EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Estudio de la práctica de diseño de software que genera patrones de código mal formado

presentada por

Ing. Claudia de las Mercedes Rodríguez Ponce

como requisito para la obtención del grado de

Maestra en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Codirector de tesis

Dr. Juan Carlos Rojas Pérez

Cuernavaca, Morelos, México. Agosto de 2026.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

Educación
Secretaría de Educación Pública

TECNOLÓGICO NACIONAL DE MÉXICO

Centro Nacional de Investigación y Desarrollo Tecnológico
Departamento de Ciencias Computacionales

Cuernavaca, Mor., 19/JUNIO/2026
OFICIO No. DCC/130/2026
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 **Claudia de las Mercedes Rodríguez Ponce** con número de control **M24CE007**, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Estudio de la práctica de diseño de software que genera patrones de código mal formado"** 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.

A T E N T A M E N T E
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. José Luis Ramírez Alcántara
Revisor 2

Dra. Gloria Piedad Gasca Hurtado
Revisor 3

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

2026 año de Margarita Maza

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

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

Centro Nacional de Investigación y Desarrollo Tecnológico
Subdirección Académica

Cuernavaca Mor, 25/junio/2026
Oficio No. SAC/182/2026
Asunto: Autorización de impresión de tesis

CLAUDIA DE LAS MERCEDES RODRÍGUEZ PONCE
CANDIDATA AL GRADO DE MAESTRA 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 **"Estudio de la práctica de diseño de software que genera patrones de código mal formado"**, 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.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
"Conocimiento y Tecnología al servicio de México"

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO
SECRETARÍA DE EDUCACIÓN PÚBLICA
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SUBDIRECCIÓN ACADÉMICA

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

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

CMAZ/lmz

2026 año de Margarita Maza

DEDICATORIAS

Esta tesis está dedicada a:

A mi madre, por ser mi refugio y apoyo incondicional. Gracias por estar a mi lado en cada desvelo y por creer en mí incluso cuando yo dudaba. Este logro es tan tuyo como mío.

A mi padre, quien desde la eternidad sigue guiando mi camino. Tu recuerdo es la luz que ilumina mis pasos y la fuerza que me impulsa a ser mejor cada día. Te extraño y te honro con este trabajo.

A mi pequeño bebé, mi motor y mi mayor inspiración. Desde que estabas en mi vientre, te convertiste en la razón de mi esfuerzo y en la promesa de un futuro mejor. Eres el regalo más grande que la vida me ha dado.

A mi familia en la distancia, porque los kilómetros no han sido barrera para sentir su amor. Gracias por sus palabras de aliento y por estar presentes en mi corazón.

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.

A la Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI) por el apoyo económico brindado durante la realización de mis estudios de Maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por su orientación técnica y por la libertad que me dio para desarrollar esta investigación. Su experiencia científica y su apoyo personal fueron los pilares necesarios para concluir este trabajo con éxito.

A mi codirector el Dr. Juan Carlos Rojas Pérez, por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

A mis revisores, Dra. Blanca Dina Valenzuela Robles, M.C. Humberto Hernández García y M.C. José Luis Ramírez Alcántara 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 mi familia.

A mis compañeros y amigos que estuvieron conmigo en todo el recorrido de mi carrera, por la convivencia que tuvimos.

RESUMEN

La acumulación de deuda técnica en el software legado se manifiesta principalmente por la necesidad recurrente de modificar el código para corregir defectos y adaptarlo a nuevos requerimientos, situación que incrementa la fragilidad del sistema y acorta significativamente su vida útil. El código mal formado constituye una de las causas principales de este problema, derivado de la omisión de principios de diseño modular y orientado a objetos. El objetivo de la presente investigación fue identificar la recurrencia relativa de las prácticas de diseño y codificación que conducen a la generación de estos defectos, provocando entropía de software y un incremento de la deuda técnica en sistemas legados.

Para ello, se realizó una Revisión Sistemática de la Literatura (RSL) que permitió identificar factores de diseño y programación asociados con código mal formado y un estudio experimental aplicado a proyectos desarrollados en lenguaje Java utilizando metricas estructurales y análisis estadístico.

Los resultados obtenidos, los cuales corresponden a la muestra analizada y a los criterios de medición definidos en la tesis evidenciaron que el 100% de los proyectos analizados, presentan al menos un factor de código mal formado, siendo la carencia de protección, la carencia de abstracción y la distribución incorrecta de responsabilidades los factores de mayor recurrencia relativa. La prueba binomial exacta confirmó con significancia estadística que la gran mayoría de las aplicaciones en operación acumulan deuda técnica. Asimismo, la prueba de Kruskal-Wallis demostró que la distribución de la deuda técnica no presentó diferencias estadísticamente significativas entre los estratos de tamaño analizados.

La correspondencia entre los factores de mayor recurrencia identificados experimentalmente y los códigos mal formados más documentados en la literatura fueron: God Class, Deficient Encapsulation, Primitive Obsession, Feature Envy y Refused Bequest. Dicha relación valida la vigencia de estos defectos en contextos reales de desarrollo y la relevancia de continuar el esfuerzo para disminuir las malas prácticas de desarrollo, mediante la propuestas metodológicas, científicas y tecnológicas.

ABSTRACT

The technical debt accumulation in legacy software is mainly manifested through the recurring need to modify code in order to correct defects and adapt it to new requirements, a situation that increases system fragility and significantly shortens its useful life. The code smell constitutes one of the main causes of this problem, resulting from the omission of modular and object-oriented design principles. The objective of this research was to identify the relative recurrence of design and coding practices that lead to the generation of these defects, causing software entropy and an increase in technical debt in legacy systems.

To achieve this, a Systematic Literature Review (SLR) was conducted to identify design and programming factors associated with code smells. In addition, an experimental study was carried out on software projects developed in Java using structural metrics and statistical analysis.

The results obtained, corresponding to the analyzed sample and the measurement criteria defined in this thesis, revealed that 100% of the projects presented at least one code smell. The most recurrent factors were lack of protection, lack of abstraction, and improper distribution of responsibilities. The exact binomial test statistically confirmed that the vast majority of software applications in operation accumulate technical debt. Furthermore, the Kruskal–Wallis test demonstrated that the distribution of technical debt did not present statistically significant differences among the analyzed project size strata.

The correspondence between the factors with the highest recurrence identified experimentally and the most documented code smells in the literature were: God Class, Deficient Encapsulation, Primitive Obsession, Feature Envy, and Refused Bequest. This relationship validates the persistence of these defects in real-world development contexts and the relevance of continuing efforts to reduce poor development practices through methodological, scientific, and technological proposals.

CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Glosario de Términos

Código Mal Formado (Smell Code)	Código que puede ser difícil de entender, mantener y depurar, el cual es generado por ignorar las mejoras prácticas de programación. Técnicamente, el código mal formado no es incorrecto, no tiene defectos, ni impide el correcto funcionamiento del programa; sin embargo, manifiesta cierta deuda técnica que conduce a su manipulación reiterada y puede generar fragilidad del software (Laobel & Betancourt, 2020).
Deuda Técnica	Es un término que se ha utilizado para describir el costo creciente por cambiar o mantener un sistema de software, debido a los atajos que se toman durante su desarrollo (Alves et al., 2014).
Entropía	Es la medida de desorden del software que refleja la complejidad de su mantenimiento. A medida que se hacen modificaciones o se agrega nuevo código éste va perdiendo su estructura inicial, aumenta su entropía y en consecuencia su fragilidad (Cap de Vila, 2021).
Sistemas legados de software	Son sistemas informáticos que han estado en funcionamiento durante un período prolongado de tiempo y que, aunque pueden haber sido efectivos en el pasado, ahora se consideran obsoletos o ineficientes en comparación con las tecnologías más recientes (Bianchiotti & Casas, 2014).
Diseño Modular	Es una técnica de diseño de software que se basa en dividir un sistema en módulos independientes y bien definidos, cada uno de los cuales tiene una responsabilidad específica y se comunica con otros módulos a través de interfaces claras y definidas (Martin, 2012).
Abstracción	Consiste en la creación de modelos de representación genéricos y representaciones conceptuales de entidades del mundo real que se pueden interpretar en una multitud de modelos específicos (García Llinás, 2010).
Encapsulamiento	Consiste en agrupar los datos (atributos) y los métodos (funciones) que operan sobre esos datos en una sola unidad, y controlar el acceso a

	ellos desde entidades externas de esa unidad (García Llinás, 2010).
Herencia	Es una relación que define una entidad en términos de otra. La herencia de clase define una nueva clase en términos de una o más clases principales. La nueva clase hereda su interfaz e implementación de sus padres. La nueva clase se llama subclase o una clase derivada. La herencia de clase combina la herencia de interfaz y la herencia de implementación (Gamma et al., 2002).
Refactorización	Se define como el proceso de cambiar un sistema de software, de tal forma que no se altere su comportamiento funcional externo del código, aunque se mejore su estructura interna (Fowler et al., 2018).
Patrones de Diseño	Son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Es una solución que se aplica una y otra vez sin hacerlo dos veces de la misma manera, a un problema que se presenta recurrentemente bajo un contexto y sistema de fuerzas específico (Gamma et al., 1994).
Fragilidad del Software	Se refiere a la susceptibilidad de un sistema informático o una aplicación a romperse o experimentar fallas cuando se enfrenta a cambios o situaciones inesperadas. Esta fragilidad puede manifestarse de diversas formas, como errores inesperados, bloqueos, pérdida de datos o mal funcionamiento general del software (Juan, 2016).
Objetos de grano grueso	Se refieren a estructuras o abstracciones que encapsulan funcionalidades amplias o de alto nivel, en contraposición a detalles específicos y de bajo nivel. Estos objetos tienden a manejar operaciones a gran escala y abstracciones generales en lugar de detalles minuciosos (Hernández Montero & Ramón Antunez, 2011).

Capítulo 1. Introducción

Uno de los desafíos más críticos en el campo de la Ingeniería de Software surge cuando los desarrolladores carecen de experiencia y habilidades en la creación de sistemas, lo que con frecuencia conduce a la producción de software o componentes técnicamente deficientes. Este fenómeno se conoce como deuda técnica.

La deuda técnica en los sistemas legados representa el costo acumulado por tomar decisiones de desarrollo apresuradas o incorrectas. En este contexto, las malas prácticas de diseño y codificación actúan como la causa directa que origina el código mal formado. Este código deficiente funciona como un indicador temprano de deuda técnica que, al no ser refactorizado, incrementa la entropía o desorden estructural del software. Como consecuencia, el sistema desarrolla fragilidad, lo que degrada severamente su mantenibilidad a largo plazo y eleva los costos de evolución. Según (Lárez Mata, 2020), la gestión de la deuda técnica debe abordarse desde las etapas iniciales del desarrollo y extenderse a lo largo de la evolución del sistema.

Numerosos estudios han documentado la existencia de factores de código mal formado y su relación con la deuda técnica, la mayoría de estas investigaciones se han centrado en describir los defectos de manera conceptual o en proponer mecanismos de detección y corrección. Sin embargo, existe evidencia limitada sobre la recurrencia relativa de estos factores en proyectos reales y sobre el grado de correspondencia entre los problemas reportados en la literatura científica y aquellos observados experimentalmente en sistemas desarrollados bajo condiciones reales. Esta brecha dificulta determinar cuáles prácticas de diseño y codificación requieren mayor atención por parte de desarrolladores e investigadores.

El propósito de esta investigación es analizar la incidencia de las prácticas de diseño y codificación que conducen a la formación de código mal formado en sistemas legados, contrastando los resultados obtenidos de factores de código mal formado que causan deuda técnica, derivados de la evaluación experimental, contra los factores de código mal formado derivados de la revisión sistemática de la literatura, para determinar el grado de atención de tales problemas en un ámbito internacional.

1.1. Organización de la Tesis

El documento de este trabajo de investigación se compone de nueve capítulos, una sección de referencias y una sección de anexos, como a continuación, se describen de forma general.

Capítulo 1. Introducción
En este capítulo se presenta una breve introducción y se describe la organización general de la tesis.

Capítulo 2. Antecedentes
En este capítulo se describe el planteamiento del problema, la solución propuesta y la justificación técnica de la investigación. Asimismo, se definen los objetivos que guían la investigación y la delimitación del mismo mediante sus alcances y limitaciones. Finalmente, se presenta el estado del arte, el cual integra tanto los antecedentes desarrollados en el laboratorio de Ingeniería de Software del CENIDET como el análisis de trabajos relacionados identificados en la literatura.

Capítulo 3. Marco Teórico
En este capítulo se muestran los conceptos básicos y fundamentos teóricos necesarios para comprender el tema científico y tecnológico que se aborda en esta tesis.

Capítulo 4. Metodología de la RSL
En este capítulo se presenta el diseño metodológico y los resultados de la Revisión Sistemática de la Literatura (RSL) realizada para fundamentar esta investigación.

Capítulo 5. Materiales y Métodos de Solución
En este capítulo se describen las métricas seleccionadas para cuantificar la recurrencia de los factores de código mal formado en los proyectos legados y la metodología a seguir en el Diseño Experimental. Asimismo, se describe el marco estadístico diseñado para el tratamiento de los datos, incluyendo los criterios de normalidad y las pruebas no paramétricas.

Capítulo 6. Diseño Experimental y Plan de Pruebas
En este capítulo se presentan las hipótesis de investigación (nula y alternativa) y se identifican las variables dependientes, independientes e intermitentes involucradas en el análisis de la deuda técnica. Se describe la estructura del plan de pruebas y los criterios de selección de los casos de estudio.

Capítulo 7. Ejecución del experimento y recolección de Datos
En este capítulo se describe el proceso técnico de ejecución del experimento llevado a cabo sobre los proyectos de software legados. Se explica el procedimiento seguido para la extracción de las métricas de software mediante una herramienta de análisis estático, así como la metodología empleada para la captura, organización y sistematización de los datos obtenidos.

Capítulo 8. Análisis Estadístico e interpretación de Resultados
En este capítulo se presenta el procesamiento estadístico de la información recolectada para la validación de las hipótesis. Se detallan los resultados de las pruebas de normalidad (Shapiro-Wilk) y el análisis comparativo mediante las pruebas no paramétricas Binomial Exacta y de Kruskal-Wallis.

Capítulo 9. Conclusiones y Trabajos Futuros
Este capítulo muestra las conclusiones y describe las aportaciones de la investigación y los trabajos futuros.

Capítulo 2. Antecedentes

2.1. Planteamiento del Problema

La acumulación de deuda técnica en el software legado, se manifiesta por la necesidad recurrente de manipular el código para la eliminación de defectos o para la incorporación de nuevos requerimientos o cambios a los requerimientos iniciales, lo cual acorta el tiempo de vida del software por la fragilidad adquirida, como efecto de la frecuente necesidad de corregirlo. Los códigos mal formados[1] representan la situación causal de este problema en el desarrollo de software, por lo tanto, es fundamental abordar esta situación mediante prácticas adecuadas tanto de diseño como de codificación para mejorar la calidad y legibilidad del código en todo momento.
Entre otras prácticas de desarrollo que contribuyen a la generación de código mal formado se encuentran: la carencia de abstracción, el encapsulamiento inadecuado debido a la falta de protección de la información, así como las dependencias excesivas entre entidades de software. Asimismo, el uso inadecuado de la herencia de implementación y la distribución incorrecta de responsabilidades debido a la falta de atención a los criterios y principios de diseño modular y orientado a objetos y a la omisión de considerar los factores de calidad.

2.2. Solución Propuesta

En este proyecto de tesis de maestría se proponen dos pasos a seguir:

1. Realizar una revisión sistemática de la literatura (SRL) para identificar los factores de la práctica de diseño de arquitecturas de software y codificación que se mencionan con más frecuencia en la literatura y que originan código mal formado, así como los métodos que se han propuesto para solventar la deuda técnica.

2. Diseñar y realizar un experimento con el objetivo de determinar desde los cinco métodos de refactorización referidos en la sección de Antecedentes, aquellos factores que causan deuda técnica y que se presentan con mayor frecuencia en código legado. El diseño experimental deberá plantear las Hipótesis de Investigación y Alternativas, así como las variables dependientes e independientes; los casos de prueba normalizados, instrumentos de pruebas; y deberá concluirse mediante la interpretación de resultados a través del análisis estadístico apropiado.

2.3. Justificación

Cuando las prácticas de diseño y codificación de software omiten la aplicación de principios modulares y orientados a objetos, así como la consideración de métricas de calidad interna, pueden generarse estructuras de código deficientes conocidas como códigos mal formados. Estos defectos constituyen factores frecuentemente asociados a la acumulación de deuda técnica, aunque esta también puede originarse por decisiones arquitectónicas, restricciones de tiempo, cambios en los requisitos, obsolescencia tecnológica o factores organizacionales. La presencia de códigos mal

[1] En este documento la frase "código mal formado" es una interpretación en español al término en inglés "code smell". Se refiere al código funcional incorrecto o aquel que causa entropía de software y deuda técnica.

formados incrementa la complejidad y fragilidad del sistema, reduciendo su mantenibilidad al requerir modificaciones recurrentes para corregir defectos o incorporar nuevos requerimientos o cambios en los requerimientos iniciales.
En esta tesis, la identificación de la recurrencia de códigos mal formados se enfoca en cinco factores críticos: la carencia de abstracción, el encapsulamiento inadecuado por falta de protección, las dependencias excesivas (alto acoplamiento), el uso incorrecto de la herencia de implementación y la distribución deficiente de responsabilidades. El interés por estudiar estos factores radica en que su persistencia puede contribuir significativamente al incremento de la deuda técnica y a la degradación progresiva de la calidad estructural del software, elevando los costos y riesgos asociados a su mantenimiento y evolución.
Finalmente, este trabajo de tesis aporta evidencia empírica para la línea de investigación del laboratorio de Ingeniería de Software del CENIDET. Mientras que proyectos previos han desarrollado métodos automáticos de refactorización incorporados a herramientas como SR2-Refactoring para atender problemas de acoplamiento, protección y abstracción, esta investigación mide y evalúa la frecuencia con la que dichos factores aparecen en sistemas legados. Al identificar cuáles factores de código mal formado presentan una mayor recurrencia relativa y una mayor asociación con la deuda técnica, se proporciona información que puede orientar la priorización de procesos de refactorización y el desarrollo de futuras soluciones automáticas para mejorar la calidad y sostenibilidad de los sistemas de software.

2.4. Objetivo General

En este contexto, el objetivo general de esta tesis en el área de Ingeniería de Software es determinar la recurrencia relativa de la práctica de desarrolladores en las etapas de diseño y codificación que conducen a código mal formado que origina deuda técnica o entropía de software en sistemas legados.

2.4.1. Objetivos Específicos

1) Determinar la recurrencia relativa de los factores de código mal formado mediante un diseño experimental aplicado a una muestra de proyectos de software legado escritos en Java, utilizando un marco de métricas de calidad estructural y análisis estadístico.

2) Establecer el grado de atención que han recibido, en otros laboratorios de investigación, los factores de código mal formado asociadas a la carencia de abstracción, falta de protección de la información, alto acoplamiento entre entidades de software, uso inadecuado de la herencia de implementación y distribución incorrecta de responsabilidades, a partir de los resultados de una Revisión Sistemática de la Literatura.

2.5. Alcances

1. El estudio experimental se aplicó a una muestra de 29 proyectos de software derivada del cálculo de muestreo con una población total de 40 proyectos escrito en lenguaje

Java versión 8.5, tomados aleatoriamente de repositorios externos disponibles a través de internet.
2. La clasificación de los proyectos como software legado se realizó desde una perspectiva operacional orientada al análisis estructural del código, sin considerar variables como antigüedad, actividad del repositorio o grado de obsolescencia tecnológica.
3. Los criterios de selección de trabajos relacionados se realizaron en términos de los últimos cinco años, escritos en lenguaje java. Se excluyeron trabajos de tesis, monografías, artículos blancos y memorias de congresos.

2.6. Limitaciones

Quedando fuera del alcance del proyecto:

1. Proyectos de software no escritos en lenguaje Java, o de otras versiones a Java 8.5.
2. La cantidad máxima de patrones de la práctica que se determinaron en este trabajo de tesis estuvo sujeta a lo contenido en los 29 proyectos de software legado bajo estudio o a los patrones de la práctica emanados del estudio de la literatura especializada; pretendiendo que se determinarán al menos tres patrones de la práctica utilizada.

2.7. Estado del Arte

2.7.1. Antecedentes

En la especialidad de Ingeniería de Software del Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) se desarrollaron varios trabajos afines a la línea de investigación de esta tesis. A continuación, se hace una breve descripción de las diferentes investigaciones.

1. Año 2004. Reestructuración de software escrito por procedimientos conducido por patrones de diseño composicionales. M.C. Armando Méndez Morales (Méndez Morales, 2004)**.**
Este trabajo tuvo como objetivo la restructuración de código escrito en C a través de la implementación de marcos de aplicaciones orientados a objetos que incorporan la arquitectura de patrones de diseño, con el fin de obtener segmentos genéricos de código reusables, que pueden ser ejercitados e incorporados en depósitos de componentes para su reuso en aplicaciones posteriores, además de repercutir en la mejor calidad del sistema y en la simplificación de las tareas de mantenimiento y reuso.

2. Año 2004. Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator. M.C. Leonor Adriana Cárdenas Robledo (Cárdenas Robledo, 2004).

En este trabajo de investigación se plantea un método de refactorización para reducir el acoplamiento entre clases, debido a muchas relaciones de asociación y/o dependencia, lo que se traduce en muchos canales de comunicación entre clases en sistemas de

Software existente escrito en lenguaje C++. Este método de refactorización es conducido según la intención del patrón de diseño “Mediator”, el cual incorpora en la arquitectura resultante la estructura de clases como se describe en la solución de este patrón de diseño en su solución. La refactorización del método funciona automáticamente y fue incorporado a la herramienta SR2 Refactoring. En este proyecto también se diseñó e implementó la métrica orientada a objetos que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF).

3. **Año 2004. Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces. M.C. Manuel Alejandro Valdés Romero** (Valdés Romero, 2004).

En ese proyecto de investigación se desarrolló un método refactorización denominado “Separación de Interfaces”, cuyo algoritmo fue implementado satisfactoriamente en la herramienta SR2 Refactoring. Este método realiza la refactorización de manera automática de código de marcos orientados a objetos escritos en C++, con el propósito de reducir el número de dependencias de herencia de interfaz entre clases, debido a la implementación nula de las interfaces en clases derivadas que son dispensables. En ese proyecto también se diseñó e implementó la métrica orientada a objetos denominada “V-DINO” para medir el grado de dependencia debido a interfaces que no se ocupan.

4. **Año 2019. Método de Re-factorización de código java con interfaces y abstracciones incorrectas. M.C. Pablo Padilla Salgado** (Padilla Salgado, 2019).

En ese trabajo de tesis se aplica el “principio de separación de interfaces”, automatizando su solución, para dividir las responsabilidades entre clases. El método localiza estructuras de código desagradable[2], tales como: abstracciones incorrectas que no respetan los principios de diseño de “sustitución” y de “única responsabilidad” en aplicaciones existentes escritas en lenguaje Java. Las cuales corrige automáticamente. El método anula las implementaciones nulas de funciones en clases derivadas y balancea las jerarquías de herencia tanto en lo vertical (clases descendientes) como en lo horizontal (clases derivadas hermanas) para distribuir las responsabilidades entre las diferentes clases de una arquitectura Orientada a Objetos.

5. Año 2020. Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente. M.C. Nélida Barón Pérez (Barón Pérez, 2020).

El método desarrollado en esta investigación tiene como objetivo aumentar la protección funcional en módulos de arquitecturas de clases. Para lograr este objetivo es necesario que cada atributo, así como cada función integrada en las clases de objetos de arquitecturas de software, presenten el calificador de alcance correcto. Este método fue integrado a la herramienta SR2-Refactoring y funciona para marcos de aplicaciones orientados a objetos escritos en lenguaje Java.

[2] Código mal formado o code smell

6. Año 2020. Re-Factorización de Código para Reducir el Acoplamiento entre Clases relacionadas por Herencia de Implementación en Arquitecturas Orientadas a Objetos. M.C. Orlando Ortiz Gutiérrez (Ortiz Gutiérrez, 2020).

En ese trabajo de tesis se diseñó e implementó un método de refactorización que disminuye el acoplamiento por herencia de implementación en arquitecturas de clases orientadas a objetos de sistemas de software existentes y que están escritos en lenguaje Java. El propósito del método es reducir la interdependencia entre objetos y clases por herencia de implementación. El método incluye cinco métricas de calidad, tres de ellas miden el factor de herencia de implementación y las dos restantes miden el factor de flexibilidad de aplicaciones existentes orientadas a objetos.

7. **Año 2022. Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos. M.C. Marisol Ramírez Cruz** (Ramírez Cruz, 2022).

En este trabajo de tesis se describe el desarrollo de dos métodos de re-factorización: el primero aporta un avance en la búsqueda de la mejora de modularidad con el aumento de la coherencia y la cohesión, de clases de objetos, el segundo método con la finalidad de reducir las dependencias entre clases de objetos en sistemas desarrollados en lenguaje Java. Estos métodos permiten extender la funcionalidad del sistema SR2–Refactoring agregando nuevos métodos para re-factorizar código.

8. Año 2023. Re-factorización de Módulos Colaborativos con Carencia de Abstracción. M.C. Fernando Sánchez Rogel (Sánchez Rogel, 2023).

En este trabajo se planteó el “Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción”. El método fue implementado en lenguaje Java para automatizar el proceso de refactorización. Se aplican las métricas de Abstracción y Factor de Polimorfismo para evaluar el grado de abstracción y de las situaciones polimórficas que contiene un proyecto o sistema de software existente en uso. Este método de refactorización tiene como objetivo principal identificar la carencia de abstracción en el diseño de arquitecturas modulares de software mediante un análisis estático de abajo hacia arriba (del código escrito en lenguaje Java hacia el diseño), con el fin de cambiar clases base y clases abstractas por interfaces.

9. Año 2023. Tratamiento de la Deuda Técnica Originada por la Carencia de Protección de Funciones Plantilla de Software Legado. M.C. Elías Alejandro Ramírez García (Ramírez García, 2023).

En este proyecto de investigación se desarrolló un método de refactorización para mejorar la protección de las funciones plantilla, funciones variantes y funciones invariantes asociadas al patrón de diseño “Template Method”. Así como su implementación en un sistema escrito en Java para su automatización. Para evaluar el funcionamiento del método de refactorización se desarrollaron cuatro métricas para medir la protección modular del patrón de diseño “Template Method”: PMMP (Protección Modular de Métodos Plantilla), PMFV (Protección Modular de Funciones Variantes), PMFI (Protección Modular de Funciones Invariantes) y PTTM (Protección Total del Template Method).

10. **Año 2023. Disminución de deuda técnica producida por arquitectura de clases con más de una responsabilidad. M.C. Juan Antonio Díaz Díaz** (Díaz, 2023).

En este trabajo de investigación, se propone un método de refactorización denominado “Método de Refactorización de Separación de Responsabilidades”, cuyo objetivo es separar las responsabilidades existentes en arquitecturas modulares de programas, escritos bajo el lenguaje Java, dejando cada uno de ellos en la mejor manera posible con una única responsabilidad. Se propone una métrica de calidad, la cual consiste en medir el número de responsabilidades: Número de Responsabilidades (NR).

2.7.2. Trabajos Relacionados

A partir de la revisión bibliográfica realizada y siguiendo la metodología de Kitchenham, el proceso de búsqueda y selección se ejecutó en bases de datos de alto impacto, específicamente Scopus e IEEE Xplore, cubriendo un periodo de estudio de 2016 a 2024. La gestión y depuración de los artículos se llevó a cabo manualmente, donde se aplicaron criterios de inclusión y exclusión rigurosos para garantizar la relevancia metodológica de las fuentes.
A continuación se presenta un breve resumen de los artículos seleccionados.

A User Feedback Centric Approach for Detecting and Mitigating God Class Code Smell Using Frequent Usage Patterns (Singh et al., 2019)**.**
Este trabajo describe un método para detectar y corregir las clases Dios (*God Class*) integrando métricas dinámicas con retroalimentación de los desarrolladores. Utiliza patrones de uso frecuente (FUP) y dos métricas específicas: AUR (uso de atributos ajenos) y COUPGC (acoplamiento entre clases). El sistema aplica umbrales dinámicos para identificar el defecto y sugiere la refactorización de tipo "Extraer Clase" para mejorar la estructura del código.

A robust multi-objective approach to balance severity and importance of refactoring opportunities (Mkaouer et al., 2017)**.**
Esta investigación presenta un modelo de optimización basado en algoritmos genéticos para planificar la refactorización de código. El enfoque busca equilibrar tres factores: la mejora de la calidad, la severidad de los códigos mal formados y la importancia de las clases afectadas. El modelo está diseñado para trabajar bajo condiciones de incertidumbre, permitiendo priorizar qué partes del código corregir primero según las necesidades del proyecto.

An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity (Akash et al., 2019)**.**

El trabajo propone una técnica que consiste en el uso de Latent Dirichlet Allocation (LDA) para el análisis semántico de los métodos y considera el acceso a variables compartidas y llamadas entre métodos para la parte estructural. Esta técnica se utilizó para dividir clases grandes (God Class) en componentes más pequeños y cohesivos. Para ello, analiza la similitud estructural (métricas de código) y semántica (temas en los nombres de métodos y variables). Utiliza algoritmos de agrupamiento (clustering) para identificar qué métodos deben permanecer juntos y cuáles deben moverse a nuevas clases independientes.

An Approach to Suggest Code Smell Order for Refactoring (Guggulothu & Moiz, 2019).
Este estudio investiga la secuencia más eficiente para eliminar códigos mal formados con el fin de reducir el esfuerzo de los desarrolladores. Utiliza aprendizaje automático para seleccionar las métricas más importantes de cada defecto. El análisis determina dependencias entre códigos mal formados , sugiriendo, por ejemplo, que corregir una clase de datos (*Data Class*) antes que una clase Dios (*God Class*) facilita la limpieza general del sistema.
An Automated Code Smell and Anti-Pattern Detection Approach (Bacon et al., 2017).
El informe detalla el funcionamiento de la herramienta Y-CSD para la detección automática de *Data Class* y *Brain Method*. El método se basa en el uso de umbrales dinámicos calculados a partir de un conjunto de datos de entrenamiento. Emplea métricas estándar como líneas de código (LOC), complejidad ciclomática (CYCLO) y número de métodos de acceso (NOAM) para mejorar la precisión de la detección frente a herramientas con umbrales fijos.

An Efficient Design Smell Detection Approach with Inter-class Relation (Zhu et al., 2023)**.**
Esta investigación presenta el enfoque DSIR para detectar defectos de diseño complejos analizando las relaciones entre clases. Utiliza redes neuronales (Bi-LSTM) para procesar el texto del código y redes gráficas (R-GCN) para entender cómo interactúan los módulos en los diagramas UML. El objetivo es identificar fallos estructurales que no se detectan analizando las clases de forma aislada.

Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability (Mehta et al., 2018).

Este trabajo evalúa mediante experimentos cómo el orden en que se quitan los códigos mal formados afecta la mantenibilidad del software. Utiliza métricas de complejidad y el Índice de Mantenibilidad (MI) para medir los resultados. El estudio concluye que seguir una secuencia específica de refactorización (por ejemplo, empezar por *Long Method* produce mejores resultados en la calidad final del sistema.

Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Prediction? An Empirical Study (Liu et al., 2018)**.**

El estudio analiza si los códigos mal formados sirven para predecir qué archivos del código cambiarán más en el futuro. La metodología compara métricas basadas en códigos mal formados contra las métricas tradicionales (CK) e incluye una medición del esfuerzo de inspección. Los resultados muestran que los códigos mal formados son indicadores más precisos para identificar partes del código propensas a cambios estructurales.

Code smells incidence: does it depend on the application domain? (Paulk, 2016)
Este estudio evalúa si el dominio de una aplicación influye en la frecuencia de aparición de factores de código mal formado. Mediante un análisis estadístico en 118 aplicaciones Java, los resultados demuestran que la incidencia de la mayoría de los código mal formados es independiente del dominio funcional, con excepción del código duplicado, cuya presencia es significativamente mayor en aplicaciones de educación.

Improving Machine Learning-based Code Smell Detection via Hyper-parameter Optimization (Shen et al., 2020)

La investigación propone mejorar la detección automática de factores de código mal formado mediante la optimización de hiperparámetros en modelos de aprendizaje automático. El análisis demuestra que el ajuste preciso de los algoritmos de clasificación incrementa la efectividad de la detección frente a los métodos convencionales basados en reglas manuales, logrando una mayor precisión en la identificación de los código mal formados *Data Class* y *Feature Envy*.

MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells (Ouni et al., 2017)
Este trabajo presenta un método multiobjetivo para automatizar la refactorización de software mediante la integración de patrones de diseño. El enfoque permite eliminar de manera sistemática factores de código mal formado mientras se introducen estructuras de diseño robustas, optimizando la mantenibilidad del sistema y asegurando la preservación de la semántica funcional del código fuente.

Resource Allocation Modeling Framework to Refactor Software Design Smells (Gupta et al., 2023).
Esta investigación propone un marco matemático para gestionar los recursos (tiempo y presupuesto) durante la refactorización. Utiliza procesos estadísticos de *Poisson* para modelar cuántos defectos se pueden corregir y optimiza la asignación de esfuerzo según el tipo de código mal formado. El modelo ayuda a decidir cuánto presupuesto invertir en cada categoría de defecto para maximizar la calidad del software.

En la Tabla 1, se presenta una breve reseña de los trabajos que mantienen una relación directa y estrecha con los objetivos de esta investigación. Para este análisis comparativo, se aplicó un filtro de relevancia metodológica con el fin de que aporte valor directo al desarrollo del estudio. Esta selección permitió que la comparación se centrara exclusivamente en investigaciones primarias que utilizan métricas cuantitativas para la detección de código mal formado y miden su frecuencia relativa.

Tabla 1: *Comparativa de Trabajos Relacionados*

Trabajo de Investigación	Objetivo Principal	¿Usa Métricas?	¿Mide Frecuencia?	Alcance
(Singh et al., 2019)	Detectar *God Class* mediante feedback y patrones.	Si (AUR, COUP).	No	Detección y mitigación en Java.
(Mkaouer et al., 2017)	Equilibrar severidad e importancia en refactorización.	Si (Calidad).	Si (Código mal formado corregidos vs. detectados**).**	Recomendación automática.
(Akash et al., 2019)	Descomponer *God Class* por similitud estructural.	Si (LCOM, C3).	No	Refactorización en Java.
(Guggulothu & Moiz, 2019)	Sugerir orden óptimo de eliminación de defectos.	Si (Gain Ratio).	Si (Distribución de código mal formado en datasets de ref./Fontana).	Optimización del esfuerzo.

Trabajo de Investigación	Objetivo Principal	¿Usa Métricas?	¿Mide Frecuencia?	Alcance
(Bacon et al., 2017)	Detección automatizada con umbrales dinámicos.	Si (LOC, CYCLO).	No	Herramienta Y-CSD.
(Zhu et al., 2023)	Detectar defectos de diseño usando redes gráficas.	Si (P, R, F1).	Si (Tasa de detección de defectos de diseño en modelos UML)	Alta granularidad con DL.
(Mehta et al., 2018)	Maximizar mantenibilidad según secuencia de eliminación.	Si (MI, CC).	Si (Conteos específicos: 1077 *Long Method* y 242 *God Class*).	Proyectos pequeños y grandes.
(Liu et al., 2018)	Predecir archivos propensos a cambios.	Si (ANOS, SCM).	Si (Presencia y ocurrencia de 14 tipos de código mal formado estudiados).	Predicción estructural.
(Paulk, 2016)	Analizar el impacto del dominio en los defectos.	Si (Complejidad).	Si (Densidad de código mal formado por cada mil líneas de código - KLOC).	Calidad según contexto de uso.
(Shen et al., 2020)	Optimizar hiperparámetros para detección.	Si (AUC).	Si (Ratio entre instancias con y sin código mal formado en el dataset).	Mejora de clasificadores ML.
(Ouni et al., 2017)	Introducir patrones y corregir defectos.	Si (CCR, QG).	Si (Frecuencia de aparición de 7 tipos de código mal formado comunes)	Recomendación estratégica.
(Gupta et al., 2023)	Optimizar asignación de recursos para refactorizar.	Si (*Poisson*).	Si (Porcentaje de código mal formados efectivamente corregido).	Gestión de recursos de calidad.
Esta Tesis	Determinar la recurrencia relativa de la práctica de desarrolladores en las etapas de diseño y codificación que conducen a código mal formado.	Si (TPM, FA, FHIAC, NR, COF).	Si (Frecuencia Relativa de código mal formado).	Detección automática y análisis de factores de código mal formado.

Capítulo 3. Marco Teórico

3.1. Paradigma de la Programación Orientado a Objetos

El paradigma orientado a objetos es una metodología de programación en el cual se modela de forma análoga a la realidad, simplificando el diseño de alto nivel y permitiendo facilidad en el mantenimiento y reúso del software. En la programación orientada a objetos (POO) se definen entidades virtuales llamadas objetos, éstos simulan a otros objetos de la realidad que tienen un estado y un comportamiento (operaciones) dado. A nivel computacional los estados son llamados “atributos” y al comportamiento se le llama “métodos”.
Para crear objetos es necesario hacer uso de clases, una clase es un molde donde se definen las propiedades comunes de un conjunto de objetos. Los objetos son instancias de una clase (Cervantes O et al., 2016).

Clase

Una clase es una plantilla para un objeto. Esta encapsula los datos (variables o campos) y de funciones (métodos) que operan sobre esos datos (Morelli & Walde, 2016).
Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones cuya ejecución conoce; dichas acciones se conocen como servicios o métodos. Una clase también incluye todos los datos necesarios para describir los objetos creados a partir de la clase. Los cuales se conocen como atributos, variables o variables instancia; el primer término se utiliza en análisis y diseño orientado a objetos, los otros, en programación orientada a objetos (Joyanes Aguilar & Zahonero Martínez, 2007).
La declaración de una clase consta de palabras reservadas e identificadores: una secuencia opcional de modificadores, la palabra reservada class, el nombre de la clase, un nombre opcional de la clase padre, una secuencia opcional de interfaces y el cuerpo de la clase con sus miembros, en la Figura 1 se muestra un ejemplo de declaración de una clase.

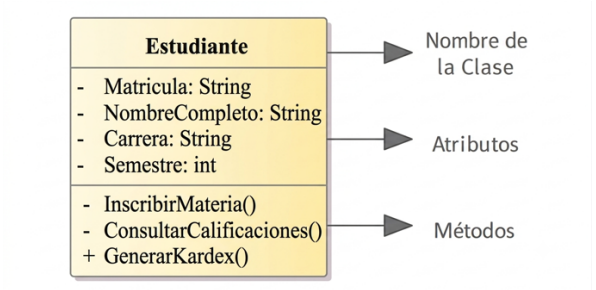

Figura 1: *Modelo Conceptual de una clase*

Objetos

Los objetos son sujetos o cosas tangibles de nuestro mundo real que tienen un estado y un comportamiento. En POO, un objeto está compuesto por datos y métodos. Estos objetos tienen la estructura y el comportamiento definido por la clase, y muestra el comportamiento reflejado por sus operaciones (métodos), que actúan sobre sus datos,
Los métodos son el único medio para acceder a los datos del objeto; no es posible hacerlo directamente. Los datos están ocultos y eso asegura que no puedan modificarse

por accidente por métodos externos al objeto (Velarde de Barraza et al., 2006), en la Figura 2 se muestra un ejemplo de objeto.

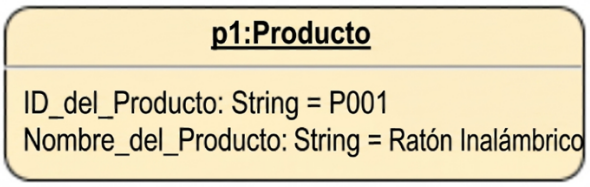

Figura 2: *Ejemplo de Objecto p1 de la clase Producto*

Atributos

Los atributos son las características individuales que diferencian a un objeto de otro; el conjunto de ellos constituye su estado; cada objeto almacena información acerca de su estado actual y en un momento dado éste corresponde a una selección determinada de valores posibles de los diversos atributos.
El estado de un objeto puede cambiar con el tiempo como consecuencia de llamadas a métodos. Es una clase, los atributos se definen mediante variables, las cuales son sitios donde se almacena la información de un programa de computadoras (Velarde de Barraza et al., 2006).

Métodos

Un método (función) es un conjunto de instrucciones o sentencias que realizan una determinada tarea, los cuales se encuentran definidos dentro de una clase; estos se identifican con un nombre y puede o no devolver un valor. En los lenguajes orientados a objetos, los métodos son operaciones o servicios que describen un comportamiento asociado a un objeto.
El comportamiento de un objeto es el conjunto de capacidades y aptitudes que describen sus operaciones, funciones y reacciones; además responde a lo que se puede hacer con dicho objeto o a los métodos que se le pueden aplicar.
Cuando se llama a un método, el flujo de control del programa se transfiere al método y se ejecutan una a una las instrucciones que lo integran; cuando se ejecutan todas, el control regresa al punto desde donde se hizo la llamada y posteriormente el programa continúa con la siguiente sentencia o instrucción (Velarde de Barraza et al., 2006).

Calificadores de Alcance

Un principio fundamental en programación orientada a objetos es la ocultación de la información; esto significa que no se puede acceder por métodos externos a la clase a determinados datos internos.
Para controlar el acceso a los miembros de la clase se utilizan diferentes calificadores de acceso: *default, privados, públicos y protegidos*, y se pueden aplicar a los atributos (datos) y métodos de una clase. A continuación, se especifica con más detalle cada uno de los modos de acceso (Velarde de Barraza et al., 2006):
• Miembros default (*friendly* ~). Es el valor por defecto si no se indica alguno, solo las clases en el mismo paquete pueden acceder a la propiedad o método.
• Miembros *privados* (-). Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos.

• Miembros *públicos* (+). Son aquellos a los que puede accederse desde dentro de la clase y también por parte del objeto.
• Miembros *protegidos* (#). Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos. Es importante aclarar que este tipo de acceso se asocia directamente con la herencia.

3.2. Principios de Diseño Orientado a Objetos

Como parte de las buenas prácticas de desarrollo de software, se han establecido diversos principios que sirven como guía para la construcción de sistemas informáticos mantenibles y adaptables a futuros cambios. Estos principios proporcionan recomendaciones que ayudan a los desarrolladores a mejorar la calidad estructural del código, reduciendo la presencia de código mal formado.

Principio de Responsabilidad Única

El principio de diseño de única responsabilidad conocido como “Single Responsibility Principle (SRP)” establece que una clase o módulo debe tener una, y sólo una razón para cambiar (C. Martin, 2018).
Durante el trabajo de esta investigación se trata específicamente una causa que origina deuda técnica en el software existente, debido al código mal formado en arquitecturas modulares que exhiben más de un motivo para cambios porque no son conformes con el principio de diseño de “Única Responsabilidad”.

Principio Abierto/Cerrado

El término del principio Abierto-Cerrado, según (Meyer, 1997), dice que: “Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertos para la extensión, pero cerrados para las modificaciones” Si existe un solo cambio en un programa que da como resultado una cascada de cambios en los módulos dependientes, el diseño se vuelve rígido. Este principio, nos aconseja refactorizar el sistema para que más cambios de ese tipo no provoquen las modificaciones.

Principio de Sustitución de Liskov

El Principio de Sustitución de Liskov establece que los objetos de una subclase deben poder sustituir a los objetos de su superclase sin alterar el comportamiento correcto del programa (Liskov & Wing, 1994),. La violación de este principio suele manifestarse mediante un uso incorrecto de la herencia, generando diseños frágiles y difíciles de mantener.

Principio de Separación de Interfaces

El Principio de Separación de Interfaces establece que una clase no debe depender de métodos que no utiliza. Para ello, se recomienda definir interfaces pequeñas y específicas en lugar de interfaces generales con múltiples responsabilidades. Su aplicación favorece diseños más cohesionados, flexibles y fáciles de mantener (Martin, 2002).

Principio de Inversión de Dependencias

El Principio de Inversión de Dependencias establece que los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. Asimismo, las abstracciones no deben depender de los detalles de implementación, sino que los detalles deben depender de las abstracciones. Este principio reduce el acoplamiento entre componentes y facilita la extensibilidad del software (Martin, 2002).

3.3. Software Legado

El software legado, también conocido como software heredado o existente, se refiere a programas antiguos que fueron desarrollados en tiempos anteriores, que aún están en operación y que requieren modificaciones frecuentes para satisfacer los cambios en los requerimientos de los negocios y plataformas de computación. Estos sistemas pueden presentar mala calidad y son costosos de mantener y evolucionar.
La evolución del software legado ocurre debido a la necesidad de adaptarse a los nuevos ambientes del cómputo y la tecnología, implementar nuevos requerimientos del negocio, operar con otros sistemas o bases de datos modernos y rediseñar la arquitectura del software para hacerla viable dentro de un ambiente de redes (Santaolaya Salgado, 2021).

3.4. Código mal formado (Code smell)

Se refiere al código fuente de un programa de computadora, que no se implementa de la mejor manera, cumple con sus objetivos o metas de valor, y puede tener alguna fragilidad, rigidez y otros problemas. El código mal formado consiste en ciertas estructuras de código que violan los principios básicos de diseño y afectan negativamente la calidad de su diseño. Técnicamente hablando, el código mal formado no es incorrecto, no son defectos o programas de bloqueo. Por el contrario, indica que hay una cierta debilidad en el diseño, lo que reduce su desarrollo y aumenta el riesgo de falla (Fowler, 1999).

3.5. Deuda Técnica

Es una metáfora que describe las concesiones entre las actividades de desarrollo, que son demoradas para conseguir el pago de compensaciones en el corto plazo, como lo es una versión del software a tiempo. El término "deuda" es usado para describir como esas actividades pueden ser saldadas en el corto plazo. Si se acumula demasiada deuda técnica en el proyecto de software, se retrasará el desarrollo, por ejemplo, por incrementar el pobre mantenimiento del código (N. Zazworka et al., 2011).

3.6. Marco orientado a objetos

Un marco orientado a objetos (conocidos como frameworks en la literatura escrita en lenguaje inglés) se define como un conjunto semi-completo de clases en colaboración que incorpora un diseño genérico, el cual puede adaptarse a una variedad de problemas específicos para producir nuevas aplicaciones hechas a la medida (René, 2003).

3.7. Teoría de la Medición

La medición sirve como una forma útil de planificar y controlar de manera sólida el desarrollo y la ejecución de proyectos de software durante el ciclo de vida del mismo. Sin embargo, al realizar mediciones, es importante conocer los niveles de escalas del proceso de medición.

La medición puede definirse como la asignación de números a objetos y eventos de acuerdo con ciertas reglas; la manera como se asignan esos números determina el tipo de escala de medición (Stevens, 1946).

Los números se asignan a los individuos de acuerdo con un procedimiento repetible cuidadosamente prescrito. La teoría de la medición es una rama de la estadística aplicada que intenta describir, categorizar y evaluar la calidad de las mediciones, mejorar la utilidad, la precisión y el significado (J. Allen & M. Yen, 1979).

Esto conduce a la existencia de diferentes tipos de escalas, por lo que el problema se transforma en explicar a) las reglas para asignar números, b) las propiedades matemáticas de las escalas resultantes, y c) las operaciones estadísticas aplicables a las medidas hechas con cada tipo de escala.

Conceptos básicos de la Teoría de la medición

Para las mediciones se pueden considerar dos sistemas relacionales: el empírico y el sistema relacional formal. Como primer punto se introducen algunas definiciones básicas; notación empírica, sistema relacional binario, medida y escala, definidas en (Briand et al., 1996; Zuse & Bollmann-Sdorra, 1992).

$$A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$$

Sea un sistema relacional empírico, donde A es un conjunto de objetos no vacío, R_i son relaciones ki-arias en A donde i = 1, ... , n y o_j, j = 1, ... , m son operaciones binarias cerradas en A. Por ejemplo:

Sistema Relacional Empírico: $A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$

$\boldsymbol{A}$ Es un conjunto no vacío de objetos empíricos que se van a medir (en nuestro caso, arquitecturas modulares).
$\boldsymbol{R_i}$ Son relaciones ki-arias en A con i =1, ..., n, por ejemplo, la relación empírica “menor o igual que”
$\boldsymbol{o_j}$ Son operaciones binarias en objetos empíricos de A que van a ser medidos (por ejemplo, secuencia de métodos) con j =1, ..., m.

$$B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$$

Sea un sistema relacional formal, donde B es un conjunto no vacío de objetos formales, por ejemplo; números o vectores, S_1, i = 1, ... , n son relaciones ki-arias en B y $\bullet_j$, j = 1, ... , m son operaciones binarias cerradas en B. Por ejemplo:

Sistema Relacional Formal: $B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$

B Es un conjunto no vacío de objetos formales (por ejemplo, números o vectores).
S_1 Son relaciones ki-arias en B con i=1, ..., n, (por ejemplo, “mayor que”, “menor o igual que”
$\bullet_j$ Son operaciones binarias cerradas en B que van a ser medidos (por ejemplo, suma o multiplicación).

Una medida μ (la letra griega "mu") se define como un mapeo μ: $A \rightarrow B$, que produce para cada objeto empírico $a \in B$ un objeto formal (valor de medición) μ(a)∈ B
Sea $A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$ un sistema relacional empírico y $B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$ un sistema relacional formal y μ una medida. La tripleta (A, B, μ) es una escala si y solo si para todo i, j y para todo a_1, ... , a_k, a, $b \in A$ sostiene que:

$$R_i\ (a_1, \dots, a_k) \Leftrightarrow \text{Si}(\mu(a_1), \dots, \mu(a_k))$$
and
$$\mu(a\ o_j\ b) = \mu(a) \bullet_j \mu(b)$$

Si $B = R$ es el conjunto de los números reales, la Tripleta (A, B, μ) es una escala real.

Escalas o niveles de medición

La medición puede ocurrir en diferentes niveles y la relación entre los valores asignados determina el nivel de medición.
Hasta la fecha, la mayoría de los psicólogos, científicos sociales e ingenieros suscriben los cuatro niveles jerárquicos de medición identificador por (Stevens, 1946): nominal, ordinal, de intervalo y de razón. Cada nivel de medición específica cómo los números que se asignan a los individuos se relacionan con la propiedad que se mide. Las primeras dos escalas (nominal y ordinal) son conocidas como escalas categóricas, y las dos últimas (intervalo y razón) como escalas numéricas.

Escala nominal

Representa la asignación de numerales menos restringida, la cual está compuesta por conjuntos de categorías en las que se clasifican los objetos. Dichos numerales son utilizados solo como etiquetas, además de utilizar palabras y letras. Una escala nominal es simplemente una colocación de datos en categorías, sin ningún orden o estructura.
Dentro de la Ingeniería de Software, un lenguaje de programación utilizado para un programa es una medida nominal, ya que permite que los programas se clasifiquen en diferentes categorías y no se disponga de ningún orden entre ellos.
Las escalas nominales son las menos informativas, pero pueden proporcionar información importante, por ejemplo, el lenguaje de programación utilizado es fundamental para interpretar modelos construidos sobre la cantidad de líneas de código.

Escala ordinal

Surge de la operación de ordenación de rangos. Se asignan números a los objetos en un orden particular, pero cualquier número que mantenga ese orden es igualmente bueno. Cualquier función creciente t es una transformación admisible.
De acuerdo con (Zuse & Bollmann-Sdorra, 1992) para poder clasificar una métrica como una escala ordinal, se debe de cumplir con el axioma de orden débil, el cual consiste en una relación binaria que debe de ser transitiva y completa:

Transitividad: P ≤• P′, P′ ≤• P″ ⇒ P ≤• P″
Completes: P ≤• P′ o P′ ≤• P

Para toda P′, P″ε P, donde P es un conjunto y ≤• es una relación empírica binaria como “igual o menos compleja que”.
Supóngase que (P, ≤•) es un sistema relacional empírico, donde P es un conjunto contable no vacío y ≤• es una relación binaria en P. Tenemos una función μ: P → R, con:

$$P' \leq\bullet\ P'' \Leftrightarrow \mu(P') \leq \mu(P'')$$

Para toda P′, P″ε P, donde P, sí o solo si, ≤• es de orden débil. Si tal homomorfismo existe, entonces; ((P, ≤•), (R, ≤), μ) es una escala ordinal.

Escala de intervalo

Las escalas de intervalo o cardinales son más refinadas, puesto que, además del orden o jerarquía entre categorías, las etiquetas o números consecutivos establecen intervalos iguales en la medición. Esta escala asigna números a los objetos de tal manera que el intervalo entre dos valores de medida es significativo en todo el rango de valores. Solo las funciones lineales positivas t(x) = ax + b son transformaciones admisibles.

Escalas de razón o proporción

Las escalas de razón son aquellas ampliamente encontradas en física y sólo son posiblescuando existen operaciones para determinar todas las cuatro relaciones: *igualdad, ordenación de rangos, igualdad de intervalos e igualdad de razones*. Una vez que la escala es erigida, sus valores numéricos puedes ser transformados. Por ejemplo, se puede decir que un segmento de programa es el doble de largo que otro segmento de programa.

Escalas absolutas

Este tipo de escalas son las más informativas, pero son usadas con muy poca frecuencia en la práctica. Por ejemplo, si se considera el atributo “número de líneas de código” de un segmento de programa (dicho atributo no es lo mismo que el tamaño, ya que es posible que se desee medir el tamaño a través de diferentes medidas).

Métricas de calidad

Las métricas de calidad son medidas cuantitativas utilizadas para evaluar atributos específicos de un producto de software, un proceso o un proyecto. Su propósito es proporcionar información objetiva sobre características como mantenibilidad, complejidad, acoplamiento, cohesión, encapsulamiento y otros factores relacionados con la calidad interna y externa del software. Estas métricas permiten identificar desviaciones respecto a criterios de calidad establecidos y apoyar la toma de decisiones durante el desarrollo, mantenimiento y evolución de los sistemas (Pressman & Maxim, 2020).

3.7. Estudio Experimental

Un estudio experimental permite al investigador elegir sus variables y mediante la manipulación de ellas en un ambiente controlado, se obtiene la evidencia para

comprobar una o más hipótesis (Hinkelmann & Kempthorne, 1994). El objetivo principal de un estudio experimental es analizar el efecto que los factores de entrada tienen sobre la variable de salida.

Un estudio experimental completo abarca las siguientes actividades:

1. Diseño experimental con una estructura lo más adecuada posible al fenómeno de estudio.
2. Experimentación de acuerdo con el plan de pruebas establecido en el diseño experimental.
3. Análisis de los resultados obtenidos y se aceptan o rechazan las hipótesis establecidas.
4. Se realizan las modificaciones oportunas para ampliar o modificar el diseño experimental.
5. Se obtienen las conclusiones apropiadas.

Diseño Experimental

Los modelos de diseño de experimentos son enfoques estadísticos clásicos cuyo propósito es determinar si ciertos factores afectan una variable de interés y, en caso afirmativo, cuantificar dicha influencia. Existen diversos enfoques para estructurar un estudio experimental; sin embargo, (Hinkelmann & Kempthorne, 1994) proponen un modelo general que establece una serie de pasos fundamentales para llevar a cabo una experimentación científica de manera sistemática.

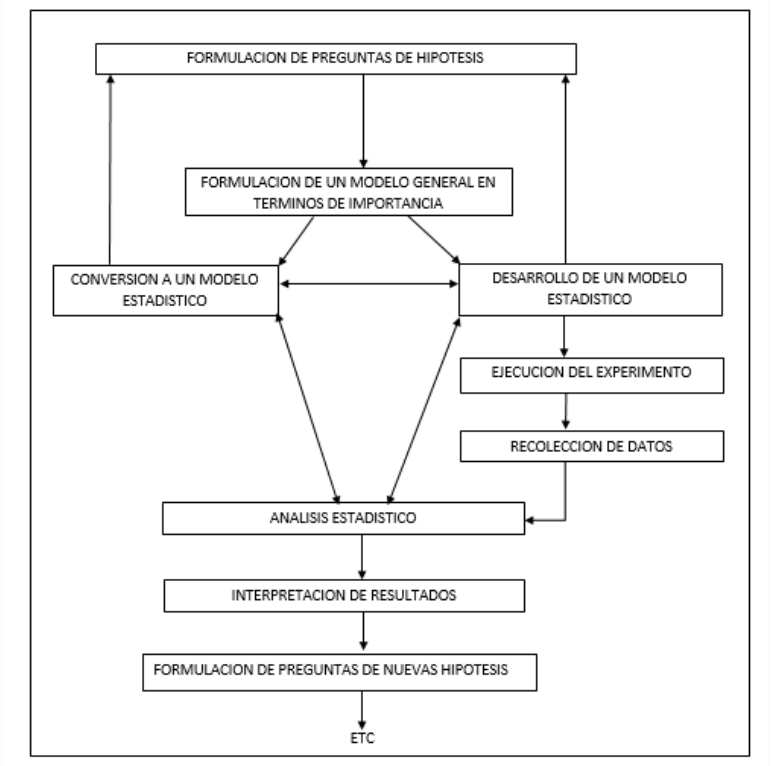

Figura 3: *Modelo general de diseño de experimentos*

El modelo propuesto por Hinkelmann y Kempthorne sirvió como referencia metodológica para la ejecución del estudio experimental en el presente trabajo de investigación.

3.8. Teoría Estadística

Hipótesis Estadisticas

Las hipótesis estadísticas son proposiciones formuladas acerca de una población o de las relaciones entre variables, cuya validez se evalúa mediante procedimientos estadísticos. Generalmente se expresan en forma de hipótesis nula (H_0) e hipótesis alternativa (H_1), las cuales permiten determinar si la evidencia observada en una muestra es suficiente para aceptar o rechazar una afirmación de investigación (Montgomery & Runger, 2018).

Variables (Hernández Sampieri et al., 2014)

Una variable es una característica, atributo o propiedad susceptible de adoptar diferentes valores dentro de una población o muestra. En el contexto de un diseño experimental, las variables se clasifican de acuerdo con la función que desempeñan en el estudio. La identificación y definición adecuada de estas variables permite garantizar la validez y confiabilidad de las observaciones realizadas durante el experimento.
Variables independientes: Pueden cambiar libremente su valor y no dependen de otra variable.
Variables dependientes: Representa la respuesta que se observan en el estudio experimental y que podrían estar influenciadas por los valores de la variable de entrada.
Variables intermitentes: Se interponen de manera indirecta para alterar de manera significativa o poco significativa la variable de salida. Para realizar un experimento robusto y confiable, es necesario aislar todos los factores externos que pudieran ocasionar ruido experimental y afectar los resultados del experimento.

Conversión y desarrollo de un modelo estadístico

Un modelo estadístico es una expresión simbólica empleada en todos los diseños experimentales y en la regresión para indicar los diferentes factores que modifiquen la variable de respuesta. El análisis estadístico de los datos ha sido de utilidad para determinar si es posible aceptar las hipótesis.
Los resultados obtenidos durante las pruebas experimentales pueden ser de distinta naturaleza, es decir, pueden ser datos paramétricos, no paramétricos, pueden tener diversas distribuciones de densidad, entre otros. Existen tantas posibilidades que es difícil suponer de entrada qué tipo de resultados se obtendrán en las pruebas experimentales (Kutner et al., 2005).

Pruebas Estadistísticas

Las pruebas estadísticas son procedimientos matemáticos utilizados para analizar datos y evaluar hipótesis mediante criterios objetivos de decisión. Su propósito es determinar si los patrones, diferencias o relaciones observados en una muestra pueden atribuirse al azar o si existe evidencia estadística suficiente para respaldar una conclusión sobre la población de estudio. La selección de una prueba estadística depende de las características de los datos, el tamaño de la muestra, la distribución de los valores observados y los objetivos de la investigación (Conover, 1999).

Tamaño de la muestra

La Ley de los Grandes Números constituye uno de los fundamentos teóricos de la inferencia estadística. Este principio establece que, a partir de cierto número de observaciones, la media muestral se estabiliza y la variación en los resultados deja de ser significativa. (Ross, 2014).

Prueba Shapiro Wilk

La prueba de Shapiro-Wilk es una prueba estadística de normalidad que permite determinar si una muestra proviene de una población con distribución normal. Se recomienda principalmente para muestras de tamaño igual o inferior a 50 observaciones, procediéndose a calcular la media y la varianza muestral (Razali & Wah, 2011).
Se rechaza la hipótesis nula de normalidad si el estadístico Shapiro-Wilk -W- es menor que el valor crítico proporcionado por la tabla elaborada por los autores para el tamaño de la muestra y el nivel de significancia dado (Shapiro & Wilk, 1965).

Prueba Binomial Exacta

La prueba Binomial Exacta permite evaluar si la proporción observada de “éxitos” en la muestra, en este caso, proyectos con deuda técnica difiere significativamente de una proporción teórica o esperada (p_0). A diferencia de las pruebas basadas en aproximaciones normales, este método calcula la probabilidad exacta de obtener una proporción igual o más extrema que la observada, bajo el supuesto de que la hipótesis nula sea verdadera (Clopper & Pearson, 1934; Conover, 1999).

El método de (Clopper & Pearson, 1934) fue utilizado para estimar los intervalos de confianza exactos al 95% asociados a la proporción observada, garantizando una interpretación precisa aún con tamaños muestrales moderados.

Prueba Kruskal-Wallis

La prueba de Kruskal–Wallis (Kruskal & Wallis, 1952) es una prueba no paramétrica utilizada para comparar si existen diferencias significativas entre dos o más grupos independientes, cuando la variable dependiente es ordinal o cuantitativa, pero no cumple el supuesto de normalidad requerido por el análisis de varianza ANOVA.
El estadístico de la prueba, denominado H, se calcula a partir de los rangos de los datos en lugar de sus valores originales, y su distribución se aproxima a una chi-cuadrado ($\chi 2$) con k−1 grados de libertad, donde k es el número de grupos.

Capítulo 4. Metodología de la RSL

4.1. Diseño de la RSL

La metodología de **Kitchenham** es un enfoque sistemático y riguroso diseñado para realizar revisiones sistemáticas de literatura (RSL) en el ámbito de la ingeniería de software. Su objetivo principal es recopilar, analizar y sintetizar la evidencia existente sobre un tema específico, garantizando que los resultados sean transparentes, reproducibles y relevantes para la comunidad científica (Kitchenham, 2007) .

Este método consta de tres fases principales:

1. **Planeación:** En esta etapa se define el protocolo de la revisión, que incluye las preguntas de investigación, los criterios de inclusión y exclusión de estudios, y las estrategias de búsqueda en bases de datos científicas.
2. **Conducción:** Implica ejecutar las búsquedas según el protocolo, seleccionar los estudios relevantes con base en los criterios establecidos, extraer datos clave utilizando un formulario diseñado previamente y evaluar la calidad de los estudios seleccionados.
3. **Reporte:** En esta etapa se sintetizan los resultados obtenidos, identificando patrones, tendencias y vacíos en la literatura.

En este proyecto de investigación, se utilizó esta metodología porque proporciona un enfoque riguroso y estructurado para realizar una **revisión sistemática de literatura**; para identificar, analizar, cuantificar y sintetizar patrones de la práctica sobre los factores de calidad estructural, en el desarrollo de software que generan entropía de software y deuda técnica.
En la Figura 4 se muestra las actividades que se realizaron durante el proceso de esta RSL siguiendo las pautas sugeridas por Kitchenham.

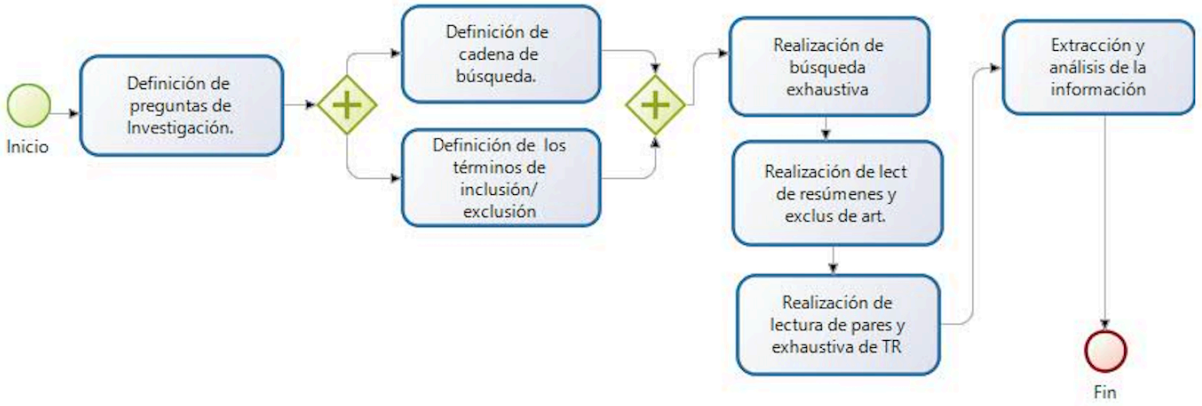

Figura 4: *Esquema de la metodología*

4.2. Preguntas de investigación

Se definieron dos preguntas de investigación para abordar la RSL.

Pregunta 1: ¿Cuáles son los principales factores de diseño y/o programación identificados en la literatura que generan código mal formado, contribuyendo a la entropía de software y deuda técnica?

Pregunta 2: ¿Cuáles son los factores de código mal formado (*code smell*) que producen entropía de software y deuda técnica que tienen poca investigación y que aún no se presenta solución alguna?

4.3. Definición de la cadena de búsqueda

El proceso consistió en hacer una búsqueda en las bases de datos de Scopus e IEEExplore, para conocer el panorama en la literatura haciendo uso de las siguientes combinaciones de palabras clave *“refactoring”*, *“code smell”*, *“smell code”*, *“technical debt”*, *“software entropy”*, *“design quality factor”*, *“programming design factor”*. Estas combinaciones de palabras clave se ampliaron utilizando operadores booleanos, dando lugar a la siguiente combinación de términos:

“(TITLE-ABS-KEY ("technical debt" OR "code smell" OR "software entropy" OR "smell code") AND TITLE-ABS-KEY (quality AND factor OR software AND design OR program AND quality OR programming AND design) AND TITLE-ABS-KEY (design AND refactoring OR program AND refactoring OR program AND design AND refactoring))”

La cadena de búsqueda propuesta se alinea con las preguntas de investigación al incluir términos clave como *technical debt, code smell* y *software entropy*, junto con *design, quality,* y *refactoring factors*. Esto permitió identificar los principales problemas que generan código mal formado y contribuyen a la entropía de software y la deuda técnica, así como explorar factores poco investigados que carecen de soluciones propuestas. El uso de conectores booleanos y la especificación en "TITLE-ABS-KEY" garantizaron precisión y amplitud, abarcando estudios relevantes sobre calidad estructural y soluciones en el diseño de software.

4.4. Definición de los criterios de exclusión e inclusión

Criterios de Inclusión:

- Artículos de estudios primarios publicados en revistas indexadas y arbitradas.
- Artículos de estudios secundarios.
- Artículos de conferencias.
- Trabajos publicados del 2016 al 2024.

Criterios de Exclusión:

- Se excluyen artículos escritos en idiomas diferentes al inglés.
- Se excluyen los estudios terciarios.
- Se excluyen libros, capítulos de libros, tesis, material didáctico, reportes, encuestas, artículos blancos, notas técnicas, artículos cortos, opiniones o artículos de debate.

4.5. Selección de los estudios

La selección y evaluación de los estudios se fundamentó en un proceso de revisión por pares, realizado de manera independiente por dos revisores, el investigador principal y el director de tesis. El proceso se estructuró en tres etapas:

1. **Primera etapa:** Se aplicaron los criterios de inclusión y exclusión de primer nivel, rango de fecha de publicación 2016-2024 mediante los filtros avanzados de las bases de datos.
2. **Segunda etapa:** Ambos revisores evaluaron de forma independiente los títulos y resúmenes de los artículos preseleccionados, aplicando los criterios metodológicos definidos. Los desacuerdos en esta fase se resolvieron mediante consenso tras una discusión técnica.
3. **Tercera etapa:** Se realizó la lectura completa de los artículos relevantes. Para garantizar el rigor metodológico, los dos revisores evaluaron cada estudio haciendo uso de los criterios de evaluación de calidad de artículos.

4.6. Reporte de Trabajos Relacionados

Se realizó una búsqueda exhaustiva en las bases de datos académicas IEEE Xplore y Scopus. Este proceso se enfocó en identificar estudios relevantes relacionados con la calidad estructural en proyectos de software, específicamente aquellos que exploren código mal formado, deuda técnica y entropía de software.

Como resultado, se obtuvieron 161 artículos potencialmente relevantes, que sirvieron como base para analizar los factores de código mal formado y patrones de diseño que afectan la estructura del software, así como para identificar áreas con vacíos de investigación o sin soluciones propuestas. En la Tabla 2 se muestra el resultado de la búsqueda.

Tabla 2: *Resultados de la Búsqueda*

Bases de Datos	Artículos sobre codigo mal formado	Artículos sobre deuda técnica	Artículos sobre entropía de software
IEEE Xplore	36	24	0
Scopus	117	59	1

Las temáticas analizadas no fueron mutuamente excluyentes, debido a que un mismo artículo científico abordó de manera simultánea más de una categoría de estudio. Por esta razón, la sumatoria de los artículos por temática difiere del total de publicaciones únicas recuperadas en cada base de datos.

4.7. Análisis de Trabajos Relacionados

En esta etapa se revisaron los títulos y resúmenes de los artículos recuperados durante la búsqueda inicial, con el objetivo de identificar aquellos que aborden aspectos clave relacionados con factores de diseño y programación que impactan la calidad estructural del software, así como aquellos que analicen la entropía y deuda técnica desde una perspectiva de calidad. Se realizó un cuidadoso proceso de selección, excluyendo los

artículos que no ofrecen información relevante para el estudio o que no se alinean con el enfoque metodológico planteado.

También se tomó en cuenta la siguiente lista de criterios de evaluación que permitieron validar la pertinencia de los casos estudio seleccionados:

- El artículo se enfoca en investigar los factores de código mal formado, técnicas, herramientas, estándares y buenas prácticas relacionadas con la detección y corrección de código mal formado, así como su impacto en la entropía y la deuda técnica del software en proyectos de desarrollo.
- El artículo ofrece una descripción clara del problema de investigación.
- El artículo sigue un proceso de investigación estructurado y fundamentado.
- El artículo expone de manera clara y detallada los resultados obtenidos.
- El artículo ha sido publicado en una revista, conferencia o congreso relevante.
- El artículo ha sido citado por otros autores.
- El artículo describe claramente trabajos futuros o alternativas de investigación.

Finalizada la tercera etapa de la revisión y luego de haber aplicado los criterios de evaluación, en la Tabla 3 se muestra el listado de artículos que fueron seleccionados, siendo un total de 55 artículos que sirvieron como base para analizar los factores de código mal formado y patrones de la práctica de diseño que afectan la estructura del software, así como para identificar áreas con vacíos de investigación o sin soluciones propuestas.

Tabla 3: *Artículos Relacionados*

Num	Referencia	Nombre del Artículo
1	(Xu et al., 2017)	A Log-linear Probabilistic Model for Prioritizing Extract Method Refactorings.
2	(Abdelaziz et al., 2018)	A Novel Approach for Improving the Quality of Software Code using Reverse Engineering.
3	M.Sangeetha & P.Sengottuvelan, 2017)	A Perspective Approach for Refactoring Framework Using Monitor based Approach.
4	(Zhang et al., 2018)	A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software.
5	(Mkaouer et al., 2017)	A robust multi-objective approach to balance severity and import of refactoring opportunities.
6	(Kaur et al., 2017)	A Support Vector Machine based Approach for Code Smell Detection.
7	(Singh et al., 2019)	A User Feedback Centric Approach for Detecting God Class Code Smell Using Frequent Usage Patterns.
8	(Akash et al., 2019)	An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity.
9	(Guggulothu & Moiz, 2019)	An Approach to Suggest Code Smell Order for Refactoring.
10	(Bacon et al., 2017)	An Automated Code Smell and Anti-Pattern Detection Approach.
11	(Zhu et al., 2023)	An Efficient Design Smell Detection Approach with Inter-class Relation.
12	(Tufano et al., 2016)	An Empirical Investigation into the Nature of Test Smells.

Num	Referencia	Nombre del Artículo
13	(Mumtaz et al., 2018)	An empirical study to improve software security through the application of code refactoring.
14	(Mehta et al., 2018)	Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability.
15	(Panigrahi et al., 2020)	Application of Naïve Bayes classifiers for refactoring prediction at the method level.
16	(Liu et al., 2018)	Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Predicton An Empirical Study.
17	(Sirikul & Soomlek, 2016)	Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs.
18	(Ubayawardana & Karunaratna, 2018)	Bug Prediction Model using Code Smells.
19	(Guggulothu & Moiz, 2019)	Code smell detection usingmulti-label classification approach
20	(Jaber, 2019)	Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability.
21	(Bahaa-Eldin, 2021)	Code Smells and Detection Techniques A Survey.
22	(Paulk, 2016)	Code smells incidence: does it depend on the application domain?
23	(Ganea et al., 2017)	Continuous quality assessment with inCode.
24	(Liu et al., 2021)	Deep Learning Based Code Smell Detection.
25	(Sharma, 2018)	Detecting and Managing Code Smells: Research and Practice.
26	(Kumar Das et al., 2019)	Detecting Code Smells using Deep Learning.
27	(Pecorelli et al., 2020)	Developer-Driven Code Smell Prioritization.
28	(Liu et al., 2016)	Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection.
29	(Rani & Chhabra, 2017a)	Evolution of Code Smells over Multiple Versions of Softwares: An Empirical Investigation.
30	(Lahti et al., 2021)	Experiences on Managing Technical Debt with Code Smells and AntiPatterns.
31	(Hu, 2020)	Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism.
32	(Sae-Lim et al., 2017)	How Do Developers Select and Prioritize Code Smells? A Preliminary Study.
33	(Bibiano et al., 2020)	How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?
34	(Oliveira et al., 2016)	Identifying Code Smells with Collaborative Practices: A Controlled Experiment.
35	(Gradišnik & Heričko, 2018)	Impact of Code Smells on the Rate of Defects in Software: A Literature Review.
36	(Pantiuchina et al., 2018)	Improving Code: The (Mis)perception of Quality Metrics.
37	(Shen et al., 2020)	Improving Machine Learning-based Code Smell Detection via Hyper-parameter Optimization.
38	(Reeshti et al., 2019)	Measuring Code Smells and Anti-Patterns.
39	(Aras & Sel'fuk, 2016)	Metric and Rufe Based Automated Detection of Antipatterns in Object-Oriented Software Systems.
40	(Merzah & Selçuk, 2017)	Metric Based Detection of Refused Bequest Code Smell.

Num	Referencia	Nombre del Artículo
41	(M. M. Rahman et al., 2018)	MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design.
42	(Ouni et al., 2017)	MORE: Amulti-objective refactoring recommendation approach to introducing design patterns and fixing code smells.
43	(Rani & Chhabra, 2017b)	Prioritization of Smelly Classes: A Two Phase Approach.
44	(M. Rahman et al., 2017)	Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity.
45	((Nyamawe et al., 2018)	Recommending Refactoring Solutions based on Traceability and Code Metrics.
46	(Turkistani & Liu, 2019)	Reducing the Large Class Code Smell by Applying Design Patterns.
47	(Arif & Rana, 2020)	Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort.
48	(Peruma et al., 2022)	Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring.
49	(Gupta et al., 2023)	Resource Allocation Modeling Framework to Refactor Software Design Smells.
50	(Sangeetha & Sengottuvelan, 2017)	Systematic Exhortation of Code Smell Detection Using JSmell for Java Source Code.
51	(Mohan et al., 2016)	Technical debt reduction using search based automated refactoring.
52	(Fernandes et al., 2024)	The Impact of a Live Refactoring Environment on Software Development.
53	(Palomba et al., 2018)	The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells.
54	(Palomba et al., 2019)	Toward a Smell-aware Bug Prediction Model.
55	(Kiss & Mihancea, 2018)	Towards Feature Envy Design Flaw Detection at Block Level.

Se identificaron 42 factores de código mal formado, de ellos, 25 en prácticas de diseño, que impactan la arquitectura, modularidad y cohesión del sistema, y 17 en prácticas de programación, que afectan la claridad y eficiencia del código. La Tabla 4 presenta el listado detallado de estos factores, incluyendo su descripción, clasificación y recurrencia de aparición en la literatura analizada.

Tabla 4: *Recurrencia de código mal formado*

Num	Factores de código mal formado	Recurrencia	Descripción	Clasificación	
				Diseño	Programación
1	Feature Envy	33	Métodos que acceden frecuentemente a datos de otra clase.	x	
2	Long Method	25	Métodos extensos que dificultan la comprensión.		x
3	God Class	25	Clases que centralizan demasiada lógica.	x	

Num	Factores de código mal formado	Recurrencia	Descripción	Clasificación	
				Diseño	Programación
4	Data Class	17	Clases que solo contienen datos y carecen de comportamiento.		x
5	Duplicated Code	15	Código repetido.		x
6	Large Class	13	Clases con demasiados métodos o responsabilidades.	x	
7	Shotgun Surgery	11	Cambios en una funcionalidad requieren modificar muchas clases.	x	
8	Message Chain	8	Secuencias largas de llamadas entre objetos.	x	
9	Lazy Class	6	Clases con funcionalidad limitada.		x
10	Spaghetti Code	4	Código desorganizado y difícil de seguir.	x	
11	Long Parameter List	4	Métodos con demasiados parámetros.		x
12	Brain Method	4	Métodos que concentran demasiada lógica compleja.		x
13	Dispersed Coupling	4	Dependencias distribuidas en múltiples lugares.	x	
14	Divergent Change	3	Clases que necesitan ser modificadas por múltiples razones.	x	
15	Primitive Obsession	3	Uso excesivo de tipos primitivos para representar conceptos complejos.		x
16	Type Checking	3	Comprobaciones explícitas de tipo en lugar de usar polimorfismo.		x
17	Blob Operation	3	Métodos que concentran demasiada lógica.	x	
18	Misplaced Class	2	Clases ubicadas en paquetes incorrectos.	x	
19	Brain Class	2	Clases que concentran demasiada lógica del sistema.	x	
20	Redundant Code	2	Código que realiza tareas duplicadas.		x
21	Cyclomatic Complexity	1	Flujo lógico complejo.		x
22	Unused Imports	1	Importaciones no utilizadas en el código.		x
23	Too Many Methods	1	Clases con demasiados métodos		x
24	Broken Hierarchy	1	Jerarquías de herencia mal diseñadas.	x	

Num	Factores de código mal formado	Recurrencia	Descripción	Clasificación	
				Diseño	Programación
25	Insufficient Modularization	1	Falta de separación adecuada en módulos.	x	
26	Deficient Encapsulation	1	Detalles internos de una clase expuestos innecesariamente.	x	
27	Internal Duplication	1	Código duplicado dentro de una misma clase.		x
28	Un-Encapsulated	1	Datos no protegidos dentro de una clase.	x	
29	Promiscuous Package	1	Paquetes con clases diversas e inconexas.	x	
30	Complex Class	1	Clases con lógica demasiado intrincada.	x	
31	Overcomplicated Expressions	1	Expresiones innecesariamente complejas.		x
32	Global Data	1	Datos accesibles globalmente.		x
33	Schizophrenic Class	1	Clases con múltiples responsabilidades no relacionadas.	x	
34	Dead Code	1	Código no utilizado.		x
35	Refused Bequest	1	Subclases que no usan funcionalidad heredada.	x	
36	Lava Flow	1	Código obsoleto que permanece en el sistema.		x
37	Inappropriate Intimacy	1	Clases que acceden excesivamente a detalles internos de otras.	x	
38	Unused Field	1	Campos declarados, pero nunca utilizados.		x
39	Distorted Hierarchy	1	Herencias mal diseñadas.	x	
40	Intensive Coupling	1	Clase que depende demasiado de los métodos de otra clase.	x	
41	Swiss Army Knife	1	Clases con demasiadas funcionalidades generales.	x	
42	Speculative Generality	1	Código escrito para un uso futuro no confirmado.	x	

El análisis de los artículos mostró diferencias significativas en el tratamiento de los 42 factores detectados, tal como se evidencia en la Tabla 13 del capítulo 8, donde se observa que los factores de alta complejidad presentan una carencia crítica de soluciones de refactorización automatizadas y validadas. El estudio revela que gran parte de la literatura se limita al diagnóstico heurístico, dejando vacíos técnicos en la mitigación de la entropía de software y la corrección de la deuda técnica en sistemas legados. Estos hallazgos se consolidaron mediante la agrupación de los artículos

estudiados por factor de código mal formado y el análisis posterior del alcance metodológico de cada investigación.

Como referencia de este proceso y sus hallazgos, se encuentra el artículo *"Systematic Review of Code Smell Patterns and Their Impact on Software Technical Debt"* (Rodríguez Ponce et al., 2025), en el cual se detalla cómo la incidencia de malas prácticas de los desarrolladores de software incrementa la entropía del software y acelera la acumulación de deuda técnica en sistemas legados. Dicha publicación profundiza en la relación entre los defectos estructurales y la degradación de la mantenibilidad, estableciendo la base científica sobre la cual se desarrolla la presente investigación.

Capítulo 5. Materiales y Método de Solución

Para realizar una evaluación objetiva y sistemática en este estudio se desarrolló un instrumento de pruebas consistente en un marco de métricas de calidad estructural, el cual permitió cuantificar el nivel de factores de deuda técnica, originada por la presencia de código mal formado en sistemas legados. En la Figura 5, se presenta este marco, el cual consiste de un conjunto de métricas automatizadas que permiten cuantificar características internas del código fuente, específicamente relacionadas con factores de código mal formado que origina deuda técnica. Como parte de este enfoque, se presenta en la Figura 6 el diagrama de clases del modelo de objetos utilizado y en la Figura 7, la estructura de datos correspondiente, para la captura de información de los sistemas legados bajo estudio, así como una descripción detallada de las métricas que serán aplicadas.

5.1. Marco de métricas

A continuación, en la Figura 5 se presenta el diagrama de clases correspondiente al marco de métricas. Las métricas utilizadas en este proyecto de investigación se encuentran resaltadas en color azul.

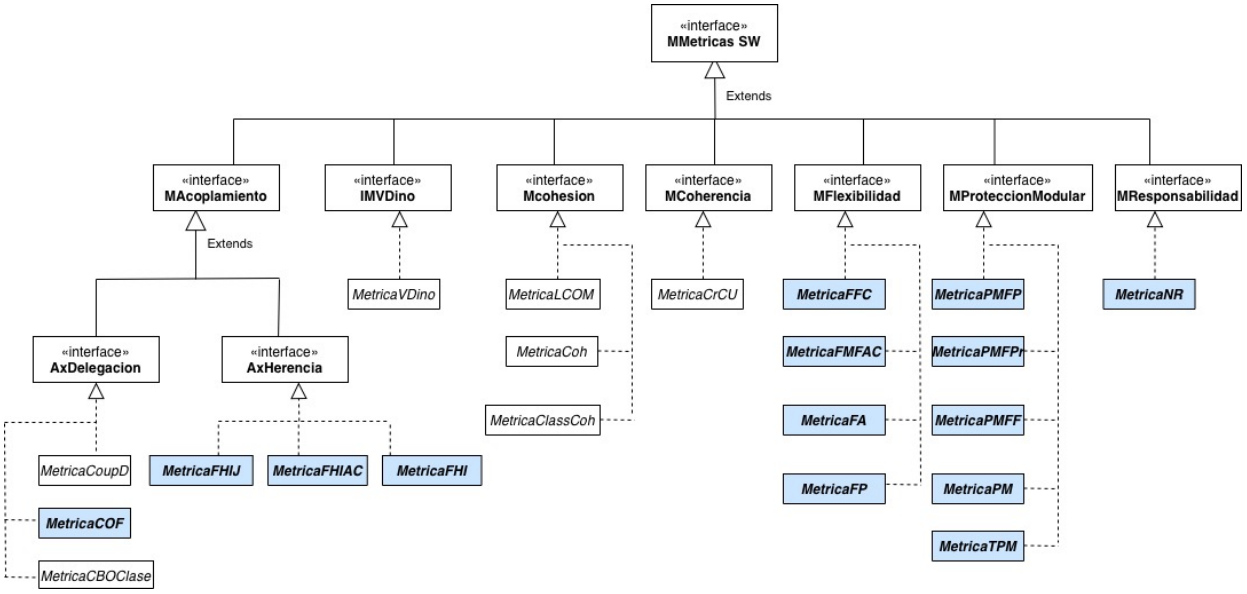

Figura 5: *Diagrama del Marco de Métricas*

5.1.1. Diagrama del Modelo de Objetos y Estructura de Datos

A continuación, en la Figura 2 se presenta el diagrama de clases correspondiente al modelo de objetos. Este diagrama ilustra las relaciones entre los elementos clave que estructuran la información de cada caso de prueba analizado. Además, se incluye una descripción detallada de cada una de las clases que conforman este modelo. Las entidades de software utilizadas en este proyecto de investigación se encuentran resaltadas en color azul.

Las clases correspondientes al modelo, permiten crear la estructura de datos que contiene los metadatos de los objetos del proyecto de entrada.

La clase *Proyecto* agrupa uno o más *Archivos* y estos a su vez está compuesto por uno o más *Paquetes*. Cada *Paquete* contiene *Secuencias*, de uno a más *Clasificadores*, que pueden ser *Clases* o *Interfaces*.

Un *Clasificador* es una entidad abstracta que modela tanto clases como interfaces. Hereda propiedades comunes a ambos tipos. A su vez, la *Clase* puede ser Base, Abstracta o Concreta, y contiene uno o más Atributos y Métodos. Los *Atributos* pueden tener distintos niveles de *Alcance* como "*public*", "*private*", "*protected*" y "*friendly*".

Los *Métodos* representan operaciones o funciones que pueden ejecutar los objetos. Estos pueden ser de tipo Constructor, Abstracto, Concreto o Virtual y contienen parámetros y sentencias. Cada método puede tener múltiples Parámetros, que son recibidos como precondiciones necesarias para su ejecución.

La clase *Relación* modela los distintos tipos de vínculos entre clasificadores, tales como Composición, Agregación, Herencia y Dependencia, permitiendo representar cómo interactúan las clases entre sí.

El modelo también considera la Sentencia, una instrucción en cualquier línea de código dentro del cuerpo de un método. Las Sentencias pueden ser extendidas por tipos concretos como Declaraciones e Instrucciones y este último a su vez puede ser una *"InstruccionCall"*.

Toda la información de este modelo de objetos se almacena en estructuras dinámicas de datos de tipo lista. En la Figura 3 se ilustra un ejemplo de cómo se representa en memoria la información obtenida por el analizador sintáctico.

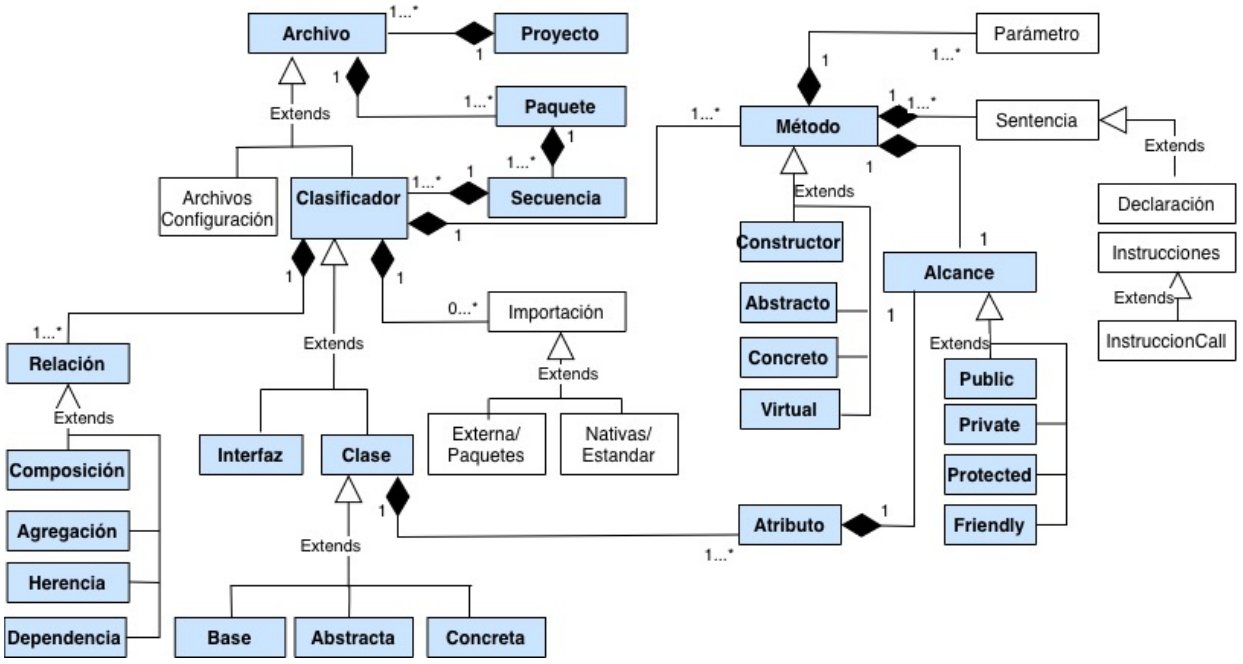

Figura 6: *Diagrama del Modelo de Objetos*

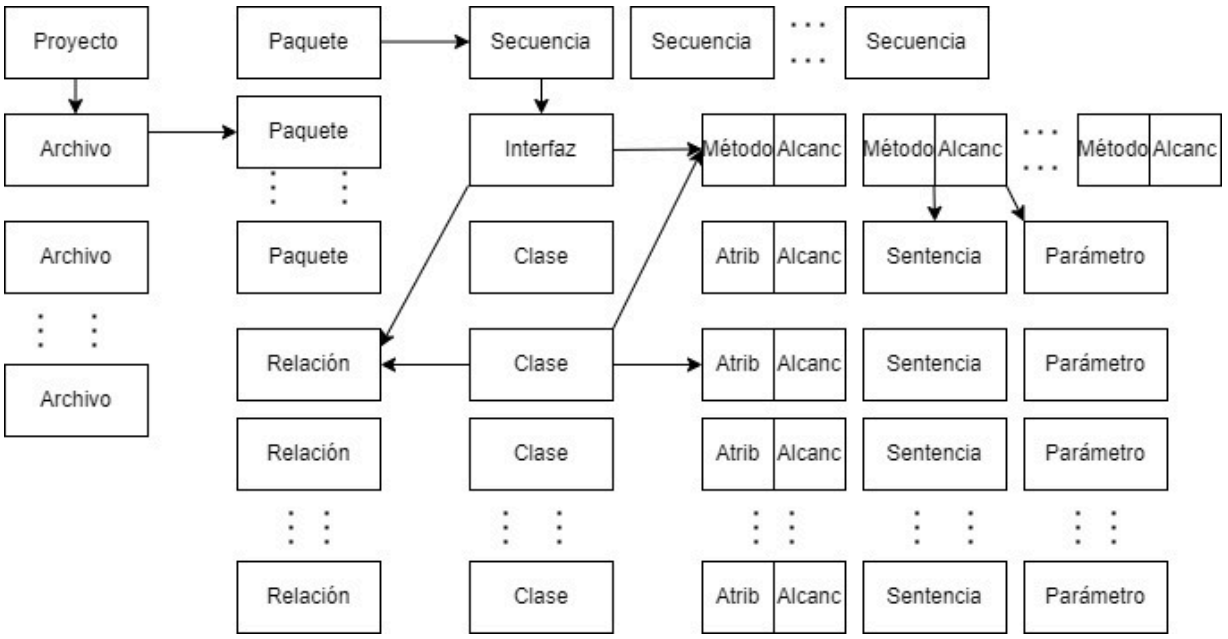

Figura 7: *Estructura de Datos del Modelo*

5.1.2. Medición de carencia de protección

Métrica de Factor de Protección Modular de Funciones Privadas (PMFP): Esta métrica consiste en detectar en cada una de las clases de un sistema legado bajo estudio, las funciones que han sido declaradas como "*private*" realizar una suma de todas estas funciones, y dividir esta suma entre el número total de funciones, posteriormente realizar una suma de todos los valores obtenidos y dividirlo entre el número total de clases.

La expresión matemática se muestra a continuación:

$$\boldsymbol{PMFP} = \frac{\sum_{\boldsymbol{ci}=\boldsymbol{1}}^{\boldsymbol{ci}=\boldsymbol{n}} \left(\frac{\sum_{\boldsymbol{fi}=\boldsymbol{0}}^{\boldsymbol{fi}=\boldsymbol{m}} \boldsymbol{FP}}{\boldsymbol{FTC}} \right)}{\boldsymbol{NTC}} \qquad (1)$$

Donde:
$\boldsymbol{FP}$ Funciones con calificador "*private*".
$\boldsymbol{fi}$ "*i-esima*" función.
$\boldsymbol{FTC}$ Número total de funciones de la clase.
$\boldsymbol{ci}$ "*i-esima*" clase.
$\boldsymbol{NTC}$ Número total de clases.

A medida que esta razón tienda al 1 indica que hay una tendencia a tener funciones "*private*" y por lo tanto se presenta una alta protección de la información. Una medida que tienda al 0 indica que hay pocas funciones "*private*", por lo que la protección de la

información se encuentra con un nivel bajo. El mejor valor será el 1, el peor valor será 0 (Barón Pérez, 2020).

Factor de Protección Modular de Funciones Protegidas (PMFPR): Esta métrica consiste en sumar las funciones de un sistema legado bajo estudio, que han sido declaradas "*protected*" de cada una de las jerarquías de herencia, y dividir esta suma entre el número total funciones, posteriormente realizar la suma de todos los valores obtenidos y dividirlo entre el número total de jerarquías de herencia (Barón Pérez, 2020).

La expresión matemática se muestra a continuación:

$$\boldsymbol{PMFPr} = \frac{\sum_{\boldsymbol{ji=1}}^{\boldsymbol{ji=n}} \left(\frac{\sum_{\boldsymbol{fi=0}}^{\boldsymbol{fi=m}} \boldsymbol{FPr}}{\boldsymbol{NTF}} \right)}{\boldsymbol{NTJ}} \tag{2}$$

Donde:
$\boldsymbol{FPr}$ Funciones con calificador "*protected*".
$\boldsymbol{fi}$ "*i-esima*" función "*protected*" de la jerarquía.
$\boldsymbol{NTF}$ Número total de funciones de la jerarquía.
$\boldsymbol{ji}$ "*i-esima*" jerarquía.
$\boldsymbol{NTJ}$ Número total de jerarquías.

A medida que esta razón tienda al 1 indica que hay una tendencia a tener funciones "*protected*" y por lo tanto se presenta alta protección de la información. Una medida que tienda al 0 indica que hay pocas funciones "*protected*", por lo que la protección de la información se encuentra con un nivel bajo. El mejor valor será el 1, el peor valor será 0 (Barón Pérez, 2020).

Factor de Protección Modular de Funciones Friendly (PMFF): Esta métrica consiste en sumar las funciones de un sistema legado bajo estudio, que han sido declaradas "*friendly*" o con calificador de alcance ("*default*"), y dividir esta suma entre el número total funciones.

La expresión matemática se muestra a continuación:

$$\boldsymbol{PMFF} = \frac{\sum_{\boldsymbol{i=0}}^{\boldsymbol{i=n}} \boldsymbol{FF}}{\boldsymbol{NTF}} \tag{3}$$

Donde:
$\boldsymbol{FF}$ Funciones con calificador "*friendly*" o "*default*".
$\boldsymbol{i}$ "*i-esima*" función "f*riendly*" o "*default*".
$\boldsymbol{NTF}$ Número total de funciones.

A medida que esta razón tienda al 1 indica que hay una tendencia a tener funciones "*friendly*" y por lo tanto se presenta una alta protección de la información entre paquetes. Una medida que tienda al 0 indica que hay pocas funciones "*friendly*" o "*default*", por lo

que la protección de funciones "*friendly*" se encuentra con un nivel bajo. El mejor valor será el 1, el peor valor será 0 (Barón Pérez, 2020).

Factor de Protección Modular (PM): Esta métrica consiste en la suma de las funciones de un sistema legado bajo estudio, que han sido declaradas con el calificador de alcance diferente a "*public*", y se divide esta suma entre el número total de funciones.

La expresión matemática se muestra a continuación:

$$\boldsymbol{PM} = \frac{\sum_{\boldsymbol{i}=\boldsymbol{0}}^{\boldsymbol{i}=\boldsymbol{n}} \boldsymbol{FNP}}{\boldsymbol{NTF}} \tag{4}$$

Donde:
$\boldsymbol{FNP}$ Funciones que no son "*public*"
$\boldsymbol{i}$ "*i-esima*" función "*public*".
$\boldsymbol{NTF}$ Número total de funciones.

A medida que esta razón tienda a 0 mayor es el problema, indicaría que hay una tendencia a tener muchas funciones públicas que no inician capacidades o requerimientos de la aplicación en evaluación y por lo tanto tienen poca protección modular y un encapsulamiento incorrecto.

Una medida que tienda al 1 indica que hay pocas funciones públicas que no son iniciadoras de requerimientos o capacidades de la aplicación en evaluación, por lo tanto, tienden a tener más capacidad de protección modular y mejor nivel de encapsulamiento. El mejor valor será de 1, el peor valor será de 0 (Barón Pérez, 2020).

Total de Protección Modular (TPM): Se ha definido esta métrica para medir el grado total de protección modular ponderada que tiene la arquitectura de clases. Esta métrica consiste en la suma de PMFP, PMFPr y PMFF entre tres.

La expresión matemática se muestra a continuación:

$$TPM = \frac{\big((PMFP * 2) + (PMFPr * 0.75) + (PMFF * 0.25)\big)}{3} \tag{5}$$

Donde:
$\boldsymbol{PMFP}$ Grado de protección modular de funciones "*private*".
$\boldsymbol{PMFPr}$ Grado de protección modular de funciones "*protected*".
$\boldsymbol{PMFF}$ Es el grado de protección modular de funciones "*friendly*" o "*default*".

La métrica fue normalizada con el objetivo de que los valores obtenidos se encuentren dentro del rango del 0 al 1. A medida que esta razón tienda a 0 mayor es el problema, indicaría que hay una tendencia a tener muchas funciones públicas que no necesariamente son iniciadoras de capacidades o requerimientos de la aplicación en

evaluación y por lo tanto tienen poca protección modular y un encapsulamiento incorrecto.

Una medida que tienda al 1 indica que hay pocas funciones públicas que no son que no necesariamente son iniciadoras de capacidades o requerimientos de la aplicación en evaluación, por lo tanto, tienden a tener más capacidad de protección modular y mejor nivel de encapsulamiento. El mejor valor será de 1, el peor valor será de 0 (Barón Pérez, 2020).

Cada una de las métricas fue sustentada en base a la teoría de la medición como escalas ordinales. Para el sustento de cada una de las métricas se tomaron tres diferentes arquitecturas de clases, demostrando que cada una de ellas cumple con los requisitos del axioma de orden débil (ser de orden débil y cumplir con la propiedad de homomorfismo (Zuse, 1995)).

5.1.3. Medición de carencia de abstracción

Factor de Abstracción (FA): La métrica de abstracción es una métrica lineal, la cual mide lo abstracto que es un paquete de software, en el sentido de que calculará la relación entre el número de clases abstractas o interfaces y el número total de clases dentro de un paquete de software en un sistema legado (Martin, 1994).

La expresión matemática se muestra a continuación:

$$FA = \frac{Na}{Nc} \tag{6}$$

Donde:
$\boldsymbol{Na}$ Es el número de clases abstractas e interfaces en el paquete.
$\boldsymbol{Nc}$ Es el número total de clases concretas en el paquete.

Esta métrica dará valores en el rango de [0, 1]. FA = 0 significa que ninguna de las clases de un paquete o módulo de programa es abstracta o interfaz, FA = 1 indica un paquete que consta únicamente de clases e interfaces abstractas. Los paquetes más abstractos son mejores porque son más fáciles de mantener y reutilizar (Sánchez Rogel, 2023).

Factor de Polimorfismo (FP): Esta métrica consiste en la proporción entre el número real de posibles situaciones polimórficas para una clase Ci entre el máximo número posible de situaciones polimórficas en Ci" (Rodríguez & Harrison, 2002)

La expresión matemática se muestra a continuación:

$$FP = \frac{\sum_{i=1}^{TC} \boldsymbol{M_o}(\boldsymbol{C_i})}{\sum_{i=1}^{TC}[\boldsymbol{M_n}(\boldsymbol{C_i}) * \boldsymbol{DC}(\boldsymbol{C_i})]} \tag{7}$$

Donde:
$\boldsymbol{Md(Ci)}= Mn(Ci)+ Mo(Ci)$
$\boldsymbol{DC(Ci)}$ es el número de descendientes de Ci.
$\boldsymbol{Mn(Ci)}$ es el número de métodos nuevos.
$\boldsymbol{Mo(Ci)}$ es el número de métodos redefinidos.
$\boldsymbol{TC}$ es el número total de clases.

La métrica fue normalizada con el objetivo de que los valores obtenidos se encuentren dentro del rango del 0 al 1. A medida que esta razón tienda a 0 será baja la utilización del polimorfismo y el valor cercano a 1 indica que casi todos los métodos potencialmente serán redefinibles, es decir, hay una posibilidad del polimorfismo (Sánchez Rogel, 2023).

5.1.4. Medición de responsabilidad

Métrica Número de Responsabilidades (NR): Se definió la métrica NR (Número de Responsabilidades) para medir el número de responsabilidades. Esto es el número de secuencias interactivas existentes dentro de arquitecturas modulares, de un sistema legado bajo estudio, para así cumplir con el *"Principio de Única Responsabilidad"*. Esta métrica consiste en dividir el número de responsabilidades ideal, en este caso, 1 entre la suma total de secuencias existentes dentro de cada arquitectura modular (paquete/módulo).

La expresión matemática se muestra a continuación:

$$NR = \frac{1}{\sum_{i=1}^{i=n} secuencia_i} \tag{8}$$

Donde:
$\boldsymbol{NR}$ Número de responsabilidades
$\boldsymbol{n}$ Número total de métodos públicos. Donde cada método público es iniciador de una secuencia interactiva de métodos, donde cada secuencia, representa una responsabilidad con una meta de valor para el usuario.
$\boldsymbol{secuencia}$ de métodos interactivos en una o más clases dentro de una arquitectura modular. Dichas secuencias inician con un método público. Las secuencias se forman por las invocaciones de un método a otro método. Una secuencia termina cuando un método ya no invoca a otro método y regresa sucesivamente el control hacia el método llamador de manera apropiada.
La métrica fue normalizada con el objetivo de que los valores obtenidos estén dentro del rango del 0 al 1, el mejor valor es 1, lo cual significa que la arquitectura modular cuenta con una única responsabilidad. El valor menos deseado es el 0, lo cual significa que las responsabilidades tienden al infinito.
$\boldsymbol{NR} \rightarrow$ A medida que el valor tienda a 0 indica que el número de responsabilidades tiende hacía el infinito.
$\boldsymbol{NR} \rightarrow$ Una medida que tienda a 1 nos indica que el número de responsabilidades disminuye.
$\boldsymbol{NR} \rightarrow$ Cuando el valor sea igual a 1, indica que la arquitectura modular solo cuenta con una única responsabilidad (Díaz, 2023)

La métrica fue sustentada con base a la teoría de la medición como escala ordinal (Zuse, 1992), para dicho sustento de la métrica, se tomaron tres arquitecturas modulares diferentes, demostrando que cada una de ellas cumple con los requerimientos del axioma de orden débil y que además se cumpla la propiedad de homomorfismo.

5.1.5. Medición de herencia de implementación

Factor de Herencia de Implementación (FHI): Esta métrica consiste en la suma de los métodos virtuales y no virtuales implementados en la clase base en una jerarquía de herencia, de un sistema legado bajo estudio, entre el número total de métodos de dicha clase base

La expresión matemática se muestra a continuación:

$$FHI = \frac{\sum M\ v + \sum M\ nv}{Tm} \tag{9}$$

Donde:
$\sum \boldsymbol{M\ v}$ Sumatoria de métodos virtuales (no abstractos) implementados en una clase base. Estos métodos pueden/deben ser redefinidos en las clases derivadas.
$\sum \boldsymbol{M\ nv}$ Sumatoria de métodos no virtuales (no abstractos) implementados en una clase base. Estos métodos no se pueden redefinir en las clases derivadas.
$\boldsymbol{Tm}$ Total de métodos en una clase base.

La métrica fue normalizada para que los valores obtenidos no dependan de la cantidad de métodos de una clase dada. El mejor valor es 0, lo cual significa que una clase derivada no hereda implementación alguna desde los métodos de su clase base. En caso contrario, el valor menos deseado es el 1, lo que significa que todos los métodos de la clase derivada heredan implementación desde los métodos de clase base (Ortiz Gutiérrez, 2020).

Factor de Herencia de Implementación de una Jerarquía de Clases (FHIJ): Consiste en la sumatoria de los cocientes, donde el numerador es la suma acumulada de métodos virtuales en la jerarquía de clases, y el divisor es la suma acumulada del total de métodos en la jerarquía de clases. Posteriormente, este resultado es dividido entre el número total de clases menos uno.

La expresión matemática se muestra a continuación:

$$\boldsymbol{FHIJ} = \frac{\sum_{i}^{n}\left(\frac{\sum_{j=0}^{j=i-1} \boldsymbol{MvC_j}}{\sum_{j=0}^{j=i-1} \boldsymbol{MC_j}}\right)}{\boldsymbol{n-1}} \tag{10}$$

Donde:
$\boldsymbol{MvC}$ Cantidad de Métodos Virtuales de la Clases

$\boldsymbol{MC}$ Cantidad de Métodos Totales de las Clases
$\boldsymbol{i}$, $\boldsymbol{j}$ Representan el recorrido de la i, "j *-esima*" clase
$\boldsymbol{n}$ Total de clases en la jerarquía

Una condición presente en la fórmula es cuando i = 0, $FHIJ$ obtiene el valor de 0.

La métrica fue normalizada para que los valores obtenidos no dependan de la cantidad de métodos de una clase dada. El mejor valor es 0, lo cual significa que no se está heredando implementación alguna de los métodos en la jerarquía, mientras que 1 es el valor menos deseado, que significa que cada clase en una jerarquía tiene el máximo de herencia de implementación (Ortiz Gutiérrez, 2020).

Factor de Herencia de Implementación de una Arquitectura de Clases (FHIAC): Esta métrica cuantifica el grado promedio de uso de herencia de implementación en las jerarquías de clases que conforman una arquitectura de software. Su cálculo se realiza mediante la sumatoria de los valores de $FHIJ$ obtenidos para cada jerarquía de clases, dividida entre el número total de jerarquías de clases en la arquitectura (NOH).

La expresión matemática se muestra a continuación:

$$FHIAC = \frac{\sum FHIJ}{NOH} \tag{11}$$

Donde:
$\sum \boldsymbol{FHIJ}$ Sumatoria de $FHIJ$ de una jerarquía de clases.
$\boldsymbol{NOH}$ Es el conteo de las jerarquías de clase.

La métrica fue normalizada para que los valores obtenidos no dependan de la cantidad de métodos de una clase dada. El mejor valor es 0, lo cual significa que ninguna de las jerarquías de la arquitectura, hereda implementación desde los métodos, mientras que 1 es el valor menos deseado, lo que significa que cada jerarquía de clases en la arquitectura tiene el máximo de herencia de implementación (Ortiz Gutiérrez, 2020).

Factor de Flexibilidad de Clases (FFC): Esta métrica consiste en la suma de los métodos virtuales o abstractos que exhiben un comportamiento polimórfico (NOP), entre el número total de sus métodos.

La expresión matemática se muestra a continuación:

$$FFC = \frac{\sum NOP}{Tm} \tag{12}$$

Donde:
$\sum \boldsymbol{NOP}$ Sumatoria de los métodos virtuales o abstractos que exhiben un comportamiento polimórfico (Bansiya & Davis, 2002).
$\boldsymbol{Tm}$ Total de métodos en una clase base.

La métrica fue normalizada para que los valores obtenidos no dependan de la cantidad de métodos de una clase dada. El mejor valor es 1, lo que significa que todos los métodos de la clase son abstractos y/o virtuales redefinidos en las clases derivadas, por lo tanto, todos exhiben un comportamiento polimórfico. En caso contrario, el valor menos deseado es el 0, lo cual significa que no se permite la redefinición de ningún método. Es decir, ninguno de los métodos es abstracto y ninguno de los métodos es virtual (Ortiz Gutiérrez, 2020).

Factor Medio de Flexibilidad de Arquitecturas de Clases (FMFAC): Esta métrica consiste en la suma de los FFC de la arquitectura del módulo entre el número total de clases.

La expresión matemática se muestra a continuación:

$$FMFAC = \frac{\sum FFC}{Tc} \tag{13}$$

Donde:
$\sum \boldsymbol{FFC}$ = Sumatoria del Factor de Flexibilidad de Clase.
$\boldsymbol{Tc}$ =Total de clases en la arquitectura.

El valor óptimo de la métrica es 1, lo cual significa que todos los métodos de las clases de la arquitectura son virtuales y/o abstractos, por lo tanto, todos exhiben un comportamiento polimórfico, y el valor menos deseado es el 0, el cual significa que ningún método de las clases de la arquitectura es abstracto y ninguno es virtual redefinido en clases derivadas, por lo tanto, ninguno tiene un comportamiento polimórfico (Ortiz Gutiérrez, 2020).

Estas cinco métricas FHI, $FHIJ$, $FHIAC$, FFC y $FMFAC$ están sustentadas según la teoría de la medición (Zuse, 1992).

5.1.6. Medición de alto acoplamiento

Factor de Acoplamiento (COF): Esta métrica calcula el factor de acoplamiento de un sistema debido a múltiples relaciones de dependencia entre clases, incluyendo las llamadas a funciones, la creación de objetos y destrucción de objetos. (Brito Abreu, 1996; Jesús et al., 2001).

La expresión matemáticas se muestra a continuación:

$$COF = \frac{\sum_{1=1}^{TC}[\sum_{j=1}^{tc} es\_cliente(C_1, C_j)]}{TC^2 - TC} \tag{14}$$

Donde:
$\boldsymbol{TC}$ es el total de clases
$\boldsymbol{TC^2 - TC}$ es el máximo número de acoplamiento en un sistema con TC clases.

$$es\_cliente(C_c, C_s) = \begin{cases} 1 \text{ si y sólo si } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \text{ en caso contrario} \end{cases}$$

La relación cliente-servidor (Cc => Cs) significa que Cc (la clase cliente) contiene al menos una referencia no basada en la herencia a una característica (método o atributo) de la clase Cs (clase servidora). Algunas de estas relaciones cliente-servidor pueden verse como comunicaciones.

En esta investigación, el acoplamiento que se mide con esta métrica, es aquel que se da por la comunicación entre la clase cliente y la clase servidora al invocar un método (llamada a método). Es decir, no se toman en cuenta otras relaciones, por ejemplo, la declaración de objetos. No se toma en cuenta la cantidad de métodos de una clase que son invocados por otra clase, sino que todos cuentan como una sola dependencia, lo único que importa es contar las clases que tienen dependencias con otras clases en particular. Se debe evitar los valores muy altos de COF, por lo que se espera que el valor sea lo más bajo posible, es decir, un valor de COF cercano a 0 indica un sistema con bajo acoplamiento, lo cual es deseable porque mejora la modularidad y facilita el mantenimiento. Por el contrario, un valor cercano a 1 refleja un alto acoplamiento entre componentes, lo que puede dificultar la evolución del sistema (Ramírez Cruz, 2022).

5.2. Selección de Métricas

De las catorce métricas definidas en el marco de evaluación, se seleccionaron cinco para la aplicación de los métodos estadísticos: TPM (Total de Protección Modular), FA (Factor de Abstracción), NR (Número de Responsabilidades), FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases) y COF (Factor de Acoplamiento). La elección de estas métricas responde a criterios de representatividad, generalidad e independencia conceptual, que permiten capturar de manera integral los principales factores estructurales asociados a la deuda técnica en sistemas orientados a objetos.
La decisión de utilizar una única métrica representativa por factor tuvo como propósito simplificar el análisis estadístico y reducir la dependencia entre variables. Varias de las métricas restantes del marco de evaluación presentan correlaciones conceptuales con los factores seleccionados o constituyen medidas derivadas de los mismos atributos estructurales, por lo que su incorporación podría introducir redundancia sin aportar información sustancialmente diferente para los objetivos de este estudio. No obstante, esta decisión implica ciertas limitaciones. Si bien las cinco métricas seleccionadas proporcionan una cobertura amplia de los factores analizados, otros indicadores podrían capturar matices específicos de calidad estructural no considerados en el presente análisis. En consecuencia, los resultados deben interpretarse como una aproximación basada en métricas representativas de cada factor y no como una caracterización exhaustiva de todos los atributos de diseño presentes en los sistemas evaluados.

- **TPM:** Se seleccionó por su capacidad para reflejar el grado de encapsulamiento y control de acceso en el código fuente. Este indicador resume el comportamiento de las diferentes protecciones modulares (“*private”*, “*protected”*, “*public”* y “*friendly”*), por lo que constituye una métrica global de protección estructural.

- **FA:** Representa el nivel de abstracción y diseño conceptual de las clases y métodos, un aspecto clave en la mantenibilidad y comprensión del software. Su inclusión permite evaluar la deuda técnica derivada de un bajo nivel de abstracción o diseño inadecuado.

- **NR:** Mide la cantidad de tareas o funcionalidades a cada módulo o paquete de programa, lo que permite detectar violaciones al principio de responsabilidad única. Es una métrica directamente vinculada con la complejidad y cohesión interna de los módulos.

- **FHIAC:** Evalúa la profundidad y uso de la herencia en toda la arquitectura, siendo un indicador de reutilización y extensión de código. Un valor alto o bajo de manera extrema puede reflejar deuda técnica estructural, ya sea por herencia excesiva o insuficiente.

- **COF:** Se seleccionó como métrica clave para representar el grado de dependencia entre clases o módulos. Un alto nivel de acoplamiento reduce la flexibilidad y aumenta el riesgo de efectos colaterales ante cambios, constituyendo uno de los factores más reconocidos de deuda técnica estructural.

Estas cinco métricas fueron consideradas las más generales y representativas del modelo de calidad estructural propuesto, ya que abarcan los aspectos esenciales de encapsulamiento, abstracción, responsabilidad, herencia y acoplamiento, los cuales integran la base de las características de diseño que influyen directamente en la acumulación de deuda técnica.

5.3. Metodología de solución aplicada al diseño experimental

La metodología de solución que se utilizó para cumplir con el primer objetivo específico, referente a determinar la recurrencia de los factores de deuda técnica establecidos, esta metodología está definida en los siguientes pasos:

1. Inició con el diseño experimental acorde al estudio de recurrencia relativa de factores de código mal formado.
2. Definición de la pregunta de investigación.
3. Definicion de las hipótesis de trabajo.
4. Identificación de las variables dependientes, las variables independientes y variables intermitentes.
5. Establecimiento de un plan de pruebas.
6. Pruebas experimentales y recopilación de los resultados.
7. Análisis estadístico de los datos e interpretaciones necesarias para aceptar o rechazar las hipótesis establecidas.

Capítulo 6. Diseño Experimental y Plan de Pruebas

La aplicación de un diseño experimental en esta investigación responde a la necesidad de evaluar de manera sistemática los factores de código mal formado presentes en sistemas de software legado. Mediante este enfoque se define una muestra de proyectos, se establecen criterios uniformes de observación y se utilizan métricas estructurales que permiten caracterizar objetivamente los factores analizados. Asimismo, el diseño experimental proporciona un marco metodológico para la recolección, organización y análisis de los datos, favoreciendo la consistencia y reproducibilidad del estudio. De esta manera, se dispone de un procedimiento controlado que facilita la comparación de los factores identificados en la literatura con aquellos observados en proyectos desarrollados en lenguaje Java.

6.1. Pregunta de Investigación

1. ¿Cuál es la frecuencia relativa de ocurrencia de deuda técnica, como consecuencia de los siguientes factores de código mal formado “arquitecturas con carencia de protección”, “arquitecturas con carencia de abstracción”, “arquitecturas con más de una responsabilidad”, “arquitecturas con herencia de implementación” y “arquitecturas con alto acoplamiento” en una muestra de Casos de Prueba de Software Legado escritos en lenguaje java?

6.2. Generación de Hipótesis

Para el presente diseño experimental, fueron establecidas las siguientes hipótesis:

Hipótesis nula ($\boldsymbol{H_0}$): Menos del 80% de las aplicaciones de software en operación presentan, con frecuencia relativa, algún factor de deuda técnica como consecuencia de: carencia de abstracción y de protección, existencia de más de una responsabilidad, uso inapropiado de herencia de implementación y alto acoplamiento ($p <= 0.80$).

Hipótesis alternativa ($\boldsymbol{H_1}$): Entre el 80% y el 100% de las aplicaciones de software en operación, presentan con frecuencia relativa algún factor de deuda técnica como consecuencia de: Carencia de abstracción y de protección, existencia de más de una responsabilidad, uso inapropiado de herencia de implementación y alto acoplamiento ($p > 0.80$).

Hipótesis operacional: El tamaño de la muestra fue calculado con el muestreo probabilístico estratificado emanado de la teoría de los grandes números. Los datos obtenidos a partir de los 29 casos de prueba analizados son lo suficientemente significativos para obtener resultados representativos de una población muestral de 40 proyectos de software.
Demostración: Desde un enfoque estadístico, el análisis de 29 casos de prueba permite aproximar la media de los resultados al valor real del fenómeno investigado. Por ello, en el ciclo experimental se definió que este número de casos es adecuado para proporcionar datos significativos que permitan aceptar o rechazar las hipótesis estadísticas.

6.3. Formulación de un modelo general en términos de importancia

En el presente estudio experimental, se han identificado tres tipos de variables: las variables dependientes, independientes e intermitentes.

Variables independientes.
Para el presente estudio experimental se han identificado las siguientes variables independientes que serían los factores que generan deuda técnica.

1. **Arquitecturas con carencia de protección:** La falta de encapsulación expone datos internos, aumentando la vulnerabilidad del sistema al permitir accesos no controlados a los atributos y métodos; se promueve más actividad de mantenimiento para efectos de cambiar los calificadores de acceso y ganar mayor protección. Mayor actividad de mantenimiento directamente genera mayor fragilidad[3] en el sistema.

2. **Arquitecturas con carencia de abstracción:** Se refiere a clases en las que la clase base no es abstracta ni virtual, lo que impide la asociación dinámica de tipos (polimorfismo) y limita la flexibilidad y la extensibilidad del diseño. La ausencia de abstracción genera código rígido y poco reutilizable, lo que dificulta la evolución del sistema y aumenta la duplicación de código.

3. **Arquitecturas con más de una responsabilidad:** En el contexto de este estudio, una responsabilidad se define como un caso de uso o una capacidad del sistema. Se identifica a través de secuencias interactivas de métodos de clases en una arquitectura orientada a objetos, invocándose sucesivamente a partir de un método público en clase pública y termina la secuencia en el método que no invoca a otro método. Las clases o módulos que manejan múltiples responsabilidades se vuelven difíciles de modificar, incrementando la complejidad y el riesgo de introducir defectos en el sistema.

4. **Arquitecturas con herencia de implementación:** Este factor se refiere a la presencia de código de implementación ubicado en una clase base dentro de una jerarquía de herencia. Las clases derivadas acceden a dicha implementación mediante invocaciones explícitas, tales como el uso de la sentencia "*super*" en Java o del calificador de alcance "*::*" en C++ o equivalentes en otros lenguajes de programación. Estas invocaciones son conocidas como "*call forward*". El uso inadecuado de la herencia de implementación reduce la flexibilidad ante cambios funcionales, provoca un alto acoplamiento entre clases, además de violar el principio de abierto/cerrado. Como consecuencia, las modificaciones en la clase padre pueden propagarse y afectar negativamente a múltiples subclases.

5. **Arquitecturas con alto acoplamiento:** Se refiere a un diseño en el que las clases están fuertemente interconectadas a través de relaciones de agregación, composición, dependencia o herencia entre sí. Esto significa que una clase

[3] En este documento la frase "fragilidad" se refiere a la susceptibilidad de un sistema informático o una aplicación a romperse o experimentar fallas cuando se enfrenta a cambios o situaciones inesperadas. Esta fragilidad puede manifestarse de diversas formas, como errores inesperados, bloqueos, pérdida de datos o mal funcionamiento general del software (Juan, 2016).

conoce demasiados detalles o necesita directamente del comportamiento o estructura interna de otras clases para alcanzar una meta de valor u objetivo de una aplicación. La dependencia excesiva entre módulos reduce la modularidad, se aumenta la complejidad y dificulta la reutilización del código y la capacidad de realizar modificaciones sin afectar otras partes del software.

Variables dependientes.
En este estudio es la *frecuencia de deuda técnica* presente en los proyectos de software analizados. Esta frecuencia se determina a partir de los valores obtenidos mediante un conjunto de métricas específicas.

Variables intermitentes.
Para realizar un experimento robusto y confiable, es necesario aislar todos los factores externos que pudieran ocasionar ruido experimental y afectar los resultados del experimento. Algunos de los factores de ruido son los siguientes:

1. **Divulgación de los criterios y principios.** Los criterios y principios del diseño modular y orientado a objetos, se dieron a conocer a principios del año 2000, por lo que se puede esperar que el código legado anterior a esos años esté plagado de deuda técnica por los factores de código mal formado que no consideran esos principios, lo que no es de esperar en código legado posterior a estos años. Se puede aislar, para normalizar esta variable, los casos de prueba fueron seleccionados de los años 2018 a 2022.

2. **Paradigma de la programación.** Para aislar esta variable, los casos de prueba se normalizaron a solo el Paradigma de Programación Orientado a Objetos.

3. **Lenguaje.** No es comparable código escrito en lenguaje de ensamble que en algún lenguaje de alto nivel, por lo tanto, para aislar esta variable, los casos de prueba se normalizaron seleccionando solo proyectos del lenguaje java.

4. **Tamaño del código legado medido en cantidad de clases y sus relaciones.** En este proyecto todas las métricas utilizadas se encuentran normalizadas por estos factores.

5. **Experiencia del desarrollador.** Un programador experimentado puede escribir código más limpio y estructurado, reduciendo la presencia de código mal formado. Esta variable no se pudo aislar debido a que no se encuentra la información de la experiencia del desarrollador, sin embargo, los casos de prueba provienen de diferentes desarrolladores.

6. **Experiencia del desarrollador en años de práctica en lenguaje java.** Un desarrollador con más años programando en Java puede aplicar mejores prácticas, minimizando problemas estructurales en el código. Esta variable no se pudo aislar debido a que no se encuentra la información de la experiencia del desarrollador, sin embargo, los casos de prueba provienen de diferentes desarrolladores.

Las variables intermitentes que no pudieron ser aisladas son fuente potencial de un sesgo en el experimento, esto habrá que considerarlo en réplicas que se desarrollen de este experimento.

6.4. Pruebas

6.4.1. Diseño del Plan de Pruebas

El presente plan de pruebas tuvo como propósito validar un conjunto de hipótesis experimentales orientadas a identificar y analizar la presencia de factores de código mal formado en software legado escrito en lenguaje Java. A continuación, se describe el Plan de Pruebas:

1. Selección de un conjunto de proyectos de software legado.
Se identificó y seleccionó aleatoriamente un conjunto de 29 proyectos de software legado escritos en lenguaje Java derivada del cálculo de muestreo de una población total de 40 proyectos, los cuales fueron obtenidos del repositorio GitHub. Estos proyectos sirvieron como casos de prueba para el estudio.

2. Cálculo de métricas de calidad estructural
A cada uno de los proyectos seleccionados se les aplicó un conjunto de métricas específicas. El marco de métricas de la Figura 5 constituye el instrumento de prueba del estudio asociadas a los siguientes factores de código mal formado, carencia de protección, carencia de abstracción, única responsabilidad, herencia de implementación y alto acoplamiento. Estas métricas permiten cuantificar objetivamente la presencia de los factores mencionados en cada caso de prueba.

3. Registro y consolidación de resultados
Los resultados obtenidos en la etapa anterior se registraron en una matriz de datos estructurada, donde se asocia cada métrica con su respectivo proyecto, permitiendo facilitar el análisis comparativo y estadístico posterior.

4. Análisis estadístico
Se aplicaron técnicas estadísticas descriptivas para:

- Verificar la presencia de los factores de código mal formado (pruebas de hipótesis H_0 y H_1).
- Determinar la frecuencia relativa de cada factor entre los casos de prueba.

5. Interpretación de resultados
A partir de un análisis estadístico, se interpretaron los hallazgos con respecto a la validez de las hipótesis planteadas, identificando patrones relevantes y extrayendo conclusiones sobre la calidad estructural del software legado analizado.

6.4.2. Casos de Pruebas

Para este estudio se seleccionaron 40 proyectos de software legado, todos escritos en Java, obtenidos del repositorio GitHub, se consideraron como software legado aquellos sistemas previamente desarrollados susceptibles de mantenimiento y evolución.

Los casos de prueba corresponden a distintos dominios de aplicación. La selección se realizó sin criterios de inclusión/exclusión específicos, dado que el objetivo fue evaluar su calidad estructural mediante un conjunto de métricas orientadas a la detección de deuda técnica; por ello, cada proyecto se trató como un caso independiente durante la selección. Posteriormente, para el análisis comparativo, los proyectos se organizaron por tamaño según su número de clases en cuatro rangos: 1–12, 13–24, 25–36, 37–42 y 43 o más.

Dado que los proyectos provienen de un repositorio público, la muestra incluye sistemas con distintos niveles de complejidad, madurez y propósito de desarrollo. En consecuencia, algunos proyectos pueden corresponder a aplicaciones de pequeña escala o desarrollos con fines académicos, situación que podría influir en la representatividad de los resultados respecto a sistemas empresariales de gran tamaño. No obstante, esta diversidad también permite analizar la recurrencia de los factores estudiados en diferentes contextos de desarrollo y constituye una limitación que debe considerarse al interpretar y generalizar los hallazgos obtenidos.

A continuación, se describen los proyectos seleccionados.

1. **ProyectoPooJava-master.** Programa que calcula parámetros de salud nutricional y dadas medidas antropométricas, contiene 3 paquetes y 15 clases.
2. **Capital-SaludGUI-master**. Sistema desarrollado en java para brindar información en un hospital de Colombia, contiene 3 paquetes y 11 clases.
3. **Farmacia-Salud-master**. Sistema desarrollado en java para la venta en línea de medicamentos, contiene 3 paquetes y 39 clases.
4. **MiSalud-main**. Es una aplicación desarrollada en lenguaje java para una droguería, contiene 5 paquetes y 11 clases.
5. **ISSS_Salud-Java-master**. Aplicación desarrollada en java para brindar información de un centro de salud, contiene 3 paquetes y 48 clases.
6. **servicio-salud-main**. Proyecto para el servicio-salud desarrollado con java y angular como framework, contiene 6 paquetes y 20 clases.
7. **PortfolioCarnetSaludJava-master**. Aplicación de escritorio desarrollada en Java Standard Edition (Java SE) para la gestión de un carnet de salud infantil de niños de hasta 5 años. La aplicación permite registrar, consultar y administrar la información médica de los niños, como vacunas, controles médicos y consultas, contiene 3 paquetes y 16 clases.
8. **Indicadores-salud-1-java-main**. Indicadores de salud como ICC, IMC y las calorías requeridas para una persona hechas en Java con JSP y Servlets y usando el patrón *strategy* para calcular las calorías requeridas, contiene 4 paquetes y 15 clases.
9. **clinica-de-salud-java-main**. Aplicación desarrollada en java para brindar información de una clínica de salud de fisioterapia, contiene 1 paquetes y 10 clases.
10. **cuidado_salud-main1**. Aplicación Java empresarial para el apoyo, cuidado y mejoramiento de la salud, contiene 4 paquetes y 22 clases.
11. **jsf_savia_salud-master**. Creación de un módulo en Java para Savia Salud, contiene 4 paquetes y 6 clases.
12. **examen-certificacion-java-main**. Aplicación web para administrar las operaciones de una clínica de salud., contiene 9 paquetes y 15 clases.

13. **TuSaludInventario-main**. Sistema de gestión de inventario realizado en lenguaje Java para una farmacia, contiene 2 paquetes y 5 clases.
14. **GeneradorCertificados-main**. Programa en Java para generar Certificados Sanitarios realizado para un Centro de Salud, contiene 1 paquetes y 8 clases.
15. **repasoNOM-master**. Programa Java de consola que sirve para repasar las Normas Oficiales Mexicanas en el área de salud, contiene 1 paquetes y 2 clases.
16. **sistemasupervisionesunidad-desarrollo**. Sistema en Java para dejar notas de Supervisión en los Centros de Salud de la Jurisdicción Sanitaria nro.16, contiene paquetes 2 y 13 clases.
17. **sistemaHospital-main**. Proyecto en Java con manejo de bases de datos MySQL dirigido a la organización del sistema para una institución privada de salud, contiene 1 paquetes y 29 clases.
18. **agendaPacienteApp-main**. Aplicación para agendar citas a pacientes que requieran atención física o mental con profesionales de la salud, con Java utilizando Spring Tools Suite 4 y MariaDB, contiene 5 paquetes y 7 clases.
19. **RUIPI-master**. (Registro Único para la Identificación de Pacientes Indígenas) es un proyecto desarrollado en Java, que implementa el lector biométrico de huellas dactilares U.are.U DigitalPersona 4500, con el fin de proporcionar una forma eficaz de identificar pacientes de Instituciones Prestadoras de Servicios de Salud (IPS) pertenecientes a comunidades, contiene 3 paquetes y 28 clases.
20. **CentroSalud-JAVAEE--master**. Aplicación desarrollada en java para brindar información de un centro de salud, contiene 2 paquetes y 27 clases.
21. **AgroEmpresarial-SistemaDeGerenciamento-master**. Sistema de Gestión para empresas comercializadoras de insumos agrícolas donde es posible vender insumos controlando su stock con entradas y salidas y una herramienta financiera encargada de gestionar los pagos, contiene 8 paquetes y 80 clases.
22. **sicAgro-master**. Sistema comercial para el control de beneficios agrícolas de una agencia, contiene 30 paquetes y 146 clases.
23. **gestion-productos-agricolas-web-main**. Gestión de productos agrícolas, contiene 4 paquetes y 7 clases.
24. **projeto-DevAgro-main**. API REST de Java para el control de la producción agrícola con interacción entre granjas, empresas, empleados y granos, contiene 9 paquetes y 27 clases.
25. **Horticultura-master**. Sistema de control de temperatura y humedad para invernaderos agrícolas, con validación de límites mínimos y máximos. Desarrollado como parte de los estudios de Programación Orientada a Objetos en Java, contiene 1 paquete y 2 clases.
26. **exposicao_ABCEL-main**. Sistema de gestión de ferias agrícolas desarrollado en Java Spring Boot, enfocado en el registro y la evaluación de productos y productores. Utiliza una base de datos relacional y ofrece informes personalizados, contiene 8 paquetes y 37 clases.
27. **RetoJava-master**. Programa en Java - Control Clientes Automotriz, contiene 6 paquetes y 13 clases.
28. **EnsambladoraAutomotriz-master.** Aplicación de productor consumidor en una línea ensambladora automotriz simulada con Java, contiene 1 paquete y 12 clases.
29. **inventario_automotriz_java-main**. Inventario para el sector automotriz. Servicios para crear, eliminar cargos de usuario, contiene 11 paquetes y 30 clases.

30. **sistemaAutomotriz-Java-main**. Aplicación de productor consumidor en una línea ensambladora automotriz simulada con Java, contiene 1 paquete y clases 10.
31. **CentroDeServicioAutomotriz-main**. Aplicación en java para un centro automotriz contiene 10 clases.
32. **Concesionaria-main**. Aplicación en java para un centro automotriz, contiene 5 paquetes y 12 clases.
33. **Agenda_taller-main**. Este programa desarrollado con Java y MySQL brinda una solución integral para la gestión de un taller de mecánica automotriz, optimizando el control de empleados, vehículos, artículos, ventas y servicios programados, contiene 9 paquetes y 21 clases.
34. **project-minha-conta-main**. Herramienta para gestionar el consumo energético, contiene 5 paquetes y 9 clases.
35. **Influenza-sul-consumo-energetico-di-pratiche-di-programmazione-e-design-pattern-main.** Análisis de la eficiencia energética en la práctica de programas y patrones de diseño en Java, contiene 18 clases.
36. **supervisor-ufg-master**. Sistema supervisor para el monitoreo y control del uso de bancos de baterías de Li-Ion para optimizar el consumo y permitir el control del uso de energía y el análisis histórico de consumo, contiene 3 paquetes y 13 clases.
37. **Green-Energy-java—main**. Energía Verde es un proyecto desarrollado para promover el uso consciente y sostenible de la energía. Consiste en una aplicación que permite a los usuarios monitorizar el consumo energético de sus dispositivos y tomar medidas para reducirlo y aumentar la eficiencia energética, contiene 1 paquete y 1 clase.
38. **powerwise-main**. Aplicación Java Spring Boot integrada con el sistema que proporciona una API RESTful segura para registrar, consultar y gestionar datos de consumo de energía, con un enfoque en la eficiencia energética y la sostenibilidad (ESG), contiene 7 paquetes y 8 clases.
39. **API-EcoMEtric-main**. EcoMetric API es una API backend en Java Spring Boot para gestionar datos de consumo energético, monitorización y relaciones con autenticación y CRUD completo. Diseñado para analizar patrones de consumo, facilita la eficiencia energética y la sostenibilidad de las empresas, contiene 8 paquetes y 43 clases.
40. **ProyectoTransicionEnergeticaJusta-master.** Desarrollar una aplicación web integral que permita la visualización y gestión de datos sobre la producción y consumo de energía renovable a nivel global, utilizando principios de programación orientada a objetos con Java, bases de datos relacionales y tecnologías web modernas, para facilitar la transición hacia un futuro energético más sostenible, contiene 9 paquetes y 21 clases.

Capítulo 7. Ejecución del experimento y recolección de datos

Con el propósito de validar las hipótesis planteadas y evaluar la presencia de factores de código mal formado en software legado, se ejecutó el experimento sobre una población de 40 proyectos desarrollados en lenguaje Java, seleccionados y organizados por la investigadora de acuerdo con su tamaño, medido a partir de la cantidad de clases que contienen. Se realizó utilizando el número de clases como criterio de clasificación debido a que constituye una medida objetiva del tamaño estructural de los sistemas orientados a objetos. Dado que las métricas empleadas en esta investigación evalúan atributos relacionados con encapsulamiento, abstracción, acoplamiento, herencia y distribución de responsabilidades, el número de clases representa una característica directamente vinculada con la estructura interna del software y con la complejidad potencial de las relaciones entre sus componentes.
Existen otros criterios de clasificación, tales como el dominio de aplicación, la antigüedad del proyecto, la actividad del repositorio o el número de desarrolladores involucrados, estos factores introducen variables externas que no forman parte del objeto de estudio y cuya medición resulta más compleja. En contraste, el número de clases es un indicador cuantificable, independiente del contexto de desarrollo y disponible para todos los proyectos analizados, lo que permite establecer categorías comparables bajo un mismo criterio estructural.

7.1. Selección y cálculo de la muestra

Para determinar el tamaño de la muestra de la población de 40 proyectos, se utilizó muestreo probabilístico estratificado que garantiza un error estándar menor al 5 % respecto de la media poblacional, asegurando la representatividad y estabilidad de los resultados. En primer lugar, los estratos se definieron según el tamaño de los proyectos por el número de clases, agrupándolos en los rangos 1–12, 13–24, 25–36, 37–42 y 43 o más.

Para determinar el tamaño de muestra se utilizó la fórmula de (Cochran, 1977), con un nivel de confianza del 95%, $z \approx 1.96$; variabilidad máxima $p=0.5$; error permitido $E=0.10$.

Nivel de confianza del 95% (Z=1.96). Se utilizó porque es el estándar más empleado en estudios de ciencias sociales e ingenierías, al ofrecer un balance adecuado entre precisión y confiabilidad estadística (Cochran, 1977).

Proporción de máxima variabilidad (p=0.5). En ausencia de información previa sobre la proporción esperada de éxito/fracaso, se utilizó $p=0.5$, ya que este valor maximiza la varianza y, por tanto, asegura un tamaño de muestra conservador y suficiente para la estimación (Cochran, 1977; Levy & Lemeshow Stanley, 2013).

Margen de error permitido (E=0.10). Se estableció un 10% como tolerancia aceptable entre los parámetros poblacionales y muestrales. Este nivel se justifica en poblaciones pequeñas, donde un error más estricto (por ejemplo, 5%) exigiría muestras muy cercanas al censo total, perdiendo eficiencia (Scheaffer et al., 2012).

La expresión matemática se muestra a continuación:

$$n_0 = \frac{Z^2 * p * (1-p)}{E^2} \quad (15)$$

donde:

- n_0 = tamaño de muestra inicial (sin corrección).
- Z = valor de la distribución normal estándar para el nivel de confianza elegido.
- p = proporción esperada de la característica de interés.
- E = error máximo permitido.

Con esta fórmula, el tamaño inicial calculado fue n_0=96.04. Posteriormente se aplicó la corrección por población finita (CPF), dado que el tamaño total de la población es limitado (N=40).

La expresión matemática se muestra a continuación:

$$n = \frac{n_0}{1 + \frac{n_0 - 1}{N}} \quad (16)$$

obteniéndose un tamaño muestral ajustado de n =28.45, el cual se redondeó al entero superior, quedando la muestra final en 29 proyectos.

7.1.1. Asignación proporcional por estratos

Una vez definido el tamaño de la muestra, se procedió a distribuirla proporcionalmente entre los estratos definidos.

La expresión matemática se muestra a continuación:

$$n_h = n * \frac{N_h}{N} \quad (17)$$

donde:

- n_h = tamaño de la muestra asignada al estrato
- n = tamaño de la muestra total
- N_h = tamaño de la población en el estrato
- N = tamaño de la población total

De esta forma, la muestra quedó distribuida en 12 proyectos en el rango de 1-12, 9 proyectos en el rango de 13-24, 4 proyectos en el rango de 25-36, 1 proyecto en el rango de 37–42 y 3 proyectos en el rango de 43+. Cinco rangos según se muestra en la Tabla 5, respetando las proporciones de la población original.

Tabla 5: *Muestra Estratificada*

Estratos (rangos)	Proyectos	Paquetes	Clases
1-12	4.MiSalud-main	5	11
1-12	25.Horticultura-master	1	2
1-12	2.Capital-SaludGUI-master	3	11
1-12	28.EnsambladoraAutomotriz-master	1	12
1-12	34.project-minha-conta-main	5	9
1-12	15.repasoNOM-master	1	2
1-12	18.agendaPacienteApp-main	5	7
1-12	11.jsf_savia_salud-master	4	6
1-12	13.TuSaludInventario-main	2	5
1-12	9.clinica-de-salud-java-main	1	10
1-12	14.GeneradorCertificados-main	1	8
1-12	37.Green-Energy-java--main	1	1
13-24	33.Agenda_taller-main	9	21
13-24	40.ProyectoTransicionEnergeticaJusta-master	9	21
13-24	8.indicadores-salud-1-java-main	4	15
13-24	36.supervisor-ufg-master	3	13
13-24	6.servicio-salud-main	6	20
13-24	16.sistemasupervisionesunidad-desarrollo	2	13
13-24	1.ProyectoPooJava-master	3	15
13-24	7.PortfolioCarnetSaludJava-master	3	16
13-24	10.cuidado_salud-main1	4	22
25-36	29.inventario_automotriz_java-main	11	30
25-36	19.RUIPI-master	3	28
25-36	24.projeto-DevAgro-main	9	27
25-36	20.CentroSalud-JAVAEE--master	2	27
37-42	3.Farmacia-Salud-master	3	39
43+	39.API-EcoMEtric-main	8	43
43+	5.ISSS_Salud-Java-master	3	48
43+	22.sicAgro-master	30	146

7.2. Aplicación del marco de métricas.

Esta etapa consistió en la aplicación del marco de métricas definido a cada uno de los proyectos. Los resultados obtenidos fueron registrados de forma estructurada en una matriz de datos, permitiendo sistematizar la información necesaria para el posterior análisis estadístico. Dicho cálculo fue realizado a través de una herramienta automatizada desarrollada en el laboratorio de Ingeniería de Software del CENIDET (***Ver Anexo A***), la cual procesa el código fuente y generan los valores de las métricas a partir de la estructura interna de los sistemas. Con el fin de garantizar la uniformidad del análisis, todos los proyectos fueron evaluados bajo los mismos criterios y procedimientos de medición. No se realizaron ajustes manuales sobre los valores obtenidos; sin embargo, previo al procesamiento se verificó que los repositorios fueran

accesibles, compilables y contuvieran los elementos estructurales necesarios para el cálculo de las métricas definidas en esta investigación. A continuación, en la Tabla 6 se presentan los datos recolectados y su organización para la interpretación de resultados.

Tabla 6: *Matriz de datos del cálculo de las métricas*

Proyectos	Estratos (rangos)	TPM	FA	NR	FHIAC	COF
4.MiSalud-main	1-12	0.002	0.154	0.027	0	0.019
25.Horticultura-master	1-12	0	0	0.167	0	0
2.Capital-SaludGUI-master	1-12	0.002	0	0.036	0	0.064
28.EnsambladoraAutomotriz-master	1-12	0	0	0.022	0	0.076
34.project-minha-conta-main	1-12	0	0	0.015	0	0.056
15.repasoNOM-master	1-12	0.06	0	0.143	0	0
18.agendaPacienteApp-main	1-12	0	0.286	0.038	0	0
11.jsf_savia_salud-master	1-12	0.003	0.333	0.016	0.778	0.067
13.TuSaludInventario-main	1-12	0.004	0	0.053	0	0.05
9.clinica-de-salud-java-main	1-12	0	0.1	0.014	0.857	0.022
14.GeneradorCertificados-main	1-12	0.005	0	0.029	0	0.018
37.Green-Energy-java--main	1-12	0	0	0.25	0	0
33.Agenda_taller-main	13-24	0.002	0	0.009	1	0.007
40.ProyectoTransicionEnergetica Justa-master	13-24	0.003	0.19	0.043	0	0.002
8.indicadores-salud-1-java-main	13-24	0	0.133	0.017	0.8	0.01
36.supervisor-ufg-master	13-24	0.02	0.308	0.143	0	0.006
6.servicio-salud-main	13-24	0	0.2	0.009	0	0.003
16.sistemasupervisionesunidad-desarrollo	13-24	0.002	0.077	0.01	0.981	0.013
1.ProyectoPooJava-master	13-24	0.001	0.067	0.004	0.925	0.057
7.PortfolioCarnetSaludJava-master	13-24	0	0.063	0.012	0.714	0.021
10.cuidado_salud-main1	13-24	0.001	0.053	0.007	0.778	0
29.inventario_automotriz_java-main	25-36	0.001	0.267	0.014	0	0.003
19.RUIPI-master	25-36	0.001	0.036	0.004	0.812	0.019
24.projeto-DevAgro-main	25-36	0	0.192	0.006	0	0
20.CentroSalud-JAVAEE--master	25-36	0	0	0.5	0	0
3.Farmacia-Salud-master	37-42	0	0	0.003	0	0.001
39.API-EcoMEtric-main	43+	0	0.186	0.015	0	0
5.ISSS_Salud-Java-master	43+	0.001	0	0.003	0	0.014
22.sicAgro-master	43+	0	0.206	0.001	0.923	0.002

Los resultados obtenidos a partir del cálculo de las métricas de calidad estructural para los 29 casos de prueba evidencian la presencia de diversos factores de código mal formado en el software legado analizado.

A continuación, en la Tabla 7 se muestra la asociación de las métricas calculadas con los factores de código mal formado:

Tabla 7: *Asociación de métricas con factores de código mal formado*

Factor de código mal formado	Métricas asociadas en la tabla
Carencia de protección	TPM
Carencia de abstracción	FA
Más de una responsabilidad	NR
Herencia de implementación	FHIAC
Alto acoplamiento	COF

Con el fin de interpretar adecuadamente los resultados obtenidos a través del cálculo de las métricas de calidad estructural, se establecieron los valores de referencia para cada una de ellas, según la descripción de cada métrica realizada en el capítulo 5 de este documento. Estos valores permitieron identificar los rangos de tendencias hacia el mejor valor y hacia el peor valor, facilitando así la detección de posibles factores de código mal formado en los proyectos evaluados. A continuación, se presenta en la Tabla 8 dichos valores para cada métrica utilizada en este estudio.

Tabla 8: *Valores de Referencias para las métricas calculadas*

Métricas	Mejor valor	Peor valor
TPM	1	0
FA	1	0
NR	1	0
FHIAC	0	1
COF	0	1

Capítulo 8. Análisis Estadístico e Interpretación de Resultados

Se presenta el análisis estadístico e interpretación de los hallazgos en los 29 proyectos analizados. Se aplicaron las pruebas de Shapiro-Wilk para normalidad y Kruskal-Wallis para la comparación de estratos. El estudio se complementa con el análisis de Pareto, el cual determina la recurrencia relativa de los factores de código mal formado para validar las hipótesis y cuantificar la deuda técnica acumulada.

8.1. Prueba de Normalidad

Para verificar la distribución de los datos correspondientes a las métricas analizadas, se aplicó la prueba de Shapiro-Wilk (Shapiro & Wilk, 1965), la cual es una de las más utilizadas para comprobar el supuesto de normalidad en muestras pequeñas y medianas. El método consistió en ordenar la muestra de menor a mayor valor, obteniendo el nuevo vector muestral.

La expresión matemática se muestra a continuación:

$$W = \frac{\sum_{i=1}^{n} a_i \, y_i^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{18}$$

Donde:

- y_i= valores ordenados de la muestra
- a_i = coeficientes derivados de los valores esperados de una normal.
- $\bar{y}$ = media de la muestra

El procedimiento consistió en ejecutar la prueba sobre cada métrica de forma individual, obteniendo como resultados el estadístico W, el valor de significancia (p-valor) y la decisión respecto a las hipótesis planteadas:

H_0: Los datos provienen de una distribución normal.

H_1: Los datos **no** provienen de una distribución normal.

Los resultados se muestran en la Tabla 9, donde se puede observar, para cada métrica, si el valor de p> 0.05, caso en el cual no se rechaza la hipótesis de normalidad o p≤ 0.05, caso en el cual se rechaza la hipótesis de normalidad**.**

Tabla 9: *Resultados de aplicar Shapiro-Willk*

Métrica	W	p-valor	Normalidad
TPM	0.3437	0	No (Se rechaza H_0)
FA	0.8327	0.0003	No (Se rechaza H_0)
NR	0.539	0	No (Se rechaza H_0)
FHIAC	0.6528	0	No (Se rechaza H_0)
COF	0.7536	0	No (Se rechaza H_0)

En términos generales, esta prueba permitió identificar cuáles métricas cumplen con el supuesto de normalidad y cuáles no.

8.2. Selección de pruebas estadísticas no paramétrica y contraste de hipótesis

El análisis de normalidad mediante la prueba de Shapiro Wilk evidenció que en todos los casos ninguna de las métricas evaluadas presenta distribución normal. En consecuencia, se descartaron procedimientos paramétricos y se adoptó un enfoque no paramétrico para el contraste de diferencias entre dominios y para la verificación de las hipótesis planteadas.

8.2.1. Prueba Binomial Exacta

Para contrastar la proporción de aplicaciones que presentan al menos un factor de deuda técnica, se empleó la prueba no paramétrica Binomial Exacta. Este tipo de prueba resulta adecuada cuando la variable de interés es dicotómica, en este caso si hay presencia o ausencia de deuda técnica y cuando el tamaño de la muestra no es lo suficientemente grande como para aplicar aproximaciones normales.

La elección de esta prueba se justificó porque los datos analizados corresponden a conteos discretos de casos con y sin deuda técnica, y no se cumplen los supuestos de normalidad requeridos por pruebas paramétricas.

La prueba binomial exacta permitió determinar con un nivel de significancia del 5% si la proporción de proyectos con factores de deuda técnica era significativamente superior a la proporción de referencia ($p_0 = 0.80$), para el análisis.

Para contrastar las hipótesis relacionadas con la proporción de aplicaciones que presentan al menos un factor de deuda técnica, se construyó una variable binaria a partir de la evaluación directa de los cinco indicadores de deuda técnica: TPM, FA, NR, FHIAC y COF, aplicando los criterios establecidos en la Tabla 5 Valores de Referencia.

Criterios de evaluación:

- **TPM, FA, NR:** Deuda técnica cuando valor < 1 (valor ideal = 1)
- **FHIAC, COF:** Deuda técnica cuando valor > 0 (valor ideal = 0)

Se definió que un proyecto presenta deuda técnica cuando al menos uno de estos indicadores cumple con los criterios establecidos, reflejando la presencia de al menos un factor de mala práctica en el diseño del software.

A partir de esta variable dicotómica con valores "sí" o "no" según la presencia de deuda técnica, se aplicó la prueba binomial exacta de una cola (prueba de cola superior), utilizando el método de Clopper-Pearson para estimar intervalos de confianza. Este procedimiento permitió evaluar si la proporción observada de aplicaciones con deuda técnica es significativamente mayor al 80%, tal como establece la hipótesis alternativa.

- **Hipótesis nula (H_0):** p <= 0.80

- **Hipótesis alternativa (H_1):** p > 0.80

A continuación, en la Figura 6 se muestra la gráfica de Pareto que presenta la distribución de frecuencia relativa de código mal formado, donde se observa que:

- Carencia de protección está presente en el 100% de los proyectos (29 proyectos)
- Carencia de abstracción se identifica en el 100% de los proyectos (29 proyectos)
- Más de una responsabilidad está presente en el 100% de los proyectos (29 proyectos)
- Alto acoplamiento afecta al 72.4% de los proyectos (21 proyectos)
- Herencia de implementación afecta al 34.5% de los proyectos (10 proyectos)

Estos resultados indican que tres de los cinco factores de deuda técnica (carencia de protección, carencia de abstracción y más de una responsabilidad) están presentes en todos los proyectos analizados, mientras que el alto acoplamiento y el uso inapropiado de herencia presentan prevalencias menores, pero aún significativas.

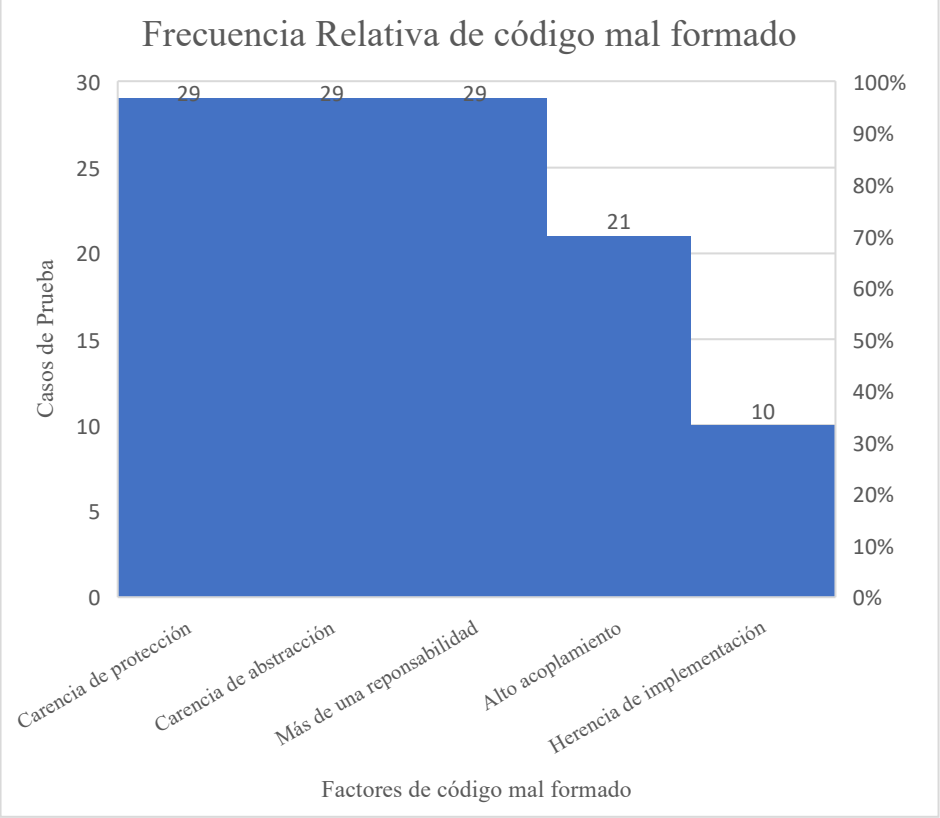

Figura 8: *Frecuencia Relativa de código mal formado*

El análisis reveló que, de los 29 proyectos evaluados, 29 (100 %) presentaron al menos un factor de deuda técnica. La prueba binomial exacta mostró un valor p de 0.0015, por lo que se rechaza H_0 la hipótesis nula a un nivel de significancia del 5%. Por lo tanto, se concluye que existe evidencia estadística para apoyar H_1 la hipótesis alternativa la cual plantea que entre el 80% y 100% de las aplicaciones en operación presentan deuda técnica como consecuencia de las malas prácticas de diseño analizadas.
La identificación de deuda técnica en el 100% de los proyectos analizados confirma la presencia generalizada de problemas de diseño en las aplicaciones de software en

operación, respaldando estadísticamente la hipótesis alternativa de alta prevalencia de deuda técnica.

8.2.2. Prueba Kruskal-Wallis

En segundo lugar, con el objetivo de comparar el comportamiento de las métricas de deuda técnica entre los diferentes estratos de proyectos, se aplicó la prueba Kruskal–Wallis (Kruskal & Wallis, 1952) debido a que los datos de las métricas estructurales no siguen una distribución normal, según los resultados obtenidos previamente en la prueba de Shapiro–Wilk. Esta característica invalida el uso de pruebas paramétricas como el ANOVA.

Las hipótesis nula y alternativa se formulan de la siguiente manera:

H_0: No hay diferencias en la distribución del índice de deuda técnica entre estratos.
H_1: Al menos un estrato difiere en su distribución del índice de deuda técnica.

Dado que las métricas analizadas son cuantitativas, no normales y con tamaños de grupo distintos entre los estratos, la prueba de Kruskal-Wallis representa una alternativa robusta y adecuada para comparar las distribuciones del índice global de deuda técnica entre los diferentes estratos de proyectos.

El análisis comparativo entre estratos se realizó por etapas, las cinco métricas (TPM, FA, NR, FHIAC, COF) se transformaron a escala 0-1, donde 0 representa ausencia de deuda técnica y 1 máxima deuda técnica, según valores de referencia preestablecidos mostrados en la Tabla 8. Esta estandarización eliminó el sesgo por dirección de valor, permitiendo consolidar un índice de deuda técnica global apto para la comparación de rangos por proyecto mediante el tratamiento estadístico de las métricas normalizadas, representando la magnitud total de deuda técnica

Una vez normalizadas, las cinco métricas contribuyeron con el mismo peso al cálculo del índice global de deuda técnica, obtenido mediante la agregación de sus valores normalizados. Esta estrategia permitió disponer de una medida única de la magnitud relativa de deuda técnica presente en cada proyecto, evitando sesgos derivados de las diferentes escalas originales de las métricas individuales.

Posteriormente, se compararon las distribuciones del índice global entre los cinco estratos definidos por tamaño del proyecto (1–12, 13–24, 25–36, 37–42 y 43 o más clases) mediante la prueba de Kruskal-Wallis. Dado que el resultado obtenido no alcanzó significancia estadística ($p = 0.565$), no se requirió análisis *post-hoc* adicional. Este análisis (que significa "después de esto") es el estudio secundario que se realiza para identificar qué pares específicos de grupos presentan distribuciones distintas, en este caso no se tuvo que realizar porque el valor de $p \geq 0.05$.

A continuación, en la Tabla 10, se presentan los resultados obtenidos al aplicar la prueba de Kruskal–Wallis a los proyectos analizados.

Tabla 10: *Resultados de Kruskal-Wallis*

Hipótesis Nula	Hipótesis Alternativa	Estadístico H	Valor p	Grados Libertad	Significancia
No hay diferencias en la distribución del índice de deuda técnica entre estratos.	Al menos un estrato difiere en su distribución del índice de deuda técnica	2.9571	0.565	4	No significativo

Los resultados obtenidos (H = 2.957, gl = 4, p = 0.565) indican que no existen diferencias estadísticamente significativas en el índice global de deuda técnica entre los estratos analizados ($p \geq 0.05$). En consecuencia, no se rechaza la hipótesis nula, lo que indica que no se encontró evidencia estadística suficiente para concluir que existen diferencias en la distribución del índice de deuda técnica entre los grupos de proyectos evaluados.

Este hallazgo implica que, independientemente del estrato al que pertenezca un proyecto, la magnitud promedio de la deuda técnica no presenta variaciones significativas, lo que podría asociarse a una similitud en las prácticas de desarrollo o en los factores estructurales de calidad entre los distintos dominios considerados.

Con el fin de complementar el análisis anterior, en la Tabla 11 se presentan las estadísticas descriptivas del índice de deuda técnica para cada estrato, incluyendo la media, mediana, desviación estándar, valores mínimo y máximo, así como la proporción de proyectos que presentan deuda técnica.

Tabla 11: *Resultado de Kruskal-Wallis por estratos de proyectos*

Estratos (rangos)	Media_Indice_Deuda	Mediana_Indice_Deuda	SD_Indice_Deuda	Min_Indice_Deuda	Max_Indice_Deuda	Proyectos_Con_Deuda
1-12	0.604	0.598	0.063	0.535	0.753	12
13-24	0.688	0.732	0.114	0.507	0.799	9
25-36	0.591	0.552	0.114	0.5	0.758	4
37-42	0.6	0.6		0.6	0.6	1
43+	0.635	0.602	0.096	0.56	0.744	3

A continuación, en la Figura 7 se muestra un gráfico de caja (*boxplot*) que representa visualmente la distribución del índice de deuda técnica normalizado para cada estrato de proyectos, permitiendo observar la dispersión, tendencia central y posibles valores atípicos dentro de cada grupo clasificados según el rango de número de clases que los conforman. En la parte superior de cada caja se indica el número total de proyectos con deuda técnica dentro de cada grupo.

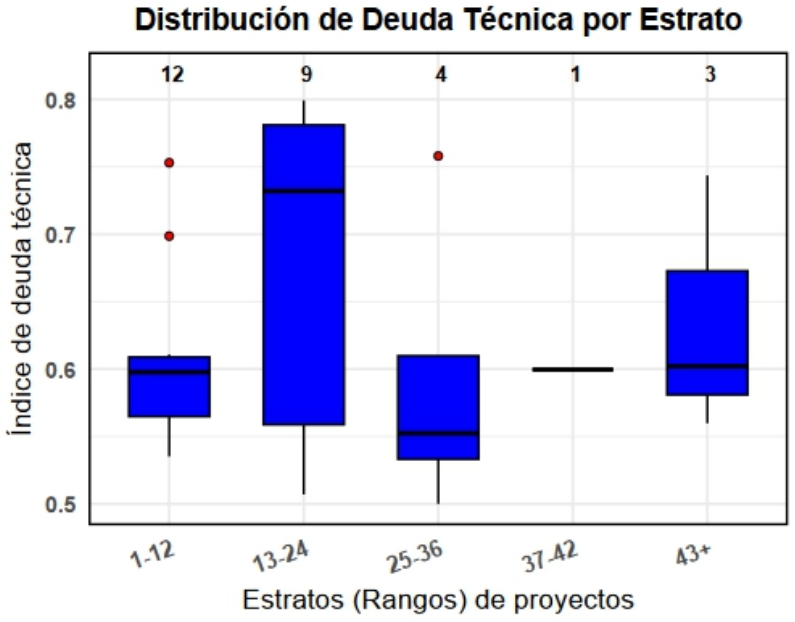

Figura 9: *Distribución del Índice de Deuda Técnica por Estrato*

Los resultados evidencian que los estratos con menor número de clases (1–12 y 13–24) concentran la mayor cantidad de proyectos con deuda técnica, con 12 y 9 casos respectivamente. En contraste, los estratos de tamaño intermedio y alto (25–36, 37–42 y 43+) presentan menos proyectos afectados (entre 1 y 4). Esto sugiere una tendencia decreciente en la incidencia de deuda técnica a medida que aumenta el tamaño del proyecto.

8.2.3. Correspondencia entre factores de código mal formado identificados en la RSL y analizados en el Diseño Experimental

Para contrastar y contextualizar los factores de código mal formado analizados en el diseño experimental, se realizó una comparación con los factores identificados previamente en la RSL (Rodríguez Ponce et al., 2025). La Tabla 12 resume esta correspondencia, agrupando los factores según su naturaleza estructural, los nombres bajo los cuales han sido reportados en estudios previos, y una breve descripción de su impacto en la calidad del software.

Tabla 12: *Correspondencia entre factores de código mal formado identificados en la RSL y analizados en el Diseño Experimental*

Factores en Diseño Experimental	Factores identificados en la RSL
Carencia de protección	Deficient Encapsulation
	Un-Encapsulated Promiscuous
	Global Data
	Inappropriate Intimacy
Carencia de abstracción	Primitive Obsession
	Type Checking
	Data Class
Más de una responsabilidad	God Class
	Divergent Change

Factores en Diseño Experimental	Factores identificados en la RSL
	Schizophrenic Class
	Swiss Army Knife
Herencia de implementación	Broken/Distorted Hierarchy
	Refused Bequest
Alto acoplamiento	Feature Envy
	Dispersed Coupling
	Intensive Coupling
	Message Chain
	Shotgun Surgery

A partir de esta correspondencia, se observó que los factores con mayor respaldo teórico como carencia de protección**,** carencia de abstracción y más de una responsabilidad también fueron los más frecuentes en los proyectos analizados, como se muestra en la Figura 8. Cada uno de estos factores se presentó en la totalidad de los casos evaluados (100%), lo que sugiere que los problemas asociados con la falta de encapsulamiento, uso de tipos primitivos y exceso de responsabilidades son predominantes en el código fuente de los proyectos estudiados.

En diferencia, los factores de alto acoplamiento (21 casos; 70%) y uso inapropiado de herencia (10 casos; 33.3%) mostraron menor frecuencia relativa, aunque mantienen relevancia al estar asociados con la propagación de errores y la complejidad estructural del sistema.

Esta relación entre los hallazgos prácticos y la evidencia teórica sugiere que los factores de calidad estructural identificados no solo se presentan de manera recurrente en contextos reales, sino que coinciden con las principales causas de degradación del diseño reportadas en la literatura.

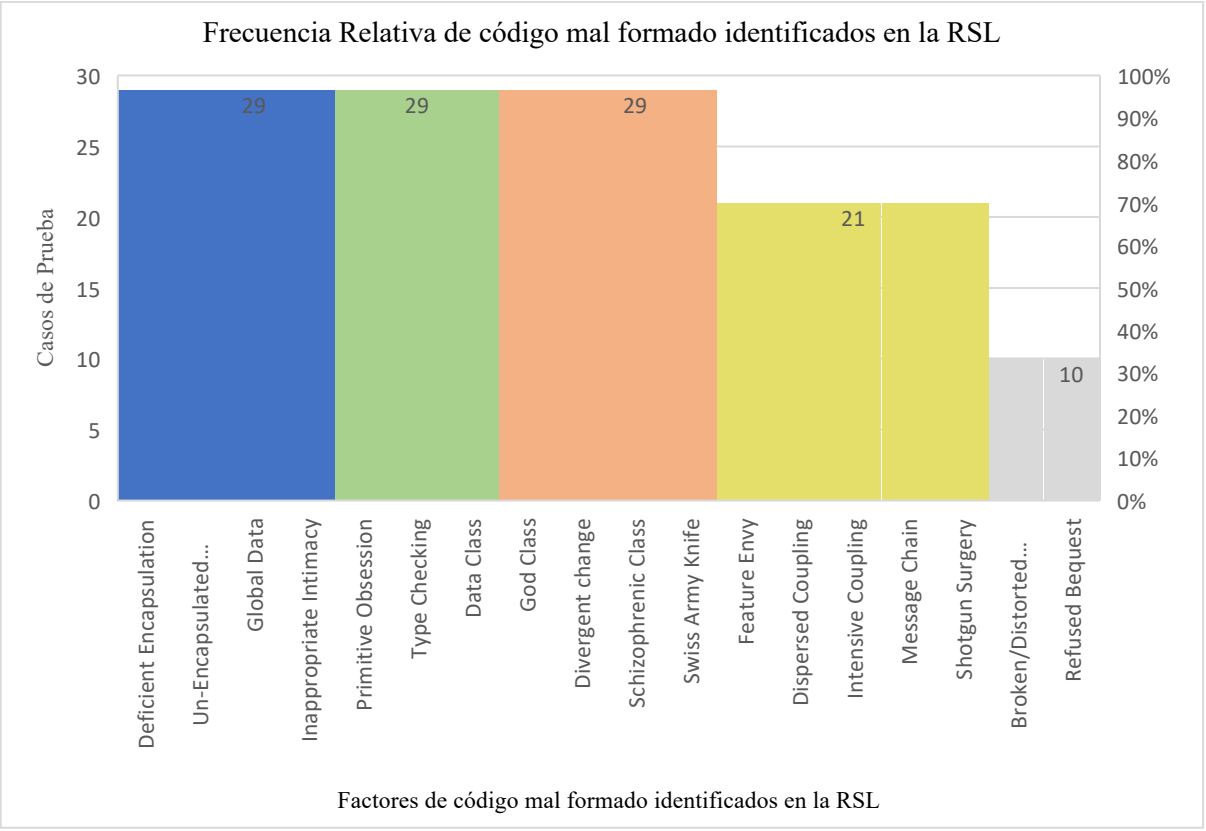

Figura 10: *Frecuencia Relativa de código mal formado identificados en la RSL*

La correspondencia observada entre los factores identificados en el diseño experimental y los códigos mal formados reportados en la RSL tiene implicaciones relevantes para la interpretación de los resultados. La coincidencia entre ambos enfoques sugiere que los problemas estructurales asociados con encapsulamiento deficiente, carencia de abstracción y distribución inadecuada de responsabilidades no constituyen fenómenos aislados de una muestra particular de proyectos, sino patrones recurrentes que han sido documentados de manera consistente tanto en estudios académicos como en sistemas reales. Este hallazgo fortalece la validez de los factores seleccionados para el estudio, ya que los resultados experimentales reproducen tendencias previamente reportadas por la comunidad científica internacional.

Por otra parte, la menor recurrencia observada en los factores asociados con herencia de implementación y alto acoplamiento no implica una menor relevancia de estos problemas, sino que podría indicar que su aparición depende de características específicas del diseño arquitectónico de cada sistema. Esta diferencia de frecuencia evidencia que no todos los factores contribuyen con la misma intensidad a la acumulación de deuda técnica, aspecto que resulta relevante para priorizar futuras estrategias de detección automática y refactorización.

En la Tabla 13, se presentan los trabajos identificados en la literatura que guardan relación directa con los factores de código mal formado y el diseño experimental de esta investigación. Los criterios de clasificación y su nomenclatura se definen a continuación:

Enfoque metodológico: Los trabajos analizados difieren en la dirección del flujo de datos para alcanzar su objetivo. El enfoque T (Top-Down) parte de modelos de diseño

o especificaciones arquitectónicas de alto nivel para identificar degradación; el enfoque B (Bottom-Up) parte del análisis estático del código fuente y métricas de software para derivar la presencia de codigo mal formado mediante ingeniería inversa, mientras que el enfoque M (Meet-in-the-Middle) realiza una integración de métricas de código con reglas de diseño de alto nivel.

Objetivo: Los trabajos analizados persiguen diferentes finalidades dentro del contexto de calidad estructural del software. La categoría D (Detección) se enfoca en identificar factores de código mal formado; P (Predicción) busca anticipar fallos o degradación estructural; E (Evaluación) analiza el impacto del código mal formado sobre calidad, mantenibilidad o seguridad; A (Automatización) incorpora mecanismos automáticos de detección y ajuste; R (Refactorización) propone mejoras estructurales del código; G (Gestión) prioriza o administra la deuda técnica; y O (Optimización) mejora el desempeño de modelos y técnicas de análisis.

Alcance: El alcance MDT (Mide Deuda Técnica), se limita a la cuantificación del esfuerzo de mantenimiento; el alcance DCS (Detecta *Code Smell* o Código mal formado), se enfoca en la identificación de factores de código mal formado; el PM (Propone Método), establece un marco de trabajo o técnica sistemática de evaluación, mientras que el alcance R (Refactorización), incluye la transformación del código para eliminar el código mal formado detectado.

Resultados: Los resultados reportados muestran beneficios principalmente orientados a la mejora del desempeño de detección y análisis estructural. La categoría MM (Mejora de métricas) refleja incrementos en precisión, recall o medida F1; PD (Precisión de detección) corresponde a modelos con mayor efectividad para identificar código mal formado; AU (Automatización) representa procesos automáticos de detección y recomendación; RC (Reducción de costos) se relaciona con disminución del esfuerzo de mantenimiento; RF (Reducción de fallos) evidencia mitigación de vulnerabilidades y defectos; PM (Propuesta de métricas/modelos) introduce nuevos indicadores o enfoques de análisis; y PR (Priorización) facilita la toma de decisiones sobre refactorización y deuda técnica.

Tabla 13: *Comparativa de trabajos relacionados de los factores de código mal formado identificados en la RSL y el Diseño Experimental*

Referencia	Enfoque Metodológico	Objetivo	Alcance	Resultados
(Zhu et al., 2023)	Deep Learning / Análisis inter-clase (B).	D	DCS	MM, PD
(Mumtaz et al., 2018)	Análisis empírico de seguridad (B).	E, R	DCS, R	RF
(Lahti et al., 2021)	Integración de *code smells* y *AntiPatterns* (M)	G	MDT, DCS, PM, R	PR, RC
(Sangeetha & Sengottuvelan, 2017)	Enfoque científico-empírico (B)	D, R	DCS, PM, R	AU
(Kaur et al., 2017)	Análisis estático basado en Support Vector Machines SVM (B)	O, D	DCS	MM, PD

Referencia	Enfoque Metodológico	Objetivo	Alcance	Resultados
(Merzah & Selçuk, 2017)	Métricas orientadas a objetos (B)	D	DCS	PM
(Paulk, 2016)	Estudio estadístico K-W y ANOVA (B)	E	DCS	PM
(Oliveira et al., 2016)	Experimento controlado (B)	E, D	DCS	MM
(Liu et al., 2016)	Adaptación dinámica por retroalimentación (B)	A, O	DCS, PM	AU, MM
(Palomba et al., 2019)	Análisis de intensidad de factores de código mal formado (B)	E, P	DCS	PD
(Liu et al., 2018)	Evaluación Effort-Aware (B)	P	DCS	PD
Esta Tesis	Análisis estático orientado al diseño (B)	D, G, E	MDT, DCS, PM	PM, PR

La revisión comparativa de los trabajos relacionados evidencia una tendencia predominante hacia enfoques Bottom-Up basados en análisis estático del código fuente, métricas orientadas a objetos, aprendizaje automático y técnicas empíricas para la detección de factores de código mal formado. Los estudios coinciden en que la combinación de métricas estructurales, modelos predictivos y procesos automatizados mejora significativamente la identificación temprana de degradación del software y la priorización de actividades de mantenimiento. Asimismo, los resultados reportan incrementos en precisión y desempeño de detección frente a métodos tradicionales, destacando la eficacia de algoritmos de Machine Learning y estrategias de refactorización para reducir vulnerabilidades, costos de mantenimiento y acumulación de deuda técnica. De manera general, se observa un claro predominio de la Detección de código mal formado, presente en la totalidad de los trabajos analizados (11 artículos). En menor proporción, los estudios incorporan propuestas metodológicas y procesos de refactorización, ambos presentes en 3 investigaciones, mientras que únicamente un trabajo aborda explícitamente la medición de deuda técnica. Esto demuestra que la mayoría de las investigaciones se concentran en actividades de identificación y análisis, más que en cuantificación integral de deuda técnica o procesos sistemáticos de corrección.

8.3. Amenazas a la validez del estudio

La experiencia, formación y conocimiento de principios de diseño orientado a objetos por parte de los desarrolladores constituyen factores que pueden influir en la calidad estructural de los proyectos analizados. Debido a la naturaleza observacional de la investigación, no fue posible controlar ni aislar esta variable durante el estudio.

Los criterios de evaluación empleados para las métricas también pueden influir en la interpretación de los resultados. En estas métricas, los valores cercanos a 1 representan una condición estructural deseable; por tanto, las desviaciones respecto a este valor reflejan distintos grados de deuda técnica, aunque no necesariamente problemas críticos de diseño. En consecuencia, los resultados deben interpretarse considerando la

magnitud de dicha desviación y no únicamente la presencia o ausencia del factor evaluado.

Finalmente, la utilización de proyectos obtenidos desde un repositorio público puede introducir variabilidad asociada a diferencias de tamaño, propósito y madurez de los sistemas evaluados, aspectos que podrían influir en la distribución observada de los factores analizados.

Capítulo 9. Conclusiones y Trabajos Futuros

9.1. Conclusiones

La presente investigación permitió analizar la recurrencia relativa de factores de código mal formado asociados a la deuda técnica en sistemas de software legado desarrollados en Java, mediante la integración de una RSL y un diseño experimental basado en métricas de calidad estructural.

El análisis experimental realizado, evidenciaron la presencia recurrente de los factores relacionados con carencia de protección, carencia de abstracción y distribución inadecuada de responsabilidades en los proyectos analizados. Estos hallazgos sugieren que las principales debilidades estructurales observadas en la muestra se encuentran asociadas con la aplicación de principios fundamentales del diseño orientado a objetos, particularmente aquellos relacionados con encapsulamiento, abstracción y asignación de responsabilidades. Asimismo, el análisis entre estratos de tamaño no proporcionó evidencia estadística suficiente para concluir que existen diferencias significativas en la distribución de la deuda técnica entre los grupos evaluados, lo que indica que la recurrencia de los factores estudiados no dependen exclusivamente del tamaño de los proyectos considerados en esta investigación.

La RSL permitió identificar una tendencia predominante hacia el estudio de factores de código mal formado relacionados con encapsulamiento deficiente, uso inadecuado de tipos primitivos, exceso de responsabilidades, acoplamiento excesivo y uso inapropiado de la herencia. La correspondencia observada entre los factores identificados en la literatura y los encontrados en el diseño experimental indica que los problemas analizados continúan siendo relevantes tanto en el ámbito académico como en contextos reales de desarrollo de software.

La integración de la evidencia obtenida mediante la RSL y el diseño experimental permitió establecer una relación entre los factores de calidad estructural evaluados y los códigos mal formados reportados con mayor frecuencia en la literatura. Esta correspondencia aporta evidencia adicional sobre la vigencia de estos problemas en sistemas orientados a objetos y respalda la utilidad de las métricas utilizadas para su caracterización y análisis.

Finalmente, los resultados de esta investigación deben interpretarse dentro del alcance definido por la muestra de proyectos analizados, las métricas empleadas y los criterios de evaluación establecidos. En este contexto, los hallazgos obtenidos contribuyen a una mejor comprensión de la recurrencia de factores asociados a deuda técnica en software legado y proporcionan una base empírica para futuras investigaciones orientadas a la detección, evaluación y mitigación de problemas de calidad estructural.

9.2. Aportaciones de la Tesis

La presente investigación aporta contribuciones en los ámbitos conceptual, metodológico, empírico y tecnológico. Desde una perspectiva conceptual, se estableció una correspondencia sistemática entre los factores de calidad estructural evaluados

experimentalmente y los principales códigos mal formados reportados en la literatura, proporcionando una base para relacionar los hallazgos empíricos con el conocimiento acumulado en el área de calidad de software y deuda técnica.

En el ámbito metodológico, se definió y aplicó un procedimiento de evaluación sustentado en las métricas TPM (Total de Protección Modular), FA (Factor de Abstracción), NR (Número de Responsabilidades), FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases) y COF (Factor de Acoplamiento), permitiendo cuantificar la recurrencia relativa de factores asociados con degradación estructural y deuda técnica en sistemas orientados a objetos.

Desde el punto de vista empírico, la investigación generó evidencia experimental a partir del análisis de 29 proyectos de software legado desarrollados en Java, contribuyendo con datos cuantitativos sobre la frecuencia relativa de los factores estudiados y su comportamiento en diferentes estratos de tamaño.

Finalmente, en el ámbito tecnológico, se emplearon herramientas de análisis estático y mecanismos automatizados para el cálculo de métricas estructurales, posibilitando una evaluación objetiva, consistente y reproducible de los atributos de calidad interna considerados en la investigación.

9.3. Trabajos a Futuro

Se propone extender el análisis a proyectos desarrollados en otros lenguajes orientados a objetos, con el fin de determinar si los patrones de recurrencia observados en Java se mantienen en diferentes entornos tecnológicos.

Se recomienda ampliar la muestra incorporando repositorios industriales y comerciales, además de proyectos de código abierto. Esto permitiría investigar si la frecuencia relativa de los factores identificados presenta diferencias significativas entre contextos académicos, comunitarios e industriales.

Otra línea de investigación consiste en analizar la influencia de variables asociadas al proceso de desarrollo, tales como la experiencia de los desarrolladores, la antigüedad del proyecto y la actividad del repositorio. En particular, resulta relevante estudiar si estos factores guardan relación con la recurrencia de problemas asociados a encapsulamiento, abstracción y distribución de responsabilidades.

Finalmente, se propone ampliar el conjunto de factores evaluados mediante la incorporación de nuevos patrones de código mal formado y otros tipos de deuda técnica, como deuda de documentación, pruebas e infraestructura, con el propósito de obtener una caracterización más completa de los mecanismos que contribuyen a la degradación estructural del software legado.

Anexo A. Aplicación del Marco de Métricas

En este anexo se documenta el proceso técnico de ejecución y cálculo de las métricas aplicadas en esta investigación. El objetivo de estas pruebas fue validar la capacidad del marco de trabajo para extraer indicadores precisos de diseño y codificación a partir de sistemas legados desarrollados en Java.

Descripción de las Fases de Ejecución

La aplicación del marco de métricas se dividió en dos fases principales, tal como se observa en las capturas de pantalla del entorno de ejecución:

1. **Selección y Carga de Proyectos (Casos de prueba)**

En la Figura 9 se muestra la interfaz de selección de los casos de prueba dentro del repositorio de sistemas legados analizados. En esta etapa, el sistema accede a la estructura de directorios de proyectos, en este caso se tomo como ejemplo dos casos de prueba como jsf_savia_salud-master y sistemasupervisionesunidad-desarrollo, asegurando que el análisis se realizara sobre el código fuente original para identificar los factores que originaban la deuda técnica.

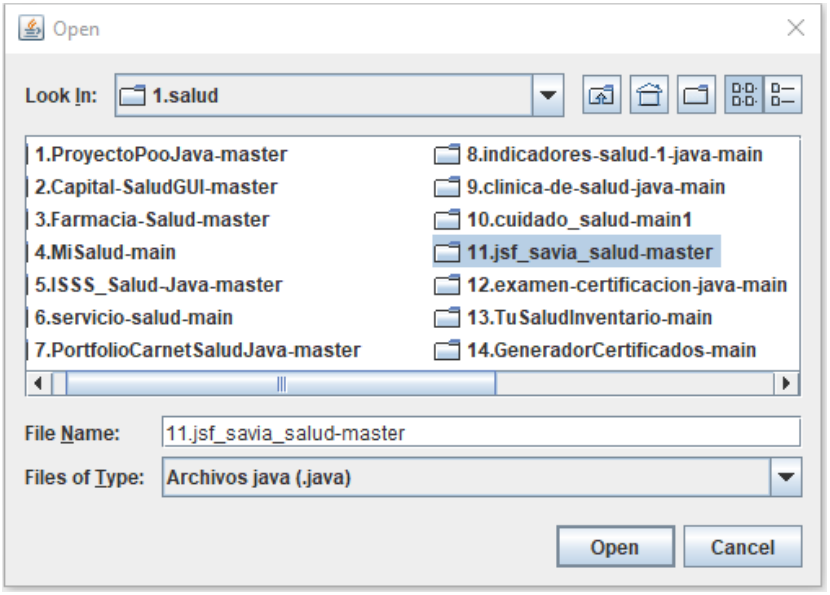

Figura 11: *Selección del Caso de Prueba*

2. **Procesamiento y Cálculo Automático**

Las Figuras 10 y 11 muestran la consola de salida tras la ejecución del algoritmo de cálculo. El sistema procesa la estructura de clases y métodos para arrojar valores normalizados en las cinco métricas estudiadas en la tesis:

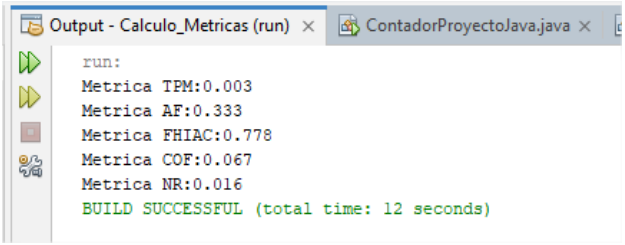

Figura 12: *Caso de Prueba 11.jsf_savia_salud-master*

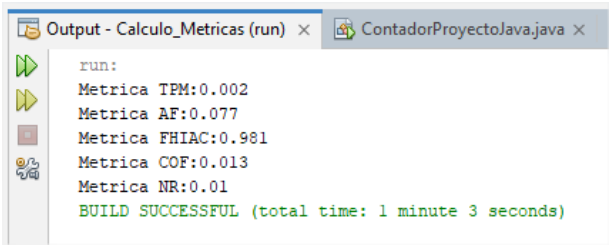

Figura 13: *Caso de Prueba 16.sistemasupervisionesunidad-desarrollo*

Los valores métricos obtenidos en estas ejecuciones, constituyen las entradas cuantitativas que se integraron en la matriz de datos principal de esta investigación. Este procedimiento de extracción se replicó de manera sistemática con el resto de los casos de prueba que conforman la muestra de sistemas legados, garantizando la homogeneidad en la recolección de los indicadores.

Referencias

Abdelaziz, T. M., Elghadhafi, H. A., & Maatuk, A. M. (2018, June 19). A novel approach for improving the quality of software code using reverse engineering. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3234698.3234729

Akash, P. S., Sadiq, A. Z., & Kabir, A. (2019). An approach of extracting God class exploiting both structural and semantic similarity. *ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, 427–433. https://doi.org/10.5220/0007743804270433

Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. (2014). Towards an ontology of terms on technical debt. *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, 1–7. https://doi.org/10.1109/MTD.2014.9

Aras, M. T., & Sel'fuk, Y. E. (2016). *Metric and Rufe Based Automated Detection of Antipatterns in Object-Oriented Software Systems*.

Arif, A., & Rana, Z. A. (2020, December 16). Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort. *2020 14th International Conference on Open Source Systems and Technologies, ICOSST 2020 - Proceedings*. https://doi.org/10.1109/ICOSST51357.2020.9332917

Bacon, Liz., Ma, Jixin., & MacKinnon, Lachlan. (2017). *An Automated Code Smell and Anti-Pattern Detection Approach*.

Bahaa-Eldin, Ayman. (2021). *Code Smells and Detection Techniques A Survey*. 411.

Barón Pérez, N. (2020). *Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet).

Bianchiotti, F., & Casas, S. (2014). *Guía para la Reingeniería de Sistemas Legados: una Experiencia Práctica y Real Revista Latinoamericana de Ingeniería de Software*. *2*(2).

Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. L., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B., Ribeiro, M., Barbosa, C., & Oliveira, D. (2020). How does incomplete composite refactoring affect internal quality attributes? *IEEE International Conference on Program Comprehension*, 149–159. https://doi.org/10.1145/3387904.3389264

Briand, L., Emam, K. E., & Morasca, S. (1996). On the Application of Measurement Theory in Software Engineering. In *Empirical Software Engineering* (1st ed., pp. 61–88).

C. Martin, R. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* (Pearson, Ed.).

Cap de Vila, A. (2021). Mantenibilidad y entropía del software. *ALBERTCAPDEVILA.NET BLOG* . https://albertcapdevila.net/mantenibilidad-entropia-software/

Cárdenas Robledo, L. A. (2004). *Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet) .

Cervantes O, J., Gómez F, M. del C., González P, P. P., & García N, A. (2016). *Introducción a la programación orientada a objetos* (1st ed.). Universidad Autónoma Metropolitana.

Clopper, C. J., & Pearson, E. S. (1934). Biometrika Trust The Use of Confidence or Fiducial Limits Illustrated in the Case of the. In *Source: Biometrika* (Vol. 26, Number 4).

Cochran, W. G. (1977). *Sampling-techniques_Cochran* (3ra Ed). Lohn Wiley & Sons.

Conover, W. J. (1999). *Practical Nonparametric Statics* (3rd ed.). Wiley.

Díaz, J. A. (2023). *Disminución de deuda técnica producida por arquitectura de clases con más de una responsabilidad.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet) .

Fernandes, S., Aguiar, A., & Restivo, A. (2024). The Impact of a Live Refactoring Environment on Software Development. *Proceedings - International Conference on Software Engineering*, 337–338. https://doi.org/10.1145/3639478.3643100

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* (Addison-Wesley, Ed.).

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2018). *Refactoring: Improving the Design of Existing Code* (Unstated Edición).

Gamma, E., Helm, R., Johnson Dr, R., & Vlissides, J. (1994). *Design Patterns Elements of Reusable Object-Oriented Software*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). *Design Patterns – Elements of Reusable Object-Oriented Software (Second).*

Ganea, G., Verebi, I., & Marinescu, R. (2017). Continuous quality assessment with inCode. *Science of Computer Programming*, *134*, 19–36. https://doi.org/10.1016/j.scico.2015.02.007

García Llinás, L. F. (2010). *Programación orientada a objetos en Java*.

Gradišnik, M., & Heričko, M. (2018, August 27). *Impact of Code Smells on the Rate of Defects in Software: A Literature Review* . http://www.webofknowledge.com/

Guggulothu, T., & Moiz, S. A. (2019). An Approach to Suggest Code Smell Order for Refactoring. *Communications in Computer and Information Science*, *985*, 250–260. https://doi.org/10.1007/978-981-13-8300-7_21

Gupta, P., Anand, A., & Mellal, M. A. (2023). Resource Allocation Modeling Framework to Refactor Software Design Smells. *International Journal of Mathematical, Engineering and Management Sciences*, *8*(2), 213–229. https://doi.org/10.33889/ijmems.2023.8.2.013

Hernández Montero, L., & Ramón Antunez, R. (2011). *Programación en paralelo: definiciones, inconvenientes y mecanismos Parallel programming: definitions, mechanisms and trouble* (Vol. 4, Number 7). http://publicaciones.uci.cu/index.php/SC|seriecientifica@uci.cu

Hernández Sampieri, R., Fernández Collado, C., & Baptista Lucio, P. (2014). *Metodología de la investigación* (6.ª ed). McGraw-Hill Education.

Hinkelmann, K., & Kempthorne, O. (1994). *Design and Analysis of Experiments. Volume 1: Introduction to Experimental Design* (John Wiley and sons, Ed.).

Hu, Jia. (2020). *Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism*.

J. Allen, M., & M. Yen, W. (1979). *Introduction to Measurement Theory* (1st Edition). Brooks/Cole Publishing Company.

Jaber, K. Mohammad. (2019). *Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability*.

Joyanes Aguilar, L., & Zahonero Martínez, I. (2007). *Estructura de Datos en Java* (Primera edición). McGraw Hill.

Juan, J. (2016, June 3). *Fragilidad del software ¿En qué estoy fallando?* https://www.genbeta.com/desarrollo/fragilidad-del-software-en-que-estoy-fallando

Kaur, A., Jain, S., & Goel, S. (2017). A Support Vector Machine Based Approach for Code Smell Detection. *Proceedings - 2017 International Conference on Machine Learning and Data Science, MLDS 2017*, *2018-January*, 9–14. https://doi.org/10.1109/MLDS.2017.8

Kiss, A., & Mihancea, P. F. (2018). Towards feature envy design flaw detection at block level. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, 544–548. https://doi.org/10.1109/ICSME.2018.00064

Kitchenham, B. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. https://www.researchgate.net/publication/302924724

Kruskal, W. H., & Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. In *Source: Journal of the American Statistical Association* (Vol. 47, Number 260).

Kumar Das, A., Yadav, S., & Dhal, S. (2019). *Detecting Code Smells using Deep Learning*. 1362.

Kutner, M. H., Nachtsheim, C. J., Neter, J., & Li, W. (2005). *Applied Linear Statistical Models* (5th ed). McGraw-Hill.

Lahti, J. R., Tuovinen, A. P., & Mikkonen, T. (2021). Experiences on Managing Technical Debt with Code Smells and AntiPatterns. *Proceedings - 2021 IEEE/ACM International Conference on Technical Debt, TechDebt 2021*, 36–44. https://doi.org/10.1109/TechDebt52882.2021.00013

Laobel, I., & Betancourt, C. (2020). *Descripción de code smells: Enfoque ontológico*.

Lárez Mata, J. J. (2020). *Calidad de Software - Deuda Técnica*. https://saber.ucab.edu.ve/handle/123456789/647

Levy, P. S., & Lemeshow Stanley. (2013). *Sampling of Populations. Methods and Applications* (4ta Ed).

Liskov, B. H., & Wing, J. M. (1994). A behavioral notion of subtyping. . *ACM Transactions on Programming Languages and Systems*, *16(6)*, 1811–1841.

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., & Zhang, L. (2021). Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, *47*(9), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

Liu, H., Liu, Q., Niu, Z., & Liu, Y. (2016). Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Transactions on Software Engineering*, *42*(6), 544–558. https://doi.org/10.1109/TSE.2015.2503740

Liu, H., Yu, Y., Li, B., Yang, Y., & Jia, R. (2018). Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Prediction? An Empirical Study. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, *2018-December*, 315–324. https://doi.org/10.1109/APSEC.2018.00046

Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices* (Prentice Hall, Ed.).

Martin, R. C. (2012). *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Addison-Wesley Professional.

Mehta, Y., Singh, P., & Sureka, A. (2018). *Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability*.

Méndez Morales, A. (2004). *Reestructuración de software escrito por procedimientos conducido por patrones de diseño composicionales* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET).

Merzah, B. M., & Selçuk, Y. E. (2017). *Metric Based Detection of Refused Bequest Code Smell*. 121.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd Edition). Prentice Hall.

Mkaouer, M. W., Kessentini, M., Cinnéide, M., Hayashi, S., & Deb, K. (2017). A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, *22*(2), 894–927. https://doi.org/10.1007/s10664-016-9426-8

Mohan, M., Greer, D., & McMullan, P. (2016). Technical debt reduction using search based automated refactoring. *Journal of Systems and Software*, *120*, 183–194. https://doi.org/10.1016/j.jss.2016.05.019

Montgomery, D. C., & Runger, G. C. (2018). *Applied Statistics and Probability for Engineers* (7th ed). Wiley.

Morelli, R., & Walde, R. (2016). *Java: Object-Oriented Problem Solving* (Third Edition). Prentice Hall.

M.Sangeetha, & P.Sengottuvelan. (2017). *A Perspective Approach for Refactoring Framework Using Monitor based Approach*.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2018). An empirical study to improve software security through the application of code refactoring. *Information and Software Technology*, *96*, 112–125. https://doi.org/10.1016/j.infsof.2017.11.010

N. Zazworka, C. Seaman, & F. Shull. (2011). Prioritizing Design Debt Investment Opportunities. *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 39.

Nyamawe, A. S., Liu, H., Niu, Z., Wang, W., & Niu, N. (2018). Recommending refactoring solutions based on traceability and code metrics. *IEEE Access*, *6*, 49460–49475. https://doi.org/10.1109/ACCESS.2018.2868990

Oliveira, R., Estácio, B., Garcia, A., Marczak, S., Prikladnicki, R., Kalinowski, M., & Lucena, C. (2016). Identifying code smells with collaborative practices: A controlled experiment. *Proceedings - 2016 10th Brazilian Symposium on Components, Architectures and Reuse Software, SBCARS 2016*, 61–70. https://doi.org/10.1109/SBCARS.2016.18

Ortiz Gutiérrez, O. (2020). *Re-Factorización de Código para Reducir el Acoplamiento entre Clases relacionadas por Herencia de Implementación en Arquitecturas Orientadas a Objetos.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet) .

Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., & Inoue, K. (2017). MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, *29*(5). https://doi.org/10.1002/smr.1843

Padilla Salgado, P. (2019). *Método de Re-factorización de código java con interfaces y abstracciones incorrectas.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet).

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., & De Lucia, A. (2018). *The scent of a smell: An Extensive Comparison between Textual and Structural Smells*. 740–740. https://doi.org/10.1145/3180155.3182530

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2019). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, *45*(2), 194–218. https://doi.org/10.1109/TSE.2017.2770122

Panigrahi, R., Kumar Kuanar, S., & Kumar, L. (2020). Application of Naïve Bayes classifiers for refactoring Prediction at the method level. *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*.

Pantiuchina, J., Lanza, M., & Bavota, G. (2018). Improving code: The (mis) perception of quality metrics. *Proceedings - 2018 IEEE International Conference on Software*

Maintenance and Evolution, ICSME 2018, 80–91. https://doi.org/10.1109/ICSME.2018.00017

Paulk, Mark. (2016). *Code smells incidence does it depend on the application domain*.

Pecorelli, F., Palomba, F., Khomh, F., & De Lucia, A. (2020). Developer-Driven Code Smell Prioritization. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 220–231. https://doi.org/10.1145/3379597.3387457

Peruma, A., AlOmar, E. A., Newman, C. D., Mkaouer, M. W., & Ouni, A. (2022). *Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring*. http://arxiv.org/abs/2203.05660

Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed).

Rahman, M. M., Riyadh, R. R., Khaled, S. M., Satter, A., & Rahman, M. R. (2018). MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design. *Software - Practice and Experience*, *48*(9), 1560–1587. https://doi.org/10.1002/spe.2591

Rahman, M., Riyadh, R. R., & Rahman, R. (2017). *Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity*.

Ramírez Cruz, M. (2022). *Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet) .

Ramírez García, E. A. (2023). *Tratamiento de la Deuda Técnica Originada por la Carencia de Protección de Funciones Plantilla de Software Legado.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet).

Rani, A., & Chhabra, J. K. (2017a). *Evolution of Code Smells over Multiple Versions of Softwares An Empirical Investigation*.

Rani, A., & Chhabra, J. K. (2017b). *Prioritization of Smelly Classes A Two Phase Approach*.

Razali, N. M., & Wah, Y. B. (2011). Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, *2*(1), 21–33.

Reeshti, Sehgal, R., Nagpal, R., & Mehrotra, D. (2019, November 21). Measuring Code Smells and Anti-Patterns . *4th International Conference on Information Systems and Computer Networks (ISCON)*.

René, S. (2003). Formal Model for Restructuring of Object-Oriented Frameworks to Architecture. *Ingeniería Investigación y Tecnología*, *15*(2), 187–198.

Rodríguez, D., & Harrison, R. (2002). *Capítulo 4 Medición en la orientación a objetos*.

Rodríguez Ponce, C. de las M., Santaolaya Salgado, R., Valenzuela Robles, B. D., & Hernández García, H. (2025). Systematic Review of Code Smell Patterns and Their Impact on Software Technical Debt. *International Journal of Combinatorial Optimization Problems and Informatics*.

Ross, S. M. (2014). *Introduction to Probability and Statistics for Engineers and Scientists* (Academic Press, Ed.; 5th ed.).

Sae-Lim, N., Hayashi, S., & Saeki, M. (2017). How do developers select and prioritize code smells? A preliminary study. *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, 484–488. https://doi.org/10.1109/ICSME.2017.66

Sánchez Rogel, F. (2023). *Re-factorización de Módulos Colaborativos con Carencia de Abstracción* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet).

Sangeetha, M., & Sengottuvelan, P. (2017). *Systematic Exhortation of Code Smell Detection Using JSmell for Java Source Code*.

Santaolaya Salgado, R. (2021). Reducción de la deuda técnica por la fragilidad modular de arquitecturas de software legado. *Cenidet/CAIS.PRY-IntD- 21.01*.

Scheaffer, R. L., Mendenhall, W., Ott, L., & Gerow, K. G. (2012). *Elementary Survey Sampling* (SEVENTH EDITION).

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). . *Biometrika*, *52*((3-4)), 591–611.

Sharma, T. (2018). Detecting and managing code smells: Research and practice. *Proceedings - International Conference on Software Engineering*, 546–547. https://doi.org/10.1145/3183440.3183460

Shen, L., Liu, W., Chen, X., Gu, Q., & Liu, X. (2020). Improving machine learning-based code smell detection via hyper-parameter optimization. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, *2020-December*, 276–285. https://doi.org/10.1109/APSEC51365.2020.00036

Singh, R., Bindal, A., & Kumar, A. (2019). A user feedback centric approach for detecting and mitigating god class code smell using frequent usage patterns. *Journal of Communications Software and Systems*, *15*(3), 245–253. https://doi.org/10.24138/jcomss.v15i3.720

Sirikul, K., & Soomlek, C. (2016). *Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs*.

Stevens, S. S. (1946). On the Theory of Scales of Measurement. *American Association for the Advancement of Science*, 677–680.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 4–15. https://doi.org/10.1145/2970276.2970340

Turkistani, B., & Liu, Y. (2019). *Reducing the Large Class Code Smell by Applying Design Patterns*.

Ubayawardana, G. M., & Karunaratna, D. D. (2018). *Bug Prediction Model using Code Smells*. 446.

Valdés Romero, M. A. (2004). *Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces.* [Investigación]. Centro Nacional de Investigación y Desarrollo Tecnológico (Cenidet)Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces. .

Velarde de Barraza, O., Murillo de Velásquez, M., Gómez de Meléndez, L., & Castillo de Krol, F. (2006). *Introducción a la Programación Orientada a Objetos* (Primera Edición). Prentice Hall.

Xu, S., Guo, C., Liu, L., & Xu, J. (2017). *A Log-linear Probabilistic Model for Prioritizing Extract Method Refactorings*.

Zhang, D., Li, B., Li, Z., & Liang, P. (2018). A preliminary investigation of self-Admitted refactorings in open source software. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, *2018-July*, 165–168. https://doi.org/10.18293/SEKE2018-081

Zhu, H., Li, Y., Li, J., & Zhang, X. (2023). An Efficient Design Smell Detection Approach with Inter-class Relation. *Proceedings of the International Conference on Software*

Engineering and Knowledge Engineering, SEKE, *2023-July*, 181–186. https://doi.org/10.18293/SEKE2023-208

Zuse, H. (1992). Properties of software measures. *Software QuaSty Journal*, *1*, 225–260.

Zuse, H. (1995). Properties of Object-Oriented Software Measures. *Proceedings of the Annual Oregon Workshop*.

Zuse, H., & Bollmann-Sdorra, P. (1992). *Measurement Theory and Software Measures* (T. Denvir, R. Herman, & R. W. Whitty, Eds.). Springer London.