EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente

presentada por

Ing. Nelida Barón Pérez

como requisito para la obtención del grado de

Maestra en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Diciembre de 2020.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Centro Nacional de Investigación y Desarrollo Tecnológico
Departamento de Ciencias Computacionales

"2020, Año de Leona Vicario, Benemérita Madre de la Patria"

Cuernavaca, Mor., 22/octubre/2020

OFICIO No. DCC/117/2020
Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

C. DR. CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de la **C. Ing. Nelida Barón Pérez**, con número de control M18CE064, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente"** 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.

Dr. René Santaolaya Salgado
Doctor en Ciencias de la Computación
4454821
Director de tesis

Dra. Olivia Graciela Fragoso Diaz
Doctora en Ciencias en Ciencias de la Computación
7420199
Revisor 1

Dr. Moisés González García
Doctor en Ciencias en la Especialidad de Ingeniería Eléctrica
7501724
Revisor 2

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

Cuernavaca, Morelos 05/noviembre/2020
OFICIO No. SAC/ 260/2020
Asunto: Autorización de impresión de tesis

NELIDA BARÓN PÉREZ
CANDIDATA AL GRADO DE MAESTRA EN CIENCIAS
DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado ***"Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente"***, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

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

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

DR. CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO SUBDIRECCIÓN ACADÉMICA

C.c.p. M.E. Guadalupe Garrido Rivera. Jefa del Departamento de Servicios Escolares
Expediente
CMAZ/CHG

DEDICATORIAS

Esta tesis está dedicada a:

A mis amados padres, por su gran apoyo, dedicación y amor, no tengo palabras para expresar lo mucho que los amo. Gracias a ustedes soy lo que soy, hoy cumplimos un sueño más. Agradezco a dios por haberme bendecido con los mejores padres que puede haber tenido.

A mi hermana Ivon, por su gran amor, cariño, paciencia y apoyo incondicional, no cabe buda que dios me ama por haberme bendecido con una hermana como tú, te amo hermana.

A mis amados hijos Joan, Emiliano y Alexis, que son toda mi fuerza e inspiración, por ustedes seguiré trabajando fuertemente y alcanzare todas las metas.

A mi hermano Cesar que siempre esta cuando más lo necesito, gracias por mostrarme que nunca debemos darnos por vencidos, te amo hermano, agradecida con dios por darme un hermano como tú.

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 Ciencia y Tecnología (CONACYT) por el apoyo económico brindado durante la realización de mis estudios de Maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por su amistad, consejos, generosidad y gran apoyo para la realización de este trabajo.

A mis revisores, Dra. Olivia Graciela Fragoso Diaz y Dr. Moisés González García por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

A la profesora Patricia Armas León por su gran generosidad, amistad y apoyo para el aprendizaje del idioma inglés.

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

A mis padres Mercedes y Cesáreo por su apoyo incondicional en todo momento de mi vida, gracias por todo mis amados padres es un orgullo ser su hija.

A mi hermana Ivon que siempre está a mi lado motivándome y apoyándome, te amo hermanita.

A mi hermano Cesar por todo tu apoyo y consejos, te amo hermano.

A mis compañeros y amigos que estuvieron conmigo en todo el recorrido de mi carrera, por la convivencia que tuvimos, confianza: Aracely, Carlos, Ana, Daniel, Diego, Irving, Moisés, Orlando, Julia, Marisol y Ricardo.

Resumen

La programación orientada a objetos es un paradigma de programación que usa objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Esta se basa en varias técnicas como: herencia, abstracción, polimorfismo y encapsulamiento. Además, se apoya en el uso de principios y criterios de diseño de software, como lo son el principio "*ocultamiento de información*" y la "*protección modular*".

El desarrollo de software con calidad nos lleva a analizar los principales problemas que se pueden presentar en el desarrollo de aplicaciones orientadas a objetos, que originan deuda técnica. Entre estos problemas está ignorar la protección modular e ignorar el principio de ocultamiento de información en base a las reglas de visibilidad, que previenen el diseño incorrecto de las diferentes entidades de software, ya sean éstas métodos, clases de objetos, módulos de programa o paquetes. Diseños incorrectos con carencia de protección modular están expuestos a la manipulación inadvertida de agentes externos, lo cual puede originar fragilidad en las entidades de software por la propagación de defectos de un módulo hacia otros módulos.

Con el objetivo de proteger de la manipulación externa a las entidades de software de un sistema, así como para alcanzar un mayor grado de encapsulamiento, en este trabajo de investigación se propone un método de refactorización denominado "Método de refactorización de calificadores de alcance", el cual identifica y coloca el calificador de alcance correcto a cada una de las funciones (métodos) que se encuentran en las clases de objetos que conforman una aplicación escrita en lenguaje Java.

Para la verificación de alcance de los objetivos, en este trabajo de investigación se propone un conjunto de cinco métricas de calidad, las cuales miden el grado de protección modular de los diferentes niveles de visibilidad: PMFP (Protección Modular de Funciones Privadas), PMFPR (Protección Modular de Funciones Protegidas), PMFF (Protección Modular de Funciones Friendly), PM (Protección Modular) y TPM (Total Protección Modular).

Para efectos de pruebas se realizaron tres casos de estudio en los cuales se aplica 1) el cálculo del conjunto de métricas definido y 2) el "Método de refactorización de calificadores de alcance". Los resultados obtenidos demuestran el correcto funcionamiento de las métricas PM y del método de refactorización. Los resultados de las pruebas demuestran que hubo una mejora de protección modular. En el caso uno, de nueve clases de la arquitectura tuvieron mejora dos de ellas, así se mejoró el 22.22% de las clases en esa arquitectura. En el caso dos, de cuarenta y siete clases de la arquitectura tuvieron mejora catorce de ellas, así se mejoró el 29.78% de las clases. En el caso tres, de cincuenta y dos clases de la arquitectura tuvieron mejora once de ellas, así de mejoró el 21.15% de las clases en esa arquitectura.

ABSTRACT

Object-oriented programming is a programming paradigm that uses objects in their interactions, to design applications and computer programs. This is based on several techniques such as: inheritance, abstraction, polymorphism and encapsulation. In addition, it is supported by the use of software design principles and criteria, such as the “information concealment” principle and “modular protection”.

The development of quality software leads us to analyze the main problems that can arise in the development of object-oriented applications, which cause technical debt. Among these problems is ignoring modular protection and ignoring the principle of hiding information based on visibility rules, which prevent the incorrect design of the different software entities, be they methods, object classes, program modules or packages. Incorrect designs with a lack of modular protection are exposed to inadvertent manipulation by external agents, which can cause fragility in software entities due to the propagation of defects from one module to other modules.

In order to protect the software entities of a system from external manipulation, as well as to achieve a higher degree of encapsulation, in this research work a refactoring method called "Refactoring method of scope qualifiers" is proposed. which identifies and places the correct scope qualifier to each of the functions (methods) found in the object classes that make up an application written in the Java language.

For the verification of the scope of the objectives, this research work proposes a set of five quality metrics, which measure the degree of modular protection of the different levels of visibility: PMFP (Modular Protection of Private Functions), PMFPR (Modular Protection of Protected Functions), PMFF (Modular Protection of Friendly Functions), PM (Modular Protection) and TPM (Total Modular Protection).

For testing purposes, three case studies were carried out in which 1) the calculation of the set of defined metrics and 2) the "Refactoring method of scope qualifiers" is applied. The results obtained demonstrate the correct operation of the PM metrics and the refactoring method. The test results show that there was a modular protection improvement. In case one, out of nine architecture classes, two of them had improvement, thus 22.22% of the classes in that architecture were improved. In case two, of forty-seven classes of architecture, fourteen of them had improvement, thus 29.78% of the classes were improved. In case three, out of fifty-two classes of architecture, eleven of them improved, thus 21.15% of the classes in that architecture improved.

Contenido

Índice de figuras

Índice de Tablas

GLOSARIO DE TÉRMINOS

Refactorizar	Como sustantivo: "Es un cambio realizado en la estructura interna del software para facilitar su comprensión y hacer más barato de modificar, sin cambiar su comportamiento observable." Como verbo: "Es la reestructura del software aplicando una serie de refactorizaciones sin cambiar su comportamiento observable" (Fowler, 2018).
Deuda técnica	Es la deuda que se acumula por la toma decisiones de diseño incorrectas o no óptimas (Girish Suryanarayana, Ganesh Samarthyam, 2015)
Código desagradable (code smell)	Son problemas asociados a fragmentos del código fuente de un software que, aunque no impiden que este funcione correctamente, deben ser reestructurados pues generan un impacto negativo en su calidad (Laobel, Betancourt, & Martínez, 2020).
Protección modular	Se entiende por protección modular al detalle que se define en el protocolo de la clase para regular la visibilidad de la información de los objetos por entidades externas, de manera que se evite la propagación de defectos o cambios de un módulo hacia otro módulo lo cual puede incrementar la fragilidad del sistema legado.
SR2 Refactoring	Acrónimo de "Sistema de Reingeniería de Software Legado para Reuso" que identifica problemas de diseño en el software y sugiere su refactorización.
Fragilidad	Cuando se realiza un cambio, partes inesperadas del sistema dejan de funcionar.
Encapsulamiento	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 o calificadores de alcance) (Llinás, 2010)
Frameworks	Un marco orientado a objetos (conocidos como frameworks en la literatura escrita en lenguaje inglés) se define como un conjunto semi-completo de clases en colaboración que incorpora un diseño genérico, el cual puede adaptarse a una variedad de problemas específicos

	para producir nuevas aplicaciones hechas a la medida (René, 2003).

Capítulo 1.- Introducción

Producir software de calidad orientado a objetos demanda al desarrollador una gran capacidad de imaginación, abstracción y creatividad, para plantear soluciones correctas a problemas prácticos de aplicaciones informáticas. Para el ser humano, estas capacidades son difíciles de ejercer y aún más al usarlas en conjunto.

Cuando el desarrollador de Software carece de experiencia y habilidad en el desarrollo de aplicaciones orientadas a objetos, suele hacerle frente a diferentes problemas que se presentan durante el desarrollo de una aplicación con malas elecciones de diseño e implementación resultando en la presencia de *"código desagradable (smell code)"* dentro de la aplicación en desarrollo el cual produce una "*deuda técnica*" en el código, además de un impacto negativo en la comprensibilidad y mantenibilidad de éste (Tufano, Oliveto, & Penta, 2017). *"Código desagradable (smell code*)" es una propiedad inherente del software que genera problemas del código o diseño que dificultan la evolución y el mantenimiento del software (Kumar, 2019) *y "deuda técnica"* se define como: la deuda que se acumula por la toma decisiones de diseño incorrectas o no óptimas (Girish Suryanarayana, Ganesh Samarthyam, 2015). Todo lo anterior nos lleva a analizar los principales problemas que se pueden presentar en el desarrollo de aplicaciones orientadas a objetos, que originan una "*deuda técnica*".

Una de las malas elecciones por parte del desarrollador es calificar, indiscriminadamente, a la mayoría o la totalidad de las funciones que conforman a las aplicaciones en desarrollo con un calificador de alcance "*public*", resultando aplicaciones que exhiben una modularidad incorrecta y bajo grado de protección modular, por lo que pueden redundar en un incorrecto nivel de encapsulamiento haciendo a estas aplicaciones poco reusables y costosas en su mantenimiento. Las reglas de visibilidad, el principio de ocultamiento de información y la protección modular previenen el diseño incorrecto de métodos, clases de objetos y módulos de programas o paquetes. Un diseño con carencia de protección modular se encuentra expuesto a manipulaciones externas pudiendo producir resultados incorrectos o falsos, además de originar fragilidad en las entidades de software por la propagación de defectos de un módulo hacia otros módulos.

En este documento de tesis se presenta el diseño e implementación de un método que tiene como objetivo aumentar la protección modular de las arquitecturas de clases. Para lograr este objetivo es necesario que cada atributo, así como cada función integradas en las clases de objetos de una arquitectura, presenten el calificador de alcance correcto de acuerdo a los cuatro niveles de visibilidad: 1) los atributos y funciones privadas ("*private*"), son las más protegidas puesto que solamente son utilizadas por otras funciones dentro de la misma clase; 2) los atributos y funciones protegidas ("*protected*"), solo pueden ser utilizadas por otras funciones de la misma clase o de clases derivadas; 3) los atributos y funciones amistosas ("*friendly*"), son aquellas utilizadas por otras funciones de cualquier clase integradas dentro del mismo paquete; 4) los atributos y las funciones públicas ("*public*"), son las más permisivas puesto que pueden ser utilizadas por cualquier función de cualquier clase en el sistema de software.

Adicionalmente, en esta tesis se realizó el diseño y la implementación de cinco métricas para medir la protección modular de arquitecturas orientadas a objetos.

Organización de este documento de tesis

El presente documento de tesis se organiza de la siguiente manera:

En el capítulo 2 se plantea el problema de la investigación, una propuesta de solución al problema planteado, también se plantean los objetivos de este trabajo de investigación, y se describe en resumen a los trabajos asociados a la misma línea de este trabajo de investigación.

El capítulo 3 muestra el marco teórico, en el cual se sustentan los conceptos fundamentales para el entendimiento de esta tesis.

En el capítulo 4 se describe el conjunto de métricas, las cuales fueron definidas, diseñadas y desarrolladas en esta tesis. Estas métricas son: PMFP (Factor de Protección Modular de Funciones Privadas), PMFPR (Factor de Protección Modular de Funciones Protegidas), PMFF (Factor de Protección Modular de Funciones Friendly), PM (Protección Modular) y TPM (Total de Protección Modular).

El capítulo 5 muestra todo el proceso para el desarrollo de la solución propuesta que consiste del método de refactorización del calificador de alcance. El proceso incluye el modelado del sistema con diagramas de casos de uso, diagramas de secuencias, diagramas de actividades y diagramas de clases.

El capítulo 6 muestra todo lo referente al plan de pruebas, en el que se definen los objetivos de las pruebas y el diseño de pruebas para comprobar el correcto funcionamiento del método.

Por último, en el capítulo 7 se presentan las conclusiones, aportaciones de esta tesis y el trabajo futuro.

Capítulo 2.- Antecedentes

2.1 Planteamiento del problema

El problema radica en que el software legado manifiesta modularidad incorrecta debido a un bajo grado de protección modular. En el modelo de programación orientada a objetos, esta situación se debe a que muchas o todas las funciones de clases de objetos son declaradas indiscriminadamente con el calificador de alcance "*public*", sin tomar en cuenta las distintas reglas de visibilidad que soportan los lenguajes de programación. Este hecho viola el principio de ocultamiento de información, produciendo un incorrecto nivel de encapsulamiento, permitiendo el acceso a los detalles internos de representación de los objetos desde cualquier agente externo, lo cual podría derivar en estados inválidos o inconsistentes de objetos.

2.2 Solución propuesta

La solución que se propuso en esta tesis consiste en desarrollar un método y su implementación para refactorizar software legado escrito en lenguaje Java, que exhibe el problema planteado. El método propuesto plantea un pre-procesamiento del código legado para cambiar el calificador de alcance "*public*" en aquellas funciones que no son puntos de entrada a servicios, capacidades o requerimientos específicos del dominio de aplicaciones, por el correspondiente calificador de alcance "*private*", "*protected*" o "*friendly*", de conformidad con las reglas de visibilidad del lenguaje Java. El método incluye un proceso de análisis estático, conducido por la "*protección modular"* y el *"principio de ocultamiento de información"*, que localiza código incorrecto en este sentido y estima la magnitud del problema planteado. El método también incluye un proceso de generación de código refactorizado con las reglas correctas de ocultamiento de información, para conseguir mejores condiciones de modularidad y encapsulamiento de objetos.

El método desarrollado se implementó en un sistema de software para automatizarlo. La forma de estimar el problema de ocultamiento de información es mediante el conjunto de métricas desarrolladas en este trabajo de tesis.

El usuario solicita el servicio de refactorización por medio de un menú de opciones, selecciona la carpeta de ubicación del código legado que requiere de la refactorización, el cual deberá estar escrito en lenguaje Java. A continuación, el servicio mide el grado de protección modular *(PM)* y a petición del usuario puede corregir la deficiencia de ocultamiento de información con el calificador de alcance correcto en las funcionalidades de cada clase de objetos. Una vez refactorizado el código legado se mide nuevamente el grado de protección modular *(PM)* de la arquitectura de clases resultante y alerta al usuario sobre el grado de mejora en esta dimensión.

2.3 Justificación

Cuando una unidad de programa carece de protección modular, permite que otras unidades de programa externas accedan a sus detalles internos de representación, lo cual produce un acoplamiento directo y en consecuencia una relación de dependencia. Las relaciones de dependencia generan fragilidad en el sistema. La fragilidad se manifiesta cuando cambios o fallas

en una unidad de programa propagan fallas inesperadas en otras unidades de programa que aparentemente no tienen relación alguna. En esta tesis una unidad de programa se refiere a una clase de objetos o a un conjunto de clases que implementan un módulo o un componente de software cuya meta de valor es la satisfacción de una capacidad o requerimiento del sistema. Un método satisface la protección modular cuando éste resulta en arquitecturas en las que los efectos anormales que ocurren en tiempo de ejecución de una unidad de programa se confinan únicamente a esa unidad y no se propaga a otras unidades externas.

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 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 declarar las funciones que son puntos de entrada a una petición de servicio, tales como las interfaces, con el calificador de alcance *"public"* y el resto de funciones que complementan la petición de servicio declararlas con el calificador "*private"*, "*protected"* o "*friendly",* según la visibilidad que corresponda para completar una secuencia interactiva que corresponde a una meta de valor para el usuario.

En la literatura revisada se advierte que no se reporta la existencia de métricas para medir el grado de protección modular. En este proyecto de tesis se incluye el diseño e implementación de cinco métricas para la medición del nivel de protección modular de los servicios de software obtenidos desde el software legado. El diseño de las cinco métricas está sustentado en la teoría de la medición como se describe en la sección de idea de solución de esta tesis.

El proyecto de tesis desarrollado aporta una extensión a la herramienta SR2-Refactoring. En su estado actual, la herramienta SR2-Refactoring no cubre la refactorización para equilibrar las relaciones entre clases, ni la refactorización 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. Esta tesis aporta en este sentido, midiendo, evaluando y refactorizando las arquitecturas de software para atender el problema de protección modular y encapsulamiento de objetos. El método desarrollado considera que el software legado que se refactoriza, posee cierto comportamiento funcional, el cual se conserva después del proceso de la refactorización.

2.4 Objetivo

2.2.1.- Objetivo General

El objetivo general de este trabajo de tesis es mejorar el diseño de arquitecturas de software legado orientado a objetos, a través de la mejora de la protección modular mediante el ocultamiento de información, utilizando las reglas de visibilidad que soporta el lenguaje de programación Java.

2.4.2 Objetivos específicos

- Mejorar la Modularidad del Software Legado escrito en lenguaje Java

- Mejorar el encapsulamiento de objetos del Software Legado escrito en lenguaje Java
- Facilitar el mantenimiento del Software Legado escrito en lenguaje Java
- Habilitar el reuso de componentes del Software Legado escrito en lenguaje Java
- Extender la funcionalidad del "SR2-Refactoring", para dar soporte a sus métodos de refactorización de alto impacto en código escrito en lenguaje Java.

2.5 Estado del Arte

Anterior a este trabajo, en el CENIDET se han desarrollado proyectos de reingeniería con el propósito de mejorar las arquitecturas de software escritos en lenguaje C++ y Java, los cuales se describen detalladamente a continuación.

2.5.1 SR2-Refactoring

El SR2-Refactoring (Sistema de Reingeniería de Software Legado para Reuso) es la herramienta principal de antecedente en la cual se encuentran implementados varios métodos de refactorización de alto impacto, que en conjunto permiten mejorar la arquitectura de sistemas legados de software existentes escritos en lenguaje Java y C++. El sistema SR2-Refactoring fue implementado en lenguaje Java, utilizando el ambiente Eclipse y, como soporte, utiliza el manejador de Base de Datos MySQL.

Actualmente, el SR2-Refactoring cuenta con dos métodos implementados para la refactorización de software legado escrito en lenguaje Java, los cuales son: 1) "Método de Separación de Interfaces" y 2) "Método de Reducción de Herencia de Implementación". Con el desarrollo de esta tesis se extiende al SR2-Refactoring con un tercer método de refactorización al cual denominamos "Método de Refactorización para la Mejora de la Protección Modular". El sistema SR2-Refactoring implementa métricas para medir el grado de código desagradable (*smell code*) que presenta el Software Legado para lo cual se ejecutan los métodos de refactorización. Como un producto derivado de esta tesis se construyó el marco orientado a objetos denominado "Marco de Medición de Factores de calidad del Software orientado a objetos". Este marco implementa una diversidad de métricas de calidad de Software Orientado a Objetos, de tal manera que permite utilizar los servicios de medición, tanto en los métodos de refactorización del SR2-Refactoring como en otros sistemas de software que los requieran.

2.5.2 "Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces" (Valdés, 2004)

En ese proyecto de investigación se desarrolló un método refactorización denominado *"Separación de Interfaces"*, cuyo algoritmo fue implementado satisfactoriamente en la herramienta SR2 Refactoring. Este método realiza la refactorización de manera automática de código de marcos orientados a objetos escritos en C++, con el propósito de reducir el número de dependencias de herencia de interfaz entre clases debido a la implementación nula de las interfaces no utilizadas en clases derivadas, lo cual viola al principio de sustitución de Barbara Liskov al debilitarse las postcondiciones en las funciones de las clases derivadas. En ese proyecto también

se diseñó e implementó la métrica orientada a objetos denominada "V-DINO" para medir el grado de dependencia debido a interfaces que no se ocupan.

2.5.3 "Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator" (Cárdenas, 2004)

En ese trabajo de investigación se plantea un método de refactorización para reducir el acoplamiento entre clases, debido a muchas relaciones de asociación y/o dependencia, lo que se traduce en muchos canales de comunicación entre clases en sistemas de Software existente escrito en lenguaje C++. Este método de refactorización es conducido según la intención del patrón de diseño *"Mediator",* el cual incorpora en la arquitectura resultante la estructura de clases como se describe en la solución de este patrón de diseño en su solución. La refactorización del método funciona automáticamente y fue incorporado a la herramienta SR2 Refactoring. En ese proyecto también se diseñó e implementó la métrica orientada a objetos que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF).

2.5.4 "Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter" (Santos, 2005)

En ese proyecto se desarrolló un algoritmo de adaptación de interfaces entre código cliente y código servidor. 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++.

2.5.5 "Método de Re-factorización de código java con interfaces y abstracciones incorrectas" (Padilla, 2019)

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

2.5.2 "Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces" (Valdés, 2004)

En ese proyecto de investigación se desarrolló un método refactorización denominado *"Separación de Interfaces"*, cuyo algoritmo fue implementado satisfactoriamente en la herramienta SR2 Refactoring. Este método realiza la refactorización de manera automática de código de marcos orientados a objetos escritos en C++, con el propósito de reducir el número de dependencias de herencia de interfaz entre clases debido a la implementación nula de las interfaces no utilizadas en clases derivadas, lo cual viola al principio de sustitución de Barbara Liskov al

debilitarse las postcondiciones en las funciones de las clases derivadas. En ese proyecto también se diseñó e implementó la métrica orientada a objetos denominada "V-DINO" para medir el grado de dependencia debido a interfaces que no se ocupan.

2.5.3 "Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator" (Cárdenas, 2004)

En ese trabajo de investigación se plantea un método de refactorización para reducir el acoplamiento entre clases, debido a muchas relaciones de asociación y/o dependencia, lo que se traduce en muchos canales de comunicación entre clases en sistemas de Software existente escrito en lenguaje C++. Este método de refactorización es conducido según la intención del patrón de diseño *"Mediator",* el cual incorpora en la arquitectura resultante la estructura de clases como se describe en la solución de este patrón de diseño en su solución. La refactorización del método funciona automáticamente y fue incorporado a la herramienta SR2 Refactoring. En ese proyecto también se diseñó e implementó la métrica orientada a objetos que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF).

2.5.4 "Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter" (Santos, 2005)

En ese proyecto se desarrolló un algoritmo de adaptación de interfaces entre código cliente y código servidor. 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++.

2.5.5 "Método de Re-factorización de código java con interfaces y abstracciones incorrectas" (Padilla, 2019)

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

2.5.6 "Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos" (Ortiz, 2020)

En ese trabajo de tesis se diseñó e implementó un método de refactorización que disminuye el acoplamiento por herencia de implementación en arquitecturas de clases orientadas a objetos de sistemas de software existentes y que están escritos en lenguaje Java. El propósito del método es reducir la interdependencia entre objetos y clases por herencia de implementación. El método

incluye cinco métricas de calidad, tres de ellas miden el factor de herencia de implementación y las dos restantes miden el factor de flexibilidad de aplicaciones existentes orientadas a objetos.

Finalmente, el sistema SR2-Refactoring cuenta con un menú, para realizar las acciones de refactorización y cálculo de métricas, así como acciones adicionales, como son la selección y comparación de archivos y el manejo de usuarios.

La mayoría de las opciones de los menús llevan a pantallas o cuadros de diálogo, y cada uno de ellos cuenta con elementos gráficos, como botones y cuadros de texto.

2.6 Trabajos Relacionados

En el desarrollo de esta tesis se realizó una revisión de nueve trabajos de investigación, los cuales presentan diferentes métodos de refactorización con fines similares al objetivo de esta tesis que consiste en mejorar el diseño de la arquitectura de software de un sistema.

"Refactoring Opportunity Identification Methodology for Removing Long Method Smells and Improving Code Analyzability" (Meananeatra, Rongviriyapanish, & Apiwattanapong, 2018)

El propósito de esa investigación es proporcionar un enfoque efectivo para identificar oportunidades de refactorización y sugerir un conjunto efectivo de éstos para eliminar completamente los métodos grandes, sin reducir la capacidad de análisis de código. Este enfoque fue denominado LMR (Large Methods Removing), el cual utiliza condiciones que habilitan la refactorización basadas en el análisis del programa y las métricas de código, con el objetivo de identificar técnicas de refactorización.

LMR utiliza dos criterios: nivel de capacidad de análisis de código y el número de estatutos de código a impactar por las refactorizaciones. LMR utiliza el análisis de efectos secundarios para asegurar la preservación del comportamiento del programa original. LMR fue integrado al paquete principal de una aplicación en java con el fin de evaluar su funcionamiento, los resultados obtenidos mostraron que los métodos que se aplican sugieren conjuntos de refactorización que pueden eliminar completamente los indeseables de métodos grandes, conservando el comportamiento original y la capacidad de análisis de código.

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

El objetivo de esa investigación es presentar un marco de refactorización de modelos impulsada por la detección y eliminación de anti-patrones que afectan al rendimiento. El marco fue implementado bajo la plataforma EPSILON. Este marco permite inspeccionar múltiples caminos y propone una gran variedad de soluciones, probando, en base los resultados obtenidos, que la automatización de la refactorización del modelo de software basado en el rendimiento puede ser beneficioso.

"Automated Refactoring of Legacy Java Software to Default Methods" (Khatchadourian & Masuhara, 2017)

Ese trabajo de investigación tiene el propósito de proporcionar un enfoque de refactorización basado en restricciones para ayudar al desarrollador a aprovechar las interfaces mejoradas para java, permitiendo que las clases hereden múltiples definiciones de las interfaces, utilizando el método de refactorización *"pull up"* y el patrón *"Skeletal Implementation",* el cual es un patrón de diseño de software que consiste en definir una clase abstracta que proporcionan implementaciones parciales de las interfaces.

Este enfoque se implementó como un plugin de Eclipse IDE y fue aplicado a 19 proyectos de código abierto para evaluar su funcionamiento y efectividad, obteniendo como resultado que el 19.63% de los métodos implementan el patrón de diseño "*Skeletal Implementation".* Estos métodos fueron refactorizados con mínima intervención del usuario a pesar de su conservadurismo y limitaciones del lenguaje.

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

En ese trabajo de investigación se describe la herramienta llamada "*MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE"* la cual tiene un enfoque de refactorización automático que transforma el código heredado escrito en lenguaje Java, para que pueda ser utilizado en construcciones de métodos default (métodos por definición). Los métodos default permiten cambiar el comportamiento de las funcionalidades de las interfaces de las librerías y se garantiza la compatibilidad binaria del código refactorizado con el código correspondiente escrito en versiones anteriores de esas interfaces. Esta herramienta fue implementada como complemento del entorno de desarrollo Eclipse. El código refactorizado que proporciona esta herramienta resulta, semánticamente, equivalente al original, es más conciso, más fácil de Comprender, menos complejo, y exhibe mayor modularidad. Esta herramienta es comparada con la herramienta PULL UP METHOD en donde ésta busca reducir el código fuente, a diferencia de la herramienta MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE que permite que las clases hereden múltiples definiciones de la interfaz. Esta herramienta fue implementada como un código libre para eclipse. Actualmente se explora minuciosamente la relación de la herramienta con la implementación de la herramienta de refactorización *"PULL UP MEMBER"*.

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

El objetivo de ese trabajo de investigación es proponer una nueva métrica de cohesión para software orientado a objetos, denominada "*Usage Pattern Based Cohesion" (UPBC)*. En el contexto de ese trabajo, un módulo es un grupo de clases, del cual es deseable mejorar su cohesión en general. La métrica utiliza el patrón de uso frecuente *(FUP),* y es calculada desde la interacción de diferentes funciones miembro. El valor de cohesión es utilizado para realizar el agrupamiento de módulos para aumentar la cohesión y disminuir el acoplamiento entre éstos. Dicha agrupación utiliza el nuevo algoritmo de agrupación llamado *"FUPClust" (Frequent Usage Pattern based Clustering)* basado en las interacciones *FUP* entre los módulos.

"***Predicting Move Method Refactoring Opportunities in Object-Oriented Code" (Dallal, 2017)***
En esa investigación se presenta una nueva métrica y un modelo para predecir con precisión si una clase incluye métodos que necesitan el método MMR (Move Method Refactoring) para reubicar métodos en otras clases, con el objetivo de obtener mayor cohesión y menor acoplamiento de clases.

La métrica propuesta considera los aspectos de cohesión y acoplamiento de las clases. Además, utiliza datos estructurales y semánticos disponibles dentro de la clase de interés. Se mueve un método de una clase a la clase en la que es utilizado con mayor frecuencia. Las técnicas que se aplican tienen varias limitaciones, como problemas de escalabilidad y no son aplicables en las primeras etapas de desarrollo. Es por esto que la métrica es aplicable durante la etapa de desarrollo. El modelo utiliza datos estructurales y semánticos que se encuentran dentro de la clase. Los modelos se aplicaron en siete sistemas orientados a objetos para evaluar empíricamente sus habilidades para predecir oportunidades de refactorización de este tipo. Como resultado, se obtuvo que la métrica es capaz de detectar correctamente más del 90% de los métodos que necesitan reubicarse dentro de las clases predichas.

"Automated refactoring to the NULL OBJECT design pattern" (Gaitani, Zafeiris, Diamantidis, & Giakoumakis, 2015)

En ese trabajo se pretende obtener un método novedoso para llevar a cabo la refactorización automática de código que implementa referencias a no deseados objetos nulos, a través de la aplicación del patrón de diseño **"***Null Object***".** El patrón de diseño *"null object"* simplifica el uso de dependencias que pueden ser no definidas. Esto se efectúa a través del uso de instancias de clases concretas que implementan las interfaces conocidas en lugar de referencias nulas. Se crea una clase abstracta que especifica varias operaciones a ser realizadas y clases concretas que extienden a esta clase y una clase *"null object"*, que proporciona una implementación vacía de esta clase y será usada correctamente donde sea necesaria la verificación de valores nulos. La refactorización identifica y elimina objetos nulos de clases optativas.

Para lograr el objetivo se introdujo un algoritmo para obtener las oportunidades de refactorización de objetos nulos, es decir, se busca la existencia de objetos con las características necesarias para llevar a cabo la refactorización. Dicho método se implementa como un complemento de Eclipse y es evaluado en un conjunto de proyectos de código abierto de Java. Como resultado de la ejecución exitosa se contribuye a la mejora de la complejidad de arquitecturas de clases.

"Automated refactoring to the Strategy design pattern" (Christopoulou, Giakoumakis, Zafeiris, & Soukara, 2012)

El objetivo de esa investigación es obtener un método para la identificación automatizada de oportunidades para refactorizar código no deseado que implementa condicionales cortas con profundo nivel de anidamiento, hacia arquitecturas que aplican el patrón de diseño "*Strategy"*, buscando reducir estructuras rígidas de código que impiden su reuso y dificultan su mantenimiento. Para los casos especiales de estas condicionales múltiples, se propone una técnica para el llevar a cabo el remplazo total de estas condicionales. El método fue implementado en el complemento de JDeodorant Eclipse, y fue probado en un conjunto de proyectos en java,

obteniendo como resultado una respuesta satisfactoria, ya que el método propuesto contribuyó a la simplificación de las declaraciones de las condicionales mejorando la extensibilidad través del patrón de diseño "*Strategy"* y confirmó la eficiencia en tiempo de ejecución de este método.

"Constructing Models for Predicting Extract Subclass Refactoring Opportunities Using Object-Oriented Quality Metrics" (Dallal, 2012)

Ese trabajo tiene como objetivo explorar y probar la calidad de las métricas de: acoplamiento, cohesión y tamaño, tanto en lo individual como combinándolas unas con otras, con el objeto de predecir la necesidad de que una clase con bajos niveles de cohesión y/o alto nivel de acoplamiento y/o de grande tamaño, necesitan el método ESR (*Extract Subclass Refactoring*) para dividirla extrayendo subclases. Se utiliza un análisis de regresión lógica invariable, para identificar clases que ameritan un proceso de refactorización. El método está planeado para la fase de mantenimiento, además este método es capaz de clasificar las clases del sistema de acuerdo con su grado de necesidad del método ESR. El análisis mostró una fuerte relación entre los atributos de calidad interna de una clase y su necesidad de ESR.

Tabla 1.- Comparativa de trabajos relacionados

Trabajo de investigación	Objetivo	Producto resultante	Tipo de proceso	Métricas usadas	Alcance
(Valdés, 2004)	Eliminar el código desagradable (smell code) de método largo	plugin	Semi-automático	Comportamiento	Implementado
(Arcelli et al., 2018)	Eliminar el código desagradable (smell code) anti patrones de rendimiento	Herramienta	Automático	No utiliza	Implementado
(Khatchadourian & Masuhara, 2017)	Eliminar el código desagradable (smell code) de código redundante para la mejora de interfaces.	Plugin de Eclipse IDE	Semi-automático	Comportamiento	Implementado
(Khatchadourian, 2017)	Reducir los problemas de herencia de implementación múltiple	Herramienta	Semi-automático	Modularidad	Implementado
(Rathee & Chhabra, 2017)	Proponer una nueva métrica de cohesión para software orientado a objetos, denominada "*Usage Pattern Based Cohesion" (UPBC)*	Algoritmo y métrica de cohesión	-	cohesión	Identificado
(Dallal, 2017)	identificar si uno de los métodos en una clase requiere moverse a otra clase, en base a su uso	Modelo y métrica	-	cohesión y acoplamiento	identificado

	Eliminar el código desagradable (smell code) de referencias nulas	Método	Automático	Modularidad	Implementado
" (Gaitani et al., 2015)	Eliminar el código desagradable de condicionales cortas con profundo nivel de anidamiento	Plugin	Automático	Modularidad	Implementado
(Christopoulou et al., 2012)	Predecir la necesidad de que una clase con bajos niveles de cohesión y/o alto nivel de acoplamiento y/o de grande tamaño	Método	Semi-automático	Acoplamiento, cohesión y tamaño	Implementado
(Dallal, 2012)	Predecir la necesidad de que una clase con bajos niveles de cohesión y/o alto nivel de acoplamiento y/o de grande tamaño, necesitan el método ESR (*Extract Subclass Refactoring*) para dividirla extrayendo subclases.	Método ESR (*Extract Subclass Refactoring*)	Automático	Tamaño, Cohesión y Acoplamiento	Implementado
Tesis	Mejorar el diseño de arquitecturas de software legado orientado a objetos, a través de la mejora de la protección modular mediante el ocultamiento de información, utilizando las reglas de visibilidad que soporta el lenguaje de programación Java.	Método (Extensión de la herramienta SR2-Refactoring)	Semi-automático	Protección modular	Implementado

Capítulo 3.- Marco Teórico

3.1 Paradigma de programación orientada a objetos

3.1.1 Clase

Una clase es una plantilla que permite definir un conjunto de objetos. Por ejemplo, un automóvil es una clase de objetos caracterizados por tener motor, cuatro llantas, etc. Un Ibiza o un Jetta son objetos particulares que se obtienen instanciando la clase automóvil para esos casos particulares (Duran Muñoz Francisco, 2017). **Esencialmente, una clase es un plan que especifica cómo construir objetos,** en ella se definen los atributos, que denotan el estado implícito y el estado explícito; los métodos que denotan el comportamiento con los cuales los objetos responden a mensajes externos; así mismo y mensajes comunes a todos los objetos que son instancias de la clase o tipo.

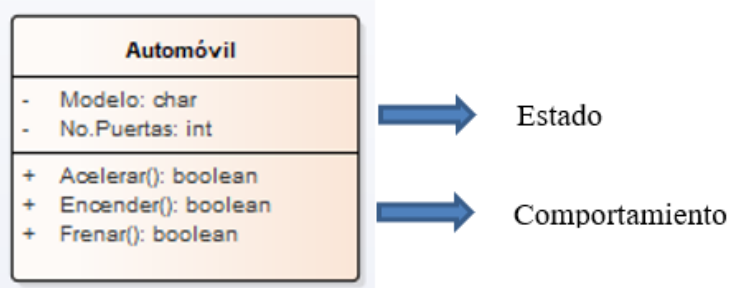


Figura 1.-Modelo conceptual de una clase

3.1.2 Objeto

Es cualquier cosa tangible o intangible que se pueda imaginar, definida frente al exterior mediante unos atributos y las operaciones que permiten modificar dichos atributos. Por ejemplo, el motor de un automóvil o una cuenta bancaria, son ejemplos de objetos, pero también lo es una estructura de datos (pila o lista). Cada objeto particular se obtiene como la especificación de una entidad más general, denominada clase (Duran Muñoz Francisco, 2017). Un objeto tiene identidad propia, encapsula su propio estado y su propio comportamiento.

3.1.3 Representación interna de clases de objetos

3.1.3.1 Atributos de clases

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos representan variables de instancia, y cada objeto particular puede tener valores distintos para estas variables lo cual es conocido como el estado del objeto (Cervantes, 2016).

El estado de un objeto puede ser implícito o explícito. El estado implícito lo definen los valores que asumen cada uno de los atributos en cierto momento. El estado explícito lo definen variables que, explícitamente, denotan un cierto estado del objeto a través de la ejecución de uno o varios casos de uso.

3.1.3.2 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.

3.1.3.3 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 acabo 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.

En la Figura 2 se puede observar un ejemplo de envío de un mensaje, en donde un objeto de clase Cliente solicita el servicio del objeto "Op" de clase Operaciones para realizar la suma de los números 5 y 6.

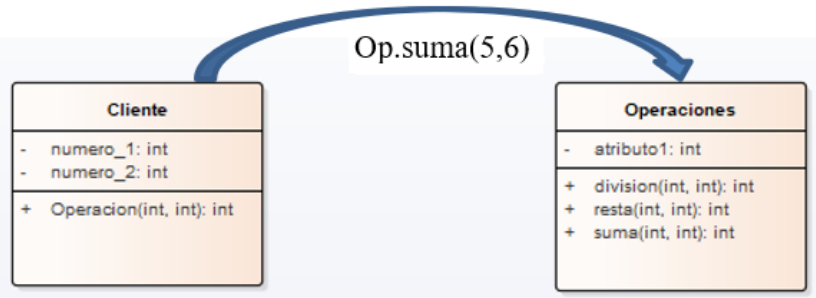


Figura 2.- Envió de mensaje en donde un objeto de clase Cliente solicita el servicio del objeto "Op" de clase Operaciones para realizar la suma de los números 5 y 6.

3.2 Modularidad y Encapsulamiento

3.2.1 Modularidad

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas llamadas módulos, los cuales deben ser tan independiente 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 de desee declararlas como públicas para permitir su manipulación desde el exterior del módulo.

3.2.2 Encapsulamiento

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 o calificadores de alcance) (Llinás, 2010) y con el principio de ocultamiento de información. El encapsulamiento permite empaquetar la funcionalidad de un objeto, de forma que se pueda cambiar la funcionalidad interna sin afectar a la visión externa de un componente de un sistema (Duran Muñoz Francisco, 2017).

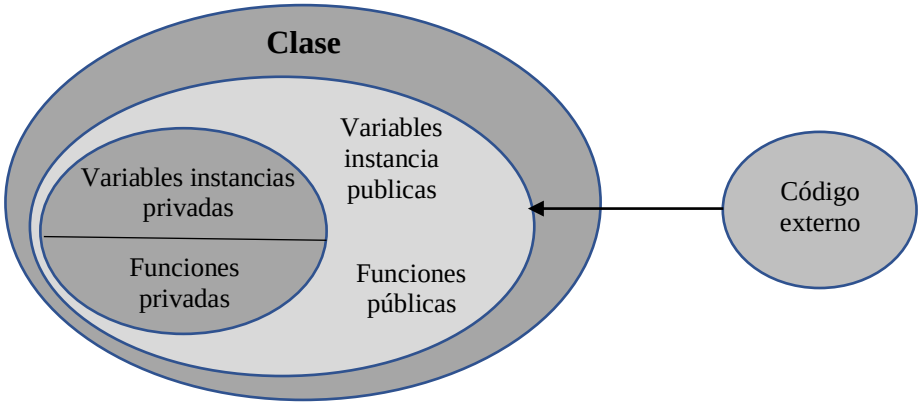


Figura 3.- Encapsulamiento con miembros públicos y privados tomada de (López, 2021)

3.3 Principio de ocultamiento de información

Este principio consiste en no mostrar al exterior los datos o las funciones que necesitan ser protegidos del exterior. Un módulo bien encapsulado sólo exhibe la información necesaria, lo cual protege los detalles internos de representación de una manipulación externa arbitraria (Duran Muñoz Francisco, 2017).

3.3.1 Protección modular

Para propósitos de este trabajo se entiende por protección modular al detalle que se define en el protocolo de la clase para regular la visibilidad de la información de los objetos por entidades externas, de manera que se evite la propagación de defectos o cambios de un módulo hacia otro modulo lo cual puede incrementar la fragilidad del sistema.

Una aplicación que carece de protección modular presenta fragilidad. Fragilidad significa que partes inesperadas del sistema dejan de funcionar correctamente, al presentarse un cambio, un error o un defecto en algún módulo. La protección modular se establece a través de las reglas de ocultamiento de información que soporta los diferentes lenguajes de programación. El lenguaje Java soporta cuatro niveles de ocultamiento de información que protegen a las clases de objetos de la manipulación externa de sus detalles de representación interna. Estos cuatro niveles de ocultamiento de información aplican tanto para datos como para funciones de clases de objetos. Estos niveles de ocultamiento de lenguaje Java son: el calificador de alcance "*public*", el calificador de alcance "*protected*", el calificador de alcance "*friendly*" y el calificador de alcance "*private*".

Aplicar protección modular significa evitar que un error o defecto en un módulo producido en tiempo de ejecución se propague hacia otros módulos relacionados que aparentemente trabajaban bien. Así mismo, que los cambios a los requerimientos iniciales o ante nuevos requerimientos en un módulo, sólo se confinen al módulo en cuestión y no se propaguen estos cambios a otros módulos relacionados.

3.3.2 Protocolo de la clase para la creación de instancias

La protección se logra a través del control de acceso de los detalles internos de representación que se define en el protocolo de la clase para el encapsulamiento de instancias u objetos. Esto significa utilizar correctamente los calificadores de alcance tanto de los atributos como de las funciones de la clase. Las interfaces deben ser publicas puesto que es a través de éstas que se ofrecen los servicios hacia el exterior; mientras que, al proteger los elementos privados, cambios en el módulo sólo afectarán a esos elementos privados y no a la interfaz, por lo tanto, los módulos clientes no serán afectados.

3.3.3 Calificadores de alcance

Se utilizan para definir la visibilidad de los miembros de una clase (atributos y funciones) y de la propia clase. El lenguaje Java define cuatro calificadores de acceso:

- Public (publico): De libre acceso. Los detalles internos de representación que han sido calificados como públicos, permiten el acceso desde cualquier objeto de cualquier clase del sistema. Este calificador de alcance no ofrece ninguna protección de los detalles internos de los objetos.

- Protected (Protegido): De acceso protegido. Este calificador de alcance permite el acceso a los detalles internos de un objeto solo a los miembros de objetos de clases derivadas, o a los miembros del propio objeto. La protección de los detalles internos del objeto consiste

en la restricción de acceso a otros objetos que no se encuentren en la jerarquía de herencia de clases.

- Private (privado): De acceso privado. Este mecanismo de protección permite el acceso de los detalles internos de representación de un objeto, sólo a los propios elementos del objeto. Este calificador de alcance es el más restrictivo.

Cuando no se especifica ninguno de los tres calificadores anteriores se tiene el nivel de acceso por defecto Friendly (amistosas).

- Friendly (amistoso). Por defecto o sin especificador de acceso predeterminado. Este nivel de acceso opera de manera similar al nivel de acceso público, sólo que éste aplica a nivel de paquete. Es decir, las entidades friendly permiten su acceso desde cualquier objeto de cualquier clase, siempre y cuando estén definidos en el ámbito del mismo paquete.

A continuación, se muestra el acceso permitido para cada calificador de alcance:

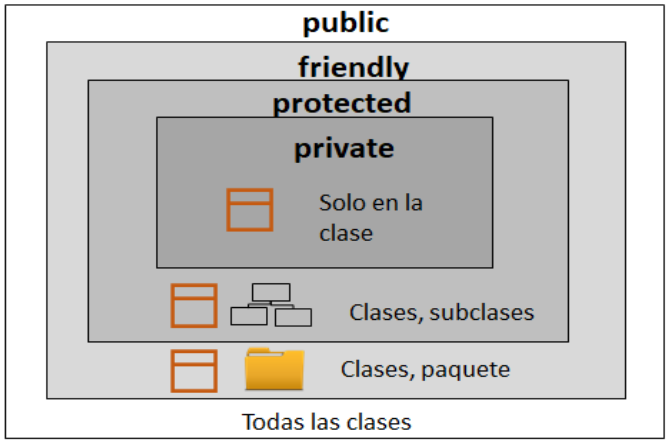


Figura 4.- Representación del acceso de los calificadores de alcance

3.4 Software legado y Marcos orientados a objetos

3.4.1 Software legado (Pressman & Ph, n.d.)

Algunos de ellos son software muy nuevo, disponible para ciertos individuos, industria y gobierno. Pero otros programas son más viejos, y en ciertos casos muy viejos. Estos programas antiguos son frecuentemente denominados como software heredado o software legado. Dayani-Fard y sus colegas describen el software heredado de la manera siguiente:

Los sistemas de software heredado fueron desarrollados hace varias décadas y han sido modificados de manera continua para que satisfagan los cambios en los requerimientos de los negocios y plataformas de computación. La proliferación de tales sistemas es causa de dolores de cabeza para las organizaciones grandes, a las que resulta costoso mantenerlos y riesgoso hacerlos evolucionar. Una característica que se puede hacer presente en el software heredado es la *mala calidad.*

Los sistemas de software evolucionan por una o varias de las siguientes razones:

- El software debe adaptarse para que cumpla las necesidades de los nuevos ambientes del cómputo y de la tecnología.
- El software debe ser mejorado para implementar nuevos requerimientos del negocio.
- El software debe ampliarse para que sea operable con otros sistemas o bases de datos modernos.
- La arquitectura del software debe rediseñarse para hacerla viable dentro de un ambiente de redes.

3.4.2 Marcos orientados a objetos (framework)

Un marco orientado a objetos (conocidos como frameworks en la literatura escrita en lenguaje inglés) se define como un conjunto semi-completo de clases en colaboración que incorpora un diseño genérico, el cual puede adaptarse a una variedad de problemas específicos para producir nuevas aplicaciones hechas a la medida (René, 2003). En general, los frameworks son construidos con base en lenguajes orientados a objetos. Esto permite una mejor modularización de los componentes y su óptimo reuso del código (Alberto, 2010).

El uso de frameworks para cualquier tipo de desarrollo reduce el tiempo de elaboración e implementación y ayuda a hacer un trabajo mantenible y escalable, según las características del mismo.

Un framework agrega funcionalidad extendida a un lenguaje de programación, automatiza muchos de los patrones de programación para orientarlos a un determinado propósito, proporcionando una estructura al código, mejorándolo y haciéndolo más entendible y sostenible, y permite separar en capas la aplicación. En general, divide la aplicación en tres capas:

- La lógica de presentación que administra las interacciones entre el usuario y el software.
- La lógica de datos que permite el acceso a un agente de almacenamiento persistente u otros.
- La lógica de dominio o de negocio, que manipula los modelos de datos de acuerdo a los comandos recibidos desde la presentación.

3.5 ANTLR (Parr, 2013).

ANTLR (ANother Tool for Language Recognition) es 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 una gramática, ANTLR genera un analizador que puede construir y recorrer árboles de análisis.

ANTLR requiere que se defina un archivo "*lexer*", en el cual se declaran todos los tokens del lenguaje. Para 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. Es más expresiva y permite describir las gramáticas de manera más sencilla.

En este trabajo de tesis la obtención de la información necesaria para implementar la refactorización y el cálculo de las métricas PM, se realizó a través de la clase

JavaBaseParserListener. En esta clase se especifica qué es lo que se debe realizar 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.

3.5.1 StringTemplate

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).

3.6 Teoría de la medición

La medición tiene como objetivo obtener una descripción numérica de los objetos / eventos / personas del mundo real por medio de un sistema de medición. La medición es usada como una forma clave para obtener información de alta calidad del mundo real, en todas las disciplinas (Stevens, 1946).

La medición se define como el proceso de asignación empírica y objetiva de símbolos a los atributos de objetos y eventos del mundo real de manera tal, que estos los represente o describa La teoría de la medición sobre los principios de la representación, proporciona una base para una ciencia de medición de aplicación universal.

La medición es un proceso inherente y consustancial a toda investigación, sea ésta cualitativa o cuantitativa. Principalmente se miden variables, lo que demanda considerar tres elementos básicos: 1) el instrumento de medición, 2) la escala de medición y 3) sistema de unidades de medición. La validez, consistencia y confiabilidad de los datos medidos dependen, en buena parte, de la escala de medición que se adopte. He ahí la importancia de profundizar en el tema de las escalas de medición (Finkelstein, 2009).

Podría decirse también que *medir* es estimar la magnitud de cierta propiedad de uno o más objetos con ayuda de un sistema métrico específico (instrumento de medición, escala de medición y unidades de medición).

3.6.1 Escalas o niveles de medición (coronado, 2007).

Una escala de medición es el conjunto de los posibles valores que una cierta variable puede tomar. Es un continuo de valores ordenados correlativamente, que admite un punto inicial y otro final. El nivel en que una variable puede ser medida determina las propiedades de medición de una variable, el tipo de operaciones matemáticas que puede usarse apropiadamente con dicho nivel, las fórmulas y procedimientos estadísticos que se utilizan para el análisis de datos y la prueba de hipótesis teóricas.

Las escalas o niveles de medición se utilizan para medir variables o atributos. Por lo general, se distinguen cuatro escalas o niveles de medición: nominal, ordinal, intervalos y escalas de proporción, cociente o razón. Las dos primeras (nominal y ordinal) se conocen como escalas categóricas, y las dos últimas (intervalo y razón) como escalas numéricas. Las escalas categóricas

se usan comúnmente para variables cualitativas, mientras que las numéricas son adecuadas para la medición de variables cuantitativas.

Tabla 2.- Ejemplos prácticos de variables y sus escalas de medición

Tipo de variable	Ejemplo de variable	Valores de la variable	Escala
Categóricas o cualitativas	Partido político	Liberal; conservador; independiente; socialista	Nominal
	Género	Mujer; hombre Masculino; femenino	Nominal
	Raza	Negro; blanco; amarillo; mestizo; mulato	Nominal
	Nivel de satisfacción	Alto; medio; bajo	Ordinal
	Calificación en el examen	A; B; C; D; E	Ordinal
Numérica o cuantitativa	Temperatura	0 - 100	Intervalo
	Coeficiente intelectual	70 - 150 puntos	Intervalo
	Peso	1 - 100 Kg	Razón
	Estatura	0 - 2.50 mts	Razón
	0 - 125 años	0 - 125 años	Razón

3.6.1.1 Escala nominal

Es la escala más elemental y la forma más rudimentaria de medir. En una escala como ésta se clasifica a las unidades de estudio (objetos, personas, etc.) en categorías, basándose en una o más características, atributos o propiedades distintivas y observadas, dándole a cada categoría un *nombre* (de ahí lo de «nominal»). Los *nombres* que se emplean en la aplicación de la escala nominal de medida no necesitan ser nombres (alfabéticos o alfanuméricos) en el sentido estricto de la palabra. También se pueden utilizar números o numerales.

3.6.1.2 Escala ordinal

Una escala de medición ordinal se logra cuando las observaciones pueden colocarse en un orden relativo con respecto a la característica que se evalúa, es decir, las categorías de datos están clasificadas u ordenadas de acuerdo con la característica especial que poseen. Aquí, las etiquetas o símbolos de las categorías sí indican jerarquía. Si utilizamos números, la magnitud de estos no es arbitraria, sino que representa el orden del rango del atributo observado. Se supone un continuo subyacente en los números, de modo que las relaciones típicas son, en este caso, «más alto que»,

«mayor que» o «preferible a». Sólo las relaciones «mayor que», «menor que» e «igual a» tienen significado en una escala de medición ordinal.

Para clasificar una métrica como una escala ordinal, debe cumplir los requisitos del **axioma de orden débil**, los cuales son: que •≥ *es* una relación binaria completa y transitiva. Las propiedades de transitividad y completitud son las siguientes (Zuse, 1995):

1. Transitividad: P •≥ P', P' •≥ P" => P •≥ P"
2. Completitud: P •≥ P' o P' •≥ P

Para todo P', P" Ɛ P, donde P es un conjunto y •≥ es una relación empírica binaria como **"igual o más compleja que"**. Supóngase que (P, •≥) es un sistema relaciona1 empírico, donde P es un conjunto contable no vacío y •≥ es una relación binaria en P. Luego existe una función μ: P → ℜ, con: P' •≥ P" ↔ μ(P') ≥ μ(P") para todo P', P" Ɛ P, sí y sólo sí, •≥ es de orden débil. Si tal homomorfismo existe, entonces, ((P, •≥), (ℜ, ≥), μ) es de escala ordinal. La medida μ en una escala es un homomorfismo.

3.6.1.3 Escala de intervalos

Las escalas de intervalo o cardinales son más refinadas puesto que además del orden o jerarquía entre categorías, las etiquetas o números consecutivos establecen intervalos iguales en la medición (las distancias entre categorías son las mismas a lo largo de toda la escala). La medición en una escala de intervalos se basa en suponer que puede conocerse exactamente la diferencia entre los objetos medidos según esta escala. Esto es, debe ser posible asignar un número a cada objeto de modo tal que la diferencia entre los objetos quede reflejada por la diferencia entre los números asignados. Por ejemplo, la diferencia entre los valores 4 y 5 es la misma que entre los valores 1 y 2, o entre 9 y 10. Un cambio unitario en la escala reflejará siempre el mismo cambio en el objeto medido.

3.6.1.4 Escala de proporción o razón

Llamadas también escalas de cocientes. Estas escalas tienen las propiedades de las ordinales y de intervalo (intervalos iguales entre categorías y aplicación de las operaciones aritméticas básicas y sus derivaciones) pero, además, el cero es real, es absoluto, no es arbitrario. Es decir, el cero representa la ausencia de la característica en cuestión; en consecuencia, los números pueden compararse como proporciones y nos permite indicar cuántas veces es más grande un objeto que otro, además de señalar la cantidad en que difieren. En una empresa, una persona que apenas ingresa a laborar, diríamos que la antigüedad de esa persona es de cero años o meses.

3.6.1.5 Propiedades de las escalas de medición

Podemos resaltar como propiedades de las escalas de medición, las siguientes:

Ordinales:

• Las categorías de los datos son mutuamente excluyentes, es decir, un objeto o individuo debe pertenecer a una de las categorías.

• Las categorías de datos están clasificadas u ordenadas de acuerdo con la característica especial que poseen.

De intervalo:

• Las categorías son mutuamente excluyentes y exhaustivas (exhaustiva: un individuo, objeto o medición debe pertenecer a una de las categorías).
• Las categorías están ordenadas de acuerdo con la cantidad de características que posean.
• Diferencias iguales en la característica están representadas por diferencias iguales en los números asignados a las categorías.

De razón:

• Las categorías son mutuamente excluyentes y exhaustivas (exhaustiva: un individuo, objeto o medición debe pertenecer a una de las categorías).
• Las categorías están ordenadas de acuerdo con la cantidad de características que posean.
• Diferencias iguales en la característica están representadas por diferencias iguales en los números asignados a las categorías.
• El punto cero refleja la ausencia de esta característica.

Capítulo 4.- Materiales y métodos de solución

En este trabajo de tesis se realizó una investigación en busca de métricas cuyo propósito es medir el grado de protección modular de una arquitectura de clases orientadas a objetos. En la literatura revisada no se reporta la existencia de métricas para medir el grado de protección modular en ninguno de los niveles de visibilidad. En este trabajo se proponen cinco métricas de calidad, las cuales miden el grado de protección modular de los diferentes niveles de visibilidad: PMFP (Factor de Protección Modular de Funciones Privadas), PMFPR (Protección Modular de Funciones Protegidas), PMFF (Protección Modular de Funciones Friendly), PM (Protección Modular) y TPM (Total Protección Modular). A continuación, se presenta el diseño de cada una de las métricas de calidad PM.

Para propósitos de las métricas PM defienden los siguientes conceptos:

Jerarquía de clases

Una jerarquía de clases es la relación de herencia que existe entre ellas, dicha relación está conformada por una clase base y una clase derivada la cual a su vez puede ejercer o no como clase base para otra clase derivada, y así sucesivamente.

En la figura 5 se muestra un diagrama de clases de la arquitectura de un sistema en donde se resaltan con diferente color las siete jerarquías de clases que conforman dicha arquitectura.

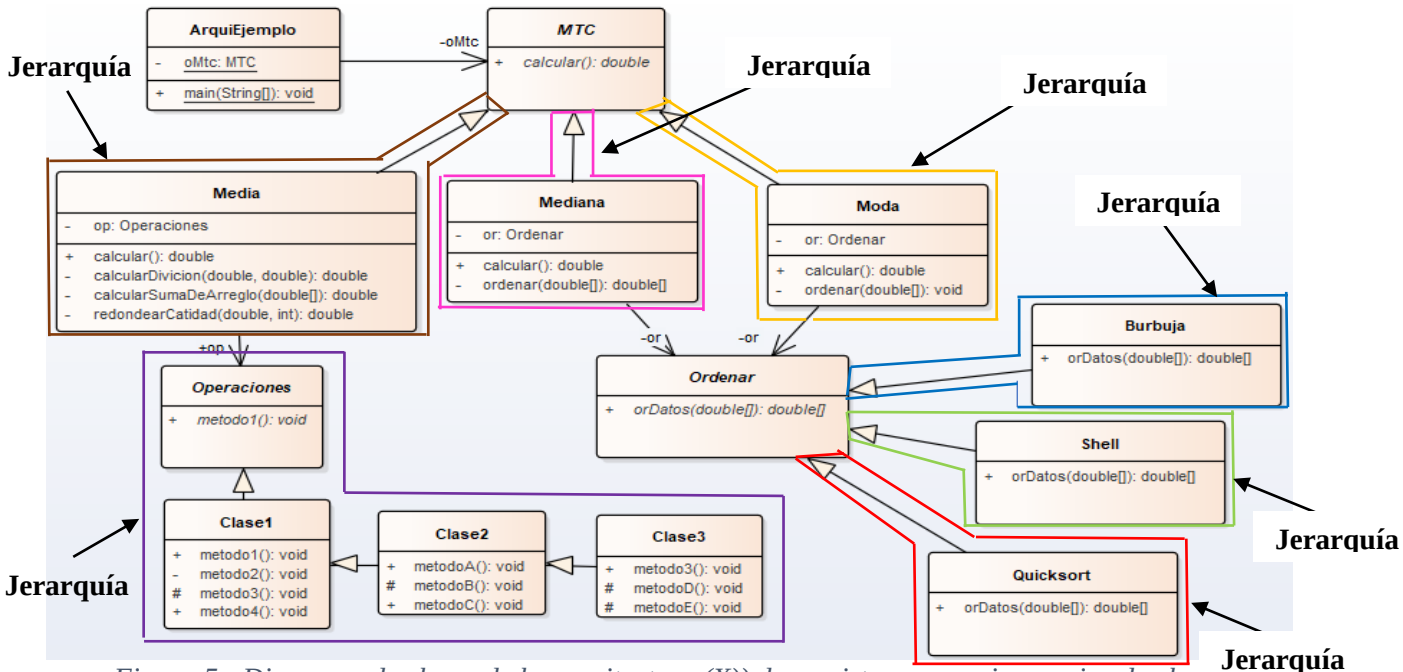


Figura 5.- Diagrama de clases de la arquitectura (X)) de un sistema y sus jerarquías de clases

4.1 Diseño de la métrica PMFP (Factor de Protección Modular de Funciones Privadas)

Se definió la métrica PMFP para medir el grado de protección modular con respecto a las funciones que han sido declaradas con el calificador de alcance "*private".* Esta métrica consiste en detectar en cada una de las clases las funciones que han sido declaradas como "*private" y* realizar una suma de todas estas funciones, entre el número total de funciones, posteriormente realizar una suma de todos los valores obtenidos y dividirlo entre el número total de clases.

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

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

Donde:

FP Funciones con calificador private.
Fi "*i-esima*" función.
FTC Número total de funciones de la clase.
Ci "i-*esima*" clase.
NTC Número total de clases.
PMFP Factor de protección modular por funciones privadas.

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

Una medida que tienda al 0 indica que hay pocas funciones "*private",* por lo que la protección de la información se encuentra con un nivel bajo. El mejor valor será el 1, el peor valor será 0.

4.2 Diseño de la métrica PMFPR (Protección Modular de Funciones Protegidas)

Se definió la métrica PMFPr para medir el grado de protección modular con respecto a las funciones que han sido declaradas con el calificador de alcance "*protected".* Esta métrica consiste en sumar las funciones que han sido declaradas "*protected*" de cada una de las jerarquías entre el número total funciones, posteriormente realizar la suma de todos los valores obtenidos y dividirlo entre el número total de jerarquías

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

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

Donde:

FPr Funciones con calificador "*protected".*
Fi "*i-esima*" función *"protected"* de la jerarquía.

NTF Número total de funciones de la jerarquía.
ji "*i-esima*" jerarquía.
NTJ Número total de jerarquías.

A medida que esta razón tienda al 1 se indicara que hay una tendencia a tener funciones *protected* y por lo tanto se presenta una alta protección de la información.

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

4.3 Diseño de la métrica PMFF (Protección Modular de Funciones Friendly)

Se definió la métrica PMFF para medir el grado de protección modular con respecto a las funciones que han sido declaradas con el calificador de alcance "*friendly"*. Esta métrica consiste en sumar las funciones que han sido declaradas "*friendly*" o son calificador de alcance (*default*) entre el número total funciones.

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

$$\mathrm{PMFF} = \frac{\sum_{i=0}^{i=n} \mathrm{FF}}{\mathrm{NTF}} \quad (3)$$

Donde:

FF Funciones con calificador "*Friendly" o "default".*
i "*i-esima*" función "*Friendly" o "default".*
NTF Número total de funciones.
PMFF Es el nivel de protección modular de funciones "*friendly"*.

A medida que esta razón tienda al 1 se indicara que hay una tendencia a tener funciones "*friendly"* y por lo tanto se presenta una alta protección de la información en cuanto funciones "*friendly"* o *default*.

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

4.4 Diseño de la métrica PM (Protección Modular)

Se ha definido una métrica para medir el grado de protección modular de un software legado. Esta métrica consiste en, la suma de las funciones que han sido declaradas con el calificador de alcance diferente a "*public"*, entre el número total de funciones.

La expresión matemática de la métrica *PM* se muestra a continuación:

$$\mathrm{PM} = \frac{\sum_{i=0}^{i=n} \mathrm{FNP}}{\mathrm{TF}}$$

Donde:

FNP Funciones que no son *"public"*
n Número total de funciones
PM Nivel de protección modular

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

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

4.5 Diseño de la métrica TPM (Total de Protección Modular)

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 de PMFP, PMFPr y PMFF entre tres.

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

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

Donde:

PMFP Grado de protección modular de funciones "*private".*
PMFPr Grado de protección modular de funciones "*protected".*
PMFF Es el grado de protección modular de funciones "*friendly" o "default".*
TPM Es el nivel total de protección modular.

La métrica fue normalizada con el objetivo de que los valores obtenidos se encuentren dentro del rango del 0 al 1. A medida que esta razón tienda a 0 mayor es el problema, indicaría que hay una tendencia a tener muchas funciones públicas que no inician capacidades o requerimientos de la aplicación en evaluación y por lo tanto tienen poca protección modular y un encapsulamiento incorrecto.

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

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

4.6 Método de refactorización de calificadores de alcance.

La Figura 6 muestra el modelo BPMN (Business Process Model and Notation) del proceso general del método de refactorización de calificadores de alcance. Este método de refactorización está conformado por los subprocesos de a) análisis de código de fuente, b) cálculo de métricas PM (de código original y de código refactorizado), c) refactorización y d) generación de código.

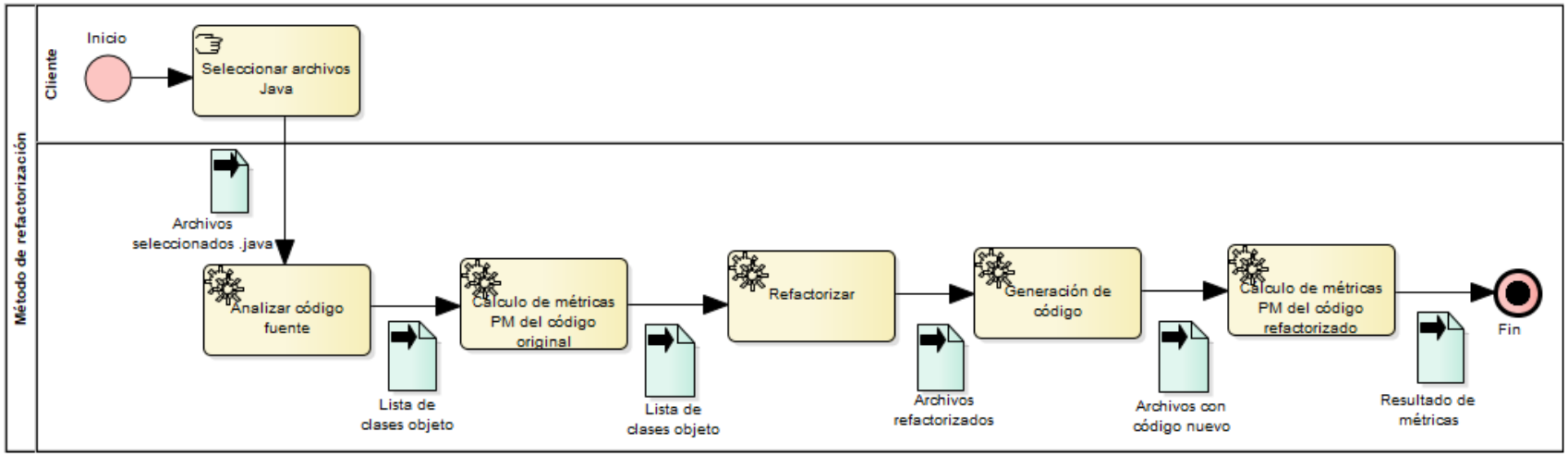


Figura 6.- Diagrama general del proceso de refactorización

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

En el subproceso de cálculo de métricas: Se 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 de refactorización de calificadores de alcance: Este subproceso recibe como entrada la información contenida en la lista compleja que resulta del subproceso de análisis. Con esta información el sistema decide cuál de los calificadores de alcance es el

correcto para cada función de cada clase de objetos en la arquitectura del código original. La decisión para asignar el calificador de alcance correcto se realiza de acuerdo con los siguientes criterios:

a) Todas las funciones localizadas en una interfaz son calificadas como *"public"*

b) Todas las funciones abstractas localizadas en clases abstractas son calificadas como *"public"*

c) La función principal (*main*) es calificada como *"public"*

d) Las funciones utilizadas por funciones de clases de diferente paquete son calificadas como *"public"*

e) Las funciones sobre-escritas que se implementan en clases derivadas heredan el calificador de alcance de la función definida en la clase base.

f) Todas las funciones utilizadas en el alcance de la misma clase son calificadas como *"private"*

g) Todas las funciones utilizadas sólo por otras funciones localizadas en clases del mismo paquete son calificadas como "*friendly*"

h) Todas las funciones implementadas en una clase base, que son utilizadas sólo por funciones de clases derivadas son calificadas como "*protected*"

i) Todas las funciones constructoras son calificadas como "*public*"

j) Todas las funciones implementadas en una clase base, utilizadas por clases derivadas de diferente paquete son calificadas como "*protected*"

k) Todas las funciones plantilla que implementan el patrón de diseño "*Template Method*", en la clase base, deben ser declaradas con el calificador "*public*", mientras que todas las funciones declaradas en la misma clase que implementan el código no variante deben ser declaradas con el calificador "*private*", para asegurar que sólo la función de plantilla las invoque. Las funciones gancho que representan al código variante que son implementadas en las clases derivadas, deben ser declaradas con calificador de alcance "*protected*", para asegurar que sólo la función plantilla los pueda invocar.

l) Reglas de precedencia de calificadores de alcance:

l.1 Funciones invocadas por funciones de cualquier otra clase deben ser calificadas como "*public",* aun cuando sean invocadas

por funciones de otras clases del mismo paquete o invocadas por funciones de clases derivadas e incluso invocadas por funciones en la misma clase.

l.2 Funciones invocadas por funciones de cualquier otra clase del mismo paquete, deben ser calificadas como "*friendly",* aun cuando sean invocadas por funciones de clases derivadas e incluso invocadas por funciones de la misma clase, pero no invocadas por funciones de clases de otros paquetes.

l.3 Funciones que no son invocadas por funciones de cualquier otra clase, o de funciones de clases del mismo paquete, pero que si son invocadas por funciones de clases derivadas deben ser calificadas como "*protected",* aun cuando también sean invocadas por funciones de la misma clase.

l.4 Funciones que no son invocadas por funciones de cualquier otra clase, o de funciones de otras clases del mismo paquete, o por funciones de clases derivadas, pero que si son invocadas por funciones de la misma clase deben ser calificadas como "*private".*

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 calificador de alcance en todas las funciones identificadas, sin alterar el comportamiento de la aplicación original.

A manera de ilustración en el siguiente ejemplo se explica cómo se utiliza una plantilla StringTemplate para el cambio de calificadores de alcance de las funciones de una clase. En el ejemplo se muestra el código de la clase con sus calificadores originales, enseguida se muestra la plantilla para la generación de código refactorizado, por último, se muestra el código resultante equivalente al código original con los calificadores de alcance correctos.

Implementación de la plantilla StringTemplate

La Figura 7 muestra el código original de la clase cCorrelation, la cual forma parte de las clases que conforman la aplicación del Marco estadístico de prueba. Se puede observar que la clase cCorrelation está conformada por dos funciones 1) la función constructora cCorrelation, 2) la función calcula. En base a un análisis manual y conforme a los criterios establecidos para la identificación de los calificadores de alcance correctos, se identifica que la función calcula debe tener un cambio de calificador de alcance: de "*public*" a "*friendly*".

```
1  package statistic;
2  public class cCorrelation extends  aStatistic
3  {
4      public aBasFunc Basicas;
5          //IVM 08/03/2003
6          //quite void, es constructor de la clase
7      public cCorrelation()
8      {
9      }
10     public void calcula(context_ctx ctx)
11     {
12         double sum1;
13         double sum2;
14         double sum3;
15         double sum4;
16         double sum5;
17         Basicas = new cSumXY();
18         Basicas.calcula(ctx);
19         sum1 = ctx.getResultado();
20         Basicas = new cSum();
21         Basicas.calcula(ctx);
22         sum2 = ctx.getSumax();
23         sum3= ctx.getSumay();
24         Basicas = new cSquareSum();
25         Basicas.calcula(ctx);
26         sum4= ctx.getSumax();
27         sum5= ctx.getSumay();
28         double corr=0;
29         corr=(ctx.n*sum1-sum2*sum3)/Math.sqrt((ctx.n*sum4-Math.pow(sum2,2))*(ctx.n*sum5-Math.pow(sum3,2)));
30         ctx.setResultado(corr);
31         }
32 }
```

Figura 7.- Código original de la clase cCorrelation

En la Figura 8 se muestra la plantilla en donde se define la estructura para la generación de una clase en lenguaje Java, en ella se indican las partes que conforman a una clase como son: paquete, importaciones, nombre, atributos, funciones, etc. Así mismo, se define la estructura de las funciones definidas en la clase indicando las partes que la conforman como son: nombre, calificador de alcance, tipo (void, boolean, etc.), parámetros, el tipo de función, etc. En la plantilla también se establecen ciertas condiciones para que el llenado se realice de manera correcta. Por ejemplo la condición <if(clase.abstracta)>abstract, indica colocar la palabra reservada "*abstract*", si la variable del objeto tipo "Clase" es verdadera. La plantilla se llena con el código refactorizado de cada clase sometida a al método de refactorización.

```
package <paquete>;

<clase.importaciones :{ imp | <imp><\n>}>

  public <if(clase.abstracta)>abstract class <endif><if(clase.esInterfaz)>interface <endif><if(!clase.abstracta &

   <clase.variables :{ atributo | <atributo>;}; separator="\n">

   <clase.metodos :{ metodo |<if(metodo.esOverride)>@Override<endif><\n><metodo.calificador> <if(metodo.abstract

}
```

Figura 8.- Fragmento de la plantilla StringTemplate de una clase en lenguaje Java

La Figura 9 muestra el código refactorizado de la clase cCorrelation generado por la plantilla, en donde se puede observar que el único cambio que sufrió el código, fue el cambio del calificador de alcance de la función calcula.

```
1   package statistic;
2     public class cCorrelation extends aStatistic {
3
4       public aBasFunc Basicas;
5
6      public  cCorrelation()
7          {
8          }
9
10        void calcula(context_ctx ctx)
11         {
12          double suml;
13            double sum2;
14            double sum3;
15            double sum4;
16            double sum5;
17            Basicas = new cSumXY();
18            Basicas.calcula(ctx);
19            suml = ctx.getResultado();
20            Basicas = new cSum();
21            Basicas.calcula(ctx);
22            sum2 = ctx.getSumax();
23            sum3= ctx.getSumay();
24            Basicas = new cSquareSum();
25            Basicas.calcula(ctx);
26            sum4= ctx.getSumax();
27            sum5= ctx.getSumay();
28            double corr=0;
29            corr=(ctx.n*suml-sum2*sum3)/Math.sqrt((ctx.n*sum4-Math.pow(sum2,2))*(ctx.n*sum5-Math.pow(sum3,2)));
30            ctx.setResultado(corr);
31         }
32  }
```

Figura 9.- Código refactorizado de la clase "cCorrelation"

En la figura 10 se muestra a detalle los subprocesos que componen al método de refactorización. En cada uno de ellos se pueden observar las actividades que realiza cada subproceso.

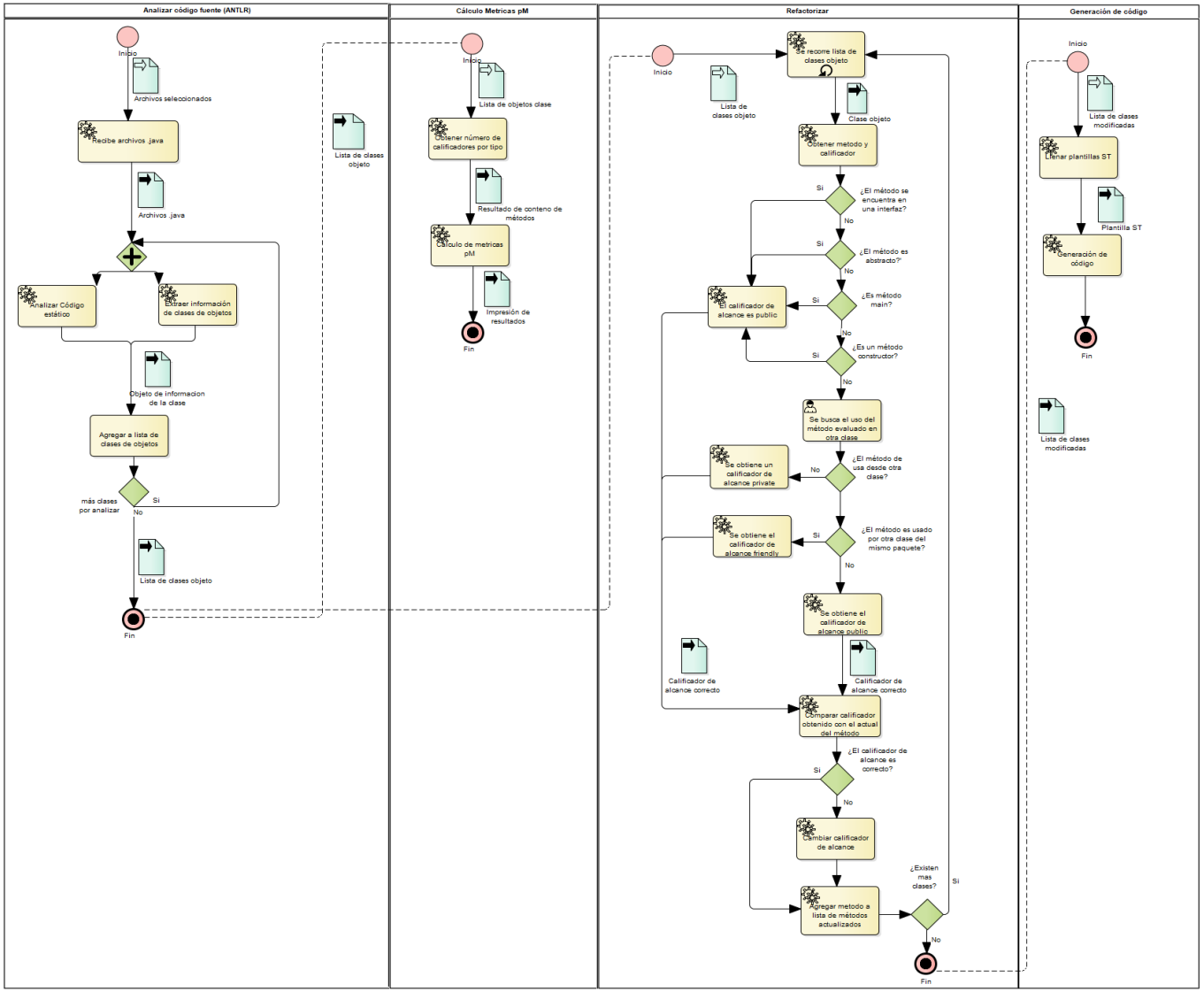


Figura 10.- Diagrama BPMN con el detalle de actividades de los subprocesos del método de refactorización de calificadores de alcance

Capítulo 5.- Desarrollo del sistema

A continuación, se describe el desarrollo del método de refactorización de calificadores de alcance. En el desarrollo se utilizaron diferentes diagramas de modelado orientado a objetos bajo el estándar UML, tales como: casos de uso, diagramas de secuencia, diagramas de clases y diagramas de actividades.

5.1 Diagrama de caso de uso del método de refactorización de calificadores de alcance

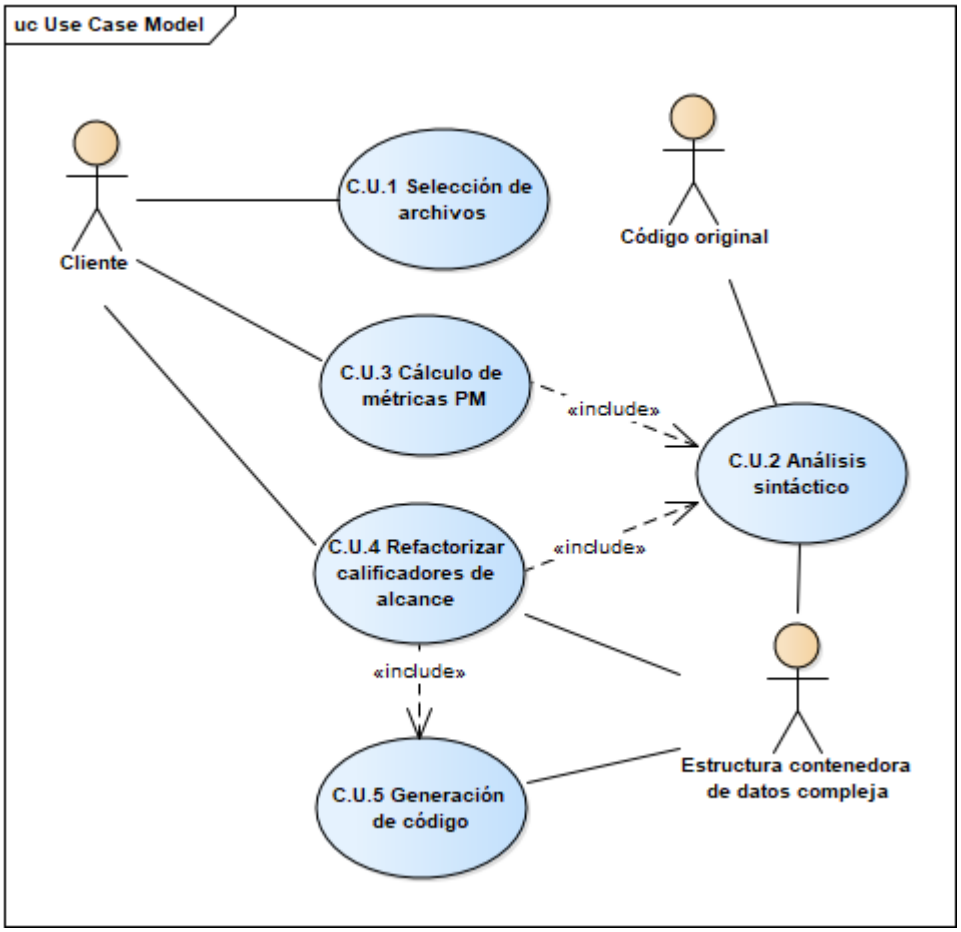


Figura 11.- Diagrama de caso de uso del método de refactorización de calificadores de alcance

A continuación de describe el análisis de cada uno de los casos de uso del diagrama de la Figura 11. En esta descripción se muestra el nombre, descripción, actores, precondiciones, escenario principal de éxito, excepciones, postcondiciones y los posibles necesarios de fracaso.

Tabla 3.- Descripción del caso de uso Selección de archivos

Nombre:	C.U.1: Selección de archivos
Descripción:	El sistema permitirá que el usuario seleccione los archivos en lenguaje Java a refactorizar.
Actores:	Usuario físico

Precondiciones:	1.- El código a refactorizar deberá estar escribo en lenguaje java. 2.- El código a refactorizar no deberá presentar errores de sintaxis.
Escenario principal de éxito:	1.- Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2.- El usuario selecciona la carpeta o los archivos con el código fuente de la aplicación a refactorizar. 3.- Cada archivo Java se almacena en una lista. 4.- Termina el caso de uso.
Excepciones:	En caso de no seleccionar ningún archivo con extensión Java, el sistema informara que no se encontró ningún archivo con extensión Java y termina el proceso de refactorización.
Postcondiciones:	Como resultado se obtiene una lista ligada de los archivos con extensión Java seleccionados, los cuales se someterán al análisis sintáctico.
Escenario de fracaso 1:	1.- No se muestra el cuadro de dialogo de selección de archivos. 2.- El usuario ejecuta nuevamente el sistema, se retoma el paso 1 del escenario principal de éxito.
Escenario de fracaso 2:	1.- Se muestra la ventana exploradora para seleccionar los archivos Java. 2.- El usuario selecciona una carpeta de archivos. 3.- El sistema no encuentra ningún archivo con extensión Java en la carpeta seleccionada. 4.- El sistema alerta al usuario que no se encontraron archivos java. 5.- Termina el caso de uso.

Tabla 4.- Descripción del caso de uso Análisis sintáctico

Nombre:	C.U.2: Análisis sintáctico
Descripción:	Cada archivo seleccionado en el C.U.1 será sometido al proceso de análisis sintáctico de código fuente bajo la gramática de lenguaje Java versión 8.0 para obtener la información necesaria para los subprocesos de cálculo de métricas PM y el de refactorización, de cada una de las funciones que se encuentran en las clases de objetos que conforman la aplicación original.
Actores:	Código fuente original y Estructura contenedora de datos compleja (lista ligada)
Precondiciones:	1.- Contar con la lista que contiene los archivos con el código fuente original derivados del C.U.1

Escenario principal de éxito:	1.- Se crea una lista para contener la información de las clases de objetos de los archivos seleccionados. 2.- Se analiza cada una de las clases de objetos resguardando su información en variables que conforman un objeto de tipo "Clase". Cada objeto de tipo "Clase" se almacena en una lista compleja de objetos de este tipo. Cada nodo de esta lista, a la vez contiene una referencia a una lista de funciones de cada objeto de tipo de "Clase". 3.- Así mismo, se analiza cada función de cada clase resguardando la información en un objeto de tipo "Metodo". Cada objeto de tipo "Metodo" se almacena en la lista de funciones del nodo correspondiente al objeto de tipo "Clase" de manera apropiada. 4.- Termina el caso de uso.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá una lista de objetos de tipo "Clase" con toda la información necesaria para los subprocesos de cálculo de métricas PM y de refactorización.
Escenario de fracaso 1:	1.- Se crea una lista de objetos de tipo "Clase" en base a los archivos seleccionados. 2.- Se analiza cada una de las clases resguardando su información en variables, así mismo se obtienen las funciones de cada una de las clases y se guardan en una lista de funciones de la misma clase. 3.- El intérprete falla en el análisis sintáctico. 4.- El intérprete no genera el código y envía un mensaje de error. 5.- Termina el caso de uso.

Tabla 5.- Descripción del caso de uso Cálculo de métricas PM.

Nombre:	C.U.3: Cálculo de métricas PM
Descripción:	El sistema realiza un análisis sintáctico sobre los archivos seleccionados en el C.U.1, para obtener la información necesaria de cada una de las funciones que se encuentran en las clases que conforman a la aplicación a refactorizar.
Actores:	Lista de objetos clase
Precondiciones:	Contar con lista de objetos de tipo "Clase" con toda la información necesaria derivados del C.U.2.

Escenario principal de éxito:	1.- Se identifican las funciones de cada tipo ("*private*", "*friendly*", "*protected*" y "*public*"). 2.- Se calculan las métricas PMFP, PMFF, PMFPR y PM. 3.- Se calcula la métrica TPM. 4.- Termina caso de uso.
Excepciones:	
Postcondiciones:	Como resultado de obtendrán los resultados de las métricas PMFP, PMFF, PMFPr, PM y TPM.
Escenario de fracaso 1:	

Tabla 6.-Descripción del caso de uso Refactorizar calificadores de alcance.

Nombre:	C.U.4: Refactorizar calificadores de alcance
Descripción:	El sistema aplica el método de refactorización de código, con el fin de corregir en la declaración de funciones, los calificadores de alcance incorrectos que fueron asignados sin tomar en cuenta las reglas de ocultamiento de información.
Actores:	Usuario físico y Estructura contenedora de datos compleja (lista ligada)
Precondiciones:	1.- Contar con la lista de objetos de tipo "Clase" con toda la información necesaria, derivada del C.U.2.
Escenario principal de éxito:	1.- Después de efectuar el C.U.2, aplicando los criterios de identificación de calificadores de alcance, se determina el calificador correcto de cada una de las funciones encontradas. 2.- Se aplica el método de refactorización de código, con el fin de corregir en la declaración de funciones el calificador de alcance. 3.- Cada objeto de tipo "Clase" con el código refactorizado se almacena en una nueva lista compleja de objetos de este tipo. 4.- Termina el caso de uso.
Excepciones:	
Postcondiciones:	Como resultado de obtendrá una lista compleja de objetos de tipo "Clase" con el código refactorizado.

Escenario de fracaso 1:	1.- Después de efectuar el C.U.2, aplicando los criterios de identificación de calificadores de alcance, se determina el calificador correcto de cada una de las funciones encontradas. 2.- El sistema determina un escenario no previsto y no sabe cómo responder, por lo termina el proceso de refactorización. 3.- Termina el caso de uso.

Tabla 7.- Descripción del caso de uso Generación de código.

Nombre:	C.U.5: Generación de código
Descripción:	El sistema alimenta a la plantilla *StringTemplate* con la información de cada objeto de tipo "Clase" presente en la lista compleja generada en el C.U.4, y genera el código refactorizado a cada una de las clases del sistema original.
Actores:	Estructura contenedora de datos compleja (lista ligada)
Precondiciones:	Contar con la lista que contiene los objetos de tipo "Clase" con el código refactorizado derivado del C.U.4
Escenario principal de éxito:	1.- Se recorre la lista ligada de la estructura de datos compleja tomando la información de cada uno de los objetos de tipo "Clase", plasmándola en la plantilla *StringTemplate* ST. 2.- Desde la plantilla *StringTemplate* se genera el código refactorizado equivalente en funcionalidad al código original de entrada. 3.- Se almacenan en una carpeta denominada "código nuevo" los archivos con el código refactorizado que sustituirá al código original.
Excepciones:	
Postcondiciones:	Código refactorizado almacenado en la carpeta "código nuevo" equivalente en funcionalidad al código original de entrada.
Escenario de fracaso 1:	1.- Se recorre la lista ligada de la estructura de datos compleja tomando la información de cada uno de los objetos de tipo "Clase", plasmándola en la plantilla "*StringTemplate"* ST. 2.- El sistema determina un escenario no previsto y no sabe cómo responder, por lo termina el proceso de generar código. 3.- Ninguno de los archivos refactorizados aparase en la carpeta "código nuevo". 4.- Termina caso uso.

5.2 Diagrama de secuencia del cálculo de métricas PM

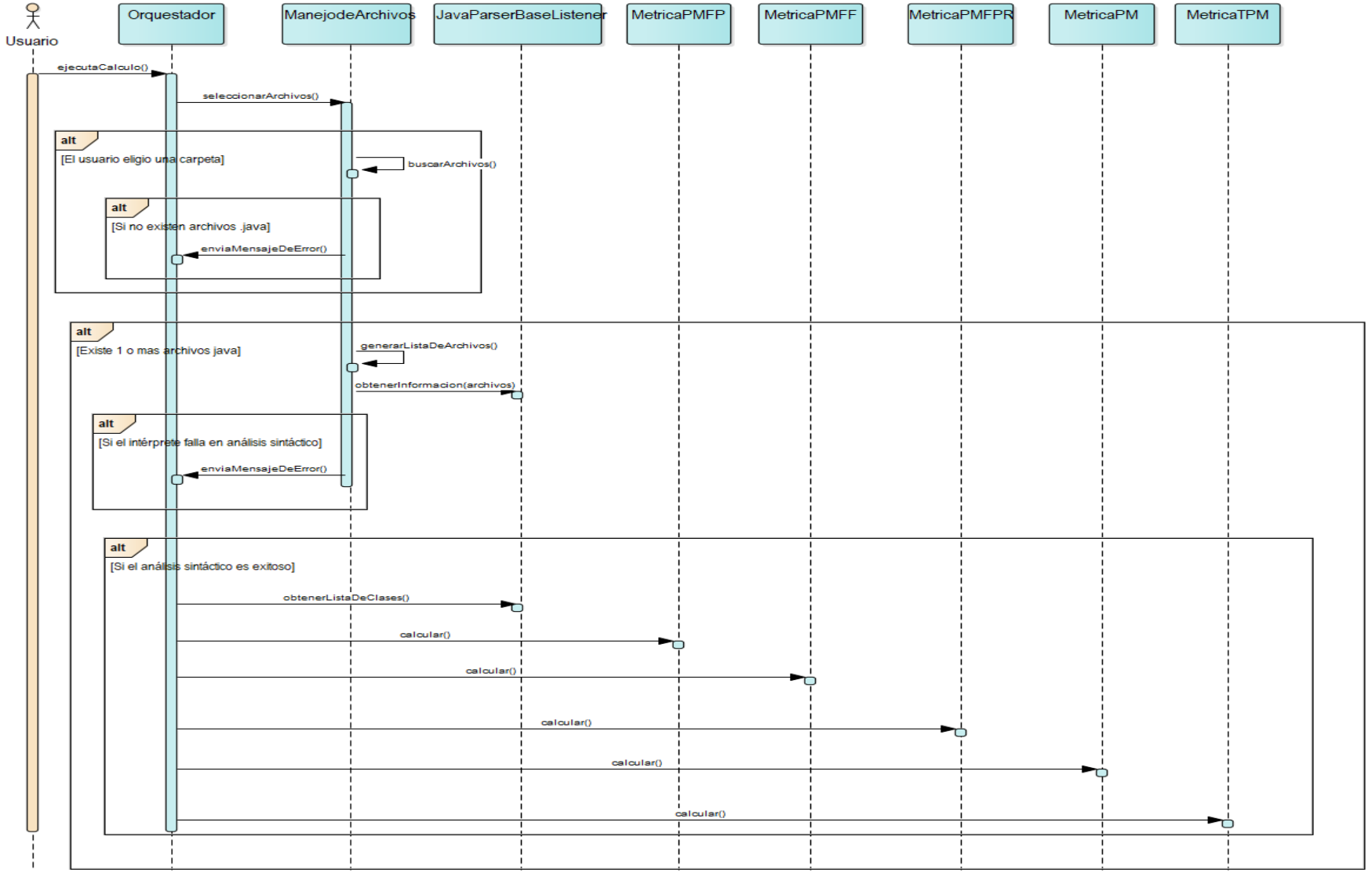


Figura 12.- Diagrama de secuencia del cálculo de métricas PM.

Descripción del diagrama de secuencia del cálculo de métricas PM Figura 12:

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 la lista de objetos tipo “Clase” con toda la información recabada.

8.- Se calcula el valor la métrica PMFP

9.- Se calcula el valor de la métrica PMFF

10.- Se calcula el valor de la métrica PMFPR

11.- Se calcula el valor de la métrica PM

12.- Se calcula el valor de la métrica TPM

13.- Se despliegan los resultados.

5.3 Diagrama de secuencia del método de refactorización de calificadores de alcance

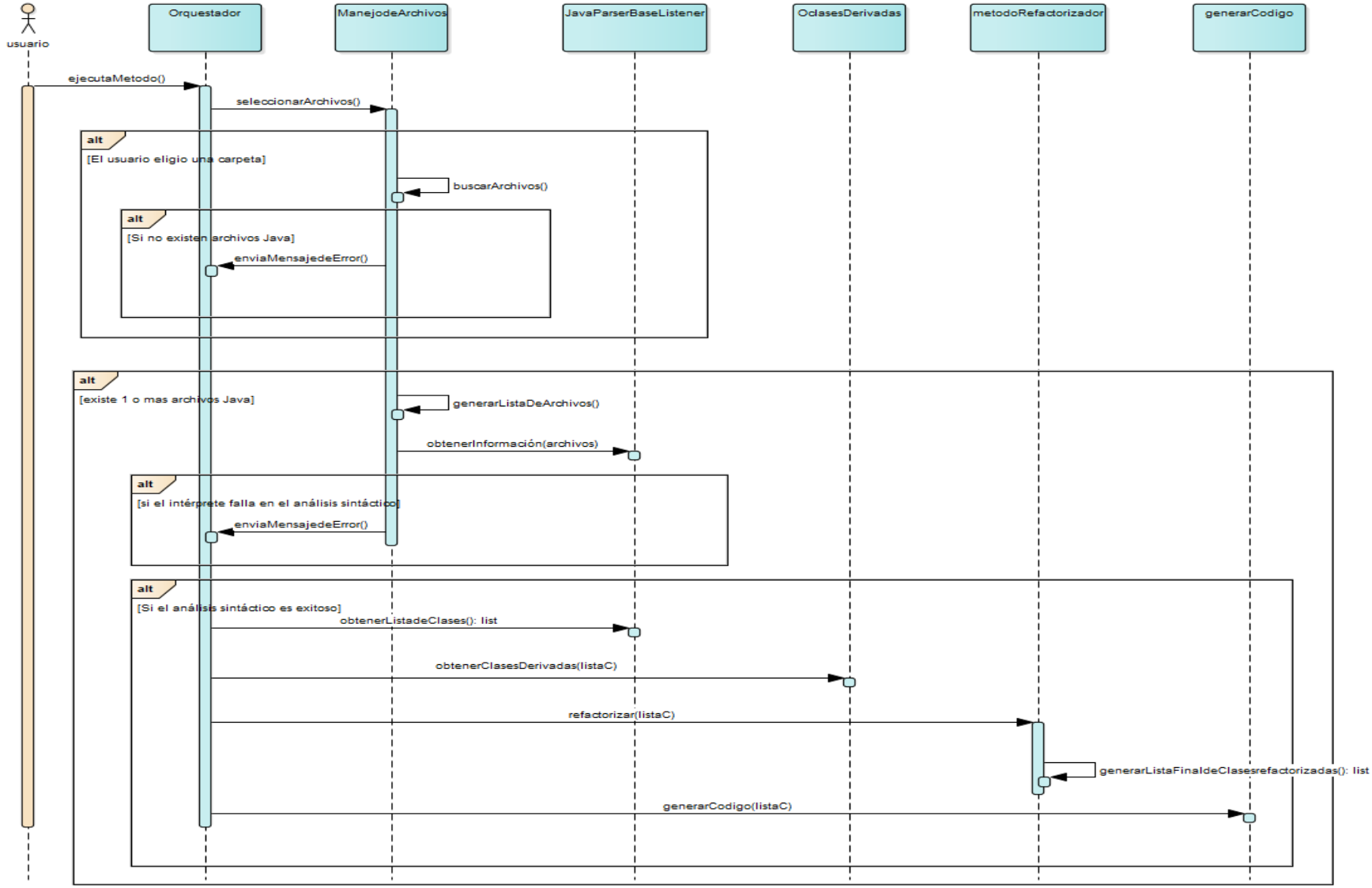


Figura 13.- Diagrama de secuencia del método o de refactorización de calificadores de alcance

Descripción del diagrama de secuencia del método de refactorización Figura 13:

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 el de refactorización de calificadores de alcance.

10.- Aplicando los criterios de identificación de calificadores de alcance, se determina el calificador 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.

5.4 Diagramas de actividades

En la Figura 14 se muestra el diagrama de actividades del método de refactorización de calificadores de alcance. Como primera actividad se encuentra la selección de archivos en donde el cliente tiene la opción de seleccionar archivos con extensión "*Java*" o una carpeta que contenga los archivos a refactorizar, si el cliente selecciona una carpeta la siguiente actividad será la búsqueda de archivos "*Java*". En caso de no encontrar archivos "*Java*" la siguiente actividad será enviar el siguiente mensaje de error; "No se seleccionó ningún archivo .java". En caso de encontrar archivos "*Java*" se procede a generar la lista de archivos seleccionados. La siguiente actividad es la obtención de información necesaria para el proceso de refactorización, dicha información es almacenada en una lista compleja de objetos de tipo "Clase". En caso que durante la obtención de información ocurriera un error se envía un mensaje de error "Se produjo un error durante el análisis", de lo contrario, si el análisis sintáctico se realiza de manera exitosa se aplica el método de refactorización de calificadores de alcance a cada una de las funciones de las clases. Posteriormente se genera una lista de objetos de tipo "Clase", los cuales contienen el nuevo código refactorizado equivalente al original, dicha lista es utilizada para la generación del código nuevo, tomando la información de cada uno de los objetos de tipo "Clase", colocándola en la plantilla StringTemplate. Como última actividad se guardan los archivos con el código refactorizado.

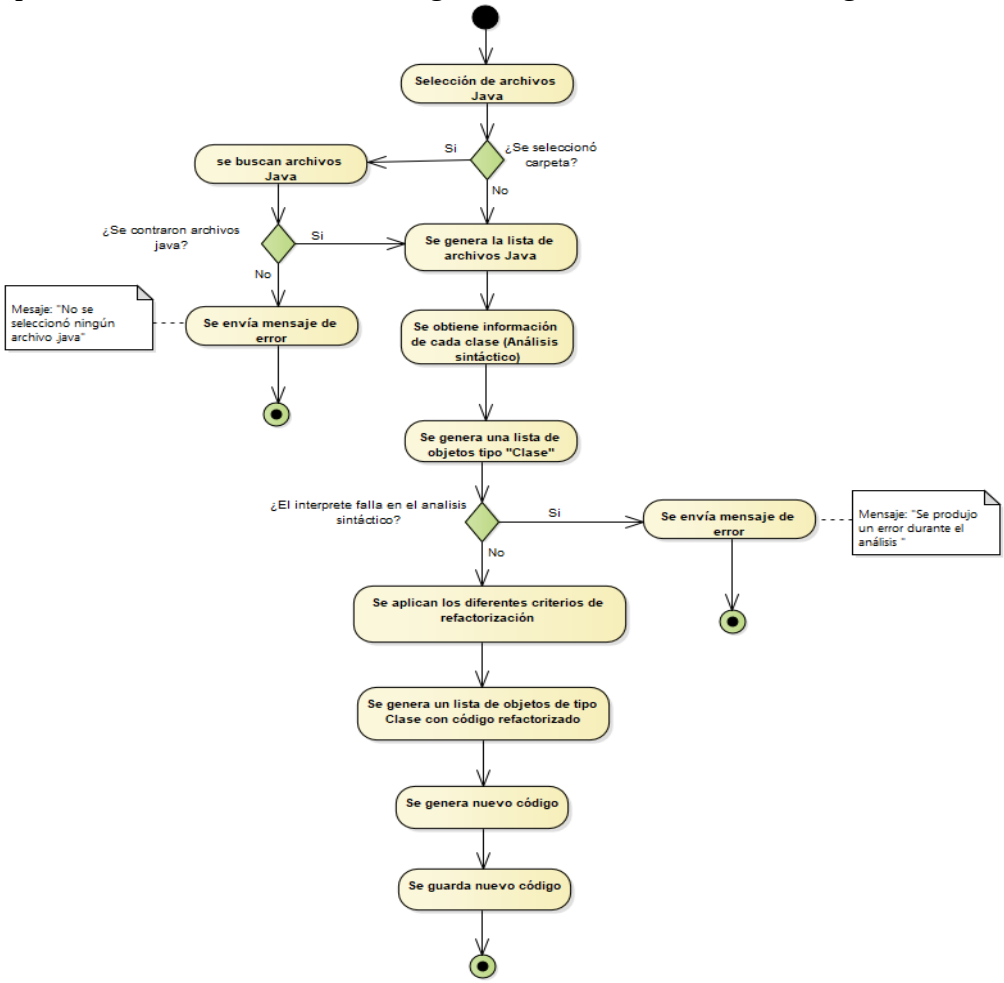


Figura 14.- Diagrama de actividades general del proceso de refactorización de calificadores de alcance

En la Figura 15 se muestra el diagrama de actividades del método “refactorizar” de la clase “Refactorizar”, donde se identifica en base a los criterios de identificación del calificador de alcance cuál calificador es el correcto para cada una de las funciones que conforman a la aplicación a refactorizar.

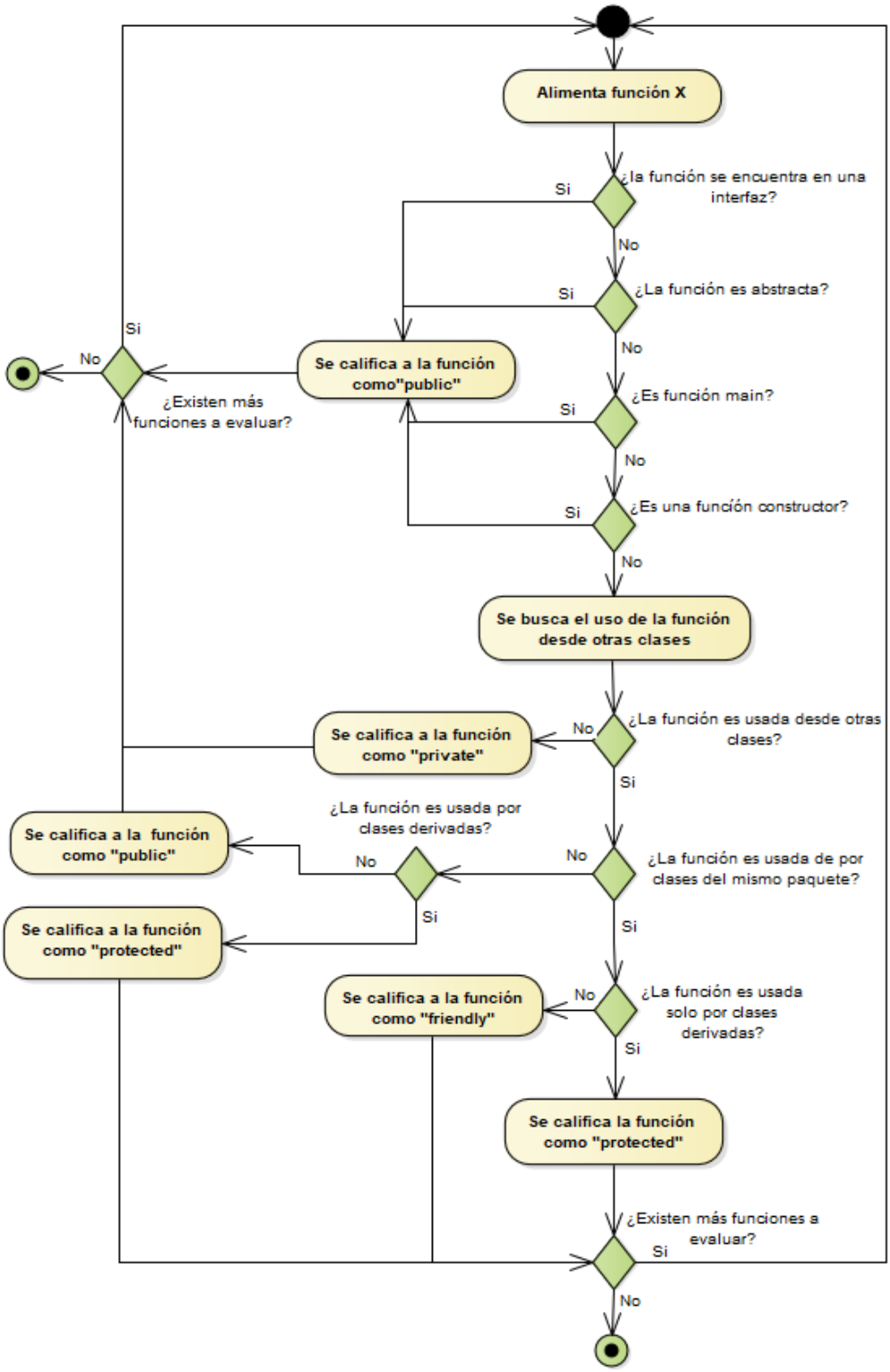


Figura 15.- Diagrama de actividades del método refactoriza

5.4 Diagrama de clases del sistema SR2-Refactoring

La Figura 16 muestra el diagrama de clases del sistema SR2-Refactoring el cual contiene los métodos de refactorización para código legado.

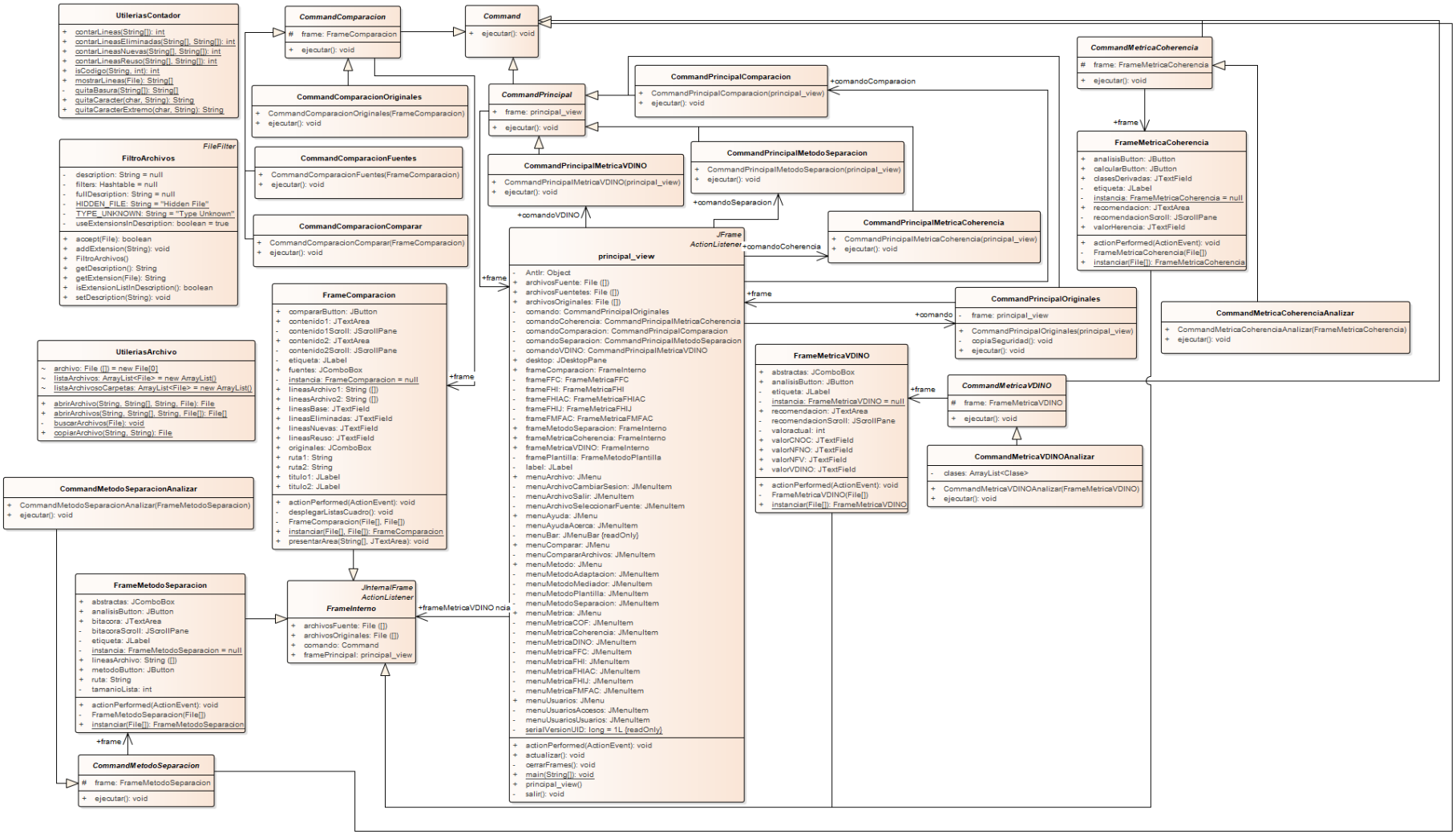


Figura 16.- Diagrama de clases del Sistema SR2-Refactoring.

5.4 Diagrama de clases del método de refactorización de calificadores de alcance

La Figura 17 muestra la arquitectura de clases del método de refactorización de calificadores de alcance, dicho método fue agregado como un paquete a la herramienta SR2-Refactoring, respetando el principio de abierto-cerrado y extendiendo su funcionalidad sin modificar el código existente. El procedimiento de importación del paquete se encuentra descrito en el Anexo B.

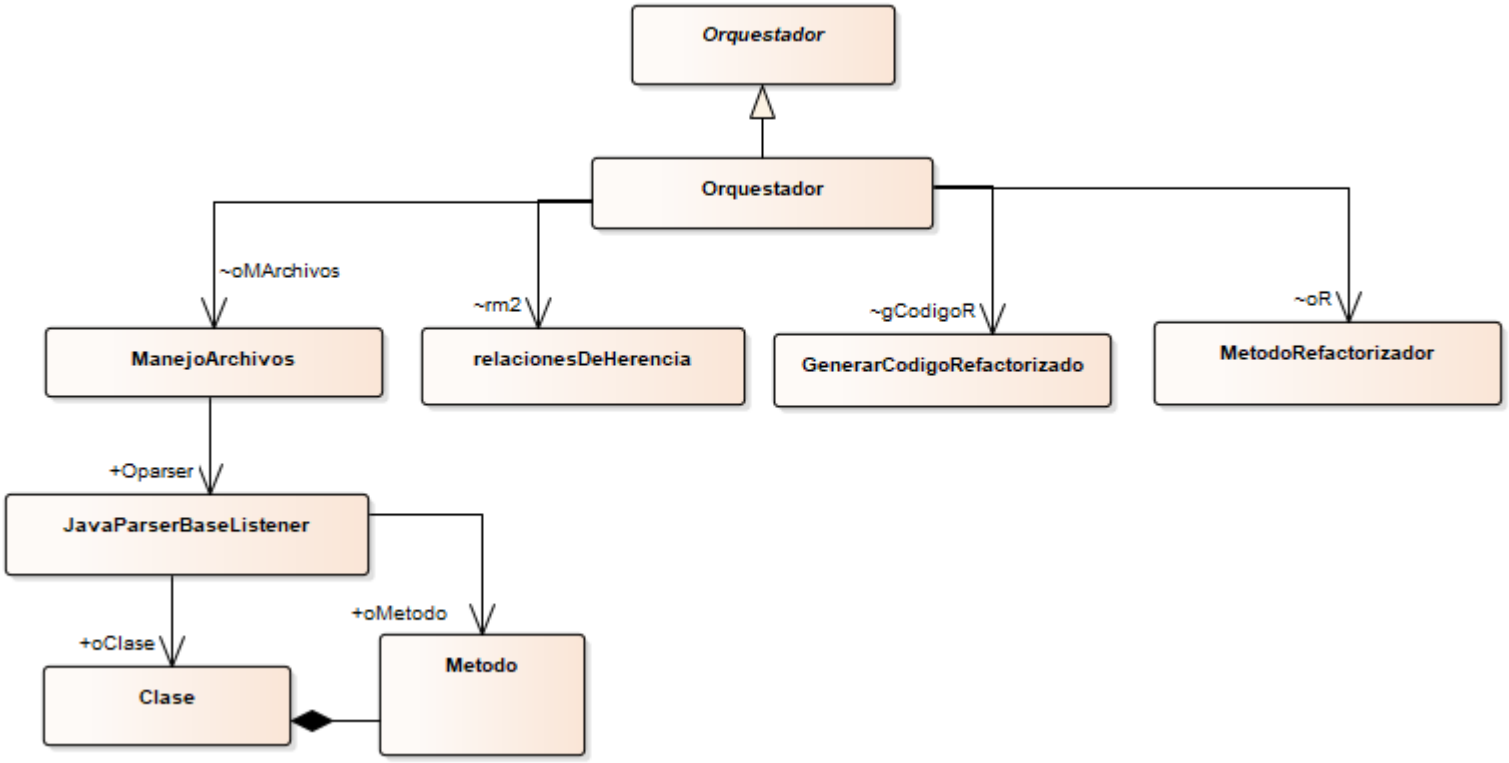


Figura 17.-Diagrama de clases del método de refactorización de calificadores de alcance.

En la Figura 18 se muestra la arquitectura final del sistema SR2-Refactoring, resalto de color verde el nuevo paquete correspondiente al método de refactorizacion de calificadores de alcance.

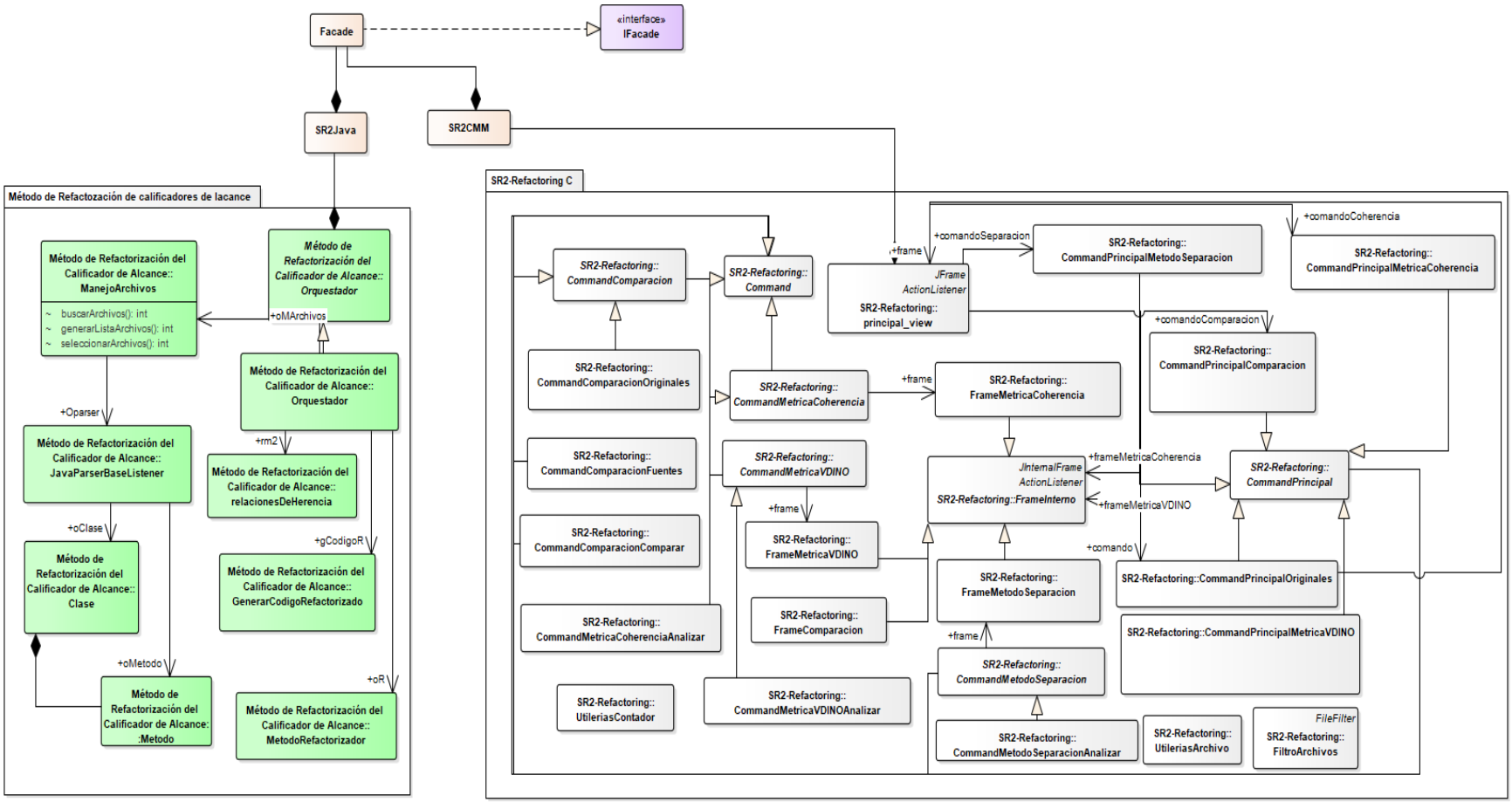


Figura 18.- Diagrama de clases del sistema SR2-Refactoring con el nuevo paquete del Método de refactorización de calificadores de alcance

5.4 Diagrama de clases del Marco de Métricas

En la Figura 19 se puede observar la arquitectura del marco orientado a objetos desarrollado para obtener servicios de cálculos de métricas (Santaolaya Salgado, R., Fragoso Diaz, O. G., Ortiz Gutierrez, O., Bautista Juarez, C. V., 2019). En la figura se enmarca en color azul la extensión de las cinco metricas de protección modular: metricaPMFP, metricaPMFF, metricaPMFPR, metricaPM y metricaTPM.

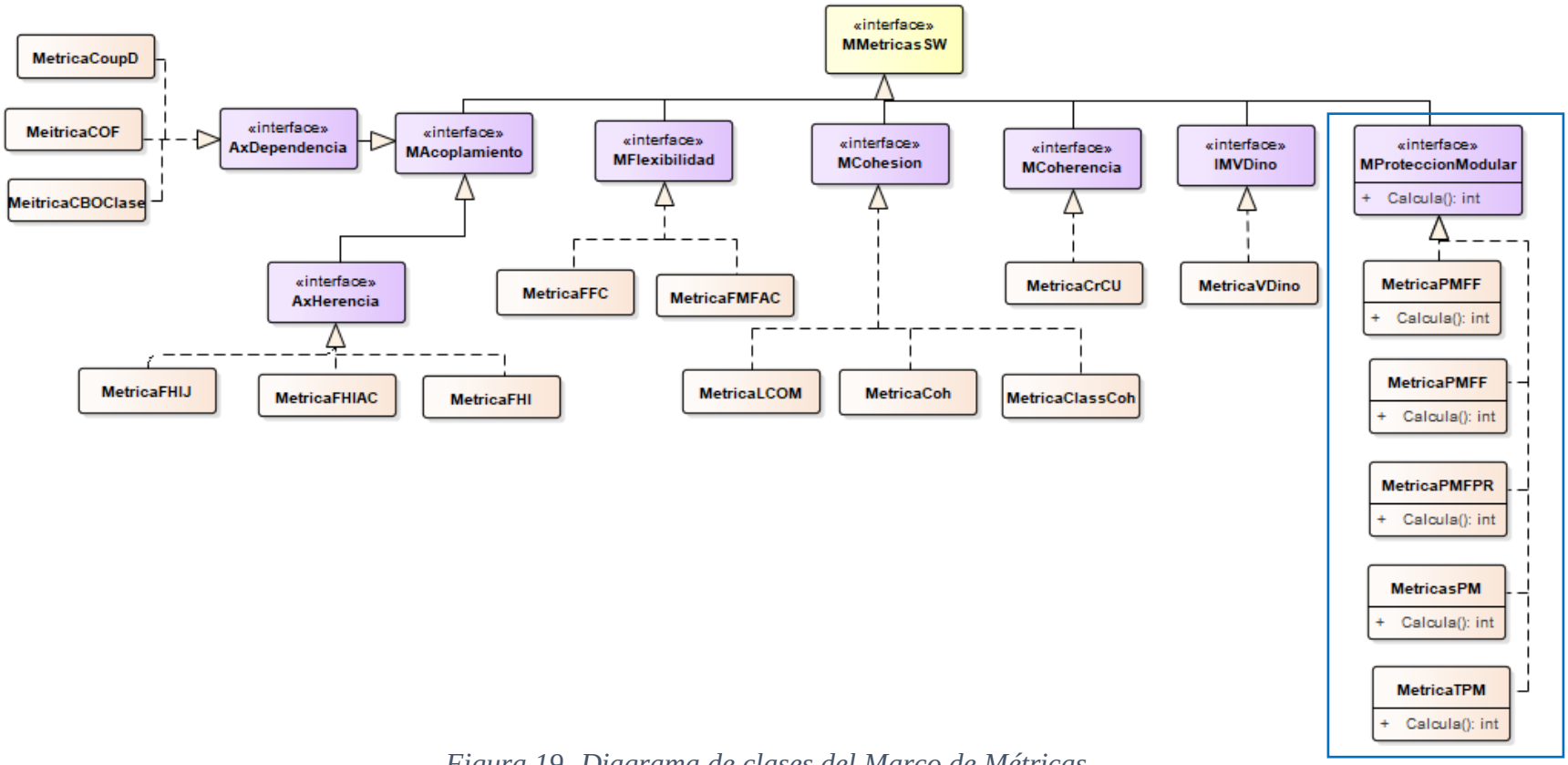


Figura 19.-Diagrama de clases del Marco de Métricas.

Capítulo 6.- Pruebas

6.1 Convención de nombres

En la Figura 20 se muestra la convención de nombres que se utilizara durante la evaluación del método de refactorización de calificadores de alcance.

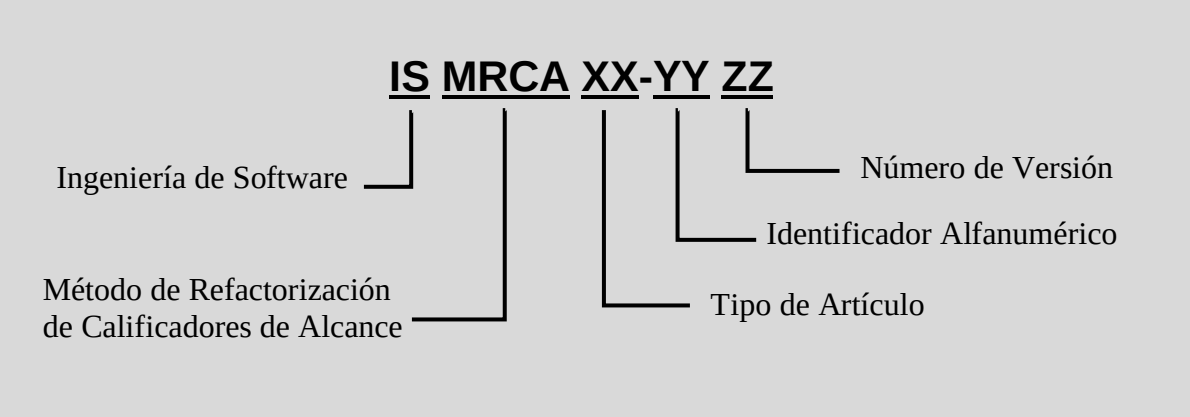


Figura 20.- Convención de nombres

Tipo de Artículo	
01	Módulos de programa.
02	Programas de control.
03	Plan de pruebas.
04	Diseño de pruebas.
05	Casos de prueba.

Programas de Control	
ISMRCA01 - nn	Módulos de programa

Módulos de programa	
ISMRCA02 - nn	Programas de control, utilerías, ordenadores, entre otros.

Documentación de pruebas	
ISMRCA03 - YYZZ	Plan de pruebas.
ISMRCA04 - YYZZ	Especificación de diseño de pruebas.
ISMRCA05 - YYZZ	Especificación de casos de prueba.

6.2 Plan de pruebas

6.2.1.- Plan de prueba: ISMRCA03 – 01: Plan de pruebas para la validación del correcto funcionamiento del método de refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles.

6.2.2.- Introducción

El método de refactorización de calificadores de alcance tiene la función de analizar cada una de las funciones que conforman a las diversas clases de una aplicación en java, con el fin de verificar que el calificador de alcance de cada función sea el correcto, de lo contrario cambiarlo por el adecuado.

6.2.3.- Artículos de prueba. Los módulos a ser probados se muestran en la Tabla 8.

Tabla 8.- Módulos de programa

Sistema	Función	No.
Método de Refactorización de calificadores de alcance	Subsistema de evaluación del calificador de alcance.	ISMRCA01 - 01
Marco de Métricas de Calidad de Arquitecturas Orientadas a Objetos	Cálculo de métricas de protección modular.	ISMRCA01 - 02

6.2.4.- Procedimientos de control de tareas mostrados en la Tabla 9.

Tabla 9.- Control de tareas.

Sistema	Función	No.
Programa de aplicación	Localización del paquete al que pertenece la clase.	ISMRCA02 - 01
Programa de aplicación	Localización de las importaciones de la clase.	ISMRCA02 - 02
Programa de aplicación	Localización del nombre de la clase.	ISMRCA02 - 03
Programa de aplicación	Localización de la clase padre de la clase (si existiera).	ISMRCA02 - 04
Programa de aplicación	Localización de atributos de la clase.	ISMRCA02 - 05
Programa de aplicación	Localización de funciones de la clase	ISMRCA02 - 06
Programa de aplicación	Localización de los atributos de funciones de clase, tales como: calificador de alcance, si es abstracto, si es estático, nombre, parámetros, cuerpo.	ISMRCA02 - 07
Programa de aplicación	Conteo del número total de funciones *"public".*	ISMRCA02 - 08

Programa de aplicación	Conteo del número total de funciones *"private"*.	ISMRCA02 - 09
Programa de aplicación	Conteo del número total de funciones *"protected"*.	ISMRCA02 - 10
Programa de aplicación	Conteo del número total de funciones *"friendly"*.	ISMRCA02 - 11
Programa de aplicación	Cálculo de la métrica *PMFP*	ISMRCA02 - 12
Programa de aplicación	Cálculo de la métrica *PMFPr*	ISMRCA02 - 13
Programa de aplicación	Cálculo de la métrica *PMFF*	ISMRCA02 - 14
Programa de aplicación	Cálculo de la métrica P*M*	ISMRCA02 - 15
Programa de aplicación	Cálculo de la métrica TPM	ISMRCA02 - 16
Programa de aplicación	Método de validación del calificador de alcance.	ISMRCA02 - 17
Programa de aplicación	Método de refactorización de calificadores de alcance.	ISMRCA02 - 18
Programa de aplicación	Llenado de la plantilla ST y generación de código refactorizado.	ISMRCA02 - 19
Programa de aplicación	Compilación del código refactorizado.	ISMRCA02 - 20
Programa de aplicación	Verificación del funcionamiento del código refactorizado.	ISMRCA02 – 21

7.- Características a ser probadas.

En la Tabla 10 se muestran las características que deben ser probadas.

Tabla 10.- Características a probar.

Diseño de Prueba No. de especificación	Descripción
ISMRCA04 – 01	Cálculo de métricas de calidad
ISMRCA04 – 02	Método de refactorización de calificadores de alcance.

8.- Características a no ser probadas.

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

9.- Enfoque.

La realización de los casos de prueba y la actividad de pruebas estará asistida por la alumna de maestría del Cenidet Nelida Barón Pérez. Esto ayudará para asegurar que las pruebas representan efectivamente el desarrollo y uso del sistema SR2 completo.

9.1.- 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.

9.2.- 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 calificador de cada una de las funciones por el calificador 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.

9.3.- Pruebas de calidad.

Las pruebas de calidad incluyen la aplicación de métricas de protección modular PMFP, PMFPr, PMFF, PM y TPM para la medición comparativa del código original contra el código orientado a objetos obtenido del proceso de refactorización.

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

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

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

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

11.- Criterio de suspensión y requisitos de reanudación.

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

12.- Liberación de pruebas.

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

6.3 Especificación del diseño de pruebas

6.3.1 Diseño de Prueba: ISMRCA04 – 01.

1.- Diseño de Prueba: ISMRCA04 – 01. Método de refactorización de calificadores de alcance.

2.- Características a ser probadas.

> 2.1. En esta prueba se evaluará el correcto funcionamiento del método de refactorización de calificadores de alcance (cambio del calificador de alcance a funciones que lo requieran).

3.- Refinamiento del enfoque

El objetivo es evaluar el cambio de calificador de alcance de las funciones que lo requieran. En particular, que la refactorización del código, refleje un aumento en la protección modular.

Antes de realizar cada caso de prueba, éste será compilado y ejecutado previamente en un compilador para el lenguaje Java, con objeto de corroborar que su construcción está o no correctamente escrita. Posteriormente, el subsistema de refactorización de código tomará este archivo y realizará un reconocimiento léxico y sintáctico, generando las estructuras de datos en memoria con la información necesaria para efectos de la refactorización.

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

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

6.3.2 Diseño de Prueba: ISMRCA04 – 02.

1.- Diseño de Prueba: ISMRCA04 – 02. Cálculo de métricas de calidad.

2.- Características a ser probadas.

> 2.1.- En esta prueba se evaluará el correcto cálculo de las métricas de protección modular (PMFP, PMFPR, PMFF, PM y TPM).

3.- Refinamiento del enfoque.

El objetivo es evaluar que el correcto funcionamiento de las métricas de protección modular al evaluar un sistema de software legado.

Antes de realizar cada caso de prueba, éste será compilado y ejecutado previamente en un compilador para el lenguaje Java, con objeto de corroborar que su construcción está o no correctamente escrita. Posteriormente, el Marco Orientado a Objetos para el Cálculo de Métricas tomará este código y realizará un reconocimiento léxico y sintáctico, para generar la estructura de datos con la información requerida para estos cálculos.

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

En cada caso de prueba se presentarán los resultados obtenidos de manera manual de las métricas PMFP, PMFF, PMFPR, PM y TPM. El criterio de evaluación de esta prueba se realizará tomando en cuenta los resultados manuales. Un caso de prueba debe considerarse válido cuando se empaten los resultados manuales con los resultados obtenidos de forma automática. Para que se pase la prueba, cada característica debe pasar todos sus casos de prueba.

6.4 Especificación de casos de prueba

6.4.1.- Caso de Prueba: ISMRCA05 - 01.

1.- Artículo de Prueba: DBLista

El DBLista es un sistema desarrollado en lenguaje java conformado por 9 clases, tiene objetivo realizar funciones con listas como son: insertar, eliminar, recorrer, etc.

2.- Características a probar, del proceso del cálculo de métricas de calidad se muestran en la Tabla 11.

Tabla 11.- Características a probar del proceso del cálculo de métricas de calidad

Característica a Ejercitar	**Diseño de Prueba No. de especificación**
Conteo de funciones *"public".*	ISMRCA04 – 02
Conteo de funciones "*private*".	ISMRCA04 – 02
Conteo de funciones *"protected".*	ISMRCA04 – 02
Conteo de funciones *"friendly".*	ISMRCA04 – 02
Cálculo del No. funciones	ISMRCA04 – 02
Cálculo correcto de las métricas de calidad PM	ISMRCA04 – 02

3.- Especificación de entrada

Las entradas al proceso de medición de las métricas de calidad son:

1.- Las clases pertenecientes a la aplicación DBLista compilado y probado previamente.

4.- Especificación de salida

A la salida del proceso de medición de las métricas de calidad se deberá tener:

1. El valor obtenido de cada una de las métricas de manera independiente.

6.4.2.- Caso de Prueba: ISMRCA05 - 02.

1.- Articulo de Prueba: DBLista

El DBLista es un sistema desarrollado en lenguaje java conformado por 9 clases, tiene objetivo realizar funciones con listas como son: insertar, eliminar, recorrer, etc.

2.- Características a probar, del proceso del método de refactorización de calificadores de alcance se muestran en la Tabla 12.

Tabla 12.- Características a probar del proceso de refactorización

Característica a Ejercitar	No. de especificación del diseño de prueba
Análisis del calificador de alcance correcto.	ISMRCA04 – 01
Cambio del calificador de alcance.	ISMRCA04 – 01
Mantener el mismo comportamiento después de la refactorización	ISMRCA04 – 01
Aumento de protección modular	ISMRCA04 – 01

3.- Especificación de entrada

Las entradas al proceso de reestructura son:

1. Las clases pertenecientes a la aplicación DBLista compilado y probado previamente.

4.- Especificación de salida

A la salida del proceso de reestructura se deberá tener:

1. Las clases pertenecientes a la aplicación DBLista refactorizadas libre de deuda técnica originado por código desagradable que se caracteriza por contar con bajos niveles de protección modular en diferentes entidades de software orientadas a objetos.

6.4.3.- Caso de Prueba: ISMRCA05 - 03.

1.- Artículo de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje java está conformado por 46 clases y tiene como objetivo realizar cálculos estadísticos automáticamente.

2.- Características a probar, del proceso del cálculo de métricas de calidad se muestran en la Tabla 13.

Tabla 13.- Características a probar del proceso del cálculo de métricas de calidad

Característica a Ejercitar	**Diseño de Prueba No. de especificación**
Conteo de funciones *"public".*	ISMRCA04 – 02
Conteo de funciones "*private*".	ISMRCA04 – 02
Conteo de funciones *"protected".*	ISMRCA04 – 02
Conteo de funciones *"friendly".*	ISMRCA04 – 02
Cálculo del No. funciones	ISMRCA04 – 02
Cálculo correcto de las métricas de calidad PM	ISMRCA04 – 02

3.- Especificación de entrada

Las entradas al proceso de medición de las métricas de calidad son:

1.- Las clases pertenecientes a la aplicación del Marco estadístico compilado y probado previamente.

4.- Especificación de salida

A la salida del proceso de medición de las métricas de calidad se deberá tener:

2. El valor obtenido de cada una de las métricas de manera independiente.

6.4.4 Caso de Prueba: ISMRCA05 - 04.

1.- Articulo de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje java está conformado por 46 clases y tiene como objetivo realizar cálculos estadísticos automáticamente.

2.- Características a probar, del proceso del método de refactorización de calificadores de alcance se muestran en la Tabla 14.

Tabla 14.- Características a probar del proceso de refactorización.

Característica a Ejercitar	**No. de especificación del diseño de prueba**
Análisis del calificador de alcance correcto.	ISMRCA04 – 01
Cambio del calificador de alcance.	ISMRCA04 – 01
Mantener el mismo comportamiento después de la refactorización	ISMRCA04 – 01
Aumento de protección modular	ISMRCA04 – 01

3.- Especificación de entrada

Las entradas al proceso de reestructura son:

2. Las clases pertenecientes a la aplicación del Marco estadístico compilado y probado previamente.

4.- Especificación de salida
A la salida del proceso de reestructura se deberá tener:

2. Las clases pertenecientes a la aplicación del Marco estadístico refactorizadas libre de deuda técnica originado por código desagradable que se caracteriza por contar con bajos niveles de protección modular en diferentes entidades de software orientadas a objetos.

6.4.5.- Caso de Prueba: ISMRCA05 - 05.

1.- Artículo de Prueba: PSP Cenidet

El MOO PSPCenidet es un sistema que mide los tiempos que una persona utiliza en realizar tareas específicas.

2.- Características a probar, del proceso del cálculo de métricas de calidad se muestran en la Tabla 15.

Tabla 15.- Características a probar del proceso del cálculo de métricas de calidad

Característica a Ejercitar	**Diseño de Prueba No. de especificación**
Conteo de funciones *"public".*	ISMRCA04 – 02
Conteo de funciones "*private*".	ISMRCA04 – 02
Conteo de funciones *"protected".*	ISMRCA04 – 02
Conteo de funciones *"friendly".*	ISMRCA04 – 02
Cálculo del No. funciones	ISMRCA04 – 02
Cálculo correcto de las métricas de calidad PM	ISMRCA04 – 02

3.- Especificación de entrada

Las entradas al proceso de medición de las métricas de calidad son:

1.- Las clases pertenecientes a la aplicación PSPCenidet compilado y probado previamente.

4.- Especificación de salida

A la salida del proceso de medición de las métricas de calidad se deberá tener:

1. El valor obtenido de cada una de las métricas de manera independiente.

6.4.6 Caso de Prueba: ISMRCA05 - 06.

1.- Articulo de Prueba: PSP Cenidet

El MOO PSP Cenidet es un sistema que mide los tiempos que una persona utiliza en realizar tareas específicas.

2.- Características a probar, del proceso del método de refactorización de calificadores de alcance se muestran en la Tabla 16.

Tabla 16.- Características a probar del proceso de refactorización

Característica a Ejercitar	No. de especificación del diseño de prueba
Análisis del calificador de alcance correcto.	ISMRCA04 – 01
Cambio del calificador de alcance.	ISMRCA04 – 01
Mantener el mismo comportamiento después de la refactorización	ISMRCA04 – 01
Aumento de protección modular	ISMRCA04 – 01

3.- Especificación de entrada

Las entradas al proceso de reestructura son:
1.-Las clases pertenecientes a la aplicación PSP Cenidet compilado y probado previamente.

4.- Especificación de salida

A la salida del proceso de reestructura se deberá tener:

3. Las clases pertenecientes a la aplicación PSP Cenidet refactorizadas libre de deuda técnica originado por código desagradable que se caracteriza por contar con bajos niveles de protección modular en diferentes entidades de software orientadas a objetos.

6.5 Ejecución del plan de pruebas

6.5.1 Caso de prueba: ISMRCA05 - 01.

Artículos de Prueba: DBLista

En la Tabla 17 se enlistan las características a probar del proceso de cálculo de métricas de calidad en la aplicación DBLista.

Tabla 17.- Características a probar, del proceso de cálculo de Métricas de calidad.

Característica a Probar
Conteo de funciones *"public".*
Conteo de funciones "*private*".

Conteo de funciones *"protected"*.
Conteo de funciones *"friendly"*.
Cálculo correcto de las métricas de protección modular (*PMFP*, *PMFPr*, *PMFF*, P*M*, TPM)

Dada la Figura 21 denominada DBLista, que representa un sistema desarrollado en lenguaje java conformado por 9 clases, que tiene como objetivo realizar funciones con listas como son: insertar, eliminar, recorrer, etc. Se procede a realizar el caso de prueba **ISMRCA05 - 01**.

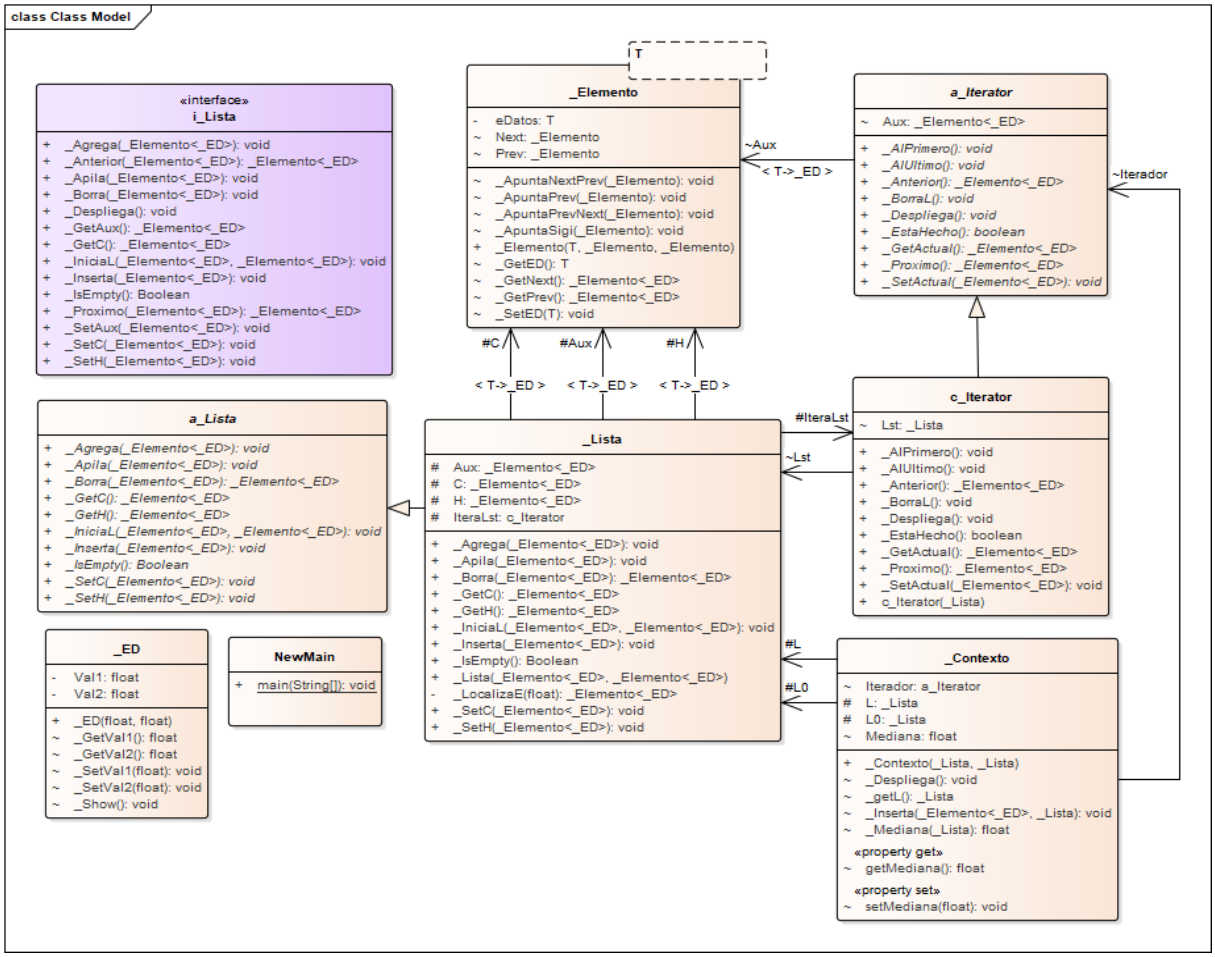


Figura 21.- Arquitectura de clases de la aplicación DBLista antes de la refactorización

Conteo de funciones "*public*", "*private*", "*protected*" y "*friendly*"

Para comprobar que el conteo de funciones de acuerdo con el tipo se realiza de manera correcta, primero se realiza el conteo manual en cada una de las funciones que conforman a la aplicación DBLista, posteriormente se realiza un conteo automático para comparar ambos resultados los cuales deberán ser iguales.

Tabla 18.-Conteo manual y automático de funciones de la aplicación DBLista

Calificador de alcance	Conteo de funciones manual	Conteo de funciones automático

"*public*"	58	No. Métodos public:58 No. Métodos private:1 No. Métodos protected:0 No. Métodos friendly:19
"private"	1	
"*protected*"	0	
"*friendly*"	19	

En la Tabla 18 se puede observar que el conteo manual y el conteo automático correspondiente a la arquitectura de la Figura 21 coinciden en los valores obtenidos de acuerdo al número de funciones por tipo, por lo anterior se comprueba que el conteo de funciones automático de acuerdo a su tipo de se lleva de manera correcta.

Cálculo correcto de las métricas de protección modular (PMFP, PMFPR, PMFF, PM, TPM)

Para comprobar que el cálculo de métricas de protección modular se realiza de manera correcta, primero se realiza el cálculo manual y posteriormente se realiza un cálculo automático para comparar ambos resultados, los cuales deberán ser iguales.

Se sustituyen los valores de acuerdo a la Figura 21 obteniendo los resultados manual y automático, los cuales de muestran en la Tabla 19.

Tabla 19.- Cálculo manual y automático de métricas PM de la aplicación DBLista

Métrica	Número de ecuación	Cálculo manual	Cálculo automático
PMFP	*EC. 1*	$PMFP = \frac{0.08}{9} = 0.009$	Metrica PMFP:0.009
PMFPr	*EC. 2*	$PMFPr = \frac{(\frac{0}{19}+\frac{0}{22})}{2} = \frac{0}{2} = 0$	Metrica PMFPr:0
PMFF	*EC. 3*	$PMFF = \frac{19}{78} = 0.244$	Metrica PMFF:0.244
PM	*EC. 4*	$PM = \frac{20}{78} = 0.256$	Metrica PM:0.256
TPM	*EC. 5*	$TPM = \frac{0.079}{3} = 0.026$	Metrica TPM:0.026

Por lo anterior demostrado en donde el resultado de cada una de las métricas PM obtenido de manera manual siempre coincidió con el resultado obtenido de manera automática, se comprueba que el cálculo de las métricas se lleva de manera correcta.

6.5.2 Caso de prueba: ISMRCA05 - 02.

Artículos de Prueba: DBLista

En la Tabla 20 se enlistan las características a probar del proceso de refactorización.

Tabla 20.- Características a probar del método de refactorización en la aplicación DBLista

Característica a probar
Cambio del calificador de alcance
Mantener el comportamiento de la aplicación después de la refactorización
Aumento de protección modular

Cambio del calificador de alcance de las funciones que lo requieren de la aplicación DBLista

Para comprobar que el cambio de calificador de alcance a funciones que lo requieren se realiza de manera correcta se realizó un análisis manual del código y de la arquitectura de la aplicación DBLista. Antes de someter el código a la refactorización, con el fin de identificar si existen funciones que deban cambiar su calificador de alcance, se analiza el alcance desde donde puede ser invocada cada una de las funciones de las clases que conforman la aplicación. En el análisis manual del código se identificaron que las funciones **_Despliega, _getL, getMediana y setMediana** de la clase **_Contexto y** la función **_SetED** de la clase **_Elemento** presentan un calificador de alcance **"friendly"**, pero ninguno de las funciones antes mencionados es invocado fuera de su clase por lo que su calificador de alcance correcto tendría que ser **"*private*"**. La Figura 22 muestra la arquitectura de clases de la aplicación DBLista refactorizada, se puede apreciar que las cinco funciones anteriormente identificadas presentan el cambio del calificador de alcance "*friendly*" por "*private*".

En la Figura 22 se muestra la arquitectura de clases de la aplicación DBLista refactorizada, en donde se puede observar que las funciones que requerían cambio de calificador identificadas en el análisis manual, cambiaron de manera satisfactoria al calificador de alcance correcto el cual también se identificó en el análisis manual.

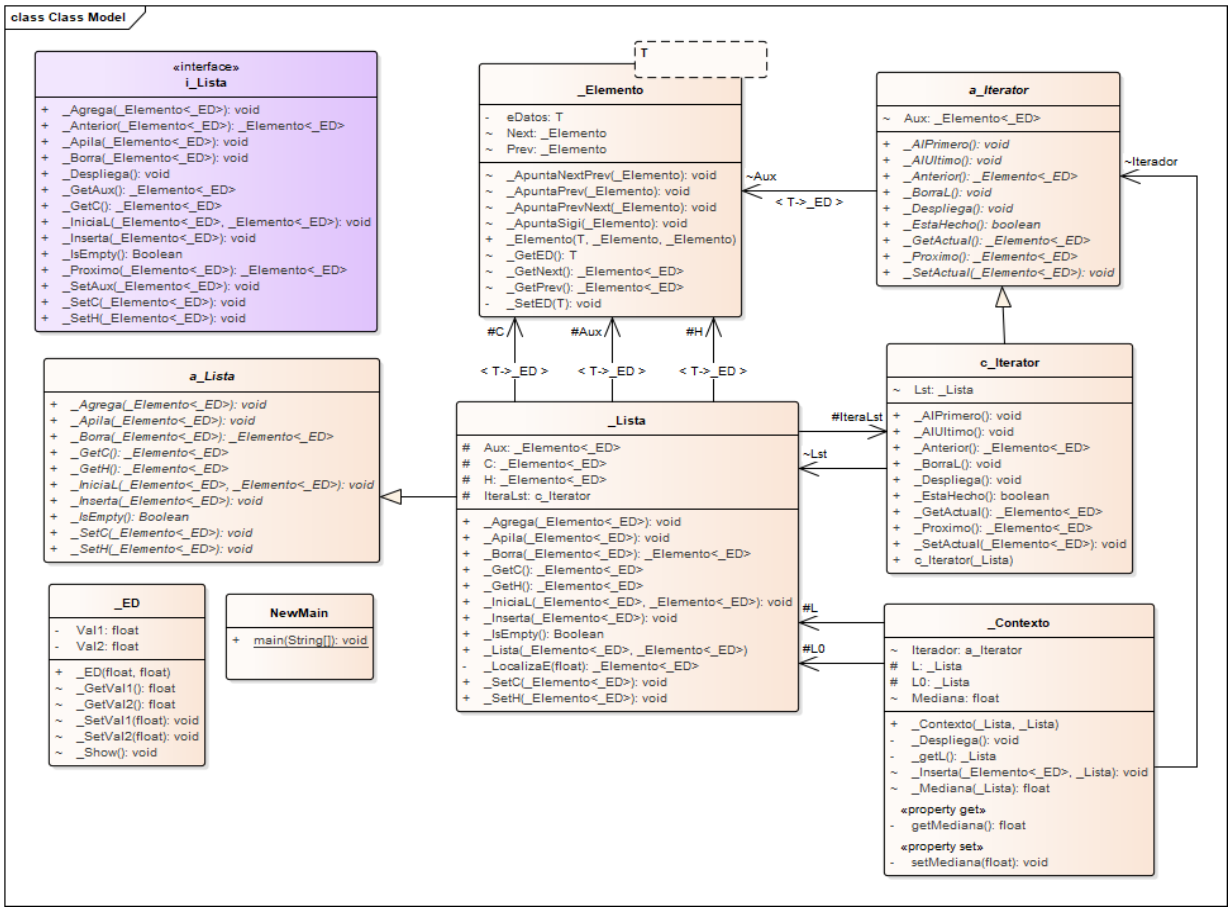


Figura 22.- Diagrama de clases de la aplicación DBLista después de la refactorización

En esta aplicación la mayoría de las funciones presentaba un calificador de alcance correcto, solo cinco del total de las funciones requerían un cambio de calificador de alcance, los cuales fueron identificadas en el análisis manual por lo que se comprueba que el método de refactorización es capaz de identificar y cambiar al calificador de alcance correcto a cada función que conforma la aplicación.

Comprobar que el comportamiento se mantiene después de la refactorización

Para comprobar que la aplicación DBLista mantiene el mismo comportamiento después de ser sometida al método de refactorización se calcula la mediana de dos conjuntos de números (listas de números) antes y después de la refactorización. La Tabla 21 que números conforman cada conjunto.

Tabla 21.- Conjunto de datos

Conjunto 1	Conjunto 2
20	13.8
2.8	34
59	1
8	890
114	34
11.9	2

160	6.6
10.3	90
229	135
23.54	267

En la Tabla 22 se muestran los resultados obtenidos por la aplicación DBLista antes y después de la refactorización, se puede observar que el comportamiento se mantiene, por lo que se comprueba que el método de refactorización no altera el compartimento de la aplicación.

Tabla 22.- Cálculos antes y después de la refactorización de la aplicación DBLista

Conjunto	Resultados antes de la refactorización	Resultado después de la refactorización
1	Mediana de la Lista es:114.0 BUILD SUCCESSFUL (total time:	Mediana de la Lista es:114.0 BUILD SUCCESSFUL (total time:
2	Mediana de la Lista es:13.8 BUILD SUCCESSFUL (total time:	Mediana de la Lista es:13.8 BUILD SUCCESSFUL (total time:

Comprobación del aumento de protección modular en la aplicación DBLista

Uno de los objetivos principales que se tiene en este trabajo de investigación, es el de aumentar la protección modular de la aplicación que se someta al método de refactorización de calificadores de alcance. Para comprobar que la protección de la aplicación DBLista aumentó dicha aplicación se somete al cálculo de las métricas PM antes y después de la refactorización, con el fin de comparar su grado de protección de los distintos niveles de protección.

La Tabla 23 muestra el número de funciones de acuerdo al calificador de alcance antes y después de la refactorización de DBLista.

Tabla 23.- Número de funciones por calificador antes y después de la refactorización de DBLista

Calificador	No. Antes de la refactorización	No. después de la refactorización
"private"	1	6
"*protected*"	0	0
"*friendly*"	19	14
"*public*"	58	58

La Tabla 24 muestra la comparación de los valores de las métricas PMFP, PMFPr, PMFF, PM y TPM.

Tabla 24.- Valores de las métricas PM antes y después de la refactorización de la aplicación DBLista

Métrica	Valor antes de la refactorización	Valor después de la refactorización
PMFP	**0.009**	**0.085**

PMFPr	0	0
PMFF	**0.244**	**0.179**
PM	0.256	0.256
TPM	**0.026**	**0.072**

El gráfico 1 muestra los valores obtenidos por las métricas PM antes y después de la refactorización de DBLista, se puede observar que la protección aumentó más a nivel privado ("*private*") obteniendo un valor más cercano al 1, por lo que la protección a nivel "*private*" de toda la arquitectura mejoró de **0.9%** a **8.5%,** ya que cinco de las funciones con calificador "*friendly*" cambiaron al calificador "*private*", por eso es que se observa que la protección a nivel "*private*" aumentó y a nivel "*friendly*" disminuyó, dejando a la arquitectura con un nivel de protección "friendly" inferior de un **24.4%** a **17.9%.** La protección a nivel "*protected*" se mantuvo igual con un valor de **0%** ya que DBLista carece de funciones con calificador de alcance "*protected*" antes y después de la refactorización. La protección modular también mantuvo su valor igual ya que las funciones públicas no aumentaron ni disminuyeron (como se muestra en la fórmula de la métrica PM donde el dividendo considera el número de funciones no públicas). En cuanto a la métrica TPM se tuvo un aumento de **4.6%**, pasando de un **2.6%** a un **7.2%**, de esta manera se demuestra que toda la arquitectura aumentó su grado total de protección modular. Esto es debido a que la aplicación DBLista después de la refactorización cuenta con un número mayor de funciones "*private*" el cual es el nivel con mayor protección por lo tanto el nivel de protección de DBLista aumento, así de comprueba que el método de refactorización desarrollado en esta tesis es capaz de aumentar el grado de protección de una aplicación escrita en lenguaje Java.

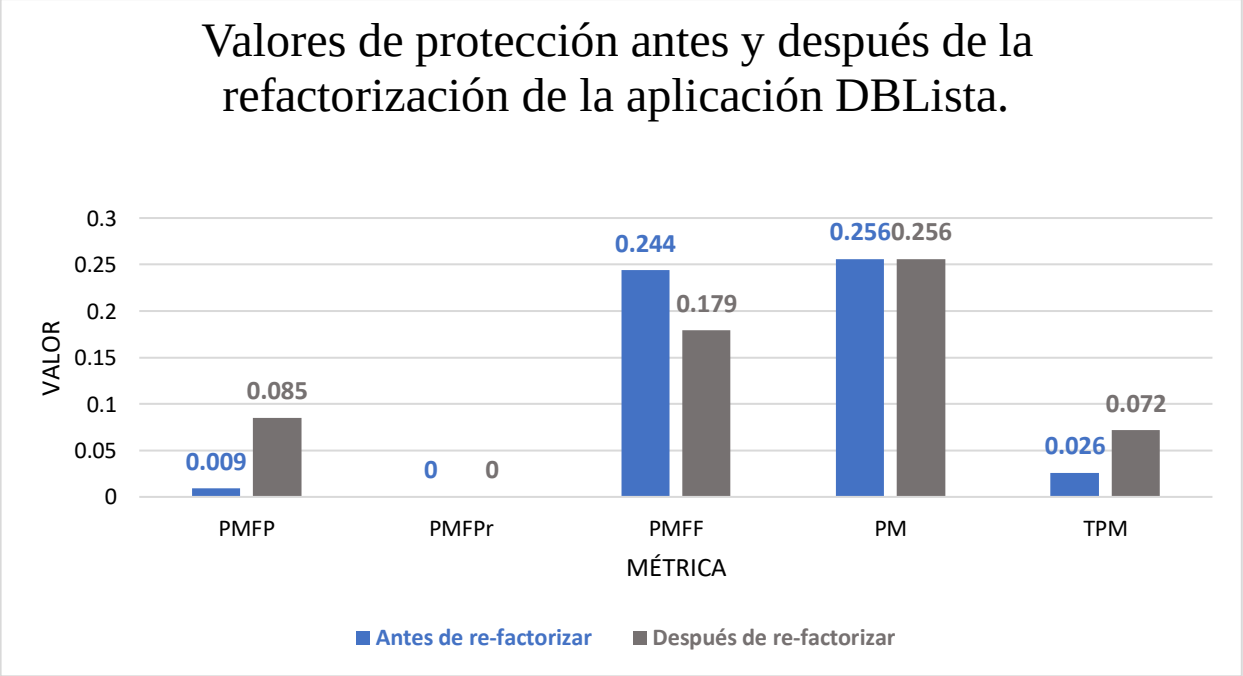


Gráfico 1.- Protección de DBLista antes y después de la refactorización.

Cabe mencionar que el nivel de protección de una aplicación no siempre aumentará, esto depende del manejo de calificadores de alcance que el programador tuvo al desarrollar la aplicación, ya que si cada una de las funciones que conforman la aplicación a refactorizar cuenta con el calificador

de alcance correcto, ninguna de las funciones sufrirá cambios y el nivel protección se mantendrá igual.

6.5.3 Caso de prueba: ISMRCA05 - 03.

Artículos de Prueba: Marco estadístico.

En la Tabla 25 se enlistan las características a probar del proceso de cálculo de métricas de calidad en la aplicación Marco estadístico.

Tabla 25.-Características a probar, del proceso de cálculo de Métricas de calidad en la aplicación Marco estadístico

Característica a Probar
Conteo de funciones *"public".*
Conteo de funciones "*private*".
Conteo de funciones *"protected".*
Conteo de funciones *"friendly".*
Cálculo correcto de las métricas de protección modular (*PMFP*, *PMFPr*, *PMFF*, *PM*, TPM)

Dada la Figura 23 denominada Marco estadístico, que representa un sistema desarrollado en lenguaje java conformado por 47 clases, el cual tiene como objetivo realizar cálculos estadísticos automáticamente. Se procede a realizar el caso de prueba **ISMRCA05 - 03**.

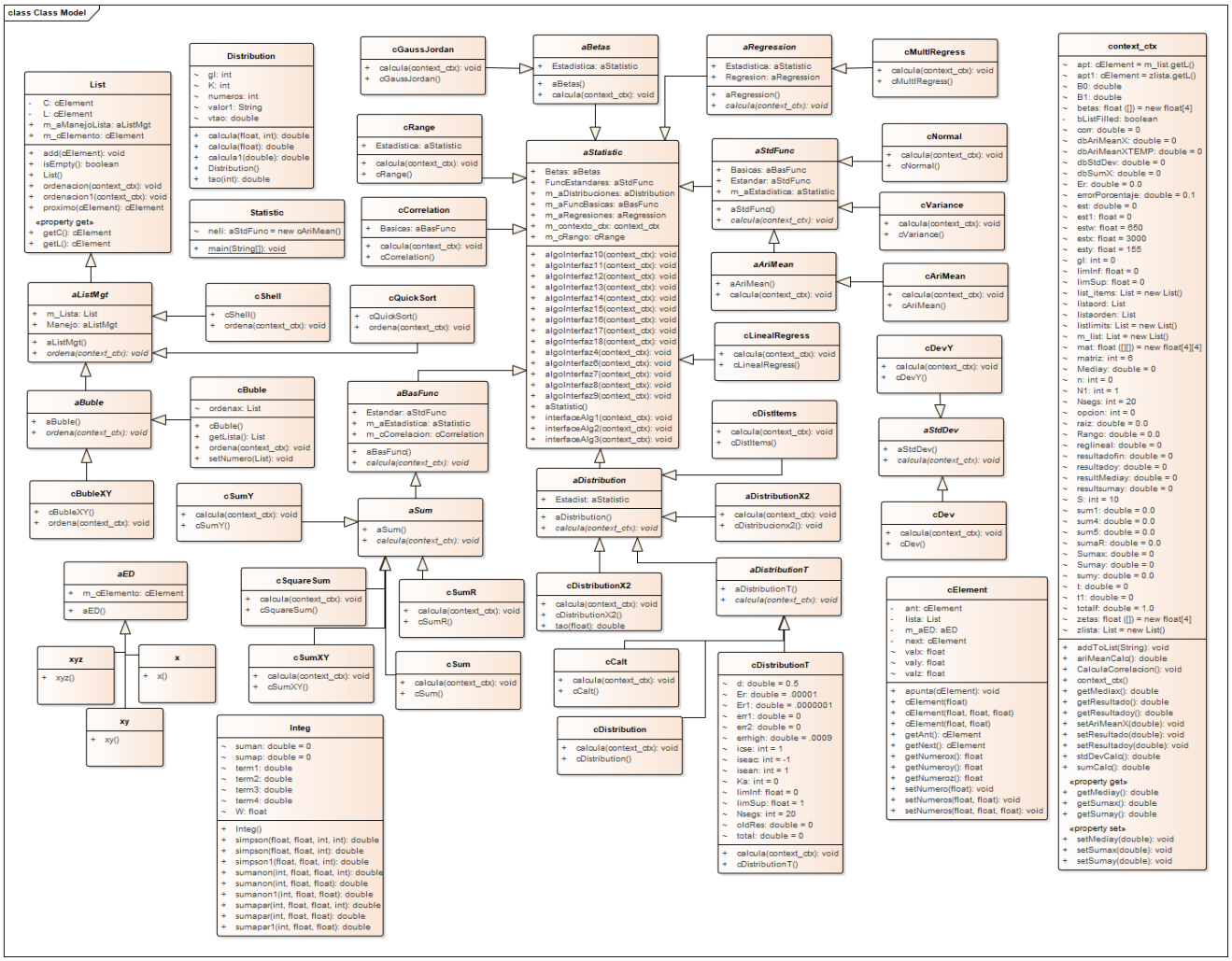


Figura 23.- Arquitectura de clases de la aplicación Marco estadístico antes de la refactorización

Conteo de funciones "public", "private", "protected" y "friendly"

Para comprobar que el conteo de funciones, de acuerdo con el tipo, se realiza de manera correcta, primero se realiza el conteo manual en cada una de las clases de la aplicación del Marco estadístico, posteriormente se realiza un conteo automático para comparar ambos resultados, los cuales deberán ser iguales.

Tabla 26.- Conteo manual y automático de funciones de la aplicación Marco estadístico

Calificador de alcance	Conteo de funciones manual	Conteo de funciones automáticamente
"*public*"	151	No. Métodos public:151 No. Métodos private:0 No. Métodos protected:0 No. Métodos friendly:0
"*private*"	0	
"protected"	0	
"friendly"	0	

En la Figura 23 se puede observar que el total de funciones que conforma la aplicación del Marco estadístico se encuentran declaradas como públicas, de igual forma como se puede observar en la Tabla 26 tanto el conteo manual como el conteo automático reflejaron que las 151 funciones que conforman a la aplicación Marco estadístico son públicas, por lo anterior se comprueba que el conteo de funciones de acuerdo a su tipo se realiza de manera correcta.

Cálculo correcto de las métricas de protección modular (PMFP, PMFPr, PMFF, PM, TPM)

Se somete a la aplicación del Marco estadístico al cálculo de métricas PM para comprobar que el cálculo de métricas de protección modular se realiza de manera correcta, primero se realiza el cálculo manual y posteriormente se realiza un cálculo automático para comparar ambos resultados, los cuales deberán ser iguales.

Se sustituyen los valores de acuerdo a la Figura 23 obteniendo los resultados manual y automático, los cuales de muestran en la Tabla 27.

Tabla 27.- Cálculo manual y automático de métricas PM de la aplicación Marco estadístico

Métrica	Número de ecuación	Cálculo manual	Cálculo automático
PMFP	*EC. 1*	$PMFP = \frac{0}{47} = 0$	Metrica PMFP:0.0
PMFPr	*EC. 2*	$PMFPr = \frac{0}{29} = 0$	Metrica PMFPr:0
PMFF	*EC. 3*	$PMFF = \frac{0}{151} = 0$	Metrica PMFF:0.0
PM	*EC. 4*	$PM = \frac{0}{151} = 0$	Metrica PM:0.0
TPM	*EC. 5*	$TPM = \frac{0+0+0}{3} = \frac{0}{3} = 0$	Metrica TPM:0.0

Por lo anterior demostrado en donde el resultado de cada una de las métricas PM obtenido de manera manual siempre coincidió con el resultado obtenido de manera automática, se comprueba que el cálculo de las métricas se lleva de manera correcta.

6.5.4 Caso de prueba: ISMRCA05 - 04.

Artículos de Prueba: Marco estadístico.

En la Tabla 28 se enlistan las características a probar del proceso de refactorización en la aplicación Marco estadístico.

Tabla 28.- Características a probar del método de refactorización en la aplicación del Marco estadístico

Características a Probar
Cambio del calificador de alcance
Comprobar que el comportamiento se mantiene después de la refactorización
Aumento de protección modular

Cambio del calificador de alcance de las funciones que lo requieren

Para comprobar que el cambio de calificador de alcance a funciones que lo requieren se realiza de manera correcta se realizó un análisis manual del código y de la arquitectura de la aplicación del Marco estadístico. Antes de someter el código a la refactorización, con el fin de identificar si existen funciones que deban cambiar su calificador de alcance, se analiza el alcance desde donde puede ser invocada cada una de las funciones de las clases que conforman la aplicación.

En el análisis manual del código se identificaron funciones que requieren un cambio de calificador, las cuales se enlistan en la Tabla 29, en donde se muestra la clase a la que pertenecen, el calificador de alcance que presentan actualmente y el calificador de alcance al que deben cambiar.

Tabla 29.- Lista de funciones de la aplicación Marco estadístico que requieren cambio de calificador de alcance

Nombre de la función	**Clase a la que pertenece**	**Calificador de alcance actual**	**Calificador de alcance correcto**
Calcula	aBetas	*"public"*	*"protected"*
cDistribucionx2	aDistributionX	*"public"*	*"private"*
interfaceAlg1	aStatistic	*"public"*	*"friendly"*
interfaceAlg2	aStatistic	*"public"*	*"friendly"*
interfaceAlg3	aStatistic	*"public"*	*"friendly"*
algoInterfaz4	aStatistic	*"public"*	*"private"*
algoInterfaz6	aStatistic	*"public"*	*"friendly"*
algoInterfaz7	aStatistic	*"public"*	*"private"*
algoInterfaz8	aStatistic	*"public"*	*"private"*
algoInterfaz9	aStatistic	*"public"*	*"friendly"*
algoInterfaz10	aStatistic	*"public"*	*"friendly"*
algoInterfaz11	aStatistic	*"public"*	*"private"*
algoInterfaz12	aStatistic	*"public"*	*"private"*
algoInterfaz13	aStatistic	*"public"*	*"private"*
algoInterfaz14	aStatistic	*"public"*	*"private"*
algoInterfaz15	aStatistic	*"public"*	*"private"*
algoInterfaz16	aStatistic	*"public"*	*"private"*
algoInterfaz17	aStatistic	*"public"*	*"private"*
algoInterfaz18	aStatistic	*"public"*	*"private"*
setNumero	cBuble	*"public"*	*"friendly"*

getLista	cBuble	*"public"*	*"private"*
calcula	cCorrelation	*"public"*	*"friendly"*
tao	cDistributionX2	*"public"*	*"private"*
setNumero	cElement	*"public"*	*"friendly"*
setNumeros	cElement	*"public"*	*"friendly"*
setNumeros	cElement	*"public"*	*"friendly"*
apunta	cElement	*"public"*	*"friendly"*
getNumerox	cElement	*"public"*	*"friendly"*
getNumeroy	cElement	*"public"*	*"friendly"*
getNumeroz	cElement	*"public"*	*"friendly"*
getNext	cElement	*"public"*	*"friendly"*
getAnt	cElement	*"public"*	*"friendly"*
calcula	cGaussJordan	*"public"*	*"protected"*
calcula	cLinealRegress	*"public"*	*"friendly"*
setResultado	context_ctx	*"public"*	*"friendly"*
setResultadoy	context_ctx	*"public"*	*"friendly"*
setSumax	context_ctx	*"public"*	*"friendly"*
setSumay	context_ctx	*"public"*	*"friendly"*
setAriMeanX	context_ctx	*"public"*	*"friendly"*
setMediay	context_ctx	*"public"*	*"friendly"*
getResultado	context_ctx	*"public"*	*"friendly"*
getResultadoy	context_ctx	*"public"*	*"private"*
getSumax	context_ctx	*"public"*	*"friendly"*
getSumay	context_ctx	*"public"*	*"friendly"*
getMediax	context_ctx	*"public"*	*"friendly"*
getMediay	context_ctx	*"public"*	*"friendly"*
ariMeanCalc	context_ctx	*"public"*	*"private"*
stdDevCalc	context_ctx	*"public"*	*"private"*
sumCalc	context_ctx	*"public"*	*"private"*
CalculaCorrelacion	context_ctx	*"public"*	*"private"*
addToList	context_ctx	*"public"*	*"friendly"*
calcula	cRange	*"public"*	*"friendly"*
calcula	Distribution	*"public"*	*"friendly"*
tao	Distribution	*"public"*	*"private"*
calcula	Distribution	*"public"*	*"friendly"*
calcula1	Distribution	*"public"*	*"friendly"*
simpson	Integ	*"public"*	*"friendly"*
sumapar	Integ	*"public"*	*"private"*
sumanon	Integ	*"public"*	*"private"*
simpson	Integ	*"public"*	*"friendly"*
sumapar	Integ	*"public"*	*"private"*
sumanon	Integ	*"public"*	*"private"*
simpson1	Integ	*"public"*	*"friendly"*
sumapar1	Integ	*"public"*	*"private"*
sumanon1	Integ	*"public"*	*"private"*

ordenacion1	List	*"public"*	*"private"*
ordenacion	List	*"public"*	*"private"*
getL	List	*"public"*	*"friendly"*
getC	List	*"public"*	*"private"*
isEmpty	List	*"public"*	*"private"*
add	List	*"public"*	*"friendly"*
proximo	List	*"public"*	*"friendly"*

En la Figura 24 se muestra la arquitectura de clases de la aplicación Marco estadístico refactorizada, en donde se puede observar que las funciones que requerían cambio de calificador identificadas en el análisis manual, cambiaron de manera satisfactoria al calificador de alcance correcto el cual también se identificó en el análisis manual.

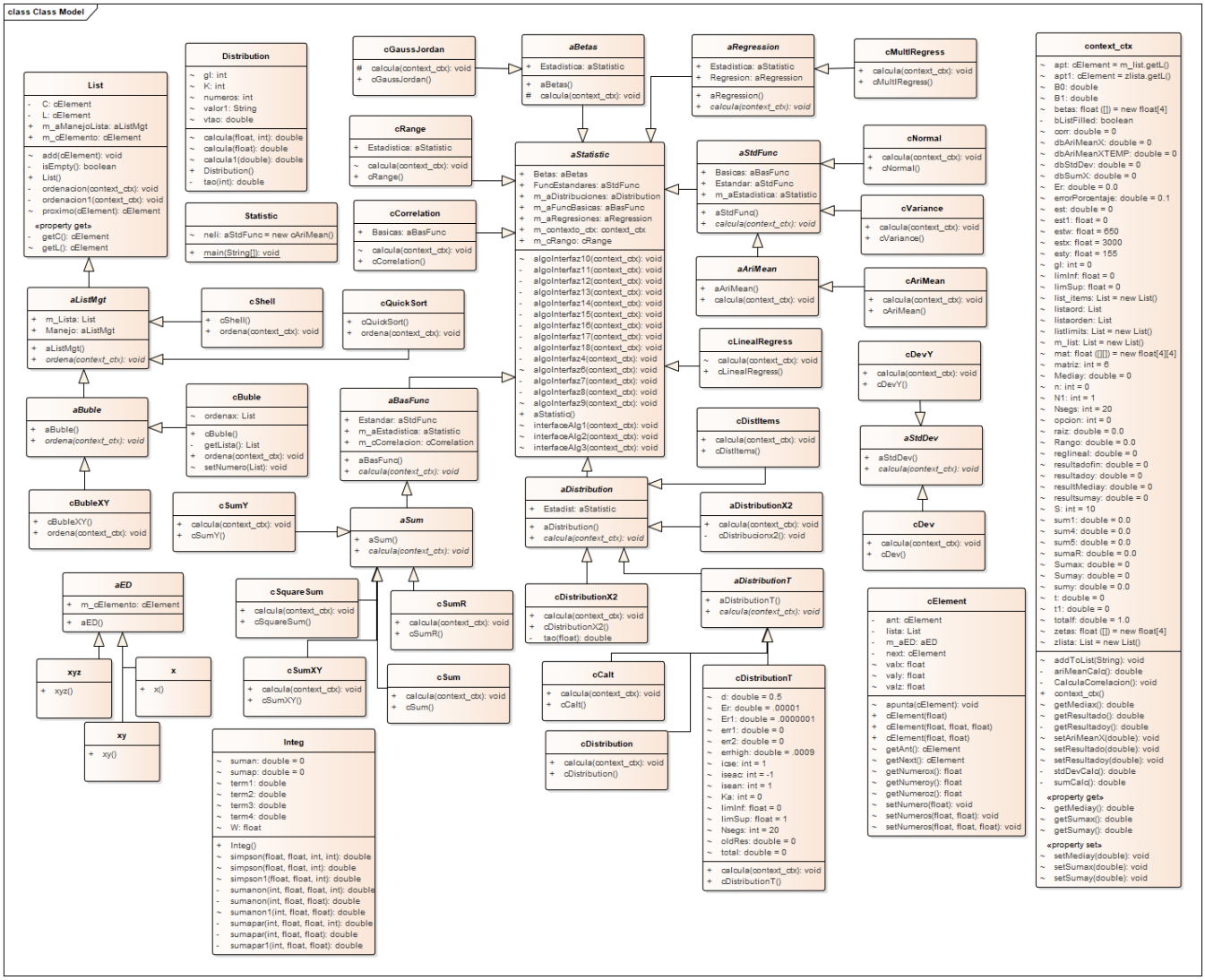


Figura 24.- Arquitectura de clases de la aplicación Marco estadístico después de la refactorización.

En base a la Figura 24 se comprueba que el método de refactorización es capaz de identificar y

cambiar al calificador de alcance correcto a cada función que conforma la aplicación.

Comprobar que el comportamiento se mantiene después de la refactorización

Para comprobar que la aplicación del Marco estadístico mantiene el mismo comportamiento después de ser sometida al método de refactorización se calcula la media de dos conjuntos de números (listas de números) antes y después de la refactorización. La Tabla 30 muestra que números conforman cada conjunto.

Tabla 30.- Lista de datos para el cálculo de la media en la aplicación Marco estadístico.

Conjunto 1	Conjunto 2
10	4
56	22
26	8
27	15
44	78
82	45
36	29
65	74
71	19
6	10

La Tabla 31 muestra los resultados obtenidos por la aplicación Marco estadístico antes y después de la refactorización, se puede observar que el comportamiento se mantuvo. De la misma manera se probaron el resto de las funciones de la aplicación del marco estadístico las cuales se realizaron correctamente, por lo que se comprueba que el método de refactorización no altera el compartimento de la aplicación.

Tabla 31.- Cálculo de la media antes y después de la refactorización de la aplicación Marco estadístico

Resultados antes de la refactorización	Resultado después de la refactorización
Media de la Lista 1: 43.3 Media de la lista 2: 30.4	Media de la Lista 1: 43.3 Media de la lista 2: 30.4

Comprobación del aumento de protección modular en la aplicación Marco estadístico

Como se indicó anteriormente uno de los objetivos principales que se tiene en este trabajo de investigación, es el de aumentar la protección modular de la aplicación que se someta al método de refactorización de calificadores de alcance, cabe mencionar que no siempre se podrá cumplir con este objetivo todo dependerá del uso de los calificadores que haya aplicado el programador en cada aplicación como se mencionó en el caso de prueba ISMRCA05 – 02 del aumento de protección modular en la aplicación del Marco estadístico**.** Para comprobar que la protección de la aplicación Marco estadístico aumento, dicha aplicación se somete al caculo de las métricas PM antes y después de la refactorización, con el fin de comparar su grado de protección de los distintos niveles de protección.

La Tabla 32 muestra el número de funciones de acuerdo al calificador de alcance antes y después de la refactorización de DBLista.

Tabla 32.- Número de funciones por calificador antes y después de la refactorización de Marco estadístico

Calificador	No. Antes de la refactorización	No. después de la refactorización
"private"	0	30
"*protected*"	0	11
"*friendly*"	0	31
"*public*"	151	79

La Tabla 33 muestra la comparación de los valores de las métricas PMFP, PMFPr, PMFF, PM y TPM.

Tabla 33.- Comparación de valores de las métricas PM antes y después de la refactorización de la aplicación Marco estadístico

Métrica	Valor antes de la refactorización	Valor después de la refactorización
PMFP	**0**	**0.07**
PMFPr	**0**	**0.001**
PMFF	**0**	**0.265**
PM	**0**	**0.477**
TPM	**0**	**0.12**

El gráfico 2 muestra los valores obtenidos por las métricas PM antes y después de la refactorización de la aplicación del Marco estadístico, se puede observar que la protección aumentó en todos los niveles obteniendo un valor más cercano al 1 el cual es el mejor valor. La protección a nivel "*private*" aumentó de **0%** a **7%**. La protección a nivel "*protected*" aumentó de **0%** a **0.1%**. La protección a nivel "*friendly*" aumento de **0%** a **20.5%**. La protección modular también aumentó de un **0%** a **47.7%** indicando que la protección modular aumento casi un **50%** ya que 72 funciones de 151 funciones declaradas como públicas cambiaron su alcance. En cuanto a la métrica TPM se tuvo un aumento de **12%**, pasando de un **0%** a un **12%**, de esta manera se demuestra que toda la arquitectura aumentó su grado total de protección modular. Esto es debido a que la aplicación del Marco estadístico después de la refactorización cuenta con un número mayor de funciones no "*public*" por lo tanto el nivel de protección del Marco estadístico aumentó, así se comprueba que el método de refactorización desarrollado en esta tesis es capaz de aumentar el grado de protección de una aplicación escrita en lenguaje Java.

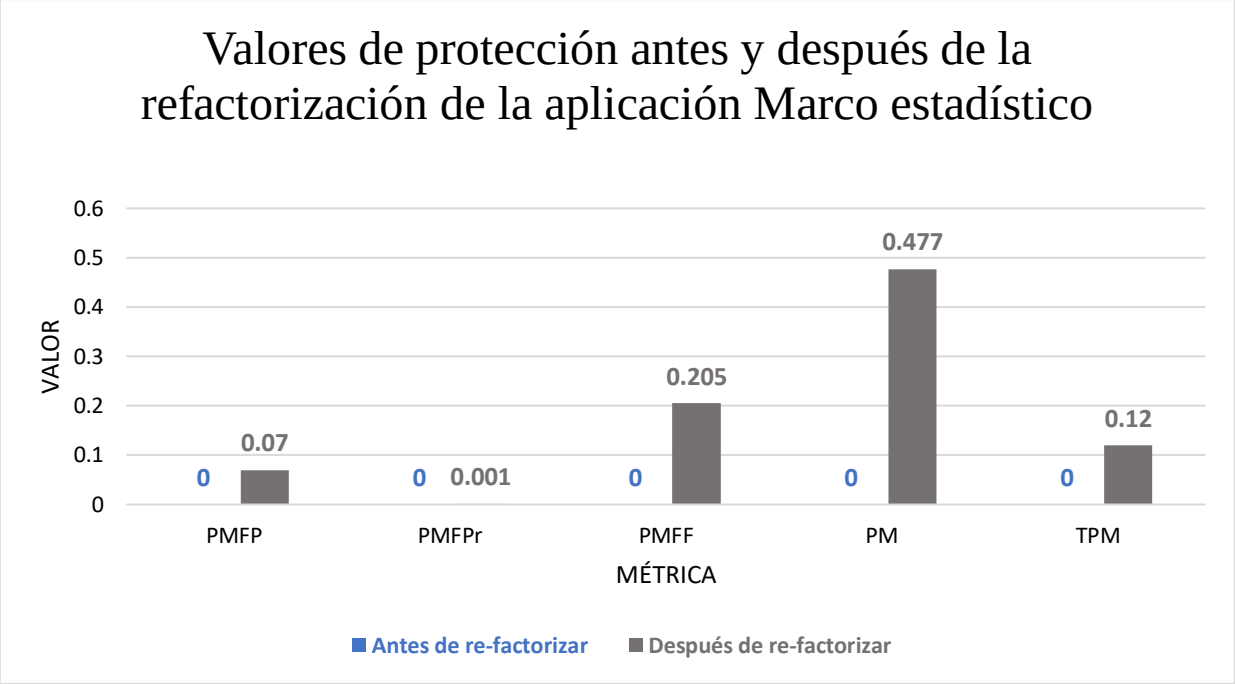


Gráfico 2.- Protección Marco estadístico y después de la refactorización.

6.5.5 Caso de prueba: ISMRCA05 - 05.

Artículos de Prueba: PSP Cenidet.

En la Tabla 34 se enlistan las características a probar del proceso de cálculo de métricas de calidad en la aplicación PSP Cenidet.

Tabla 34.- Características a probar, del proceso de cálculo de Métricas de calidad en la aplicación PSP Cenidet

Característica a Probar
Conteo de funciones *"public".*
Conteo de funciones "*private*".
Conteo de funciones *"protected".*
Conteo de funciones *"friendly".*
Cálculo correcto de las métricas de protección modular (*PMFP*, *PMFPr*, *PMFF*, PM, TPM)

.

Dada la Figura 25 denominada PSP Cenidet, que representa un sistema desarrollado en lenguaje java conformado por 52 clases, el cual tiene como objetivo medir los tiempos que una persona utiliza en realizar tareas específicas. Se procede a realizar el caso de prueba **ISMRCA05 - 05**.

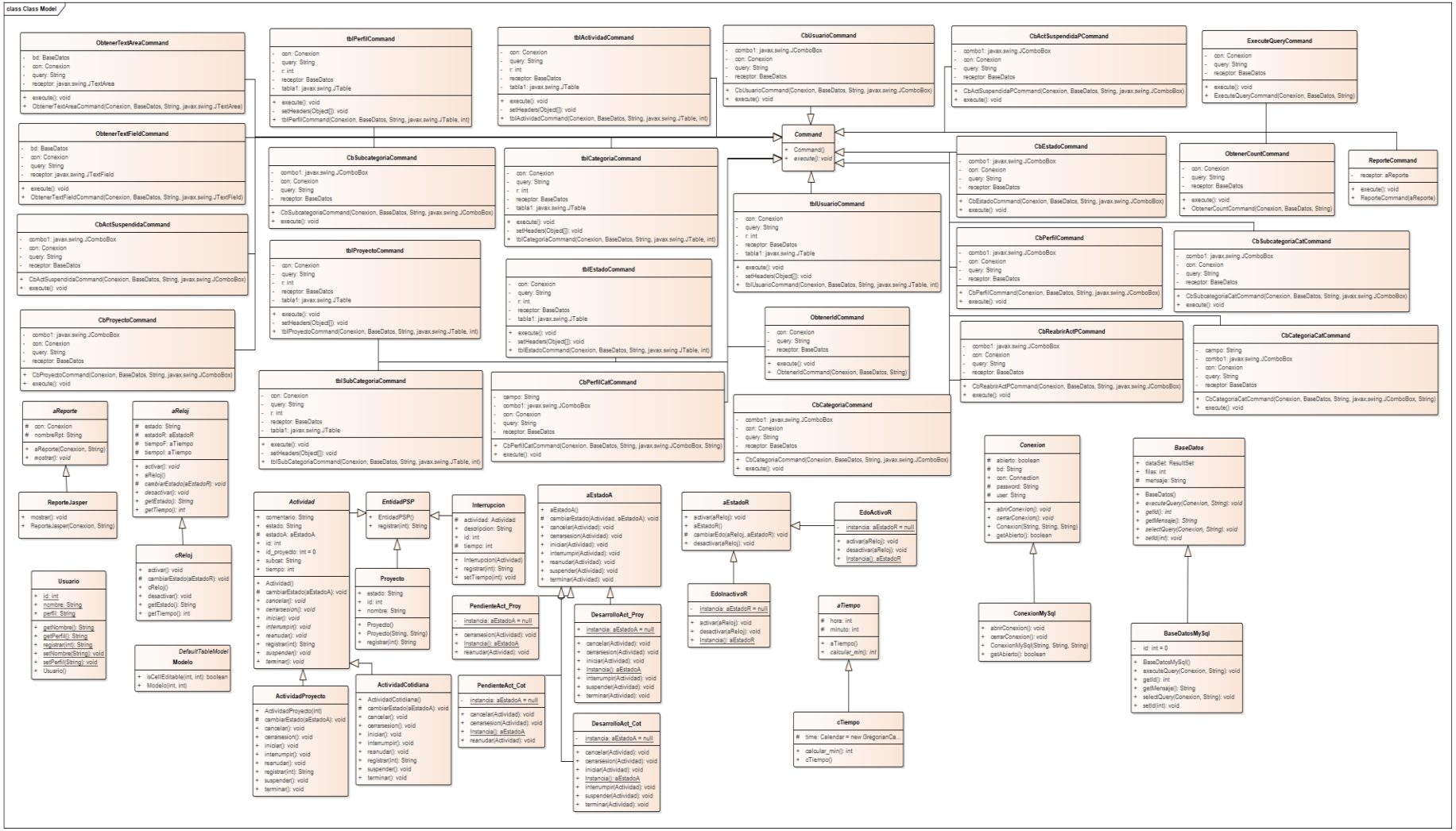


Figura 25.- Arquitectura de clases de la aplicación PSP Cenidet

Conteo de funciones "*public*", "*private*", "*protected*" y "*friendly*"

Para comprobar que el conteo de funciones, de acuerdo con el tipo, se realiza de manera correcta, primero se realiza el conteo manual en cada una de las clases de la aplicación PSP Cenidet posteriormente se realiza un conteo automático para comparar ambos resultados, los cuales deberán ser iguales.

Tabla 35.- Conteo manual y automático de funciones de la aplicación PSP Cenidet

Calificador de alcance	Conteo de funciones manual	Conteo de funciones automáticamente
"public"	171	No. Métodos public:171 No. Métodos private:7 No. Métodos protected:7 No. Métodos friendly:0
"private"	7	
"protected"	7	
"Friendly"	0	

En la Figura 25 se puede observar que la mayoría de las funciones que conforman la aplicación PSP Cenidet se encuentran declaradas como públicas, así mismo la Tabla 35 muestra que en el conteo manual y el conteo automático se obtuvieron los mismos valores, por lo que se comprueba que el conteo de funciones de acuerdo a su tipo de se lleva acabo de manera correcta.

Cálculo correcto de las métricas de protección modular (PMFP, PMFPr, PMFF, PM, TPM)

Para comprobar que el cálculo de métricas de protección modular se realiza de manera correcta, primero se realiza el cálculo manual y posteriormente se realiza un cálculo automático para comparar ambos resultados, los cuales deberán ser iguales.

Se sustituyen los valores de acuerdo a la Figura 25 obteniendo los resultados manual y automático, los cuales de muestran en la Tabla 36.

Tabla 36.- Cálculo manual y automático de métricas PM de la aplicación PSP Cenidet

Métrica	Número de ecuación	Cálculo manual	Cálculo automático
PMFP	*EC. 1*	$PMFP = \frac{2.3333}{52} = 0.045$	Metrica PMFP:0.045
PMFPr	*EC. 2*	$PMFPr = \frac{0.9194}{40} = 0.022$	Metrica PMFPr:0.022
PMFF	*EC. 3*	$PMFF = \frac{0}{185} = 0$	Metrica PMFF:0.0

PM	*EC. 4*	$PM = \frac{14}{185} = 0.076$	Metrica PM:0.076
TPM	*EC. 5*	$TPM = \frac{0.0165}{3} = 0.036$	Metrica TPM:0.036

Por lo anterior demostrado en donde el resultado de cada una de las métricas PM obtenido de manera manual siempre coincidió con el resultado obtenido de manera automática, se comprueba que el cálculo automático de las métricas PM se realiza de manera correcta.

6.5.6 Caso de prueba: ISMRCA05 - 06.

Artículos de Prueba: PSP Cenidet.

En la Tabla 37 se enlistan las características a probar del proceso de refactorización en la aplicación PSP Cenidet.

Tabla 37.- Características a probar del método de refactorización en la aplicación PSP Cenidet

Característica a Probar
Cambio del calificador de alcance
Comprobar que el comportamiento se mantiene después de la refactorización
Aumento de protección modular

Cambio del calificador de alcance de las funciones que lo requieren de la aplicación PSP Cenidet

Para comprobar que el cambio de calificador de alcance a funciones que lo requieren se realiza de manera correcta se realizó un análisis manual del código y de la arquitectura de la aplicación PSP Cenidet. Antes de someter el código a la refactorización, con el fin de identificar si existen funciones que deban cambiar su calificador de alcance, se analiza el alcance desde donde puede ser invocada cada una de las funciones de las clases que conforman la aplicación. En el análisis manual del código se identificaron funciones que requieren un cambio de calificador, las cuales se enlistan en la Tabla 38, en donde de muestra la clase a la que pertenecen, el calificador de alcance que presentan actualmente y el calificador de alcance al que deben cambiar.

Tabla 38.- Lista de funciones de la aplicación PSP Cenidet que requieren cambio de calificador de alcance.

Nombre de la función	Clase a la que pertenece	Calificador de alcance actual	Calificador de alcance correcto
Instancia	PendienteAct_Proy	*"public"*	*"friendly"*
cerrarsesion	PendienteAct_Proy	*"public"*	*"protected"*
reanudar	PendienteAct_Proy	*"public"*	*"protected"*
Instancia	PendienteAct_Cot	*"public"*	*"friendly"*

cancelar	PendienteAct_Cot	*"public"*	*"protected"*
cerrarsesion	PendienteAct_Cot	*"public"*	*"protected"*
reanudar	PendienteAct_Cot	*"public"*	*"protected"*
getPerfil	Usuario	*"public"*	*"private"*
Instancia	EdoInactivoR	*"public"*	*"friendly"*
activar	EdoInactivoR	*"public"*	*"protected"*
desactivar	EdoInactivoR	*"public"*	*"protected"*
Instancia	EdoActivoR	*"public"*	*"friendly"*
activar	EdoActivoR	*"public"*	*"protected"*
desactivar	EdoActivoR	*"public"*	*"protected"*
Instancia	DesarrolloAct_Proy	*"public"*	*"friendly"*
cancelar	DesarrolloAct_Proy	*"public"*	*"protected"*
cerrarsesion	DesarrolloAct_Proy	*"public"*	*"protected"*
iniciar	DesarrolloAct_Proy	*"public"*	*"protected"*
interrumpir	DesarrolloAct_Proy	*"public"*	*"protected"*
suspender	DesarrolloAct_Proy	*"public"*	*"protected"*
terminar	DesarrolloAct_Proy	*"public"*	*"protected"*
Instancia	DesarrolloAct_Cot	*"public"*	*"friendly"*
iniciar	DesarrolloAct_Cot	*"public"*	*"protected"*
cancelar	DesarrolloAct_Cot	*"public"*	*"protected"*
cerrarsesion	DesarrolloAct_Cot	*"public"*	*"protected"*
interrumpir	DesarrolloAct_Cot	*"public"*	*"protected"*
suspender	DesarrolloAct_Cot	*"public"*	*"protected"*
terminar	DesarrolloAct_Cot	*"public"*	*"protected"*
cambiarEstado	cReloj	*"protected"*	*"public"*
getAbierto	ConexionMySql	*"public"*	*"protected"*
getAbierto	Conexion	*"public"*	*"protected"*
cambiarEstado	aReloj	*"protected"*	*"public"*
cancelar	aEstadoA	*"public"*	*"protected"*
cerrarsesion	aEstadoA	*"public"*	*"protected"*
iniciar	aEstadoA	*"public"*	*"protected"*
interrumpir	aEstadoA	*"public"*	*"protected"*
reanudar	aEstadoA	*"public"*	*"protected"*
suspender	aEstadoA	*"public"*	*"protected"*
terminar	aEstadoA	*"public"*	*"protected"*

En la Figura 26 se muestra la arquitectura de clases de la aplicación PSP Cenidet refactorizada, en donde se puede observar que las funciones que requerían cambio de calificador identificados en el análisis manual, cambiaron de manera satisfactoria al calificador de alcance correcto el cual también se identificó en el análisis manual.

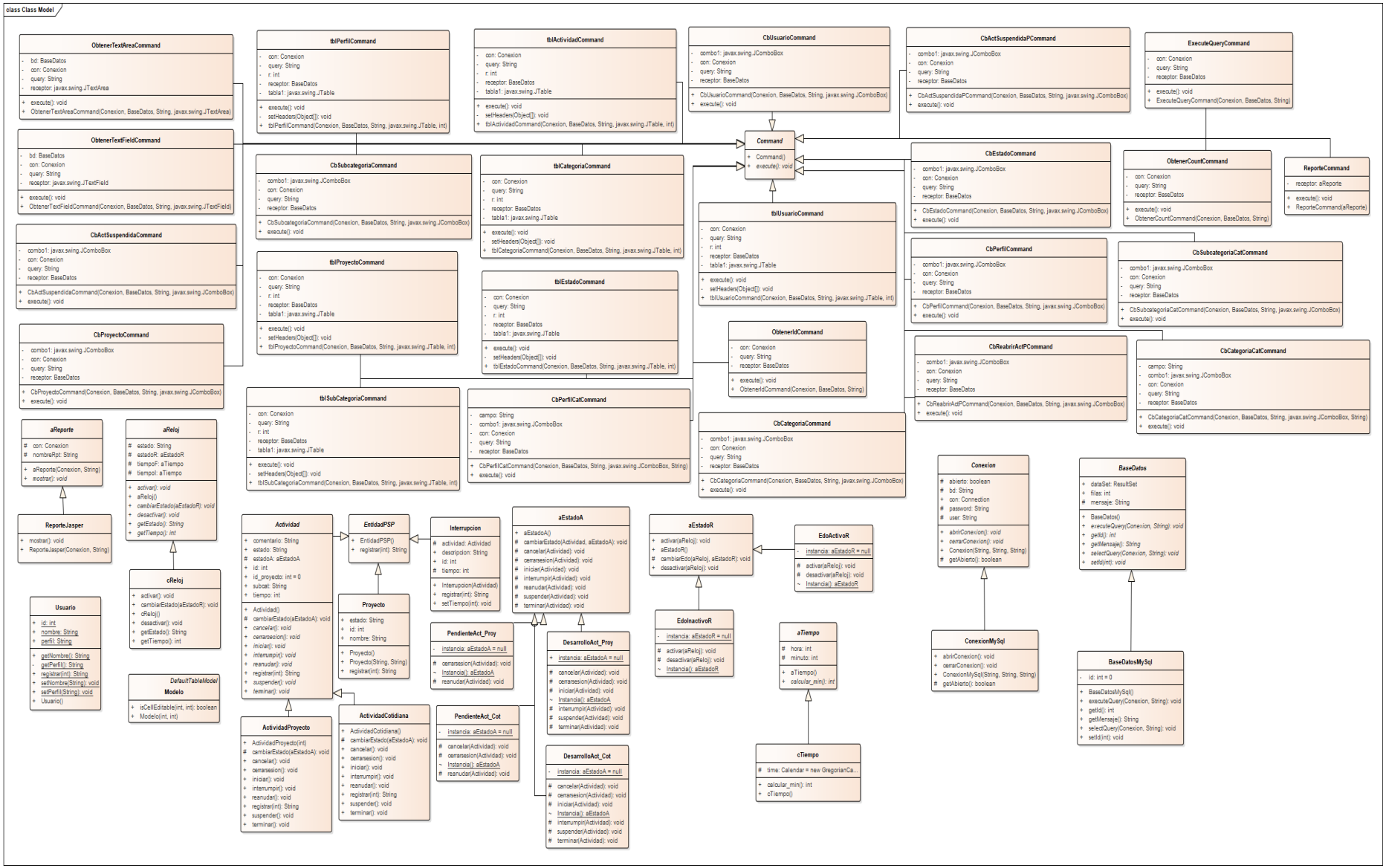


Figura 26.- Arquitectura de clases de la aplicación PSP Cenidet después de la refactorización

En la Figura 26 se puede observar como las funciones cambiaron su calificador de alcance al correcto como anteriormente se había establecido, por lo que se comprueba que el método de refactorización es capaz de identificar y cambiar al calificador de alcance correcto a cada función que conforman a la aplicación.

Comprobar que el comportamiento se mantiene después de la refactorización de la aplicación PSP Cenidet.

Para comprobar que la aplicación PSP Cenidet mantiene el mismo comportamiento después de ser sometida al método de refactorización se agrega un usuario antes y después de la refactorización para verificar el comportamiento de la aplicación. La figura 27 muestra los usuarios registrados en PSP Cenidet antes de la refactorización, se puede observar que solo se cuenta con dos usuarios registrados.

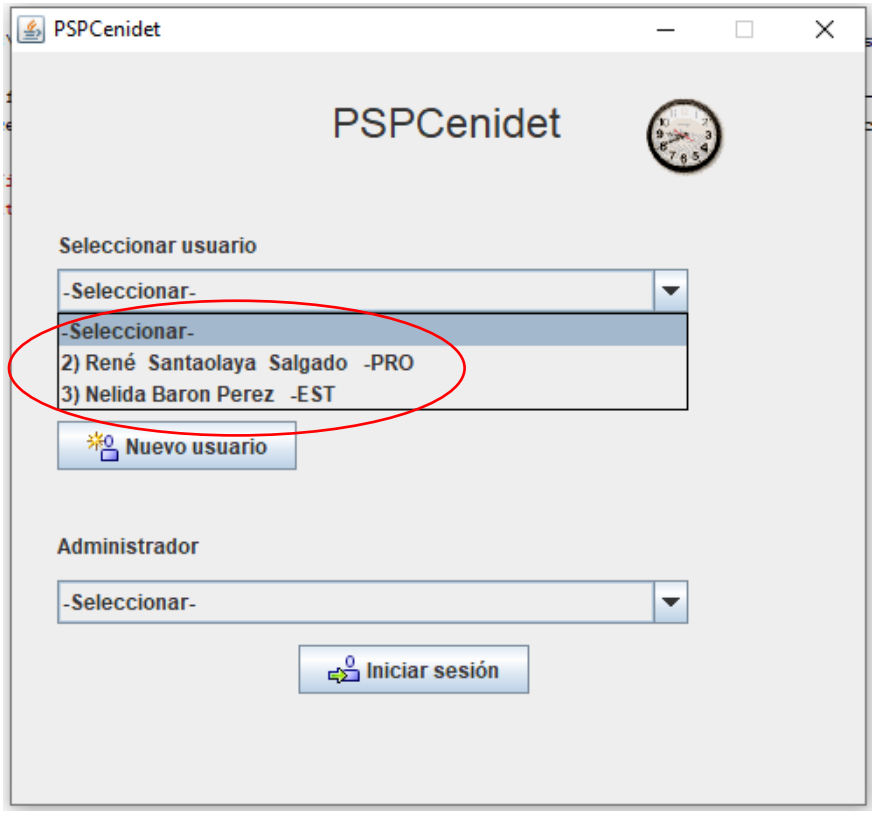


Figura 27.- Usuarios registrados en PSP Cenidet antes de la refactorización

Se registra un usuario más como se muestra en la Figura 28.

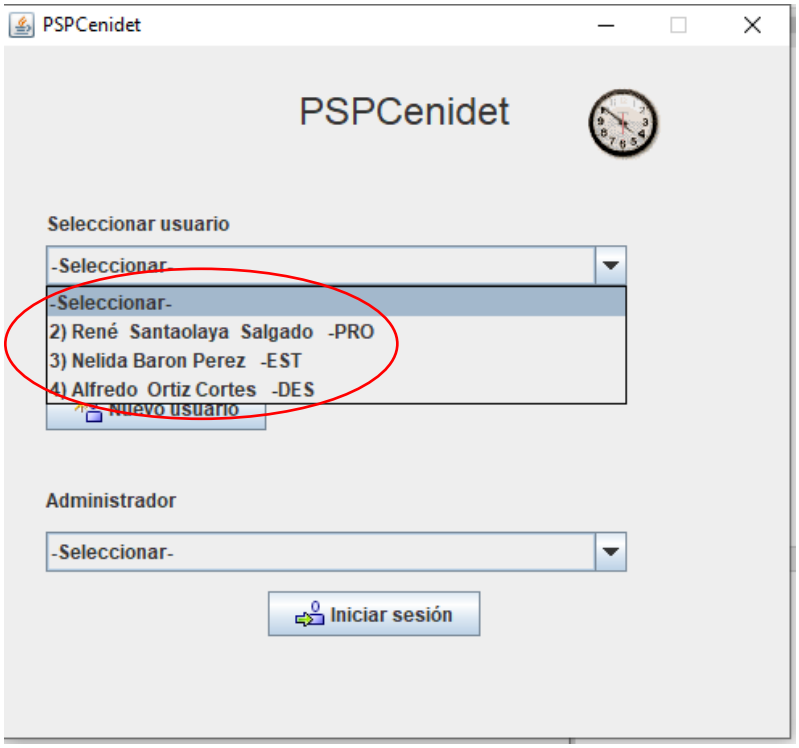


Figura 28.- Registro de un usuario nuevo en PSP Cenidet.

Se agrego un nuevo usuario después de la refactorización el cual se registró de manera exitosa como se muestra en la Figura 29, de la misma manera se probaron tres funciones más de la aplicación PSP Cenidet las cuales se realizaron de manera exitosa, por lo que se comprueba que el comportamiento de la aplicación se mantiene igual después de la refactorización

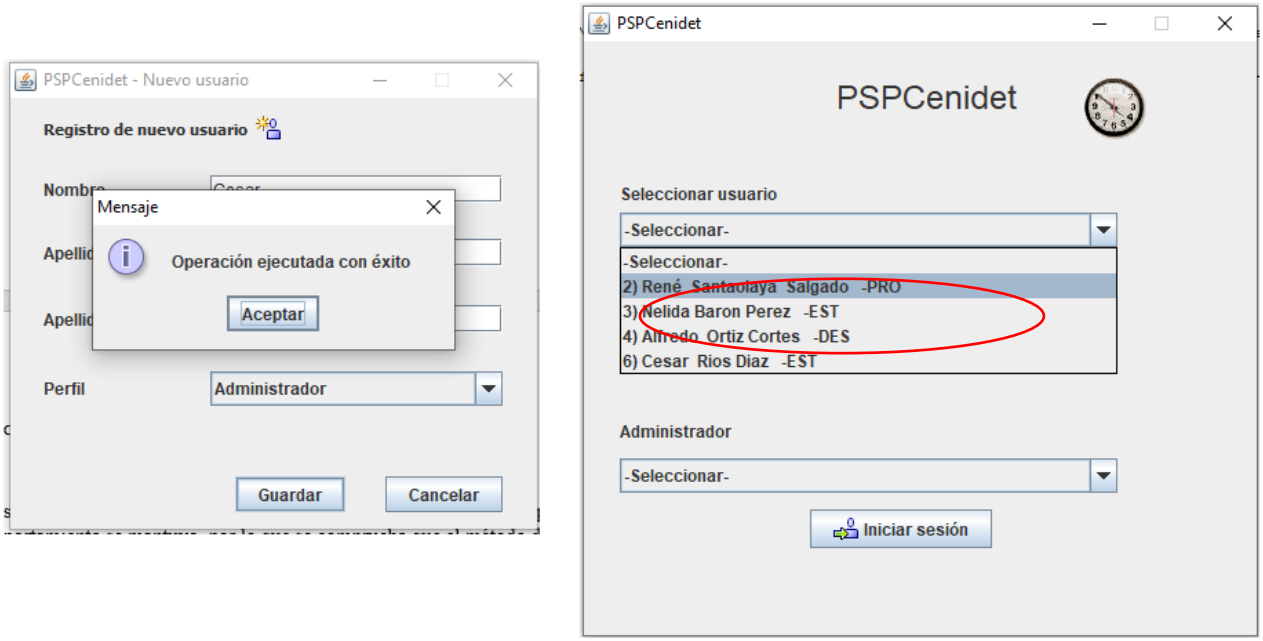


Figura 29.- Registro de un nuevo usuario en PSP Cenidet después de la refactorización.

En base a los resultados obtenidos por la aplicación Marco estadístico antes y después de la refactorización, se puede observar que el comportamiento se mantuvo, por lo que se comprueba que el método de refactorización no altera el compartimento de la aplicación.

Comprobación del aumento de protección modular en la aplicación Marco estadístico

Para comprobar que la protección de la aplicación PSP Cenidet aumento, dicha aplicación se somete al caculo de las métricas PM antes y después de la refactorización, con el fin de comparar su grado de protección de los distintos niveles de protección.

La Tabla 39 muestra el número de funciones de acuerdo al calificador de alcance antes y después de la refactorización de PSP Cenidet.

Tabla 39.- Número de funciones por calificador antes y después de la refactorización de PSP Cenidet

Calificador	No. Antes de la refactorización	No. después de la refactorización
"private"	7	13
"*protected*"	7	43
"*friendly*"	0	6
"*public*"	171	123

En la Tabla 40 se muestra la comparación de los valores de las métricas PMFP, PMFPr, PMFF, PM y TPM.

Tabla 40.- Comparación de valores de las métricas PM antes y después de la refactorización de la aplicación PSP Cenidet.

Métrica	Valor antes de la refactorización	Valor después de la refactorización
PMFP	**0.045**	**0.048**
PMFPr	**0.022**	**0.132**
PMFF	**0**	**0.032**
PM	**0.076**	**0.276**
TPM	**0.036**	**0.068**

El gráfico 3 muestra los valores obtenidos por las métricas PM antes y después de la refactorización de la aplicación PSP Cenidet, se puede observar que la protección aumentó en todos los niveles obteniendo un valor más cercano al 1. La protección a nivel "*private*" de toda la arquitectura mejoró de **4.5%** a **4.8%**. La protección a nivel "*protected*" de toda la arquitectura mejoró de **2.2%** a **13.2%** siendo el nivel en donde más aumento la protección**.** La protección a nivel "*friendly*" de toda la arquitectura mejoró de **0%** a **3.2%**. La protección modular también aumento de un **7.6%** a **27.6%**. En cuanto a la total protección modular se puede observar que aumentó un **3.2%**, pasando de un **3.6%** a un **6.8%,** de esta manera se demuestra que toda la arquitectura aumentó su grado total de protección modular, así se comprueba que el método de

refactorización desarrollado en esta tesis es capaz de aumentar el grado de protección de una aplicación escrita en lenguaje Java.

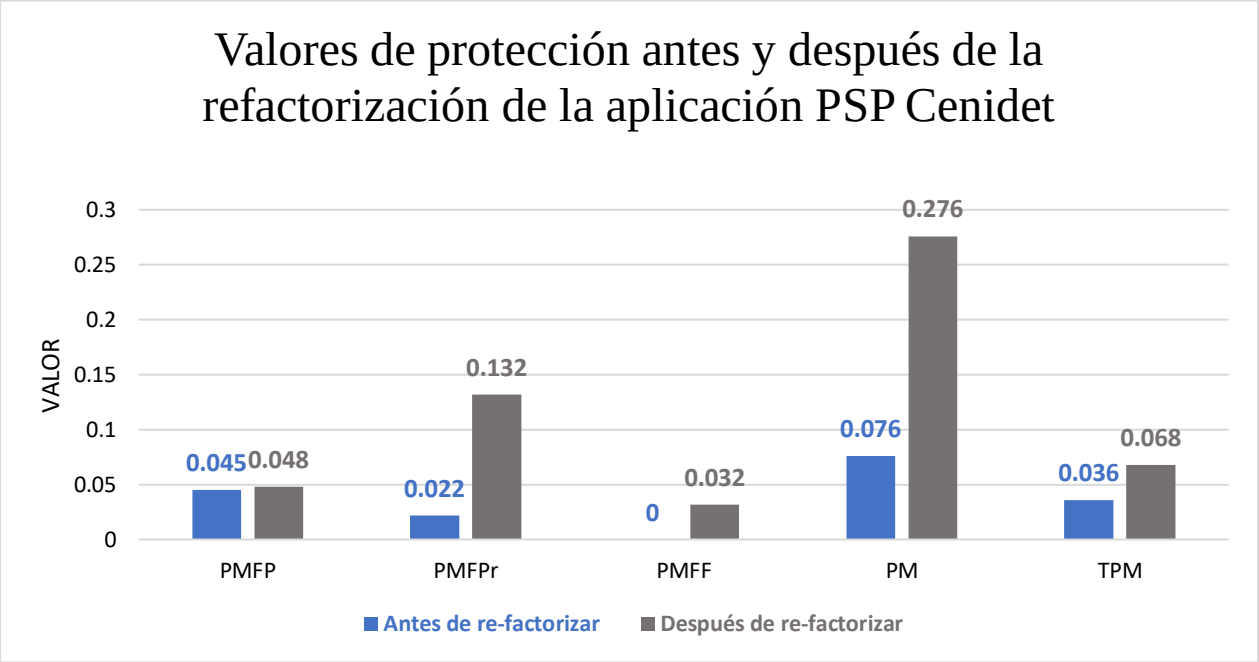


Gráfico 3.- Protección PSP Cenidet antes y después de la refactorización.

Capítulo 7.- Conclusiones y trabajos futuros

7.1 Conclusiones

El uso adecuado de los calificadores de alcance y principio de ocultamiento de información garantiza un correcto nivel de protección y encapsulamiento, permitiendo que la información declarada como privada sólo pueda ser accedida por el interior de los objetos instanciados, y no por cualquier otro agente externo. Adicionalmente, el encapsulamiento permite empaquetar el funcionamiento interno de un objeto, permitiendo modificarlo sin afectar el funcionamiento de otros objetos aun cuando sean del mismo tipo.

Dentro de los aprendizajes obtenidos durante el desarrollo de esta investigación se encuentran:

- El uso correcto de los diferentes niveles de acceso en el lenguaje Java (*"private", "protected", "fiendly" y "public"*), ya que 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 objetos y 2) resultados falsos o incorrectos.

- Que los niveles de acceso son una parte fundamental para lograr un mayor nivel de encapsulamiento, asegurando que el contenido de la información de un objeto se encuentra seguro del mundo exterior.

- Con base en las pruebas realizadas se observa que el uso de funciones públicas predomina y aunque este tipo de diseño no provoque errores sintácticos, las funciones tienden a presentar un bajo grado de protección modular, por lo que pueden redundar en un incorrecto nivel de encapsulamiento.

- El desarrollo de este sistema facilita la medición del grado de protección modular, así mismo la mejora cuando un sistema cuenta con un bajo grado de protección modular lo que favorece: a) al encapsulamiento, lo que fomenta robustez del sistema; b) la flexibilidad y genericidad, lo que fomenta el reuso de software; c) la autonomía modular, lo que fomenta 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.

Se demostró que el método de refactorización de calificadores de alcance mejora aspectos de diseño correspondientes a los niveles de acceso, lo que puede llevar a disminuir la deuda técnica que se produce al ignorar la protección modular e ignorar el principio de ocultamiento de datos en base a las reglas de visibilidad, logrando aumentar la protección modular de un sistema escrito en lenguaje Java. Así se demostró que el método de refactorización ayuda a aplicar de manera correcta el principio de ocultamiento de la información, previendo la manipulación inadvertida de los detalles internos de las entidades de software desde 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.

- Que algunas de las palabras reservadas no se encuentren implementadas en la plantilla "*Stringtemplate"*.

Si se llegase a cambiar la versión de la gramática de Java, se tendría que verificar si la nueva versión tiene variaciones en las reglas gramaticales usadas para la obtención de la información y adaptarlas a los cambios. Las reglas gramaticales utilizadas por el método de refactorización de calificadores de alcance son: *"enterPackageDeclaration", "enterImportDeclaration", "enterTypeDeclaration","enterClassDeclaration","enterTypeParameters","enterInterfaceDeclaration", "exitClassBody", "exitInterfaceDeclaration", "enterClassBody", "enterMethodDeclaration", "exitMethodBody", "enterConstructorDeclaration", "exitConstructorDeclaration", "enterTypeType", "enterInterfaceMethodDeclaration", "enterVariableDeclarator", "enterClassOrInterfaceType", "enterAnnotation", "enterLocalVariableDeclaration" y "enterLocalTypeDeclaration".*

El uso de los métodos del SR2-Refactoring deben ser aplicados en orden secuencial como se muestra en la Figura 30. Es decir, el orden de aplicación de los métodos de refactorización debe ser aplicado de la siguiente manera: 1) Método de refactorización de calificadores de alcance, 2) Método para Aumentar la flexibilidad en sistemas Carentes, 3) Método de reducción de herencia de implementación, 4) Método para la Mejora de la Modularidad y Método para Reducir las Dependencias Entre Clases y 5) Método de separación de interfaces.

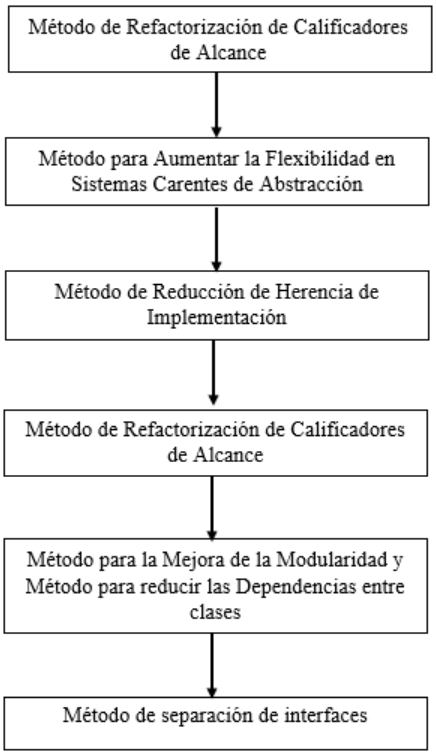


Figura 30.- Orden secuencial de los métodos de refactorización

En base a las pruebas realizadas se puede concluir que el método de refactorización desarrollado en esta investigación, cumple con el objetivo general de esta tesis, el cual consiste de la mejora del diseño de arquitecturas de software legado orientado a objetos, a través de la aplicación de la protección modular para el ocultamiento de información, utilizando las reglas de visibilidad que soporta el lenguaje de programación Java. El método de refactorización de calificadores de alcance fue desarrollado para ser aplicado en sistemas programados en lenguaje Java para colocar el calificador de alcance correcto a cada una de las funciones que conforman un sistema de software. El método se implementó en el sistema SR2-Refactoring, extendiendo su funcionalidad, por lo cual, se cumplen con los objetivos específicos de esta tesis, los cuales son: a) Mejorar la Modularidad del Software Legado escrito en lenguaje Java, como se demuestra con los valores obtenidos de las métricas PMFP, PMFPr, PMFF, PM y TMP en las pruebas que aplicaron tanto al código original como al código refactorizado; b) Mejorar el encapsulamiento de objetos del Software Legado escrito en lenguaje Java, al igual que en el inciso a), esto se demuestra con la aplicación de las métricas PMFP, PMFPr, PMFF, PM y TMP; c) Facilitar el mantenimiento del Software Legado escrito en lenguaje Java, al quedar protegidas de la manipulación externa, las entidades de software pierden fragilidad, es decir, los cambios o defectos en un módulo solamente se confinan a ese módulo, esto facilita el mantenimiento ante defectos y cambios o aumento de requisitos; d) Habilitar el reuso de componentes del Software Legado escrito en lenguaje Java, la mejora de modularidad y encapsulamiento origina autonomía de las entidades de software lo cual se refleja en la facilidad para su reuso; y e) Extender la funcionalidad del "SR2-Refactoring", para dar soporte a sus métodos de refactorización de alto impacto en código escrito en lenguaje Java, la extensión al SR2-Refactoring consiste en agregar el método de refactorización de calificadores de alcance a los métodos que ya soporta este sistema.

7.2.- Aportaciones de la tesis

7.2.1.- Método de refactorización de calificadores de alcance.

- El método es capaz de aumentar la protección modular de sistemas escritos en lenguaje Java, mediante la colocación de calificadores de alcance 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 en los diferentes niveles de visibilidad.

1. PMFP (Protección Modular de Funciones Privadas)
2. PMFPr (Protección Modular de Funciones Protegidas)
3. PMFF (Protección Modular de Funciones Friendly)
4. PM (Protección Modular)
5. TPM (Total Protección Modular)

7.2.3.- Extensión a la herramienta de refactorización denominada SR2-Refactoring

- Se integró al SR2-Refactoring, el método de calificadores de alcance para refactorizar código en lenguaje java, implementando el analizador léxico y sintáctico en el metalenguaje ANTLR.

- Además, se integró al marco orientado objetos para obtener servicios de cálculo de métricas un conjunto de métricas PM (PMFP, PMFPr, PMFF, PM y TPM).

7.3.- Trabajo a Futuro

7.3.1.- Identificación del patrón de diseño *Témplate Method*

Actualmente el método de refactorización de calificadores de alcance no cuenta con la capacidad para identificar la presencia de funciones de plantilla que utilizan el patrón de diseño "*Template Method".*

Todas las funciones plantilla que implementan el patrón de diseño "Template Method", en la clase base, deben ser declaradas con el calificador "public", mientras que todas las funciones declaradas en la misma clase que implementan el código no variante deben ser declaradas con el calificador "private", para asegurar que sólo la función de plantilla las invoque. Las funciones gancho que representan al código variante que son implementadas en las clases derivadas, deben ser declaradas con calificador de alcance "protected", para asegurar que sólo la función plantilla los pueda invocar.

Anexo A.- Sustento de las métricas PMFP, PMFPr, PMFF, PM y TPM como escalas ordinales

Factor de Protección Modular (*PMFP*) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1.- P es el conjunto de funciones con calificador de alcance "*private*".
2.- •≥ es una relación empírica entre el número de funciones privadas la cual describe si existe menor o mayor grado de protección de la información con respecto al calificador de alcance "*private*".
3.- ℟ es el conjunto de números reales.
4.- ≥ "mayor o igual que" es una relación binaria entre números.

Entonces la métrica PMFP será considerada como una escala ordinal si:

1. La relación binaria •≥, es de orden débil
2. P1 •≥ P2 => PMFP (Arquitectura C) ≥ PMFP (Arquitectura B).

Comprobación

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (1), para realizar el cálculo de la PMFP de las arquitecturas A, B y C, mostradas en las Figuras 31, 32 y 33 respectivamente.

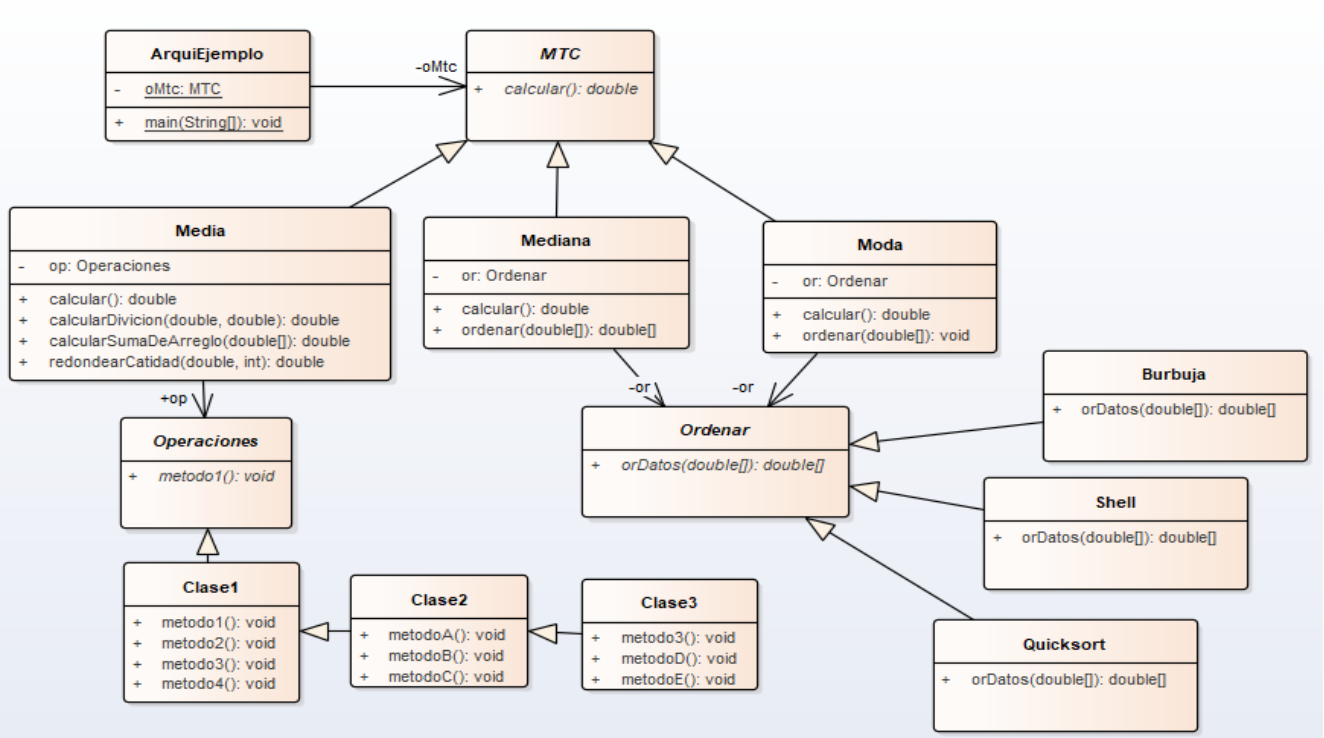


Figura 31.- Diagrama de clases de la arquitectura (A) de un sistema

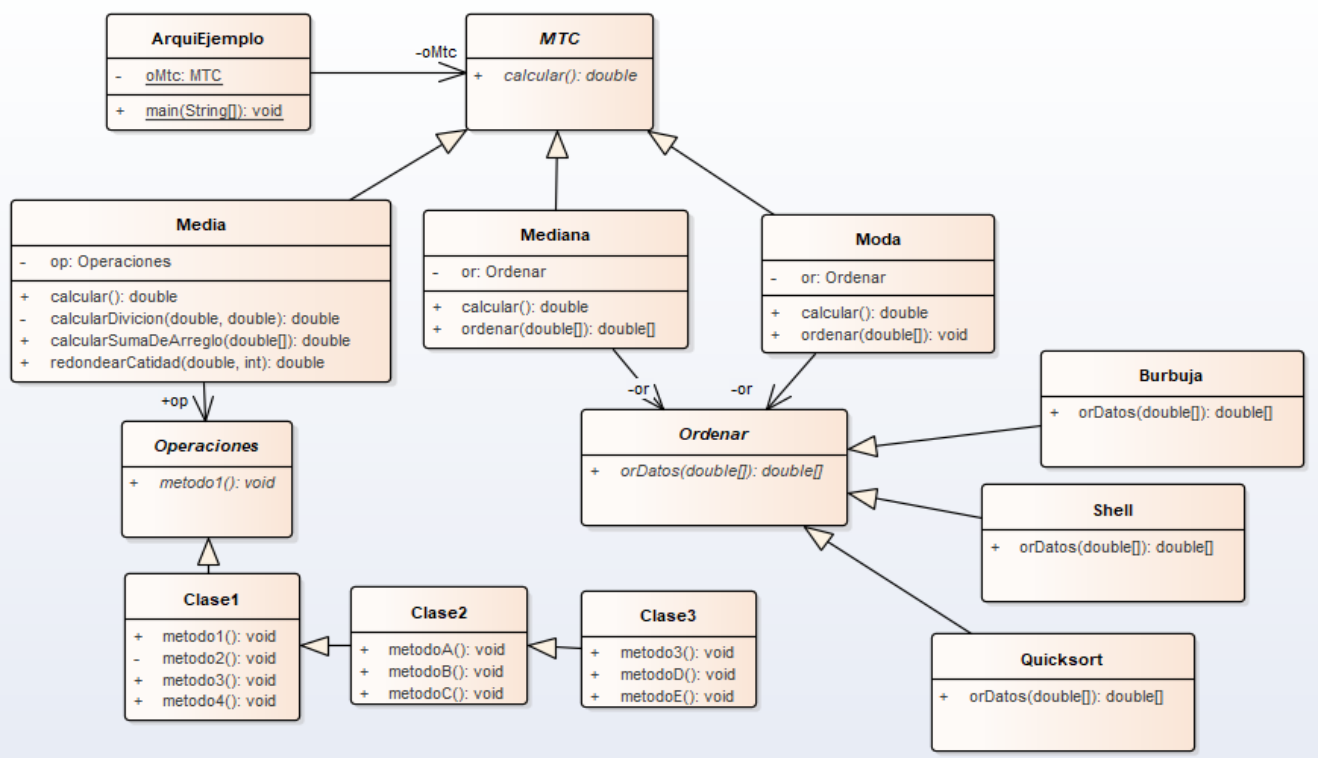


Figura 32.- Diagrama de clases de la arquitectura (B) de un sistema.

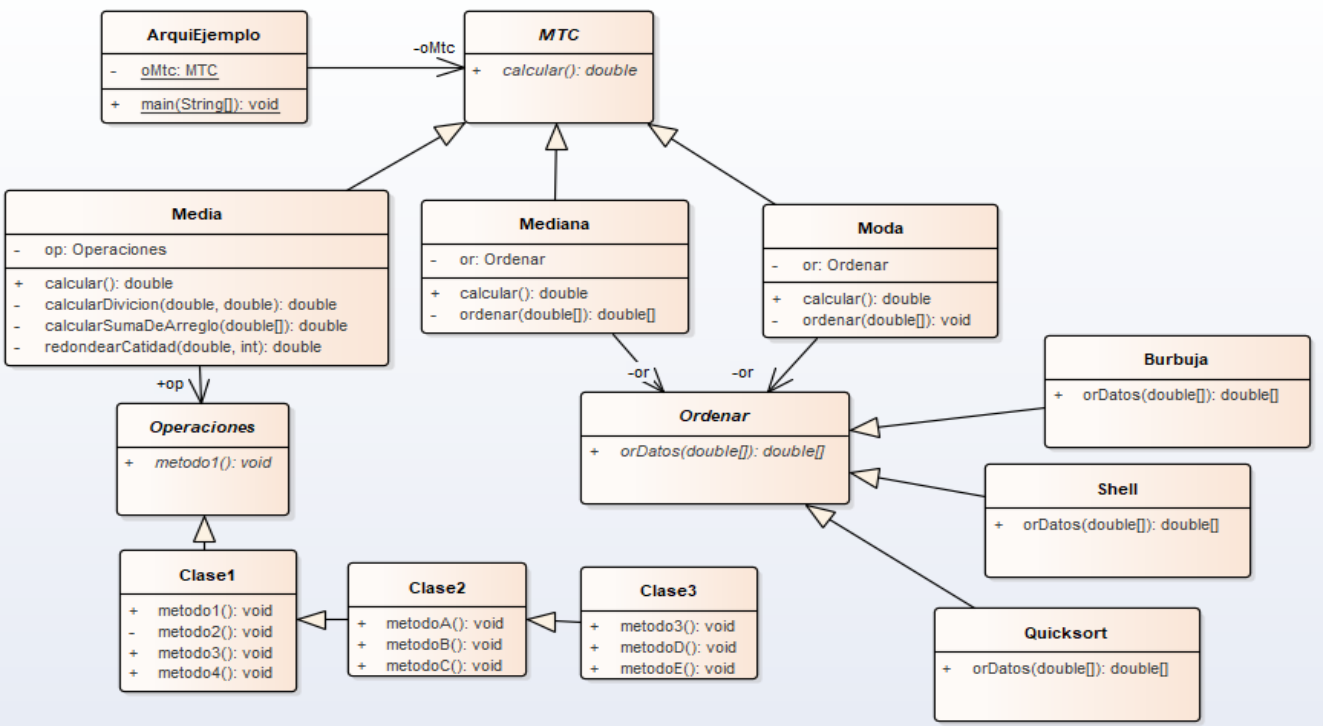


Figura 33.- Diagrama de clases de la arquitectura (C) de un sistema.

Sustituyendo los valores en la métrica PMFP se obtienen los siguientes resultados:

Tabla 41.- Resultados manuales y automáticos de la métrica PMFP de las arquitecturas A, B y C

	Resultados Manuales	Resultados obtenidos implementando métricas en código
P3	P3 = Diagrama A: $PMFP = \frac{0}{13} = 0$	Output Debugger Console × Metricas (run) × run: Metrica PMFP:0 BUILD SUCCESSFUL (total time: 17 seconds)
P2	P2 = Diagrama B: $PMFP = \frac{0.5}{13} = 0.03$	Debugger Console × Metricas (run) × run: Metrica PMFP:0.038 BUILD SUCCESSFUL (total time: 16 seconds)
P1	P1 = Diagrama C: $PMFP = \frac{2}{13} = 0.15$	Output Debugger Console × Metricas (run) × run: Metrica PMFP:0.154 BUILD SUCCESSFUL (total time: 14 seconds)

Relación transitiva

P1 •≥ P2, P2 •≥ P3 <=> P1 •≥ P3

⇕ Equivalente a:

PMFP (arquitectura C) •≥ PMFP (arquitectura B), PMFP (arquitectura B) •≥ PMFP (arquitectura A) => PMFP (arquitectura C) •≥ PMFP (arquitectura A).

Comprobación formal es una relación binaria transitiva:
Reflejado en números, se tiene que:

$$0.15 \geq 0.03,\ 0.03 \geq 0 <=> 0.15 \geq 0$$

Se puede observar que la arquitectura C presenta mayor grado de protección modular por funciones "*private*" debido al mayor número de funciones privadas que presenta. Mientras que la arquitectura A presenta un grado de protección menor, concluyendo que la métrica PMFP es una relación binaria transitiva.

Relación completa

Si se tienen P1 y P2 se debe poder decir que P1 tiene un mayor o igual grado de protección modular de funciones privadas que la P2, o viceversa. Es decir:

$$P1 \bullet\geq P2 \text{ o } P2 \bullet\geq P1$$

Comprobación formal es una relación binaria completa
Sustituyendo los valores obtenidos aplicando la métrica PMFP a las arquitecturas C y B, obtenemos siguiente resultado: $0.15 \geq 0.03$

Dado que la arquitectura C con valor de PMFP = 0.15 y la arquitectura B con valor de PMFP = 0.03 se tiene que: PMFP (arquitectura C) ≥ PMFP (arquitectura B), por lo tanto, se concluye que la métrica PMFP es una relación binaria completa, siempre fue posible determinar cuando existía mayor o igual grado de protección modular de funciones privadas.

Orden de clasificación

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera:

$$P1 \bullet\geq P2 <=> PMFP(P1) \geq PMFP(P2)$$

El orden que se crea por la métrica PMFP debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es un homomorfismo

Sea una clase con un total de 4 funciones tenemos que:

(4 funciones "*private"*) •≥ (3 funciones "*private"*) •≥ (1 función "*private"*)

⇕ Equivalente a:

$$1 \geq 0.75 \geq 0.25$$

Se puede observar que al aplicar los valores a la métrica PMFP se obtiene el mismo comportamiento. Concluyendo que la métrica PMFP es un homomorfismo.

Conclusión

Al cumplirse las condiciones (1) la relación •≥ es de orden débil (es una relación binaria que es transitiva y completa), y (2) es un homomorfismo (P1 •≥ P2 => PMFP (Arquitectura C) ≥ PMFP (Arquitectura B)). Se concluye que la métrica PMFP es de escala ordinal (Zuse, 1995).

Protección Modular de Funciones Protegidas (PMFPr) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1.- P es el conjunto de funciones con calificador de alcance "*protected*".
2.- •≥ es una relación empírica entre el número de funciones protegidas la cual describe si existe menor o mayor grado de protección de la información con respecto al calificador de alcance "*protected*".
3.- ℝ es el conjunto de números reales.
4.- ≥ "mayor o igual que" es una relación binaria entre números.

Entonces la métrica PMFPr será considerada como una escala ordinal si:

1. La relación binaria •≥, es de orden débil
2. P1 •≥ P2 => PMFPr (Arquitectura E) ≥ PMFPr (Arquitectura D).

Comprobación

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (2), para realizar el cálculo de la PMFPr de las arquitecturas C, D y E, mostradas en las Figuras 33, 34 y 35 respectivamente.

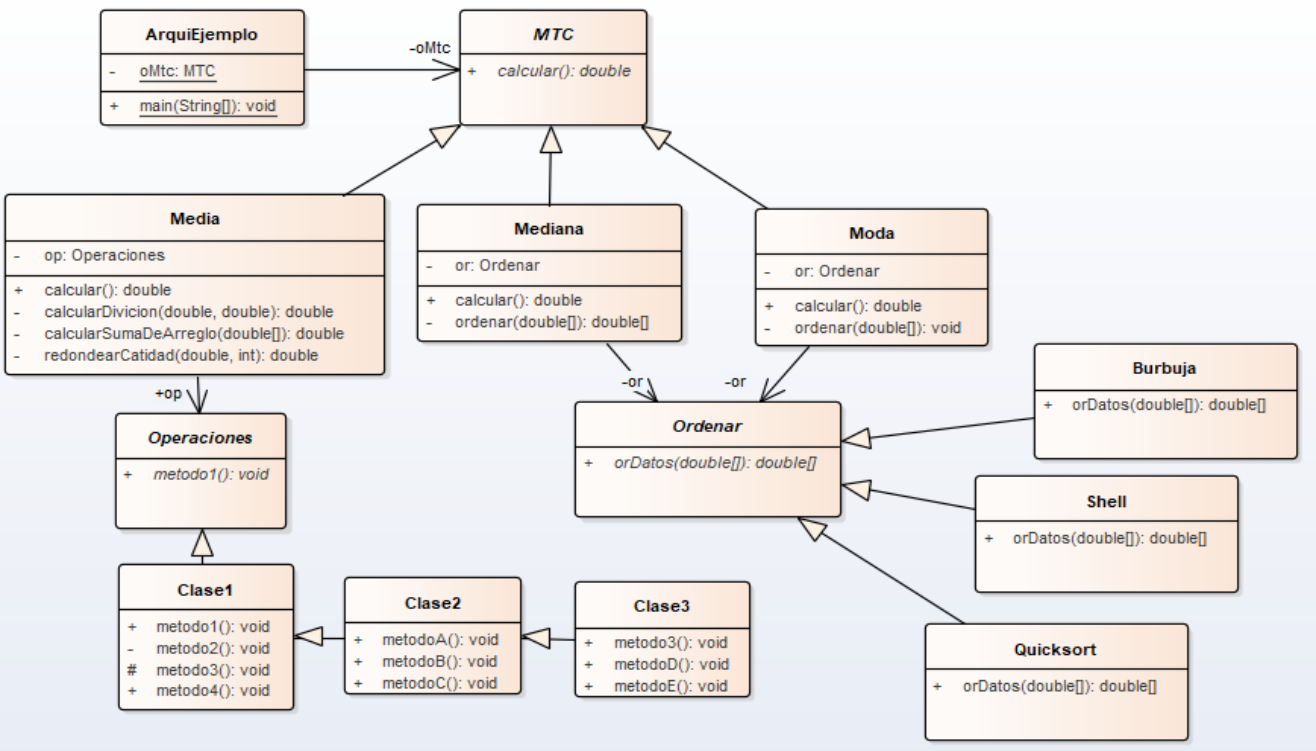


Figura 34.- Diagrama de clases de la arquitectura (D) de un sistema.

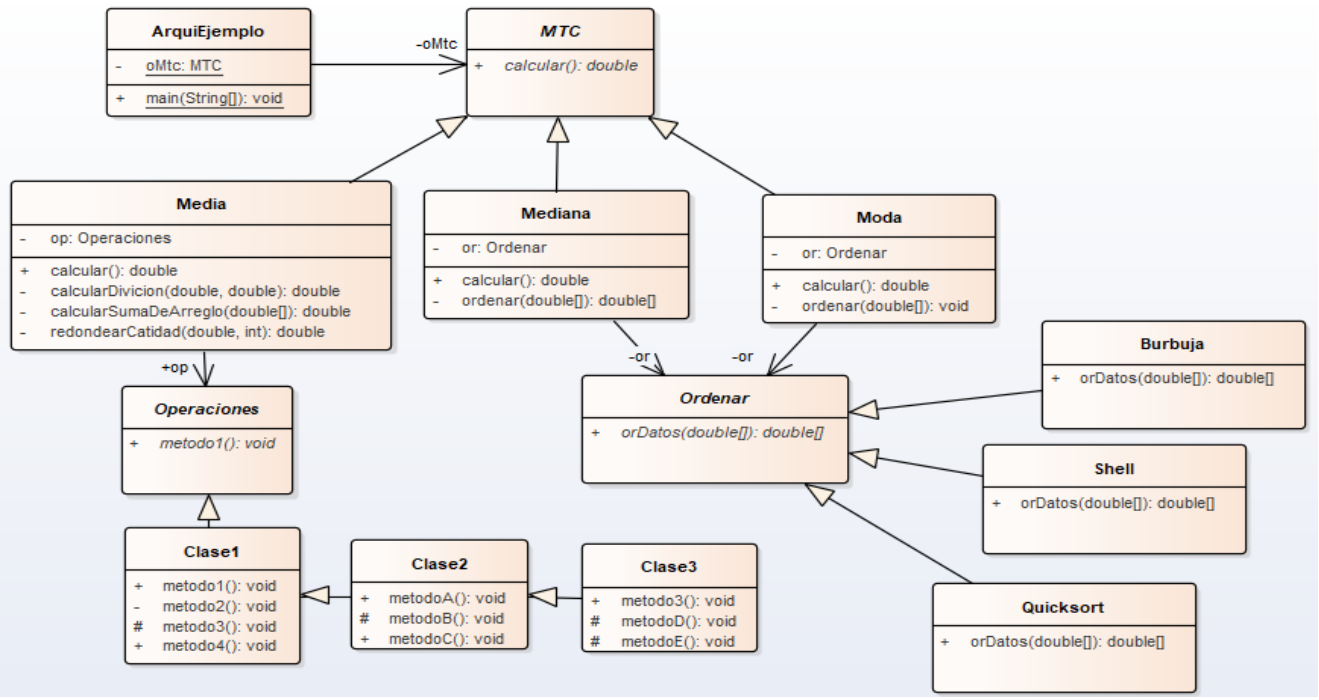
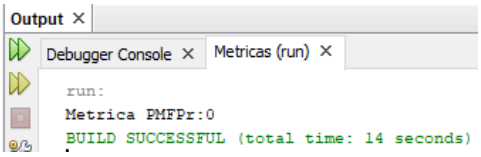
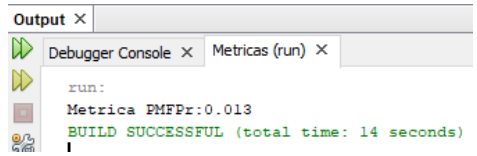
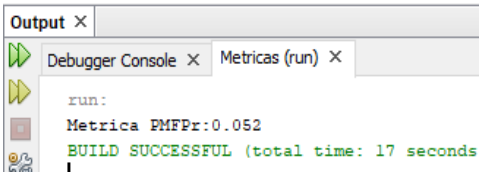


Figura 35.- Diagrama de clases de la arquitectura (E) de un sistema.

Sustituyendo los valores en la métrica PMFPr tomando los valores de las Arquitecturas C, D y E, se obtienen los siguientes valores:

Tabla 42.- Resultados manuales y automáticos de la métrica PMFPr de las arquitecturas C, D y E

	Resultados Manuales	Resultados obtenidos implementando métricas en código
P3	PMFPr (Diagrama C): $PMFPr = \frac{\left(\frac{0}{11}\right) + \left(\frac{0}{5}\right) + \left(\frac{0}{3}\right)+\left(\frac{0}{3}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)}{7} = 0$	
P2	PMFPr (Diagrama D): $PMFPr = \frac{\left(\frac{1}{11}\right) + \left(\frac{0}{5}\right) + \left(\frac{0}{3}\right)+\left(\frac{0}{3}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)}{7} = 0.01$	
P1	PMFPr (Diagrama E): $PMFPr = \frac{\left(\frac{4}{11}\right) + \left(\frac{0}{5}\right) + \left(\frac{0}{3}\right)+\left(\frac{0}{3}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)+\left(\frac{0}{2}\right)}{7} = 0.05$	

Relación transitiva

P1 •≥ P2, P2 •≥ P3 <=> P1 •≥ P3

⇕ Equivalente a:

PMFPr (Diagrama E) •≥ PMFPr (Diagrama D), PMFPr (Diagrama D) •≥ PMFPr (Diagrama C) => PMFPr (Diagrama E) •≥ PMFPr (Diagrama C)

Comprobación formal es una relación binaria transitiva:

Reflejado en números, se tiene que:

$$0.05 \geq 0.01,\ 0.01 \geq 0 \Rightarrow 0.05 \geq 0$$

Se puede observar que la arquitectura E presenta mayor grado de protección debido a que más funciones han sido declaradas como "*protected*", mientras que la arquitectura C presenta menos protección de la información. Concluyendo así, en que la métrica PMFPr es una relación binaria transitiva.

Relación completa

Si se tienen P1 y P2 se debe poder decir que P1 tiene un mayor o igual grado de protección modular de funciones protegidas que P2, o viceversa. Es decir:

$$P1 \bullet\geq P2 \text{ o } P2 \bullet\geq P1$$

Comprobación formal es una relación binaria completa

Sustituyendo los valores obtenidos aplicando la métrica PMFPr a las arquitecturas E y D, obtenemos siguiente resultado: $0.05 \geq 0.01$

Dado que la arquitectura E con valor de PMFPr = 0.05 y la arquitectura D con valor de PMFPr = 0.01 se tiene que: PMFPr (arquitectura E) ≥ PMFPr (arquitectura D), por lo tanto, se concluye que la métrica PMFPr es una relación binaria completa, ya que siempre fue posible determinar cuando existía mayor o igual grado de protección modular de funciones protegidas.

Orden de clasificación

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera:

$$P1 \bullet\geq P2 <=> PMFPr(P1) \geq PMFPr(P2)$$

El orden que se crea por la métrica PMFPr debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es un homomorfismo

Para la comprobación de este caso se toma que el número total de jerarquías es uno, con un total de 4 funciones.

(3 funciones "*protected*") •≥ (2 funciones "*protected*") •≥ (1 función "*protected*")

⇕ Equivalente a:

$1 \geq 0.75 \geq 0.25$

Se puede observar que entre mayor sea el número de funciones "*protected"*, dentro de una jerarquía mayor será la protección modular en cuanto al calificador de alcance "*protected*", de igual manera se puede observar que al aplicar los valores a la métrica PMFPr se obtiene el mismo comportamiento. Concluyendo que la métrica PMFPr es un homomorfismo.

Conclusión

Al cumplirse las condiciones (1) la relación •≥ es de orden débil (es una relación binaria que es transitiva y completa), y (2) es un homomorfismo (P1 •≥ P2 => PMFPr(P1) ≥ PMFPr(P2)). Se concluye que la métrica PMFPr es de escala ordinal (Zuse, 1995).

Protección Modular de Funciones Friendly (PMFF) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1.- P es el conjunto de funciones con calificador de alcance "*friendly*"
2.- •≥ es una relación empírica entre el número de funciones "*friendly*" que describe si existe menor o mayor grado de protección (uso del calificador de alcance "*friendly*").
3.- ℝ es el conjunto de números reales.
4.- ≥ "mayor o igual que" es una relación binaria entre números.

Entonces la métrica PMFF será considerada como una escala ordinal si:

1. La relación binaria •≥, es de orden débil
2. P1 •≥ P2 => PMFF (Arquitectura G) ≥ PMFF (Arquitectura F).

Comprobación

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (3), para realizar el cálculo de la PMFF de las arquitecturas E, F y G mostradas en las Figuras 35, 36 y 37 respectivamente.

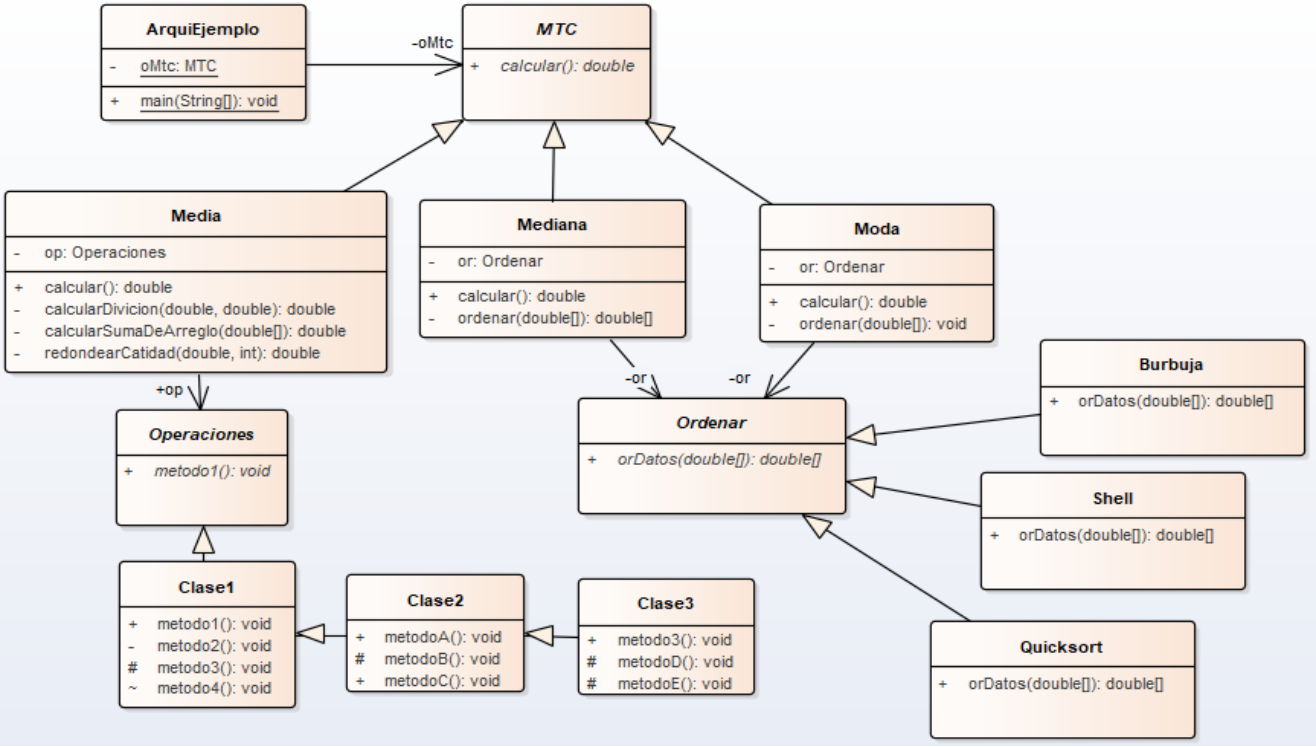


Figura 36.- Diagrama de clases de la arquitectura (F) de un sistema.

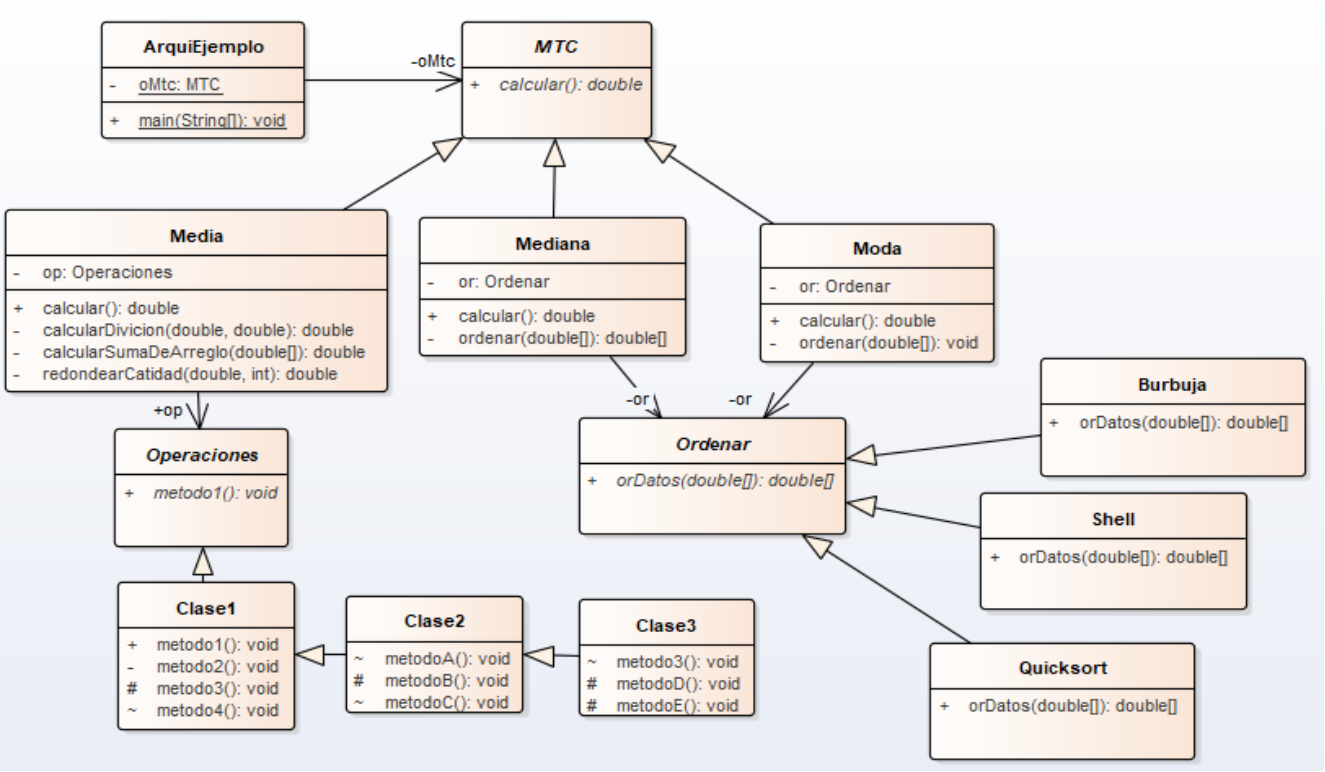


Figura 37.- Diagrama de clases de la arquitectura (G) de un sistema.

Sustituyendo los valores en la métrica PMFF tomando los valores de las Arquitecturas E, F y G, se obtienen los siguientes valores:

Tabla 43.- Resultados manuales y automáticos de la métrica PMFF de las arquitecturas E, F y G.

	Resultados Manuales	Resultados obtenidos implementando métricas en código
P3	PMFF (Diagrama E): $PMFF = \frac{0}{25} = 0$	Output Debugger Console Metricas (run) run: Metrica PMFF:0 BUILD SUCCESSFUL (total time: 14 seconds)
P4	PMFF (Diagrama F): $PMFF = \frac{1}{25} = 0.04$	Output Debugger Console Metricas (run) run: Metrica PMFF:0.04 BUILD SUCCESSFUL (total time: 18
P5	PMFF (Diagrama G): $PMFF = \frac{4}{25} = 0.16$	Output Debugger Console Metricas (run) run: Metrica PMFF:0.16 BUILD SUCCESSFUL (total time: 18 seconds)

Relación transitiva

$$P1 \bullet\geq P2,\ P2 \bullet\geq P3 \Rightarrow P1 \bullet\geq P3$$

$$\Updownarrow \text{ Equivalente a:}$$

$$PMFF\ (Diagrama\ G) \bullet\geq PMFF\ (Diagrama\ F),\ PMFF\ (Diagrama\ F) \bullet\geq PMFF\ (Diagrama\ E) \Rightarrow PMFF\ (Diagrama\ G) \bullet\geq PMFF\ (Diagrama\ E)$$

Comprobación formal es una relación binaria transitiva:

Reflejado en números, se tiene que:

$$0.16 \geq 0.04,\ 0.04 \geq 0 \Rightarrow 0.16 \geq 0$$

Se puede observar que la arquitectura E presenta un mayor grado de protección modular de funciones "*friendly*" a diferencia de la arquitectura E el cual presenta un menor grado de protección. Concluyendo así, en que la métrica PMFF es una relación binaria transitiva.

Relación completa

Si se tienen P1 y P2 se debe poder decir que P1 tiene un mayor o igual grado de protección modular de funciones "*friendly*" que P2, o viceversa. Es decir:

$$P1 \bullet\geq P2 \text{ o } P2 \bullet\geq P1$$

Comprobación formal es una relación binaria completa

Sustituyendo los valores obtenidos aplicando la métrica PMFF a las arquitecturas G y F, obtenemos siguiente resultado: $0.16 \geq 0.04$

Dado que la arquitectura G con valor de PMFF = 0.16 y la arquitectura B con valor de PMFF = 0.04 se tiene que: PMFF (arquitectura G) ≥ PMFF (arquitectura F), por lo tanto, se concluye que la métrica PMFF es una relación binaria completa. Concluyendo siempre fue posible determinar cuándo se presentaba un mayor o igual grado de protección modular de funciones "*friendly*", por lo que la métrica PMFF es una relación binaria completa.

Orden de clasificación

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera:

$$P1 \bullet\geq P2 < \Rightarrow PMFF(P1) \geq PMFF(P2)$$

El orden que se crea por la métrica PMFF debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es un homomorfismo

Arquitectura F Arquitectura G Arquitectura E

(4 funciones "*friendly*") •≥ (1 función "*friendly*") •≥ (0 Funciones "*friendly*")

⇕ Equivalente a:

$0.16 \geq 0.04 \geq 0$

Se puede observar que entre mayor sea el número de funciones "*friendly*" mayor será la protección modular de funciones "*friendly*". Al aplicar la métrica PMFF en las arquitecturas E, F y G se obtiene el mismo comportamiento. Concluyendo que la métrica PMFF es un homomorfismo.

Conclusión

Al cumplirse las condiciones (1) la relación •≥ es de orden débil (es una relación binaria que es transitiva y completa), y (2) es un homomorfismo (P1 •≥ P2 => PMFF(P1) ≥ PMFP(P2)). Se concluye que la métrica PMFF es de escala ordinal.

Protección Modular (PM) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1.- P es el conjunto de funciones con calificador de alcance diferente a "*public*".
2.- •≥ es una relación empírica entre el número de funciones públicas que describe si existe menor o mayor grado de protección de la información.
3.- ℟ es el conjunto de números reales.
4.- ≥ "mayor o igual que" es una relación binaria entre números.

Entonces la métrica PM será considerada como una escala ordinal si:

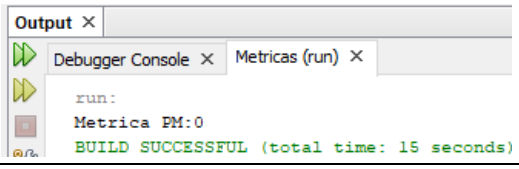
1. La relación binaria •≥, es de orden débil
2. P1 •≥ P2 => PM (Arquitectura G) ≥ PM (Arquitectura D).

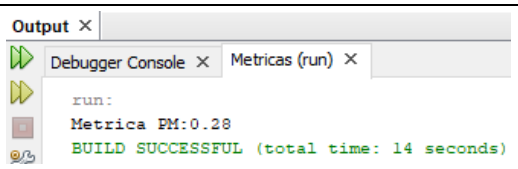
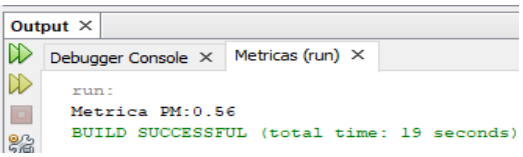
Comprobación

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (4), para realizar el cálculo de la PM de las arquitecturas A, D y G mostradas en las Figuras 31, 34 y 37 respectivamente.

Sustituyendo los valores en la métrica PM tomando los valores de las Arquitecturas A, D y G, se obtienen los siguientes valores:

Tabla 44.- Resultados manuales y automáticos de la métrica PM de las arquitecturas A, D y G.

	Resultados Manuales	Resultados obtenidos implementando métricas en código
P3	PM (Arquitectura A): $PM = \frac{0}{25} = 0$	

P2	PM (Arquitectura D): $PM = \frac{7}{25} = 0.28$	
P1	P1 = PM (Arquitectura G): $PM = \frac{14}{25} = 0.56$	

Relación transitiva

P1 •≥ P2, P2 •≥ P3 => P1 •≥ P3

Equivalente a:

PM (Arquitectura G) •≥ PM (Arquitectura D), PM (Arquitectura D) •≥ PM (Arquitectura A) => PM (Arquitectura G) •≥ PM (Arquitectura A)

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera: P1 ●≥ P2 <=> TPM (arquitectura C) ≥ TPM (arquitectura B). El orden que se crea por la métrica PM debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es una relación binaria transitiva:

Reflejado en números, se tiene que:

$$0.16 \geq 0.04, 0.04 \geq 0 \Rightarrow 0.16 \geq 0$$

En base a los resultados obtenidos se puede observar que entre mayor número de funciones no publicas la protección modular aumenta, en el caso de la arquitectura A se puede observar que todas las funciones con "*public*" por lo que tiene una total ausencia de protección de la información. Concluyendo así, en que la métrica PM es una relación binaria transitiva

Relación completa

Si tienen P1 y P2 se debe poder decir que P1 tiene un mayor o igual grado de protección modular que la P2, o viceversa:

P1 •≥ P2 o P2 •≥ P1

Dado que la arquitectura G con valor de PM = 0.56 y la arquitectura D con valor de PM = 0.28 se tiene que: PM (Arquitectura G) ≥ PM (Arquitectura D), por lo tanto, se concluye que la métrica PM es una relación binaria completa.

Orden de clasificación

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera:

$$P1 \bullet\geq P2 <=> PM(P1) \geq PM(P2)$$

El orden que se crea por la métrica PM debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es un homomorfismo

Arquitectura G (14 funciones $\overline{publıc}$) $\bullet\geq$ Arquitectura D (7 funciones $\overline{publıc}$) $\bullet\geq$ Arquitectura A (0 Funciones $\overline{publıc}$)

$\Updownarrow$ Equivalente a:

$0.56 \geq 0.28 \geq 0$

Se puede observar que entre mayor número de funciones no "*public*" ("*private*", "*protected*", "*friendly*") la protección modular es mayor, de igual forma se puede observar que al introducir los valores en la métrica PM, se observa el mismo comportamiento que en la experimentación. Concluyendo así, en que la métrica PM es un homomorfismo.

Conclusión

Al cumplirse las condiciones (1) la relación $\bullet\geq$ es de orden débil (es una relación binaria que es transitiva y completa), y (2) es un homomorfismo (P1 $\bullet\geq$ P2 => PM (arquitectura G) $\geq$ PM (arquitectura D)). Se concluye que la métrica PM es de escala ordinal.

Nivel de Total Protección Modular (TPM) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1.- P es el conjunto de funciones con calificador de alcance diferente a "*public*".
2.- $\bullet\geq$ es una relación empírica entre el número de no públicas que describe si existe menor o mayor grado de protección de la información.
3.- $\mathbb{R}$ es el conjunto de números reales.
4.- $\geq$ "mayor o igual que" es una relación binaria entre números.

Entonces la métrica TPM será considerada como una escala ordinal si:

1. La relación binaria $\bullet\geq$, es de orden débil
2. P1 $\bullet\geq$ P2 => TPM (Caso 3) $\geq$ TPM (Caso 2).

Comprobación

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (5), para realizar el cálculo de la TPM de los casos 1, 2 y 3 mostrados a continuación:

Caso 1:

$$PMFP = \frac{0}{13} = 0$$

$$PMFPr = \frac{\left(\frac{0}{10}\right) + \left(\frac{0}{4}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right)}{7} = 0$$

$$PMFF = \frac{0}{24} = 0$$

Caso 2:

$$PMFP = \frac{0.5}{13} = 0.03$$

$$PMFPr = \frac{\left(\frac{1}{10}\right) + \left(\frac{0}{4}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right)}{7} = 0.01$$

$$PMFF = \frac{1}{24} = 0.04$$

Caso 3:

$$PMFP = \frac{2}{13} = 0.15$$

$$PMFPr = \frac{\left(\frac{4}{10}\right) + \left(\frac{0}{4}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{2}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right) + \left(\frac{0}{1}\right)}{7} = 0.05$$

$$PMFF = \frac{4}{24} = 0.16$$

Sustituyendo los valores en la métrica TPM de los Casos 1, 2 y 3 se obtienen los siguientes resultados:

Tabla 45.- Resultados manuales y automáticos de la métrica TPM del Caso 1.

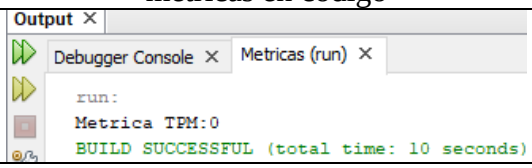
Resultados Manuales	Resultados obtenidos implementando métricas en código
$TPM = \frac{((0*2) + (0*0.75) + (0*0.25))}{3} = \frac{0}{3} = 0$	

Tabla 46.- Resultados manuales y automáticos de la métrica TPM del Caso 2.

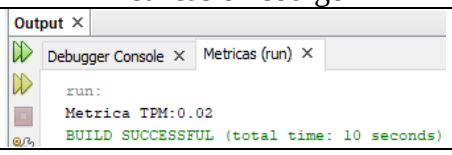
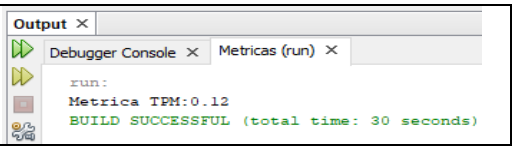
Resultados Manuales	Resultados obtenidos implementando métricas en código
$TPM = \frac{((0.03*2) + (0.01*0.75) + (0.04*0.25))}{3} = \frac{0.07}{3} = 0.02$	

Tabla 47.- Resultados manuales y automáticos de la métrica TPM del Caso 3.

Resultados Manuales	Resultados obtenidos implementando métricas en código

$$TPM = \frac{((0.15*2) + (0.05*0.75) + (0.16*0.25))}{3} = \frac{0.37}{3} = 0.12$$



Relación transitiva

P1 •≥ P2, P2 •≥ P3 => P1 •≥ P3

⇕ Equivalente a:

TPM (Caso 3) •≥ TPM (Caso 2), TPM (Caso 2) •≥ TPM (Caso 1) => TPM (Caso 3) •≥ TPM (Caso 1)

Comprobación formal es una relación binaria transitiva:
Reflejado en números, se tiene que:

$$0.12 \geq 0.02, 0.02 \geq 0 \Rightarrow 0.12 \geq 0$$

En base a los resultados obtenidos se puede observar que entre mayor número de funciones no publicas la protección modular aumenta, en el Caso 1 se puede observar que todas las funciones son "*public*" por lo que tiene una total ausencia de protección de la información. Concluyendo así, en que la métrica TPM es una relación binaria transitiva

Relación completa
Si tienen P1 y P2 se debe poder decir que P1 tiene un mayor o igual grado de protección modular que la P2, o viceversa:

P1 •≥ P2 o P2 •≥ P1

Dado que el Caso 3 con un valor de TPM = 0.12 y el Caso 2 con un valor de TPM = 0.02 se tiene que: TPM (Caso 3) ≥ PM (Caso 2), por lo tanto, se concluye que la métrica TPM es una relación binaria completa.

Orden de clasificación

El homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos, se denota de la siguiente manera:

P1 •≥ P2< => TPM(Caso 3) ≥ TPM (Caso 2)

El orden que se crea por la métrica TPM debe ser el mismo que el orden creado empíricamente mediante experimentación, o viceversa.

Comprobación formal es un homomorfismo

Caso 3 Caso 2 Caso 1
(14 funciones no "*public*") •≥ (7 funciones no "*public*") •≥ (0 Funciones no "*public*")

⇕ Equivalente a:

$0.12 \geq 0.02 \geq 0$

Se puede observar que entre mayor sea el número de funciones no publicas mayor será el grado de protección modular. Al aplicar la métrica a los Casos 1, 2 y 3 se obtiene el mismo comportamiento. Concluyendo que la métrica TPM es un homomorfismo.

Conclusión

Al cumplirse las condiciones (1) la relación $\bullet\geq$ es de orden débil (es una relación binaria que es transitiva y completa), y (2) es un homomorfismo (P1 $\bullet\geq$ P2 => TPM (Caso 3) ≥ TPM (Caso 2). Se concluye que la métrica TPM es de escala ordinal.

Anexo B.- Importación del método de refactorización de protección modular

El método de refactorización de protección modular se desarrolló como un módulo independiente, dicho módulo se agregó a la herramienta SR2-Refactoring el siguiente procedimiento:

1.- Se agrego a la herramienta SR2-refactoring el paquete "MetodoRefactorizarCalificador" el cual contiene la interfaz del método de refactorización de calificadores de alcance, así mismo se agregó el ejecutable del método de refactorización.

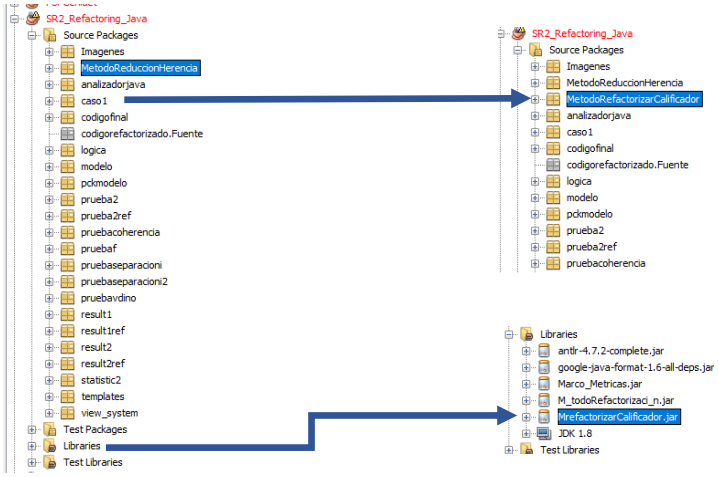


Figura 38.- Importación del paquete del método de refactorización de calificadores de alcance

2.- Se agrego a la interfaz de la herramienta SR2-refactoring la opción de seleccionar el "Método de Refactorización de Calificadores de Alcance" en la clase "principal_view", cabe mencionar que dicha clase fue la única a la que se le agrego código, el resto del código fue totalmente respetado.

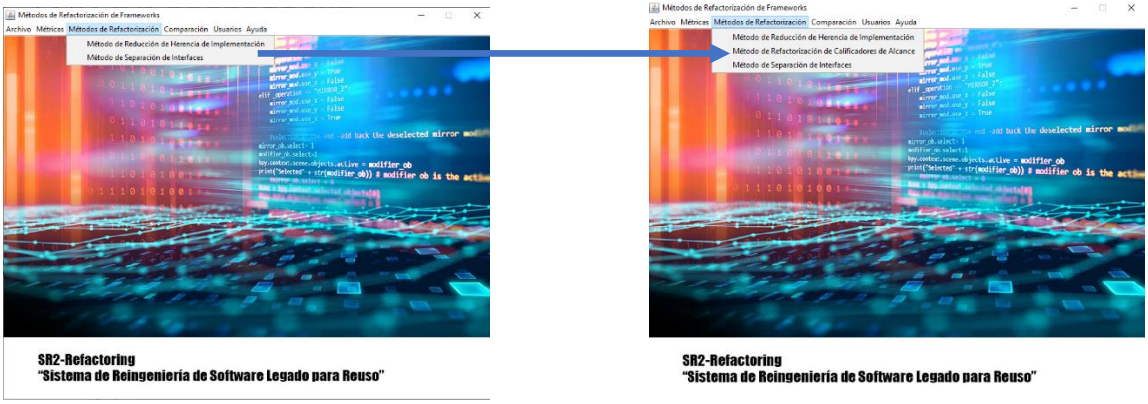


Figura 39.- Modificación de la interfaz de la herramienta SR2-Refactoring

Referencias

Alberto, D. (2010). Design Framework for the Development Dynamic Web Applications.

Arcelli, D., Cortellessa, V., & Pompeo, D. Di. (2018). Performance-driven software model refactoring, *95*(March 2017), 366–397. https://doi.org/10.1016/j.infsof.2017.09.006

Cárdenas. (2004). *Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator*.

Cervantes, J. (2016). *Introducción a la programación orientada a objetos*.

Christopoulou, A., Giakoumakis, E. A., Zafeiris, V. E., & Soukara, V. (2012). Automated refactoring to the Strategy design pattern. *Information and Software Technology*, *54*(11), 1202–1214. https://doi.org/10.1016/j.infsof.2012.05.004

Dallal, J. Al. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, *54*(10), 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004

Dallal, J. Al. (2017). Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, *92*, 105–120. https://doi.org/10.1016/j.infsof.2017.07.013

Duran Muñoz Francisco, G. L. F. (2017). *Programación orientada a objetos con Java*.

Finkelstein, L. (2009). widely-defined measurement – an analysis of challenges.

Gaitani, M. A. G., Zafeiris, V. E., Diamantidis, N. A., & Giakoumakis, E. A. (2015). Automated refactoring to the Null Object design pattern. *Information and Software Technology*, *59*, 33–52. https://doi.org/10.1016/j.infsof.2014.10.010

Girish Suryanarayana, Ganesh Samarthyam, T. S. (2015). *Refactoring for Software Design Smells: Managing Technical Debt*.

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

Khatchadourian, R., & Masuhara, H. (2017). Automated Refactoring of Legacy Java Software to Default Methods. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, 82–93. https://doi.org/10.1109/ICSE.2017.16

Kumar, V. G. (2019). *ICICCT 2019 – System Reliability, Quality Control, Safety, Maintenance and Management*.

Llinás, L. F. G. (2010). *Programación orienta a objetos en Java*.

Meananeatra, P., Rongviriyapanish, S., & Apiwattanapong, T. (2018). Refactoring opportunity identification methodology for removing long method smells and improving code analyzability. *IEICE Transactions on Information and Systems*, *E101D*(7), 1766–1779. https://doi.org/10.1587/transinf.2017KBP0026

Ortiz. (2020). Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos.

Padilla. (2019). *Método de Re-factorización de código java con interfaces y abstracciones*

incorrectas.

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

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

Pressman, R. S., & Ph, D. (n.d.). *Ingeniería del software*.

Rathee, A., & Chhabra, J. K. (2017). Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering, 7.

René, S. (2003). Modelo formal para la reestructura de marcos orientados a objetos hacia arquitecturas modelo-vista-adaptador Formal Model for Restructuring of Object-Oriented Frameworks to Architecture. *Ingeniería Investigación y Tecnología*, *15*(2), 187–198. https://doi.org/10.1016/S1405-7743(14)72209-7

Santaolaya Salgado, R., Fragoso Diaz, O. G., Ortiz Gutierrez, O., Bautista Juarez, C. V., & B. M. (2019). Marco Orientado a Objetos para la Medición de Calidad de Arquitecturas de Software.

Santos. (2005). *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter*.

Stevens, S. S. (1946). On the Theory of Scales of Measurement.

Tufano, M., Oliveto, F. P. G. B. R., & Penta, M. D. A. D. L. D. P. (2017). When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away).

Valdés. (2004). *Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces*.

Zuse, H. (1995). Properties of Object-Oriented Software Measures.