EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

REDUCCION DE LA FRAGILIDAD DEL SOFTWARE LEGADO ORIGINADO POR EL ACCESO A DATOS COMPARTIDOS

presentada por

Ing. Miguel Romero Meza

como requisito para la obtención del grado de

Maestro en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Diciembre de 2022.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO

Centro Nacional de Investigación
y Desarrollo Tecnológico
Departamento de Ciencias Computacionales

Cuernavaca, Mor., 13/marzo/2024

OFICIO No. DCC/033/2024
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 **MIGUEL ROMERO MEZA** con número de control M21CE026, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"REDUCCIÓN DE LA FRAGILIDAD DEL SOFTWARE LEGADO ORIGINADA POR EL ACCESO A DATOS COMPARTIDOS"** 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.

RENE SANTAOLAYA SALGADO
Director de tesis

BLANCA DINA VALENZUELA ROBLES
Revisor 1

JUAN CARLOS ROJAS PÉREZ
Revisor 2

SEP TecNM CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
RECIBIDO
SUBDIRECCIÓN ACADÉMICA

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

EDUCACIÓN | 13 MAR 2024 | CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO | RECIBIDO

Cuernavaca, Mor., 14/marzo/2024
No. De Oficio: SAC/116/2024
Asunto: Autorización de impresión de tesis

MIGUEL ROMERO MEZA
CANDIDATO AL GRADO DE MAESTRO
EN CIENCIAS DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"REDUCCIÓN DE LA FRAGILIDAD DEL SOFTWARE LEGADO ORIGINADA POR EL ACCESO A DATOS COMPARTIDOS"**, 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 M[illegible]"

EDUCACIÓN | [illegible]
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

DEDICATORIAS

Esta tesis está dedicada a:

A mis padres (Miguel y María), por su gran amor y apoyo, por sembrar en mí los conocimientos que hoy me ponen en la situación de obtener el grado de Maestría.

A mis maestros del CENIDET que me han dado las herramientas y conocimientos para continuar con mis objetivos académicos.

AGRADECIMIENTOS

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

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

Al Consejo Nacional de Humanidades, Ciencia y Tecnología (CONAHCYT) por el apoyo económico brindado durante la realización de mis estudios de Maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por su generosidad al darme la oportunidad de trabajar en este tema de tesis, por su paciencia, confianza y consejos para culminar esta investigación.

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

A mi revisora de tesis, Dra. Blanca Dina Valenzuela Robles, por el tiempo, observaciones y comentarios puntuales en el desarrollo de este trabajo de investigación.

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

A mis compañeros que me han aportado ideas y apoyado durante el desarrollo de esta investigación.

A mis padres por el gran apoyo incondicional en mi desarrollo académico.

A mi asistente Enid Abi Leija Salvador por su apoyo, en la facilitación de tareas específicas para la recopilación de información y presentaciones de este proyecto.

Y por último quiero agradecerme por creer en mí. Quiero agradecerme por hacer todo este gran trabajo. Quiero agradecerme por no tener días libres. Quiero agradecerme por nunca renunciar. Quiero agradecerme por siempre dar y tratar de dar más sin recibir. Quiero agradecerme por tratar de hacer el bien más que el mal. Quiero agradecerme por ser yo en todo momento

RESUMEN

La programación orientada a objetos (POO) se erige como un paradigma de desarrollo de software que privilegia la interacción entre objetos en la concepción de aplicaciones informáticas, basándose en conceptos esenciales como la herencia, abstracción, polimorfismo y encapsulamiento. En este contexto, la protección modular se alza como un pilar fundamental, junto con el principio de ocultamiento de información, en la salvaguarda de la integridad y cohesión de los sistemas.

El presente trabajo de investigación se enmarca en la búsqueda de mecanismos eficaces para asegurar la protección de los atributos de clase, complementando así los esfuerzos previos centrados en la protección de funciones de clase. A través del desarrollo de un método de refactorización denominado "Método de Refactorización de Calificadores de Alcance de atributos de clase", se persigue la identificación y colocación precisa de los calificadores de alcance en los atributos de las clases que conforman una aplicación Java.

Como parte del proceso de evaluación y verificación de los objetivos propuestos, se proponen cinco métricas de calidad: PMAP (Protección Modular de Atributos Privados), PMAPr (Protección Modular de Atributos Protegidos), PMAF (Protección Modular de Atributos Friendly), PMA (Protección Modular de Atributos) y TPMA (Total Protección Modular de Atributos). Estas métricas se erigen como herramientas fundamentales para medir el grado de protección modular en diferentes niveles de visibilidad.

La validez y eficacia de los métodos propuestos se pone a prueba mediante cuatro casos de estudio. En cada uno de ellos, se aplica el cálculo del conjunto de métricas definido y el " Método de Refactorización de Calificadores de Alcance de atributos de clase ". Los resultados obtenidos reflejan una mejora significativa en la protección modular de las clases analizadas. En particular, se observa en evidencia el impacto positivo de las intervenciones propuestas en la calidad y seguridad del sistema.

Este trabajo constituye un valioso aporte en el campo de la protección modular de atributos de clase en aplicaciones Java, ofreciendo una metodología sólida y eficiente para mejorar la integridad y seguridad de los sistemas orientados a objetos.

ABSTRACT

Object-oriented programming (OOP) stands as a software development paradigm that privileges object interaction in the design of computer applications, relying on essential concepts such as inheritance, abstraction, polymorphism, and encapsulation. In this context, modular protection emerges as a fundamental pillar, along with the principle of information hiding, in safeguarding the integrity and cohesion of systems.

The present research work is framed within the search for effective mechanisms to ensure the protection of class attributes, complementing previous efforts focused on class function protection. Through the development of a refactoring method called "Class Attribute Scope Qualifier Refactoring Method", the precise identification and placement of scope qualifiers in the attributes of classes that constitute a Java application are pursued.

As part of the evaluation and verification process of the proposed objectives, five quality metrics are proposed: PMAP (Private Modular Attribute Protection), PMAPr (Protected Modular Attribute Protection), PMAF (Friendly Modular Attribute Protection), PMA (Modular Attribute Protection), and TPMA (Total Modular Attribute Protection). These metrics serve as fundamental tools to measure the degree of modular protection at different visibility levels.

The validity and effectiveness of the proposed methods are tested through four case studies. In each of them, the calculation of the defined set of metrics and the "Class Attribute Scope Qualifier Refactoring Method" are applied. The results obtained reflect a significant improvement in the modular protection of the analyzed classes. In particular, the positive impact of the proposed interventions on the quality and security of the system is evident.

This work constitutes a valuable contribution in the field of modular protection of class attributes in Java applications, offering a solid and efficient methodology to enhance the integrity and security of object-oriented systems.

CONTENIDO

Capitulo 1 – INTRODUCCIÓN

La ***refactorización*** se ha convertido en una tarea de alta importancia en los procesos de mantenimiento de software, no solamente por la intención de mejorarlo desde un punto de vista estético o perfeccionista, es más bien una estrategia definida, compuesta por un conjunto de técnicas simples que se enfocan en la búsqueda de defectos en el código legado.

El desarrollo de software ha demandado mejoras en sus procesos y entornos para lograr mejora de calidad. Actualmente, uno de los paradigmas de programación que más contribuye a la calidad de software es la Programación Orientada a Objetos (POO)[1]. Sin embargo, las reglas de este paradigma por sí solas no garantizan alta calidad. En el artículo *"patrones de diseño y anti-patrones*" (Gustavo Damián Campo, 2009) se menciona que, hacer uso de las técnicas de refactorización puede generar código más modular, más fácil de entender, de mejor diseño y, por lo tanto, con mayor calidad y con menor esfuerzo para su mantenimiento. En este mismo artículo, el autor comenta que los desarrolladores de Software generan experiencia y habilidades, y adquieren mayor capacidad en este rubro. Pero, ellos pueden incurrir en buena o mala experiencia. La buena experiencia ha dado como resultado a los patrones de diseño. La mala práctica suele producir sistemas o unidades de programa que técnicamente quedan a deber, lo cual se le conoce como ***deuda técnica*** y da como resultado anti-patrones. Los anti-patrones son soluciones que solo tendrán un impacto negativo a largo plazo en el código. Es en este punto donde la ***refactorización*** busca llevar a un estado más saludable el código.

La ***fragilidad*** y la ***rigidez*** son dos dimensiones que contribuyen a incrementar la deuda técnica. Ambas son factores de calidad del software estrechamente relacionadas. A mayor rigidez del software, menor es la capacidad de adaptación a cambios. La rigidez dificulta los cambios del software. Menor fragilidad y menor rigidez facilitan cambios ante nuevos requerimientos o nuevos comportamientos del software existente. Fragilidad se describe como la dificultad de adaptación a cambios en el software sin que se afecte al sistema total. Los cambios realizados a un módulo de un sistema de software conducen a fallas en otros módulos relacionados que aparentemente se desempeñan sin problema alguno. Conforme la fragilidad empeora, incrementa la probabilidad de que el software falle en partes inesperadas. (Juan J., 2016)

Las arquitecturas de software ***altamente acopladas*** son susceptibles a la fragilidad. Un factor que produce acoplamiento es el acceso directo o indirecto a datos de un módulo por funciones desde otros módulos externos. Esta situación obedece a que, indiscriminadamente, muchos o quizás la totalidad de los atributos o variables de instancia, o de clase en el software legado, son calificadas con el modificador de acceso *public o friendly*, lo cual, genera el acoplamiento entre módulos por el acceso indirecto a estos datos desde módulos externos.

---

[1] De este punto en adelante utilizaré la abreviatura POO para referirme al modelo de Programación Orientado a Objetos

La manera de evitar este acoplamiento es mediante el criterio de protección modular y el principio de ocultamiento de datos. Este principio establece que la información contenida en un módulo de programa deberá ser privada a ese módulo, a menos que, deliberadamente se abra la permisibilidad de acceso externo limitado. Para ello, se deben utilizar de manera correcta las reglas de visibilidad de los lenguajes de programación.

Un diseño arquitectural eficaz en el criterio de protección modular impide efectos anormales originados por cambios a los requerimientos iniciales, o por fallas originadas por defectos que ocurren en tiempo de ejecución en un módulo. Las fallas sólo se propagan a otros módulos relacionados.

En esta tesis se plantea que es necesario un preprocesamiento sobre el software legado de entrada, para asegurar que los datos de cada clase se protejan con el correspondiente modificador de acceso, de conformidad con su ámbito de alcance. Además, en este trabajo de tesis se propone reducir o eliminar la deuda técnica de sistemas de software que puede conducir a fragilidad. Esto se conducirá a través de la protección de los atributos de clases con el uso de las reglas de visibilidad de sistemas legados, desarrollados bajo el paradigma POO.

El principal trabajo de investigación desarrollado en el cuerpo académico de Ingeniería de Software del DCC, CENIDET, que es antecedente de esta propuesta de tesis, fue desarrollado por Nélida Barón (Barón Pérez, 2020). En esa tesis se aplicó el mismo concepto de protección modular, pero fue eliminado el acoplamiento producido por las dependencias funcionales entre módulos de sistemas de software. Por su parte, en este nuevo trabajo se pretende reducir el acoplamiento por el acceso indirecto a variables de módulos por otros módulos externos. Se complementa así el trabajo de investigación antecedente.

Capítulo 2 – ANTECEDENTES

2.1 PLANTEAMIENTO DEL PROBLEMA

Una *“causa de la fragilidad del software es derivada por el alto acoplamiento de clases debido al acceso indirecto de atributos de clase calificados como públicos produciendo un incorrecto nivel de encapsulamiento”.* Ante este escenario, *“el tiempo de vida útil sin fallas del software legado se reduce, debido a la deuda técnica originada por la fragilidad que exhiben estos sistemas”.* A medida que un sistema de software es modificado por los cambios continuos se incrementa su fragilidad y la probabilidad de fallas de estos sistemas es incrementada. Esta es una situación problemática causada por la violación el criterio de protección de modular y al principio de ocultamiento de información.

2.2 PROPUESTA DE SOLUCION

La solución propuesta, consiste en desarrollar un método y su automatización, para refactorizar software legado que exhiba el problema planteado anteriormente. La implementación de este método en un sistema de refactorización deberá, reducir el acoplamiento derivado del acceso indirecto de atributos de clase por entidades externas, e incrementar su protección modular a través de las reglas de ocultamiento de datos. El método se encargará de ajustar los modificadores de acceso siguiendo las normas de ocultamiento del lenguaje Java, con el fin de prevenir la creación de accesos indirectos en la estructura del código. En otras palabras, si un atributo es invocado dentro de la misma clase, se examinará su modificador de acceso y, en caso de que difiera de 'private', el método procederá a realizar la modificación correspondiente. Además, se verificarán las invocaciones 'protected' dentro de las clases derivadas, incluso cuando éstas se encuentren en otros paquetes. De igual manera, se analizarán las invocaciones 'friendly' dentro del mismo paquete y 'public' para clases no derivadas fuera del paquete. De este modo, las modificaciones en el sistema estarán directamente relacionadas con la manera en que las variables son accedidas desde entidades externas en el proyecto, lo que contribuirá a evitar cualquier interferencia en la funcionalidad inherente del sistema. El método deberá incluir un proceso de análisis estático, conducido por el *criterio de protección modular* y el *principio de ocultamiento de datos*, que localice y estime el problema de ocultamiento de información, y un proceso de generación de código con las reglas correctas de ocultamiento de información, para conseguir mejores condiciones de modularidad y encapsulamiento de objetos. El método propuesto será implementado en un sistema de software para automatizarlo.

El usuario deberá solicitar el método por medio de un menú de opciones y seleccionar la carpeta de ubicación del código sujeto a la evaluación para refactorizar, el cual deberá estar escrito en lenguaje Java en la versión 8. El sistema de refactorización deberá medir el grado de protección modular de atributos por medio de las métricas orientadas a la *Protección Modular de Atributos*. A petición del usuario deberá corregir los defectos de ocultamiento de información con el modificador de acceso correcto en los atributos de cada clase de objetos, conducido por el *criterio de protección modular* y el *principio de ocultamiento de datos*. Finalmente deberá medir nuevamente el grado de estas métricas en la arquitectura de clases refactorizada y alertar al usuario sobre el grado de mejora en estas dimensiones.

Para la refactorización del software se requiere del siguiente proceso:

- Identificar solamente archivos de tipo Java.
- Guardar una copia del archivo original.
- Establecer métricas que ayuden a evaluar la protección modular basada en los atributos de clases.

- El uso de un analizador sintáctico que permita recabar los metadatos de información necesarios para el proceso de cálculo de métricas como para el proceso de refactorización.
- Aplicar el cálculo de las métricas al código original para evaluar la necesidad de corregir la protección modular.
- De ser necesario, identificar los elementos a corregir analizando el acceso a dichos elementos desde alguna entidad interna o externa, y cambiar su modificador de acceso si fuera necesario.
- Generar una carpeta de archivos nueva que contenga el código refactorizado
- Nuevamente, aplicar el cálculo de las métricas al código refactorizado para evaluar la mejora en protección modular mediante la comparación con el resultado de las métricas aplicadas al código original.

Para implementar el método de refactorización se siguió la siguiente metodología:

- Las métricas de protección modular de métodos (PMM) declaradas en trabajo de tesis "*Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente*" (Nélida Barón, 2020) fueron estudiadas y analizadas a fin de adaptarlas o bien para generar nuevas métricas aplicables a este trabajo de tesis.
- Para analizar el código legado sujeto a estudio y recabar los metadatos necesarios tanto para el cálculo de las métricas como para el proceso de refactorización se utilizó la herramienta ANTLR (ANother Tool for Language Recognition) [PARR, 2013]. Ésta contiene un potente generador de analizadores para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. Es ampliamente utilizado para construir lenguajes, herramientas y marcos. A partir de la gramática de un determinado lenguaje de programación, ANTLR genera un analizador que puede construir y recorrer árboles de análisis sintácticos.
- ANTLR requiere que se defina un archivo "lexer", en el que se declaran todos los tokens del lenguaje. Para analizar la sintaxis, ANTLR requiere de un archivo "parser", en el cual se indica la estructura sintáctica del lenguaje por medio de producciones. ANTLR utiliza la notación EBNF (Extended Backus–Naur Form) para le definición de la sintaxis. La Notación de Bakus-Naur Extendida (EBNF) es una extensión de la notación BNF desarrollada originalmente por Niklaus Wirth. Esta notación es más expresiva y permite describir las gramáticas de manera simple.
- En este trabajo de tesis la obtención de la información necesaria para implementar la refactorización y el cálculo de las métricas orientadas a la Protección Modular de Atributos, se realizó a través de la clase *JavaBaseParserListener*. En esta clase se especifica lo que ~~se~~ debe realizarse al iniciar o terminar una regla gramatical, ya que

esta clase contiene dos métodos de cada regla gramatical, una función al iniciar la regla y otra función al terminar la regla.

- Una vez que se refactoriza el código para ganar en protección modular es necesario la generación del nuevo código, para esto se utiliza la herramienta *StringTemplate. Esta* es una librería de ANTLR que potencia la funcionalidad del metacompilador en la parte de la generación del código. *StringTemplate* es un motor de plantillas Java para generar código fuente, páginas web, correos electrónicos o cualquier otra salida de texto con formato. *StringTemplate* es particularmente bueno para generar código, las máscaras de sitios múltiples y la internacionalización/localización (Parr, 2019).
- Con respecto al uso, selección, generación y guardado de archivo se hizo uso de la libreria *JFileChooser*

2.3 JUSTIFICACION

Una unidad de software con alto valor de protección modular favorece su encapsulamiento y por tanto su reusabilidad, así como la reducción en los costos de mantenimiento a largo y corto plazo, mientras que la carencia de protección modular demerita estas propiedades e incrementa su complejidad. Cuando se refactoriza un módulo para ganar en protección modular, es necesario el ocultamiento de información, tanto de las estructuras de datos como de las funcionalidades que no son puntos de entrada a servicios o interfaces para los clientes, y que sólo son utilizadas internamente para complementar aquellos servicios requeridos a través de las funciones que si son interfaces al cliente.

La idea principal de este proyecto de tesis es refactorizar el código legado para ocultar los atributos de clases de objetos a entidades externas, de conformidad con su ámbito de alcance. La calificación apropiada de cada atributo de clase depende de la permeabilidad deliberada de los desarrolladores y las reglas de alcance *public*, *private*, *protected* y *friendly*, soportadas por el lenguaje Java. El proyecto incluye la medición del grado de protección modular del software legado, a través del cálculo de las métricas orientadas a la Protección Modular de Atributos,. Para demostrar la mejora de protección modular a través del proceso de refactorización, en este proyecto de tesis estas métricas son aplicadas antes y después de la refactorización.

Las métricas PM diseñadas y desarrolladas en el principal trabajo de antecedente, fueron ajustadas apropiadamente para que aplique a la protección modular de los atributos de las clases.

El proyecto de tesis que se propone es de desarrollo tecnológico, y pretende aportar una extensión a la herramienta SR2-Refactoring para dar soporte al sistema de transformación de software legado, el cual pretende obtener unidades de software reusable, tales como componentes, servicios web y/o microservicios en un nivel correcto de granulación, desde arquitecturas orientadas a objetos existentes y escritos en lenguaje Java.

El método que fue desarrollado en este trabajo de tesis considera que el software legado que se desea refactorizar, de origen posee cierto comportamiento, el cual tendrá que conservarse después del proceso de la refactorización.

En su estado actual, la herramienta SR2-Refactoring no cubre la refactorización de atributos de clases de objetos, para mejorar la protección modular y el encapsulamiento de las entidades de software o componentes cuyo código de origen esté escrito en el lenguaje Java. El proyecto propuesto pretende aportar en este sentido, midiendo, evaluando y refactorizando las arquitecturas de software para atender el problema de protección modular y encapsulamiento de objetos planteado.

2.4 OBJETIVOS

2.4.1 OBJETIVO GENERAL

El objetivo principal de este trabajo de tesis es Reducir la Deuda Técnica en la dimensión de fragilidad que se origina en los activos de software legado debido al acoplamiento de clases a consecuencia del acceso indirecto de atributos de clase calificados incorrectamente.

2.4.2 OBJETIVOS PARTICULARES

- Alcanzar un mayor grado de protección modular de software legado y/o de Marcos Orientados a Objetos.
- Mejorar el Diseño de Arquitecturas de Componentes de Software Existentes escritos en lenguaje Java.
- Extender la funcionalidad del “SR2-Refactoring”, para dar soporte a sus métodos de re- factorización de alto impacto en código escrito en lenguaje Java.
- Escribir un artículo en español que contenga la descripción del método y resultados obtenidos.

2.5 ESTADO DEL ARTE

2.5.1 TRABAJOS QUE ANTEDECEN

Los integrantes del cuerpo académico de Ingeniería de Software del Departamento de Ciencias Computacionales (DCC/CENIDET), han desarrollado varios métodos de

refactorización con el objetivo de mejorar el diseño de arquitecturas de software legado escritos en lenguaje Java y C++. A continuación, se detallan, cada uno de los proyectos realizados en este cuerpo académico:

S2R-Refactoring

El Sistema de Reingeniería de Software Legado para Reuso (SR2-Refactoring) es la principal herramienta de antecedente que ha estado en desarrollo. Este sistema está desarrollado en lenguaje Java, utilizando el entorno Eclipse y utilizando como soporte el gestor de bases de datos de MySQL. Actualmente en el SR2-Refactoring se han implementado varios métodos de refactorización, que juntos pueden mejorar el diseño de arquitecturas de sistemas de software legados y de marcos de aplicaciones orientados a objetos (MOOs), escritos en lenguajes Java y C ++. Los métodos de refactorización para código escrito en lenguaje C++ incluidos son:

1) “Refactorización de marcos orientados a objetos para reducir el acoplamiento aplicando el patrón de diseño Mediator” (Cárdenas Roblero, Leonor, 2004)
2) “Método de refactorización de marcos orientados a objetos por la separación de interfaces” (Valdez Marero, 2004)
3) “Adaptación de interfaces de marcos de aplicaciones orientados a objetos, usando el patrón de diseño Adapter” (Santos Castillo, 2005)

Por otro lado, los métodos de refactorización que integran al SR2-Refactoring de software escrito en lenguaje Java son:

1) “Métodos de refactorización de código Java con interfaces y abstracciones incorrectas” (Padilla Salgado, 2019)
2) “Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos” (Ortiz Gutiérrez, 2020)
3) “Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente” (Barón Pérez, 2020)

Para cumplir con el desarrollo del proyecto SR2-Refactoring, dentro del DCC/CENIDET se han planteado e implementado y se continúan realizando varios proyectos de reingeniería para la refactorización de Software Legado.

Trabajos Antecedentes

A continuación, se mencionan las tesis que se han desarrollado en el CENIDET para la refactorización de software legado.

- Fernándo Sánchez Rogel, “Refactorización de módulos colaborativos con carencia de abstracción”, tesis de maestría en desarrollo en el Tecnológico Nacional de México/Centro Nacional de Investigación y Desarrollo Tecnológico. con fecha de terminación 31 de septiembre del 2023 (Sanches Rogel,2023). Su objetivo es realizar una refactorización integral de los módulos colaborativos que actualmente presentan deficiencias en términos de abstracción.

- Elías A. Ramírez García, “Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado”, tesis de maestría en desarrollo en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 31 de marzo del 2023. (Ramírez García,2023) Su objetivo es reducir la deuda técnica de sistemas de software existentes que es producida por el código desagradable derivado de una deficiencia en la protección modular de métodos en el patrón de diseño plantilla.

- Juan Antonio Diaz Diaz, “Disminución de Deuda Técnica producida por arquitecturas de clases con más de Una Responsabilidad”, tesis de maestría en desarrollo en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 17 de marzo del 2023. (Diaz Diaz, 2023) Su objetivo es reducir la deuda técnica de sistemas de software existentes que es producida por el código desagradable derivado de arquitecturas de clases que contienen más de una responsabilidad.

- Marisol Ramírez Cruz, “Métodos de Re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 16 de diciembre del 2022. (Ramírez Cruz, 2022) El objetivo de este trabajo es mejorar el diseño de arquitecturas de componentes de software existentes en lenguaje Java mediante la reducción del acoplamiento de marcos de aplicaciones y mejora de la modularidad en suficiencia y completitud.

- Nélida Barón Pérez, “Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 11 de diciembre del 2020. (Barón Pérez, 2020) En este trabajo se trata de refactorizar el código desagradable producido por clases de objetos que tienen métodos indiscriminadamente desprotegidos del acceso externo, pero no trata la protección de los atributos de clases de objetos. La tesis que se propone en este documento es una extensión a la tesis de Nélida Barón Pérez, que específicamente trata la protección adecuada a los atributos de clases en arquitecturas orientadas a objetos.

- Orlando Ortíz Gutiérrez, “Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 15 de junio del 2020. (Ortíz Gutiérrez, 2020) El propósito de este trabajo 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.

- Pablo Padilla Salgado, “Métodos de refactorización de código Java con interfaces y abstracciones incorrectas”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 29 de julio del 2019. (Padilla salgado, 2019) En este trabajo se aplica el “principio de separación de interfaces”, automatizando su solución, para dividir las responsabilidades entre clases. El método localiza estructuras de código desagradable, 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 se corrigen automáticamente. El método elimina 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.

- Luis Esteban Santos Castillo, “Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 27 de enero del 2005 (Santos Castillo, 2005) Este método resuelve el problema de que las interfaces de comunicación entre un cliente y servidor no empatan en tipo y/o número de parámetros. El método genera de manera semi-automática adaptadores de las interfaces, para ajustarlas a las necesidades del código cliente. El método fue integrado al sistema SR2 Refactoring y aplica para código existente escrito en lenguaje C++.

- Leonor Adriana Cárdenas Robledo, “Re-factorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 19 de agosto del 2004. (Cárdenas Robledo, 2004) El objetivo de esta tesis es presentar una solución para reducir el acoplamiento de marcos de aplicaciones orientados a objetos debido al alto grado de dependencias

entre las diferentes clases del marco; incorporando a su arquitectura una estructura dirigida por el patrón de diseño ‘Mediator’.

- Manuel Alejandro Valdés Marrero, “Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha determinación 2 de diciembre del 2004. (Valdés Marrero, 2004) El objetivo principal de este trabajo de tesis es instrumentar un mecanismo basado en patrones de diseño, que permita refactorizar la arquitectura de clases de Marcos de Aplicaciones Orientados a Objetos escritos en lenguaje de programación C++ que presentan el problema de dependencia de interfaces que no son utilizadas y por lo tanto violan el principio de sustitución.

- Laura Alicia Hernández Moreno, “Factorización De Funciones Hacia Métodos De Plantilla”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha determinación 2 de febrero del 2003. (Hernández Moreno, 2003) El objetivo de este trabajo es organizar el software legado, desarrollado bajo el paradigma de programación por procedimientos, en lenguaje de programación "C", para su reuso en futuras aplicaciones del mismo dominio. Con el propósito de tener beneficios en cuanto a la productividad de los programadores, de la calidad de los sistemas de software desarrollados y para un mejor enfrentamiento a la complejidad de los actuales sistemas de software.

A continuación, se enlistan los trabajos de tesis de refactorización que actualmente se encuentran bajo desarrollo.

- Octavio Tonatiuh Flores Millán, “Tratamiento de la deuda técnica por la definición de entidades lógicas aisladas en sistemas de software existentes”, tesis de maestría en desarrollo en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico. El objetivo de este trabajo se centra en abordar la deuda técnica relacionada con el código dispensable en sistemas de software existentes. Se propone un enfoque que consiste en identificar y definir entidades lógicas aisladas dentro del código base, con el objetivo de eliminar o refactorizar el código que ha sido declarado, pero no utilizado en ninguna parte del sistema. El tratamiento de esta deuda técnica busca mejorar la mantenibilidad y la calidad del software, reduciendo la complejidad y los riesgos asociados con la presencia de código dispensable.

- Miguel Romero Meza, “Reducción de la deuda técnica por la Fragilidad Modular de arquitecturas de software legado”, tesis de maestría en desarrollo en el TecNM

/Centro Nacional de Investigación y Desarrollo Tecnológico. Este trabajo se enfoca en abordar la deuda técnica asociada con la fragilidad modular de arquitecturas de software legado, particularmente mediante el tratamiento adecuado de las variables de clase. Se propone una estrategia que consiste en aplicar el calificador de alcance correcto a las variables de clase, evitando el acceso indirecto a información privada y promoviendo una mayor cohesión y encapsulamiento en el código base. Este enfoque busca mejorar la robustez y la mantenibilidad del software, reduciendo la complejidad y los riesgos inherentes a una arquitectura frágil y poco modularizada.

- Alejandra Ruiz Bustos, "*PLUG-IN para integrar los Métodos de Refactorización del SR2-Refactoring al IDE-Eclipse",* tesis de maestría en desarrollo en el TecNM /Centro Nacional de Investigación y Desarrollo Tecnológico. Este trabajo se enfoca en la creación de un plug-in para el IDE Eclipse con el propósito de integrar los métodos de refactorización del SR2-Refactoring. La iniciativa busca hacer que estos métodos sean más accesibles a los desarrolladores durante el proceso de desarrollo de software. La integración del SR2-Refactoring en Eclipse permitirá a los desarrolladores utilizar las capacidades de refactorización de manera más eficiente y conveniente, directamente desde su entorno de desarrollo preferido. Esto promoverá una mayor adopción de prácticas de refactorización y mejorará la calidad del código en proyectos desarrollados en Eclipse.

- Ricardo Tello Diaz, "Re-factorización de arquitecturas de software con carencia de abstracciones" tesis de maestría en desarrollo en el TecNM /Centro Nacional de Investigación y Desarrollo Tecnológico. Este trabajo se centra en abordar la carencia de abstracciones en arquitecturas de software a través de la refactorización. La iniciativa tiene como objetivo mejorar la claridad y la estructura del código base mediante la introducción de abstracciones adecuadas. Se propone identificar áreas del código donde la falta de abstracción dificulta la comprensión y la modificación del software, y luego aplicar técnicas de refactorización para introducir capas de abstracción apropiadas. La incorporación de abstracciones mejorará la mantenibilidad, la extensibilidad y la reutilización del código, facilitando así el desarrollo y la evolución del software a largo plazo.

2.5.2 TRABAJOS RELACIONADOS

A continuación, se describen los trabajos relacionados al tema, derivados de una búsqueda de la literatura especializada.

“Controlling Technical Debt Remediation in Outsourced Enterprise Systems Maintenance: An Empirical Analysis”, (Ramasubbu & Kemerer, 2021)

Al tomar atajos y violar los estándares de programación, la confiabilidad del software disminuye. Ya sea por inexperiencia, por malos hábitos del programador o por querer presentar algo de manera apresurada. Esto trae consigo el aumento de deuda técnica por la acumulación de obligaciones de mantenimiento. Como consecuencia, la deuda técnica se vuelve un desafío, y aun no existen procesos bien establecidos para poder reducirla.

Múltiples empresas con el propósito de dar mantenimiento a su software cargado de deuda técnica, han recurrido a la subcontratación de otras empresas de desarrollo. Sin embargo, a raíz del factor humano esta colaboración, puede no ser la mejor opción para pagar la deuda técnica. Estas empresas subcontratadas con el propósito de brindar un mejor servicio, suelen solicitar al acceso total al proyecto, pero las empresas que las contratan suelen negarla con el propósito de proteger su propiedad intelectual. La sinergia que existe entre el cliente y el proveedor del servicio puede modificar los resultados de refactorización del software legado.

Esta investigación es una recopilación de la literatura sobre el equilibrio de la deuda técnica y el control de la información por parte de las empresas en el contexto de la subcontratación de TI para plantear la siguiente pregunta de investigación general: ¿El equilibrio del control de la información en los proyectos de mantenimiento de sistemas empresariales subcontratados mejora la remediación de la deuda técnica?

***“Refactoring for Software Architecture Smells“*, (Samarthyam, Suryanarayana & Sharma, 2017)**

La publicación de este articulo ofrece un panorama amplio de lo que es la arquitectura desagradable, del inglés *Architecture smells*. Utiliza la alegoría de una ciudad en crecimiento y desarrollo para hacer la comparativa con el software. Al ir creciendo las demandas de ésta, se tendrán que hacer modificaciones. Explica cómo el tráfico de información puede llevar a una situación de crisis cuando se tiene demasiada deuda técnica en el código. Además, se presentan las clasificaciones de la deuda técnica. Del mismo modo, la refactorización comprende las siguientes categorías:

- Implementación: alcance e impacto limitado.
- Diseño: alcance e impacto medio.
- Arquitectura: múltiples componentes abarcado y daño a nivel de sistema.

También se plantea que la refactorización es algo que le ha interesado mucho al mundo académico. En donde se han podido desarrollar diferentes procesos, técnicas, prácticas e incluso herramientas para la refactorización, pero a pesar de esto, al colectivo de ingenieros poco les llama la atención.

Un concepto interesante es el de los anti-patrones, que es como también se les llama para referirse a la *Architecture smells*. Los autores agregan las posibles direcciones que la investigación de la refactorización podría tomar dado que no existe una métrica sólida o universal que nos pueda llevar a la detección de toda la deuda técnica en un código. Se habla de un posible catálogo de todos los posibles factores que aumentan la deuda técnica, de mejorar herramientas que ya hacen un cálculo de esta deuda que a pesar de que algunas de estas herramientas están incluidas en los IDE de desarrollo tienen un desempeño pobre.

“Technical Debt Reduction using Search Based Automated Refactoring”, (Mohan, M., Greer, D., & McMullan, P., 2016)

Este artículo expone cómo la refactorización se ha vuelto un proceso valioso durante el desarrollo de software. Esencialmente se usa para pagar la deuda técnica. Al mismo tiempo, refuerza la definición de que la deuda técnica reduce la calidad del software a largo plazo debido a las malas prácticas de los desarrolladores, ya sea por desconocimiento o por presentar algo de manera rápida.

Con el propósito de examinar varias herramientas de refactorización, distintos proyectos escritos en lenguaje Java han sido examinados y refactorizados por estas herramientas. Estas herramientas a través de métricas de comportamiento y estructura presentan los resultados de la reducción de deuda técnica. Al mismo tiempo se menciona que la calidad del software no es algo que se pueda medir de manera tan fácil sin embargo se hacen uso de las métricas de detección y reducción de deuda técnica, de abstracción, acoplamiento y herencia como la mejor opción. También, menciona que las características del acoplamiento deben tener como resultado un mínimo de acoplamiento en un programa, si es posible 0. El autor maneja 2 dimensiones como una sola, el acoplamiento y la coherencia. Finalmente, el autor proporciona una lista de diferentes tipos de software que están dirigidos a la refactorización. Sin embargo, ninguno de estos está orientado a resolver el problema que presenta la investigación de la propuesta de tesis que se presenta en este documento. Estos se enfocan en diferentes áreas para resolver diversos problemas a través de los diferentes tipos de búsqueda y diferentes técnicas de refactorización presentados en el libro *“Refactoring Improving the Design of Existing Code Addison Wesley Professional”* de Martin Fowler. Los programas de refactorización mencionados en la página 4 de ese artículo se presentan en la siguiente tabla 2.1.

Tabla 1: Programas de refactorización encontrados durante la investigación

Programa	**Características**
A-CMA	Refactoriza los programas Java utilizando el código de bytes de Java como entrada. Una ventaja de esta herramienta sobre muchas otras es que tiene muchas opciones de refactorización, así como métricas disponibles y es altamente configurable. La herramienta tiene la opción de crear y seleccionar diferentes configuraciones de métricas y acciones de refactorización.
TrueRefactor	Detecta y elimina instancias de clases grandes, clases perezosas, métodos largos, campos temporales o instancias de cirugía de escopeta.
Wrangler	Mejora la modularidad de los programas eliminando los malos olores de código. La herramienta inspecciona un gráfico de módulo y un gráfico de llamada de función que genera para el programa.
Evolution doctor	Elimina dependencias circulares y reorganiza archivos de código fuente
Kirk y col	Es una herramienta de detección de olores de código que se puede utilizar como complemento para Java IDE Eclipse. La herramienta se usa para detectar clases diosas y clases de datos, pero no se puede usar para resolverlas.
Trifu y col	Han creado el Asistente de refactorización avanzada combinando tres piezas de software para manejar cada etapa del enfoque. Utilizan “estrategias de corrección” para detectar problemas en el código, analizarlos y luego refactorizarlos.
FermatT	Esta herramienta puede utilizar la búsqueda de colinas o un algoritmo genético para refactorizar el código y contiene una selección de 20 refactorizaciones disponibles para su uso.

“JCaliper: Search-Based Technical Debt Management”, (Kouros, Chaikalis, Arvanitou, Chatzigeorgiou, Ampatzoglou, Amanatidis, 2019)

Este artículo menciona que el mantenimiento del software y la evolución del mismo son actividades fundamentales dentro de la literatura de ingeniería de software. Y estos se enfrentan al obstáculo de la calidad estructural del sistema dando origen a la deuda técnica. El autor insiste en la importancia de la deuda técnica y como ésta es un problema que además de reducir la calidad del software, aumenta los costos de su mantenimiento.

Con el propósito ayudar a los desarrolladores, se presenta la herramienta *JCaliper*. Esta es una extensión del entorno de desarrollo Eclipse, que ayuda a detectar las deficiencias relacionadas con la arquitectura del software. Además de detectar este tipo de deuda técnica sugiere una estrategia viable de pago de la deuda técnica proporcionando mejoras en la

arquitectura trabajado con los patrones de diseño, *Factory method, Prototype, Singleton, Bridge, Composite, Flyweight y Strategy GoF* específicamente.

El articulo pone a prueba esta extensión a través de la *Fitness Function,* que se define como una función que toma una solución candidata al problema como entrada y produce como salida código nuevo el cuál "encaja" y evalúa que la solución sea "buena" con respecto al problema en consideración.

Este artículo, utiliza SBSE (Ingeniería de software basada en búsquedas) como el principio para evaluar la deuda técnica (es decir, la distancia entre el diseño real y el valor óptimo obtenido de la optimización del espacio de búsqueda) y propone un conjunto de técnicas de refactorización para lograrlo. Para facilitar el análisis de sistemas a gran escala, además de los conocidos algoritmos de búsqueda, se aplican algunas optimizaciones. La aplicación en 6 sistemas OSS (Operations Support Systems) muestra que existe una correlación entre la tasa de crecimiento y la evolución de la calidad. En términos generales, siempre que el número de entidades y clases aumente rápidamente, se puede observar una disminución de la calidad. Sin embargo, el equipo de diseño a menudo se las arregla para agregar funcionalidad a un Ritmo rápido sin signos de que el software está envejeciendo.

"Making Software Refactorings Safer", (Eilertsen, 2016)

Este trabajo consiste de una tesis de investigación que tiene como propósito mejorar la corrección de los casos de refactorización *Extract Local Variable* y *Extract And Move Method* en el lenguaje de programación Java, mediante la generación de comprobaciones dinámicas en forma de declaraciones de aserción. Describe el desarrollo de la prueba de concepto del Eclipse *Plugin: Safer Refactoring Plug-in*, que proporciona dos alternativas directas a la refactorización de Eclipse, y presenta los resultados del estudio de caso sobre si se puede encontrar un código Java tan extremo en el proyecto Interfaz de usuario de Eclipse JDT.

Se hace hincapié en el hecho de que los programadores utilizan la refactorización aún para cambios pequeños en el código y en cómo se apoyan de algunas herramientas que son complementos a los entornos de desarrollo. Estos complementos pueden ayudar a la medición de la deuda técnica y a proporcionar formas de refactorización. El gran problema con ellas es que son herramientas poco documentadas. La implementación de estas herramientas, puede llegar a provocar cambios en la semántica del código, que no se muestran como errores de compilación. Para poder detectar estos casos es necesario tener un conocimiento detallado sobre el lenguaje de programación sobre el que se trabaja. Es precisamente este tipo de errores lo que se busca evitar en la investigación de este proyecto de tesis.

La pregunta que plantea resolver es si *"se pueden hacer más seguras las refactorizaciones codificando las condiciones previas como comprobaciones dinámicas"* (Eilertsen, 2016). La investigación demuestra que hay casos que sólo se pueden verificar dinámicamente y se puede insertar aserciones que hacen esta verificación. Según los resultados de los experimentos, son pocos los casos en donde estas afirmaciones realmente son activadas. El

autor concluye afirmativamente a la pregunta planteada. Sin embargo, el autor recalca la necesidad de más investigaciones antes que sólo una sea útil para los desarrolladores.

"*Refactoring for Software Design Smells*", (Suryanarayana, Samarthyam & Sharma, 2015)

Este Libro se centra en el tratamiento de la deuda técnica, abordando específicamente los Problemas de Encapsulamiento (Encapsulation Smells). El libro, del cual sólo se tiene acceso a los capítulos 4 y 5 debido a restricciones de derechos de autor, explora la encapsulación con fugas, que se define como la exposición de detalles de implementación a través de la interfaz pública de una abstracción.

Los autores analizan cómo la falta de encapsulamiento efectivo puede provocar dificultades en la modificación de la implementación y permitir a los clientes acceder directamente a partes internas de los objetos, lo que podría resultar en corrupción de estado y fragilidad. Se destaca la importancia de separar la interfaz de la abstracción de su implementación para lograr una encapsulación eficaz.

Se identifica la violación del principio de encapsulamiento cuando los detalles de implementación se hacen públicos, y se señala la falta de conciencia por parte de los desarrolladores sobre lo que debería estar "oculto". Los autores abordan la experiencia y conocimientos de los programadores como elementos clave para identificar y separar los aspectos de implementación e interfaz de una abstracción.

El capítulo proporciona ejemplos concretos de encapsulamiento ineficaz, junto con posibles estrategias de refactorización y prácticas recomendadas para que los desarrolladores consideren.

En resumen, este trabajo aborda aspectos cruciales relacionados con la gestión de la deuda técnica, específicamente en el contexto de los problemas de encapsulamiento, proporcionando señalamientos valiosos y directrices prácticas para los profesionales del desarrollo de software.

"Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations", (Lacerda, Petrillo, Pimenta & Guéhéneuc, 2020)

Este artículo presenta una revisión sistemática de la literatura centrada en las investigaciones relacionadas con *code smells* y *refactoring* en el ámbito de la ingeniería de software. El objetivo principal es destacar la brecha existente entre *code smells* y la *refactorización* en el estado actual de la investigación en este campo. Los autores emplean literatura sistemática proveniente de encuestas previas, revisiones sistemáticas de literatura y mapeos sistemáticos. Identifican las principales observaciones y desafíos relacionados con los code smells y la refactorización, utilizando ocho bases de datos científicas y abordando cinco preguntas de

investigación, lo que resulta en la identificación de 40 estudios secundarios entre los años 1992 y 2018.

Los puntos sobresalientes de este estudio incluyen:

- La mayoría de los estudios abordan *code smells* y refactorización de manera independiente.
- El tema central en la mayoría de los estudios es de *code smells*.
- La aparición de *code smells* como resultado de otros *code smells* cercanos es un área que requiere mayor investigación.
- Se identifican varios enfoques de detección, principalmente basados en métricas, siendo las estrategias y reglas las más citadas. Sin embargo, la inconsistencia en los resultados de los enfoques de detección y los resultados generados es notable.
- La refactorización de extracción ha sido la más explorada en los estudios.
- Se ha investigado solo un conjunto limitado de refactorizaciones, alrededor de 27 de las 72 posibles, abriendo oportunidades para estudios que evalúen otras refactorizaciones.
- Se identifican 162 herramientas distintas para la detección de code smells, siendo CCFinder la más citada.
- Se detectan 24 herramientas de refactorización distintas, siendo JDeodorant la más citada.
- Existe una brecha en el desarrollo de herramientas de apoyo a la refactorización, particularmente en la exploración de técnicas aún no aprovechadas.
- A pesar de la disponibilidad de varias herramientas de detección de code smells, muchas de ellas están descontinuadas o presentan baja precisión.

“Constructing Models for Predicting Extract Subclass Refactoring Opportunities Using Object-Oriented Quality Metrics”, (Dallal, 2017)

El propósito de este trabajo consiste en explorar y evaluar la calidad de indicadores específicos, a saber, acoplamiento, cohesión y tamaño, tanto de manera individual como en combinación, con el fin de prever la necesidad de realizar *Extract Subclass Refactoring* (ESR) en clases que presentan baja cohesión, alto acoplamiento o un tamaño significativo. Este tipo de refactorización implica la división del software mediante la extracción de subclases. Para lograr este objetivo, se emplea un análisis de regresión logística invariable, que identifica las clases que justifican el proceso de reconstrucción. El método se diseñó para su implementación en la fase de mantenimiento del software y tiene la capacidad de clasificar el sistema según el grado de demanda del método ESR. Los resultados del análisis revelan una

relación estrecha entre los atributos internos de calidad de la clase y la necesidad de aplicar ESR.

“Performance-Driven Software Model Refactoring” (Arcelli, Cortellessa, & Pompeo, 2018)

El propósito central de esta investigación es la eliminación de anti-patrones que inciden en el rendimiento a través de un marco especializado en la refactorización de modelos de software. La plataforma EPSILON se implementa como el marco de referencia para llevar a cabo este proceso. Este framework ofrece la capacidad de examinar múltiples trayectorias y presenta diversas soluciones para abordar los anti-patrones identificados. Los resultados obtenidos respaldan la afirmación de que la automatización de la refactorización del modelo de software, centrada en el rendimiento, puede generar beneficios significativos.

“Managing Technical Debt in Software Engineering” (Avgeriou, Kruchten, Ozkaya, & Seaman, 2016)

Este artículo presenta el programa y los resultados del Seminario Dagstuhl 16162, titulado "Gestión de la Deuda Técnica en Ingeniería de Software". El evento reúne a investigadores, profesionales y proveedores de herramientas de la academia e industria con interés en la base teórica de la deuda técnica y su gestión, abarcando desde la medición y el análisis hasta la prevención. El proyecto que se describe en este artículo tiene como objetivo establecer un entendimiento común de los conceptos clave de la deuda técnica y construir una hoja de ruta para investigaciones futuras, abordando las siguientes preguntas:

- ¿Cómo definimos y modelamos la deuda técnica?
- ¿Cómo gestionamos la deuda técnica?
- ¿Cómo establecemos una base empírica y de ciencia de datos para la deuda técnica?

Los autores destacan claramente lo siguiente: *"La metáfora de la deuda técnica proporciona un mecanismo eficaz para comunicar los intercambios de diseño entre los desarrolladores y otros tomadores de decisiones. Cuando se gestiona eficazmente, la deuda técnica ofrece una forma de medir la capacidad de mantenimiento actual de sistemas y corregir el rumbo cuando ese nivel no es deseable. Aunque disciplinas como la sostenibilidad, el mantenimiento y la evolución del software, la refactorización, la calidad del software y la ingeniería de software empírica han aportado resultados relevantes para la gestión de la deuda técnica, ninguna de ellas por sí sola es suficiente para modelar, gestionar y comunicar las diversas facetas de los problemas de compensación de diseño involucrados en la gestión de la deuda técnica"*.

“Cohesion & Coupling Contents” (和太, 2018)

En este artículo, el autor aborda la noción de encapsulamiento, señalando la ambigüedad con la que a menudo se utiliza este término. Se distingue entre el ocultamiento de información, considerado como un principio, y la encapsulación, entendida como una técnica. La definición propuesta para el ocultamiento de información es la de *"ocultar el diseño de almacenamiento físico de los datos, de modo que cualquier cambio se limite a un subconjunto reducido del programa total"*. En contraste, la encapsulación se define como *"el proceso de compartimentar los elementos de una abstracción que constituyen su estructura y comportamiento; la encapsulación sirve para separar la interfaz contractual de una abstracción y su implementación"*.

Se incorpora el concepto de cohesión, entendida como el *"grado en el que los elementos de un módulo pertenecen juntos"*. Además, se presenta el acoplamiento, definido como *"la forma y el grado de interdependencia entre los módulos de software"*. Ambos conceptos son introducidos por Larry Constantine y se contrastan en el contexto de buenas prácticas de programación. Durante la investigación se hacen un análisis sobre los tipos de acoplamientos los cuales se describen en la tabla 2.

Tabla 2: Tipos de acoplamientos encontrados en la investigación

Tipo	**Descripción**
Acoplamiento de contenido	También conocido como Acoplamiento patológico, ocurre cuando un módulo depende o se basa en el funcionamiento interno de otro módulo (por ejemplo, acceder a datos locales de otro módulo). Cambiar la forma en que el segundo módulo produce datos conducirá a cambiar el módulo dependiente. (和太, 2018)
Acoplamiento común	También conocido como Acoplamiento global, ocurre cuando dos módulos comparten el mismo conjunto de datos globales (por ejemplo, una variable global). Cambiar el recurso compartido implica cambiar todos los módulos que lo utilizan. (和太, 2018)
Acoplamiento externo	Ocurre cuando dos módulos comparten un formato de datos impuesto externamente, protocolo de comunicación o interfaz del dispositivo. Esto está relacionado con la comunicación con herramientas y dispositivos externos. (和太, 2018)
Acoplamiento de control	Un módulo controla el flujo de otro, pasándole información sobre qué hacer (por ejemplo, pasar una bandera de qué hacer). (和太, 2018)

Acoplamiento de sellos	Se produce cuando los módulos comparten una estructura de datos compuestos y usan solo una parte, posiblemente una parte diferente (por ejemplo, pasar un registro completo a una función que solo necesita un campo). Esto puede llevar a cambiar la forma en que un módulo lee un registro. (和太, 2018)
Acoplamiento de datos	Ocurre cuando los módulos comparten datos a través de parámetros. Cada dato es una pieza elemental, y estos son los únicos datos compartidos (por ejemplo, pasar un número entero a una función que calcula una raíz cuadrada). (和太, 2018)
Acoplamiento de mensajes	Este es el tipo de acoplamiento más flojo, logrado mediante la descentralización estatal y la comunicación de componentes a través de parámetros o paso de mensajes. (和太, 2018)
Sin acoplamiento	Los módulos no se comunican en absoluto entre sí. (和太, 2018)

Además, se señala que el acoplamiento puede afectar el rendimiento del sistema, ya sea débil o estrechamente acoplado. La investigación destaca que, para optimizar el rendimiento en tiempo de ejecución, se debe minimizar la longitud del mensaje y maximizar su significado. Un enfoque sugerido para reducir el acoplamiento es el Diseño funcional, que busca limitar las responsabilidades de los módulos a lo largo de la funcionalidad. El artículo concluye resaltando cómo el acoplamiento aumenta entre dos clases A y B en diversas situaciones definidas por la relación entre ambas clases.

“Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering” (Rathee & Chhabra, 2017)

El propósito de este informe de investigación es presentar una nueva métrica de correlación para sistemas de software orientados a objetos, denominada Asociación de Construcción de Patrones de Uso (UPBC, por sus siglas en inglés). En este contexto, un módulo se define como un conjunto de clases cuya coherencia general se busca mejorar. La medida del modelo de uso frecuente (FUP, por sus siglas en inglés) se calcula a partir de la interacción de diversas funciones de pertenencia. El valor de cohesión se emplea para orquestar el ensamblaje de unidades, con el objetivo de incrementar la cohesión y reducir el acoplamiento entre ellas. Este proceso de agrupación se basa en un algoritmo innovador denominado "FUPClust" (clustering basado en modelos de uso frecuente), fundamentado en las interacciones FUP entre módulos.

“Refactoring effect on internal quality attributes: What haven’t they told you yet?” (Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, Willian Oizumi, 2020)

El propósito de esta investigación es evidenciar los beneficios de la refactorización, no solo en términos de los atributos de cohesión y acoplamiento, sino también en relación con la complejidad, herencia y tamaño. A menudo, la literatura presupone que los cambios introducidos en el código mediante la refactorización son beneficiosos. Pero para alcanzar mejoras significativas se requiere lo que se conoce como re-refactoring. A través del análisis de los cinco atributos mencionados anteriormente, se emplea un enfoque descriptivo y pruebas para comprender el efecto de la refactorización.

El análisis revela que las operaciones de refactorización mejoran atributos relacionados con el tipo específico de refactorización aplicado. Cuando se implementan operaciones de refactorización, estos atributos tienden a experimentar mejoras, o al menos, no deterioran la calidad del software. Esta tendencia positiva se observa incluso al agregar características adicionales a la refactorización.

“Defaultification Refactoring: A Tool for Automatically Converting Java Methods to Default” (Khatchadourian, 2017)

Este trabajo presenta una herramienta denominada "GENERACIÓN DE IMPLEMENTACIÓN A INTERFACES", la cual adopta un enfoque de refactorización automática para transformar código heredado escrito en Java, permitiendo su adaptación a métodos por defecto (default methods). Estos métodos virtuales se emplean para modificar la funcionalidad de las interfaces de la biblioteca y asegurar la compatibilidad binaria del código refactorizado con versiones anteriores del código escrito para las interfaces.

La herramienta se integra con el entorno de desarrollo Eclipse y ofrece un código refactorizado que es lingüísticamente equivalente al original. Además, el código resultante es más conciso, comprensible, menos complejo y más modular. Se realiza una comparación con la herramienta "PULL UP METHOD", ya que ambas buscan la reducción del código fuente. A diferencia de la herramienta "SKELETAL INTERFACE TO INTERFACE", que permite que las clases hereden múltiples definiciones de interfaz. La herramienta se presenta como código abierto para Eclipse, y actualmente se está explorando a fondo la relación entre esta herramienta y la aplicación de reconstrucción "PULL UP MEMBER".

“Refactor a field to a property” (Microsoft Docs, 2021)

Este artículo aborda la implementación de herramientas de desarrollo de código, específicamente plugins y utilidades diseñados para simplificar y mejorar la calidad del software, sirviendo como apoyo para los programadores. Microsoft se suma a esta tendencia con una función de refactorización centrada en el encapsulamiento. Diseñada para lenguajes

como C# y Visual Basic, esta función permite a una clase que tiene atributos declarados de manera pública cambiar su modificador de acceso al tipo private, generando simultáneamente dos nuevos métodos públicos: uno para obtener el valor y otro para asignar uno.

A pesar de brindar protección modular a los atributos, la creación de dos métodos con modificadores de acceso público introduce un nuevo elemento con una carencia de protección modular. Es esencial señalar que esta no es la solución propuesta por la investigación en cuestión, pero ejemplifica la relevancia de abordar esta problemática.

“Unveiling process insights from refactoring practices” (Joao Caldeira, Fernando Brito e Abreu, Jorge Cardoso, Jose Pereira dos Reis, 2020)

En este artículo, los autores conceptualizan el proceso de refactorización como una actividad que requiere una estimación de esfuerzo. Este esfuerzo abarca aspectos como la comprensión del software, la detección de oportunidades de refactorización y la predicción del tiempo necesario para la resolución. La identificación de componentes a refactorizar y la elección del método a adoptar continúan siendo desafíos clave en la investigación, ya que las diversas formas de refactorización impactan las consecuencias y frecuencias de estas actividades.

El estudio también destaca el crecimiento y la diversidad de los repositorios de software, que abarcan distintos lenguajes y frameworks. Se revela que proyectos en diversos sectores pueden estar escritos en entre 5 y 7 lenguajes de programación diferentes.

El artículo aborda el caso de la refactorización manual frente a la automática. La refactorización automática demostró reducir la complejidad del software de manera más eficiente y con procesos más simples en comparación con la refactorización manual. Los programadores utilizaron herramientas nativas de Eclipse para la refactorización manual. Además, se identificaron correlaciones entre la complejidad del software y la complejidad del proceso. A través de métricas impulsadas por procesos, se desarrollaron modelos predictivos destinados a anticipar el tipo de método de refactorización más adecuado para el software.

“How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow” (Anthony Peruma, Steven Simmons, Eman Abdullah, AlOmar, Christian D. Newman, Mohamed Wiem, Mkaouer, Ali Ouni , 2021)

Este artículo tiene como objetivo examinar la refactorización como un caso de estudio dentro de las plataformas de asistencia a desarrolladores de código. Es bien conocido que el proceso de refactorización que es esencial para el mantenimiento y la evolución del software. Sin embargo, debido a la complejidad de un sistema y la variabilidad en la experiencia de los desarrolladores, este proceso puede presentar desafíos. Por lo tanto, los desarrolladores a menudo recurren a comunidades en línea, como Stack Overflow, en busca de ayuda.

El estudio empírico revela que Stack Overflow es una fuente popular para abordar los desafíos relacionados con la refactorización. Se observa un crecimiento constante en la refactorización de código tipificado dinámicamente, especialmente en lenguajes como Python y JavaScript. Se destacan aspectos importantes de la refactorización, como la mejora de la legibilidad y la promoción de la reutilización del código. Además, se abordan no solo cuestiones relacionadas con el código, sino también elementos vinculados a bases de datos. La discusión sobre el uso de herramientas, con un enfoque en la automatización del proceso, también es un tema significativo en la comunidad.

Como trabajo futuro, los autores planean llevar a cabo encuestas dirigidas a desarrolladores junior y senior para explorar los desafíos generales en el ámbito de la refactorización, complementando así los hallazgos actuales del estudio.

“Automated refactoring of legacy Java software to enumerated types” (Khatchadourian, R., 2017)

El propósito de esta investigación radica en introducir un enfoque de preservación semántica que no solo fortalece la seguridad del tipo, sino que también genera código más comprensible, elimina complejidades innecesarias y aborda los problemas de fragilidad inherentes a la composición por separado.

El estudio presenta un algoritmo de inferencia de tipo interprocedural diseñado para rastrear el flujo de valores enumerados. Este algoritmo se ha implementado como un plugin de código abierto para Eclipse, accesible al público en general, y ha sido sometido a evaluación experimental utilizando 17 puntos de referencia de Java de gran escala.

Los resultados indican que el costo del análisis es práctico y que el algoritmo es capaz de refactorizar con éxito un número significativo de campos hacia tipos enumerados. Este trabajo representa un avance importante al proporcionar herramientas automatizadas que respaldan la migración de software heredado de Java a las tecnologías Java más modernas.

“Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering” (Amit Ratheea, Jitender Kumar Chhabrab, R., 2017)

Este artículo se centra en la introducción de una métrica innovadora para evaluar la cohesión en base a patrones de uso. Dado que el acoplamiento y la cohesión son aspectos críticos para la evaluación de calidad y modularidad en el nivel estructural de un sistema de software, este documento considera inicialmente la clase como un módulo y luego extiende esta conceptualización a un grupo de clases con el propósito de mejorar la cohesión general. La métrica propuesta utiliza patrones de uso frecuente (FUP) extraídos de diversas interacciones de funciones miembro para capturar la cohesión del módulo.

Las métricas, como LCOM1, LCOM2 y LCOM3, adoptadas son de naturaleza no estándar. El algoritmo propuesto sigue una naturaleza recursiva, donde los sistemas de software son examinados manualmente y los datos de FUP se extraen. El valor de cohesión medido se emplea para llevar a cabo el agrupamiento de módulos con el fin de aumentar la cohesión y reducir el acoplamiento entre módulos simultáneamente.

Se planea una comparación futura con otras métricas basadas en cohesión, así como la posibilidad de automatizar la métrica. La propuesta de una métrica de UPBC que mide la cohesión a nivel de módulo mediante FUP representa un enfoque novedoso para evaluar la cohesión del software. Esta nueva métrica de cohesión utiliza una metodología de agrupamiento que puede consolidar varias clases en un solo grupo.

“On the Documentation of Refactoring Types” (Amit Ratheea, Jitender Kumar Chhabrab, R., 2017)

En este artículo, se aborda la creación de un algoritmo predictivo para la identificación de metodologías de refactorización, considerándolo como un problema de clasificación de múltiples clases. Específicamente, el algoritmo clasifica los mensajes de *commit* en sistemas versionados que incorporan refactorización, focalizándose en seis tipos de refactorización a nivel de método y empleando nueve algoritmos de aprendizaje automático supervisados. Al comparar la eficacia de este enfoque con los puntos de referencia basados en palabras clave, los resultados demuestran un rendimiento superior en relación con el enfoque basado en palabras clave.

Como trabajo futuro, se plantea la posibilidad de extender este enfoque a otros proyectos desarrollados en diversos lenguajes de programación y áreas temáticas. Además, se considera la utilización de la extensión Refactoring Miner, la cual respalda la identificación de refactorizaciones a un nivel más detallado, incluyendo refactorizaciones de bajo nivel. Este enfoque promete contribuir al campo de documentación de tipos de refactorización, mejorando la capacidad de comprensión y seguimiento de los cambios realizados en proyectos de software.

A continuación, la tabla 3 es una comparativa sobre los artículos antes mencionados.

Tabla 3. tabla comparativa de trabajos relacionados

TRABAJO DE INVESTIGACION	OBJETIVO	PRODUCTO RESULTANTE	TIPO DE PROCESO	METRICAS USADAS	ALCANCE
Ramasubbu & Kemerer, 2021	Contrlar la deuda tecnica	Análisis	-	No usa	Implementado

TRABAJO DE INVESTIGACION	OBJETIVO	PRODUCTO RESULTANTE	TIPO DE PROCESO	METRICAS USADAS	ALCANCE
Samarthyam, Suryanarayana & Sharma, 2017	Refactorización de Architecture Smells	Análisis	-	No usa	Implementado
Mohan, M., Greer, D., & McMullan, P., 2016	Presentar los resultados de diferentes herramientas para la refactorización y reducción de acoplamiento	Análisis	Automático	Se enlistan 24 métricas	Implementado
Kouros, Chaikalis, Arvanitou, Chatzigeorgiou, Ampatzoglou, Amanatidis, 2019	Reducción de deuda basado en patrones de diseño	Plugin	Semi automático	acoplamiento (CBO) y cohesión (LCOM)	Implementado
Eilertsen, 2016	mejorar la corrección de los casos de refactorización Extract Local Variable y Extract And Move Method en el lenguaje de programación Java mediante la generación de comprobaciones dinámicas.	Plugin	Semi automático	acoplamiento entre clases	Implementado
Suryanarayana, Samarthyam	Hace hincapié en la	Métodos de	Manual	Acoplamiento	Implementado

TRABAJO DE INVESTIGACION	OBJETIVO	PRODUCTO RESULTANTE	TIPO DE PROCESO	METRICAS USADAS	ALCANCE
& Sharma, 2015	ocultación de datos dentro la arquitectura de software y evitar las declaraciones públicas de las variables de clase.	refactorización y prácticas para desarrolladores			
Lacerda, Petrillo, Pimenta & Guéhéneuc, 2020	resaltar la brecha entre code smells y refactoring en el estado actual	Análisis	---	---	Implementado
Dallal, 2017	probar la calidad de los siguientes indicadores: acoplamiento, cohesión	Análisis	Automático	Extract Subclass Refactoring	Implementado
0Arcelli, Cortellessa, & Pompeo, 2018	eliminación de anti-patrones que afectan el rendimiento	Análisis	Automático	Se enlistan 31 metricas de modelo, diseño y desempeño	Implementado
Avgeriou, Kruchten, Ozkaya, & Seaman, 2016	Prevención y gestión de deuda técnica	Análisis	Manual	Métricas de código, arquitectura y diseño	Implementado
和太, 2018	describe la ocultación de información como un principio y la	Análisis	---	---	Implementado

TRABAJO DE INVESTIGACION	OBJETIVO	PRODUCTO RESULTANTE	TIPO DE PROCESO	METRICAS USADAS	ALCANCE
	encapsulación como una técnica.				
Rathee & Chhabra, 2017	proponer una nueva métrica de correlación para el software orientado a objetos	métrica de correlación para el software orientado a objetos	Automático	Usage Pattern Building Association	Implementado
Eduardo Fernández, Alexander Chávez, Alessandro García, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, Willian Oizumi, 2020	Análisis de refactorización. Con métricas de 5 atributos de calidad.	Análisis	Automático	Cohesion,acoplamiento,complejidad, herencia, tamaño	Implementado
Khatchadourian, 2017	Reducir los problemas de herencia de implementación múltiple	Herramienta	Semiautomático	Modularidad	Implementado
Microsoft Docs, 2021	Encapsulamiento de atributos	plugin	Automático	Protección modular	Implementado
Joao Caldeira, Fernando Brito e Abreu, Jorge Cardoso, Jose Pereira dos Reis, 2020	Comparación de refactorización automática y manual	Análisis	Automático/ manual	----	Caso de estudio

TRABAJO DE INVESTIGACION	OBJETIVO	PRODUCTO RESULTANTE	TIPO DE PROCESO	METRICAS USADAS	ALCANCE
Anthony Peruma, Steven Simmons, Eman Abdullah, AlOmar, Christian D. Newman, Mohamed Wiem, Mkaouer, Ali Ouni , 2021	Estudio en plataformas de ayuda a desarrolladores para refactorizar	análisis	---	----	Caso de estudio
Khatchadourian, R., 2017	desarrolló un algoritmo de inferencia	plugin	Semi-Automática	---	Implementado
Amit Ratheea, Jitender Kumar Chhabrab, R., 2017	Presentación de nueva métrica	métrica	Manual	Cohesión	Implementado
Amit Ratheea, Jitender Kumar Chhabrab, R., 2017	Algoritmo de predicción de metodologías	Algoritmo	Automática	---	Implementado
Este trabajo	Método de refactorización de protección de variables	Algoritmo	Automática	PMA	Implementado

3.1 DEFINICIÓN DE CONCEPTOS

Refactorización	(Martin Fowler, 1999) "*Un cambio realizado en la estructura interna del software para que sea más fácil de entender y más económico de modificar sin cambiar su comportamiento observable*"
Fragilidad	Cuando se realizan cambios, las partes inesperadas del sistema dejarán de funcionar. La vulnerabilidad reduce en gran medida la credibilidad del personal de diseño y mantenimiento. (Santaolaya Salgado, Rene,2021)
Rigidez	Es difícil cambiar, porque cada cambio tendrá demasiado impacto en otras partes del sistema. Esto hace que sea imposible estimar el costo del cambio. (Santaolaya Salgado, Rene,2021)
Flexibilidad	Las arquitecturas de software que siguen el principio "abierto-cerrado" son flexibles, es decir, permiten cambiar el comportamiento de las unidades o componentes del programa o extenderlos a nuevos comportamientos o funciones sin editar la definición actual ni afectar a otras partes sistemáticas. (Santaolaya Salgado, Rene,2021)
Marcos de Aplicaciones Orientados a Objetos	(Barnes, Kölling ,2007) Conjunto de clases en colaboración que determinan las capacidades funcionales de dominios de aplicaciones
Encapsulamiento	(Barnes, Kölling ,2007) El encapsulamiento hace referencia a ocultar los detalles internos de implementación de un objeto a los demás, aun cuando sean de la misma clase. Esta propiedad permite asegurar que el contenido de la información de un objeto se encuentra seguro del mundo exterior. La forma de implementar el encapsulamiento en una clase se logra a través del uso de las reglas de

	visibilidad, también conocidos como modificadores de acceso. El encapsulamiento apropiado en las clases reduce el acoplamiento y, por lo tanto, lleva a un mejor diseño.
Smell Code (Código Indeseable)	(Martin Fowler, 1999) 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 indeseable consiste en ciertas estructuras en el código que violan los principios básicos de diseño y afectan negativamente la calidad de su diseño. Técnicamente hablando, el código no deseado 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.
Acoplamiento	Cantidad de información que se transmite a través de un canal de comunicación de una entidad de software a otra entidad de software. (Santaolaya Salgado, Rene,2021)
Coherencia	Es la cantidad de dependencias o canales de comunicación que tiene una entidad de software con otras que ayuden a llegar a su meta de valor. Si la entidad no requiere de otra entidad, se dice que es altamente coherente. (Santaolaya Salgado, Rene,2021)

3.2 PARADIGMA DE PROGRAMACIÓN ORIENTADA A OBJETOS

3.2.1 CLASE

En el paradigma de programación orientada a objetos, una clase es una estructura que describe un conjunto de objetos que comparten las mismas propiedades y comportamientos. Una clase funciona como una plantilla o molde que define los atributos y métodos que tendrán los objetos que se creen a partir de ella. En una clase se definen los atributos, que representan el estado de los objetos, y los métodos, que indican su comportamiento y su capacidad para

responder a los mensajes que reciben. La creación de un objeto a partir de una clase se conoce como instancia. (Sznajdleder, P. A. ,2020)

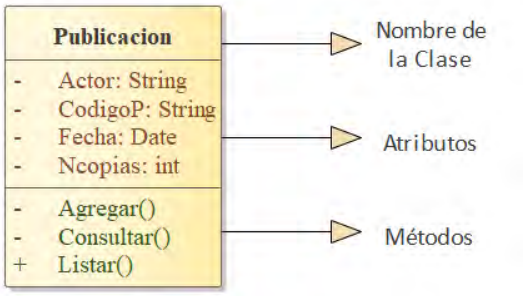


Figura 1 Modelo conceptual de una clase

3.2.2 OBJETO

Un objeto es una instancia de alguna clase, es decir, una entidad que posee un estado y un comportamiento definidos por los atributos y métodos de la clase a la que pertenece. Los objetos se crean a partir de una clase y tienen una identidad única, encapsulando su propio estado y comportamiento. El estado de un objeto se puede definir de manera implícita, a través de los valores de sus atributos, o explícita, definida por variables explicitas de estados, tales como alguna enumeración que define los estados del objeto. (Sznajdleder, P. A. ,2020)

3.2.3 REPRESENTACIÓN INTERNA DE CLASES DE OBJETOS

3.2.3.1 ATRIBUTOS DE CLASES

Los atributos son las variables de instancia que definen las características individuales de un objeto y determinan su estado y apariencia. Cada objeto particular puede tener valores distintos para estos atributos, lo que se conoce como su estado. El estado de un objeto puede ser implícito, a través de los valores de los atributos, o explícito, a través de variables que denotan un cierto estado del objeto mediante la ejecución de casos de uso. Los atributos son una parte fundamental de la definición de una clase y son esenciales para la creación de objetos y su manipulación a través de los métodos. (Sznajdleder, P. A. ,2020)

3.2.3.2 MÉTODOS DE CLASE

El comportamiento de una clase se define con la creación de métodos. Los métodos son las funciones que denotan el comportamiento o conducta de un objeto como respuesta a mensajes o peticiones de servicio desde agentes externos. El conjunto de los métodos de un objeto determina el comportamiento general del objeto (Duran Muñoz Francisco, 2017). Siguiendo el

ejemplo de la clase automóvil, un objeto de este tipo puede tener varios comportamientos como son: arrancar, acelerar, frenar, etc. (Sznajdleder, P. A. ,2020)

3.2.3.3 MENSAJES DE CLASE

Un mensaje es una petición de servicio enviada a un objeto para que éste responda de una determinada manera, realizando una de sus operaciones. Si el receptor de la solicitud acepta el mensaje, aceptará la responsabilidad de llevar a cabo la acción. En respuesta al mensaje, el receptor se comportará de una determinada forma.

Cada mensaje consta de tres partes:

1. Identidad del objeto al que va dirigida la petición de servicio (mensaje).
2. Operación solicitada (método).
3. Información adicional (argumentos), necesaria para poder ejecutar la operación solicitada.

(Sznajdleder, P. A. ,2020)

3.3 MODULARIDAD Y ENCAPSULAMIENTO

3.3.1 MODULARIDAD

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas llamadas módulos, los cuales deben ser tan independientes como sea posible de la aplicación en sí y de las partes restantes.

La modularidad cuenta con cinco principios los cuales son:

1. unidades lingüísticas de módulos: cada módulo debe corresponder a una unidad sintáctica en el lenguaje utilizado.
2. pocas interfaces (alta cohesión): cada módulo deberá comunicarse con el menor número posible de módulos.
3. pequeñas interfaces (acoplamiento débil): si dos módulos se comunican, deben de intercambiar tan poca información como sea posible.
4. interfaces explicitas: siempre que dos módulos “A” y “B” estén relacionados, la comunicación debe ser establecida en el texto.
5. Ocultamiento de datos: toda la información acerca de un módulo deberá de ser protegida para prevenir una alteración de sus estructuras internas sin autorización, las cuales no son importantes para entidades externas; a menos que se desee declararlas como públicas para permitir su manipulación desde el exterior del módulo.
   (Sznajdleder, P. A. ,2020)

3.3.2 ENCAPSULAMIENTO: OCULTANDO LOS DETALLES INTERNOS DE UN OBJETO

El encapsulamiento es un concepto clave en la programación orientada a objetos (POO), que se refiere a la ocultación de los detalles internos de un objeto a los demás, incluso si son de la misma clase. Este mecanismo protege el contenido de la información del objeto del mundo exterior. Para implementar el encapsulamiento, se utilizan las reglas de visibilidad, también conocidos como modificadores de acceso, junto con el principio de ocultamiento de información. Este principio permite empaquetar la funcionalidad de un objeto para cambiar su funcionalidad interna sin afectar la visión externa del componente en el sistema. (Sznajdleder, P. A. ,2020)

3.4 PRINCIPIO DE OCULTAMIENTO DE INFORMACIÓN: PROTEGIENDO LOS DATOS Y LAS FUNCIONES

El principio de ocultamiento de información consiste en no mostrar al exterior los datos o funciones que necesitan ser protegidos. Un módulo bien encapsulado solo exhibe la información necesaria, protegiendo los detalles internos de manipulación externa arbitraria. Si se carece de protección modular, la aplicación será frágil y partes inesperadas del sistema dejarán de funcionar correctamente ante cambios, errores o defectos en cualquier módulo.

La protección modular se establece a través de las reglas de ocultamiento de información que soportan los diferentes lenguajes de programación. En Java, existen cuatro niveles de ocultamiento de información: público, protegido, friendly y privado, que protegen a las clases de objetos de la manipulación externa de sus detalles de representación interna. Estos niveles se aplican tanto a los datos como a las funciones de las clases de objetos.

Aplicar protección modular significa evitar que un error o defecto de fragilidad en un módulo producido en tiempo de ejecución se propague hacia otros módulos relacionados que aparentemente trabajaban bien. También significa que los cambios en los requerimientos iniciales o nuevos requerimientos en un módulo solo afecten a dicho módulo, sin propagarse a otros módulos relacionados. (Sznajdleder, P. A. ,2020)

3.4.1 PROTECCIÓN MODULAR: EVITANDO LA FRAGILIDAD

La protección modular se refiere al detalle que se define en el protocolo de la clase para regular la visibilidad de la información de los objetos por entidades externas, evitando la propagación de defectos o cambios de un módulo hacia otro, lo que puede incrementar la fragilidad del sistema.

Una aplicación que carece de protección modular presenta fragilidad. Para evitarla, se utilizan las reglas de ocultamiento de información que soportan los diferentes lenguajes de programación. (Sznajdleder, P. A. ,2020)

3.4.2 PROTOCOLO DE LA CLASE PARA LA CREACIÓN DE INSTANCIAS: CONTROLANDO EL ACCESO

El control de acceso es fundamental en la programación orientada a objetos para proteger los detalles internos de representación y garantizar un encapsulamiento adecuado de las instancias u objetos. El protocolo de la clase define cómo se deben utilizar los modificadores de acceso de los atributos y funciones de la clase para lograr una protección efectiva. Las interfaces, que ofrecen servicios hacia el exterior, deben ser públicas, mientras que los elementos privados deben ser protegidos para evitar que cambios en el módulo afecten a la interfaz y a los módulos clientes. (Sznajdleder, P. A. ,2020)

3.4.3 MODIFICADORES DE ACCESO

Los modificadores de acceso definen la visibilidad de los miembros de una clase (atributos y funciones) y de la propia clase. Los siguientes son los niveles de protección de acceso en java:

- *Public* (público): de libre acceso, no ofrece protección de los detalles internos.
- *Protected* (protegido): de acceso protegido, permite el acceso solo a miembros de objetos de clases derivadas o al propio objeto.
- *Private* (privado): de acceso privado, el más restrictivo, solo permite el acceso a los propios elementos del objeto.
- *Friendly* (amistoso): nivel de acceso por defecto, similar al público, pero solo a nivel de paquete.

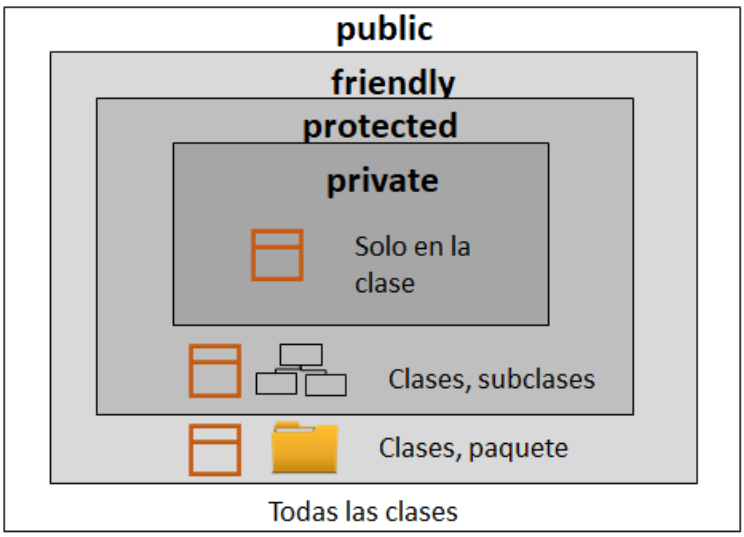


Figura 2.- Representación del acceso de los modificador de acceso (Nelida Baron, 2020)

La figura 2 muestra el acceso permitido para cada modificador de acceso. Es importante utilizar correctamente los modificadores de acceso para lograr una protección efectiva y un encapsulamiento adecuado de las instancias u objetos en la programación orientada a objetos. (Sznajdleder, P. A. ,2020)

3.5 SOFTWARE LEGADO Y MARCOS ORIENTADOS A OBJETOS

3.5.1 SOFTWARE LEGADO

El software legado, también conocido como software heredado o existente, se refiere a programas antiguos que fueron desarrollados hace varias décadas y han sido modificados continuamente 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, Rene,2021)

3.6 REFACTORIZACIÓN EN EL DESARROLLO DE SOFTWARE

En el contexto del desarrollo de software, la refactorización se refiere al proceso de reestructurar el código fuente de una aplicación sin alterar su comportamiento observable. De acuerdo con la definición de Fowler (2018), la refactorización es un cambio realizado en la estructura interna del software para hacerlo más fácil de entender y más económico de modificar.

La refactorización se lleva a cabo mediante la aplicación de una serie de técnicas de mejora de código, con el fin de mejorar su estructura interna y facilitar su mantenimiento. Estas técnicas incluyen la eliminación de código duplicado, la simplificación de métodos complejos, la separación de responsabilidades, correcto encapsulamiento y la creación de abstracciones adecuadas.

Es importante destacar que la refactorización no altera el comportamiento externo del programa, lo que significa que, si el programa se ejecuta antes y después de la refactorización con la misma entrada, la salida será la misma. En otras palabras, las refactorizaciones

preservan el comportamiento del programa, de manera que cuando se cumplen las precondiciones de la refactorización, no se producen fallos en el programa.

La refactorización es una técnica fundamental en el desarrollo de software moderno, ya que permite mejorar la calidad del código y reducir la complejidad del sistema, lo que a su vez facilita su mantenimiento y evolución. Además, la refactorización también puede mejorar el rendimiento y la eficiencia del software, ya que la eliminación de código redundante y la simplificación de métodos complejos pueden ayudar a reducir los tiempos de ejecución y el consumo de recursos del sistema.

3.7 ANTLR (PARR, 2013).

ANTLR (ANother Tool for Language Recognition) es una herramienta que permite generar analizadores para procesar y manipular archivos estructurados o binarios. Es ampliamente utilizado para crear lenguajes, herramientas y marcos de trabajo. ANTLR utiliza una gramática para generar un analizador que puede construir y recorrer árboles de análisis. Es necesario definir un archivo *lexer* que declare todos los tokens del lenguaje, así como un archivo *parser* que indique la estructura sintáctica del lenguaje mediante producciones, utilizando la notación EBNF para su definición.

En este trabajo de investigación, se utilizó la clase *JavaBaseParserListener* para obtener la información necesaria para implementar la refactorización y calcular las métricas PM. Esta clase especifica qué se debe realizar al iniciar o finalizar una regla gramatical, ya que contiene dos métodos para cada regla gramatical, una función al inicio y otra al final.

3.7.1 *STRINGTEMPLATE* (Parr, 2019)

StringTemplate es una librería de ANTLR que potencia la funcionalidad del metacompilador en la generación de código. Es un motor de plantillas Java que permite generar código fuente, páginas web, correos electrónicos o cualquier otra salida de texto con formato. *StringTemplate* es especialmente útil para generar código, máscaras de sitios múltiples y para la internacionalización/localización.

Capítulo 4 - ESTUDIO DE MÉTRICAS PMA

4.1 DISEÑO DE MÉTRICAS

Para este caso se tomaron como base las métricas realizadas por Nélida Barón en el trabajo antecedente este (Nélida Barón, 2020), adaptándolas a la medición de atributos de clases.

4.1.1 DISEÑO DE LA MÉTRICA PMAP (FACTOR DE PROTECCIÓN MODULAR POR VARIABLES *PRIVATE*)

Se define la métrica PMAP para medir el grado de protección modular con respecto a las variables de clase que han sido declaradas con el modificador de acceso *private.* Esta métrica consiste en detectar en cada una de las clases las variables que han sido declaradas como *private y* realizar una suma de todas estas variables, entre el número total de variables, posteriormente realizar una suma de todos los valores obtenidos y dividirlo entre el número total de variables de todas las clases.

La expresión matemáticas de la métrica PMAP se muestra a continuación:

$$PMAP = \frac{\sum_{ci=1}^{ci=n} \left(\frac{\sum_{vi=0}^{vi=m} AP}{TAC} \right)}{NTAC}$$

Donde:
AP: atributos con modificador de acceso *private*.
Vi: "*i-esima*" variable.
TAC: Número total de atributos de la clase.
Ci: "i-*esima*" clase.
NTAC: Número total atributos de todas las clases.
PMAP: Factor de protección modular de variables *private*.

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

4.1.2 DISEÑO DE LA MÉTRICA PMAPr (FACTOR DE PROTECCIÓN MODULAR POR VARIABLES *PROTECTED*)

Se definió la métrica PMAPr para medir el grado de protección modular con respecto a las variables que han sido declaradas con el modificador de acceso *protected*. Está métrica consiste en sumar las variables que han sido declaradas *protected* de cada una de las jerarquías entre el número total variables, posteriormente realizar la suma de todos los valores obtenidos y dividirlo entre el número total de atributos de las jerarquías

La expresión matemáticas de la métrica PMAPr se muestra a continuación:

$$PMAPr = \frac{\sum_{ji=1}^{ji=n}\left(\frac{\sum_{vi=0}^{vi=m} APr}{NTC}\right)}{NTC}$$

Donde:
APr: atributos con modificador de acceso *protecte* en una jerarquía de herencia de clases.
Vi: "*i-esima*" variables *protected* de la jerarquía.
NTA: Número total de atributos de la jerarquía.
Ji: "*i-esima*" jerarquía.
NTAJ: Número total de atributos de las jerarquías.
PMAPr: Factor de protección modular por variables *protected*.

A medida que esta razón tienda al 1 se indicará que hay una tendencia a tener variables *protected* y por lo tanto se presenta una alta protección de la información en las jerarquías.
Una medida que tienda al 0 indica que hay pocas variables *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.

4.1.3 DISEÑO DE LA MÉTRICA PMAF (FACTOR DE PROTECCIÓN MODULAR POR VARIABLES *FIENDLY*)

Se definió la métrica PMAF para medir el grado de protección modular con respecto a las variables que han sido declaradas con el modificador de acceso *friendly.* Esta métrica consiste en sumar las variables que han sido declaradas *friendly* o con modificador de acceso *default* entre el número total variables de todas las clases.
La expresión matemáticas de la métrica PMAF se muestra a continuación:

$$PMAF = \frac{\sum_{ji=1}^{ji=n}\left(\frac{\sum_{vi=0}^{vi=m} AF}{NTA}\right)}{NTAC}$$

Donde:
AF: atributos con modificador de acceso *friendly* de una clase de objetos.
Vi: "*i-esima*" variable.
TAC: Número total de atributos de la clase.
ji: "i-*esima*" clase.
NTAC: Número total de atributos de todas las clases.
PMAF: Factor de protección modular por variables *fiendly*.

A medida que esta razón tienda al 1 se indicará que hay una tendencia a tener variables *friendly* y por lo tanto se presenta una mayor protección de la información en cuanto variables *public y* una menos protección de la información en cuanto variables *private y protected*.

Una medida que tienda al 0 indica que hay pocas variables *friendly* o *default,* por lo que la protección de variables *friendly* se encuentra con un nivel bajo. El mejor valor será el 1, el peor valor será 0.

4.1.4 DISEÑO DE LA MÉTRICA PMA (PROTECCIÓN MODULAR DE ATRIBUTOS)

Se ha definido una métrica para medir el grado de protección modular de un software legado. Esta métrica consiste en, la suma de las variables que han sido declaradas con el modificador de acceso diferente a *public*, entre el número total de atributos de todas las clases.

La expresión matemática de la métrica PMA se muestra a continuación:

$$PM = \frac{\sum_{ji=1}^{ji=n}\left(\frac{\sum_{vi=0}^{vi=m} APN}{TAC}\right)}{NTAC}$$

Donde:
APN: atributos con modificador de acceso no *public*.
Vi: "*i-esima*" variable.

TAC: Número total de atributos de la clase.
ji: “i-*esima*” clase.
NTAC: Número total de atributos de todas las clases.
PMA: Factor de protección modular.

A medida que esta razón tienda a 0, mayor es el problema, indicaría que hay una tendencia a tener muchas variables públicas y por lo tanto tienen poca o nula protección modular y un encapsulamiento incorrecto. Una medida que tienda al 1 indica que hay pocas variables públicas, 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.

4.1.5 DISEÑO DE LA MÉTRICA TPMA (TOTAL DE PROTECCIÓN MODULAR DE ATRIBUTOS)

Se ha definido una métrica para medir el grado total de protección modular que tiene una arquitectura de clases. Esta métrica consiste en la suma ponderada de PMFP, PMFPr y PMFF entre tres.
La expresión matemática de la métrica TPMA se muestra a continuación:

$$TPMA = \frac{\big((PMAP * 1) + (PMAPr * 0.75) + (PMAF * 0.25)\big)}{NTAC}$$

PMAP: Grado de protección modular de atributos *private.*
PMAPr: Grado de protección modular de atributos *protected.*
PMAF: Es el grado de protección modular de atributos *friendly o default.*
TPMA: Es el nivel total de protección modular.
NTAC: Número total de atributos de todas las 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 mayor es el problema, indicaría que hay una tendencia a tener muchas variables con un bajo grado de protección y por lo tanto un encapsulamiento incorrecto. Una medida que tienda al 1 indica que hay pocas variables con un bajo grado de protección, por lo tanto, tienden a tener un mejor nivel de encapsulamiento.

Capítulo 5 – DISEÑO DEL METODO DE REFACTORIZACION

La Figura 3 muestra el modelo BPMN (*Business Process Model and Notation*) del proceso general del método de refactorización de modificadores de acceso. Este método de refactorización está conformado por los subprocesos de a) análisis de código de fuente, b) cálculo de métricas (de código original y de código refactorizado), c) refactorización y d) generación de código.

5.1 DIAGRAMA BPMN

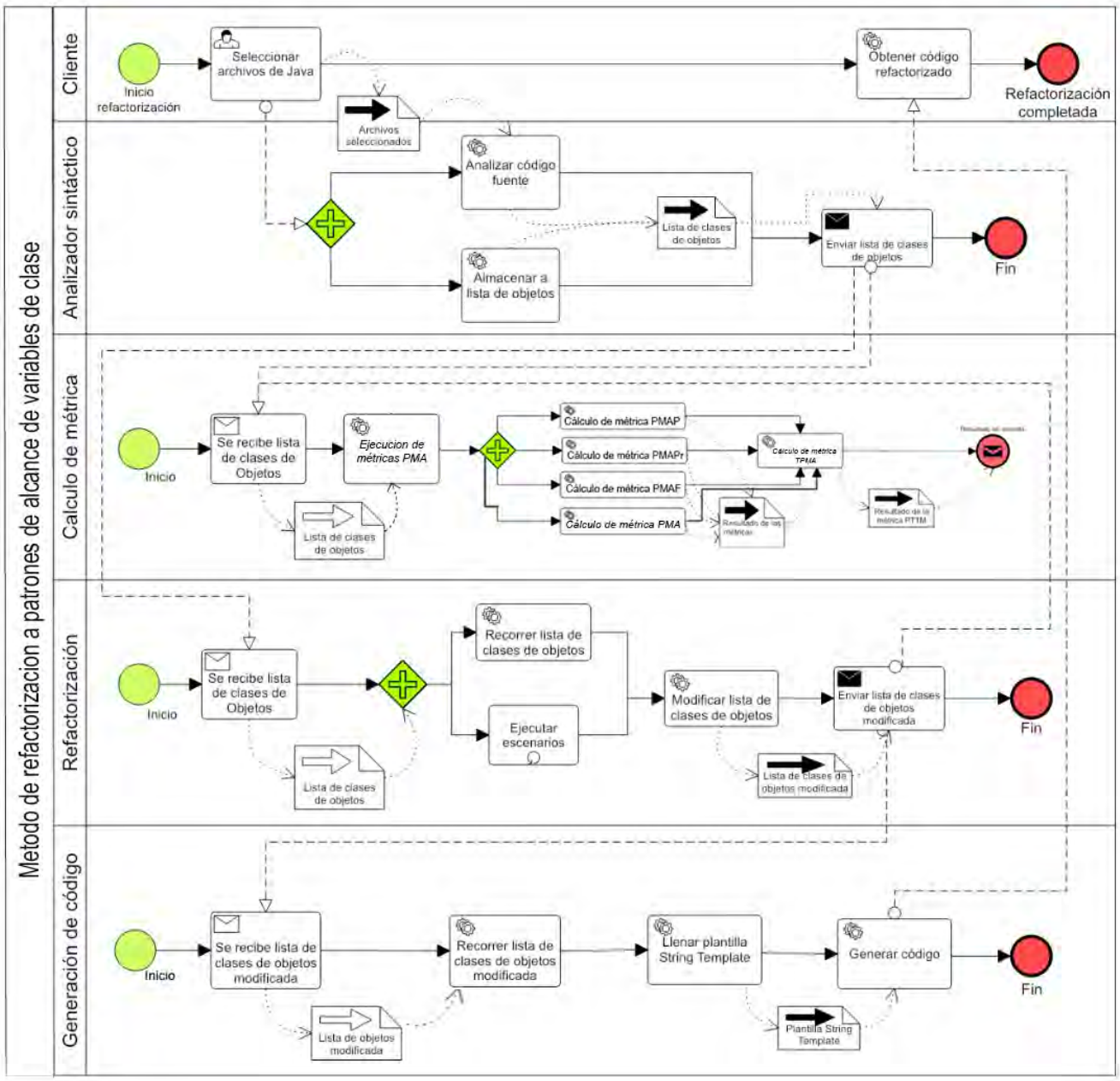


Figura 3. Diagrama BPMN del proceso de refactorización actualizado

En la figura 3 se observa el manejo de los subprocesos siguientes:

El subproceso de análisis de código fuente recibe como entrada los archivos del código fuente original de la aplicación que se requiere refactorizar. Este proceso realiza un análisis sintáctico utilizando la herramienta ANTLR con la gramática del lenguaje Java en su versión 8.0, para sustraer la información necesaria para los subprocesos de cálculo de métricas, refactorización de modificadores de acceso y generación de código refactorizado. El resultado de salida de este subproceso de obtiene una lista compleja con la información obtenida.

El subproceso de cálculo de métricas calcula el grado de protección modular en los 4 niveles de visibilidad (*private*, *protected*, *friendly* y *public*) y el nivel protección modular total del código original de la aplicación a refactorizar y el nivel protección modular total del código refactorizado.

En el subproceso Generar código nuevo: Este subproceso recibe como entrada el código refactorizado contenido en una lista compleja resultado del subproceso de refactorización. Con esta información el sistema utiliza la plantilla *StringTemplate* para generar el código refactorizado equivalente a la aplicación original. El código refactorizado contiene los cambios pertinentes del modificador de acceso en todas las funciones identificadas, sin alterar el comportamiento de la aplicación original.

5.2 DIAGRAMA DE CASOS DE USO DEL METODO DE REFACTORIZACION

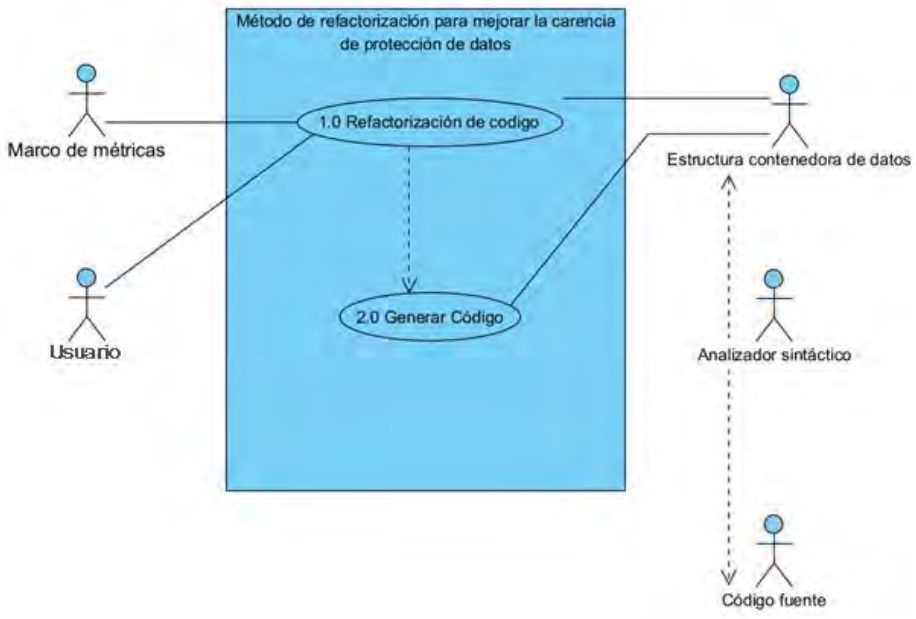
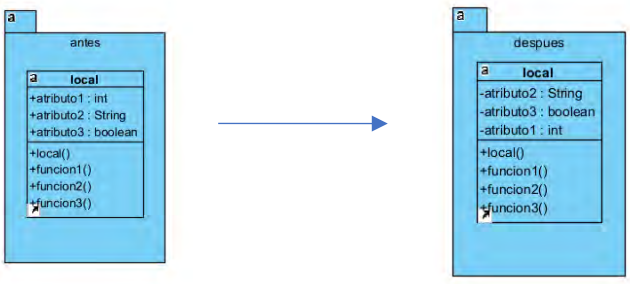
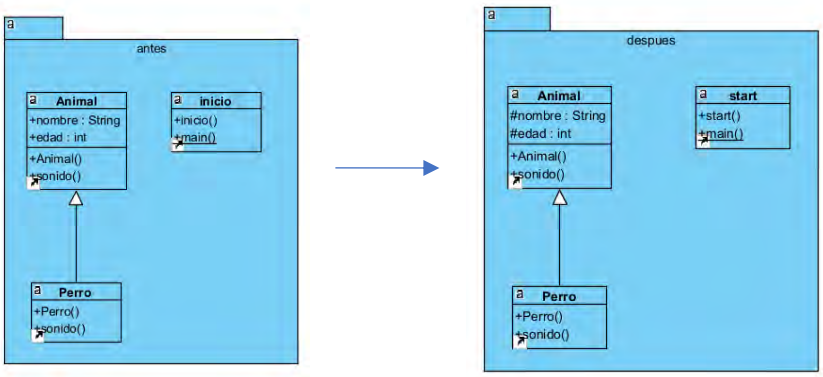


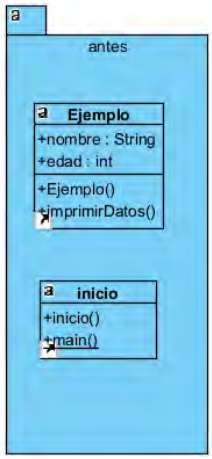
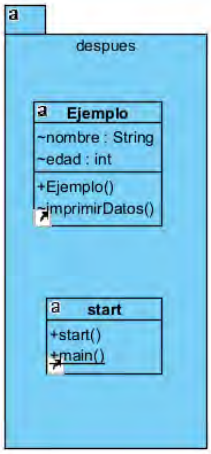
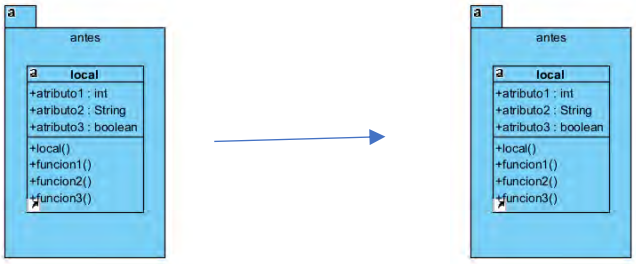
Figura 4. Caso de uso para refactorización de modificadores de acceso

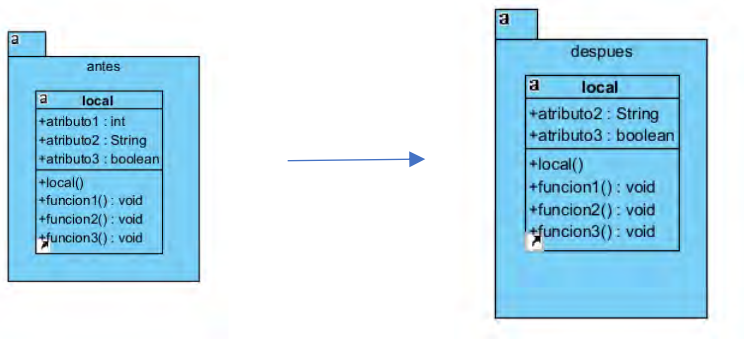
5.3 ANALISIS DE ESCENARIOS DEL METODO DE REFACTORIZACION

Nombre:	Refactorizar modificadores de acceso
Descripción:	El sistema aplica el método de refactorización de código, con el fin de corregir en la declaración de atributos, los modificadores de acceso incorrectos que fueron asignados sin tomar en cuenta las reglas de ocultamiento de información.
Actores:	Usuario físico y Estructura de datos (lista ligada compleja)

Precondiciones:	1.- Contar con la lista de objetos de tipo “Clase” con toda la información necesaria, derivada de la generación de código.
Escenario principal de éxito:	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación del método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso, aplicando los criterios de identificación de modificadores de acceso, se determina el modificador correcto de cada uno de los atributos encontrados. 3.- Genera código con la plantilla *String Template*. 4.- Termina el caso de uso “generar código”. 5.- Aplicación terminal del cálculo de métricas al código fuente refactorizado. 6.- El sistema a través del cálculo de métricas, detecta que no fue necesario realizar cambios. Nota: En algunos casos, es posible que los modificadores de acceso estén bien utilizados y no requieran ningún cambio. En estos casos, la comparativa antes y después de la refactorización puede mostrar que no se han producido cambios sustanciales en el código. Por lo que el resultado de las métricas no tendrá cambios significativos, previo y después a la refactorización.
Escenario alternativo (1)	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación el método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso. 3.- Detección de atributos con modificador de acceso *public* o *friendly* que son llamados únicamente por funciones pertenecientes a la misma clase que los contiene por lo que su modificador de acceso se modifica a *private*.  4.- Genera código nuevo con la plantilla *String Template*. 5.- Termina el caso de uso “generar código”.

	6.- Aplicación terminal del cálculo de métricas al código fuente refactorizado.
Escenario alternativo (2)	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación del método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso. 3.- Detección de atributos con modificador de acceso *public* o *friendly* que son llamados únicamente por otra entidad de software en una clase derivada (clase hija que hereda de una clase padre) dentro o fuera del mismo paquete y que está siendo llamado por un método de esta clase, por lo que su modificador de acceso debe ser modificado de *public* a *protected*.  4.- Genera código nuevo con la plantilla *String Template*. 5.- Termina el caso de uso. 7.- Aplicación terminal del cálculo de métricas al código fuente refactorizado.
Escenario alternativo (3)	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación del método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso. 3.- Detección de atributos con modificador de acceso *public* que son llamados únicamente por otra entidad de software o clase que se ubique dentro del mismo paquete (no solamente las clases hijas), por lo que su modificador de acceso debe ser modificado de public a *friendly*.

	  4.- Genera código nuevo con la plantilla *String Template*. 5.- Termina el caso de uso. 6.- Aplicación terminal del cálculo de métricas al código fuente refactorizado.
Escenario alternativo (4)	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación del método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso. 3.- Detección de atributos con modificador de acceso *public* que son llamados por otra entidad de software o clase que se ubique fuera del paquete (exceptuando las clases hijas), por lo que su modificador de acceso debe ser conservado como *public*.  4.- Genera código nuevo con la plantilla *String Template*. 5.- Termina el caso de uso. 6.- Aplicación inicial del cálculo de métricas al código fuente original.

Escenario alternativo (5)	1.- Aplicación inicial del cálculo de métricas al código fuente original. 2.- Aplicación del método de refactorización de código, con el fin de corregir en la declaración de atributos el modificador de acceso. 3.- Detección de atributos, donde se comprueba si más de una función de la misma clase hace uso de una variable declarada *privada*, en caso que solo un método lo llame, el dato se mueve al interior de ese único método que lo usa.  4.- Genera código nuevo con la plantilla *String Template*. 5.- Termina el caso de uso. 6.- Aplicación inicial del cálculo de métricas al código fuente original.
Excepciones:	Atributos declarados dentro de los parámetros de un método
Postcondiciones:	Como resultado de obtendrá una lista compleja de objetos de tipo “Clase” con el código refactorizado.
Escenario de fracaso 1:	1.- El usuario inicializa el proceso de refactorización 2.- No selecciona ningún archivo 3.- Termina caso de uso

Nombre:	Generación de código
Descripción:	El sistema hace uso de la estructura contenedora de datos para llenar la plantilla *String Template* para la generación de código nuevo en un directorio especifico.
Actores:	Lista de objetos de clase
Precondiciones:	1.- Contar con lista de objetos de nombre “Clase” con toda la información necesaria derivados del análisis sintáctico.

	2.- Haber ejecutado el método de refactorización sobre la lista de objetos de nombre “Clase”
Escenario principal de éxito:	1.- El sistema hace pasar cada elemento de la lista contenedora de objetos por la plantilla *String Template* 2.- Durante el llenado de la plantilla los elementos de la lista encajan con los elementos de la plantilla para la generación de archivos de texto nuevos. 3.- Para evitar la sobreescritura de alguna clase con el mismo nombre, el sistema crea carpetas con el nombre del paquete en el directorio de salida para guardar las clases. 4.- El código nuevo es probado para evaluar la funcionalidad.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá una carpeta con los archivos de tipo Java correspondientes al código refactorizado.
Escenario de fracaso 1:	1.- El sistema hace pasar cada elemento de la lista contenedora de objetos por la plantilla *String Template.* 2.- Durante el llenado de la plantilla, los elementos de la lista no encajan con los elementos de la plantilla para la generación de archivos de texto nuevos. 3.- Para evitar la sobreescritura de alguna clase con el mismo nombre, el sistema crea carpetas con el nombre del paquete en el directorio de salida para guardar las clases. 4.- El código nuevo es probado para evaluar la funcionalidad. 5.- El código se verá comprometido en caso de usar palabras reservadas o librerías fuera del estándar de JAVA

5.4 DIAGRAMA DE CLASES DEL METODO DE REFACTORIZACION

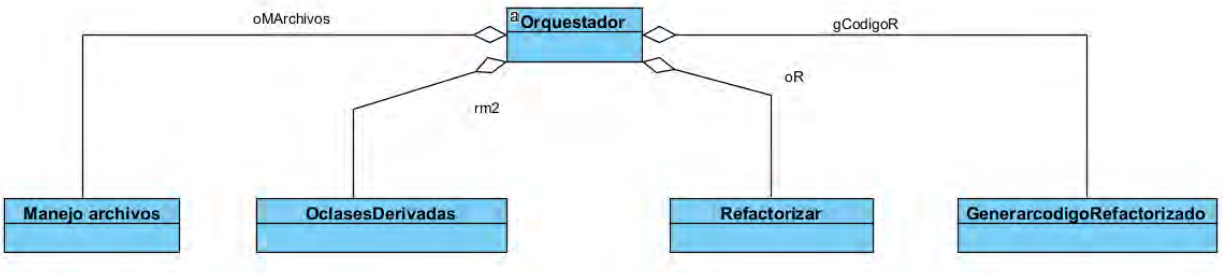


Figura 5. Diagrama de clases del método de refactorización actualizado

La figura 5 muestra la arquitectura de clases del método de refactorización de modificadores de acceso, dicho método será agregado como un paquete a la herramienta SR2-Refactoring, respetando el principio de abierto-cerrado y extendiendo su funcionalidad sin modificar el código existente.

5.5 DIAGRAMA DE SECUENCIA DEL METODO DE REFACTORIZACION

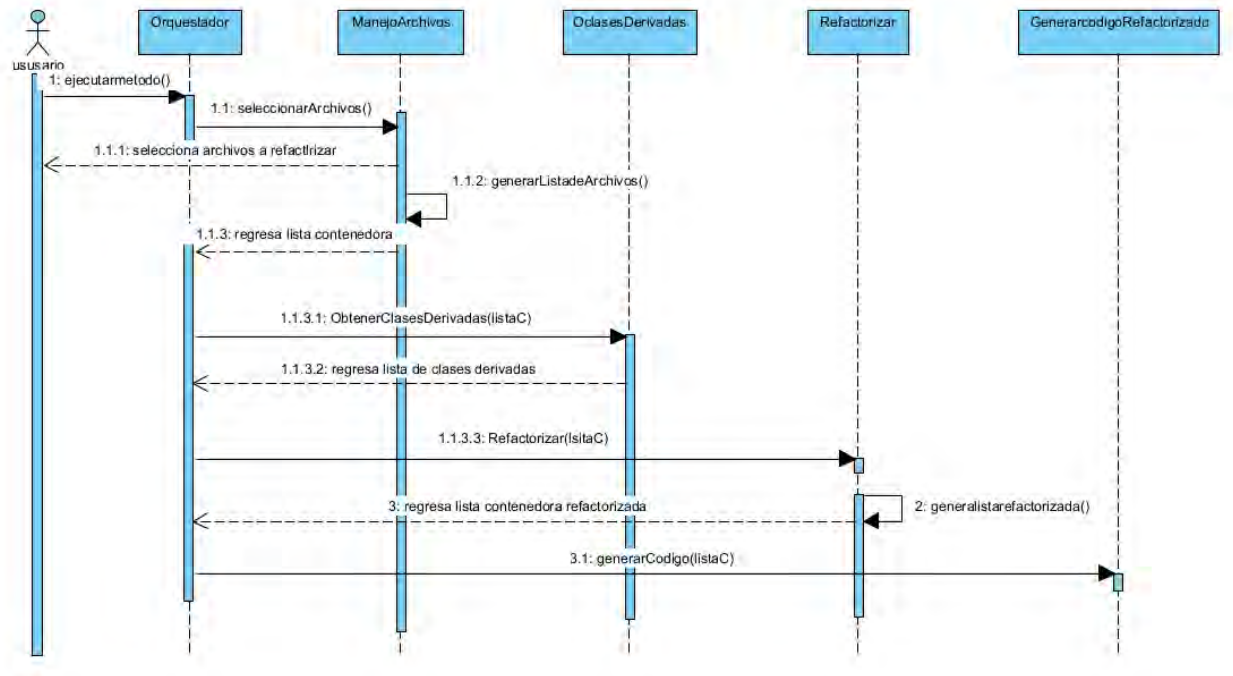


Figura 6. Diagrama de secuencia del proceso de refactorización actualizado

Descripción del diagrama de secuencia del método de refactorización Figura 6:
1.- El cliente ejecuta el método de refactorización
2.- El cliente carga o selecciona los archivos Java a ser refactorizados
3.- Si el cliente elige una carpeta, el sistema busca todos los archivos Java
4.- Si el sistema no encuentra ningún archivo Java el sistema envía un mensaje de error
5.- Si el sistema encuentra uno o más archivos con extensión Java, el sistema genera la lista que contiene a los archivos seleccionados
6.- El sistema obtiene la información necesaria de cada una de las clases objeto para el proceso de refactorización.
7.- El sistema obtiene las relaciones de herencia de cada objeto de tipo “Clase”.
8.- El sistema obtiene la lista de objetos tipo “Clase” con toda la información recabada.
9.- Se aplica el método de refactorización de modificadores de acceso.

10.- Aplicando los criterios de identificación de modificadores de acceso, se determina el modificador correcto de cada una de las funciones encontradas.
11.- Se genera una lista compleja que almacena cada objeto de tipo “Clase” con el código refactorizado.
12.- Tomando la información de cada uno de los objetos de tipo “Clase”, plasmándola en la plantilla *StringTemplate* ST se genera el código refactorizado equivalente al código original.
13.- El sistema almacena en una carpeta denominada “código nuevo” los archivos con el código refactorizado.

5.6 DIAGRAMA DE CASOS DE USO DE METRICAS

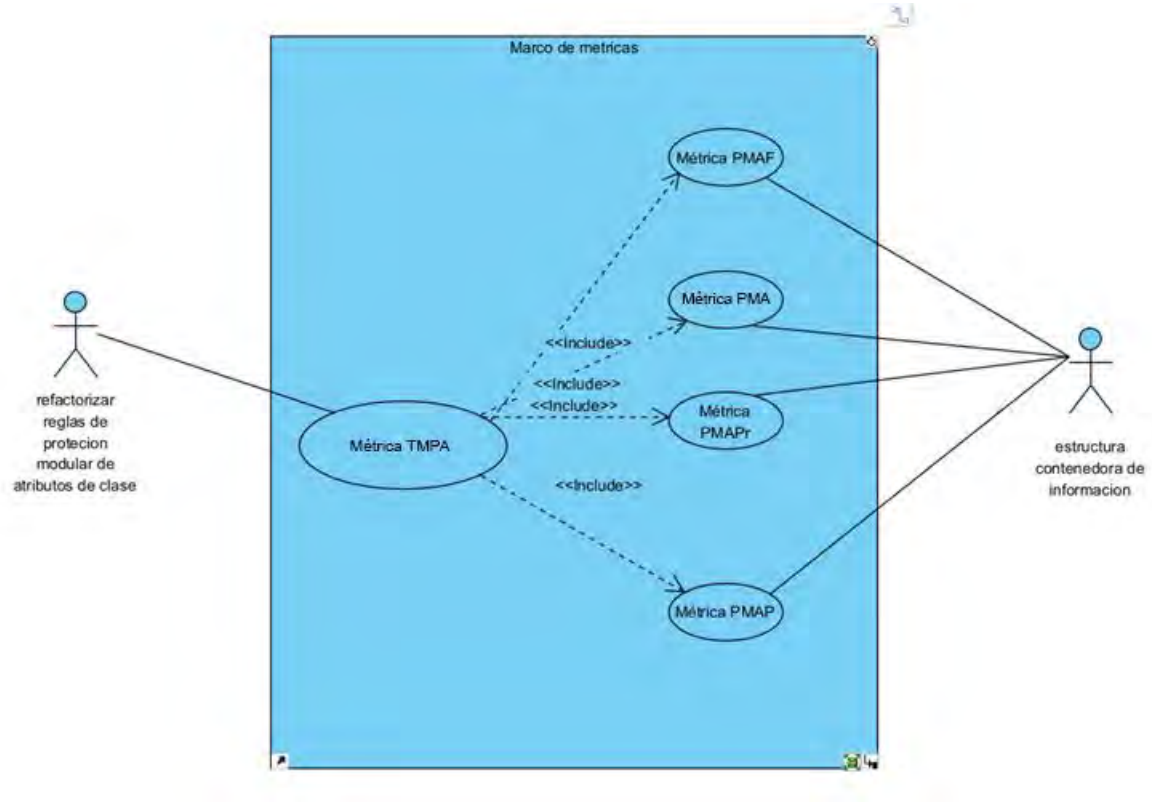


Figura 7. Diagrama de caso de uso para e marco de métricas

La figura 7 muestra el caso de uso del marco de métricas contemplado como un subproceso y no como parte del proceso principal. Según lo establecido en el diagrama BPMN.

5.7 ANALISIS DE ESCENARIOS DEL METRICAS

Nombre:	Cálculo de métrica PMAF

Descripción:	Esta métrica consiste en sumar las variables que han sido declaradas *friendly* o con modificador de acceso *default* y dividir entre el número total variables.
Actores:	Estructura contenedora de datos.
Precondiciones:	Contar con lista de objetos de tipo Clase con toda la información necesaria derivados del análisis sintáctico.
Escenario principal de éxito:	1.- Se seleccionan los archivos para el método de refactorización. 2.- Se realiza el análisis sintáctico y creación de estructura contenedora. 3.- Se recorre e identifican los atributos *friendly* de todas las clases y se acumula la suma. 4.- Se acumula el número total de atributos de todas las clases. 5.- Se divide el número total de atributos *friendly* entre el número total de atributos de todas las clases. 6.- Se retorna el valor obtenido.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá un dato de tipo double que contendrá el valor del cálculo.
Escenario de fracaso 1:	1.- Se selecciona la carpeta de archivos. 2.- El analizador no encuentra ningún archivo.

Nombre:	Cálculo de métrica PMAP
Descripción:	Esta métrica consiste en sumar las variables que han sido declaradas *private* y dividir entre el número total variables.
Actores:	Estructura contenedora de datos.
Precondiciones:	Contar con lista de objetos de tipo “Clase” con toda la información necesaria derivados del análisis sintáctico.
Escenario principal de éxito:	1.- Se seleccionan los archivos para el método de refactorización. 2.- Se realiza el análisis sintáctico y creación de estructura contenedora. 3.- Se identifican los atributos *private* de todas las clases y se acumula la suma. 4.- Se acumula el número total de atributos de todas las clases.

	5.- Se divide el número total de atributos *private* entre el número total de atributos de todas las clases. 6.- Se retorna el valor obtenido.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá una un dato de tipo double que contendrá el valor del cálculo.
Escenario de fracaso 1:	1.- Se selecciona la carpeta de archivos. 2.- El analizador no encuentra ningún archivo.

Nombre:	Cálculo de métrica PMAPr
Descripción:	Esta métrica consiste en sumar las variables que han sido declaradas *protected* y dividir entre el número total variables.
Actores:	Estructura contenedora de datos.
Precondiciones:	Contar con lista de objetos de tipo “Clase” con toda la información necesaria derivados del análisis sintáctico.
Escenario principal de éxito:	1.- Se seleccionan los archivos para el método de refactorización. 2.- Se realiza el análisis sintáctico y creación de estructura contenedora. 3.- Se identifican los atributos *protected* de todas las clases y se acumula la suma. 4.- Se acumula el número total de atributos de todas las clases. 5.- Se divide el número total de atributos *protected* entre el número total de atributos de todas las clases. 6.- Se retorna el valor obtenido
Excepciones:	
Postcondiciones:	Como resultado de obtendrá una un dato de tipo double que contendrá el valor del cálculo.
Escenario de fracaso 1:	1.- Se selecciona la carpeta de archivos. 2.- El analizador no encuentra ningún archivo.

Nombre:	Cálculo de métrica PMA
Descripción:	Esta métrica consiste en sumar las variables que no han sido declaradas como *public* y dividir entre el número total variables.
Actores:	Estructura contenedora de datos.
Precondiciones:	Contar con lista de objetos de tipo Clase con toda la información necesaria derivados del análisis sintáctico.
Escenario principal de éxito:	1.- Se seleccionan los archivos para el método de refactorización. 2.- Se realiza el análisis sintáctico y creación de estructura contenedora. 3.- se identifican los atributos *private*, *friendly*, *protected* de todas las clases se acumula la suma. 4.- Se acumula el número total de atributos de la clase. 5.- se divide el número total de atributos no *public* entre el número total de atributos de la clase. 6.- se retorna el valor obtenido.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá el valor de tipo double calculado.
Escenario de fracaso 1:	1.- Se selecciona la carpeta de archivos. 2.- El analizador no encuentra ningún archivo.

Nombre:	Cálculo de métrica TPMA
Descripción:	Esta métrica tiene el propósito de medir el grado total de protección modular que tiene una arquitectura de clases
Actores:	Estructura contenedora de datos
Precondiciones:	Se debe tener el resultado previo del cálculo de métricas PMAP, PMAF, PMAPr y PMA.
Escenario principal de éxito:	1.- Se toma el resultado de la métrica PMAP y se multiplica por 1 2.- Se toma el resultado de la métrica PMAPr y se multiplica por 0.75 3.- Se toma el resultado de la métrica PMAF y se multiplica por 0.25

	4.- Se realiza la sumatoria de los tres productos y se divide entre el número total de atributos
Excepciones:	
Postcondiciones:	Como resultado de obtendrá el valor de tipo double calculado.
Escenario de fracaso 1:	1.- Se selecciona la carpeta de archivos 2.- El analizador no encuentra ningún archivo

5.8 DIAGRAMA DE CLASES DE MÉTRICAS

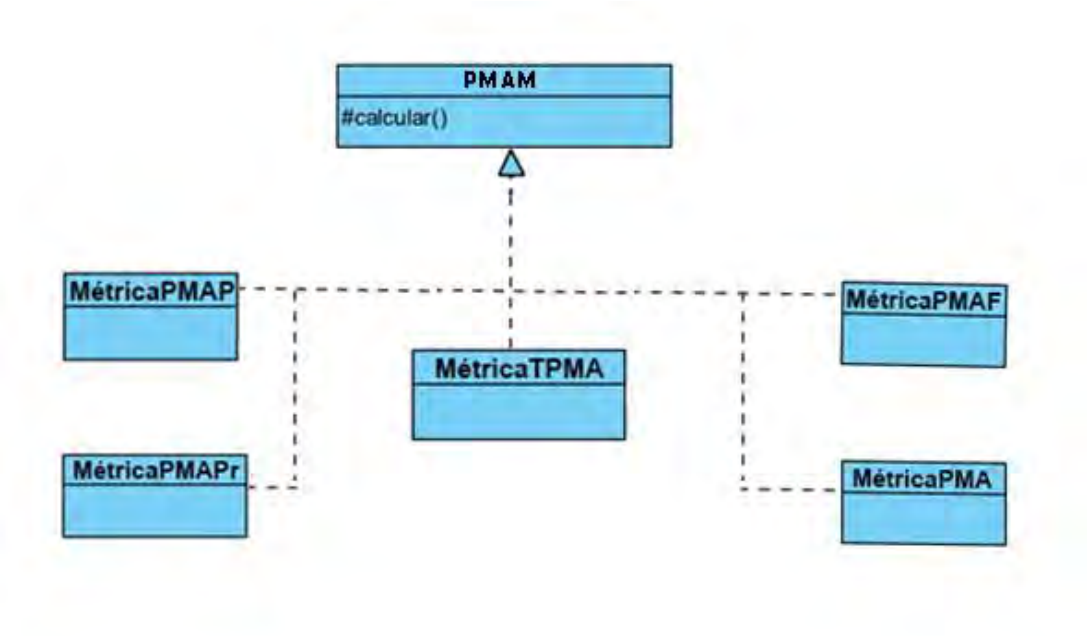


Figura 8. Diagrama de clases para el marco de métricas

5.9 DIAGRAMA DE SECUENCIA DEL MARCO DE METRICAS

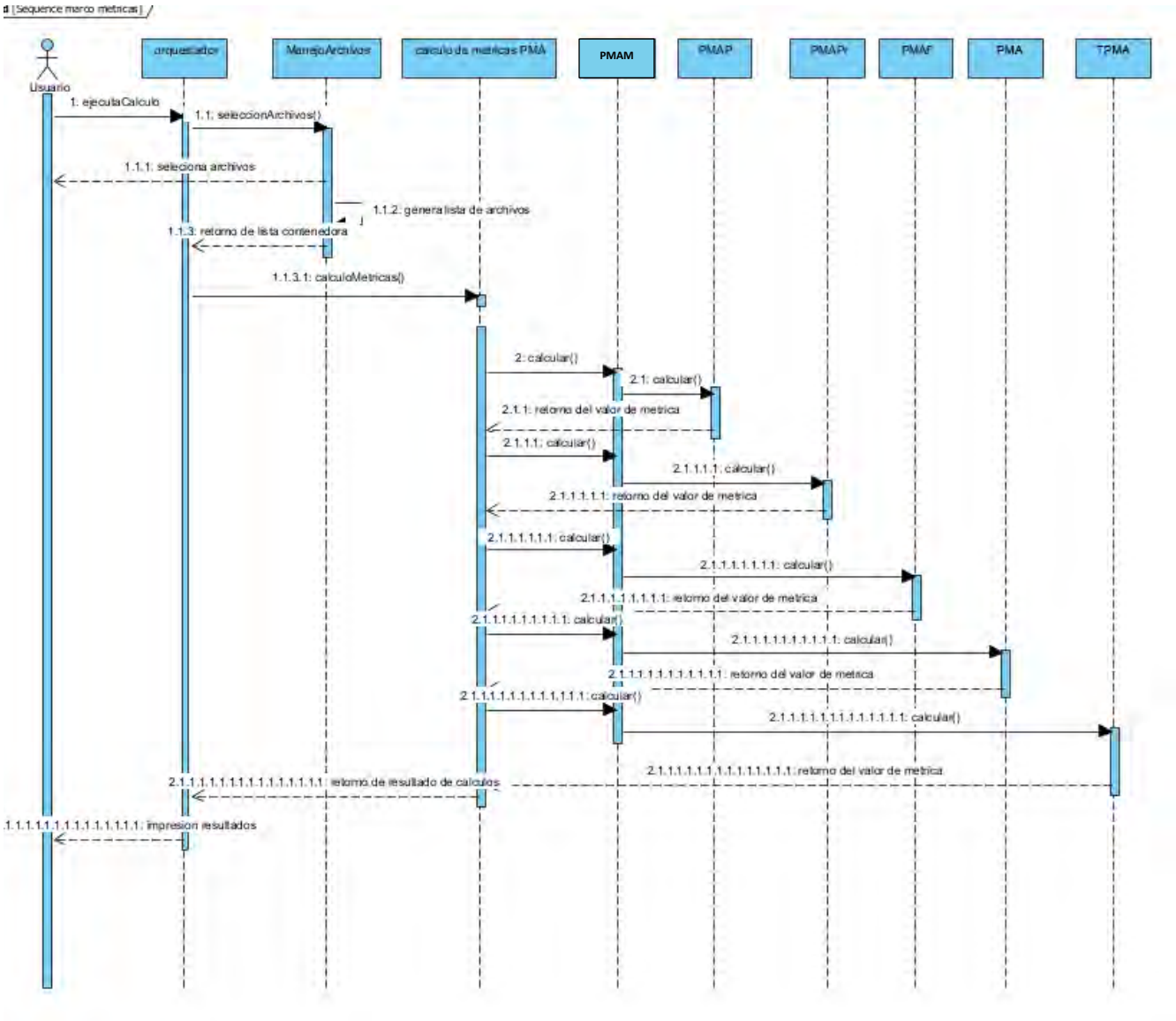


Figura 9. Diagrama de secuencia para el marco de métricas

Descripción del diagrama de secuencia del método de refactorización de la Figura 9:
1.- El cliente ejecuta el método de refactorización.
2.- El cliente carga o selecciona los archivos Java a ser refactorizados.
3.- Si el cliente elige una carpeta, el sistema busca todos los archivos Java en esa carpeta.
4.- Si el sistema no encuentra ningún archivo Java el sistema envía un mensaje de error.
5.- Si el sistema encuentra uno o más archivos con extensión Java, el sistema genera una lista que contiene a los archivos seleccionados.
6.- El sistema obtiene la información necesaria de cada una de las clases de objetos para el proceso de cálculo de métricas.
7.- El sistema obtiene las relaciones de herencia de cada entidad de tipo “Clase”.
8.- El sistema obtiene una lista de entidades tipo “Clase” con toda la información recabada.

9.- se ejecuta la métrica PMAP y se obtiene un resultado.
10.- se ejecuta la métrica PMAPr y se obtiene un resultado.
11.- se ejecuta la métrica PMAF y se obtiene un resultado.
12 se ejecuta la métrica PMA y se obtiene un resultado.
13.- se ejecuta la métrica TPMA y se obtiene un resultado.
14.- los resultados de la métricas son capturados y expuestos al cliente.

Capitulo 6 - PRUEBAS

6.1 DISEÑO DE PRUEBAS

6.1.1 IDENTIFICADOR DEL DOCUMENTO

ISMRCAC0301.

La convención de la nomenclatura de los identificadores es la siguiente:

- IS = Ingeniería de Software.
- MRCAC = Método de Refactorización de Modificadores de Atributos de Clase.

Tipo de artículo:

- 01 = Módulos de programa.
- 02 = Programas de control.
- 03 = Plan de pruebas.
- 04 = Diseño de pruebas.
- 05 = Casos de pruebas.

Identificador:

- XX = Identificador Numérico

6.1.2 ALCANCE

Este plan de pruebas abarca una evaluación completa del método de refactorización de reglas de protección de atributos de clase, así como la evaluación de las métricas para la protección modular de las mismas.

6.1.3 REFERENCIAS.

Los siguientes documentos fueron usados como fuente de información para este plan de pruebas:

- Documento de análisis y diseño.

6.1.4 PUNTOS DE PRUEBA

Los siguientes módulos que serán probados se muestran a continuación:

Tabla 4. Módulos a probar en el sistema

Sistema	**Función**	**Identificador**
Proceso de análisis del código fuente.	Ejecución de un analizador léxico, sintáctico y semántico para la obtención de estructuras complejas de datos almacenadas en una lista.	ISMRCAC0100.
Método de refactorización de reglas de protección de atributos de clase.	Subsistema de evaluación y refactorización de reglas de protección atributos	ISMCAC0200.
Marco de métricas de calidad de arquitecturas orientadas a objetos.	Cálculo de métricas de protección modular.	ISMRCAC0300.

6.1.5 PROCEDIMIENTOS DE CONTROL DE TAREAS

Los procedimientos de control para las tareas del método de refactorización y para las tareas de la evaluación de las métricas.

Tabla 5. Funcionalidades a probar e identificador

Sistema	**Función**	**Identificador**
Programa de aplicación.	Localización del paquete al que pertenece la clase.	ISMRCAC0101.
Programa de aplicación.	Localización de las importaciones de la clase.	ISMRCAC0102.
Programa de aplicación.	Localización del nombre de la clase.	ISMRCAC0103.
Programa de aplicación.	Localización de la clase padre de la clase. (si es que existe).	ISMRCAC0104.
Programa de aplicación.	Localización de los atributos de la clase.	ISMRCAC0105.
Programa de aplicación.	Método de validación del modificador de acceso.	ISMRCAC0106.

Programa de aplicación.	Identificación de acceso a variables en el rango *public*	ISMRCAC0201.
Programa de aplicación.	Identificación de acceso a variables en el rango *protected*	ISMRCAC0202.
Programa de aplicación.	Identificación de acceso a variables en el rango *friendly*	ISMRCAC0203.
Programa de aplicación.	Identificación de acceso a variables en el rango *private*	ISMRCAC0204.
Programa de aplicación.	Método de refactorización de modificadores de acceso.	ISMRCAC0205.
Programa de aplicación.	Mover variable a método local	ISMRCAC0206.
Programa de aplicación.	Generación de código refactorizado	ISMRCAC0207.
Programa de aplicación.	Llenado de la plantilla *String Template* y generación de código refactorizado.	ISMRCAC0208.
Programa de aplicación.	Compilación del código refactorizado.	ISMRCAC0209.
Programa de aplicación.	Verificación del funcionamiento del código refactorizado.	ISMRCAC0210.
Programa de aplicación.	Verificación del aumento de protección modular.	ISMRCAC0211.
Programa de aplicación.	Cálculo de la métrica *"PMAP"*.	ISMRCAC0301.
Programa de aplicación.	Cálculo de la métrica *"PMAPr"*.	ISMRCAC0302.
Programa de aplicación.	Cálculo de la métrica *"PMAF"*.	ISMRCAC0303.
Programa de aplicación.	Cálculo de la métrica *"PMA"*.	ISMRCAC0304.
Programa de aplicación.	Cálculo de la métrica *"TPMA"*.	ISMRCAC0305
Programa de aplicación.	Verificación del aumento de protección modular.	ISMRCAC0306

6.1.6.- CARACTERÍSTICAS PARA SER PROBADAS

Las características que deben ser probadas son:

Tabla 6. Características a probar en las pruebas

Diseño de prueba	**Identificador**
Proceso de análisis del código fuente.	ISMRCAC0400
Cálculo de métricas de protección modular	ISMRCAC0401
Método de refactorización de reglas de atributos de clase	ISMRCAC0402
Funcionalidad del código refactorizado equivalente al código original	ISMRCAC0403

6.1.7.- CARACTERÍSTICAS PARA NO PROBAR

A continuación, las características que no serán probadas:

1. Los casos a probar no incluirán todas las combinaciones sintácticas del lenguaje Java.
2. El método de refactorización no señala si el código legado está libre de errores.
3. La interfaz no es probada.

6.1.8.- EFOQUE

El maestrante del CENIDET Miguel Romero Meza realizó las pruebas, de esta manera se puede verificar que las pruebas son de acorde al desarrollo del sistema SR2.

Pruebas Del Proceso De Análisis Del Código Fuente

La comprobación del proceso de análisis de código fuente, se realizará mediante la obtención de la información de cada una de las clases que se reciban como entrada, como son: paquete, librerías, nombre, funciones y atributos de cada de una de ellas, comprobándose además la generación de la lista de clases objeto requeridas por el método de refactorización.

Pruebas De Refactorización

La validación del método de refactorización de arquitecturas de marcos orientados a objetos con funciones globalmente visibles, que consiste en cambiar si fuera necesario el modificador de cada una de los atributos por el modificador correcto. La validación se realizará mediante la ejecución del código original contra la ejecución del código refactorizado. Comprobando, que, bajo las mismas entradas, ambos sistemas deberán comportarse de la misma manera y ofrecer los mismos resultados.

Pruebas De Calidad

Las pruebas de calidad son dadas por la aplicación de las métricas de protección modular (PMAP, PMAPr, PMAF y PMA, TPMA) del código refactorizado y en la comparativa de la medición del código legado.

Pruebas De Funcionalidad

La validación final para comprobar que todo el proceso funciona de manera correcta, consiste en probar exhaustivamente que el código refactorizado funciona exactamente igual que el código original de entrada.

6.1.9.- CRITERIOS APROBADO/DESAPROBADO

Aprobación/Desaprobación Pruebas Del Proceso De Análisis Del Código Fuente

Para los casos de prueba del proceso de análisis del código fuente en Java, el criterio aprobación/desaprobación se realizará mediante la comparación del análisis y la obtención de información manual contra el análisis y la obtención de información de manera automática. En ambos casos se deberá obtener la misma información, comprobándose además la generación correcta de la lista de clases objeto requeridas por el proceso de refactorización.

Para los casos de prueba del método de refactorización, el criterio aprobación/desaprobación se realizará mediante la comparativa de los resultados obtenidos de forma manual contra los resultados obtenidos de forma automática en cada caso de prueba.

Para los casos de prueba de calidad, el criterio aprobación/desaprobación será mediante la comparación del resultado obtenido por calculo manual contra el resultado obtenido de manera automática, ambos deberán ser los mismos.

Para las pruebas de funcionalidad, el criterio de aprobación/desaprobación consiste en hacer pruebas comparativas exhaustivas de ambos códigos; pruebas unitarias para cada función y pruebas de aceptación para capacidad del sistema original, que satisface cada requerimiento. Comparativamente, los resultados deben ser exactamente los mismos.

6.1.10.- CRITERIOS DE SUSPENSIÓN Y REANUDACIÓN

Las pruebas no se suspenderán definitivamente, cada vez que se presente una prueba desaprobada se procederá a evaluar y corregir el error.

6.1.11.- LIBERACIÓN DE PRUEBAS

La liberación y aceptación de las pruebas será dada mediante la entrada y salida de datos de las pruebas.

6.1.12.- DISEÑO DE PRUEBAS

Diseño De Prueba ISMRCAC0400

1. Diseño de prueba:
   ISMRTM0400 - Proceso de análisis del código fuente.
2. Características a ser aprobadas:
   - Se evaluará la correcta obtención de información de la representación de un proyecto Java en una estructura de datos mediante pruebas unitarias y de integración.
   - Se evaluará la lista contenedora de estructuras complejas de datos.
3. Refinamiento del enfoque:
   El objetivo es obtener una estructura compleja de datos en la cual se represente una clase del lenguaje de programación Java.
   Se debe comprobar que una lista logre almacenar de una a varias estructuras complejas de datos.
4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del análisis del código fuente será dada por la comparativa de las partes de las estructuras de datos obtenidas automáticamente con las esperadas manualmente mediante pruebas unitarias. Para que esta comparación sea aprobatoria deben ser resultados iguales los obtenidos de manera automática con los resultados esperados manualmente.
   Finalmente se realizará la prueba de integración para mostrar en consola las estructuras de datos generadas automáticamente.

Diseño De Prueba ISMRCAC0401

1. Diseño de prueba:
   ISMRCAC0401 - Cálculo de métricas de protección modular
2. Características a ser aprobadas:
   - Se evaluará la correcta aplicación de las métricas para calcular la protección modular atributos de clase (PMAP, PMAPr, PMAF y PMA, TPMA).
3. Refinamiento del enfoque:
   El objetivo es evaluar la correcta aplicación de las métricas para calcular la protección modular en el software legado. El código legado deberá estar libre de fallas y/o defectos por lo que será compilado y ejecutado en un compilador para el lenguaje de programación Java, posteriormente el marco orientado a objetos para el cálculo de estas métricas realizará un reconocimiento léxico y sintáctico del código legado para generar una estructura de datos con la información necesaria para el cálculo de las métricas.

4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del cálculo de las métricas será dada por la comparativa de los resultados obtenidos manualmente con los resultados obtenidos

automáticamente, para que esta comparación sea aprobatoria deben ser resultados iguales en cada una de las métricas PMAP, PMAPr, PMAF y PMA, TPMA.

Diseño De Prueba ISMRCAC0402

1. Diseño de prueba:
   ISMRCAC0402 - Método de refactorización de reglas protección de atributos de clase
2. Características a ser aprobadas:
   - Se evaluará el funcionamiento correcto del método de refactorización de reglas de protección de atributos de clase
   - La correcta identificación de los accesos a atributos de clase en sus diferentes rangos
   - La correcta verificación o cambio de modificadores de acceso
   - Movilización de variables a métodos locales
3. Refinamiento del enfoque:
   El objetivo es evaluar el cambio correcto del modificador de acceso de los atributos de clase que implementa un código legado y que requieran de la refactorización.
   El código legado con “oloroso” no deberá presentar fallas, por lo que será compilado y ejecutado en un compilador para el lenguaje de programación Java, así como verificar el comportamiento del software legado; posteriormente el subsistema de refactorización de código tomará el archivo que implementa el código legado para realizar un reconocimiento léxico, sintáctico y semántico con el objetivo de generar estructuras de datos con la información necesaria para la refactorización.
4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del método de refactorización será dada por la comparativa de los resultados obtenidos de la aplicación de las métricas automáticamente entre la arquitectura refactorizada y los resultados obtenidos de la arquitectura antes de la refactorización. Para que esta comparación sea aprobatoria debe existir mejora en el resultado de la arquitectura refactorizada contra los resultados obtenidos del cálculo de las métricas del código legado, así como también debe ser igual el comportamiento externo del software legado.

Diseño De Prueba ISMRCAC0403

1. Diseño de prueba:
   ISMRCAC0403 - Funcionalidad del código refactorizado equivalente al código original
2. Características a ser evaluadas: Se evaluará la equivalencia funcional entre el código refactorizado y el código original. Esto incluye la correcta ejecución de las funcionalidades, la preservación de la lógica de negocio y el cumplimiento de los requisitos funcionales establecidos para el software.

3. Refinamiento del enfoque: El objetivo principal es garantizar que el código refactorizado conserve la misma funcionalidad que el código original. Para ello, se realizará una revisión exhaustiva del código refactorizado, asegurando que todas las funciones y características operen de manera idéntica a como lo hacen en el código original. Además, se verificará que cualquier mejora o cambio introducido durante el proceso de refactorización no afecte negativamente la funcionalidad del software.
4. Aprobación/desaprobación de la evaluación de las características: La aprobación de esta prueba se basará en la comparación directa entre el comportamiento del código refactorizado y el código original. Se verificará que todas las funcionalidades se ejecuten de manera correcta y que no haya discrepancias significativas en los resultados obtenidos entre ambas versiones del software.

6.1.13 - ESPECIFICACIÓN DE CASOS DE PRUEBA

Diseño de Prueba ISMRCAC0501

Esta prueba tiene el objetivo de comprobar el correcto listado de los elemento de los archivos de entrada en listas correspondientes.

Características a probar:

Característica a Ejecutar	Diseño de Prueba No. de especificación
Localización del paquete al que pertenece la clase.	ISMRCAC0400.
Localización de las importaciones de la clase.	ISMRCAC0400.
Localización del nombre de la clase.	ISMRCAC0400.
Localización de la clase padre de la clase. (si es que existe).	ISMRCAC0400.
Localización de los atributos de la clase.	ISMRCAC0400.

Como entrada se recibe el código legado compilado y probado, el código se analiza para obtener la estructura de datos compleja, así como la lista contenedora de dichas estructuras.

Como salida se espera mostrar en consola la lista contenedora de estructuras complejas de información pobladas con dichas estructuras representativas de clases en Java.

Diseño de Prueba ISMRCAC0502

Esta prueba tiene el propósito de evaluar la protección modular de los atributos de clase.

Características a probar:

Característica a Ejecutar	Diseño de Prueba No. de especificación
Cálculo de la métrica *"PMAP"*.	ISMRCAC0400.
Cálculo de la métrica *"PMAPr"*.	ISMRCAC0400.
Cálculo de la métrica *"PMAF"*.	ISMRCAC0400.
Cálculo de la métrica *"PMA"*.	ISMRCAC0400.
Cálculo de la métrica *"TPMA"*.	ISMRCAC0400

Como entrada se recibe la lista contenedora de estructuras de clases.

Como salida se espera la salida de un archivo con lo resultados de todos los archivos evaluados por cada una de las métricas.

Diseño de Prueba ISMRCAC503

Esta prueba tiene el propósito de aplicar las reglas establecidas para el cambio de modificadores de acceso de los atributos.

Características a probar:

Característica a Ejecutar	Diseño de Prueba No. de especificación
Método de validación del modificador de acceso.	ISMRCAC0400.
Identificación de acceso a variables en el rango público	ISMRCAC0400.
Identificación de acceso a variables en el rango protected	ISMRCAC0400.
Identificación de acceso a variables en el rango friendly	ISMRCAC0400.
Identificación de acceso a variables en el rango private	ISMRCAC0400.
Método de refactorización de modificadores de acceso.	ISMRCAC0400.
Mover variable a método local	ISMRCAC0400.
Generación de código refactorizado	ISMRCAC0400.
Llenado de la plantilla *"String Template"* y generación de código refactorizado.	ISMRCAC0400.
Compilación del código refactorizado.	ISMRCAC0400.
Verificación del funcionamiento del código de salida.	ISMRCAC0400.

Verificación del aumento de protección modular.	ISMRCAC0222.

Como entrada se recibe la lista contenedora de estructuras de clases.

Como salida se espera la correcta identificación de llamadas de variables de estructuras complejas de información pobladas las estructuras representativas de clases en Java. Mostrando en consola parte de este proceso de identificación y refactorización.

6.2 EJECUCION DE PRUEBAS

6.2.1 - Caso de Prueba ISMRCAC0501

Nombre del Caso de prueba: PSP3.java

En la siguientes figura se muestra sistema para realizar pruebas, un sistema de 8 clases desarrollado del proyecto PSP3. El sistema está escrito en lenguaje Java y ha sido compilado y revisado su funcionamiento.

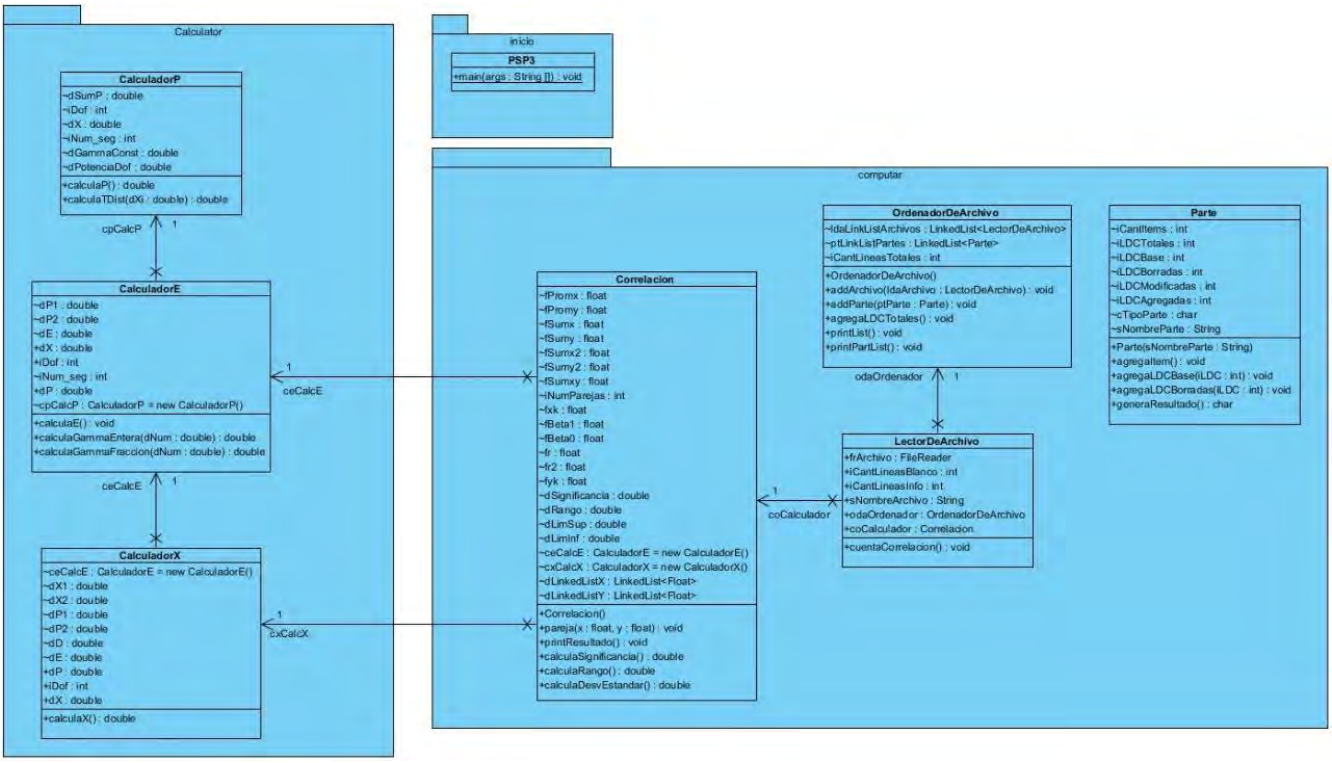
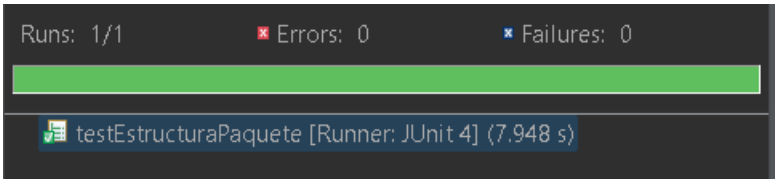


Figura 10. Diagrama de clases de proyecto para pruebas

Tabla 7. Pruebas hechas a las primeras funcionalidades

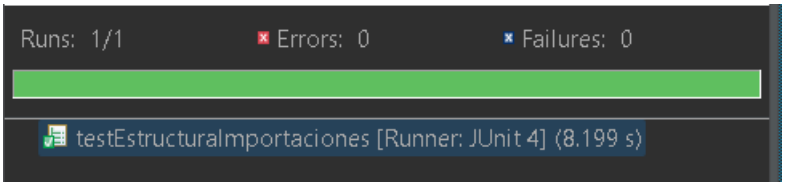
Características a probar	**Pruebas unitarias**
Localización del paquete al que pertenece la clase.	

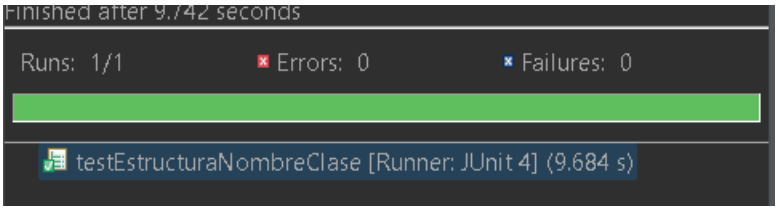
```java
public void testEstructuraPaquete() {
    int n;
    objetos();
    for (n=0;n<listaC.size();n++) {

        switch(n) {

            case 0:{
                assertEquals(listaC.get(n).getPaquete(),"Calculator");
                break;
            }
            case 1:{
                assertEquals(listaC.get(n).getPaquete(),"Calculator");
                break;
            }
            case 2:{
                assertEquals(listaC.get(n).getPaquete(),"Calculator");
                break;
            }
            case 3:{
                assertEquals(listaC.get(n).getPaquete(),"computar");
                break;
            }
            case 4:{
                assertEquals(listaC.get(n).getPaquete(),"computar");
                break;
            }
            case 5:{
                assertEquals(listaC.get(n).getPaquete(),"computar");
                break;
            }
            case 6:{
                assertEquals(listaC.get(n).getPaquete(),"computar");
                break;
            }
            case 7:{
                assertEquals(listaC.get(n).getPaquete(),"inicio");
                break;
            }
        }
    }
}
```



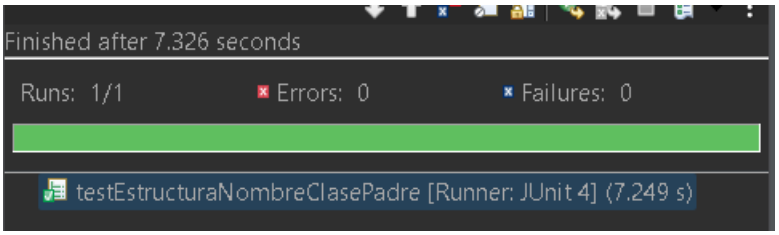
Localización de las importaciones de la clase.	

```java
public void testEstructuraImportaciones() {
        int n;
        ArrayList<String> imports=new ArrayList();
        objetos();
        for (n=0;n<listaC.size();n++) {
            imports.clear();
            switch(n) {

                case 0:{
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 1:{
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 2:{
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 3:{
                    imports.add("import java.text.DecimalFormat ;");
                    imports.add("import java.util.LinkedList ;");
                    imports.add("import Calculator.CalculadorE ;");
                    imports.add("import Calculator.CalculadorX ;");
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 4:{
                    imports.add("import java.io . * ;");
                    imports.add("import java.util.regex . * ;");
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 5:{
                    imports.add("import java.util . * ;");
                    imports.add("import java.io . * ;");
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 6:{
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
                case 7:{
                    imports.add("import java.io . * ;");
                    imports.add("import computar.Correlacion ;");
                    imports.add("import computar.LectorDeArchivo ;");
                    imports.add("import computar.OrdenadorDeArchivo ;");
                    assertEquals(listaC.get(n).getImportaciones(),imports);
                    break;
                }
            }
        }
    }
```



Localización del nombre de la clase.	 

```java
public void testEstructuraNombreClase() {
        int n;
        objetos();
        for (n=0;n<listaC.size();n++) {

            switch(n) {

                case 0:{
                    assertEquals(listaC.get(n).getNombre(),"CalculadorE");
                    break;
                }
                case 1:{
                    assertEquals(listaC.get(n).getNombre(),"CalculadorP");
                    break;
                }
                case 2:{
                    assertEquals(listaC.get(n).getNombre(),"CalculadorX");
                    break;
                }
                case 3:{
                    assertEquals(listaC.get(n).getNombre(),"Correlacion");
                    break;
                }
                case 4:{
                    assertEquals(listaC.get(n).getNombre(),"LectorDeArchivo");
                    break;
                }
                case 5:{
                    assertEquals(listaC.get(n).getNombre(),"OrdenadorDeArchivo");
                    break;
                }
                case 6:{
                    assertEquals(listaC.get(n).getNombre(),"Parte");
                    break;
                }
                case 7:{
                    assertEquals(listaC.get(n).getNombre(),"PSP3");
                    break;
                }
            }
        }
    }
```

Localización de la clase padre de la clase. (si es que existe).	 

```java
public void testEstructuraNombreClasePadre() throws IOException {
        int n;
        objetos();
        for (n=0;n<listaC.size();n++) {

            switch(n) {

                case 0:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 1:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 2:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 3:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 4:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 5:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 6:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
                case 7:{
                    assertEquals(listaC.get(n).getClasePadre(),"");
                    break;
                }
            }
        }
    }
```

Localización de los atributos de la clase.	

```java
public void testEstructuraAtributosDeClase() throws IOException {
    int n;
    ArrayList<String> variables=new ArrayList();
    objetos();
    for (n=0;n<listaC.size();n++) {
        variables.clear();
        switch(n) {

            case 0:{
                variables.add(" double dP1");
                variables.add(" double dP2");
                variables.add(" double dE");
                variables.add(" public double dX");
                variables.add(" public int iDof");
                variables.add(" int iNum_seg");
                variables.add(" public double dP");
                variables.add(" CalculadorP cpCalcP = new CalculadorP ( )");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 1:{
                variables.add(" double dSumP");
                variables.add(" int iDof");
                variables.add(" double dX");
                variables.add(" int iNum_seg");
                variables.add(" double dGammaConst");
                variables.add(" double dPotenciaDof");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 2:{
                variables.add(" CalculadorE ceCalcE = new CalculadorE ( )");
                variables.add(" double dX1");
                variables.add(" double dX2");
                variables.add(" double dP1");
                variables.add(" double dP2");
                variables.add(" double dD");
                variables.add(" double dE");
                variables.add(" public double dP");
                variables.add(" public int iDof");
                variables.add(" public double dX");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 3:{
                variables.add(" float fPromx");
                variables.add(" float fPromy");
                variables.add(" float fSumx");
                variables.add(" float fSumy");
                variables.add(" float fSumx2");
                variables.add(" float fSumy2");
                variables.add(" float fSumxy");
                variables.add(" int iNumParejas");
                variables.add(" float fxk");
                variables.add(" float fBeta1");
                variables.add(" float fBeta0");
                variables.add(" float fr");
                variables.add(" float fr2");
                variables.add(" float fyk");
                variables.add(" double dSignificancia");
                variables.add(" double dRango");
                variables.add(" double dLimSup");
                variables.add(" double dLimInf");
                variables.add(" CalculadorE ceCalcE = new CalculadorE ( )");
                variables.add(" CalculadorX cxCalcX = new CalculadorX ( )");
                variables.add(" LinkedList < Float > dLinkedListX");
                variables.add(" LinkedList < Float > dLinkedListY");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 4:{
                variables.add(" public FileReader frArchivo");
                variables.add(" public int iCantLineasBlanco");
                variables.add(" public int iCantLineasInfo");
                variables.add(" public String sNombreArchivo");
                variables.add(" public OrdenadorDeArchivo odaOrdenador");
                variables.add(" public Correlacion coCalculador");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 5:{
                variables.add(" LinkedList < LectorDeArchivo > ldaLinkListArchivos");
                variables.add(" LinkedList < Parte > ptLinkListPartes");
                variables.add(" int iCantLineasTotales");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 6:{
                variables.add(" int iCantItems");
                variables.add(" int iLDCTotales");
                variables.add(" int iLDCBase");
                variables.add(" int iLDCBorradas");
                variables.add(" int iLDCModificadas");
                variables.add(" int iLDCAgregadas");
                variables.add(" char cTipoParte");
                variables.add(" String sNombreParte");
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
            case 7:{
                assertEquals(listaC.get(n).getVariables(),variables);
                break;
            }
        }
    }
```

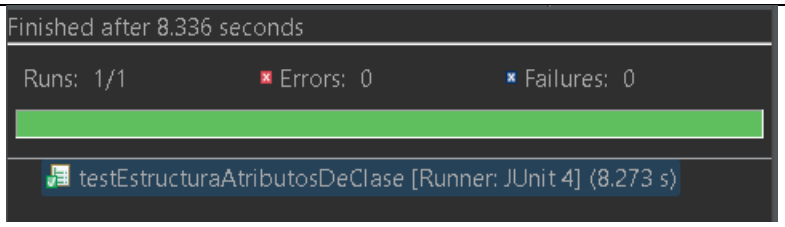


Tabla 1. Pruebas unitarias a los elementos de la estructura contenedora

6.2.2.- CASOS DE PRUEBA

Para las siguientes pruebas se toman en cuenta 4 proyectos de diferente magnitud para contrastar y reafirmar resultados comparando los resultados del sistema de métricas con un conteo y calculo manual de los datos.

6.2.2.1 Nombre del Caso de prueba: PSP3.java

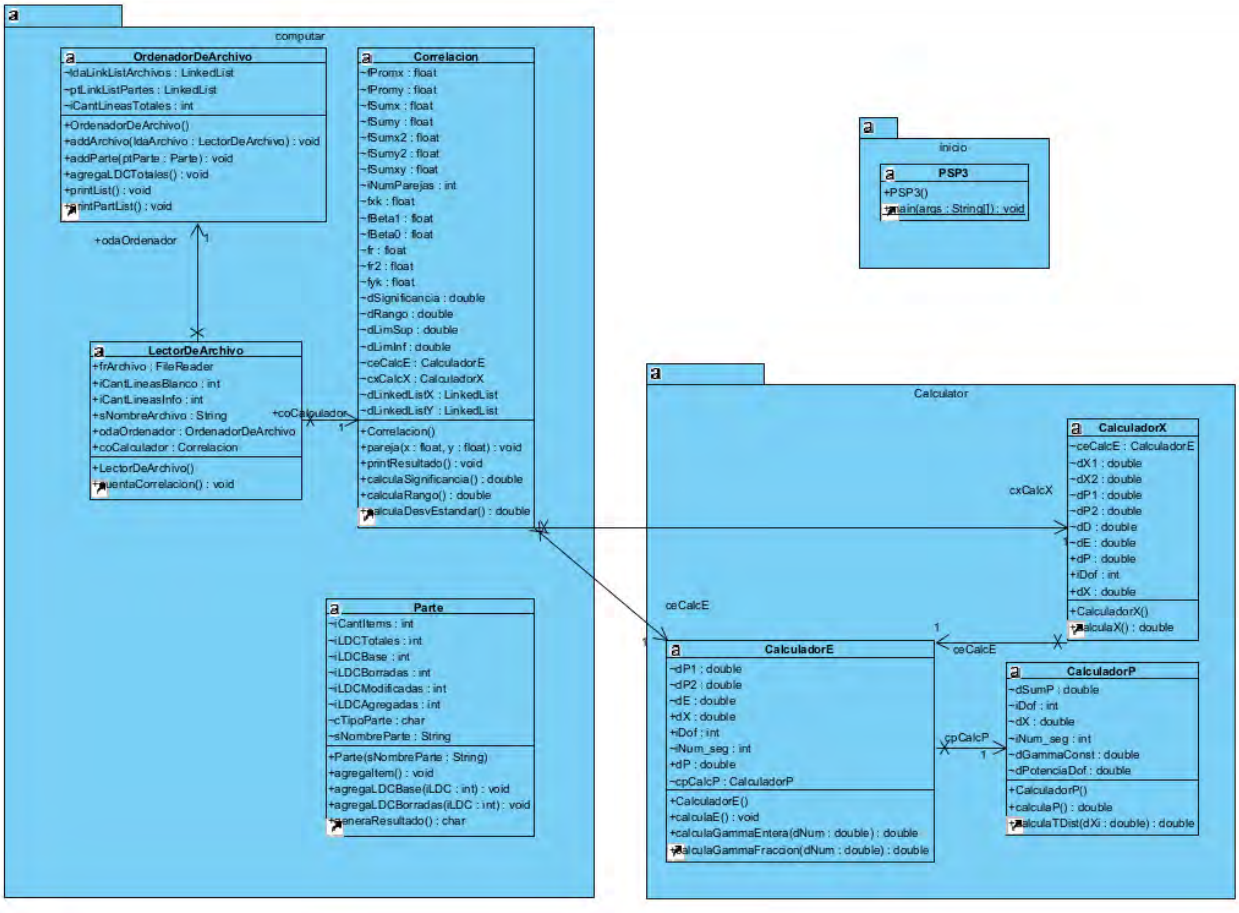


Figura 11. Diagrama de clases de proyecto PSP3 para pruebas de métricas

Tabla 8. Tabla de conteo de atributos del proyecto PSP3 previo a la refactorización

Atributos	Valor
Total, de clases	8
Total, de atributos	63
Atributos privados	0
Atributos protegidos	0
Atributos friendly	51
Atributos públicos	12

Tabla 9. Tabla de comparación de resultados manuales y automáticos del proyecto PSP3 previo a la refactorización

<table>
<tr><th colspan="3">Pruebas unitarias</th></tr>
<tr><th>Características a probar</th><th>Resultado manual</th><th>Resultado de sistema</th></tr>
<tr><td>Cálculo de la métrica “PMAP”.</td><td rowspan="5"><table><tr><td>PMAP</td><td>0</td></tr><tr><td>PMAPr</td><td>0</td></tr><tr><td>PMAF</td><td>0.80952381</td></tr><tr><td>PMA</td><td>0.80952381</td></tr><tr><td>TMPA</td><td>0.202380952</td></tr></table></td><td rowspan="5">Resultado de metrica PMAP : 0.0000
Resultado de metrica PMAPr : 0.0000
Resultado de metrica PMAF : 0.8095
Resultado de metrica PMA : 0.8095
Resultado de metrica TPMA : 0.2024</td></tr>
<tr><td>Cálculo de la métrica “PMAPr”.</td></tr>
<tr><td>Cálculo de la métrica “PMAF”.</td></tr>
<tr><td>Cálculo de la métrica “PMA”.</td></tr>
<tr><td>Cálculo de la métrica “TPMA”.</td></tr>
</table>

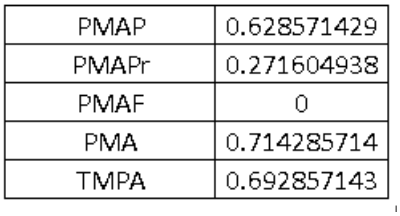
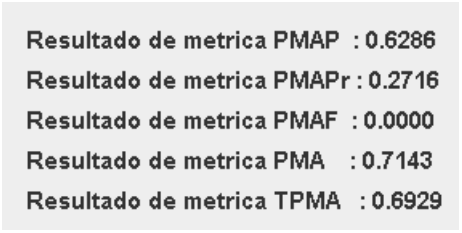
6.2.2.2 Nombre del Caso de prueba: SUDOKU.java

Sudoku es un sistema legado cuya descripción se puede ver en la tabla 10 y los resultados de la métricas son observables en la tabla 11.

Tabla 10. Tabla de conteo de atributos del proyecto SUDOKU previo a la refactorización

Atributos	Valor
Total, de clases	28
Total, de atributos	70
Atributos privados	44
Atributos protegidos	6
Atributos friendly	0
Atributos públicos	20

Tabla 11. Tabla de comparación de resultados manuales y automáticos del proyecto SUDOKU previo a la refactorización

Pruebas unitarias		
Características a probar	**Resultado manual**	**Resultado de sistema**
Cálculo de la métrica *"PMAP"*.		
Cálculo de la métrica *"PMAPr"*.		
Cálculo de la métrica *"PMAF"*.		
Cálculo de la métrica *"PMA"*.		
Cálculo de la métrica *"TPMA"*.		

6.2.2.3 2 Nombre del Caso de prueba: SUSHI.java

Sushi es un sistema legado cuya descripción se puede ver en la tabla 12 y los resultados de la métricas son observables en la tabla 13.

Tabla 12. Tabla de conteo de atributos del proyecto SUSHI previo a la refactorización

Atributos	Valor
Total, de clases	67

Total, de atributos	75
Atributos privados	7
Atributos protegidos	1
Atributos friendly	30
Atributos públicos	37

Tabla 13. Tabla de comparación de resultados manuales y automáticos del proyecto SUSHI previo a la refactorización

<table>
<tr><th colspan="3">Pruebas unitarias</th></tr>
<tr><th>Características a probar</th><th>Resultado manual</th><th>Resultado de sistema</th></tr>
<tr><td>Cálculo de la métrica “PMAP”.</td><td rowspan="5"><table><tr><td>PMAP</td><td>0.093333333</td></tr><tr><td>PMAPr</td><td>0.008333333</td></tr><tr><td>PMAF</td><td>0.4</td></tr><tr><td>PMA</td><td>0.506666667</td></tr><tr><td>TMPA</td><td>0.203333333</td></tr></table></td><td rowspan="5">Resultado de metrica PMAP : 0.0933
Resultado de metrica PMAPr : 0.0083
Resultado de metrica PMAF : 0.4000
Resultado de metrica PMA : 0.5067
Resultado de metrica TPMA : 0.2033</td></tr>
<tr><td>Cálculo de la métrica “PMAPr”.</td></tr>
<tr><td>Cálculo de la métrica “PMAF”.</td></tr>
<tr><td>Cálculo de la métrica “PMA”.</td></tr>
<tr><td>Cálculo de la métrica “TPMA”.</td></tr>
</table>

6.2.2.4 Nombre del Caso de prueba: RESTAURANTE.java

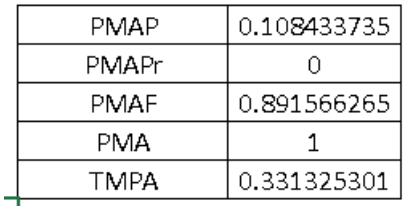
Restaurante es un sistema legado cuya descripción se puede ver en la tabla 14 y los resultados de la métricas son observables en la tabla 15.

Tabla 14. Tabla de conteo de atributos del proyecto RESTAURANTE previo a la refactorización

Atributos	Valor
Total, de clases	89
Total, de atributos	83
Atributos privados	9
Atributos protegidos	0
Atributos friendly	74

Atributos públicos	0

Tabla 15. Tabla de comparación de resultados manuales y automáticos del proyecto RESTAURANTE previo a la refactorización

Pruebas unitarias		
Características a probar	**Resultado manual**	**Resultado de sistema**
Cálculo de la métrica *"PMAP"*.		
Cálculo de la métrica *"PMAPr"*.		
Cálculo de la métrica *"PMAF"*.		
Cálculo de la métrica *"PMA"*.		
Cálculo de la métrica *"TPMA"*.		

En todos los casos antes mencionados se encontró una coincidencia casi exacta siendo la principal diferencia el número de decimales contemplados para el cálculo para el marco de métricas los cuales son 4 y tiende a redondear algunos resultados.

6.2.3.- Caso de Prueba ISMRCAC0503

En el siguiente bloque de pruebas con los mismos 4 proyectos, se harán pasar por el método de refactorización teniendo en cuenta lo siguiente:

Características a probar

- Método de validación del modificador de acceso.
- Identificación de acceso a variables en el rango *public*
- Identificación de acceso a variables en el rango *protected*
- Identificación de acceso a variables en el rango *friendly*
- Identificación de acceso a variables en el rango *private*
- Cambio de modificador de acceso
- Mover variable a método local
- Método de refactorización de modificadores de acceso.
- Llenado de la plantilla "String Template" y generación de código refactorizado.

- Compilación del código refactorizado.
- Verificación del funcionamiento del código de salida.
- Verificación del aumento de protección modular.

6.2.3.1 Nombre del Caso de prueba: PSP3.java

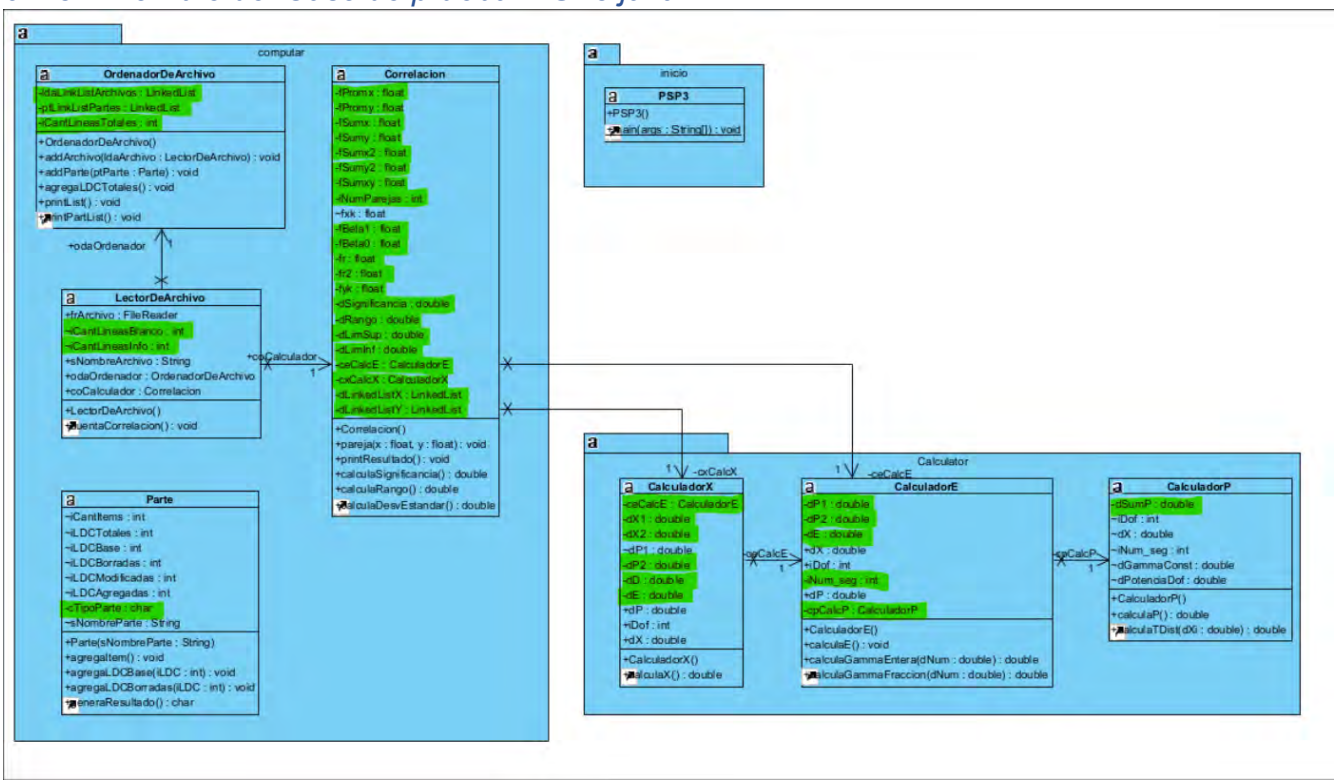


Figura 15. Diagrama de clases de proyecto PSP3 refactorizado

Posterior a la refactorización los resultados fueron los siguientes:

Tabla 16. Tabla de conteo de atributos del proyecto PSP3 después de la refactorización

Atributos	Valor
Total, de clases	8
Total, de atributos	63
Atributos privados	37
Atributos protegidos	0
Atributos friendly	16
Atributos públicos	10

Con los datos recolectados se procede a realizar pruebas de protección modular y contrastar con cálculos manuales:

Tabla 17. Tabla de comparación de resultados de métricas manuales y automáticos del proyecto PSP3 después de la refactorización

<table>
<tr><th colspan="3">Pruebas unitarias</th></tr>
<tr><th>Características a probar</th><th>Resultado manual</th><th>Resultado de sistema</th></tr>
<tr><td>Cálculo de la métrica “PMAP”.</td><td rowspan="6"><table><tr><td>PMAP</td><td>0.587301587</td></tr><tr><td>PMAPr</td><td>0</td></tr><tr><td>PMAF</td><td>0.253968254</td></tr><tr><td>PMA</td><td>0.841269841</td></tr><tr><td>TMPA</td><td>0.650793651</td></tr></table></td><td rowspan="6">Resultado de metrica PMAP : 0.5873
Resultado de metrica PMAPr : 0.0000
Resultado de metrica PMAF : 0.2540
Resultado de metrica PMA : 0.8413
Resultado de metrica TPMA : 0.6508</td></tr>
<tr><td>Cálculo de la métrica “PMAPr”.</td></tr>
<tr><td>Cálculo de la métrica “PMAF”.</td></tr>
<tr><td>Cálculo de la métrica “PMA”.</td></tr>
<tr><td>Cálculo de la métrica “TPMA”.</td></tr>
</table>

Pruebas de funcionalidad:

El proyecto PSP3 se encarga de hacer un análisis de acuerdo a ciertos datos dados al programa. Para la prueba de compilado se usarán los siguientes datos de prueba:

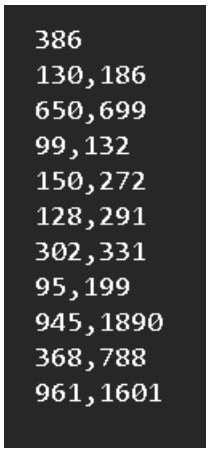
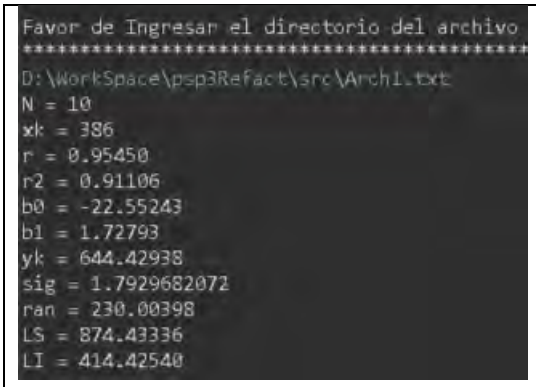
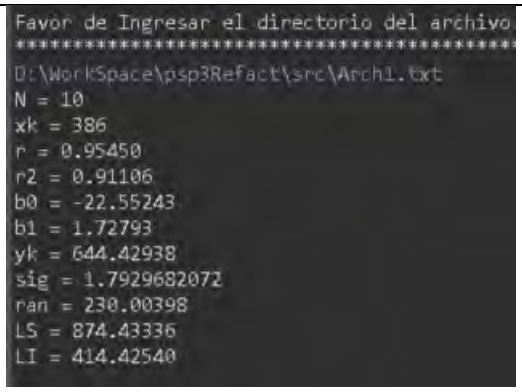


Figura 16. Datos de prueba de compilado para el proyecto PSP3

Tabla 18. Comparativa de resultado de compilado de código antes y después de la refactorización del proyecto PSP3

Comprobación de compilación de código antes y después de refactorizar	
ANTES	DESPUES





6.2.3.2 Nombre del Caso de prueba: SUDOKU.java

Posterior a la refactorización los resultados fueron los siguientes:

Tabla 19. Tabla de conteo de atributos del proyecto SUDOKU después de la refactorización

Atributos	Valor
Total, de clases	28
Total, de atributos	62
Atributos privados	44
Atributos protegidos	4
Atributos friendly	0
Atributos públicos	14

Tabla 20. Tabla de comparación de resultados de métricas manuales y automáticos del proyecto SUDOKU después de la refactorización

Pruebas unitarias		
Características a probar	**Resultado manual**	**Resultado de sistema**
Cálculo de la métrica *"PMAP"*.	PMAP 0.709677419 PMAPr 0. 166666667 PMAF 0 PMA 0. 774193548 TMPA 0.758064516	**Resultado de metrica PMAP : 0.7097** **Resultado de metrica PMAPr : 0.1667** **Resultado de metrica PMAF : 0.0000** **Resultado de metrica PMA : 0.7742** **Resultado de metrica TPMA : 0.7581**
Cálculo de la métrica *"PMAPr"*.		
Cálculo de la métrica *"PMAF"*.		
Cálculo de la métrica *"PMA"*.		

Cálculo de la métrica *"TPMA"*.		

Pruebas de funcionalidad:

En este proyecto, al tratarse de un juego, y no realizar cálculos sistemáticos, una de las formas de probar su compilación es la ejecución del mismo añadiendo los primeros datos y la visualización del mismo en consola.

Tabla 21. Comparativa de resultado de compilado de código antes y después de la refactorización del proyecto SUDOKU

Comprobación de compilación de código antes y después de refactorizar	
ANTES	DESPUES
	

6.2.3.3 Nombre del Caso de prueba: SUSHI.java

Posterior a la refactorización los resultados fueron los siguientes:

Tabla 22. Tabla de conteo de atributos del proyecto SUSHI después de la refactorización

Atributos	Valor
Total, de clases	67
Total, de atributos	74
Atributos privados	32

Atributos protegidos	14
Atributos friendly	5
Atributos públicos	23

Tabla 23. Tabla de comparación de resultados de métricas manuales y automáticos del proyecto SUSHI después de la refactorización

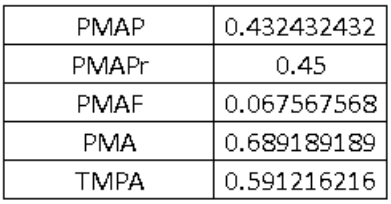
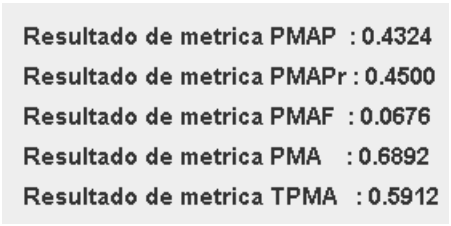
Pruebas unitarias		
Características a probar	**Resultado manual**	**Resultado de sistema**
Cálculo de la métrica *"PMAP"*.		
Cálculo de la métrica *"PMAPr"*.		
Cálculo de la métrica *"PMAF"*.		
Cálculo de la métrica *"PMA"*.		
Cálculo de la métrica *"TPMA"*.		

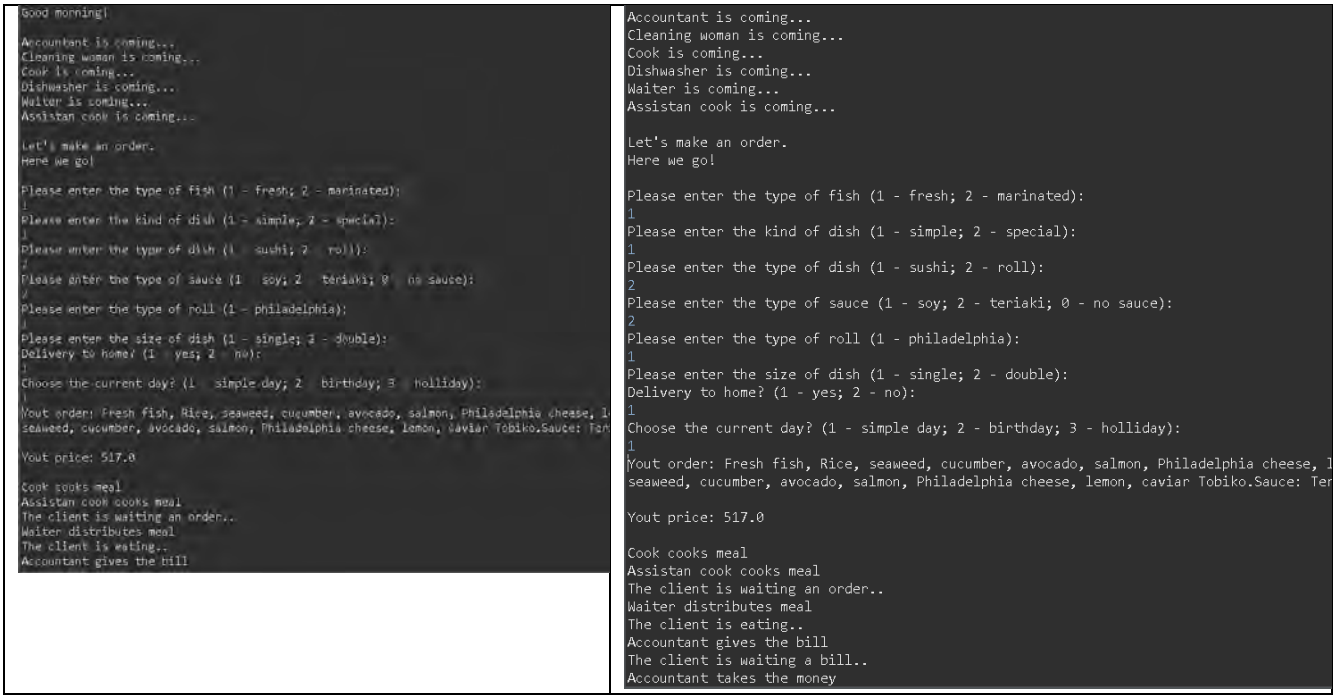
Tabla 10. Comparativa de resultado de métricas manuales y automáticas del proyecto SUSHI después de refactorizar

Pruebas de funcionalidad:

En este proyecto, al tratarse de seleccionador de opciones para el servicio de sushi, la forma de probar su compilación es la ejecución del mismo con los mismos datos y la visualización del mismo en consola y la ejecución de los mismos.

Tabla 24. Comparativa de resultado de compilado de código antes y después de la refactorización del proyecto SUSHI

Comprobación de compilación de código antes y después de refactorizar	
ANTES	DESPUES



6.2.3.4 Nombre del Caso de prueba: RESTAURANTE.java

Posterior a la refactorización los resultados fueron los siguientes:

Tabla 25. Tabla de conteo de atributos del proyecto RESTAURANTE después de la refactorización

Atributos	Valor
Total, de clases	89
Total, de atributos	83
Atributos privados	68
Atributos protegidos	3
Atributos friendly	12
Atributos públicos	0

Tabla 26. Tabla de comparación de resultados de métricas manuales y automáticos del proyecto RESTAURANTE después de la refactorización

<table>
<tr><th colspan="3">Pruebas unitarias</th></tr>
<tr><th>Características a probar</th><th>Resultado manual</th><th>Resultado de sistema</th></tr>
<tr><td>Cálculo de la métrica “PMAP”.</td><td rowspan="6"><table><tr><td>PMAP</td><td>0.819277108</td></tr><tr><td>PMAPr</td><td>0.523364486</td></tr><tr><td>PMAF</td><td>0. 144578313</td></tr><tr><td>PMA</td><td>1</td></tr><tr><td>TMPA</td><td>0.88253012</td></tr></table></td><td rowspan="6">Resultado de metrica PMAP : 0.8193
Resultado de metrica PMAPr : 0.5234
Resultado de metrica PMAF : 0.1446
Resultado de metrica PMA : 1.0000
Resultado de metrica TPMA : 0.8825</td></tr>
<tr><td>Cálculo de la métrica “PMAPr”.</td></tr>
<tr><td>Cálculo de la métrica “PMAF”.</td></tr>
<tr><td>Cálculo de la métrica “PMA”.</td></tr>
<tr><td>Cálculo de la métrica “TPMA”.</td></tr>
</table>

Pruebas de funcionalidad:

En este proyecto, al tratarse de seleccionador de opciones para el servicio de restaurante, la forma de probar su compilación es la ejecución del mismo añadiendo los mismos datos y la visualización del mismo en consola y la ejecución de los mismos.

Tabla 27. Comparativa de resultado de compilado de código antes y después de la refactorización del proyecto RESTAURENTE

Comprobación de compilación de código antes y después de refactorizar	
ANTES	DESPUES

6.3 ANALISIS DE RESULTADOS

A continuación, se muestran los resultados de las métricas antes y después de la refactorización de cada proyecto. Donde:

PMAP: Protección Modular de Atributos Públicos

PMAPr: Protección Modular de Atributos Protegidos

PMAF: Protección Modular de Atributos Friendly

PMA: Protección Modular de Atributos

TPMA: Total de Protección Modular de Atributos

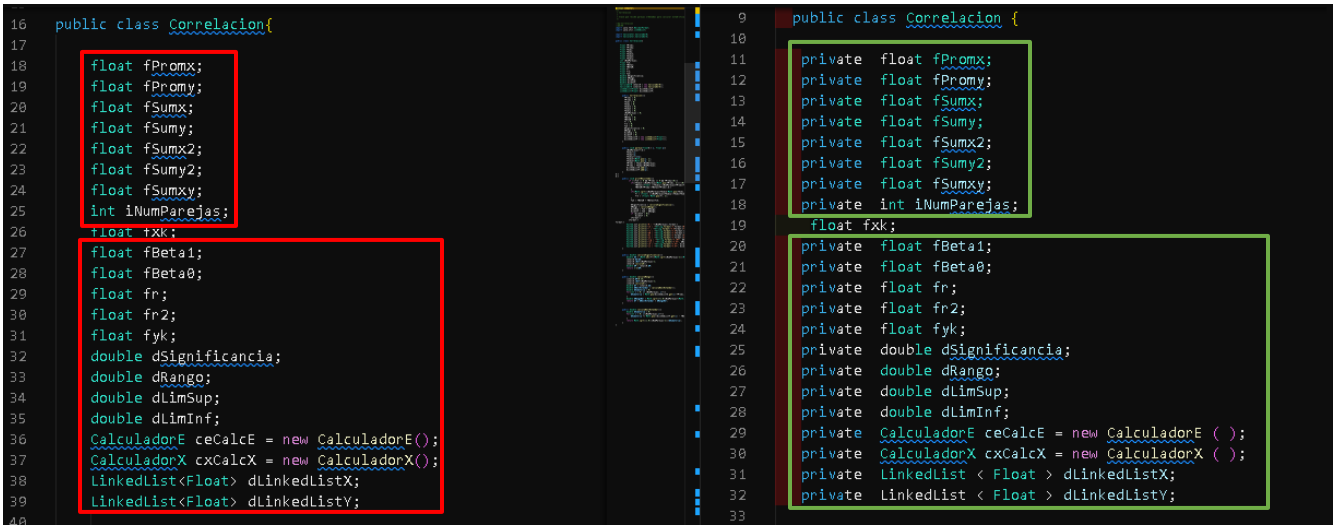
6.3.1 PROYECTO PSP3

Tabla 28. Tabla comparativa de resultado de conteo de atributos antes y después de la refactorización del proyecto PSP3

Atributos	Antes	Después
Totales	63	63
Privados	0	37
Protegidos	0	0
Friendly	51	16
Públicos	12	10

Como se ilustra en la tabla 28, tras el proceso de refactorización, se ha observado un significativo fortalecimiento en la modularidad y protección de los atributos en el programa. Inicialmente, la totalidad de los atributos estaba distribuida sin restricciones de acceso, con 63 en total. Después de la refactorización, se ha logrado una redistribución más segura y controlada, con 37 atributos marcados como 'private', atributos que solo son referenciados dentro de la misa clase donde son definidos, indicando un aumento considerable en la encapsulación y limitación del acceso directo. No se han identificado atributos con modificadores 'protected', ya que el diseño actual no requiere dicha visibilidad externa. Además, se ha reducido el número de atributos con acceso 'friendly' de 51 a 16, enfocándose únicamente en las relaciones internas dentro del mismo paquete, la reducción de estos atributos de ver reflejada en el aumento de atributos “private”. La cantidad de atributos ´public´ ha experimentado una disminución de 12 a 10, indicando una mayor restricción en el acceso desde clases externas. Este proceso ha culminado en una estructura más segura y coherente, favoreciendo la modularidad y evitando accesos indeseados a los atributos del programa.

Como se puede apreciar en la Figura 20, una de las clases más afectadas por estos cambios es la clase Correlacion.Java, donde aproximadamente 21 atributos experimentaron un cambio en su modificador de acceso.



```
public class Correlacion{

    float fPromx;
    float fPromy;
    float fSumx;
    float fSumy;
    float fSumx2;
    float fSumy2;
    float fSumxy;
    int iNumParejas;
    float fxk;
    float fBeta1;
    float fBeta0;
    float fr;
    float fr2;
    float fyk;
    double dSignificancia;
    double dRango;
    double dLimSup;
    double dLimInf;
    CalculadorE ceCalcE = new CalculadorE();
    CalculadorX cxCalcX = new CalculadorX();
    LinkedList<Float> dLinkedListX;
    LinkedList<Float> dLinkedListY;
```

```
public class Correlacion {

    private  float fPromx;
    private  float fPromy;
    private  float fSumx;
    private  float fSumy;
    private  float fSumx2;
    private  float fSumy2;
    private  float fSumxy;
    private  int iNumParejas;
     float fxk;
    private  float fBeta1;
    private  float fBeta0;
    private  float fr;
    private  float fr2;
    private  float fyk;
    private  double dSignificancia;
    private  double dRango;
    private  double dLimSup;
    private  double dLimInf;
    private  CalculadorE ceCalcE = new CalculadorE ( );
    private  CalculadorX cxCalcX = new CalculadorX ( );
    private  LinkedList < Float > dLinkedListX;
    private  LinkedList < Float > dLinkedListY;
```

Figura 20. Comparativa de la clase Coorelacion.Java antes y después de la refactorización

RESULTADOS DEL REFACTORIZACION AL SISTEMA PSP3		
Métricas	ANTES	DESPUES
PMAP	0	0.587301587
PMAPr	0	0
PMAF	0.80952381	0.253968254
PMA	0.80952381	0.841269841
TMPA	0.202380952	0.650793651

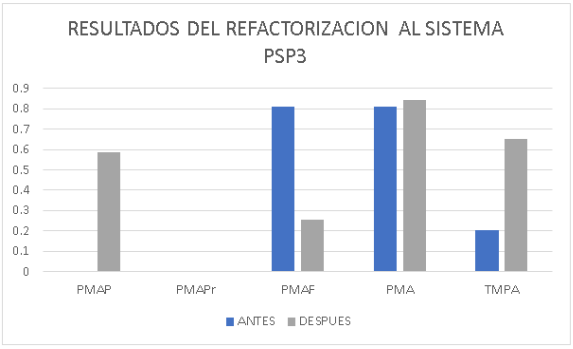


Figura 21. Comparativa de la clase Coorelacion.Java antes y después de la refactorización

La refactorización del sistema PSP3 ha dado lugar a notables mejoras en las métricas que evalúan la protección modular de los atributos. Antes de la intervención, todas las métricas mostraban valores indicativos de una protección modular insuficiente. Tras la refactorización, se evidencia un aumento sustancial en la Protección Modular de Atributos Públicos (PMAP), que ha pasado de un valor nulo a 0.587301587, indicando una restricción más eficaz en el acceso a los atributos desde clases externas. La Protección Modular de Atributos Friendly (PMAF) ha experimentado una significativa disminución, reduciéndose de 0.80952381 a 0.253968254, señalando una limitación más estricta en el acceso a atributos dentro del mismo paquete. Además, la Protección Modular de Atributos (PMA) ha experimentado un incremento de 0.80952381 a 0.841269841, indicando una mayor encapsulación y restricción general de los atributos. Estos cambios han contribuido a una mejora notable en la Total de Protección Modular de Atributos (TPMA), que ha pasado de 0.202380952 a 0.650793651, reflejando una

protección modular global más robusta después de la refactorización. Estos resultados resaltan la efectividad de la intervención en fortalecer la modularidad y la seguridad del sistema PSP3.

6.3.2 PROYECTO SUDOKU

Tabla 29. Tabla comparativa de resultado de conteo de atributos antes y después de la refactorización del proyecto PSP3

Atributos	Antes	Después
Totales	70	62
Privados	44	44
Protegidos	6	4
Friendly	0	0
Públicos	20	14

Como se ilustra en la tabla 29, la refactorización del sistema ha resultado en mejoras sustanciales en la protección modular de los atributos. Inicialmente, el programa contaba con un total de 70 atributos, distribuidos de manera variada en términos de modificadores de acceso. Después de la intervención, se ha observado una disminución en el número total de atributos a 62, indicando una optimización y consolidación del diseño. Esta reducción de 8 atributos de debe a que los mismos fueron referenciados por un solo método dentro de la misma clase donde fueron declarados, por lo que su declaración al principio de la clase innecesaria.

Específicamente, se ha mantenido la privacidad de 44 atributos, todos ellos designados como 'private', lo que refleja una encapsulación efectiva y una restricción rigurosa al acceso directo desde clases externas. La reducción de atributos protegidos de 6 a 4 sugiere una optimización en la exposición controlada de atributos a clases derivadas, resaltando una gestión más eficiente de la herencia. La disminución en atributos públicos de 20 a 14 confirma una reducción en la visibilidad global de los atributos, priorizando una mayor encapsulación, estos 6 atributos forman parte de aquellos que fueron situados dentro de una única función que los referencia.

Cabe destacar que la reducción en el número total de atributos también sugiere una reubicación eficiente de atributos locales, consolidándolos en la única función que los utiliza, lo que contribuye a una mayor cohesión y legibilidad en el código. En resumen, la refactorización ha logrado un aumento significativo en la protección modular, con cambios estratégicos que favorecen la encapsulación y organización eficiente de los atributos en el sistema.

En este caso la reducción y mínimo aumento en las métricas se origina en la reubicación de estos 8 atributos que pasan de ser atributos de clase a atributos de un método.

Figura 22. Comparativa de la clase Board y el movimiento a método local de la variable DEFAULT_HOLES_VALUES

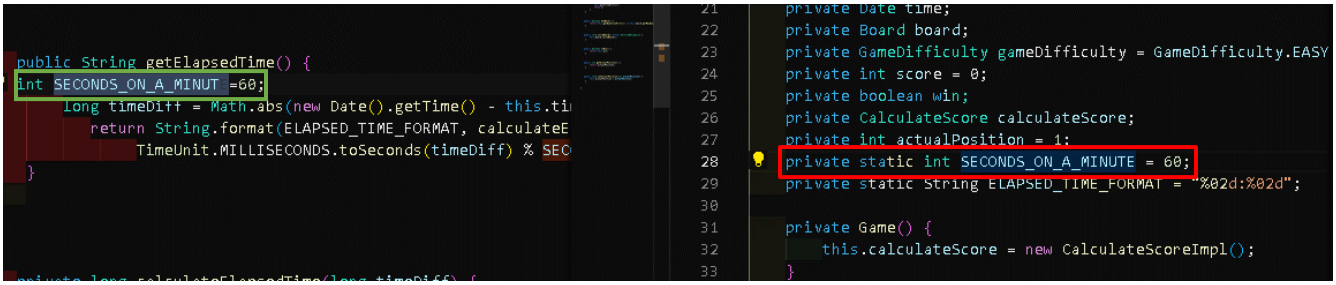


Figura 23. Comparativa de la clase Game y el movimiento a método local de la variable SECONDS_ON_A_MINUTE

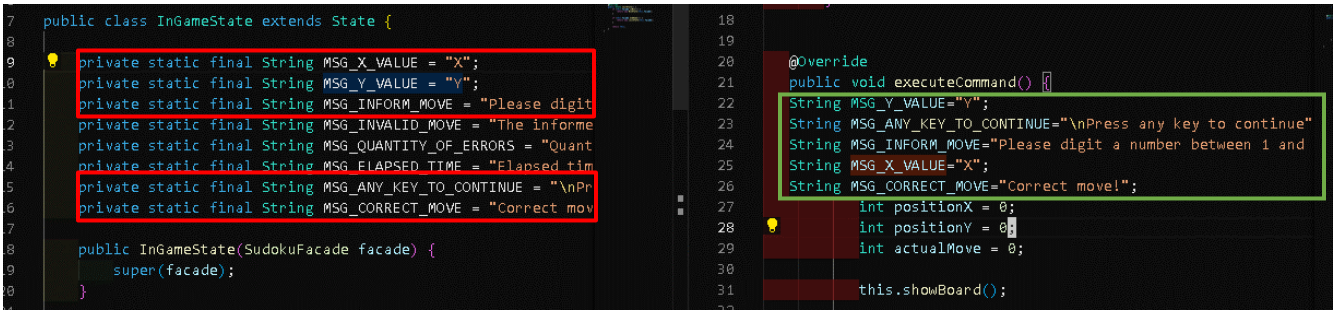


Figura 24. Comparativa de la clase InGameState y el movimiento a método local de 5 variables

RESULTADOS DEL REFACTORIZACION AL SISTEMA SUDOKU		
Métricas	ANTES	DESPUES
PMAP	0.628571429	0.709677419
PMAPr	0.271604938	0. 166666667
PMAF	0	0
PMA	0.714285714	0. 774193548
TMPA	0.692857143	0.758064516

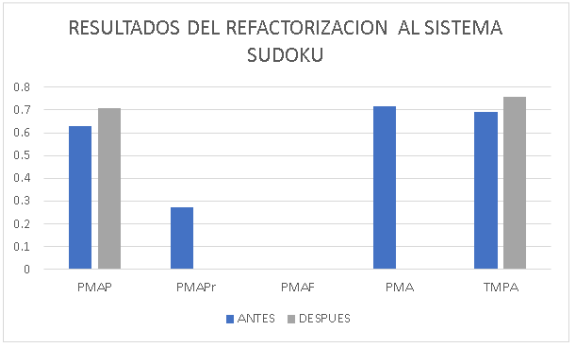


Figura 25 Comparativa de resultado de conteo de atributos de métricas antes y después de la refactorización del proyecto SUDOKU

La refactorización del sistema Sudoku ha generado mejoras notables en las métricas de protección modular, indicando un fortalecimiento en la seguridad y encapsulación de los atributos. Antes de la refactorización, la Protección Modular de Atributos Públicos (PMAP) estaba en 0.628571429, y tras la refactorización ha experimentado un aumento a 0.709677419, señalando una restricción más efectiva en el acceso a los atributos desde clases externas.

La Protección Modular de Atributos Protegidos (PMAPr) ha experimentado una disminución significativa, reduciéndose de 0.271604938 a 0.166666667. Este cambio sugiere una optimización en la exposición controlada de atributos a clases derivadas, indicando una gestión más eficiente de la herencia.

La Protección Modular de Atributos (PMA) ha experimentado un aumento, pasando de 0.714285714 a 0.774193548, indicando una mayor encapsulación y restricción general de los atributos. Este fortalecimiento en la protección modular también se refleja en la Total de Protección Modular de Atributos (TMPA), que ha pasado de 0.692857143 a 0.758064516, evidenciando una protección modular global más robusta después de la refactorización.

En resumen, la refactorización ha resultado en un aumento en la protección modular, con cambios estratégicos que mejoran la seguridad y eficiencia en la gestión de los atributos en el sistema Sudoku.

6.3.3 PROYECTO SUSHI

Tabla 30. Tabla comparativa de resultado de conteo de atributos antes y después de la refactorización del proyecto SUSHI

Atributos	Antes	Después
Totales	75	74
Privados	7	32
Protegidos	1	14
Friendly	30	5
Públicos	37	23

Como se muestra en la tabla 30, revela cambios significativos en la protección modular de los atributos después de la refactorización del programa. En este contexto, se observa que se ha logrado una reducción en el número total de atributos, pasando de 75 a 74. Este ajuste se traduce en una reubicación estratégica de atributos, donde 1 atributo específico, "Chain c1=new TipsBirthDay();" de la clase UI, ha sido movido localmente a un método de clase.

La refactorización ha generado un aumento considerable en el número de atributos privados, que ha pasado de 7 a 32. Este incremento refleja una mayor encapsulación y restricción en el acceso directo desde clases externas, concentrando el acceso a través de métodos GET y SET.

Los atributos protegidos también han experimentado un incremento, elevándose de 1 a 14. Esta mejora indica una gestión más eficiente de la herencia y una mayor especificidad en la exposición controlada de atributos a clases derivadas.

Por otro lado, se ha observado una disminución sustancial en los atributos friendly, reduciéndose de 30 a 5. Esta reducción refleja una estrategia de limitación de acceso a funciones dentro del mismo paquete, enfocándose en una mayor coherencia y seguridad en el diseño al cambiar su calificador de alcance a private y uno siendo reubicado al interior de una función.

Los atributos públicos también han experimentado una disminución, pasando de 37 a 23. Esta reducción indica un esfuerzo deliberado por limitar la visibilidad global de los atributos, favoreciendo una mayor encapsulación y control de acceso al cambiar el calificador de alcance a private y protegido.

En conjunto, estos cambios demuestran una mejora significativa en la protección modular, donde la reducción total de atributos se traduce en una reubicación estratégica, fortaleciendo la encapsulación y limitando el acceso de manera más precisa y controlada en el sistema.

Figura 26. Comparativa de la clase UI en la protección de sus atributos

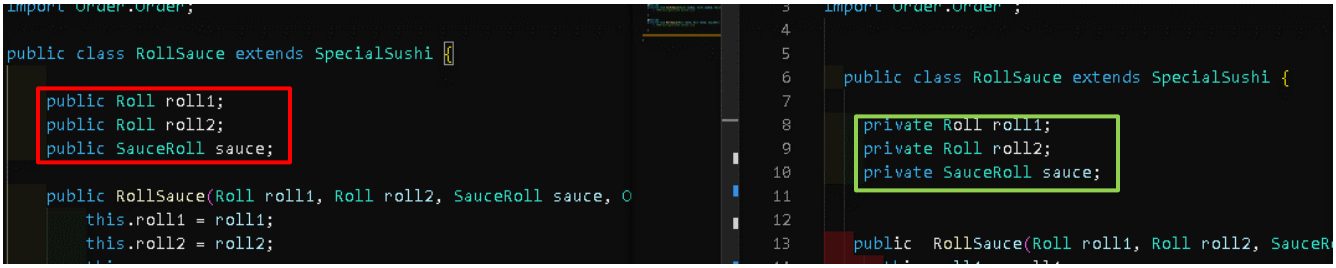


Figura 27. Comparativa de la clase RollSauce en la protección de sus atributos

RESULTADOS DEL REFACTORIZACION AL SISTEMA SUSHI		
Métricas	ANTES	DESPUES
PMAP	0.093333333	0.432432432
PMAPr	0.008333333	0.45
PMAF	0.4	0.067567568
PMA	0.506666667	0.689189189
TMPA	0.203333333	0.591216216

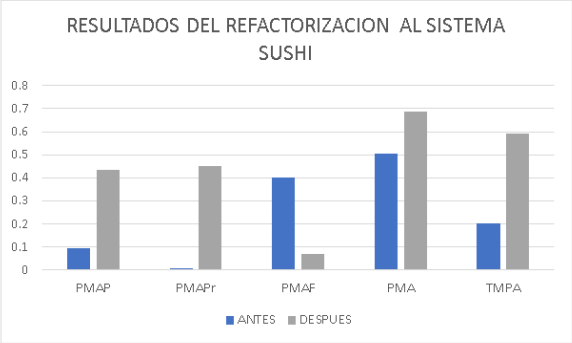


Imagen 28. Tabla comparativa de resultado de conteo de atributos de métricas antes y después de la refactorización del proyecto SUSHI

La refactorización del sistema Sushi ha resultado en mejoras sustanciales en las métricas de protección modular, evidenciando un fortalecimiento significativo en la seguridad y encapsulación de los atributos. Antes de la refactorización, la Protección Modular de Atributos Públicos (PMAP) estaba en un nivel bajo, con un valor de 0.093333333. Tras la refactorización, esta métrica ha experimentado un incremento sustancial, alcanzando un valor de 0.432432432, lo que indica una restricción más efectiva en el acceso a los atributos desde clases externas.

La Protección Modular de Atributos Protegidos (PMAPr) también ha experimentado un aumento significativo, pasando de 0.008333333 a 0.45. Este cambio sugiere una optimización en la exposición controlada de atributos a clases derivadas, indicando una gestión más eficiente de la herencia.

La Protección Modular de Atributos Friendly (PMAF) ha experimentado una reducción drástica, disminuyendo de 0.4 a 0.067567568. Esta disminución refleja una estrategia de limitación más estricta en el acceso a funciones dentro del mismo paquete, destacando una mayor coherencia en el diseño y reduciendo la dependencia de atributos friendly.

La Protección Modular de Atributos (PMA) ha experimentado un incremento notable, pasando de 0.506666667 a 0.689189189, indicando una mayor encapsulación y restricción general de los atributos.

En consecuencia, la Total de Protección Modular de Atributos (TMPA) ha experimentado una mejora sustancial, elevándose de 0.203333333 a 0.591216216, evidenciando una protección modular global más robusta después de la refactorización.

En resumen, la intervención ha resultado en un aumento significativo en la protección modular, con cambios estratégicos que mejoran la seguridad y eficiencia en la gestión de los atributos en el sistema Sushi.

6.3.4 PROYECTO RESTAURANTE

Tabla 31. Tabla comparativa de resultado de conteo de atributos antes y después de la refactorización del proyecto SUSHI

Atributos	Antes	Después
Totales	83	83
Privados	9	68
Protegidos	0	3
Friendly	74	12
Públicos	0	0

Como se muestra en la tabla 31, la refactorización del sistema ha resultado en cambios sustanciales en la protección modular de los atributos, evidenciando una mejora significativa en la seguridad y encapsulación del código. Antes de la refactorización, el programa presentaba un total de 83 atributos, distribuidos de manera considerable en la categoría de atributos friendly. Después de la refactorización, aunque el número total de atributos se mantiene en 83, se ha logrado una redistribución más efectiva en términos de modificadores de acceso.

Específicamente, se ha observado un aumento sustancial en el número de atributos privados, que ha pasado de 9 a 68. Esta elevación indica una mayor encapsulación y restricción en el acceso directo desde clases externas, mejorando así la seguridad y modularidad del sistema.

La introducción de atributos protegidos, que antes era nula, ha alcanzado un total de 3 después de la refactorización. Esta inclusión sugiere una optimización en la gestión de la herencia y una mayor especificidad en la exposición controlada de atributos a clases derivadas.

En contraste, la categoría de atributos friendly ha experimentado una reducción significativa, pasando de 74 a 12. Esta disminución refleja una estrategia deliberada de limitar el acceso a

funciones dentro del mismo paquete, favoreciendo una mayor coherencia y seguridad en el diseño.

La eliminación completa de atributos públicos después de la refactorización indica un esfuerzo por limitar la visibilidad global de los atributos, priorizando una mayor encapsulación y control de acceso.

En conjunto, estos cambios demuestran una mejora sustancial en la protección modular del sistema, con un enfoque estratégico que favorece la encapsulación y limita el acceso de manera más precisa y controlada en el código.

RESULTADOS DEL REFACTORIZACION AL SISTEMA RESTAURANTE		
Métricas	ANTES	DESPUES
PMAP	0.108433735	0.819277108
PMAPr	0	0.523364486
PMAF	0.891566265	0. 144578313
PMA	1	1
TMPA	0.331325301	0.88253012

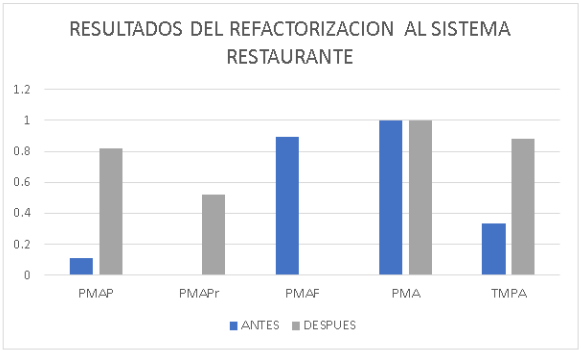


Figura 29. Comparativa de resultado de conteo de atributos de métricas antes y después de la refactorización del proyecto RESTAURANTE

La refactorización del sistema Restaurante ha resultado en mejoras significativas en las métricas de protección modular, reflejando un substancial fortalecimiento en la seguridad y encapsulación de los atributos. Antes de la refactorización, la Protección Modular de Atributos Públicos (PMAP) era relativamente baja, con un valor de 0.108433735. Sin embargo, después de la refactorización, esta métrica ha experimentado un aumento considerable, alcanzando un valor de 0.819277108, lo que indica una restricción más efectiva en el acceso a los atributos desde clases externas.

La Protección Modular de Atributos Protegidos (PMAPr), que inicialmente estaba en cero, ha experimentado un incremento significativo, alcanzando un valor de 0.523364486 después de la refactorización. Este cambio refleja una optimización en la exposición controlada de atributos a clases derivadas, indicando una gestión más eficiente de la herencia.

La Protección Modular de Atributos Friendly (PMAF) ha experimentado una reducción drástica, disminuyendo de 0.891566265 a 0.144578313. Esta disminución refleja una estrategia deliberada de limitación más estricta en el acceso a funciones dentro del mismo paquete, destacando una mayor coherencia en el diseño y reduciendo la dependencia de atributos friendly.

La Protección Modular de Atributos (PMA) ha permanecido constante en 1 antes y después de la refactorización, indicando un alto nivel de encapsulación y restricción general de los atributos.

En consecuencia, la Total de Protección Modular de Atributos (TMPA) ha experimentado un incremento sustancial, elevándose de 0.331325301 a 0.88253012, evidenciando una protección modular global más robusta después de la refactorización.

En resumen, la intervención ha resultado en un aumento significativo en la protección modular, con cambios estratégicos que mejoran la seguridad y eficiencia en la gestión de los atributos en el sistema Restaurante.

Capítulo 7- CONCLUSIONES

7.1 Conclusiones

El uso correcto de los modificadores de acceso y el principio de ocultación de la información asegura un buen nivel de protección y encapsulamiento, asegurando que la información declarada privada solo se puede acceder dentro de la entidad de software en el que es declarado, y no desde otra entidad de software externa capaz de acceder a ella indirectamente. Además, la encapsulación permite la autonomía de los objetos y por lo tanto coordinar el funcionamiento interno de los objetos para que pueda modificarlos sin afectar el comportamiento de otros objetos que puedan ser del mismo tipo.

Aprendizajes obtenidos durante el desarrollo de esta investigación:

- El uso correcto de los modificadores de acceso en el lenguaje Java (*private*, *protected*, *friendly* y *public*). Restringir el acceso a los miembros de una clase es una parte fundamental de la programación orientada a objetos, ya que ayuda a evitar:
  1) el mal uso de la información que puede ser confidencial o delicada en su uso
  2) resultados falsos o incorrectos.
- Proteger los atributos es fundamental para evitar posibles accesos y modificaciones a sus valores desde entidades de software que no deberían tener este acceso.
- Con base en las pruebas se puede determinar que un proyecto puede llegar a tener un resultado de 1 en la métrica TPMA, siendo que existen las llamadas funciones GET y SET que permiten acceder y leer o modificar los valores de los atributos respectivamente. Sin embargo, se puede dar el caso de que estos métodos no tengan un modificador de acceso correcto por lo que se recomienda usar el método de refactorización de Nélida Barón dedicado a la protección funcional de los métodos posteriormente a este método de refactorización.

- Las pruebas han demostrado un incremento en los atributos privados después de la refactorización sin provocar errores sintácticos o de ejecución o de compilación.
- El desarrollo de este sistema de refactorización favorece la medición de protección modular de atributos, al mismo tiempo que ayuda a la mejora de un sistema legado cuando este cuenta con un bajo grado de protección modular de atributos, lo cual favorece al encapsulamiento, genericidad y la autonomía modular lo que fomenta el reuso de software y la independencia de las entidades de software. En conclusión, de este punto, se fomenta una mayor calidad del software tanto a nivel de diseño como a nivel de código y se facilita su mantenimiento.
- Existen aquellos atributos que han sido declarados y en ningún momento suelen ser accedidos por ninguna entidad de software, también llamado “código dispensable” (Alexander, S. ,2019). Así mismo se identificaron entidades de software declaradas sin que estas contuviesen algún método o atributo declarado, se recomienda en trabajos futuros la eliminación de este “código dispensable”.
- La construcción del método no toma de forma previa la arquitectura del código legado, por lo que sus cambios son en base a las llamadas y uso de los atributos dentro del mismo. Por lo que es posible que algunos patrones de diseño puedan presentar dificultades a la hora de ser integrados a la estructura contenedora como es el caso del patrón de diseño Iterador, el cual en su estructura tiene 2 clases dentro del mismo archivo java, un escenario no contemplado al crear la estructura contenedora. Por lo que se recomienda en trabajos futuros un análisis de la arquitectura previo a la refactorización.
- La mayor experiencia en desarrollo no es sinónimo de calidad, ya que aún con una gran cantidad de experiencia los desarrolladores están expuestos a errores o deficiencias de diseño.

El proceso de refactorización de modificadores de acceso de atributos ha sido comprobado como una técnica eficaz para mejorar los aspectos de diseño relacionados con los niveles de acceso en un sistema de software escrito en lenguaje Java. Esta técnica puede reducir la deuda técnica asociada con la falta de protección modular y el incumplimiento del principio de ocultamiento de datos mediante el uso de reglas de visibilidad. Los resultados demuestran que el uso de esta técnica de refactorización puede aumentar la protección modular de un sistema de software, lo que a su vez ayuda a aplicar el principio de ocultamiento de información de manera adecuada. Esto previene la manipulación accidental de los detalles internos de las entidades de software por parte de agentes externos.

Algunas de las condiciones en las cuales se podrían generar los escenarios de fracaso son:

- Que el usuario selecciona una carpeta que no contenga ningún archivo Java, por lo tanto, el método de refactorización no podrá ejecutarse y el proceso terminara sin hacer nada.

- Que algunas de las palabras reservadas no se encuentren implementadas en la plantilla *StringTemplate*. Teniendo en cuenta a aquellas librerías desarrolladas por el propio desarrollador.

Con base en las pruebas realizadas, se puede concluir que la técnica de refactorización desarrollada en este trabajo cumple con el objetivo de mejorar el diseño de arquitecturas de software orientadas a objetos utilizando las convenciones de visualización admitidas por el lenguaje de programación Java. Se desarrolló un método para refactorizar modificadores de alcance a tributos de clase para aplicar a sistemas programados en el lenguaje Java 8 con el fin de establecer los modificadores de alcance correctos para cada característica que conforma un sistema de software. Este incrementará las capacidades del sistema SR2-Refactoring, logrando un objetivo específico y la expansión del sistema en las siguientes características:

a) Facilitar el mantenimiento del Software legado escrito en lenguaje Java 8, al quedar protegidas de la manipulación externa
b) Habilitar el reuso de componentes del Software Legado escritos en lenguaje Java 8 mejorando el encapsulamiento y modularidad.
c) Incrementar las capacidades del “SR2-Refactoring" y apoyar los métodos de refactorización que tienen un gran impacto en el código de texto del lenguaje Java, el incremento de capacidad consiste en agregar el método de refactorización de modificadores de acceso de atributos de clase que ya soporta este sistema.

7.2.- APORTACIONES DE LA TESIS

7.2.1.- MÉTODO DE REFACTORIZACIÓN DE MODIFICADORES DE ACCESO.

El método es capaz de aumentar la protección modular de sistemas escritos en lenguaje Java, mediante la colocación de modificadores de acceso que impiden el acceso externo indiscriminado a cada una de las funciones de las clases que conforman el sistema a refactorizar, así mismo se mejora el nivel de encapsulamiento.

7.2.1.- CONJUNTO DE MÉTRICAS PARA MEDIR EL GRADO DE PROTECCIÓN MODULAR DE ATRIBUTOS DE CLASE.

1. PMAP (Protección Modular de Atributos Privados)
2. PMAPr (Protección Modular de Atributos Protegidos)
3. PMAF (Protección Modular de Atributos Friendly)
4. PMA (Protección Modular de Atributos)
5. TPMA (Total Protección Modular de Atributos)

7.3.- TRABAJO A FUTURO

7.3.1.- IDENTIFICACIÓN DEL CODIGO DISPENSABLE

Actualmente el método de refactorización de modificadores de acceso no tiene la capacidad de tratar un tipo de código identificado como “código muerto”, “código inalcanzable” o “código dispensable” (Alexander, S. ,2019).

El "código muerto" o "código inalcanzable" es un problema común en el desarrollo de software y se refiere a partes de código que no son utilizadas en ninguna parte del programa. Esto puede deberse a una mala planificación o cambios en los requisitos del software a lo largo del tiempo. Estos fragmentos de código pueden afectar negativamente la calidad y la eficiencia del software y, por lo tanto, es importante encontrar y eliminar el código muerto.

BIBLIOGRAFIA

N. Barón, “Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existentes”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2020.

Cárdenas Robledo Leonor Adriana, “Refactorización de marcos orientados a objetos para reducir el acoplamiento aplicando el patrón de diseño Mediator”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2004.

Cárdenas Robledo Leonor Adriana, “Método De Refactorización De Marcos De Aplicaciones Orientados A Objetos Por La Separación De Inteifaces”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2004.

Pablo Padilla Salgado, “Método De Re-factorización De Código Java Con Interfaces Y Abstracciones Incorrectas”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2019.

Orlando Ortiz Gutierrez, “Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2020.

Nelida Baron Perez, “Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2020.

Sanchez Rogel, Fernando, “Re-factorización de arquitecturas de software con carencia de abstracciones”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2023.

Elías A. Ramírez García, “Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2023.

Juan Antonio Diaz Diaz, “Disminución de Deuda Técnica producida por arquitecturas de clases con más de Una Responsabilidad”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2023.

Marisol Ramírez Cruz, “Métodos de Re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2022.

Orlando Ortíz Gutiérrez, “Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2020.

Pablo Padilla Salgado, “Métodos de refactorización de código Java con interfaces y abstracciones incorrectas”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2019.

Luis Esteban Santos Castillo, “Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2005.

Leonor Adriana Cárdenas Robledo, “Re-factorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2004.

Manuel Alejandro Valdés Marrero, “Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2004.

Laura Alicia Hernández Moreno, “Factorización De Funciones Hacia Métodos De Plantilla”, National Technology of Mexico/National Center for Technological Research and Development (CENIDET), 2003.

Sznajdleder, P. A. El gran libro de Java a Fondo 4a Ed.. España: Marcombo. (2020).

Santaolaya Salgado, Rene REDUCCIÓN DE LA DEUDA TÉCNICA POR LA FRAGILIDAD MODULAR DE ARQUITECTURAS DE SOFTWARE LEGADO, cenidet/CAIS.PRY-intD-21.01, (marzo, 2021)

M. Fowler, K. Beck, J. Brant, and W. Opdyke, Refactoring: Improving the design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999

Warren S. Torgerson, Theory and methods of scaling. New York: John Wiley and Sons, Inc., 1958.

Narayan Ramasubbu, Chris F. Kemerer, “Controlling Technical Debt Remediation in Outsourced Enterprise Systems Maintenance: An Empirical Analysis”, Journal of Management Information Systems, 2021

Ganesh Samarthyam, Girish Suryanarayana, Tushar Sharma, “Refactoring for Software Architecture Smells”, 2017

Mohan, M., Greer, D., & McMullan, P., “Technical Debt Reduction using Search Based Automated Refactoring”, Journal of Systems and Software ,2016

Nikhil Oswal, “Technical Debt_Identify Measure and Monitor”, School of Electrical Engineering and Computer Science (EECS), 2015

Gustavo Damián Campo, “Patrones de Diseño, Refactorización y Antipatrones.

Ventajas y Desventajas de su Utilización en el Software Orientado a Objetos”, 2009

Panagiotis Kouros, Theodore Chaikalis, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Theodoros Amanatidis, “JCaliper: Search-Based Technical Debt Management”, 2019

Anna Maria Eilertsen, “Making Software Refactorings Safer”, Department of Informatic, 2016

Girish Suryanarayana, Ganesh Samarthyam & Tushar Sharma, “Refactoring for Software Design Smells”, Elsevier, 2015

Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, Yann Gaël Guéhéneuc, “Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations”, University of Vale do Rio dos Sinos Polytechnic School, 2020

Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo, “Performance-driven Software Model Refactoring”, University of L’Aquila, L’Aquila, Italy, 2018

Jehad Al Dallal, “Constructing Models for Predicting Extract Subclass Refactoring Opportunities Using Object-Oriented Quality Metrics”, Department of Information Science Kuwait University, 2017

Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman, “Managing Technical Debt in Software Engineering”, University of Groningen, 2016

Parr, T. (2013). The Definitive ANTLR 4 Reference. Retrieved from https://www.antlr.org/

Parr, T. (2019). StringTemplate. Retrieved from https://www.stringtemplate.org/

和太, “Cohesion & Coupling Contents”. Nanjing University, 21 de julio de 2018.

Rathee, A., & Chhabra, J. K. Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering, 2017.

Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, Willian Oizumi, “Refactoring effect on internal quality attributes: What haven’t they told you yet?” (2020, 1 octubre).ScienceDirect. https://www.sciencedirect.com/science/article/abs/pii/S0950584920301142

Khatchadourian, R. (2017). Defaultification Refactoring : A Tool for Automatically Converting Java Methods to Default, 984–989.

T. (2021, 6 agosto). Refactor a field to a property - Visual Studio (Windows). Microsoft Docs. https://docs.microsoft.com/en-us/visualstudio/ide/reference/encapsulate-field?view=vs-2022

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