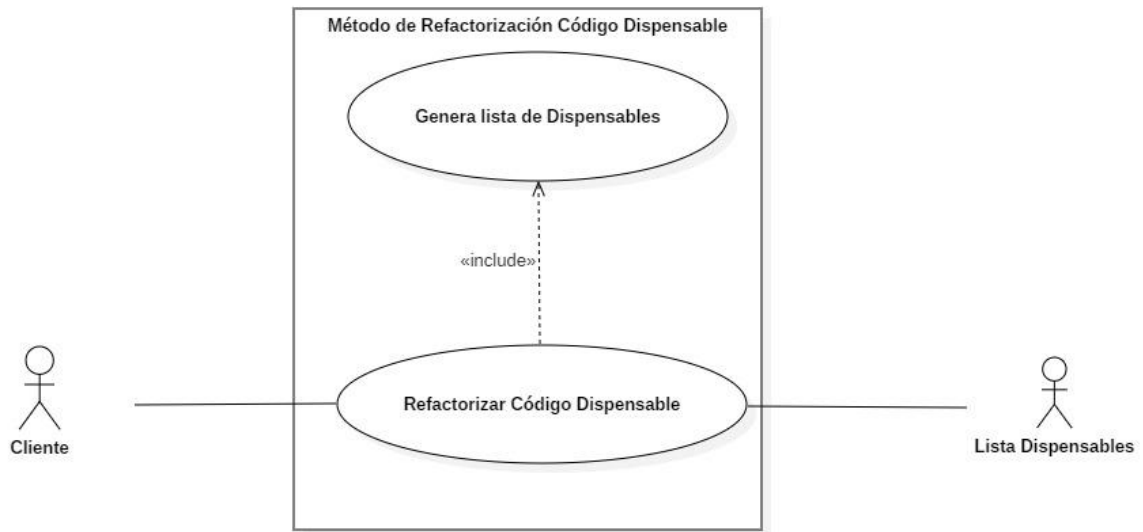


Figura 3 Diagrama C.U. Refactorizar Código Dispensable

Tabla 3

Descripción del Caso de Uso "Refactorización de Código Dispensable"

Nombre:	D.C.U: 01 Refactorización de Código Dispensable
Descripción	El sistema permitirá realizar la refactorización de código dispensable con el fin de corregir las declaraciones de métodos, funciones, variables, clases, llamadas a métodos, expresiones booleanas, valores predeterminados, código comentado, los cuales han sido declarados en el sistema, pero que no tiene comportamiento alguno.
Actores:	Usuario físico o lógico
Precondiciones:	1.-El código a refactorizar deberá estar escrito en lenguaje JAVA. 2.-Para poder llevar a cabo la refactorización el código debe respetar el paradigma de programación orientada a objetos.

	3.-El código debe contener algún escenario de los presentados en el análisis de escenarios, ya que si no existiera se entendería que el código no presenta problemas en cuestión a código dispensable.
Flujo de Eventos:	1.-Seleccionar los archivos de código escrito en lenguaje java a refactorizar. 2.- Se llevará a cabo un análisis sintáctico con la gramática de java para obtener la lista de clases, métodos, variables, que no se ocupan. 3.- Se llevará a cabo la refactorización del código para eliminar todos los elementos dispensables, planteados en esta tesis.
Excepciones:	1.-Si el archivo de código seleccionado no está en lenguaje Java, el sistema mostrará un mensaje de error y no permitirá continuar con la refactorización. 2.-Si el código no sigue el paradigma de programación orientada a objetos, el sistema mostrará un mensaje de error indicando que el código no cumple con los requisitos necesarios. 3.-Si no se identifican elementos dispensables en el análisis sintáctico, el sistema informará que no se encontraron problemas de código dispensable y detendrá el proceso de refactorización.
Postcondiciones:	1.-El código ha sido refactorizado, eliminando todos los elementos dispensables detectados. 2.-El sistema generará un informe con los cambios realizados, detallando las clases, métodos y variables eliminadas o modificadas durante el proceso de refactorización. 3.-El código resultante deberá ser funcional y mantener el comportamiento esperado, sin introducir nuevos errores o anomalías.

Los actores son los siguientes:

Cliente: Representa una entidad física (persona) o una entidad lógica (sistema externo) que usará este método de refactorización. Existen 2 funciones que puede ejecutar el usuario, las cuales son:

1. Seleccionar los archivos que se van a analizar.
2. Solicitar la ejecución del método de refactorización, ejecutando el caso de uso de *Refactorización de Código Dispensable*.

Código Fuente: Será(n) el(los) archivo(s) Java original(es) que el Cliente seleccionará, en el caso de uso “*Seleccionar Archivos*”. Serán los archivos donde se encuentra el código que se desea analizar para aplicar el método de refactorización.

Lista de Dispensables: Representa el subsistema encargado de realizar el análisis sintáctico. Recibe como entrada un proyecto de software escrito en lenguaje Java y regresa como salida un modelo de objetos con la información sobre los archivos que contienen el código fuente, para la refactorización.

5.1 Descripción de Escenarios

A continuación, se describe el análisis del Diagrama General de caso de uso. En esta descripción se muestra el nombre, actores, descripción, precondiciones, postcondiciones, escenario principal de éxito, escenarios alternos y los posibles escenarios de fracaso.

Tabla 4
Seleccionar Archivos

Nombre:	Seleccionar Archivos.
Actores:	Usuario, Código Fuente.
Descripción:	El usuario deberá de seleccionar los archivos del sistema legado de software, del cual se requiere la refactorización, para capturar los metadatos del código original y realizar una copia de seguridad
Precondiciones:	1. Los archivos a analizar deberán estar en un mismo directorio. 2. El código original deberá estar libre de errores de sintaxis. 3. El código original deberá compilar correctamente.
Postcondiciones:	Como resultado se obtendrá una lista de clases de objeto de los archivos con extensión Java seleccionados, los cuales se someterán al análisis sintáctico.
Escenario principal de éxito:	1. Se mostrará el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario seleccionará el directorio o el conjunto de archivos Java (.java) de un sistema de software legado que requiere la refactorización. 3. El sistema verificará que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema extraerá la información necesaria de los archivos, tanto para la aplicación del método de refactorización. Esto es, la información sobre la interacción entre métodos de clases u objeto, tales como “el nombre del paquete” o “espacio de nombres”, “nombres de clases”, “nombre de atributos”, “nombre de métodos”, “parámetros de métodos”, etc. 5. El sistema almacenará la información extraída en una estructura de datos no persistente. Se usará un modelo de objetos. 6. El sistema conservará la información para los procesos de refactorización. 7. El sistema realizará una copia de seguridad del código fuente original en directorios replicados.
Escenario alternativo (1)	- Se mostrará el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona de diferentes directorios los archivos Java (.java) a extraer de un sistema de software legado que se requiere la refactorización o el cálculo de métricas. 3. Se ejecutan los puntos 3-7 del escenario principal de éxito.

Escenario de fracaso (1)	1. El usuario no selecciona ningún proyecto durante 5 minutos, entonces el proceso será abortado y tendrá que reiniciarse las veces que sean necesarias.
Escenario de fracaso (2)	1. Alguno(s) de lo(s) archivo(s) con el código fuente están disponibles sólo para su lectura. En este caso el sistema debe enviar un mensaje para que el usuario modifique los derechos de acceso y reiniciar el caso de uso.
Escenario de fracaso (3)	1. Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona un directorio. 3. El sistema verifica que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema no encuentra ningún archivo Java (.java) en el directorio seleccionado. El sistema alertará al usuario que no se encontraron archivos Java (.java). 6. En este caso el sistema debe enviar un mensaje al usuario indicándolo que no encontró archivos Java (.java).
Escenario de fracaso (4)	1. Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona un directorio. 3. El sistema verifica que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema no encuentra ningún archivo Java (.java) en el directorio seleccionado. 5. El sistema realizará la refactorización con los archivos encontrados, pero la funcionalidad original no será conservada. 6. Localizar la funcionalidad de los archivos faltantes. 7. Incorporar los archivos faltantes. 8. Reiniciar la refactorización.
Escenario de fracaso (1)	1. El usuario no selecciona ningún proyecto durante 5 minutos, entonces el proceso será abortado y tendrá que reiniciarse las veces que sean necesarias.
Escenario de fracaso (2)	1. El sistema concluye su funcionamiento sin calcular la métrica.

Tabla 5
Descripción Refactorización de código

Nombre:	Refactorización de código
Actores:	Usuario, Sistema, Código Fuente, Analizador Sintáctico, Modelo de Objetos
Descripción:	El sistema aplicará el método de refactorización, con el fin de eliminar el código dispensable
Precondiciones:	1. El código legado existente a refactorizar deberá estar libre de defectos y/o errores, es decir, debe compilar y funcionar correctamente.
Postcondiciones:	1. El código refactorizado se encuentra libre de código desagradable. 2. La arquitectura de software del código original es mejorada al remover el código dispensable. 3. Se ve disminuida la deuda técnica. 4. La funcionalidad del código refactorizado no cambia con respecto al código original. 5. Se obtendrán los archivos con el código refactorizado en directorios replicados de salida de manera correspondiente.
Escenario principal de éxito	1. El usuario seleccionará la ruta del proyecto de software de entrada a refactorizar. 2. El sistema ejecuta el subsistema “Analizador Sintáctico” para realizar el análisis de código fuente. 3. El sistema recibe el modelo de objetos obtenida por el subsistema “Analizador Sintáctico”, con tenida en una estructura contenedora de datos. 4. El sistema analizará la información contenida en la estructura de datos para identificar el código dispensable 5. El sistema realiza la eliminación de código dispensable

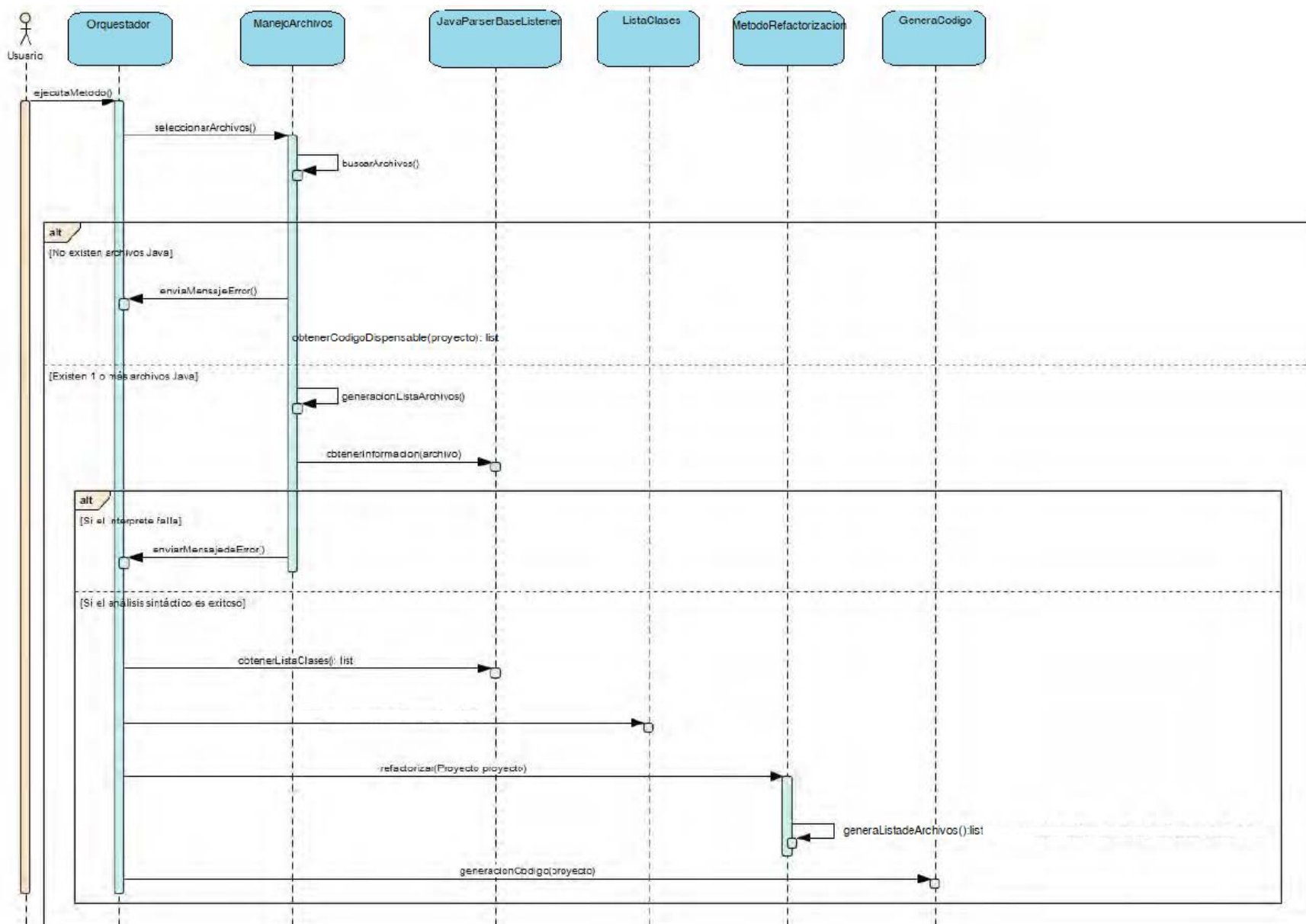


Figura 4 Diagrama de Secuencia del Sistema de Refactorización de Código Dispensable

El diagrama de secuencia de la Figura 4 describe el proceso de refactorización de la siguiente manera:

1. El cliente inicia el proceso de refactorización.
2. El cliente selecciona los archivos en Java a refactorizar.
3. Si el cliente selecciona una carpeta, el sistema busca todos los archivos Java dentro de la misma.
4. Si no se encuentran archivos Java, el sistema muestra un mensaje de error.
5. Si se encuentran archivos Java, el sistema genera una lista de clases.
6. El sistema obtiene la información relevante de las clases abstractas y las clases base para la refactorización.
7. Se calcula la métrica de legibilidad y de espacio de memoria para determinar si la refactorización es viable.
8. El sistema aplica la refactorización, eliminando el código dispensable.
9. El sistema genera el código refactorizado equivalente al original.
10. Se recalculan las métricas para evaluar las mejoras del código.
11. El sistema almacena el proyecto refactorizado en una carpeta denominada <Código_Refactorizado>.

En la Figura 4 se muestra el diagrama C1 “Arquitectura general del sistema”

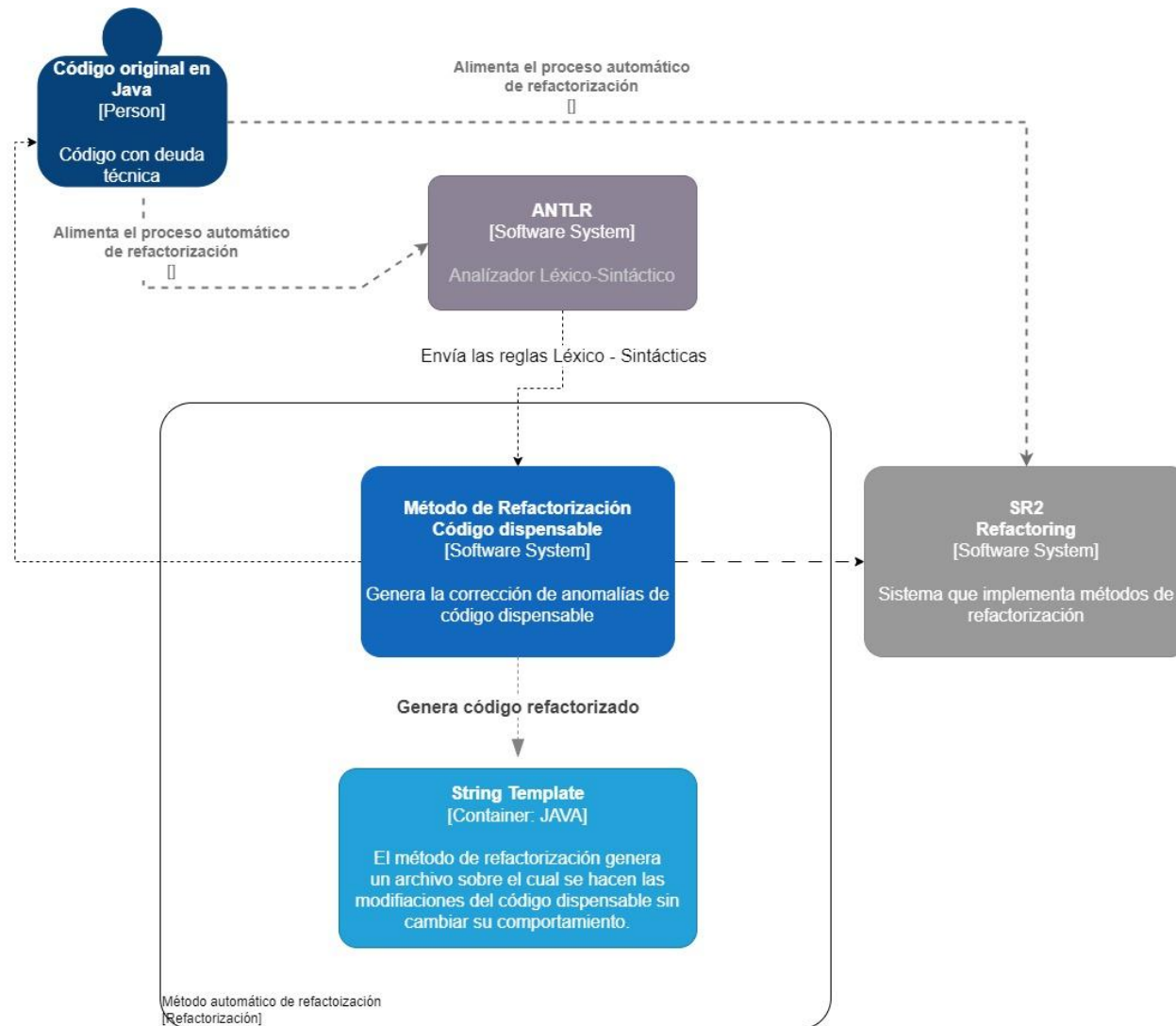


Figura 5 Diagrama C1 Arquitectura general del sistema

En la Figura 5, se describe el proceso automático de refactorización de código dispensable en Java. El flujo es el siguiente:

1. **Código original en Java [Person]**: Es el código fuente que contiene deuda técnica y que alimenta el proceso automático de refactorización.
2. **ANTLR [Software System]**: Un analizador léxico-sintáctico que recibe las reglas léxicas y sintácticas, y las envía al método de refactorización.
3. **Método de Refactorización de Código Dispensable [Software System]**: Es el sistema que implementa el proceso de corrección de anomalías de código dispensable.
4. **String Template [Container: Java]**: Este componente genera un archivo que contiene las modificaciones de refactorización sin cambiar el comportamiento del código.
5. **SR2 Refactoring [Software System]**: Es el sistema que implementa métodos de refactorización adicionales, conectado con el proceso general.

En la Figura 6 muestra el diagrama C2 “Contenedores”

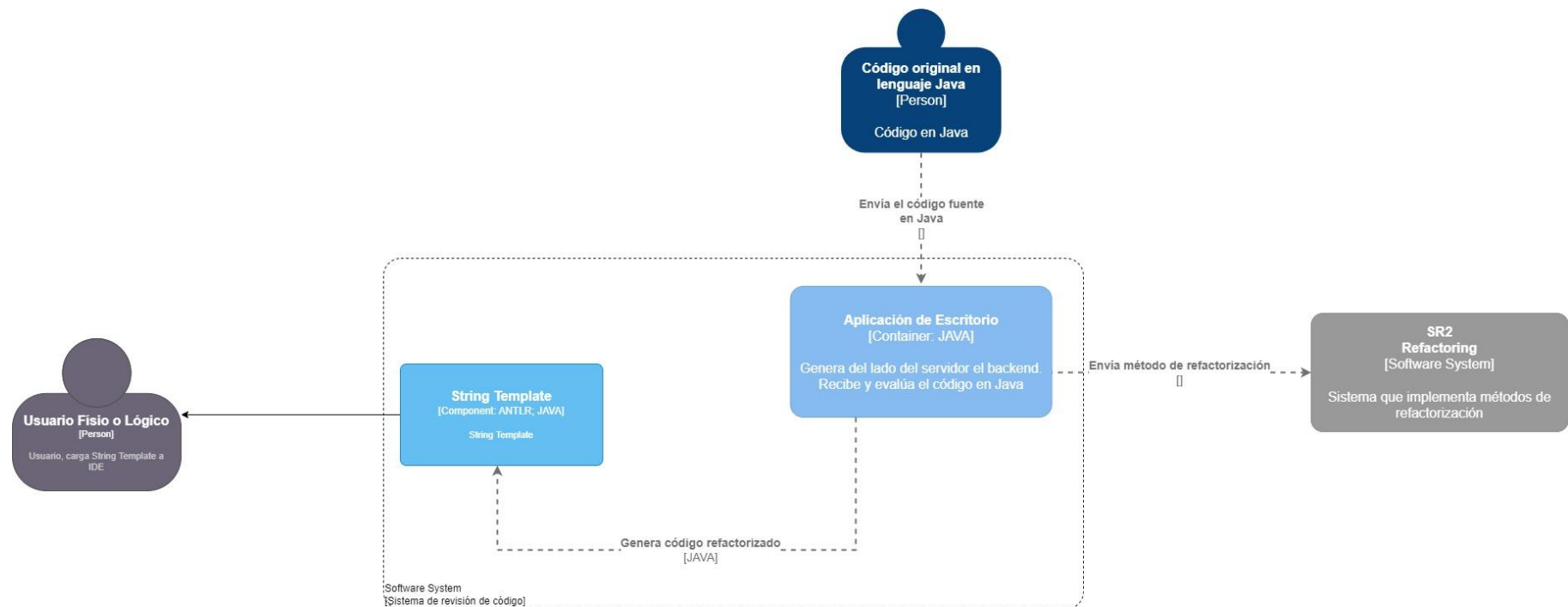


Figura 6 Diagrama C2 Contenedores

En la Figura5 se describe cómo un usuario físico o lógico interactúa con el sistema de refactorización:

1. **Usuario físico o lógico [Person]:** El usuario utiliza el sistema para cargar plantillas en un archivo String Template.
2. **String Template [Component: ANTLR, JAVA]:** Este componente actúa como intermediario entre el código fuente en Java y la aplicación que evalúa dicho código.
3. **Aplicación de Escritorio [Container: JAVA]:** Esta aplicación genera el backend y evalúa el código fuente en Java.
4. **SR2 Refactoring [Software System]:** Aquí se recibe y se aplica el método de refactorización.

El modelo muestra el flujo desde que el usuario físico o lógico introduce datos hasta que el sistema SR2 refactoriza el código.

En la Figura 7 se muestra el diagrama C3 de “Componentes”

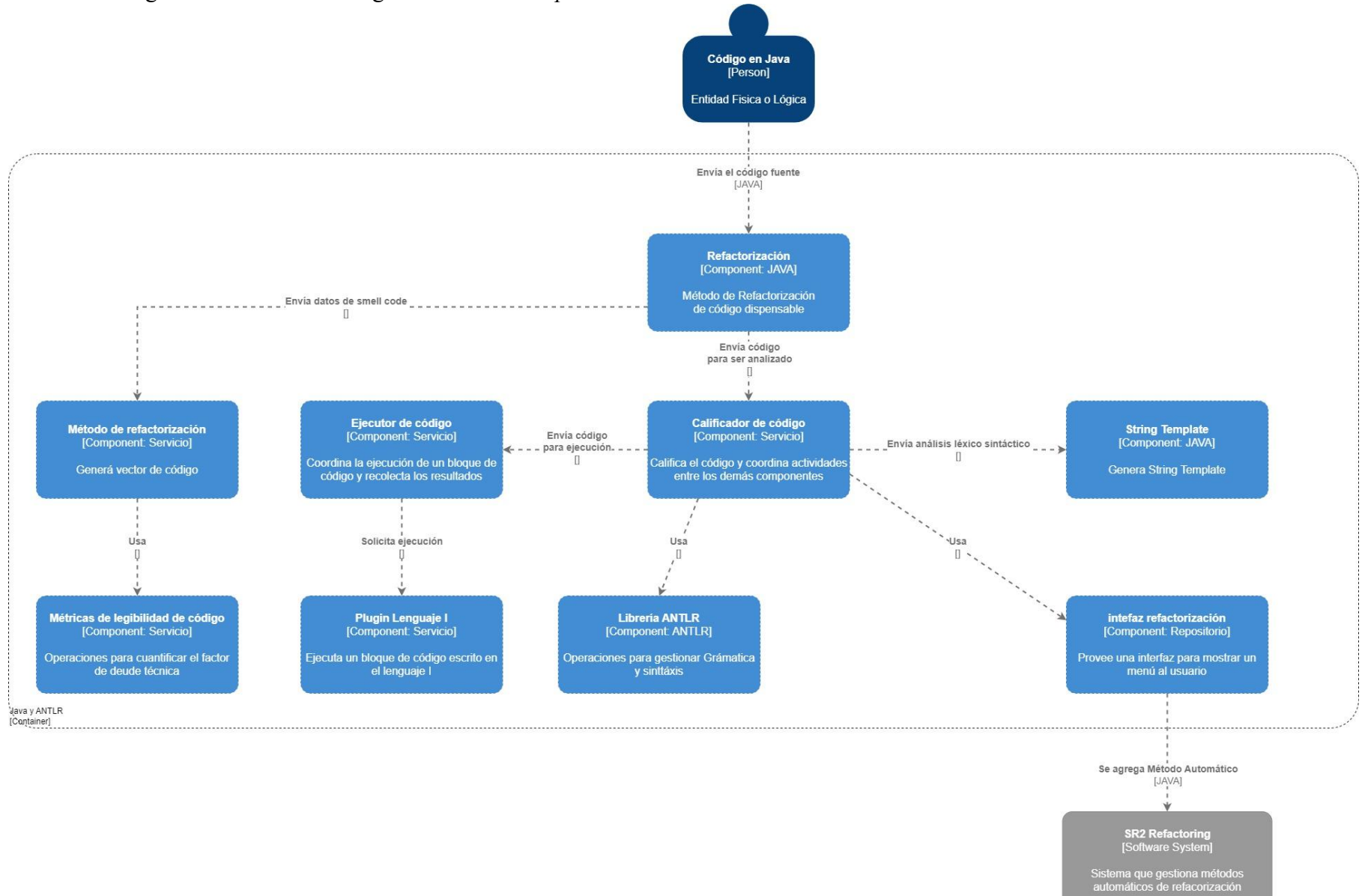


Figura 7 Diagrama C3 Componentes

En este último modelo se detallan más componentes dentro del sistema de refactorización. El flujo es el siguiente:

1. **Código en Java [Person]:** El código fuente se envía al componente de refactorización.
2. **Refactorización [Component: JAVA]:** Este componente toma el código dispensable y lo elimina.
3. **Método de Refactorización [Component: Servicio]:** Genera el vector de código para el proceso de refactorización.
4. **Calificador de código [Component: Servicio]:** Coordina actividades entre los componentes y califica el código.
5. **Librería ANTLR [Component: ANTLR]:** Provee las operaciones para gestionar la gramática y sintaxis del código.
6. **String Template [Component: JAVA]:** Genera las plantillas de refactorización.
7. **Métricas de legibilidad de código [Component: Servicio]:** Permite medir el espacio de memoria y legibilidad.
8. **Ejecutor de código [Component: Servicio]:** Ejecuta bloques de código y coordina los resultados.

5.3 Diagrama de Clases del Método de Refactorización de Código Dispensable

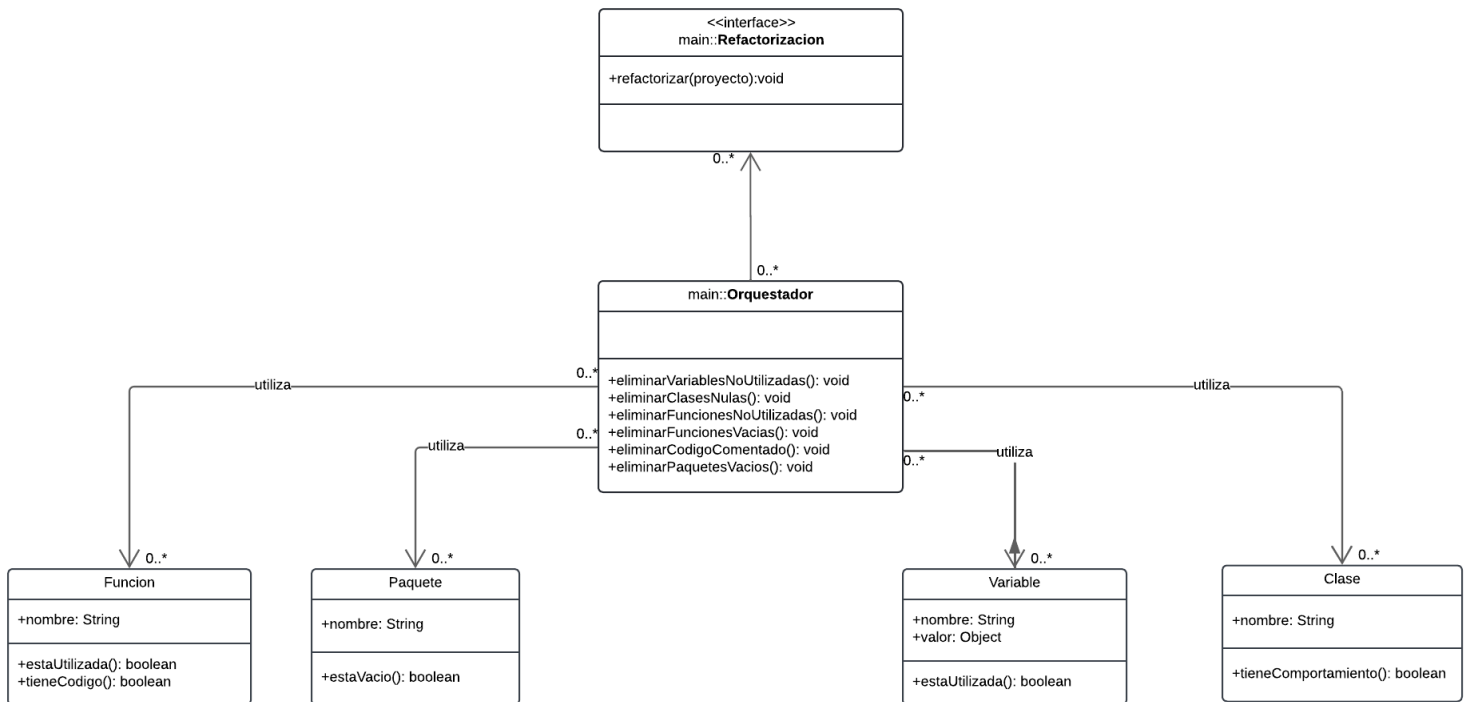


Figura 8 Diagrama de Clases Método Refactorización Código Dispensable

La arquitectura se compone principalmente de una interfaz, una clase principal denominada Orquestador, y varias clases auxiliares que representan los elementos del código que se deben evaluar y refactorizar. A continuación, se describe cada uno de los componentes del sistema y su interacción.

1. Interfaz Refactorizacion

La interfaz Refactorizacion actúa como un contrato que define el método refactorizar(proyecto). Este método tiene la responsabilidad de ejecutar la refactorización sobre un proyecto de software. La implementación de este método varía según la clase que lo implemente, permitiendo flexibilidad para adaptarse a diferentes tipos de proyectos.

2. Clase Orquestador

La clase Orquestador constituye el núcleo de la arquitectura, ya que se encarga de coordinar las diversas acciones de refactorización. Implementa métodos específicos para eliminar elementos innecesarios en el código fuente:

- eliminarVariablesNoUtilizadas()
- eliminarClasesNulas()
- eliminarFuncionesNoUtilizadas()

- `eliminarFuncionesVacias()`
- `eliminarCodigoComentado()`
- `eliminarPaquetesVacios()`

Estos métodos encapsulan la lógica necesaria para identificar y eliminar estos componentes, contribuyendo así a la mejora del código. La clase Orquestador opera como un gestor central que utiliza diversas instancias de otras clases especializadas para cumplir con su tarea.

3. Clases Especializadas

Las clases especializadas representan los diferentes elementos del código que pueden ser objeto de refactorización. Cada una de estas clases cuenta con atributos y métodos específicos que permiten evaluar si el elemento es redundante o innecesario:

- **Clase Funcion:** Representa una función en el código. Contiene atributos como nombre y métodos como `estaUtilizada()` y `tieneCodigo()`, que determinan si la función está siendo utilizada o si contiene código ejecutable.
- **Clase Variable:** Representa una variable en el código. Contiene atributos como nombre y valor, así como el método `estaUtilizada()`, que verifica si la variable está en uso dentro del proyecto.
- **Clase Clase:** Representa una clase del código fuente. A través del método `tieneComportamiento()`, se verifica si la clase contiene algún comportamiento o método que justifique su permanencia en el código.
- **Clase Paquete:** Representa un paquete de clases. Con el método `estaVacio()`, se puede identificar si el paquete no contiene clases útiles y, por tanto, es candidato a ser eliminado.

4. Relaciones e Interacciones

La clase Orquestador utiliza (relación utiliza) instancias de las clases Funcion, Variable, Clase y Paquete para llevar a cabo su tarea de refactorización. La multiplicidad 0..* en las relaciones indica que el Orquestador puede interactuar con múltiples instancias de estas clases, lo que le permite escanear y procesar una gran cantidad de elementos en el código.

Esta estructura modular y orientada a objetos favorece la escalabilidad del sistema. A medida que crece la base de código, el Orquestador puede manejar eficientemente la refactorización de proyectos más grandes y complejos, permitiendo la eliminación de elementos innecesarios.

CAPÍTULO 6 PRUEBAS

6.1 Análisis de Escenarios

En el desarrollo de software, el código dispensable es aquel que no aporta valor al funcionamiento del sistema y, por tanto, debe ser eliminado para mejorar la eficiencia y la legibilidad del código. Un escenario, en este contexto, se define como una situación o conjunto de circunstancias específicas en las que se identifica la presencia de elementos de código que no son necesarios. Estos escenarios han sido definidos en esta tesis con el fin de proporcionar una estructura clara y organizada para identificar y clasificar los distintos tipos de código dispensable que se encuentran comúnmente durante el desarrollo y mantenimiento de software. Al abordar cada escenario, es posible aplicar estrategias de refactorización que optimicen el código, reduciendo su complejidad y facilitando su comprensión y mantenimiento a lo largo del ciclo de vida del software.

Los escenarios presentados a continuación fueron seleccionados debido a que representan los tipos de código dispensable más comunes y recurrentes en proyectos de software. Estos escenarios son: **Variables no utilizadas, clases nulas, funciones no utilizadas, funciones vacías, código comentado y paquetes vacíos**. La decisión de centrarse en estos seis escenarios responde a que, aunque existen otros tipos de código dispensable, estos son los que tienen una influencia más directa y tangible en la complejidad del software. Su eliminación mejora la legibilidad y el espacio de memoria, reduce la carga cognitiva para los desarrolladores y disminuye el riesgo de errores futuros.

Al limitar el enfoque a estos casos, el método de refactorización se centra en las áreas de mayor impacto, evitando una dispersión hacia escenarios menos comunes, que podrían desviar la atención del objetivo principal: disminuir la deuda técnica por código dispensable.

Para llevar a cabo el análisis léxico y sintáctico de las arquitecturas modulares, se utilizará el analizador ANTLR, junto con la gramática para el lenguaje Java en su versión 8.0.

A continuación, se presentan los diferentes escenarios encontrados:

Escenario 1 Variables no utilizadas: Las variables que se declaran, pero nunca se usan; ocupan espacio en la memoria y hacen que el código sea más difícil de leer y mantener.

Ejemplo:

```
public class Main {  
    public static void main(String[] args) {  
        int x= 0;  
        int y= 0;  
        String a = "";  
        String cadena = "";  
        double c = 0.0;  
        float d = 0.0F;  
        System.out.printf("Hello and welcome!");  
  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```

Figura 9 Variables no utilizadas

Escenario 2 Clases nulas o aisladas: Clases declaradas en el código, que están vacías y no tienen comportamiento alguno.

Ejemplo:

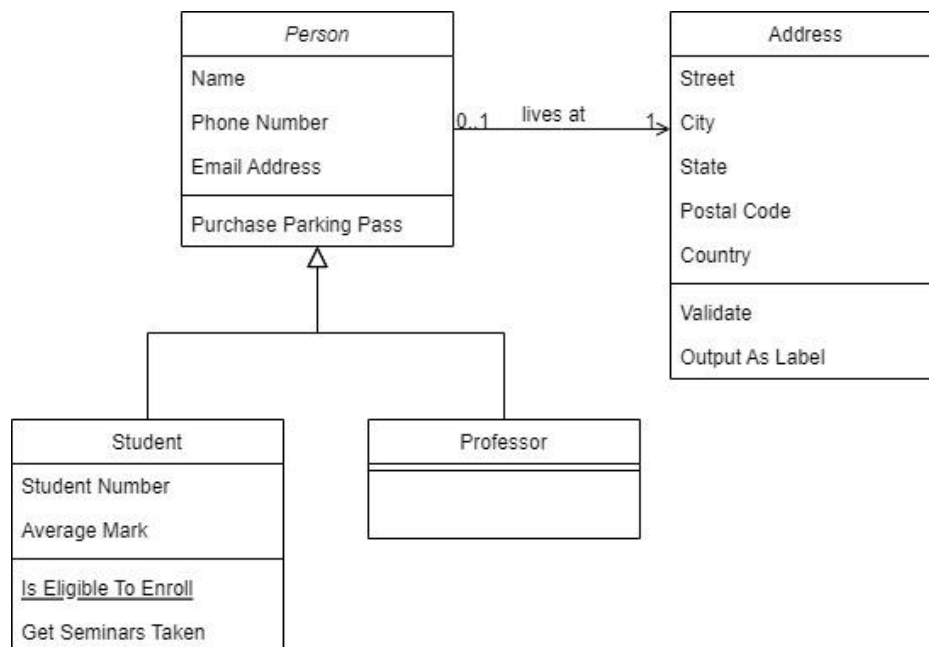


Figura 10 Diagrama de Clases, con Clase no utilizada

Escenario 3 Funciones no utilizadas: Si hay funciones que se han definido, pero no se utilizan en ninguna parte del programa, es mejor eliminarlas para evitar la confusión y el desorden.

Ejemplo:

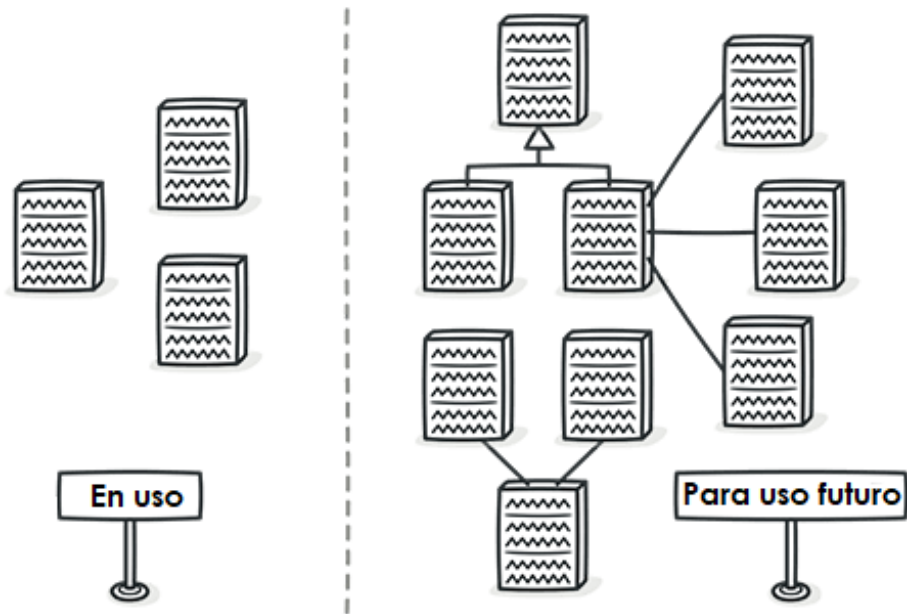


Figura 11 Funciones no Utilizadas

Escenario 4 Funciones vacías: Funciones definidas en el programa que no tienen comportamiento.

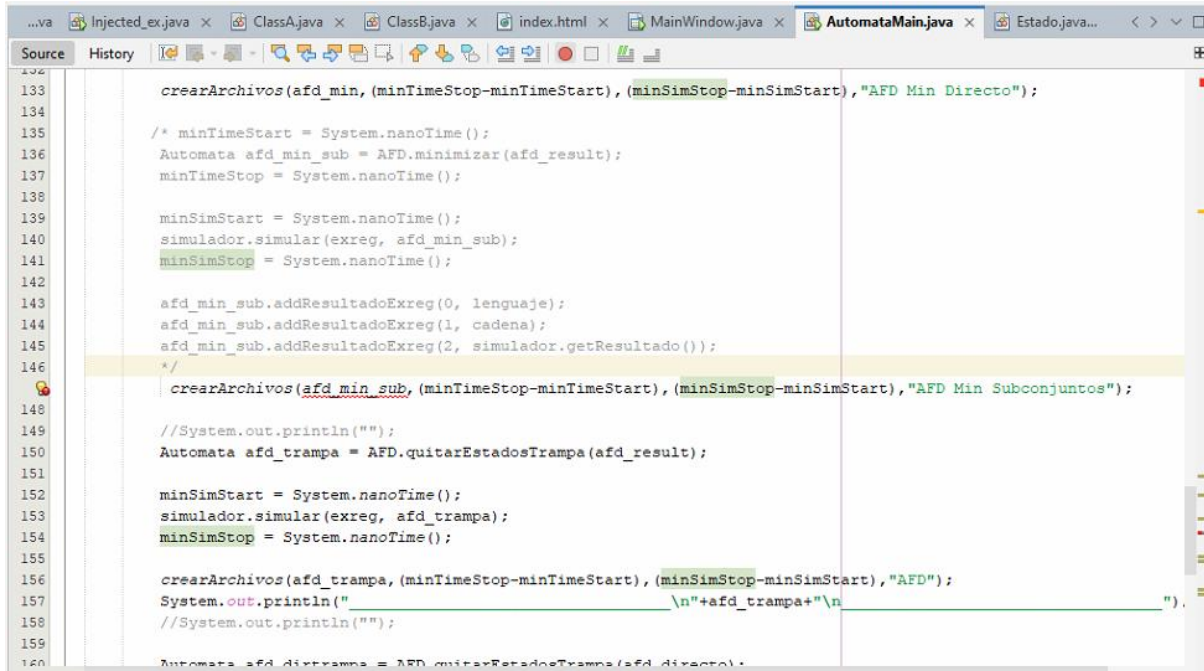
Ejemplo:

```
public class MiClase {  
    // Función vacía  
    public void funcionVacía() {  
        // No hace nada  
    }  
}
```

Figura 12 Función sin comportamiento

Escenario 5 Código comentado: Si existe código que ha sido comentado y ya no es necesario, es mejor eliminarlo. El código comentado puede ser confuso y hacer que el código sea difícil de leer.

Ejemplo:



```
133 crearArchivos(afd_min, (minTimeStop-minTimeStart), (minSimStop-minSimStart), "AFD Min Directo");
134
135 /* minTimeStart = System.nanoTime();
136 Automata afd_min_sub = AFD.minimizar(afd_result);
137 minTimeStop = System.nanoTime();
138
139 minSimStart = System.nanoTime();
140 simulador.simular(exreg, afd_min_sub);
141 minSimStop = System.nanoTime();
142
143 afd_min_sub.addResultadoExreg(0, lenguaje);
144 afd_min_sub.addResultadoExreg(1, cadena);
145 afd_min_sub.addResultadoExreg(2, simulador.getResultado());
146 */
147 crearArchivos(afd_min_sub, (minTimeStop-minTimeStart), (minSimStop-minSimStart), "AFD Min Subconjuntos");
148
149 //System.out.println("");
150 Automata afd_trampa = AFD.quitarEstadosTrampa(afd_result);
151
152 minSimStart = System.nanoTime();
153 simulador.simular(exreg, afd_trampa);
154 minSimStop = System.nanoTime();
155
156 crearArchivos(afd_trampa, (minTimeStop-minTimeStart), (minSimStop-minSimStart), "AFD");
157 System.out.println("\n"+afd_trampa+"\n");
158 //System.out.println("");
159
160 Automata afd_dirtrampa = AFD.quitarEstadosTrampa(afd_directo);
```

Figura 13 Código comentado

Escenario 6 Paquetes nulos o vacíos: son aquellos que no contienen clases, funciones o archivos. Aunque a menudo se crean con la intención de organizar mejor el código, si no se utilizan, pueden generar confusión y desorden en el proyecto.

Ejemplo:



```
Proyecto/
|
|-- src/
|   |-- com/
|   |   |-- miempresa/
|   |   |   |-- controlador/ <-- Paquete vacío o nulo (sin clases)
|   |   |   |-- modelo/
|   |   |   |   |-- Usuario.java
|   |   |   |   |-- Producto.java
|   |   |   |   |-- (Más clases...)
|   |   |   |-- servicio/ <-- Paquete vacío o nulo (sin clases)
|   |   |   |-- utilidades/
|   |   |   |   |-- Validador.java
|   |   |   |   |-- Conversor.java
|   |   |-- otro_paquete/
|   |   |   |-- (Sin clases)
```

Figura 14 Paquetes vacíos

Figura 12 “Paquetes nulos o vacíos”

6.2 Convención de nombres

La Figura 12 muestra la convención de nombres que se utilizará durante la evaluación del método de refactorización de Código Dispensable.

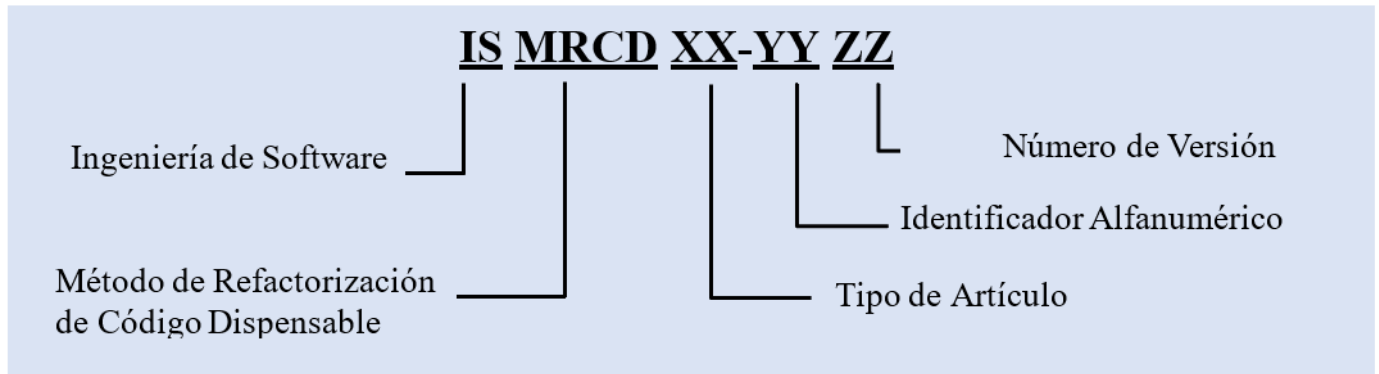


Figura 15 Convención de nombres

Tabla 6
Convención de nombres

Tipo de Artículo	
01	Módulos de programa.
02	Programas de control.
03	Plan de pruebas.
04	Diseño de pruebas.
05	Casos de prueba.
Programas de Control	
ISMRC01 - YYYY	Módulos de programa
Módulos de programa	
ISMRC02 - YYYY	Programas de control, utilerías, ordenadores, entre otros.
Documentación de pruebas	
ISMRC03 - YYYY	Plan de pruebas.
ISMRC04 - YYYY	Especificación de diseño de pruebas.
ISMRC05 - YYYY	Especificación de casos de prueba.

6.3 Plan de pruebas

1.- Plan de prueba: ISMRCD 03 – 01

Plan de pruebas para la validación del correcto funcionamiento del método de refactorización de arquitecturas de marcos orientados a objetos con código dispensable.

2.- Introducción

El método de refactorización de código dispensable tiene la función de analizar cada una de las funciones y clases en una aplicación Java, con el objetivo de identificar y eliminar código innecesario, con el fin de verificar que se mejore la legibilidad y el espacio de memoria.

3.- Artículos de prueba. Los módulos a ser probados se muestran en la Tabla 7.

Tabla 7
Módulos de programa

Sistema	Función	No.
Método de Refactorización de Código Dispensable	Subsistema de evaluación de Código Dispensable	ISMRCDD01 - 01
Métrica de legibilidad y espacio de memoria	Cálculo de métricas de legibilidad y espacio de memoria.	ISMRCDD01 - 02

4.- Procedimientos de control de tareas mostrados en la Tabla 8.

Tabla 8
Control de tareas

Sistema	Función	No.
Programa de aplicación	Localización del paquete al que pertenece la clase.	ISMRCDD02 - 01
Programa de aplicación	Localización de las importaciones de la clase.	ISMRCDD02 - 02
Programa de aplicación	Localización del nombre de la clase.	ISMRCDD02 - 03
Programa de aplicación	Localización de la clase padre de la clase (si existiera).	ISMRCDD02 - 04
Programa de aplicación	Localización de atributos de la clase.	ISMRCDD02 - 05
Programa de aplicación	Localización de métodos de la clase	ISMRCDD02 - 06
Programa de aplicación	Localización de los atributos de funciones de clase, tales como: calificador de alcance, si es abstracto, si es estático, nombre, parámetros, contenido.	ISMRCDD02 - 07

Programa de aplicación	Medición del espacio de memoria	ISMRCDD02 – 09
Programa de aplicación	Método de Refactorización de Código Dispensable	ISMRCDD02 – 10
Programa de aplicación	Llenado de la plantilla ST y generación de código refactorizado.	ISMRCDD02 – 11
Programa de aplicación	Compilación del Código Refactorizado.	ISMRCDD02 – 12
Programa de aplicación	Verificación del funcionamiento del Código Refactorizado.	ISMRCDD02 – 13
Programa de aplicación	Cálculo de la Métrica de Legibilidad	ISMRCDD02 - 08

6.4 Características a ser probadas.

En la Tabla 9 se muestran las características que deben ser probadas.

Tabla 9
Características a probar

Diseño de Prueba No. de especificación	Descripción
ISMRCDD04 – 01	Cálculo de métricas de calidad
ISMRCDD04 – 02	Método de refactorización de Código Dispensable

6.4.1 Características a no ser probadas.

- Los casos de prueba no incluirán todas las posibles construcciones sintácticas y combinaciones de éstas, para código escrito en lenguaje “java”.
- El método no indicará si el código de entrada se encuentra libre de errores.
- No se pretende comprobar que la totalidad del proceso de reingeniería para reuso es automático, sino que se reconoce que es necesaria cierto nivel de intervención del experto en el dominio o el experto en programación.
- Así mismo, no se pretende comprobar la interfaz del sistema.

6.4.2 Enfoque

La realización de los casos de prueba y la actividad de pruebas estará asistida por la alumna de maestría del *cenidet* Octavio Tonatíuh Flores Millán. Esto ayudará para asegurar que las pruebas representan efectivamente el desarrollo y uso del sistema SR2 completo.

- **Pruebas del proceso de análisis del código fuente.** La verificación del proceso de análisis de código fuente se llevará a cabo obteniendo información de cada clase que se reciba como entrada, incluyendo el paquete, las librerías, el nombre, las funciones y los atributos de cada clase. Además, se comprobará la correcta generación de la lista de clases objetivo necesarias para el método de refactorización.
- **Pruebas de refactorización.** La validación del método de refactorización de código dispensable consiste en identificar y eliminar el código innecesario. La validación se realizará ejecutando el código original y comparándolo con el código refactorizado. Ambos sistemas deberán comportarse de manera idéntica y ofrecer los mismos resultados bajo las mismas entradas, asegurando que la eliminación del código dispensable no afecta el funcionamiento del sistema.
- **Pruebas de calidad.** Las pruebas de calidad incluyen la aplicación de métricas de legibilidad y medición de espacio de memoria para la medición comparativa del código original contra el código orientado a objetos obtenido del proceso de refactorización.

6.4.3 Criterio Pasa / No-pasa de casos de prueba.

Para los casos de prueba del proceso de análisis del código fuente en java, el criterio pasa / no- pasa se realizará mediante la comparación del análisis y la obtención de información manual contra el análisis y la obtención de información de manera automática. En ambos casos se deberá obtener la misma información, comprobándose además la generación correcta de la lista de clases objeto requeridas por el proceso de refactorización.

Para los casos de prueba del método de refactorización, el criterio pasa / no pasa se realizará mediante la comparativa de los resultados obtenidos de forma manual contra los resultados obtenidos de forma automática en cada caso de prueba.

Para los casos de prueba de calidad, el criterio pasa / no-pasa será mediante la comparación del resultado obtenido por calculo manual contra el resultado obtenido de manera automática, ambos deberán ser los mismos.

6.4.4 Criterio de suspensión y requisitos de reanudación.

En ningún caso se suspenderán definitivamente las pruebas. Cada vez que se presente que un caso no-pasa la prueba, inmediatamente se procederá a evaluar y corregir el error, permaneciendo en la prueba de este caso hasta que ya no se presenten dificultades con el caso.

6.4.5 Liberación de pruebas.

La entrada y salida de los datos especificados en cada caso de prueba es suficiente para la aceptación de cada uno de los subsistemas descritos.

6.5 Especificación del diseño de pruebas

6.5.1 Diseño de Prueba: ISMRCD04 – 01.

- 1.- Diseño de Prueba: ISMRCD04 – 01. Método de refactorización de código dispensable.
- 2.- Características a ser probadas.
 - 2.1. En esta prueba se evaluará el correcto funcionamiento del método de refactorización de código dispensable.
- 3.- Refinamiento del enfoque

El objetivo es evaluar el correcto funcionamiento del método de refactorización de código dispensable, eliminando el código que no participa en el sistema. Antes de ejecutar cada caso de prueba, el código será compilado y ejecutado previamente en un compilador de Java, con el fin de verificar que su construcción sea correcta. Luego, el sistema de refactorización de código dispensable analizará el archivo, realizando un análisis léxico y sintáctico para generar las estructuras de datos necesarias en memoria, preparándolo para el proceso de refactorización.

- 4.- Criterio pasa / no-pasa de evaluación de características.

En cada caso de prueba se especificarán las entradas que el sistema requiere y las salidas o resultados que obtiene, de igual forma se presentarán los resultados obtenidos de manera manual. El criterio de evaluación de esta prueba se realizará tomando en cuenta los resultados manuales. Un caso de prueba debe considerarse válido cuando se empaten los resultados manuales con los resultados obtenidos de forma automática. Para que se pase la prueba, cada característica debe pasar todos sus casos de prueba.

6.5.2 Diseño de Prueba: ISMRCD04 – 02.

- 1.- Diseño de Prueba: ISMRCD04 – 02. Cálculo de métricas de calidad.
- 2.- Características a ser probadas.
 - 2.1.- En esta prueba se evaluará el correcto cálculo de la métrica de legibilidad y medición de espacio de memoria.
- 3.- Refinamiento del enfoque.

El objetivo es evaluar que el correcto funcionamiento de la métrica de legibilidad y medición de espacio de memoria al evaluar un sistema de software legado.

Antes de ejecutar cada caso de prueba, el código será compilado y ejecutado previamente en un compilador de Java, con el objetivo de verificar si está correctamente escrito. Posteriormente, el Marco Orientado a Objetos para el Cálculo de Métricas procesará este código, realizando un análisis léxico y sintáctico para generar la estructura de datos necesaria con la información requerida para realizar dichos cálculos.

4.- Criterio pasa / no-pasa de evaluación de características.

En cada caso de prueba se presentarán los resultados obtenidos de manera manual de la métrica legibilidad y medición de espacio de memoria. El criterio de evaluación de esta prueba se realizará tomando en cuenta los resultados manuales. Un caso de prueba debe considerarse válido cuando se empaten los resultados manuales con los resultados obtenidos de forma automática. Para que se pase la prueba, cada característica debe pasar todos sus casos de prueba.

6.6 Casos de prueba

Para evaluar de manera exhaustiva la efectividad del método de refactorización de código dispensable, se llevó a cabo un proceso riguroso de selección de casos de prueba. Se eligieron proyectos con más de 20 clases de diversas aplicaciones y estructuras de código, obtenidos de repositorios públicos en GitHub. Estos proyectos fueron seleccionados no solo por su tamaño, sino también por su relevancia y complejidad, asegurando así una muestra representativa y adecuada para el análisis. La diversidad de los casos permite observar el comportamiento del método de refactorización en diferentes contextos y escenarios, lo cual es crucial para validar su aplicabilidad general.

El proceso de selección consideró aspectos como la cantidad de líneas de código, la diversidad en el uso de patrones de diseño y la presencia de código potencialmente dispensable. Se buscó incluir proyectos de diferentes dominios y con diferentes grados de madurez para abarcar un amplio espectro de situaciones reales. Esta estrategia permite identificar tanto las fortalezas como las posibles limitaciones del método propuesto, proporcionando una visión integral de su desempeño.

A continuación, se detallan los casos de prueba seleccionados, organizados según los criterios de selección mencionados. Esta información servirá como base para los análisis y comparaciones posteriores, permitiendo una evaluación detallada y fundamentada del impacto de la refactorización de código dispensable en la calidad y mantenibilidad del software.

Caso de Prueba ISMRCD05 - 01: "Hospital Project"

El Caso de Prueba ISMRCD05 - 01, titulado *"Hospital Project"*, se centrará en la refactorización del código del sistema de gestión hospitalaria, compuesto por 35 clases. El principal objetivo es mejorar la calidad del código, eliminando elementos innecesarios, simplificando estructuras y optimizando el uso de memoria, sin alterar el comportamiento ni las funcionalidades del sistema. Es fundamental que el sistema funcione de manera idéntica antes y después de la refactorización.

Objetivos del Caso de Prueba:

1. Refactorización del Código:

- Eliminar código innecesario como variables no utilizadas, clases vacías, métodos no utilizados, funciones vacías y código comentado.

2. Comportamiento Consistente Antes y Después:

- Asegurarse de que el sistema se comporte de manera idéntica antes y después de la refactorización. Esto implica que todas las funcionalidades como la gestión de pacientes, personal médico, inventarios, citas y facturación, operen de la misma manera sin introducir errores o cambios en los resultados.
- Validar que todas las interacciones y flujos de trabajo entre las clases (las 35) se mantengan intactos y sin alteraciones.

3. Optimización del Uso de Memoria:

- Reducir el uso de memoria eliminando objetos y variables innecesarias, sin comprometer la funcionalidad o la eficiencia del sistema.
- Identificar posibles mejoras en la gestión de recursos y asegurarse de que la memoria liberada tras la refactorización no afecte negativamente al rendimiento general.

Criterios de Éxito:

Este caso de prueba será exitoso si:

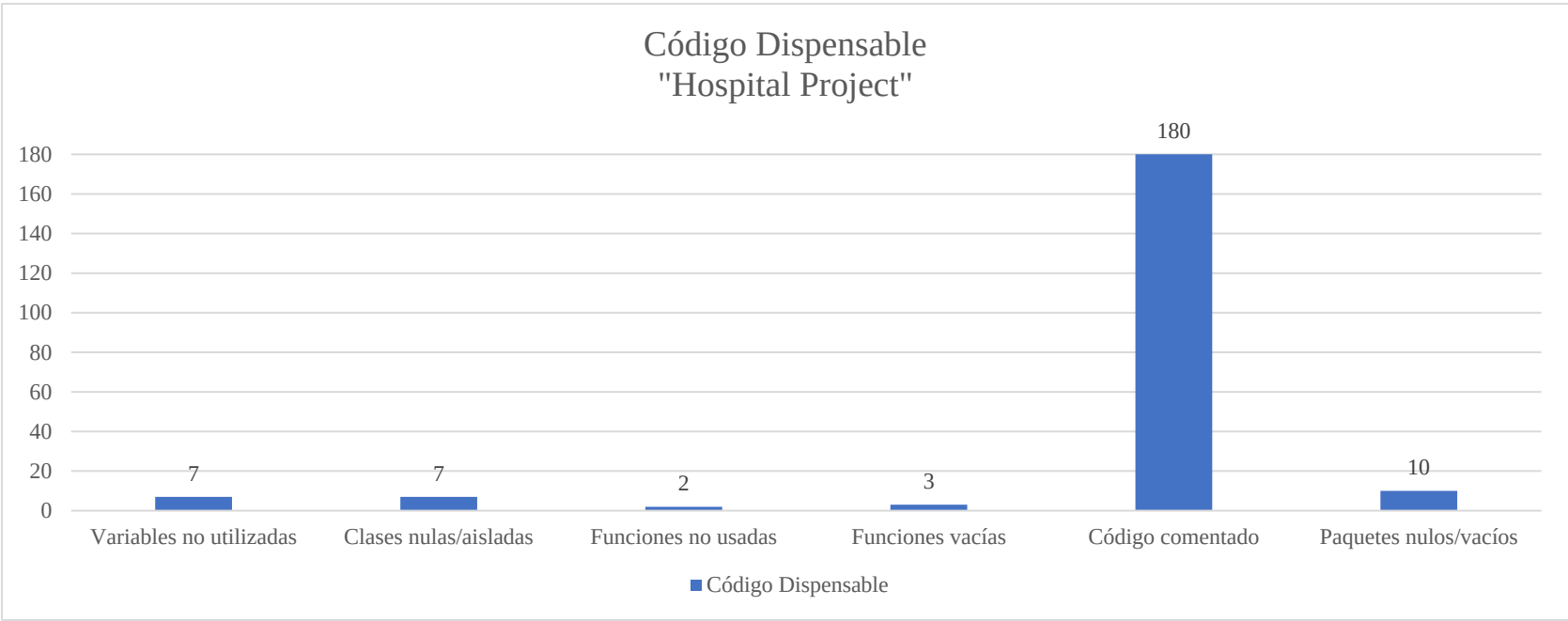
- Se realiza una refactorización efectiva del código, eliminando código innecesario y mejorando la estructura interna, sin introducir errores o cambiar el comportamiento funcional del sistema.
- El sistema se comporta exactamente igual antes y después de la refactorización, manteniendo todas las funcionalidades y resultados esperados sin variaciones.
- Se reduce el uso de memoria y se optimizan los recursos del sistema, pero sin comprometer el rendimiento ni la experiencia del usuario final.

Este caso de prueba asegura que el sistema "Hospital Project" se mantenga funcional y eficiente, pero con un código más limpio, legible y optimizado, garantizando su correcto funcionamiento antes y después de la refactorización.

- Las gráficas de barras muestran que se ha reducido el código innecesario y mejorado la eficiencia en el uso de memoria sin afectar el comportamiento del sistema.
- La comparación gráfica refleja que el sistema se comporta de manera idéntica antes y después de la refactorización.
- Los resultados visuales proporcionan evidencia cuantitativa de que la refactorización ha logrado mejoras significativas en la legibilidad y el uso de memoria.

Este enfoque basado en la teoría de la medición asegura una representación visual clara y cuantificable del impacto de la refactorización de código dispensable, proporcionando una forma precisa de verificar que el sistema es más eficiente y mantenible sin cambiar su comportamiento.

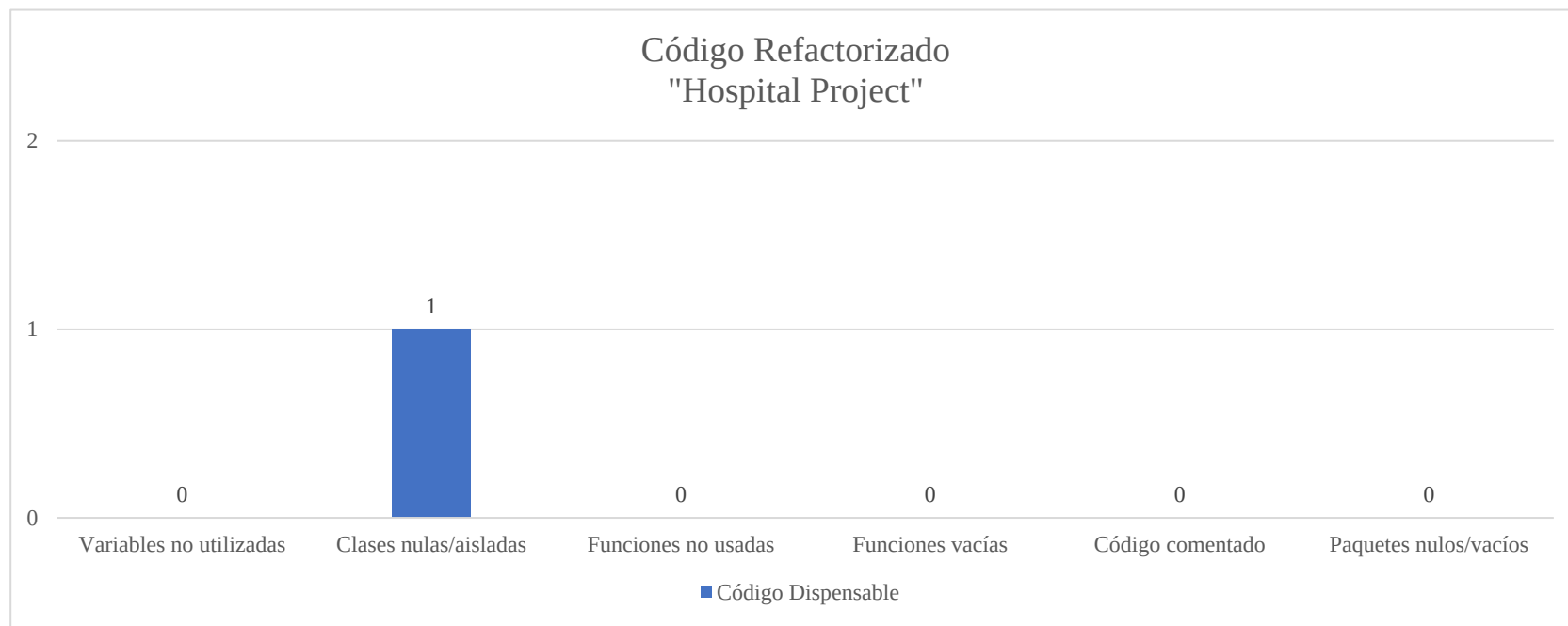
Caso de Prueba ISMRCD05 - 01 "Hospital Project"



Gráfica 1 Código Dispensable “Hospital Project”

Gráfica de Código Dispensable del Caso de Prueba ISMRCD05 - 01 "Hospital Project" antes de la Refactorización

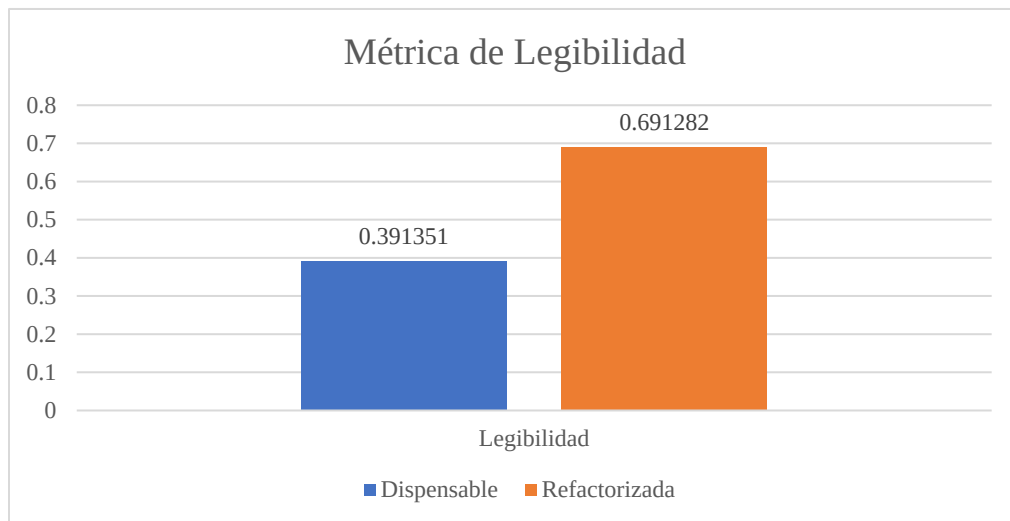
La Gráfica 1 refleja un análisis de la cantidad de entropía generada por código dispensable del proyecto "Hospital Project" antes de aplicar el método de refactorización. Tras la refactorización, se realizará una nueva medición para asegurar que todas las categorías de código dispensable hayan sido significativamente reducidas, manteniendo la misma funcionalidad del sistema sin introducir errores ni alteraciones en su comportamiento.



Gráfica 2 Código Refactorizado “Hospital Project”

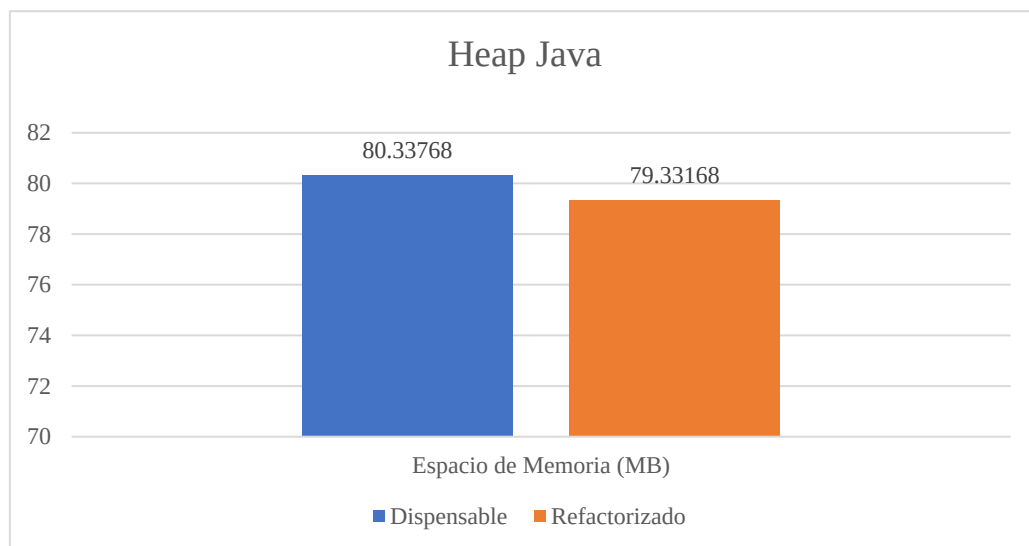
La Gráfica 2 refleja los resultados del análisis tras la aplicación del método de refactorización. Comparada con la Gráfica 1, la gráfica muestra una mejora sustancial en todas las categorías de código dispensable después de la refactorización. Los elementos innecesarios que ocupaban espacio y reducían la claridad del código han sido prácticamente eliminados, excepto por una clase nula/aislada que aún permanece siendo esta la clase que contiene la función **main()**.

Métrica de Legibilidad y Medición de espacio de Memoria Caso de Prueba ISMRCD05 – 01 "Hospital Project"



Gráfica 3 Métrica de Legibilidad "Hospital Project"

En la Gráfica 3, se compara la legibilidad del código antes y después de aplicar técnicas de refactorización. Observamos dos barras que representan la "Legibilidad Dispensable" y la "Legibilidad Refactorizada". La legibilidad refactorizada presenta un valor cercano a 0.7, mientras que la legibilidad dispensable está alrededor de 0.4. Esto sugiere que el proceso de refactorización ha incrementado significativamente la legibilidad del código, lo que indica que ahora es más fácil de entender y mantener.



Gráfica 4 Métrica de Espacio de Memoria "Hospital Project"

En la Gráfica 4, se mide el uso de memoria antes y después de refactorizar el código. Vemos dos barras que representan el "Espacio de Memoria Dispensable" y el "Espacio de Memoria Refactorizada". Aquí se observa una reducción significativa en el uso de memoria después de la refactorización. El espacio de memoria dispensable es superior a 80, mientras que el espacio de memoria refactorizada se encuentra por debajo de 79. Este descenso indica que la refactorización no solo mejoró la legibilidad del código, sino también su eficiencia en cuanto al uso de recursos, específicamente en el manejo de la memoria.

Caso de Prueba ISMRCD05 - 02: "Elevador"

El Caso de Prueba ISMRCD06 - 02, titulado "Elevador", se centrará en la refactorización del código del sistema de control de un elevador, compuesto por 67 clases. El principal objetivo es mejorar la calidad del código, eliminando elementos innecesarios, simplificando estructuras y optimizando el uso de memoria, sin alterar el comportamiento ni las funcionalidades del sistema. Es fundamental que el sistema funcione de manera idéntica antes y después de la refactorización.

Objetivos del Caso de Prueba:

1. Refactorización del Código:

- Eliminar código innecesario, como variables no utilizadas, clases vacías, métodos no utilizados, funciones vacías y código comentado.

2. Comportamiento Consistente Antes y Después:

- Asegurarse de que el sistema se comporte de manera idéntica antes y después de la refactorización. Esto implica que todas las funcionalidades como el manejo de solicitudes de piso, control de movimiento del elevador, gestión de emergencias y estados del sistema operen de la misma manera, sin introducir errores o cambios en los resultados.
- Validar que todas las interacciones y flujos de trabajo entre las clases (las 67) se mantengan intactos y sin alteraciones.

3. Optimización del Uso de Memoria:

- Reducir el uso de memoria eliminando objetos y variables innecesarias, sin comprometer la funcionalidad o la eficiencia del sistema.
- Identificar posibles mejoras en la gestión de recursos y asegurarse de que la memoria liberada tras la refactorización no afecte negativamente al rendimiento general.

Criterios de Éxito:

Este caso de prueba será exitoso si:

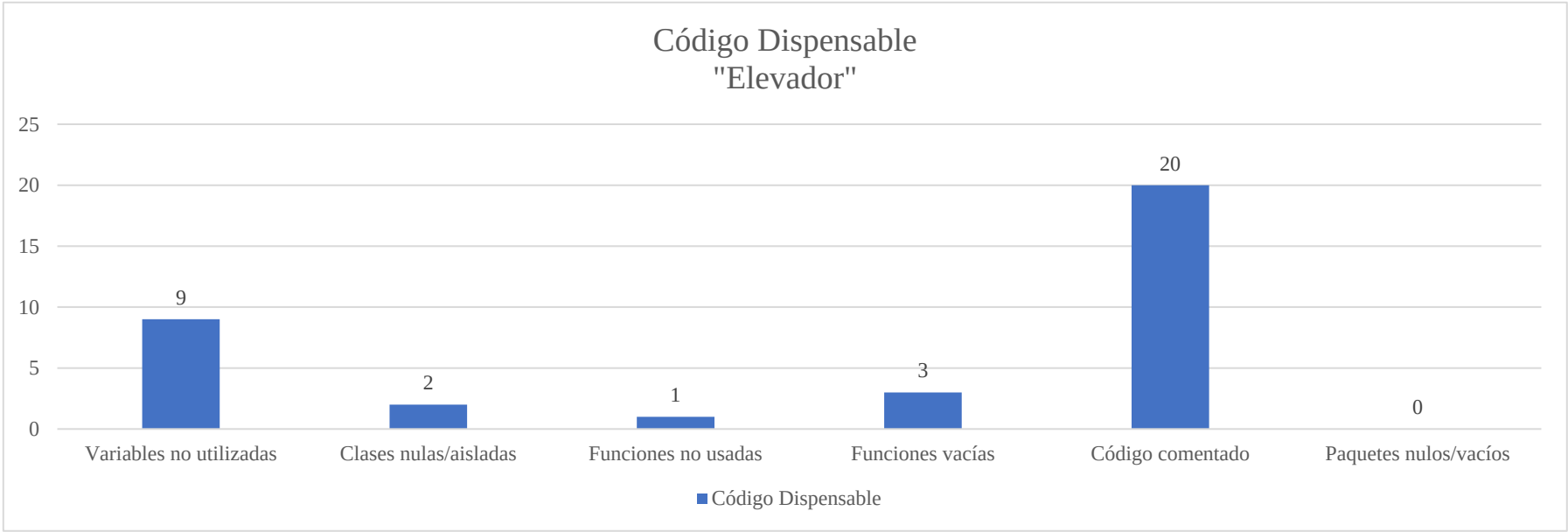
- Se realiza una refactorización efectiva del código, eliminando código innecesario y mejorando la estructura interna, sin introducir errores o cambiar el comportamiento funcional del sistema.
- El sistema se comporta exactamente igual antes y después de la refactorización, manteniendo todas las funcionalidades y resultados esperados sin variaciones.
- Se reduce el uso de memoria y se optimizan los recursos del sistema, pero sin comprometer el rendimiento ni la experiencia del usuario final

Este caso de prueba asegura que el sistema "Elevador" se mantenga funcional y eficiente, pero con un código más limpio, legible y optimizado, garantizando su correcto funcionamiento antes y después de la refactorización.

- Las gráficas de barras mostrarán que se ha reducido el código innecesario y mejorado la eficiencia en el uso de memoria sin afectar el comportamiento del sistema.
- La comparación gráfica reflejará que el sistema se comporta de manera idéntica antes y después de la refactorización.
- Los resultados visuales proporcionarán evidencia cuantitativa de que la refactorización ha logrado mejoras significativas en la legibilidad y el uso de memoria.

Este enfoque basado en la teoría de la medición asegura una representación visual clara y cuantificable del impacto de la refactorización de código dispensable, proporcionando una forma precisa de verificar que el sistema es más eficiente y mantenible sin cambiar su comportamiento.

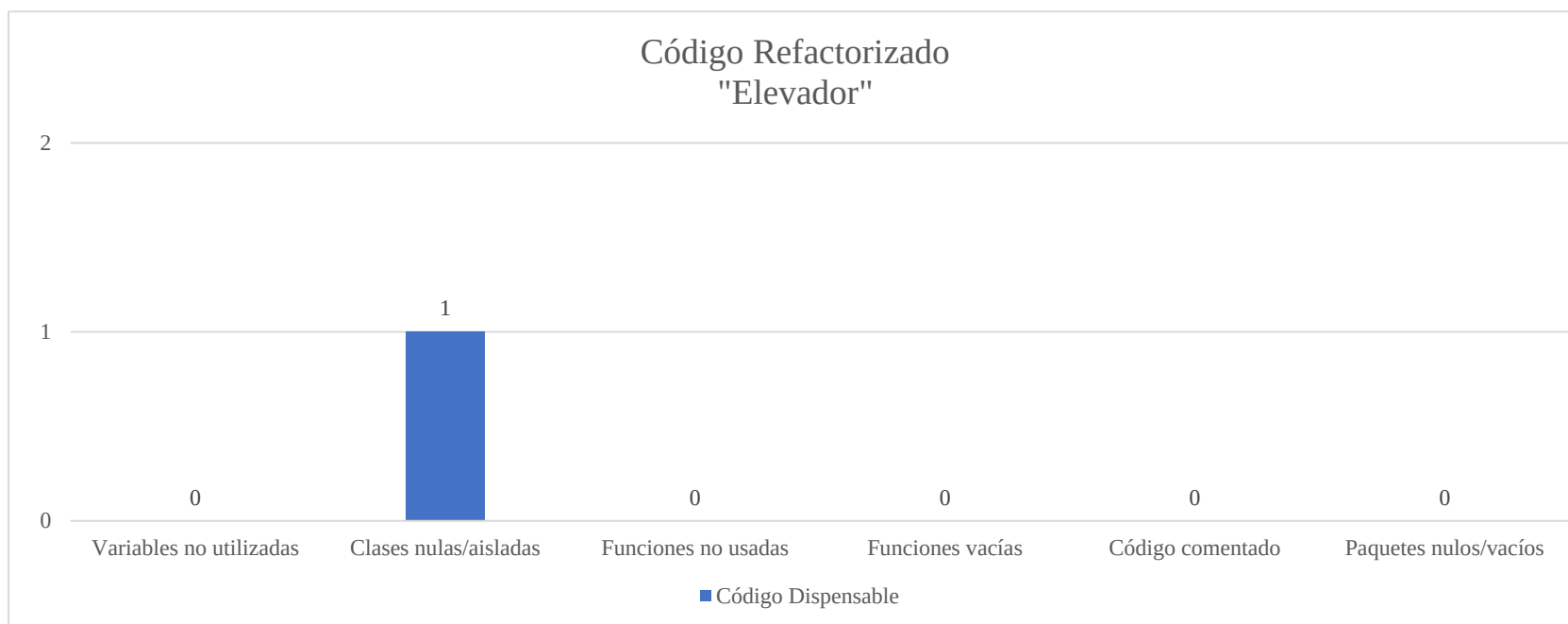
Caso de Prueba ISMRCD05 - 02 "Elevador"



Gráfica 5 Código Dispensable “Elevador”

Gráfica de Código Dispensable del Caso de Prueba ISMRCD05 - 02: "Elevador" antes de la Refactorización

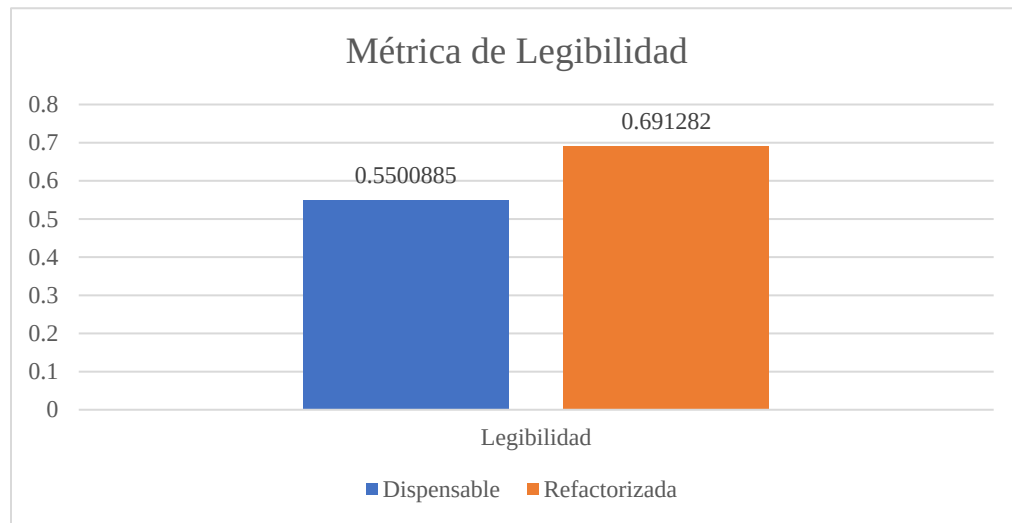
La Gráfica 5 refleja un análisis de la cantidad de entropía generada por código dispensable del proyecto "*Elevador*" antes de aplicar el método de refactorización. Tras la refactorización, se realizará una nueva medición para asegurar que todas las categorías de código dispensable hayan sido significativamente reducidas, manteniendo la misma funcionalidad del sistema sin introducir errores ni alteraciones en su comportamiento.



Gráfica 6 Código Refactorizado "Elevador"

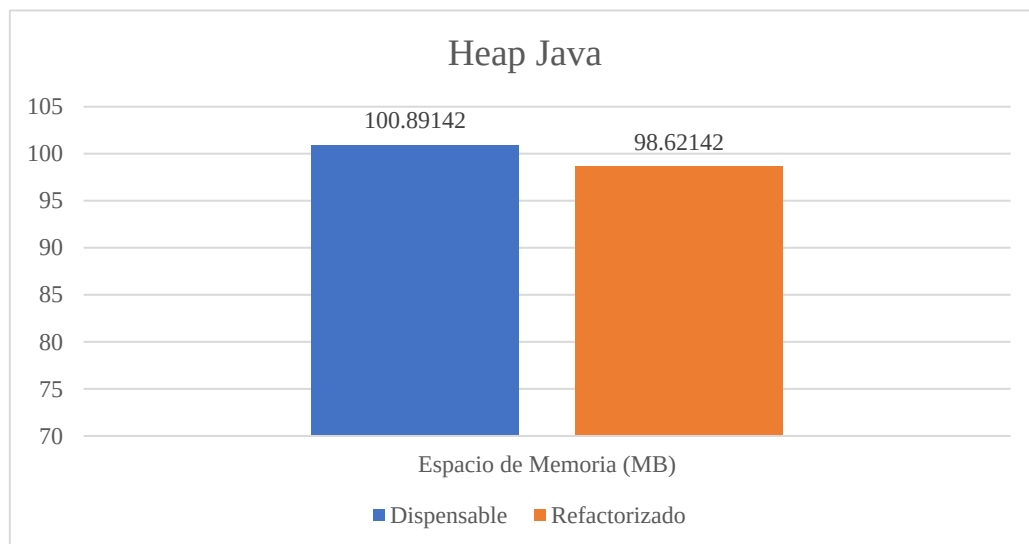
La Gráfica 6 refleja los resultados del análisis tras la aplicación del método de refactorización. Comparada con la Gráfica 5, la gráfica muestra una mejora sustancial en todas las categorías de código dispensable después de la refactorización. Los elementos innecesarios que ocupaban espacio y reducían la claridad del código han sido prácticamente eliminados, excepto por una clase nula/aislada que aún permanece siendo esta la clase que contiene la función `main()`.

Métrica de Legibilidad y Medición de espacio de Memoria Caso de Prueba ISMRCD05 – 02 "Elevador"



Gráfica 7 Métrica de Legibilidad "Elevador"

En la Gráfica 7, se compara la legibilidad del código antes y después de aplicar técnicas de refactorización. Observamos dos barras que representan la "Legibilidad Dispensable" y la "Legibilidad Refactorizada". La legibilidad refactorizada presenta un valor cercano a 0.5, mientras que la legibilidad dispensable está alrededor de 0.7. Esto sugiere que el proceso de refactorización ha incrementado significativamente la legibilidad del código, lo que indica que ahora es más fácil de entender y mantener.



Gráfica 8 Métrica de Espacio de Memoria "Elevador"

En la Gráfica 8, se mide el uso de memoria antes y después de refactorizar el código. Vemos dos barras que representan el "Espacio de Memoria Dispensable" y el "Espacio de Memoria Refactorizada". Aquí se observa una reducción significativa en el uso de memoria después de la refactorización. El espacio de memoria dispensable es superior a 100, mientras que el espacio de memoria refactorizada se encuentra por debajo de 100. Este descenso indica que la refactorización no solo mejoró la legibilidad del código, sino también su eficiencia en cuanto al uso de recursos, específicamente en el manejo de la memoria.

Caso de Prueba ISMRCD05 - 03: "Password Manager"

El Caso de Prueba ISMRCD05 - 03, titulado " Password Manager", se centrará en la refactorización del código del sistema de control de un elevador, compuesto por 105 clases. El principal objetivo es mejorar la calidad del código, eliminando elementos innecesarios, simplificando estructuras y optimizando el uso de memoria, sin alterar el comportamiento ni las funcionalidades del sistema. Es fundamental que el sistema funcione de manera idéntica antes y después de la refactorización.

Objetivos del Caso de Prueba:

4. Refactorización del Código:

- Eliminar código innecesario, como variables no utilizadas, clases vacías, métodos no utilizados, funciones vacías y código comentado.

5. Comportamiento Consistente Antes y Después:

- Asegurarse de que el sistema se comporte de manera idéntica antes y después de la refactorización. Esto implica que todas las funcionalidades como el manejo de solicitudes de piso, control de movimiento del elevador, gestión de emergencias y estados del sistema operen de la misma manera, sin introducir errores o cambios en los resultados.
- Validar que todas las interacciones y flujos de trabajo entre las clases (las 105) se mantengan intactos y sin alteraciones.

6. Optimización del Uso de Memoria:

- Reducir el uso de memoria eliminando objetos y variables innecesarias, sin comprometer la funcionalidad o la eficiencia del sistema.
- Identificar posibles mejoras en la gestión de recursos y asegurarse de que la memoria liberada tras la refactorización no afecte negativamente al rendimiento general.

Criterios de Éxito:

Este caso de prueba será exitoso si:

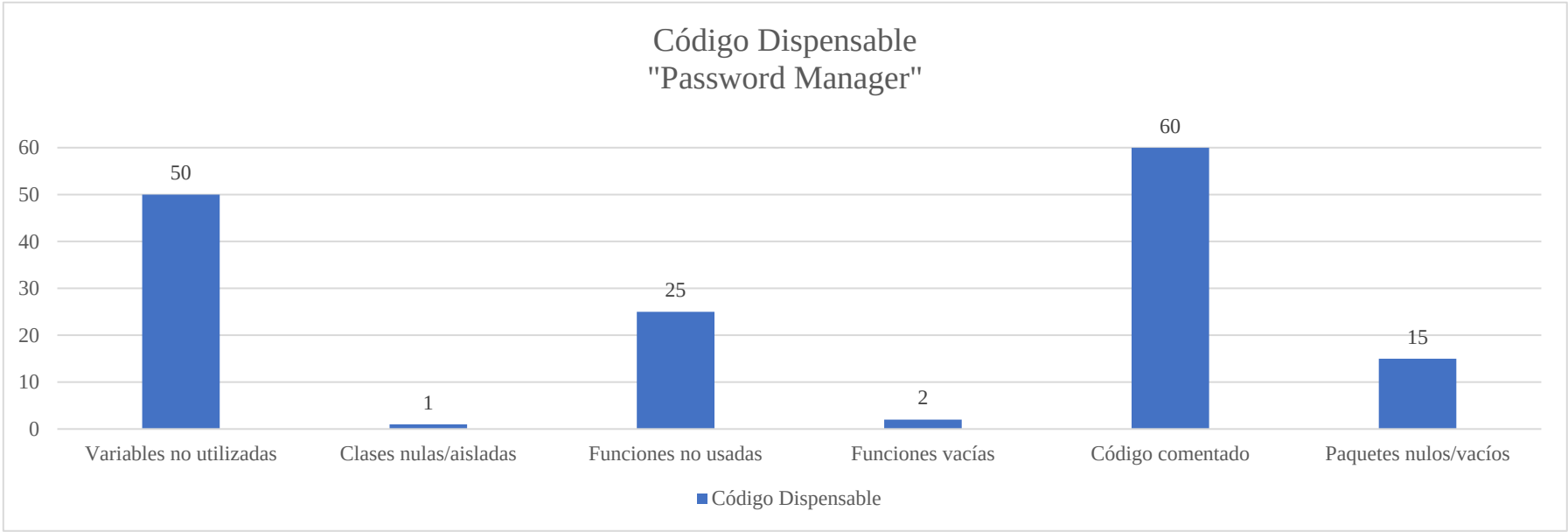
- Se realiza una refactorización efectiva del código, eliminando código innecesario y mejorando la estructura interna, sin introducir errores o cambiar el comportamiento funcional del sistema.
- El sistema se comporta exactamente igual antes y después de la refactorización, manteniendo todas las funcionalidades y resultados esperados sin variaciones.
- Se reduce el uso de memoria y se optimizan los recursos del sistema, pero sin comprometer el rendimiento ni la experiencia del usuario final.

Este caso de prueba asegura que el sistema "Password Manager" se mantenga funcional y eficiente, pero con un código más limpio, legible y optimizado, garantizando su correcto funcionamiento antes y después de la refactorización.

- Las gráficas de barras mostrarán que se ha reducido el código innecesario y mejorado la eficiencia en el uso de memoria sin afectar el comportamiento del sistema.
- La comparación gráfica reflejará que el sistema se comporta de manera idéntica antes y después de la refactorización.
- Los resultados visuales proporcionarán evidencia cuantitativa de que la refactorización ha logrado mejoras significativas en la legibilidad y el uso de memoria.

Este enfoque basado en la teoría de la medición asegura una representación visual clara y cuantificable del impacto de la refactorización de código dispensable, proporcionando una forma precisa de verificar que el sistema es más eficiente y mantenible sin cambiar su comportamiento.

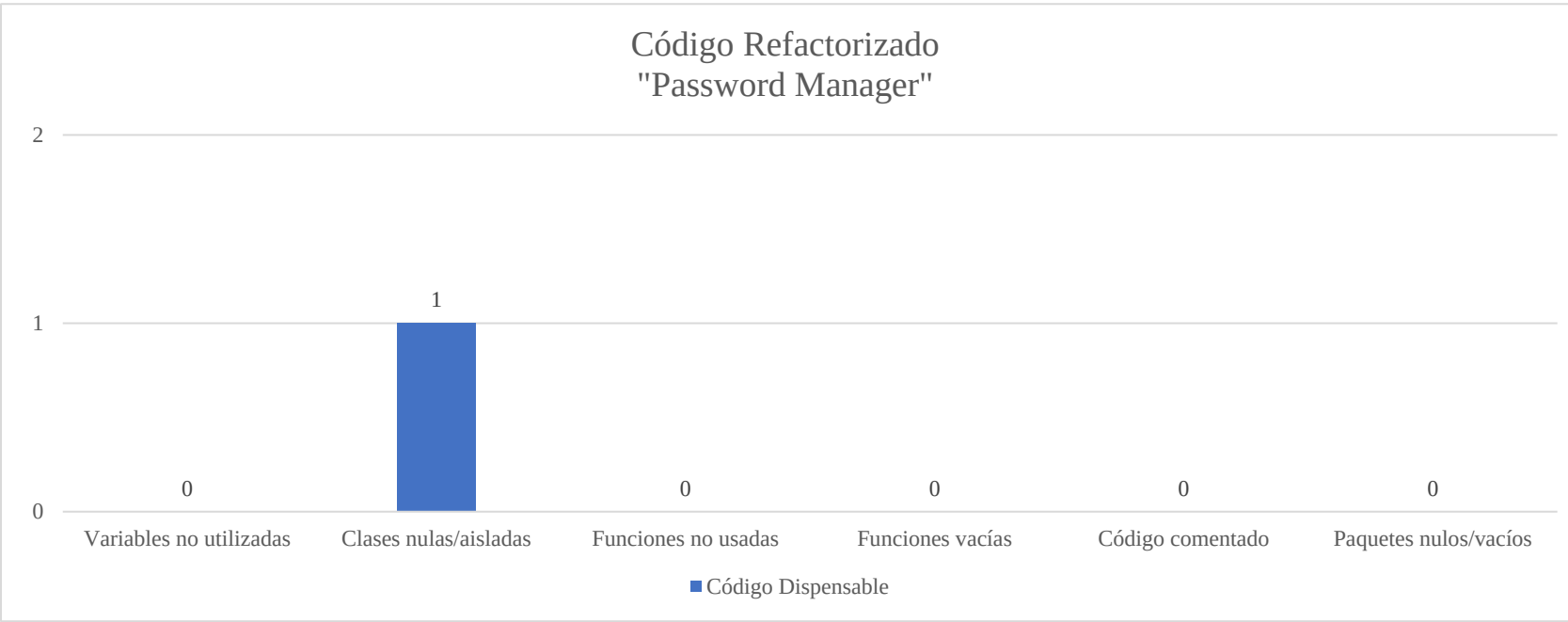
Caso de Prueba ISMRCD05 - 03 "Password Manager"



Gráfica 9 Código Dispensable “Password Manager”

Gráfica de Código Dispensable del Caso de Prueba ISMRCD05 - 03: "Password Manager" antes de la Refactorización

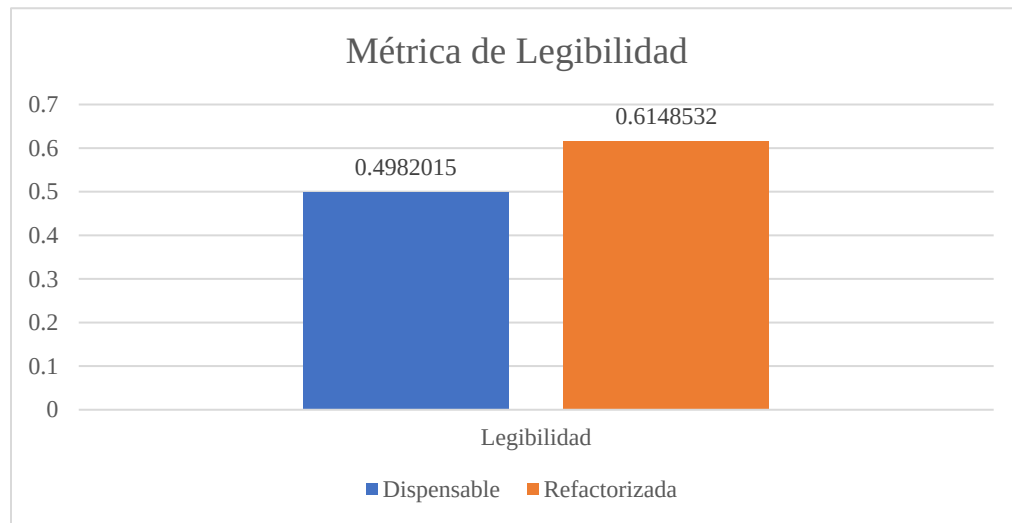
La Gráfica 9 refleja un análisis de la cantidad de entropía generada por código dispensable del proyecto "Password Manager" antes de aplicar el método de refactorización. Tras la refactorización, se realizará una nueva medición para asegurar que todas las categorías de código dispensable hayan sido significativamente reducidas, manteniendo la misma funcionalidad del sistema sin introducir errores ni alteraciones en su comportamiento.



Gráfica 10 Código Refactorizado "Password Manager"

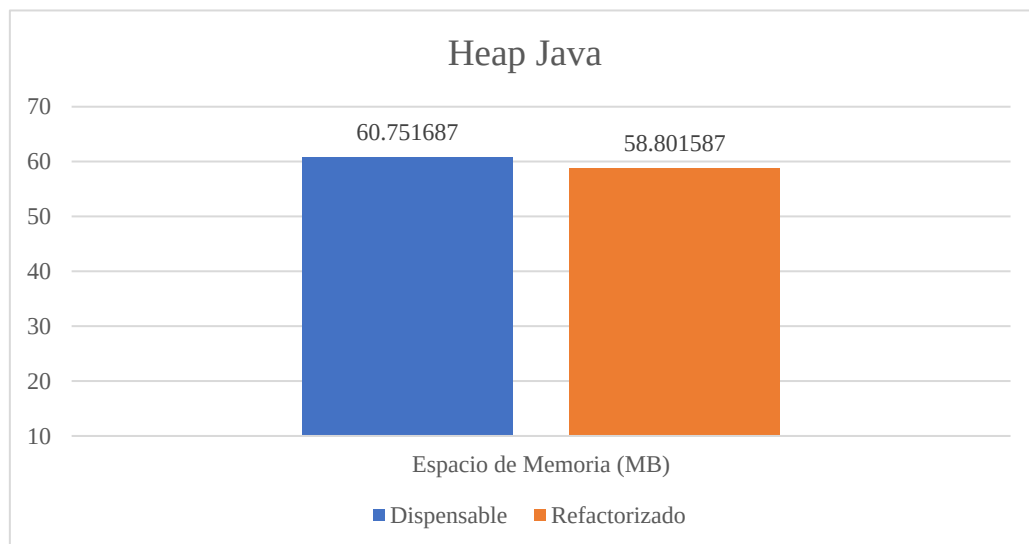
La Gráfica 10 refleja los resultados del análisis tras la aplicación del método de refactorización. Comparada con la Gráfica 9, la gráfica muestra una mejora sustancial en todas las categorías de código dispensable después de la refactorización. Los elementos innecesarios que ocupaban espacio y reducían la claridad del código han sido prácticamente eliminados, excepto por una clase nula/aislada que aún permanece siendo esta la clase que contiene la función **main()**.

Métrica de Legibilidad y Medición de espacio de Memoria Caso de Prueba ISMRCD05 – 03 "Password Manager"



Gráfica 11 Métrica de Legibilidad “Password Manager”

En la Gráfica 11, se compara la legibilidad del código antes y después de aplicar técnicas de refactorización. Observamos dos barras que representan la "Legibilidad Dispensable" y la "Legibilidad Refactorizada". La legibilidad refactorizada presenta un valor cercano a 0.6, mientras que la legibilidad dispensable está alrededor de 0.5. Esto sugiere que el proceso de refactorización ha incrementado significativamente la legibilidad del código, lo que indica que ahora es más fácil de entender y mantener.



Gráfica 12 Métrica de Espacio de Memoria “Password Manager”

En la Gráfica 12, se mide el uso de memoria antes y después de refactorizar el código. Vemos dos barras que representan el "Espacio de Memoria Dispensable" y el "Espacio de Memoria Refactorizada". Aquí se observa una reducción significativa en el uso de memoria después de la refactorización. El espacio de memoria dispensable es superior a 60, mientras que el espacio de memoria refactorizada se encuentra por debajo de 59. Este descenso indica que la refactorización no solo mejoró la legibilidad del código, sino también su eficiencia en cuanto al uso de recursos, específicamente en el manejo de la memoria.

Caso de Prueba ISMRCD05 - 04: "Tetris for Desktop"

El Caso de Prueba ISMRCD05 - 04, titulado "Tetris for Desktop", se centrará en la refactorización del código del sistema de control de un elevador, compuesto por 61 clases. El principal objetivo es mejorar la calidad del código, eliminando elementos innecesarios, simplificando estructuras y optimizando el uso de memoria, sin alterar el comportamiento ni las funcionalidades del sistema. Es fundamental que el sistema funcione de manera idéntica antes y después de la refactorización.

Objetivos del Caso de Prueba:

7. Refactorización del Código:

- Eliminar código innecesario, como variables no utilizadas, clases vacías, métodos no utilizados, funciones vacías y código comentado.

8. Comportamiento Consistente Antes y Después:

- Asegurarse de que el sistema se comporte de manera idéntica antes y después de la refactorización. Esto implica que todas las funcionalidades como el manejo de solicitudes de piso, control de movimiento del elevador, gestión de emergencias y estados del sistema operen de la misma manera, sin introducir errores o cambios en los resultados.
- Validar que todas las interacciones y flujos de trabajo entre las clases (las 61) se mantengan intactos y sin alteraciones.

9. Optimización del Uso de Memoria:

- Reducir el uso de memoria eliminando objetos y variables innecesarias, sin comprometer la funcionalidad o la eficiencia del sistema.
- Identificar posibles mejoras en la gestión de recursos y asegurarse de que la memoria liberada tras la refactorización no afecte negativamente al rendimiento general.

Criterios de Éxito:

Este caso de prueba será exitoso si:

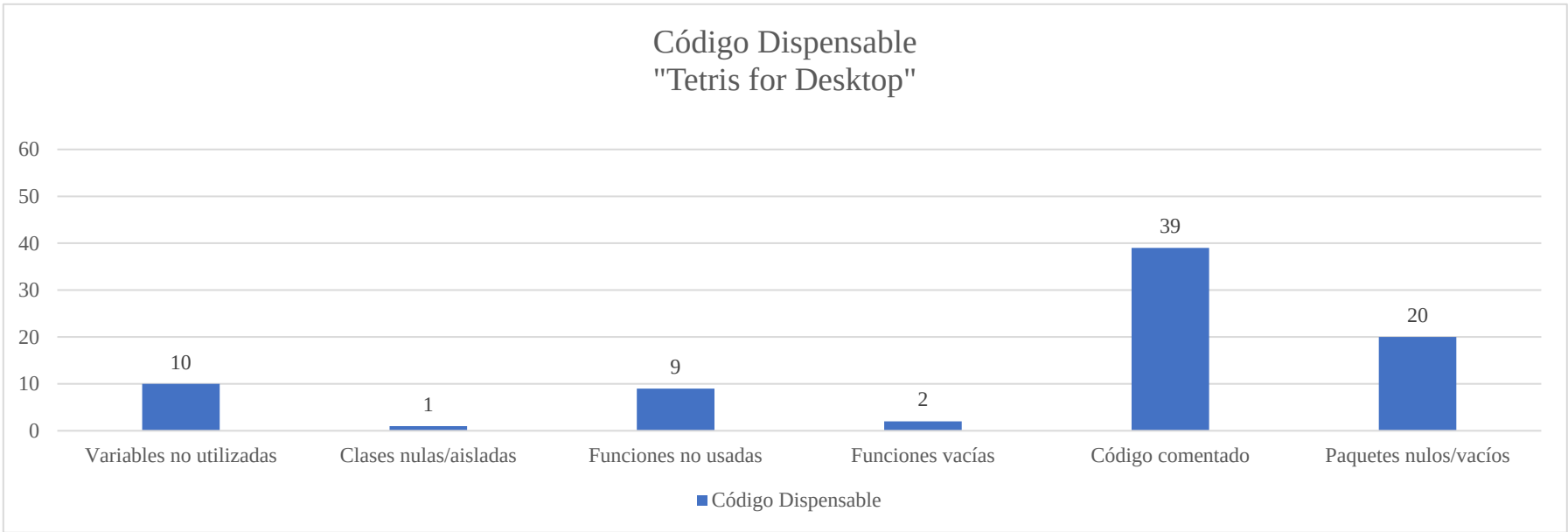
- Se realiza una refactorización efectiva del código, eliminando código innecesario y mejorando la estructura interna, sin introducir errores o cambiar el comportamiento funcional del sistema.
- El sistema se comporta exactamente igual antes y después de la refactorización, manteniendo todas las funcionalidades y resultados esperados sin variaciones.
- Se reduce el uso de memoria y se optimizan los recursos del sistema, pero sin comprometer el rendimiento ni la experiencia del usuario final.

Este caso de prueba asegura que el sistema "Tetris for Desktop" se mantenga funcional y eficiente, pero con un código más limpio, legible y optimizado, garantizando su correcto funcionamiento antes y después de la refactorización.

- Las gráficas de barras mostrarán que se ha reducido el código innecesario y mejorado la eficiencia en el uso de memoria sin afectar el comportamiento del sistema.
- La comparación gráfica reflejará que el sistema se comporta de manera idéntica antes y después de la refactorización.
- Los resultados visuales proporcionarán evidencia cuantitativa de que la refactorización ha logrado mejoras significativas en la legibilidad y el uso de memoria.

Este enfoque basado en la teoría de la medición asegura una representación visual clara y cuantificable del impacto de la refactorización de código dispensable, proporcionando una forma precisa de verificar que el sistema es más eficiente y mantenible sin cambiar su comportamiento.

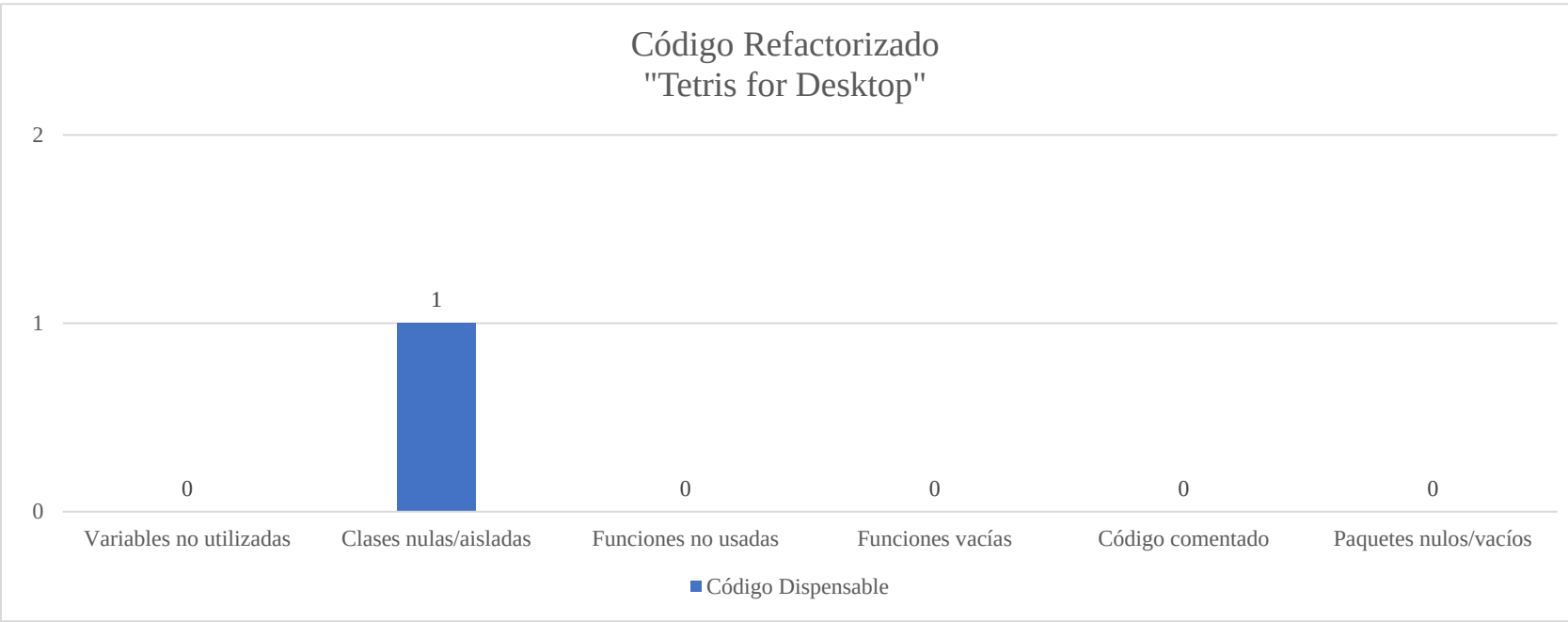
Caso de Prueba ISMRCD05 - 04 "Tetris for Desktop"



Gráfica 13 Código Dispensable “Tetris for Desktop”

Gráfica de Código Dispensable del Caso de Prueba ISMRCD05 - 04: "Tetris for Desktop" antes de la Refactorización

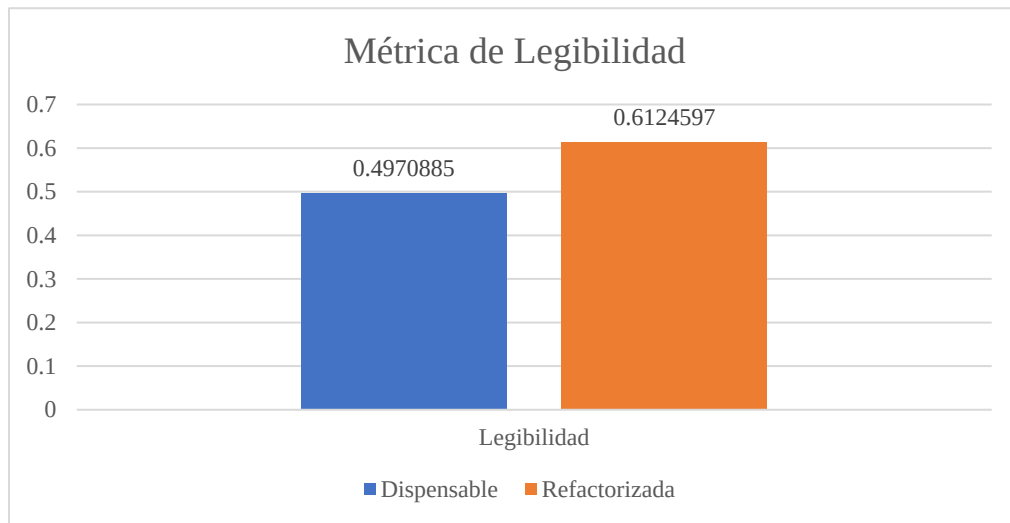
La Gráfica 13 refleja un análisis de la cantidad de entropía generada por código dispensable del proyecto *"Tetris for Desktop"* antes de aplicar el método de refactorización. Tras la refactorización, se realizará una nueva medición para asegurar que todas las categorías de código dispensable hayan sido significativamente reducidas, manteniendo la misma funcionalidad del sistema sin introducir errores ni alteraciones en su comportamiento.



Gráfica 14 Código Refactorizado “Tetris for Desktop”

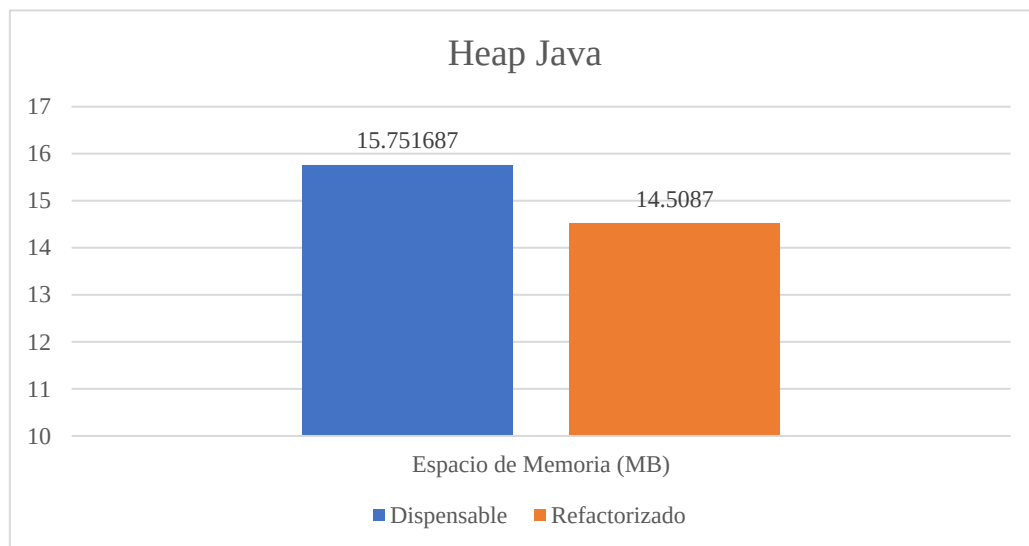
La Gráfica 14 refleja los resultados del análisis tras la aplicación del método de refactorización. Comparada con la Gráfica 13, la gráfica muestra una mejora sustancial en todas las categorías de código dispensable después de la refactorización. Los elementos innecesarios que ocupaban espacio y reducían la claridad del código han sido prácticamente eliminados, excepto por una clase nula/aislada que aún permanece siendo esta la clase que contiene la función **main()**.

Métrica de Legibilidad y Medición de espacio de Memoria Caso de Prueba ISMRCD05 – 04 "Tetris for Desktop"



Gráfica 15 Métrica de Legibilidad "Tetris for Desktop"

En la Gráfica 15, se compara la legibilidad del código antes y después de aplicar técnicas de refactorización. Observamos dos barras que representan la "Legibilidad Dispensable" y la "Legibilidad Refactorizada". La legibilidad refactorizada presenta un valor cercano a 0.6, mientras que la legibilidad dispensable está alrededor de 0.4. Esto sugiere que el proceso de refactorización ha incrementado significativamente la legibilidad del código, lo que indica que ahora es más fácil de entender y mantener.



Gráfica 16 Métrica de Espacio de Memoria "Password Manager"

En la Gráfica 16, se mide el uso de memoria antes y después de refactorizar el código. Vemos dos barras que representan el "Espacio de Memoria Dispensable" y el "Espacio de Memoria Refactorizada". Aquí se observa una reducción significativa en el uso de memoria después de la refactorización. El espacio de memoria dispensable es superior a 15, mientras que el espacio de memoria refactorizada se encuentra por debajo de 15. Este descenso indica que la refactorización no solo mejoró la legibilidad del código, sino también su eficiencia en cuanto al uso de recursos, específicamente en el manejo de la memoria.

Caso de Prueba ISMRCD05 - 05: "Universal Calculator"

El Caso de Prueba ISMRCD05 - 05, titulado "Universal Calculator", se centrará en la refactorización del código del sistema de control de un elevador, compuesto por 15 clases. El principal objetivo es mejorar la calidad del código, eliminando elementos innecesarios, simplificando estructuras y optimizando el uso de memoria, sin alterar el comportamiento ni las funcionalidades del sistema. Es fundamental que el sistema funcione de manera idéntica antes y después de la refactorización.

Objetivos del Caso de Prueba:

10. Refactorización del Código:

- Eliminar código innecesario, como variables no utilizadas, clases vacías, métodos no utilizados, funciones vacías y código comentado.

11. Comportamiento Consistente Antes y Después:

- Asegurarse de que el sistema se comporte de manera idéntica antes y después de la refactorización. Esto implica que todas las funcionalidades como el manejo de solicitudes de piso, control de movimiento del elevador, gestión de emergencias y estados del sistema operen de la misma manera, sin introducir errores o cambios en los resultados.
- Validar que todas las interacciones y flujos de trabajo entre las clases (las 15) se mantengan intactos y sin alteraciones.

12. Optimización del Uso de Memoria:

- Reducir el uso de memoria eliminando objetos y variables innecesarias, sin comprometer la funcionalidad o la eficiencia del sistema.
- Identificar posibles mejoras en la gestión de recursos y asegurarse de que la memoria liberada tras la refactorización no afecte negativamente al rendimiento general.

Criterios de Éxito:

Este caso de prueba será exitoso si:

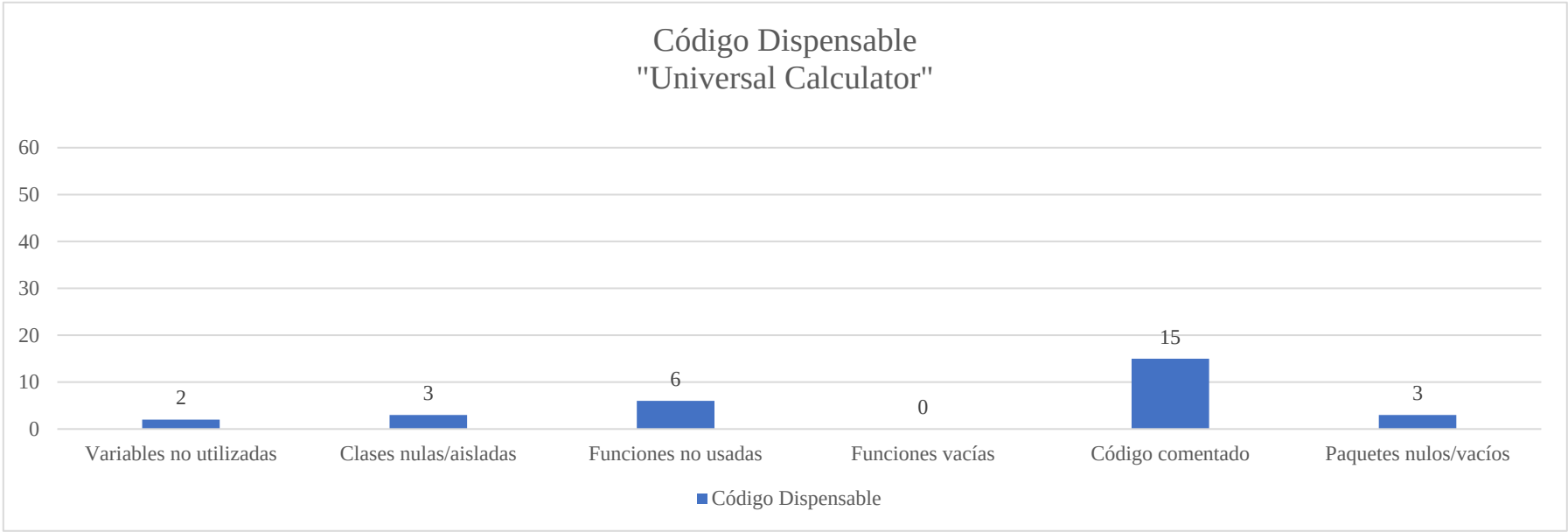
- Se realiza una refactorización efectiva del código, eliminando código innecesario y mejorando la estructura interna, sin introducir errores o cambiar el comportamiento funcional del sistema.
- El sistema se comporta exactamente igual antes y después de la refactorización, manteniendo todas las funcionalidades y resultados esperados sin variaciones.
- Se reduce el uso de memoria y se optimizan los recursos del sistema, pero sin comprometer el rendimiento ni la experiencia del usuario final.

Este caso de prueba asegura que el sistema "Universal Calculator" se mantenga funcional y eficiente, pero con un código más limpio, legible y optimizado, garantizando su correcto funcionamiento antes y después de la refactorización.

- Las gráficas de barras mostrarán que se ha reducido el código innecesario y mejorado la eficiencia en el uso de memoria sin afectar el comportamiento del sistema.
- La comparación gráfica reflejará que el sistema se comporta de manera idéntica antes y después de la refactorización.
- Los resultados visuales proporcionarán evidencia cuantitativa de que la refactorización ha logrado mejoras significativas en la legibilidad y el uso de memoria.

Este enfoque basado en la teoría de la medición asegura una representación visual clara y cuantificable del impacto de la refactorización de código dispensable, proporcionando una forma precisa de verificar que el sistema es más eficiente y mantenible sin cambiar su comportamiento.

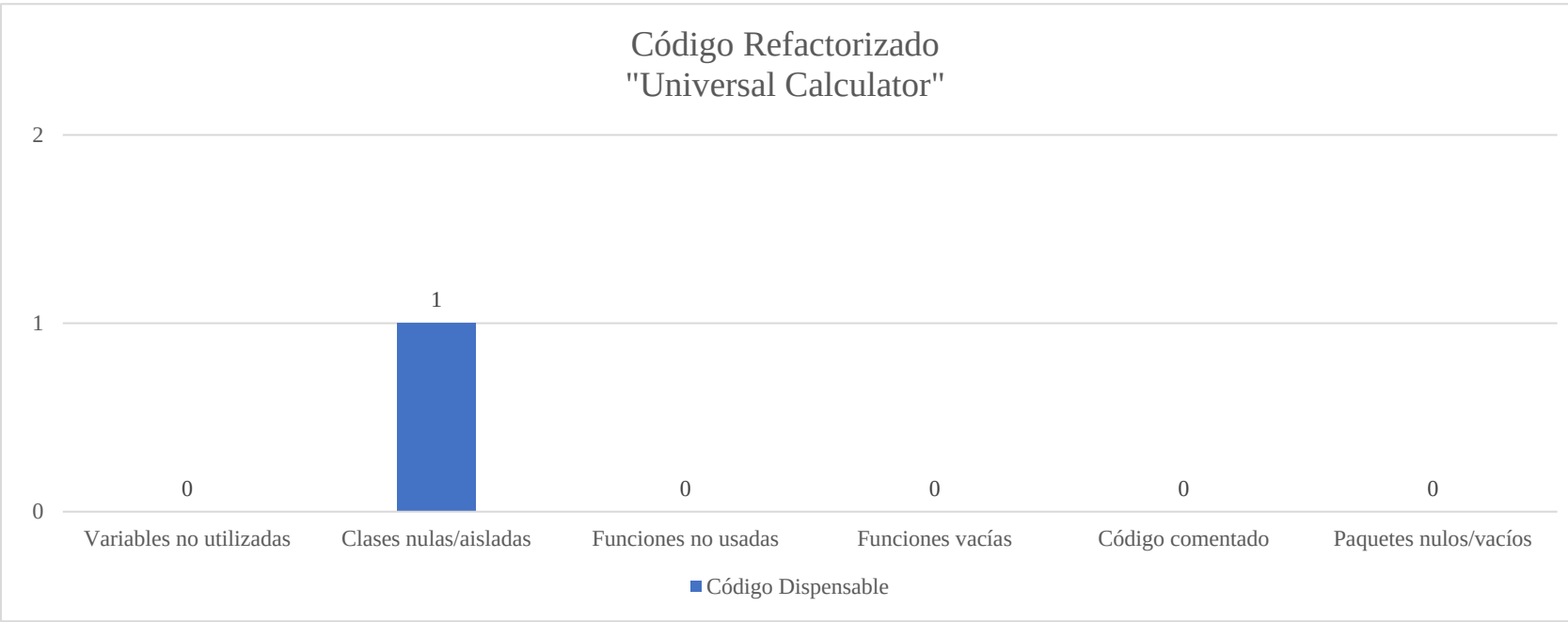
Caso de Prueba ISMRCD05 - 05 "Universal Calculator"



Gráfica 17 Código Dispensable “Universal Calculator”

Gráfica de Código Dispensable del Caso de Prueba ISMRCD05 - 05: "Universal Calculator" antes de la Refactorización

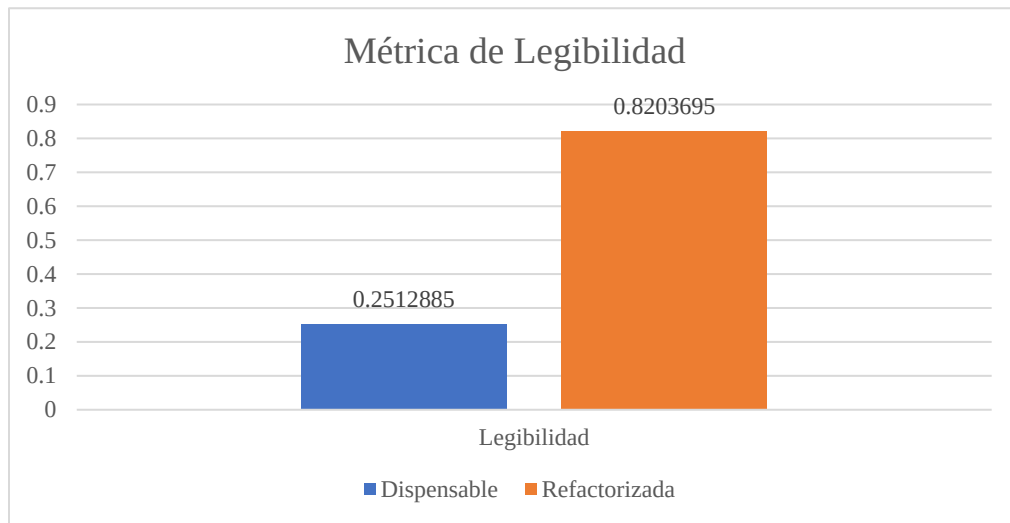
La Gráfica 17 refleja un análisis de la cantidad de entropía generada por código dispensable del proyecto *"Universal Calculator"* antes de aplicar el método de refactorización. Tras la refactorización, se realizará una nueva medición para asegurar que todas las categorías de código dispensable hayan sido significativamente reducidas, manteniendo la misma funcionalidad del sistema sin introducir errores ni alteraciones en su comportamiento.



Gráfica 18 Código Refactorizado “Universal Calculator”

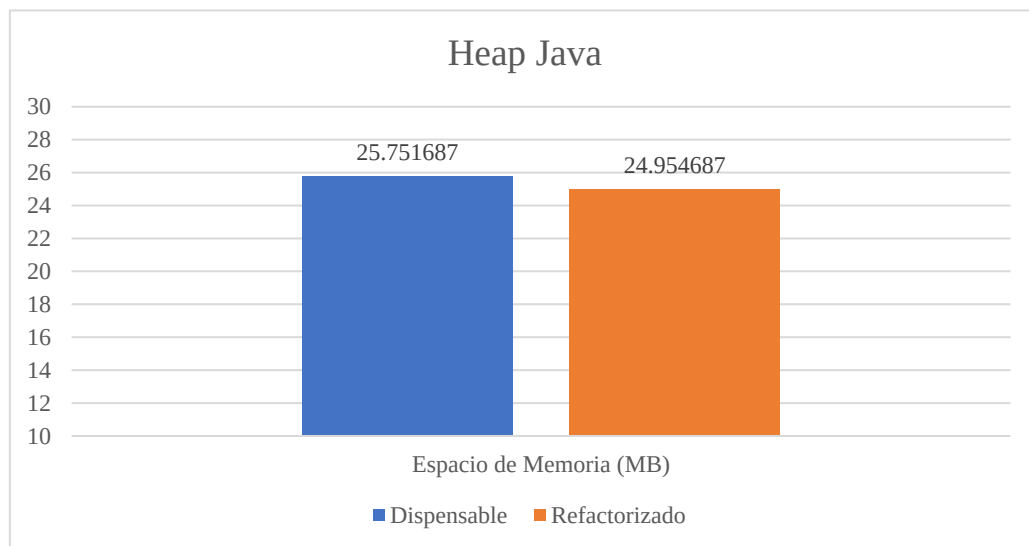
La Gráfica 18 refleja los resultados del análisis tras la aplicación del método de refactorización. Comparada con la Gráfica 17, la gráfica muestra una mejora sustancial en todas las categorías de código dispensable después de la refactorización. Los elementos innecesarios que ocupaban espacio y reducían la claridad del código han sido prácticamente eliminados, excepto por una clase nula/aislada que aún permanece siendo esta la clase que contiene la función **main()**.

Métrica de Legibilidad y Medición de espacio de Memoria Caso de Prueba ISMRCD05 – 05 "Universal Calculator"



Gráfica 19 Métrica de Legibilidad "Universal Calculator"

En la Gráfica 19, se compara la legibilidad del código antes y después de aplicar técnicas de refactorización. Observamos dos barras que representan la "Legibilidad Dispensable" y la "Legibilidad Refactorizada". La legibilidad refactorizada presenta un valor cercano a 0.8, mientras que la legibilidad dispensable está alrededor de 0.2. Esto sugiere que el proceso de refactorización ha incrementado significativamente la legibilidad del código, lo que indica que ahora es más fácil de entender y mantener.



Gráfica 20 Métrica de Espacio de Memoria "Password Manager"

En la Gráfica 16, se mide el uso de memoria antes y después de refactorizar el código. Vemos dos barras que representan el "Espacio de Memoria Dispensable" y el "Espacio de Memoria Refactorizada". Aquí se observa una reducción significativa en el uso de memoria después de la refactorización. El espacio de memoria dispensable es superior a 25, mientras que el espacio de memoria refactorizada se encuentra por debajo de 25. Este descenso indica que la refactorización no solo mejoró la legibilidad del código, sino también su eficiencia en cuanto al uso de recursos, específicamente en el manejo de la memoria.

CAPÍTULO 7 CONCLUSIONES Y TRABAJO FUTURO

7.1 Conclusiones

Reducción Significativa del Código:

La refactorización aplicada permitió una reducción considerable en el total de líneas de código, eliminando métodos, clases y elementos que no aportaban valor al sistema. Esto no solo disminuyó la complejidad del código, sino que también mejoró su estructura, facilitando futuras revisiones y mantenimientos, se reducen los posibles puntos de error y se mejora la eficiencia del sistema, permitiendo que los desarrolladores puedan enfocarse en implementar nuevas funcionalidades sin el obstáculo de un código dispensable.

Optimización del Uso de Memoria:

La eliminación de variables de referencia, atributos y objetos innecesarios resultó en una notable disminución del uso de memoria dentro del sistema. Esta optimización es especialmente beneficiosa en entornos de producción con alta carga o en sistemas donde los recursos son limitados, ya que contribuye directamente a un mejor rendimiento del software. Con un uso de memoria más eficiente, se reduce la probabilidad de errores relacionados con la gestión de recursos y se minimiza el impacto en el rendimiento durante el tiempo de ejecución, mejorando la estabilidad y la experiencia del usuario.

Reducción de Manipulación de Código por Mantenimiento Adaptativo y Fallos:

La implementación de este método de refactorización permitió una reducción en la deuda técnica acumulada en los sistemas tomados como caso de prueba, lo que a su vez mejoró el rendimiento y la legibilidad del software a largo plazo. La eliminación de código dispensable y la optimización del uso de memoria no solo contribuyeron a mantener un sistema eficiente y manejable, sino que también redujeron la necesidad de intervenciones frecuentes de mantenimiento adaptativo y corrección de fallos.

Mejora de la Legibilidad y Productividad en el Desarrollo:

La refactorización del código hizo que este fuera más claro y fácil de entender para los desarrolladores, promoviendo un entorno de desarrollo más productivo y menos propenso a errores. La mejora en la legibilidad del código no solo reduce el tiempo de adaptación para nuevos miembros del equipo, sino que también disminuye el riesgo de introducir errores durante el mantenimiento. Un código más limpio y organizado facilita la identificación de problemas y acelera el desarrollo de nuevas funcionalidades, contribuyendo a un ciclo de vida de desarrollo más eficiente.

Impacto en la Calidad y Mantenibilidad a Largo Plazo:

En conjunto, estos esfuerzos de refactorización contribuyeron significativamente al éxito y la longevidad de los sistemas de software, alineándolos mejor con sus objetivos de calidad y valor. La reducción de código dispensable y la optimización de recursos aseguran una base de código más sostenible y fácil de mantener en el futuro. Este enfoque no solo mejora la calidad del software actual, sino que también prepara al sistema para adaptarse más fácilmente a futuros cambios o expansiones, garantizando así una mayor calidad y mantenibilidad en el largo plazo. La representación visual de estos logros a través de gráficas demuestra de forma cuantitativa el impacto positivo de la refactorización, proporcionando evidencia sólida del valor de estos ajustes en la estructura y desempeño del sistema.

7.2 Aportaciones de la tesis

Método de Refactorización de Código Dispensable

El método es capaz de eliminar el código dispensable en sistemas legados escritos en lenguaje Java, mediante la búsqueda de líneas de código que no contribuyen al valor del sistema. Esto incluye la identificación de atributos, variables de referencia, métodos y clases que no participan en la funcionalidad esencial del software. Al detectar estos elementos innecesarios, el método procede a su eliminación, optimizando así el rendimiento del sistema y reduciendo la complejidad y el mantenimiento requerido.

Una vez que la arquitectura modular se encuentra refactorizada, se respetará el principio de “Abierto-Cerrado”, así como también con el principio de Sustitución de Liskov, haciendo que su mantenimiento sea más fácil y sea reusable.

7.3 Trabajos Futuros

7.3.1 Evaluar sistemas legados que cuenten con código dispensable, del tipo “Importaciones no utilizadas” después de realizar la refactorización a escenarios que cuenten con código dispensable.

Uno de los pasos en la mejora continua de sistemas legados es la evaluación de importaciones no utilizadas. Después de una refactorización completa, es importante revisar y eliminar todas las bibliotecas y paquetes importados que no son necesarios para el funcionamiento del sistema. Este proceso no solo mejora la eficiencia y velocidad del sistema, sino que también reduce el riesgo de vulnerabilidades de seguridad que pueden surgir de dependencias innecesarias.

7.3.2 Evaluar sistemas legados que cuenten con código dispensable, del tipo “Paquetes no utilizados” después de realizar la refactorización a escenarios que cuenten con código dispensable.

Similar a las importaciones no utilizadas, los paquetes no utilizados también deben ser identificados y eliminados después de una refactorización. Este paso asegura que el sistema esté libre de componentes redundantes que pueden aumentar la complejidad y dificultar el mantenimiento. La eliminación de paquetes innecesarios simplifica la estructura del proyecto y facilita la comprensión y modificación del código por parte de los desarrolladores.

7.3.4 Evaluar sistemas legados que cuenten con código dispensable, del tipo “Algoritmos Strategy” después de realizar la refactorización a escenarios que cuenten con código dispensable.

La refactorización de algoritmos “Strategy” también es una tarea en la mejora de sistemas legados. Los algoritmos Strategy permiten a los desarrolladores definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Sin embargo, a lo largo del tiempo y con múltiples cambios en el sistema, algunos de estos algoritmos pueden volverse innecesarios o ineficientes. La evaluación y refactorización de estos algoritmos después de una limpieza inicial asegura que solo las estrategias más eficientes y necesarias permanezcan en el sistema, mejorando así su rendimiento y mantenibilidad.

Esta evaluación integral no solo optimizará el rendimiento del sistema, sino que también simplificará su estructura, facilitando futuras modificaciones y mejoras.

La refactorización de sistemas legados es un proceso continuo y detallado que requiere atención a múltiples aspectos del código. A medida que se identifican y eliminan componentes innecesarios, el sistema se vuelve más eficiente y fácil de mantener, lo que es importante para su evolución y adaptación a nuevas necesidades y tecnologías.

REFERENCIAS

1. Al Dallal, J., & Abdin, A. (2017). Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, PP(1), 1. <https://doi.org/10.1109/TSE.2017.2658573>
2. Arif, A., & Rana, Z. (2020). *Refactoring of code to remove technical debt and reduce maintenance effort* (pp. 1-7). <https://doi.org/10.1109/ICOSST51357.2020.9332917>
3. Barón. (2019). *Método de refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles*.
4. Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy Information Systems: Issues and Directions. *IEEE Software*.
5. Booch, G. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison-Wesley.
6. Buse, R. P. L., & Weimer, W. R. (2008). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 34(5), 1-12.
7. Cárdenas. (2004). *Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator*.
8. Cunningham, W. (1992). The WyCash Portfolio Management System. *OOPSLA '92 Experience Report*.
9. Díaz. (2023). *Disminución de deuda técnica producida por arquitectura de clases con más de una responsabilidad*.
10. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
11. Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
12. Gaitani, M., Zafeiris, V., Diamantidis, N., & Giakoumakis, E. A. (2014). Automated refactoring to the NULL OBJECT design pattern. *Information and Software Technology*, 59, [páginas no especificadas]. <https://doi.org/10.1016/j.infsof.2014.10.010>
13. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
14. García Peñalvo, F. J., Marqués Corral, J. M., & Maudes Raedo, J. M. (1997). *Análisis y Diseño Orientado al Objeto para Reutilización*.
15. Halstead, M. H. (1977). *Elements of software science*. North-Holland.
16. Ivar Jacobson. (1992). *Object-oriented software engineering*. ACM Press. <http://archive.org/details/objectorientedso00jaco>
17. Khatchadourian, R. (2016). Automated refactoring of legacy Java software to enumerated types. *Automated Software Engineering*.
18. Krantz, D. H., Luce, R. D., Suppes, P., & Tversky, A. (1971). *Foundations of measurement* (Vol. 1). Academic Press.
19. L. R. Lara. (2004). El efecto de la entropía en los softwares multimediales educativos. *Revista Iberoamericana de Educación*, 34(2), 1-13. <https://doi.org/10.35362/rie3422996>
20. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
21. Leffingwell, D. (2011). *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley.
22. Liu, H., Li, B., Yang, Y., Ma, W., & Jia, R. (2018). Exploring the impact of code smells on fine-grained structural change-proneness. *International Journal of Software Engineering and Knowledge Engineering*, 28(10), 1487-1516. <https://doi.org/10.1142/S0218194018500432>

23. Liskov, B., & Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
24. Martin, C., & Martin, R. (2006). *Agile Principles, Patterns, and Practices in C#*. Prentice Hall.
25. McConnell, S. (2004). *Code Complete* (2nd ed.). Microsoft Press.
26. Meza. (2022). *Reducción de la fragilidad del software legado originado por el acceso a datos compartidos*.
27. Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice Hall.
28. Murphy-hill, X. G. E. (2014). Manual Refactoring Changes with Automated Refactoring Validation.
29. Ortiz. (2020). *Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos*.
30. Padilla. (2019). *Método de Re-factorización de código java con interfaces y abstracciones*.
31. Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Retrieved from <https://www.antlr>
32. Posnett, D., Hindle, A., & Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 2011 ACM/IEEE International Conference on Software Engineering* (pp. 1-4). ACM.
33. Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach* (6th ed.). McGraw-Hill.
34. Ramírez. (2023). *Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado*.
35. Sánchez. (2023). *Refactorización de Módulos Colaborativos con Carencia de Abstracción*.
36. Santos. (2005). *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter*.
37. Sarkawt Othman, Z., & Kaya, M. (2019). *Refactoring code clone detection* (pp. 1-6). <https://doi.org/10.1109/ISDFS.2019.8757479>
38. Şanlıalp, İ., Ozturk, M., & Yiğit, T. (2022). Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices. *Electronics*, 11(3), 442. <https://doi.org/10.3390/electronics11030442>
39. Singh, S., & Kaur, S. (2016). A systematic literature review: Refactoring for disclosing code smells in object-oriented software. *Ain Shams Engineering Journal*. <https://doi.org/10.1016/j.asej.2017.03.002>
40. Sirqueira, T., Brandl, A., Pedro, E., Silva, R., & Araujo, M. (2016). Code smell analyzer: A tool to teaching support of refactoring techniques source code. *IEEE Latin America Transactions*, 14(2), 877-884. <https://doi.org/10.1109/TLA.2016.7437235>
41. Software Guru. (2019, noviembre 29). Muestra de casos reales de arquitectura de software. REAL. <https://sg.com.mx/real/> (Accedido el 23 de septiembre de 2022).
42. Sommerville, I. (2011). *Software Engineering* (9th ed.). Addison-Wesley.
43. Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, 103(2684), 677-680. <https://doi.org/10.1126/science.103.2684.677>
44. Valdés. (2004). *Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces*.
45. Yli-Huumo, J., Maglyas, A., & Smolander, K. (2016). How do software development teams manage technical debt? – An empirical study. *Journal of Systems and Software*, 120, [páginas no especificadas]. <https://doi.org/10.1016/j.jss.2016.05.018>
46. Yoshida, N., Numata, S., Choi, E., & Inoue, K. *Proactive clone recommendation system for extract method refactoring*. Nagoya University, Osaka University, Nara Institute of Science and Technology, Japan.
47. Zuse, H. (1995). *A framework of software measurement*. Walter de Gruyter.