Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

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

presentada por
Ing. Orlando Ortiz Gutierrez

como requisito para la obtención del grado de
Maestro en Ciencias de la Computación

Director de tesis
Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Junio de 2020.

cenidet®
Centro Nacional de Investigación
y Desarrollo Tecnológico

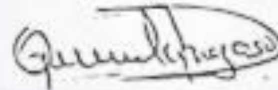
Cuernavaca, Mor., 14/mayo/2020

OFICIO No. DCC/043/2020
Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

C. DR. GERARDO VICENTE GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial del **C. Ing. Orlando Ortiz Gutierrez**, con número de control M18CE010, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos"** 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 Díaz
Doctora en Ciencias en Ciencias de la Computación
7420199
Revisor 1

M.C. Humberto Hernández García
Maestro en Ciencias con Especialidad en Sistemas Computacionales
7573641
Revisor 2

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

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

Cuernavaca, Morelos 26/mayo/2020
OFICIO No. SAC/ 152/2020
Asunto: Autorización de impresión de tesis

ORLANDO ORTIZ GUTIÉRREZ
CANDIDATO AL GRADO DE MAESTRO EN CIENCIAS
DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado ***"Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos"***, 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. GERARDO VICENTE GUERRERO RAMÍREZ
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
GVGR/CHG

DEDICATORIAS

Esta tesis está dedicada a:

A mis padres quienes con su amor, paciencia y esfuerzo me han permitido llegar a cumplir hoy un sueño más, gracias por inculcar en mí el ejemplo de esfuerzo y valentía, de no temer las adversidades porque Dios está conmigo siempre.

A mi hermano por su cariño y apoyo incondicional, durante todo este proceso, por estar conmigo en todo momento gracias. A toda mi familia porque con sus oraciones, consejos y palabras de aliento hicieron de mí una mejor persona y de una u otra forma me acompañan en todos mis sueños y metas.

Finalmente quiero dedicar esta tesis a todos mis amigos, por apoyarme cuando más las necesito, por extender su mano en momentos difíciles y por el amor brindado cada día.

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 la Universidad de Medellín por brindarme la oportunidad de realizar una estancia en el mes de octubre del 2019, por el espacio, experiencia y el tiempo para completar mis actividades de la maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por su generosidad al brindarme la oportunidad de recurrir a su capacidad y experiencia científica en un marco de confianza, afecto y amistad, fundamentales para la concreción de este trabajo.

A mis revisores, Dra. Olivia Graciela Fragoso Diaz y M.C. Humberto Hernández García por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

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

A mis padres por brindarme todo su apoyo y por haberme forjado como la persona que soy en la actualidad, muchos de mis logros se los debo a ustedes, entre los cuales se incluye este. Ustedes me motivaron constantemente para alcanzar mis sueños.

A mis compañeros y amigos que estuvieron conmigo en todo el recorrido de mi carrera, por la convivencia que tuvimos, confianza: Julia, Juan Vicente, Valentino, Antonio, Violeta, Heidi, Juan Carlos, Iván, Nelida, Marisol y Ricardo.

RESUMEN

La programación orientada a objetos es un paradigma de programación que utiliza objetos de datos para desarrollar sistemas de software. Se basa en técnicas como la herencia, abstracción y polimorfismo, con la finalidad de crear software de calidad. Además, se apoya en el uso de principios de diseño de software, como lo son los principios de *"Abierto-Cerrado"* y el de *"Inversión de Dependencias"*.

En los sistemas de software se pueden encontrar Marcos de Aplicaciones Orientados a Objetos (MAOO), cada marco es un conjunto de clases que trabajan colaborativamente para ofrecer soluciones a problemas específicos y servicios dentro de un mismo dominio de aplicaciones.

Al utilizar un MAOO se suelen reducir los tiempos de desarrollo de un sistema de software, ya que, si se tiene uno para cumplir con una tarea específica, dicha función se puede reusar para otro sistema sin tener que modificar el código original del MAOO, siempre que se respeten los principios de diseño de software. Si un MAOO no cumple con los principios de diseño se suelen producir unidades de programa con poca reusabilidad y con un alto costo en su mantenimiento.

Uno de los problemas que se presenta al no respetar los principios de diseño de software es la herencia de implementación la cual viola dos principios de diseño importantes, el principio de *"Abierto-Cerrado"* y el principio de *"Inversión de Dependencias"*. La herencia de implementación es una relación donde una subclase hereda la implementación funcional de un método de su clase base (Pinto, 2015). Esta herencia de implementación hace que aumente el acoplamiento y disminuya la flexibilidad entre clases, ya que, las subclases dependen de las clases base para poder realizar alguna función, impidiendo satisfacer las demandas funcionales requeridas por nuevas aplicaciones desarrolladas a partir de éstas, por mecanismos de herencia, integración, ensamble y/o composición de componentes reusables.

Para resolver este problema se propone un método de refactorización denominado "Reducción de Herencia de Implementación" el cual consiste en invertir las llamadas directas de código de cliente a código de librerías (call forward) por llamadas de código de librerías hacia código de cliente (call back).

A pesar de que el acoplamiento por herencia de implementación es un problema existente, no se tiene una forma de medir el grado en que afecta a los MAOO. Por este motivo se propone un conjunto de cinco métricas de calidad, tres de ellas son para medir el factor de herencia de implementación, las cuales son: Factor de Herencia de Implementación (FHI), Factor de Herencia de Implementación por Jerarquía de Clases (FHIJ) y Factor Herencia de Implementación de una Arquitectura de Clases (FHIAC); las dos métricas restantes son para medir el factor de flexibilidad, estas son: Factor de Flexibilidad de Clase (FFC) y Factor Medio de Flexibilidad de Clases(FMFC).

Se presentan dos casos de estudio en donde se realiza el cálculo del conjunto de métricas definido y se aplica el método de "Reducción de Herencia de Implementación". Además, se muestran las ventajas que se obtienen al aplicar este método de refactorización a los marcos orientados a objetos.

ABSTRACT

Object-oriented programming is a programming paradigm that uses data objects to develop software systems. It is based on techniques such as inheritance, abstraction and polymorphism, in order to create quality software. In addition, it relies on the use of software design principles, such as the "Open-Closed" and "Dependency Inversion" principles.

In software systems you can find Object Oriented Framework (OOF), each framework is a set of classes that work collaboratively to offer solutions to specific problems and services within the same application domain.

When using a OOF, the development times of a software system are usually reduced, since, if you have one to fulfill a specific task, this function can be reused for another system without having to modify the original OOF code, always that the principles of software design are respected. If a OOF does not comply with the design principles, program units are often produced with little reusability and high maintenance costs.

One of the problems that arises from not respecting the principles of software design is the implementation inheritance which violates two important design principles, the "Open-Closed" principle and the "Dependency Inversion" principle. Implementation inheritance is a relationship where a subclass inherits the functional implementation of a method of its base class (Pinto, 2015). This inheritance of implementation increases coupling and reduces flexibility between classes, since subclasses depend on the base classes to be able to perform some function, preventing meeting the functional demands required by new applications developed from them, by mechanisms of inheritance, integration, assembly and / or composition of reusable components.

To solve this problem, a refactoring method called “Reduction of Implementation Inheritance” is proposed, which consists of reversing direct calls from client code to library code (call forward) for calls from library code to client code (call back).

Although coupling by implementation inheritance is an existing problem, there is no way to measure the degree to which it affects the OOF. For this reason, a set of five quality metrics is proposed, three of them are to measure the implementation inheritance factor, which are: Implementation Inheritance Factor (FHI), Implementation Inheritance Factor by Class Hierarchy (FHIJ) and Implementation Inheritance Factor of a Class Architecture (FHIAC); The two-remaining metrics are for measuring the flexibility factor, these are: Class Flexibility Factor (FFC) and Average Class Flexibility Factor (FMFC).

Two case studies are presented in which the calculation of the defined set of metrics is performed and the method of “Reduction of Implementation Inheritance” is applied. In addition, the advantages obtained by applying this refactoring method to object-oriented frameworks are shown.

Contenido

Capítulo 1.- INTRODUCCIÓN 17
Capítulo 2.- ANTECEDENTES 20
2.1.- Planteamiento del Problema 20
2.2.- Objetivo 22
2.2.1.- Objetivo General 22
2.2.2.- Objetivos específicos 22
2.3.- Estado del Arte 22
2.4.- Trabajos Relacionados 24
Capítulo 3.- MARCO TEÓRICO 33
3.1.- Marco de Aplicaciones Orientado a Objetos (MAOO) 33
3.1.1.- Clase 34
3.1.2.- Jerarquía de Clases 34
3.1.3.- Herencia 35
3.1.4.- Polimorfismo 35
3.1.5.- Abstracción 35
3.1.6.- Encapsulado 35
3.2.- Principios de Diseño Orientado a Objetos 36
3.2.1.- Principio de Diseño *"Abierto-Cerrado"* 36
3.2.2.- Principio de Diseño *"Inversión de Dependencias"* 36
3.3.- Refactorización 36
3.4.- Métricas Orientadas a Objetos 37
3.5.- Patrones de Diseño 37
3.5.1.- Patrón de Diseño *"Template Method"* 38
3.6.- Teoría de la Medición 40
3.6.1.- Escalas o Niveles de Medición 41
3.6.2.- Clasificación de escalas 42
Capítulo 4.- MATERIALES Y MÉTODOS DE SOLUCIÓN 45
4.1.- Métrica FHI (Factor de Herencia de Implementación) 45
4.2.- Métrica FHIJ (Factor de Herencia de Implementación de una Jerarquía de Clases) 46
4.3.- Métrica FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases) 47
4.4.- Métrica FFC (Factor de Flexibilidad de Clases) 47
4.5.- Métrica FMFAC (Factor Medio de Flexibilidad de Arquitecturas de Clases) 48
4.6.- Refactorización por el Método de Reducción de Herencia de Implementación 50
Capítulo 5.- DESARROLLO DEL SISTEMA 55

5.1.- Análisis de casos de uso del sistema 55
5.2.- Explicación de Actores 58
5.3.- Diseño detallado del sistema 59
5.4.- Diagrama de Clases del Sistema 67
5.5.- Descripción de las Clases Entidad 68
5.6.- Análisis del Código fuente 69
Capítulo 6.- Pruebas 71
6.1.- Convención de nombres 71
6.2.- Plan de pruebas 72
6.3.- Especificación del Diseño de Pruebas 75
6.3.1.- Diseño de Prueba ISMRHI04 – 01 75
6.3.2.- Diseño de Prueba ISMRHI04 – 02 76
6.4.- Especificación de Casos de Prueba 77
6.4.1.- Caso de Prueba ISMRHI05 – 01 77
6.4.2.- Caso de Prueba ISMRHI05-02 78
6.4.3.- Caso de Prueba ISMRHI05 – 03 79
6.4.4.- Caso de Prueba ISMRHI05-04 80
6.5.- Ejecución de Pruebas 81
6.5.1.- Caso de Prueba ISMRHI05-01 81
6.5.2.- Caso de Prueba ISMRHI05-02 84
6.5.3.- Caso de Prueba ISMRHI05-03 94
6.5.4.- Caso de Prueba ISMRHI05-04 98
Capítulo 7.- CONCLUSIONES Y TRABAJO A FUTURO 111
7.1.- Conclusiones 111
7.2.- Aportaciones de la tesis 113
7.3.- Trabajo a Futuro 114
Anexo A.- Métricas FHI, FHIJ, FHIAC, FFC y FMFAC como escalas ordinales 115
Factor de Herencia de Implementación (FHI) como escala ordinal 115
Factor de Herencia de Implementación por Jerarquía de Clases (FHIJ) como Escala Ordinal 117
Factor de Herencia de Implementación de una Arquitectura de Clases (FHIAC) como escala ordinal 120
Factor de Flexibilidad de Clases (FFC) como escala ordinal 122
Factor Medio de Flexibilidad de Arquitecturas de Clases (FMFAC) como escala ordinal 125

Índice de Figuras

Figura 1.- Clases pertenecientes al MAOO de estadística. ... 21
Figura 2.- Ejemplo de una jerarquía de clases. ... 34
Figura 3.- Ejemplo de dos jerarquías de clases. ... 35
Figura 4.- Estructura del patrón de diseño del “Método de la Plantilla” ... 39
Figura 5.- Ejemplo de una función. ... 41
Figura 6.- Relaciones entre clases de escalas. ... 43
Figura 7.- Diagrama BPMN del Método de Reducción de Herencia de Implementación ... 50
Figura 8.- Clases pertenecientes al MAOO de estadística. ... 52
Figura 9.- Clases refactorizadas pertenecientes al MAOO estadístico. ... 52
Figura 10.- Recorrido del ejemplo propuesto en el diagrama BPMN del método de refactorización. ... 53
Figura 11.- Diagrama de Casos de Uso del Sistema SR2-Refactoring. ... 56
Figura 12. Diagrama del Caso de Uso Manejar Archivo. ... 56
Figura 13.- Diagrama del caso de uso Análisis Sintáctico. ... 57
Figura 14.- Diagrama del caso de uso Medición. ... 57
Figura 15.- Diagrama de Caso de Uso Refactorización. ... 58
Figura 16.- Diagrama de secuencia del caso de uso Manejar Archivo. ... 59
Figura 17.- Diagrama de secuencia del caso de uso Análisis Sintáctico. ... 60
Figura 18.- Diagrama de secuencia del caso de uso Medición. ... 61
Figura 19.- Diagrama de actividades del método calcular. ... 63
Figura 20.- Diagrama de secuencia del caso de uso Refactorización. ... 64
Figura 21.- Diagrama de actividades del método refactorizaClaseBase. ... 65
Figura 22.- Diagrama de actividades del método refactorizaClaseDerivada. ... 66
Figura 23.- Arquitectura del SR2-Refactoring. ... 67
Figura 24.- Arquitectura del sistema de refactorización. ... 68
Figura 25.- Convención de nombres. ... 71
Figura 26.- Captura del resultado de la media aritmética antes de la refactorización. ... 82
Figura 27.- Captura del resultado de la media aritmética después de la refactorización. ... 82
Figura 28.- Arquitectura de las clases para medir la métrica FHI. ... 84
Figura 29.- Resultado de la métrica FHI utilizando el sistema SR2-Refactoring. ... 85
Figura 30.- Arquitectura de las clases para medir la métrica FHIJ: ... 86
Figura 31.- Resultado de la métrica FHIJ utilizando el sistema SR2-Refactoring. ... 87
Figura 32.- Arquitectura de clases para medir la métrica FHIAC. ... 89
Figura 33.- Resultado de la métrica FHIAC utilizando el sistema SR2-Refactoring. ... 90
Figura 34.- Clase aBetas. ... 90
Figura 35.- Resultado de la métrica FFC utilizando el sistema SR2-Refactoring. ... 91
Figura 36.- Resultado de la métrica FMFAC utilizando el sistema SR2-Refactoring. ... 93
Figura 37.- Usuarios antes de la refactorización. ... 94
Figura 38.- Actividades del usuario Orlando Ortiz Gutierrez antes de la refactorización. ... 94
Figura 39.- Usuarios del sistema PSPCenidet. ... 95
Figura 40.- Registro de actividades. ... 95
Figura 41.- Arquitectura de las clases para medir la métrica FHI. ... 98
Figura 42.- Resultado de la métrica FHI utilizando el sistema SR2-Refactoring. ... 99
Figura 43.-Arquitectura de las clases para medir la métrica FHIJ. ... 100
Figura 44.- Resultado de la métrica FHIJ utilizando el sistema SR2-Refactoring. ... 101
Figura 45.- Arquitectura de las clases para medir la métrica FHIAC. ... 104
Figura 46.- Resultado de la métrica FHIAC utilizando el sistema SR2-Refactoring. ... 105
Figura 47.- Clase BaseDatos. ... 106

Figura 48.- Resultado de la métrica FFC utilizando el sistema SR2-Refactoring. 107
Figura 49.- Resultado de la métrica FMFAC utilizando el sistema SR2-Refactoring......................... 108
Figura 50.- Secuencia de Aplicación de los Métodos de Refactorización del SR2-Refactoring.......... 112
Figura 51.- Arquitectura de clases de un sistema.. 116
Figura 52.- Conjunto de jerarquías de clases. 119
Figura 53.- Arquitecturas de clases.. 121
Figura 54.- Arquitectura de clases de un sistema.. 123
Figura 55.- Arquitectura de clases de un sistema.. 126

GLOSARIO DE TÉRMINOS

Acoplamiento	Es el grado en que los componentes de software dependen el uno del otro (Gamma, Helm, Johnson, & Vlissides, 2002).
Acoplamiento Entre Objetos (CBO) (por sus siglas en inglés Coupling Between Objects)	Es un recuento del número de clases que están acopladas a una clase particular, es decir, donde los métodos de una clase llaman a los métodos o acceden a las variables de la otra.
Call Back	Llamada directa de código de librerías hacia código de cliente.
Call Forward	Llamada directa de código de cliente hacia código de librerías.
Clase	Una unidad de aplicación diseñada para instanciar objetos, en ella se proporcionan uno o más métodos, los cuales están diseñados para realizar las tareas de esa clase (Deitel P. J., 2014).
Código Desagradable (smell code)	Se refiere al código fuente de programas de computadora que, aunque no de la mejor manera, cumple con su meta de valor u objetivo, y que posiblemente podría presentar algún problema de fragilidad, rigidez, escalabilidad y movilidad.
Cohesión de una Clase (CC) (por sus siglas en inglés Class Cohesion)	Es la relación entre la suma de las similitudes entre todos los pares de métodos y el número total de pares de métodos.
Deuda Técnica	Construcciones de diseño o implementación que son convenientes a corto plazo, pero configuran un contexto técnico que puede hacer que un cambio futuro sea más costoso o imposible (Kruchten, 2016).
Default Method (Método Predeterminado)	Permiten agregar nuevas funcionalidades a las interfaces de sus bibliotecas y garantizar la compatibilidad binaria con el código escrito para versiones anteriores de esas interfaces (Oracle, 2017).
Extensibilidad	Es una medida de la capacidad que tiene un componente o un sistema de software para extender su comportamiento a nuevas funciones definidas por el usuario o desarrollador de aplicaciones.

Factor de Flexibilidad de Clase (FFC)	Para una clase base dada, es la suma del NOP, entre el número total de sus métodos.
Factor de Herencia de Implementación (FHI)	Para una clase base dada, es la suma de los métodos virtuales y no virtuales implementados en la clase base, entre el número total de métodos.
Factor Medio de Flexibilidad de Arquitecturas de Clases(FMFAC)	La suma de los FFC de la arquitectura del módulo entre el número total de clases.
Falta de Cohesión de Métodos (LCOM) (por sus siglas en inglés Lack Of Cohesion of Methods)	LCOM muestra el grado de falta de cohesión entre los métodos de una clase. En otras palabras, revela la medida en que los métodos de una clase trabajan para realizar una sola responsabilidad.
Flexibilidad	La facilidad con la que un sistema o componente puede modificarse para su uso en aplicaciones o entornos distintos de aquellos para los que fue específicamente diseñado (IEEE Standard Glossary of Software Engineering Terminology, 1990).
Fragilidad	Cuando se realiza un cambio, partes inesperadas del sistema dejan de funcionar.
Herencia	Es una relación que define una entidad en términos de otra. La herencia de clase define una nueva clase en términos de una o más clases principales. La nueva clase hereda su interfaz e implementación de sus padres. La nueva clase se llama subclase o una clase derivada. La herencia de clase combina la herencia de interfaz y la herencia de implementación (Gamma et al., 2002).
Herencia de implementación	Es una relación donde una subclase hereda la implementación del comportamiento de su clase base (Pinto, 2015).
Herencia de Implementación de una Arquitectura de Clases (FHIAC)	Para una arquitectura de clases es la suma del FHIJ entre el número de jerarquías de clases (NOH).
Herencia de Implementación por Jerarquía de Clases (FHIJ)	Para una rama de la jerarquía de clases es la suma del FHI de cada subclase entre el número total de clases en esa jerarquía menos 1.
Herencia de interfaz	Se produce cuando una subclase hereda la descripción del comportamiento de su clase base y proporciona la implementación en sí misma (Pinto, 2015).
Implementación de un método	Son las secuencias de instrucciones que un método tiene en su cuerpo.

Inmovilidad	Atributo de una aplicación de software, en el cual, su código es difícil de reusar en otras aplicaciones, porque no puede separarse de la aplicación actual.
JDeodorant	Es un complemento de Eclipse que identifica problemas de diseño en el software, conocidos como *"código desagradable"*, y los resuelve mediante la aplicación de refactorizaciones apropiadas. Emplea una variedad de métodos y técnicas novedosas para identificar *"código desagradable"* y sugerir las refactorizaciones apropiadas que los resuelven (Eclipse, 2015).
Líneas de Código (LOC) (por sus siglas en inglés Lines Of Code)	Es una métrica de software utilizada para medir el tamaño de un programa de computadora contando el número de líneas en el texto del código fuente del programa.
MAOO[1]	Marco de Aplicaciones Orientado a Objetos. Es un conjunto de clases que trabajan colaborativamente para ofrecer soluciones a problemas específicos y servicios dentro de un mismo dominio de aplicaciones.
Método Ponderados por Clase (WMC) (por sus siglas en inglés Weighted Methods per Class)	Es la suma de todas las complejidades de los métodos en la clase.
Métodos Abstractos	Los métodos abstractos en una clase base deben ser implementados en las clases derivadas. Estos métodos no heredan implementación alguna.
Métodos No Virtuales	Los métodos no virtuales en una clase base no pueden ser redefinidos en las clases derivadas. Estos métodos producen herencia de implementación. En lenguaje java no existen métodos no virtuales como sucede en lenguaje C++.
Métodos Virtuales	Los métodos virtuales en una clase base pueden, o no, ser implementados en las clases derivadas, a través de toda la jerarquía de clases. Solamente se hereda la implementación de aquellos métodos que no son redefinidos en las clases derivadas.
Métrica	Una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado (IEEE Standard Glossary of Software Engineering Terminology, 1990).
Número de Hijos (NOC) (por sus silgas en inglés Number of Children)	El número de subclases inmediatas de una clase.

[1] En esta tesis el término MA00 se utiliza para hacer referencia al término en inglés "Framework", el cual equivale a los términos en español, "Marco", "Marco de Aplicaciones", "Marco de Componentes Reusables" y "Marco de Aplicaciones Orientados a Objetos".

Numero de jerarquías (NOH) (por sus siglas en inglés Number of Hierarchies)	Es el conteo de las jerarquías de clase (Bansiya & Davis, 2002).
Número de Métodos Polimorficos (NOP) (por sus siglas en inglés Number Of Polymorphic Methods)	Métodos virtuales que exhiben un comportamiento polimórfico. (Bansiya & Davis, 2002).
Profundidad del árbol de herencia (DIT) (por sus siglas en ingles Depth of Inheritance Tree)	Es la longitud máxima de una ruta, desde una clase derivada a una clase base, en la estructura de herencia de una arquitectura de clases.
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" (Martin Fowler, 2009).
Respuesta para una Clase (RFC) (por sus siglas en inglés Response For A Class)	La métrica Respuesta para clase (RFC) es el número total de métodos que potencialmente se pueden ejecutar en respuesta a un mensaje recibido por un objeto de una clase. Este número es la suma de los métodos de la clase, y todos los métodos distintos se invocan directamente dentro de los métodos de la clase.
Reusabilidad	El grado en que se puede usar un módulo de software u otro producto de trabajo en más de un programa de computadora o sistema de software.
Rigidez	Es difícil de cambiar porque cada cambio tiene demasiados efectos en otras partes del sistema.
Skeletal Implementation	Es un patrón de diseño de software que consiste en definir una clase abstracta que proporciona una implementación de interfaz parcial.
SR2 Refactoring	Acrónimo de "Sistema de Reingeniería de Software Legado para Reuso" que identifica problemas de diseño en el software y sugiere su refactorización.
Template Method	Método de plantilla, es un patrón de diseño conocido como *"Template Method"* definido en el catálogo de patrones de diseño de (Gamma et al., 2002). Su objetivo es definir el esqueleto de un algoritmo en una operación, difiriendo algunos pasos a sus subclases. También permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

Capítulo 1.- INTRODUCCIÓN

Actualmente producir software orientado a objetos exige del ser humano una gran capacidad de imaginación, abstracción y creatividad. Uno de los objetivos de los desarrolladores de software es resolver problemas informáticos de manera práctica. Esto conlleva la utilización en conjunto de las capacidades antes mencionadas, lo cual es una tarea difícil de realizar.

Cuando el desarrollador de software carece de experiencia y habilidad en el desarrollo, suelen tomarse decisiones equivocadas de diseño arquitectural, que conducen a la *"deuda técnica",* lo cual genera *"código desagradable (smell code)"* en unidades de programa con poca reusabilidad y con un alto costo en su mantenimiento. *"Deuda técnica"* se define como: Construcciones de diseño o implementación que son convenientes a corto plazo, pero configuran un contexto técnico que puede hacer que un cambio futuro sea más costoso o imposible (Kruchten, 2016) y *"código desagradable (smell code)"* se refiere al código fuente de programas de computadora que, aunque no de la mejor manera, cumple con su meta de valor u objetivo, y que posiblemente podría presentar algún problema de fragilidad, rigidez, escalabilidad y movilidad. Una de tales decisiones de diseño se presenta en sistemas de software legado con herencia de implementación en sus jerarquías de clases, lo cual puede provocar un alto grado de dependencias, generando poca flexibilidad para cambiar dinámicamente su funcionamiento, se disminuye su reuso, lo cual provoca desatender el principio de diseño *"Abierto-Cerrado"*. Este problema se manifiesta debido al incorrecto diseño de arquitecturas de software en las cuales la implementación de funciones se realiza en las clases base, lo cual produce rigidez funcional e impide el cambio de comportamientos de diferentes aplicaciones a través de las propiedades de asociación dinámica y del polimorfismo de la programación orientada a objetos. Dicho de otra manera,

la funcionalidad de clases derivadas está fuertemente atada a la implementación funcional definida en su clase base, y esta implementación funcional no puede ser modificada dinámicamente. Adicionalmente, dado que la funcionalidad de clases derivadas depende de la funcionalidad de la clase base, las clases derivadas no pueden ser utilizadas en otros contextos diferentes, adoleciendo de la funcionalidad de la clase base. Esto origina un problema de movilidad e impide su reuso.

Cuando se refactoriza la arquitectura de clases para ganar autonomía, es necesario invertir la interacción entre funciones en las jerarquías de clases; cambiando las llamadas directas a funciones de las clases base o librerías, desde el código de clases del cliente o clases derivadas en la jerarquía; por llamadas a funciones de clases del cliente desde las funciones de las clases base o librerías, a través de llamadas invertidas (call backs). Esto puede conseguirse utilizando el patrón de diseño *"Template Method"* del catálogo de Gamma. De este modo se gana en autonomía de clases y en flexibilidad para que diferentes clientes puedan cambiar a sus necesidades el comportamiento funcional de las clases base, a través de la propiedad de polimorfismo.

En este documento de tesis se presenta el diseño e implementación de un método cuyo propósito es disminuir el acoplamiento por herencia de implementación en arquitecturas de software orientado a objetos. Además, este método y el cálculo semi-automático del conjunto de métricas de calidad de software definidas, se integran al Sistema de Reingeniería de Software Legado para Reuso como una extensión para incrementar la flexibilidad funcional y la autonomía de código legado orientado a objetos.

Asimismo, se considera que el código original a refactorizar posee un cierto comportamiento funcional, el cual se preserva totalmente después del proceso de refactorización.

El desarrollo de este sistema ayuda a reducir la interdependencia entre objetos y clases, tanto de software legado como de marcos de aplicaciones orientados a objetos. También, ayuda a disminuir la complejidad del mantenimiento y, por ende, disminuir el costo de desarrollo, además de que incorpora al software legado una estructura que mejora su reuso y simplifica el protocolo de interacción entre objetos.

Organización de este documento de tesis:

Los nombres de clases de un marco orientado a objetos se representan entre paréntesis angulares, ejemplo: <nombre de la clase> y los nombres de los métodos de una clase, se representan entre comillas, ejemplo: "nombre del método".

Para describir todo el trabajo que se realizó para desarrollar el método de Reducción de Herencia de Implementación y el diseño e implementación de las métricas de Flexibilidad y de Factor de Herencia de Implementación de esta tesis, a continuación, se muestra la organización del documento.

En el capítulo 2 se muestra la problemática de la investigación, así como también se muestra un resumen de los trabajos y antecedentes relacionados a la línea de esta tesis.

El capítulo 3 muestra el marco teórico, en donde se enlistan los conceptos utilizados en esta tesis.

En el capítulo 4, se muestra el conjunto de métricas definido en esta tesis, el cual contiene las métricas: FHI (Factor de Herencia de Implementación), FHIJ (Factor de Herencia de Implementación de una Jerarquía de Clases), FHIAC (Factor de Herencia de

Implementación de una Arquitectura de Clases), FFC (Factor de Flexibilidad de Clase) y FMFAC (Factor Medio de Flexibilidad de Arquitecturas de Clases).

El capítulo 5 muestra todo el análisis que se hizo para desarrollar el método de reducción de herencia de implementación. Como parte del análisis, se encuentran los diagramas de casos de uso, de secuencias, de actividades y de clases. Así como también se describe ANTLR (ANother Tool for Language Recognition), herramienta utilizada para realizar el análisis del código fuente.

El capítulo 6 contiene las pruebas que se utilizaron para evaluar el correcto funcionamiento del método de Reducción de Herencia de Implementación y del cálculo automático de las métricas definidas en esta tesis.

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

Capítulo 2.- ANTECEDENTES

En este capítulo se presenta lo siguiente: el planteamiento del problema que dio origen a la investigación, la solución utilizada, los objetivos general y específicos de la tesis. Así como también el estado del arte y trabajos relacionados a esta investigación.

2.1.- Planteamiento del Problema

El problema radica en que el software legado exhibe dependencias funcionales entre sus jerarquías de clases de objetos. Esto es debido a que las entidades de software legadas exhiben un fuerte acoplamiento entre las clases por la herencia de implementación funcional entre éstas, lo cual impide satisfacer las demandas funcionales requeridas por nuevas aplicaciones desarrolladas a partir de éstas, por mecanismo de integración, ensamble y/o composición de componentes reusables.

En la Figura 1 se muestra parte de la arquitectura de clases de un MAOO llamado Marco Estadístico, el cual tiene el objetivo de realizar cálculos estadísticos automáticamente.

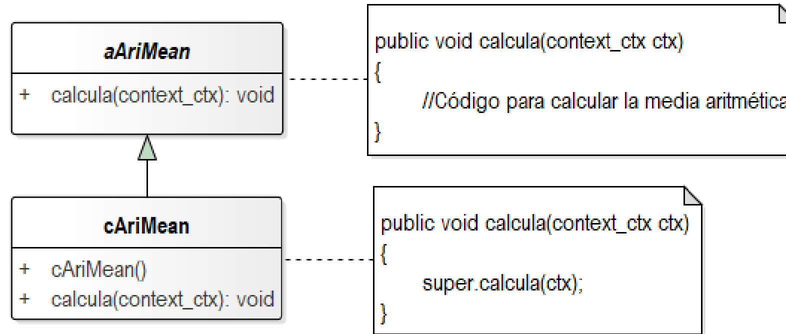


Figura 1.- Clases pertenecientes al MAOO de estadística.

El marco estadístico requiere una lista de números como entrada, sobre los que se van a realizar las operaciones para obtener los valores buscados. Tiene la finalidad de realizar cálculos estadísticos como lo son, las medidas de tendencia central, dispersiones, distribuciones, regresiones y correlaciones, así como también se tienen ordenaciones de datos y solución de sistemas de ecuaciones lineales.

Las clases que se muestran en la Figura 1, tienen como objetivo calcular la media aritmética del conjunto de números que el marco recibe como entrada. La clase *"aAriMean"* contiene el método *calcula(),* que contiene el proceso para obtener la media aritmética y la clase *"cAriMean"* está llamando a ese método desde su método *calcula()*, mediante la cláusula *"super"*. Lo que provoca que la clase *"cAriMean"* esté directamente acoplada a su clase base. Si en algún momento se quiere extender el marco estadístico, para cambiar su comportamiento, por ejemplo, para calcular la media geométrica, ésta sería una acción difícil de realizar, además, se desatenderían los principios de *"Abierto-Cerrado" y el de "Inversión de Dependencias"*.

El problema resulta grave cuando se considera la arquitectura completa del MAOO, en donde, si se tienen demasiadas clases que hereden el comportamiento de los métodos de sus clases base, el acoplamiento por herencia del MAOO aumentaría, afectando la reusabilidad, extensibilidad y flexibilidad. La reusabilidad del MAOO se ve disminuida por el problema de inmovilidad; esto es, la dificultad de reusar partes del MAOO en otras aplicaciones, porque no pueden separarse de la aplicación actual. Asimismo, la flexibilidad también se ve afectada por la rigidez, es decir, no se puede cambiar el comportamiento del MAOO sin modificarlo. No es deseable modificar un MAOO cuyo objetivo ideal es precisamente facilitar el reuso, la flexibilidad y la extensibilidad.

Solución del Problema

La solución que se plantea en este trabajo de tesis es invertir las llamadas directas (call forwards) por llamadas inversas (call backs) en las jerarquías de herencia de clases; a partir de un sistema automático de análisis sintáctico, para medir el grado de acoplamiento por relaciones de herencia de implementación del software legado o marcos de aplicaciones orientados a objetos escritos en el lenguaje Java; así como para descubrir las dependencias de herencia de implementación, denotadas por llamadas directas desde el código del cliente hacia el código de las librerías (call forward); para invertir el orden de las llamadas a call backs. Es decir, se plantea cambiar la herencia de implementación por herencia de interfaz,

denotada ésta por llamadas desde el código de las librerías hacia el código del cliente, utilizando el patrón de diseño "Template Method" en el método de refactorización que se plantea como solución en esta tesis.

Cabe aclarar que el factor de acoplamiento que se mide en esta tesis, se refiere específicamente al tipo de acoplamiento por llamadas a métodos entre clases relacionadas por herencia, ya que únicamente se miden los canales de comunicación en la jerarquía de herencia de clases. No se toma en cuenta, otro tipo de relaciones, por ejemplo, relaciones de asociación, composición y dependencias.

El sistema funciona de la siguiente manera:

1. El usuario solicita el servicio por medio de un menú de opciones y selecciona la (las) unidad (es) de programa escrita(s) en lenguaje Java que requieren de la refactorización.
2. El servicio mide el grado de acoplamiento por relaciones de herencia en arquitecturas de clases de programas legados, o marcos orientados a objetos, para aceptar la refactorización automática de las unidades de programa.
3. Se aplica el método de refactorización denominado "Reducción de Herencia de Implementación"
4. Al final, nuevamente se mide el grado de acoplamiento de las arquitecturas de clases de programa resultantes y se notifica al usuario sobre el grado de mejora en esta dimensión.

2.2.- Objetivo

2.2.1.- Objetivo General

Mejorar el diseño de arquitecturas del software legado orientado a objetos en su flexibilidad y autonomía en relación al acoplamiento, encaminadas a lograr mayor calidad para facilitar su reuso y su mantenimiento.

2.2.2.- Objetivos específicos

- Mejorar el Diseño de Arquitecturas de 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.

2.3.- Estado del Arte

Siguiendo el enfoque de refactorización, en el CENIDET se han desarrollado varios proyectos de reingeniería con el objetivo de mejorar arquitecturas de software legado escritos en lenguaje Java y C++. A continuación, se detallan, cada uno de los proyectos realizados en el departamento de ingeniería de software:

2.1.1.- *SR2-Refactoring.*

El Sistema de Reingeniería de Software Legado para Reuso (SR2-Refactoring) es la principal herramienta de antecedente que se tiene. Esta herramienta implementa diversos métodos de refactorización, que en conjunto permiten mejorar

la arquitectura de marcos de aplicaciones orientados a objetos de dominios, escritos en lenguaje Java y C++. Actualmente en el SR2-Refactoring, los métodos de refactorización implementados para refactorizar código legado escrito en lenguaje Java son: "Reducción de Herencia de Implementación" y "Separación de Interfaces". El sistema SR2-Refactoring fue desarrollado en lenguaje Java, utilizando el ambiente Eclipse, y como soporte utiliza el manejador de Base de Datos MySQL. Este sistema cuenta con manejo de usuarios, además de un menú con el cual se pueden realizar las acciones de refactorización y cálculo de métricas.

2.1.2.- El proyecto de tesis de maestría: ***"Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator"***, realizado por Leonor Adriana Cárdenas Robledo (Cárdenas, 2004). En este proyecto se planteó un método de refactorización para reducir el acoplamiento, por relaciones de asociación y composición entre clases de marcos de aplicaciones orientados a objetos escritos en lenguaje C++, incorporando a su arquitectura una estructura dirigida por el patrón de diseño 'Mediator'. El método fue implementado en la herramienta SR2 Refactoring, que realiza refactorización de manera automática, esta herramienta incluye la implementación de la métrica que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF) entre las clases antes y después de haber aplicado el proceso de refactorización.

2.1.3.- El proyecto de tesis de maestría: ***"Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces"***, realizado por Manuel Alejandro Valdés Marrero (Valdes, 2004). En el proyecto se planteó un método de refactorización denominado *"Separación de Interfaces"* para código de marco orientados a objetos escritos en C++. La intención del método es reducir el número de dependencias entre clases debido a la implementación de interfaces no utilizadas. El algoritmo de este método fue implementado satisfactoriamente en la herramienta SR2 Refactoring que hace refactorización de manera automática. El método incluye el diseño e implementación de la métrica V-DINO que mide el grado de dependencia debido a interfaces que no se ocupan.

2.1.4.- El proyecto de tesis de maestría: ***"Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter"***, realizado por Luis Esteban Santos Castillo (Castillo, 2005). En el proyecto se desarrolló un algoritmo de adaptación de interfaces entre código cliente y código servidor. Este método fue integrado al sistema SR2 Refactoring y aplica para código escrito en lenguaje C++. El algoritmo realiza una adaptación semi-automática de interfaces, que permite al usuario tomar decisiones sobre la situación de adaptación y así, ajustarla a las necesidades del código cliente.

2.1.5.- El proyecto de tesis de maestría: ***"Método de Re-factorización de código java con interfaces y abstracciones incorrectas"***, realizado por Pablo Padilla Salgado (Padilla, 2019). En este proyecto se desarrolló el sistema para el método de refactorización denominado *"Separación de Interfaces"* descrito en (Valdes, 2004), para operar sobre aplicaciones escritas en lenguaje Java. Así mismo, se desarrolló

para lenguaje Java, el cálculo automático de la métrica V-DINO que mide el grado de dependencia debido a interfaces que no se ocupan. Adicionalmente, este método equilibra la profundidad de jerarquías de clases (DIT – Depth of Inheritance Tree) con el número de clases hermanas que descienden del mismo padre (NOC – Number Of Children).

Los proyectos aquí mencionados tuvieron la finalidad de refactorizar código escritos en lenguajes de programación C++ y Java, así como extender la funcionalidad del SR2 Refactoring. El proyecto desarrollado en la tesis que se describe en este documento aporta al sistema SR2-Refactoring un método de refactorización para reducir el acoplamiento de clases relacionadas por herencia de implementación, con la finalidad de reducir el acoplamiento por herencia de implementación y mejorar el diseño de su arquitectura, además dicho método está dirigido hacia entidades de software escritas en lenguaje java.

2.4.- Trabajos Relacionados

Para la evaluación de los trabajos relacionados se realizó un estudio de 7 trabajos de investigación, en los cuales se llevaron a cabo diferentes métodos de refactorización con la finalidad de mejorar el diseño de la arquitectura de software de un sistema.

- ***Automated refactoring of super-class method invocations to the Template Method design pattern*** (Zafeiris, Poulias, Diamantidis, & Giakoumakis, 2017)

El objetivo de este trabajo de investigación se centra en la refactorización automatizada de las instancias del patrón de código "*Call super"*, cambiando su implementación hacia la estructura del patrón diseño "*Template Method*", para sustituir la herencia de implementación por la herencia de la interfaz. "*Call super"* es un patrón de código que emplea la herencia de implementación para extender el comportamiento de un método concreto. En *"Call super"* el método predominante remplaza un fragmento de código por la palabra clave *"super"*, en la clase base se ubica este fragmento de código en un método, el cual es invocado desde el método predominante a través de la palabra reservada *"super"*. Este trabajo estudia una implementación típica del patrón de código "*Call super*", donde el método sobre-escrito declara una única sentencia de SuperInvocation en su cuerpo. Los autores usan el término *"Call super"* para referirse a tales instancias del patrón de código. Los autores proponen un algoritmo de identificación que procesa la representación de un árbol de sintaxis abstracto (AST) con todas las clases del proyecto y analiza sus métodos de instancia que incluyen sentencias de SuperInvocation. El algoritmo se basa en un conjunto amplio de condiciones previas a la

refactorización, que impiden la introducción de errores de compilación y contribuyen a la preservación del comportamiento externo del sistema después de la refactorización.

- ***Automated Refactoring of Legacy Java Software to Default Methods*** (Khatchadourian & Masuhara, 2017a)**.**

El objetivo de esta publicación es presentar un enfoque de refactorización automático y basado en restricciones; que ayuda a los desarrolladores a aprovechar las interfaces mejoradas, introducidas en Java 8. Con la nueva versión de Java, los programadores pueden escribir métodos predeterminados *(Default Methods)*, que son métodos que se encuentran en interfaces y tienen una implementación que se heredará a las clases derivadas.

El enfoque infiere qué métodos se pueden migrar de manera segura a los métodos predeterminados *(Default Methods)*, a través de una formulación exhaustiva de condiciones previas de refactorización. Se presentan restricciones que involucran métodos predeterminados *(Default Methods)*. Además, se identifican todos los cambios de código necesarios para realizar la migración.

Para evaluar la efectividad del enfoque se usaron 19 aplicaciones Java de código abierto y bibliotecas de diferentes tamaños y dominios. Los resultados muestran que la herramienta fue capaz de refactorizar el 19.63% de todos los métodos que posiblemente participen en el patrón de diseño "*Skeletal Implementation"* con una intervención mínima.

- ***Defaultification refactoring: A tool for automatically converting Java methods to default.***

El propósito de la investigación de (Khatchadourian & Masuhara, 2017b) es exponer una herramienta de refactorización automática llamada *"Migrate Skeletal Implementation To Interface"* para transformar el código legado en Java para usar una nueva construcción de métodos default. La herramienta, implementada como un complemento de Eclipse, está impulsada por un enfoque de refactorización basado en restricciones, eficiente y completamente automatizado. El código resultante es semánticamente equivalente al original, más fácil de comprender, menos complejo y exhibe modularidad mejorada.

En este trabajo de investigación se hacen las siguientes contribuciones específicas:

Detalles de implementación Se presenta en detalle un tratamiento exhaustivo de los aspectos novedosos de la implementación de la herramienta. Esto incluye la arquitectura de la herramienta, uso de la API, representación de datos, algoritmos, problemas y limitaciones de implementación. Además, se explora minuciosamente la

relación de la herramienta con la implementación de la refactorización “PULL UP MEMBER”.

Perspectiva del usuario Se proporciona una visión general amplia de cómo se utiliza la herramienta para realizar refactorizaciones a gran escala.

- ***Predicting move method refactoring opportunities in object-oriented code*** (Al Dallal, 2015)**.**

La finalidad de (Al Dallal, 2015) es introducir una métrica y un modelo correspondiente para predecir con precisión si una clase incluye métodos que necesitan ser reubicados en otras 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 aplica una técnica estadística para construir modelos de predicción para las clases que incluyen métodos que necesitan reubicarse. Los modelos se aplican en siete sistemas orientados a objetos para evaluar empíricamente sus habilidades para predecir oportunidades de una refactorización de este tipo.

Los resultados muestran que los modelos de predicción basados en la métrica propuesta tienen capacidades de predicción superior y que la métrica es capaz de detectar correctamente más del 90% de los métodos que necesitan reubicarse dentro de las clases predichas. Se espera que la métrica propuesta y los modelos de predicción correspondientes ayuden en gran medida a los ingenieros de software, tanto en la localización de las clases que incluyen métodos que necesitan reubicarse como en la identificación de estos métodos dentro de las clases predichas.

- ***Automated refactoring to the Strategy design pattern.***

El trabajo de investigación de (Christopoulou, Giakoumakis, Zafeiris, & Soukara, 2012) tiene como objetivo la identificación automática de oportunidades de refactorización hacia arquitecturas que aplican el patrón de diseño Strategy y la eliminación a través del polimorfismo de los respectivos "*códigos smell*" (código desagradable) que están relacionados con el uso extensivo de enunciados condicionales complejos.

Se introduce un algoritmo para la identificación automática de oportunidades de refactorización hacia el patrón de diseño Strategy. Las refactorizaciones sugeridas se aplican a declaraciones condicionales que se caracterizan por analogías con el patrón de diseño Strategy, en términos del propósito y el modo de selección de las estrategias.

Además, este trabajo especifica el procedimiento para refactorizar hacia Strategy las declaraciones condicionales identificadas. El algoritmo de identificación y el procedimiento de refactorización se implementan e integran en el plug-in JDeodorant

Eclipse. El método se evalúa en un conjunto de proyectos Java, en términos de calidad de las refactorizaciones sugeridas y la eficiencia en tiempo de ejecución.

El método propuesto para la refactorización automatizada hacia Strategy contribuye a la simplificación de las declaraciones condicionales. Además, mejora la extensibilidad del sistema a través del patrón de diseño Strategy y confirma la eficiencia en tiempo de ejecución de este método.

- ***Automated refactoring to the Null Object design pattern.***

El propósito de (Gaitani, Zafeiris, Diamantidis, & Giakoumakis, 2015) es proponer un método novedoso para refactorizar automáticamente hacia el patrón de diseño *"Null Object"*. Este patrón elimina las condiciones de verificación nulas, asociadas con campos de clase, es decir, los campos que no se inicializan en todas las instancias de clases y, por lo tanto, su uso debe protegerse para evitar las referencias nulas.

Se especifica el procedimiento de transformación del código fuente y un amplio conjunto de condiciones previas de refactorización para refactorizar de forma segura un campo opcional y sus condicionales de verificación nula asociados al patrón de diseño *"Null Object*". El método se implementa como un complemento de Eclipse y se evalúa en un conjunto de proyectos de código abierto de Java.

Los resultados del rendimiento en tiempo de ejecución resaltan el potencial de aplicar este método a una amplia gama de proyectos de diferente escala.

Dicho método automatiza la eliminación de los condicionales de verificación nula a través de la refactorización al patrón de diseño *"NULL OBJECT"*.

- ***Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering.***

En este documento (Rathee & Chhabra, 2018) se propone una nueva métrica de cohesión para software orientado a objetos, denominada "*Usage Pattern Based Cohesion" (UPBC)*, que se calcula a nivel de módulo. Se considera la clase como un módulo inicialmente y, posteriormente, un grupo de clases, es decir, un paquete; se considera como un módulo con el objetivo de mejorar la cohesión general.

La medida del valor de cohesión se usa para realizar el agrupamiento de módulos con el fin de aumentar la cohesión y disminuir el acoplamiento entre módulos. La agrupación se realiza mediante el uso de un nuevo algoritmo de agrupamiento propuesto

llamado *“FUPClust” (Frequent Usage Pattern based Clustering)* basado en las interacciones *FUP* entre los módulos.

El enfoque propuesto se aplica a dos sistemas de software en lenguaje Java, los resultados obtenidos muestran una mejora significativa en la cohesión de ambos sistemas.

Para comparar los diferentes enfoques de los trabajos relacionados se tomaron en cuenta los siguientes aspectos:

Producto Resultante: El producto resultado del trabajo de investigación puede ser una herramienta (H), un plug-in para eclipse (P), un análisis (A) y finalmente una extensión (E) a una herramienta.

Aportación: Dentro de las aportaciones de los artículos se pueden encontrar los siguientes: métodos (M), algoritmos (A), modelos (MD) y métricas (MT).

Alcance: Los alcances se pueden clasificar de la siguiente manera, implementado (IM) e identificado (ID).

Tipos de procesos: Los procesos utilizados en los trabajos de investigación se clasifican dependiendo de la intervención del usuario en algún nivel. Se describen como: Automático (A) sin ninguna intervención del usuario, Semi-automático (S) se refiere a automático con una mínima intervención del usuario y Manual (M) se refiere a una máxima intervención del usuario en el proceso.

Métricas Usadas:

- Autonomía (MAUT): Cohesión (LCOM), Coherencia (RFC), Bajo Acoplamiento (CBO) y Pocos canales de comunicación, Estado Exclusivo y Ocultamiento de Datos.
- Modularidad (MMOD): Movilidad, Legibilidad, Complejidad (LOC, CC, WMC), Encapsulamiento de estado y comportamiento.
- Reusabilidad (MREU): Abstracción, Herencia (DIT, NOC) y Flexibilidad (DIT, NOC).

La Tabla 1 muestra una panorámica del resultado de la comparación de los diferentes trabajos de investigación.

Tabla 1.- Comparación de trabajos relacionados.

Trabajo	Objetivo	Método	Aportación	Producto Resultante	Alcance	Tipos de procesos	Métrica Usada	Meta
(Zafeiris, Poulias, Diamantidis, & Giakoumakis, 2017)	Sustituir la herencia de implementación por la herencia de interfaz.	Refactorización que transforme las llamadas “Call Super” hacia el diseño del método de la plantilla.	M, A	P	IM	A	MREU	- Acoplamiento por herencia DIT.
(Khatchadourian & Masuhara, 2017a)	Presentar resultados detallados de la experimentación de la herramienta.	Refactorización de código heredado que utilice el patrón Skeletal Implementation hacia métodos default.	M	H	IM	S	MMOD	+ Modularidad
(Khatchadourian & Masuhara, 2017b)	Sustituir el código heredado de java por la nueva construcción default.	Refactorización de código heredado que utilice el patrón Skeletal Implementation hacia métodos default.	M	H	IM	S	MMOD	+ Modularidad

(Al Dallal, 2015)	Sustituir los métodos que necesitan MMR (Move Method Refactoring).	Se aplica una técnica estadística para construir modelos de predicción para las clases que incluyen métodos que necesitan MMR.	MT	MT - MMRI (MMR Indicator) y MMRIC	ID	-	MAUT, MREU	Solo se definió el método.
(Christopoulou, Giakoumakis, Zafeiris, & Soukara, 2012)	Identificar las oportunidades de refactorización de "código smells" y sustituirlas por el patrón de diseño Strategy.	Refactorización de declaraciones condicionales hacia el diseño Strategy.	M, A	P	IM	A	MREU	+ Extensibilidad
(Gaitani et al., 2015)	Sustituir las condicionales de verificación nula por el patrón de diseño Null Object.	Refactorización de una clase que tiene, el rol de contexto y un campo que es opcional y se puede inicializar con una instancia de Null Object.	M, A	P	IM	A	MMOD	- Tiempo de ejecución
(Arcelli, Cortellessa, & Di Pompeo, 2018)	Presentar una nueva métrica de cohesión y un nuevo algoritmo de agrupamiento.	1.- Extraer el FUP para cada elemento de software. 2.- Calcula la cohesión de cada	M, A	MT - UPBC (Usage Pattern Based Cohesion)	ID	-	MAUT	+ Cohesión - Acoplamiento

		elemento de software utilizando la medida de medición de cohesión propuesta. 3.- Usar el valor de cohesión calculado para agrupar elementos en elementos más cohesivos usando los pasos algorítmicos propuestos.						
Tesis	Sustituir la herencia de implementación por la herencia de interfaz.	Refactorización que transforme las llamadas directas de código del cliente a código de librerías (call forward) hacia llamadas del código de librerías hacia código del cliente (call back).	M, A	EXT – SR2 Refactorin g	IM	A	MAUT, MREU	+ Autonomía + Reusabilidad

Como se puede observar en la Tabla 1, el sistema que se desarrolló en esta tesis tiene el objetivo de invertir llamadas directas de código de cliente a código de librerías (call forward) hacia llamadas directas de código de librerías hacia código de cliente (call back). Las principales diferencias de esta tesis con los trabajos relacionados son las siguientes:

1. El proceso se realiza desde marcos orientados a objetos o desde código legado.
2. El proceso es de "abajo hacia arriba" el cual realiza un análisis estático de código fuente, generándose un parser y un árbol de sintaxis abstracto (AST) a partir de la gramática del lenguaje java y acciones semánticas, representadas en lenguaje ANTLR4.
3. Mediante el parser se reconocen las llamadas directas, se extrae información relativa para la transformación a llamadas invertidas y la información para la aplicación de las métricas de calidad FHI, FHIJ, FHIAC, FFC y FMFAC.
4. Inversión de las llamadas directas por llamadas inversas en las cuales las librerías del marco orientado a objetos invocan o realizan llamadas a código del cliente.
5. Esta herramienta mide FHI, FHIJ, FHIAC, FFC y FMFAC tanto del código fuente del marco orientado a objetos original, como del código del marco orientado a objetos refactorizado, de tal manera que se mida el nivel de mejora estructural.
6. El diseño e implementación de las métricas de calidad FHI, FHIJ, FHIAC, FFC y FMFAC, fueron sustentadas por la teoría de la medición.

Capítulo 3.- MARCO TEÓRICO

En este capítulo se darán a conocer todos los conceptos utilizados en este trabajo de tesis, con el fin de otorgar al lector una mejor compresión del tema.

3.1.- Marco de Aplicaciones Orientado a Objetos (MAOO)

Es un conjunto de clases que trabajan colaborativamente para ofrecer soluciones genéricas, el cual puede ser ajustado a problemas específicos y servicios dentro de un mismo dominio de aplicaciones. El objetivo que se pretende alcanzar al desarrollar MAOO's, es el reuso de una arquitectura para los problemas de un dominio en particular. En la medida que el MAOO madure, se tendrá la mayoría de la funcionalidad requerida en el dominio, el cual podrá ser reusado en nuevas aplicaciones del mismo dominio. La reusabilidad del MAOO se refiere a la reutilización de sus componentes de software para reducir el tiempo de desarrollo de nuevas aplicaciones, además de reducir los costos de producción. El código desarrollado con la intención de reusarse, es el principal atributo de la reusabilidad. Además del tiempo y los costos, existen otros beneficios que la reusabilidad aporta a las industrias de software, tales como: confiabilidad y un servicio de mejor calidad (Padhy, Singh, & Chandra, 2018).

La reusabilidad es uno de los objetivos de la programación orientada a objetos. Al desarrollar un sistema utilizando el paradigma orientado a objetos, la clase proporciona los mecanismos de encapsulado, abstracción y de ocultamiento de datos, además de ser un componente elemental en la reusabilidad. Una clase proporciona mecanismos de reuso en dos niveles: como representación de una abstracción de diseño, que se puede extender o especializar, y como una fábrica de objetos que comparten la estructura y el comportamiento definido por la clase (García Peñalvo, Marqués Corral, & Maudes Raedo, 1997). A

continuación, se describen los conceptos que se manejan dentro del paradigma orientado a objetos.

3.1.1.- Clase

El paradigma orientado a objetos se basa en la noción de clase. Una clase es un elemento de software que describe un tipo de dato abstracto y su implementación parcial o total. Un tipo de dato abstracto representa una lista de operaciones o características, y a las propiedades de estas operaciones, aplicables a todas las instancias u objetos de la clase, (Meyer, 1988).

3.1.2.- Jerarquía de Clases

Es la relación de herencia entre clases, que ésta conformada por una clase base y una derivada, y ésta a su vez puede ejercer o no como clase base para otra clase derivada, y así sucesivamente. En la Figura 2 se muestra el ejemplo de una jerarquía de clases:

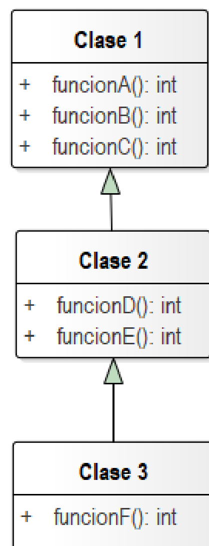


Figura 2.- Ejemplo de una jerarquía de clases.

En la Figura 3 se muestra otro ejemplo, en donde se aprecia que una arquitectura de clases contiene dos jerarquías. Las clases 1, 2 y 3 conforman la primera jerarquía y las clases 1, 2 y 4 la segunda.

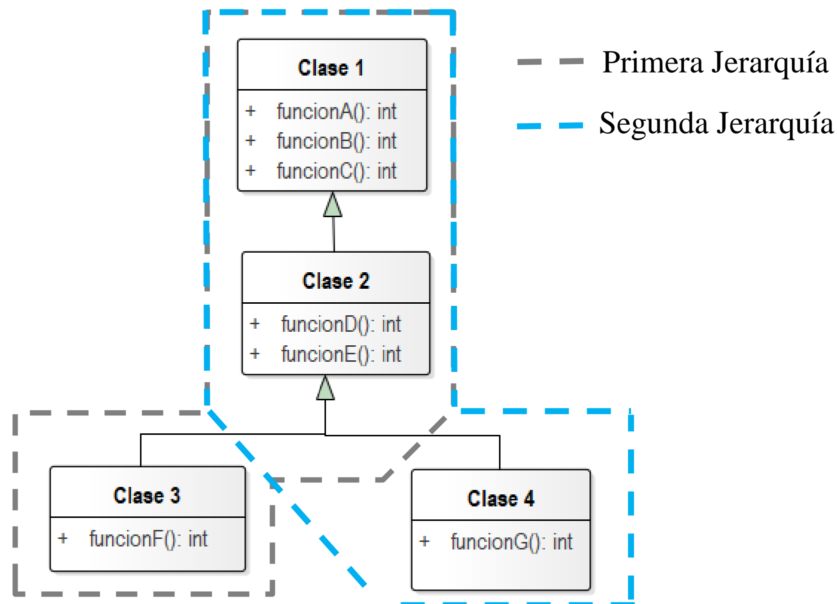


Figura 3.- Ejemplo de dos jerarquías de clases.

3.1.3.- Herencia

La propiedad más relevante del modelo orientado a objetos es el mecanismo de herencia. Utilizada para controlar la complejidad potencial resultante de una gran cantidad de clases que integran un sistema de software. Una clase es heredera de otra cuando incorpora las características de la otra además de las suyas. (Un descendiente es un heredero directo o indirecto; la noción inversa es ancestro).

La herencia es uno de los conceptos centrales de los métodos orientados a objetos y tiene profundas consecuencias en el proceso de desarrollo de software (Meyer, 1988).

3.1.4.- Polimorfismo

Polimorfismo es la propiedad del paradigma orientado a objetos para que los objetos de una clase cambien su tipo en tiempo de ejecución. De esta manera representa una ampliación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local) acepta direcciones de objetos de dicha clase y de sus clases derivadas.

3.1.5.- Abstracción

El proceso de extracción de las características esenciales de los datos mediante la definición de los tipos de datos y sus características funcionales asociadas, sin tener en cuenta los detalles de representación.

3.1.6.- Encapsulado

Representa una técnica de desarrollo de software que proporciona las especificaciones precisas para un módulo. Consiste en aislar una función del sistema o un conjunto de datos y las operaciones sobre esos datos, dentro del módulo.

3.1.7.- Ocultamiento de Datos

El mecanismo que hace que ciertos métodos o datos no sean aptos para el uso de los clientes se llama ocultamiento de datos. Es una técnica de protección de información que previene la manipulación externa de los detalles internos de representación de las diferentes entidades de software. Al describir una clase, en ocasiones se tiene que incluir a uno o más métodos o datos que se requieren proteger y que la clase necesita solo para fines internos. Una consecuencia inmediata de esta regla es que la comunicación entre clases debe ser estrictamente limitada. En particular, un buen lenguaje orientado a objetos no debería ofrecer ninguna noción de variable global; las clases intercambian información exclusivamente a través de llamadas a funciones y a través del mecanismo de herencia (Meyer, 1988).

3.2.- Principios de Diseño Orientado a Objetos

3.2.1.- Principio de Diseño *"Abierto-Cerrado"*

Este principio menciona que los módulos deben estar abiertos y cerrados. La contradicción entre los dos términos solo es aparente ya que corresponden a objetivos de una naturaleza diferente (Meyer, 1988):

- Se dice que un módulo está abierto si está disponible para tener extensión. Por ejemplo, debería ser posible expandir su conjunto de operaciones o agregar campos a sus estructuras de datos.

- Se dice que un módulo está cerrado a modificaciones a los detalles internos de su representación y solo está disponible para su uso por otros módulos. Esto supone que el módulo ha recibido una descripción estable y bien definida. En el nivel de implementación, el cierre de un módulo también implica que puede compilarlo, tal vez almacenarlo en una biblioteca y ponerlo a disposición de otros (sus clientes) para que lo utilicen. En el caso de un módulo de diseño o especificación, cerrar un módulo simplemente significa tenerlo aprobado por la administración, agregarlo al repositorio oficial del proyecto de elementos de software aceptados (a menudo llamado la línea de base del proyecto) y publicar su interfaz para el beneficio de otros módulos.

3.2.2.- Principio de Diseño *"Inversión de Dependencias"*

El principio de inversión de dependencia (DIP, por sus siglas en inglés) nos dice que los sistemas más flexibles son aquellos en los que las dependencias del código fuente se refieren solo a abstracciones, no a concreciones.

En un lenguaje de tipo estático, como java, esto significa que las declaraciones *use*, *imports* e *include* deben referirse solo a los módulos fuente que contienen interfaces, clases abstractas o algún otro tipo de declaración abstracta. No se debe depender de nada concreto.

3.3.- Refactorización

La refactorización es el proceso de cambiar un sistema de software que mejore su estructura interna, de tal manera que no altere el comportamiento externo del código. Es una forma disciplinada de limpiar el código que minimiza las posibilidades de introducir errores.

En esencia, cuando se refactoriza, se está mejorando el diseño del código después de que se ha escrito.

"Mejora del diseño después de que se haya escrito" (Martin Fowler, 2009). En la comprensión actual del desarrollo de software, se cree que se diseña y después se codifica. Un buen diseño es lo primero, y la codificación es lo segundo. Con el tiempo, el código se modificará y la integridad del sistema, junto con su estructura de acuerdo con ese diseño, se desvanecerá gradualmente. Refactorizar es lo contrario de esta práctica.

Con la refactorización, se toma código fuente degradado y volver a trabajarlo para reconsiderar su diseño arquitectural hacia su mejora. Cada paso es simple, incluso simplista. Se mueve un campo de una clase a otra, se extrae el código de un método para convertirlo en su propio método y se mueve un código hacia arriba o hacia abajo en una jerarquía. Sin embargo, el efecto acumulativo de estos pequeños cambios puede mejorar radicalmente el diseño. Es el reverso exacto de la noción normal de deterioro del software.

Con la refactorización se aprende desde la construcción del sistema, cómo mejorar el diseño. La estructura resultante conduce a un programa con un diseño que se mantiene bien a medida que continúa el desarrollo.

3.4.- Métricas Orientadas a Objetos

Las métricas orientadas a objetos son unidades de medida, que se utilizan para caracterizar productos y procesos en el ámbito de la ingeniería de software. Si se usan correctamente, permiten identificar y cuantificar posibles mejoras, además de realizar estimaciones significativas. El reciente impulso hacia la tecnología orientada a objetos obliga al crecimiento de las métricas de software orientadas a objetos (Rajnish & Bhattacherjee, 2007).

El Glosario Estándar IEEE de Términos de Ingeniería de Software (IEEE Standard Glossary of Software Engineering Terminology, 1990) define una métrica como "una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado".

La evaluación de la calidad del software implica un conjunto de procesos que consumen una gran cantidad de tiempo y recursos para mantener un nivel suficiente de calidad en el software.

Cuando los procesos se están desviando, las métricas de calidad son los indicadores que permiten a los gerentes tomar medidas proactivas para volverlos a controlar. También pueden determinar el grado de éxito o fracaso de un producto final evaluando sus atributos internos (Chawla & Chhabra, 2013).

3.5.- Patrones de Diseño

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución a ese problema, de tal manera que puede usar esta solución un millón de veces, sin tener que hacer lo mismo dos veces (Alexander et al., 1977). Aunque el autor estaba hablando de patrones en edificios y ciudades, lo que dice es cierto sobre los patrones de diseño orientados a objetos. Las soluciones se expresan en términos de objetos e interfaces en lugar de paredes y puertas, pero en el centro de ambos tipos de patrones es una solución a un problema en un contexto.

En general, un patrón tiene cuatro elementos esenciales:

1. El **nombre del patrón** es un identificador que se utiliza para describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras. Nombrar un patrón aumenta inmediatamente el vocabulario de diseño. Permite diseñar a un nivel más alto de abstracción.

2. El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Podría describir problemas de diseño específicos, como la forma de representar algoritmos como objetos. Podría describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. A veces, el problema incluirá una lista de condiciones que deben cumplirse antes de que tenga sentido aplicar el patrón.

3. La **solución** describe los elementos que componen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o implementación concreta en particular, porque un patrón es como una plantilla que se puede aplicar en muchas situaciones diferentes. En cambio, el patrón proporciona una descripción abstracta de un problema de diseño y cómo lo resuelve una disposición general de elementos (clases y objetos en este caso).

4. Las **consecuencias** son los resultados y las compensaciones de aplicar el patrón. Aunque las consecuencias a menudo no se expresan cuando se describen las decisiones de diseño, son críticas para evaluar las alternativas de diseño y para comprender los costos y beneficios de aplicar el patrón. Las consecuencias para el software a menudo se refieren a compensaciones de espacio y tiempo. También pueden abordar problemas de lenguaje e implementación. Dado que la reutilización es a menudo un factor en el diseño orientado a objetos, las consecuencias de un patrón incluyen su impacto en la flexibilidad, extensibilidad o portabilidad de un sistema. Enumerar estas consecuencias explícitamente ayuda a comprenderlas y evaluarlas.

3.5.1.- Patrón de Diseño *"Template Method"*

- **Aplicabilidad**
  El patrón Método de Plantilla debe usarse:

  - Para implementar las partes invariantes de un algoritmo una vez y dejarlo en subclases para implementar el comportamiento que puede variar.

  - Cuando el comportamiento común entre las subclases debe factorizarse y localizarse en una clase común para evitar la duplicación de código. Este es un buen ejemplo de "refactorización para generalizar" según lo descrito por Opdyke y Johnson (Opdyke & Johnson, 1993). Primero identifica las diferencias en el código existente y luego separa las diferencias en nuevas operaciones. Finalmente, reemplaza el código diferente con un método de plantilla que invoca a cada una de las nuevas operaciones.

  - Para controlar las extensiones de subclases. Puede definir un método de plantilla que invoca a las operaciones (enganchadas) en puntos específicos, permitiendo así extensiones solo en esos puntos.

- **Estructura**

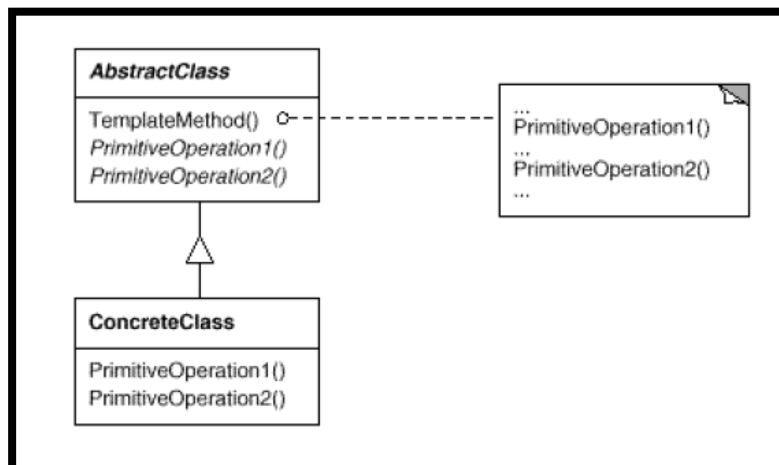


Figura 4.- Estructura del patrón de diseño del "Método de la Plantilla"

- **Participantes**

AbstractClass

- Define operaciones primitivas abstractas que las subclases concretas definen para implementar los pasos de un algoritmo.

- Implementa una método de plantilla que define el esqueleto de un algoritmo. El método plantilla llama operaciones primitivas, así como operaciones definidas en AbstractClass o las de otros objetos.

ConcreteClass

- Implementa las operaciones primitivas para llevar a cabo pasos específicos de subclase del algoritmo.

Colaboraciones

ConcreteClass se basa en AbstractClass para implementar los pasos invariantes del algoritmo**.**

- **Problema**

  Dos componentes diferentes tienen similitudes significativas, pero no demuestran reuso de una interfaz o implementación común. Si es necesario un cambio común se duplica el esfuerzo.

- **Intención**

  Define el esqueleto (plantilla) de un algoritmo en una operación, aplazando algunos pasos a subclases. El método de plantilla permite que las subclases definan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

- **Consecuencias**

  El método de plantilla es una técnica fundamental para el reuso de código. Son particularmente importantes en las bibliotecas de clases, porque son los medios para factorizar el comportamiento común en las clases de bibliotecas. El método de plantilla conduce a una estructura de control invertida que a veces se conoce como "el principio de Hollywood", es decir, "No nos llames, te llamaremos" (E. Sweet, 1985). Esto se refiere a cómo una clase padre llama a las operaciones de una subclase (call back) y no al revés (call forward).

3.6.- Teoría de la Medición

Aunque las matemáticas pueden considerarse la ciencia abstracta fundamental, siempre se ha motivado por preocupaciones en el mundo real y físico. Por lo tanto, no es sorprendente que las matemáticas incluyan una rama llamada **teoría de la medición** (Gary, 2010). Ésta se definió como la asignación de números a objetos de forma sistemática, como un medio para representar sus propiedades (Allen & Jen, 1979).

Se piensa en una medida como una forma de asociar un número, representando algunos atributos, con un objeto físico. Tal asociación, se suele llamar **mapeo** o **función** en matemáticas. Formalmente, se puede dar la siguiente definición (Gary, 2010):

1.- Sea A un conjunto de objetos físicos o empíricos. Sea B un conjunto formal objetos, como números. Una medida μ (la letra griega "mu") se define como un mapeo $\mu: A \rightarrow B$.

El requisito para que la medida sea un mapeo uno-a-uno, garantiza que cada objeto tiene una y solo una medida. No se requiere que cada número en el conjunto B, sea la medida de algún objeto en el conjunto A (Figura 5).

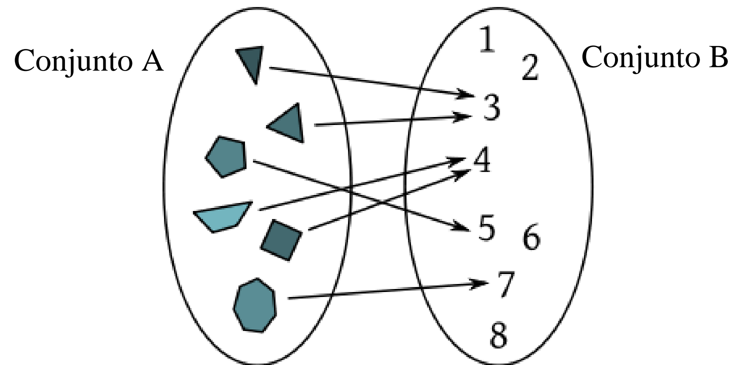


Figura 5.- Ejemplo de una función.

3.6.1.- Escalas o Niveles de Medición

Una asociación que se establece entre el mundo real y valores de medida, se suele denominar **escala de medición** (Calero, 2002). La teoría de la medición permite formalizar la relación entre dos escalas de medición con esta definición (Gary, 2010):

*Sea (A, B, µ) una escala, donde el conjunto de objetos en B, es el conjunto de los números reales. Sea que la notación µ(A) significa que el conjunto de todos los números reales que son mediciones de algún objeto en A. Esto es que µ(A) es el conjunto de medidas de los objetos en A. (En matemáticas, a esto se la llama el rango de µ.) Luego un mapeo t: µ (A) → B se define como una **transformación admisible**, si y solo si, (A, B, t°µ) también es una escala.*

Si se tiene una escala de medida para un cierto tipo de objeto, se pueden inventar otras escalas igualmente buenas, mediante la aplicación de transformaciones admisibles a la escala original. Por ejemplo, a partir de la escala Fahrenheit para medir temperatura, se puede inventar la escala Celsius aplicando la transformación t (x) = 5/9 x + 160/9. Así como también, desde una escala de longitud en pulgadas, se puede inventar una escala en centímetros, aplicando la transformación t (x) = 2.54 x. En este último ejemplo tanto la longitud en pulgadas como la longitud en centímetros, ambas son escalas, confirmando la transformación admisible.

Ahora se tiene una forma de definir la importancia de una declaración hecha sobre las medidas de objetos:

*Sea (A, B, µ) una escala, donde el conjunto de objetos en B es el conjunto de reales números. Una declaración sobre las medidas µ(a) de los objetos en A es **significativa**, si y solo si, el valor de verdad (si es verdadero o falso) de esa afirmación no cambia después de aplicar cualquier transformación admisible a µ.*

Esta definición requiere, por ejemplo, que cualquier declaración significativa sobre la longitud de un objeto medida en pulgadas también debería ser verdadera si el objeto se mide en centímetros.

3.6.2.- Clasificación de escalas

En las siguientes descripciones se asumirá que se tiene una escala (A, B, μ), donde B es el conjunto de números reales y transformaciones t. Se pueden describir cinco tipos de escalas que se caracterizan por sus transformaciones admisibles (Gary, 2010):

3.6.2.1.- Escalas Nominales

Las escalas nominales simplemente dan "nombres" numéricos a los objetos. (La palabra nominal se deriva del latin *nomin*, que significa nombre). Cualquier numeración es tan buena como cualquier otra, así que cualquier función uno-a-uno de t, es una transformación admisible. Un ejemplo de una escala nominal es: los números de camiseta de los jugadores de fútbol, lo cual únicamente significa una calificación al portador de la camiseta.

3.6.2.2.- Escalas Ordinales

Las escalas ordinales asignan números a objetos en un orden particular, pero cualquier número que mantiene ese orden es igualmente bueno. Cualquier función estrictamente creciente de t, es una transformación admisible. Un ejemplo es la escala de Mohs para la dureza de los minerales. La escala original asigna, por ejemplo, 1 al talco, 7 al cuarzo y 10 al diamante. Años después, se creó una escala que asignó 1 al talco, 8 al cuarzo y 15 al diamante. Los números difieren, pero el orden sigue siendo el mismo. En este caso se cumple que aun cuando se transforme la escala no cambia el orden creciente, por lo tanto, es una transformación admisible.

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

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

para todo P', P'' ∈ 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** → $\mathfrak{R}$, con:

$$\mathbf{P}' \bullet\geq \ \mathbf{P}'' \leftrightarrow \mu(\mathbf{P}') \geq \mu(\mathbf{P}''),$$

para todo P', P'' ∈ P**,** sí y sólo sí, •**≥** es de orden débil. Si tal **homomorfismo** existe, entonces, ((**P**, •≥),($\mathfrak{R}$,≥), μ) es una escala ordinal. Un homomorfismo es una función que preserva las operaciones definidas en los objetos de la función. Este término proviene de las palabras griegas para mismo (homos) y forma (morphe). La medida μ en una escala es un homomorfismo.

3.6.2.3.- Escalas de Intervalo

Las escalas de intervalo asignan números a los objetos, de tal manera que el intervalo entre dos valores medidos es significativo en todo el rango de valores. Solo las funciones lineales positivas $t(x) = ax + b$ son transformaciones admisibles. Las escalas de temperatura Fahrenheit y Celsius son escalas de intervalo. Una diferencia de 10 grados entre 20° y 30° y una diferencia de 10 grados entre 70° y 80° ambos significan lo mismo con respecto a la cantidad de calor que se requiere para elevar la temperatura de un objeto. La transformación de escala en grados del intervalo 20-30 al intervalo 70-80 no influye en la cantidad de calor requerida para elevar la temperatura del objeto, conservándose la linealidad.

3.6.2.4.- Escalas de Ratio (Proporción, Razón)

Las escalas de ratio asignan valores, de tal manera que la relación de dos medidas es significativa. Las únicas transformaciones admisibles son funciones lineales positivas, de la forma $t(x) = ax$. La longitud es una escala de ratio, independientemente de la unidad de medida, porque los conceptos de proporción como "dos veces más" son significativos.

3.6.2.4.- Escalas Absolutas

Las escalas absolutas tienen solo una forma de medir objetos y, por lo tanto, la única transformación admisible es la identidad $t(x) = x$. El conteo es el ejemplo más común de escala absoluta. Supóngase que se quiere medir el tamaño del personal de un proyecto de software y hacer declaraciones significativas sobre el tamaño del personal. Contar a las personas es unidad obvia. No se puede pensar otro transformación t, que no sea la transformación de identidad que podría hacer declaraciones como: "Mi proyecto tiene 5 personas" y "Mi proyecto tiene t(5) personas en él ", ambos son verdaderos, para todos los proyectos de 5 personas.

Se puede notar que esta secuencia de escalas es cada vez más restrictiva. Por ejemplo, cada escala ordinal es también una escala nominal, pero no al revés. Cada escala de intervalo es también una escala ordinal (y por lo tanto una escala nominal), pero no al revés. Las relaciones entre las clases de escalas se muestran en el siguiente diagrama de Venn.

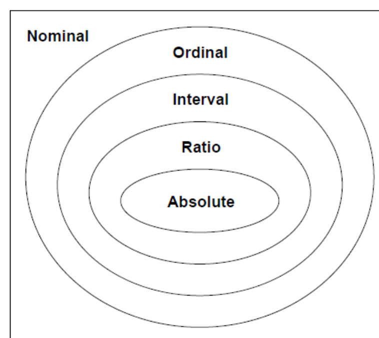


Figura 6.- Relaciones entre clases de escalas.

Capítulo 4.- MATERIALES Y MÉTODOS DE SOLUCIÓN

En este documento de tesis se realizó una investigación de las métricas enfocadas a medir la flexibilidad y el acoplamiento por herencia de implementación. En la literatura revisada no se encontraron métricas cuyo propósito sea medir el factor de flexibilidad y de herencia de implementación desde métodos abstractos. Consecuentemente, en esta tesis se definió un conjunto de cinco métricas de calidad, tres de ellas son para medir el factor de herencia de implementación: Factor de Herencia de Implementación (FHI), Factor de Herencia de Implementación por Jerarquía de Clases (FHIJ) y Factor Herencia de Implementación de una Arquitectura de Clases (FHIAC); las dos restantes son para medir el factor de flexibilidad: Factor de Flexibilidad de Clase (FFC) y Factor Medio de Flexibilidad de Clases(FMFC). En este capítulo, se muestra la descripción detallada de cada una de las métricas.

4.1.- Métrica FHI (Factor de Herencia de Implementación)

Se definió la métrica para medir el factor de herencia de implementación que tiene una clase con respecto a otra. Dada una clase derivada, esta métrica consiste en la suma de los métodos virtuales y no virtuales implementados en su clase base, entre el número total de métodos de dicha clase base. La expresión matemática de la métrica FHI se muestra a continuación:

$$FHI = \frac{\Sigma Mv + \Sigma Mnv}{Tm} \qquad (1)$$

En donde:

ΣMv = Sumatoria de métodos virtuales (no abstractos) implementados en una clase base. Estos métodos pueden/deben ser redefinidos en las clases derivadas.
ΣMnv = Sumatoria de métodos no virtuales (no abstractos) implementados en una clase base. Estos métodos no se pueden redefinir en las clases derivadas.
Tm = Total de métodos en una clase base.

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

4.2.- Métrica FHIJ (Factor de Herencia de Implementación de una Jerarquía de Clases)
Se definió una métrica para medir el factor de herencia de implementación que tiene una jerarquía de clases. Dada una jerarquía de clases, FHIJ se calcula de la siguiente manera: Primero se realiza la sumatoria de los cocientes, donde el numerador es la suma acumulada de métodos virtuales en la jerarquía de clases, y el divisor es la suma acumulada del total de métodos en la jerarquía de clases. Posteriormente, este resultado es dividido entre el número total de clases menos uno.
La expresión matemática de la métrica FHIJ se muestra a continuación:

$$FHIJ = \frac{\sum_{i}^{n}\left(\sum_{j=0}^{j=i-1} MvC_j \Big/ \sum_{j=0}^{j=i-1} MC_j\right)}{n - 1} \qquad (2)$$

En donde:
Mv = Cantidad de Métodos Virtuales de la Clases
M = Cantidad de Métodos Totales de las Clases
i, j =Representan el recorrido de la i, j-esima clase
n = Total de clases en la jerarquía

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

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

4.3.- Métrica FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases)

Se definió una métrica para medir el factor de herencia de implementación que tiene una arquitectura de clases. Dada una arquitectura de clases, esta métrica consiste en la suma de los FHIJ de las jerarquías de clases en la arquitectura, entre el número total de jerarquías de clases en la arquitectura (NOH). La expresión matemática de la métrica FHIAC se muestra a continuación:

$$FHIAC = \frac{\Sigma FHIJ}{NOH} \qquad (3)$$

En donde:
$\Sigma FHIJ$ = Sumatoria de FHIJ de una jerarquía de clases.
NOH = Es el conteo de las jerarquías de clase.

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

4.4.- Métrica FFC (Factor de Flexibilidad de Clases)

Se definió una métrica para medir el factor de flexibilidad que tiene una clase con respecto a otra. Dada una clase base, esta métrica consiste en la suma del NOP, entre el número total de sus métodos. La expresión matemática de la métrica FFC se muestra a continuación:
En donde:

ΣNOP = Sumatoria de los métodos virtuales que exhiben un comportamiento polimórfico (Bansiya & Davis, 2002).
Tm = Total de métodos en una clase base.

La métrica fue normalizada para que los valores obtenidos no dependan de la cantidad

$$FFC = \frac{\Sigma NOP}{Tm} \qquad (4)$$

de métodos de una clase dada. El mejor valor es 1, lo cual significa que todos los métodos de la clase son abstractos y/o virtuales redefinidos en las clases derivadas, por lo tanto, todos exhiben un comportamiento polimórfico. En caso contrario, el valor menos deseado es el 0, lo cual significa que no se permite la redefinición de ningún método. Es decir, ninguno de los métodos es abstracto y ninguno de los métodos es virtual.

4.5.- Métrica FMFAC (Factor Medio de Flexibilidad de Arquitecturas de Clases)

Se definió una métrica para medir el factor de flexibilidad que tiene una arquitectura de clases. Esta métrica consiste en la suma de los FFC de la arquitectura del módulo entre el número total de clases. La expresión matemática de la métrica FMFAC se muestra a continuación:

$$FMFAC = \frac{\Sigma FFC}{Tc} \tag{5}$$

En donde:
ΣFFC = Sumatoria del Factor de Flexibilidad de Clase.
Tc = Total de clases en la arquitectura.

De la misma manera el valor óptimo de FMFAC es 1, lo cual significa que todos los métodos de las clases de la arquitectura son virtuales y/o abstractos, por lo tanto, todos exhiben un comportamiento polimórfico, y el valor menos deseado de FMFAC es el 0, el cual significa que ningún método de las clases de la arquitectura es abstracto y ninguno es virtual redefinido en clases derivadas, por lo tanto, ninguno tiene un comportamiento polimórfico.

En el

Anexo A.- Métricas FHI, FHIJ, FHIAC, FFC y FMFAC como escalas ordinales, se realizan ejemplos de cálculos en diferentes arquitecturas y se comprueba que todas estas métricas cumplieron con los requerimientos para establecerse como escalas ordinales, los cuales son: que sean de orden débil y que además se cumpla la propiedad de homomorfismo. Según la teoría de la medición (Suze, 1992), estos requerimientos sustentan a las métricas.

4.6.- Refactorización por el Método de Reducción de Herencia de Implementación

En la Figura 7 se muestra el diagrama del proceso general de refactorización, que consiste de un subproceso llamado **Refactoriza Clase Base y Derivada,** dentro del carril **Refactorización de Clases.**

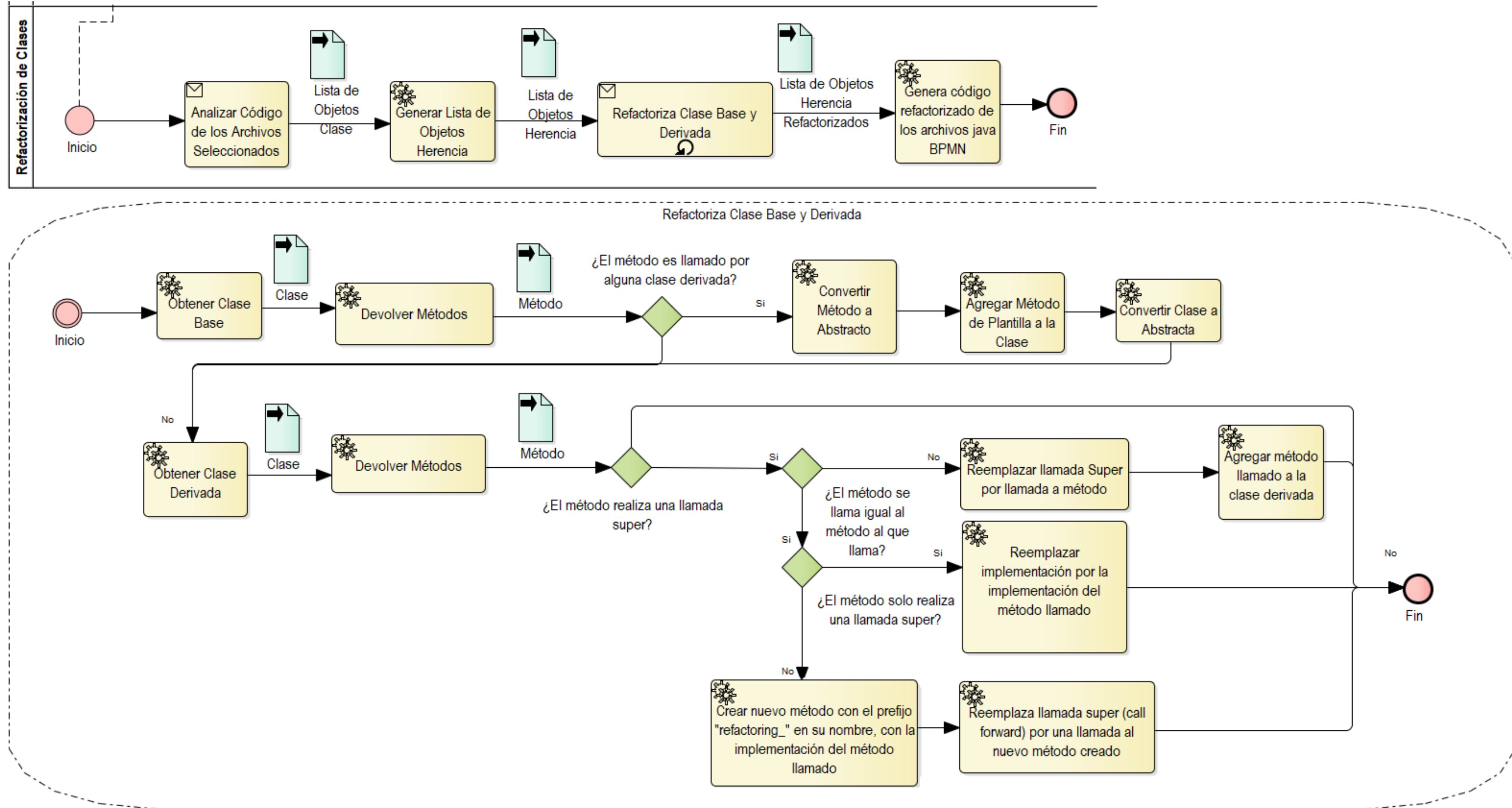


Figura 7.- Diagrama BPMN del Método de Reducción de Herencia de Implementación

En las Figura 8 y Figura 9 se puede observar una arquitectura conformada por una clase base; <aAriMean> y una clase derivada; <cAriMean>. En donde, a manera de ejemplo se hará referencia a cada paso en el sub-proceso **Refactoriza Clase Base y Derivada** de la Figura 7, para realizar el método de refactorización. Para empezar con el proceso de refactorización, se ejecuta lo siguiente:

1. De una relación de herencia, se selecciona cual es la clase base; en este ejemplo es <aAriMean>.

2. Se obtienen todos los métodos de la clase base y se guardan en una lista de métodos. En este ejemplo se almacena únicamente el método “calcula”.

3. Se detecta qué método está siendo invocado por una llamada *“super”*, en este caso es el método “calcula” de la clase base.

4. Se cambia la declaración del método “calcula” de la clase base, convirtiéndolo en un método abstracto.

5. Siguiendo la estructura del patrón de diseño “*Template Method”,* se agrega un método plantilla a la clase base <aAriMean>.

6. La declaración de la clase base se cambia a clase abstracta, abstract <aAriMean>.

7. Posteriormente, se selecciona la clase derivada de la clase base, en este caso es la clase <cAriMean>.

8. Se identifica cada uno de los métodos de la clase derivada y se guardan en una lista. En este ejemplo se almacenan en la lista los métodos: “cAriMean” y “calcula”.

9. Consecutivamente, se busca qué método está realizando una llamada *“super”,* desde la clase derivada. En este ejemplo el método llamado por la cláusula *“super”* también se llama “calcula”, y en su implementación únicamente está realizando una llamada *“super”*.

10. Por último, se reemplaza la implementación del método “calcula” de la clase <aAriMean> al cuerpo del método “calcula” de la clase <cAriMean>.

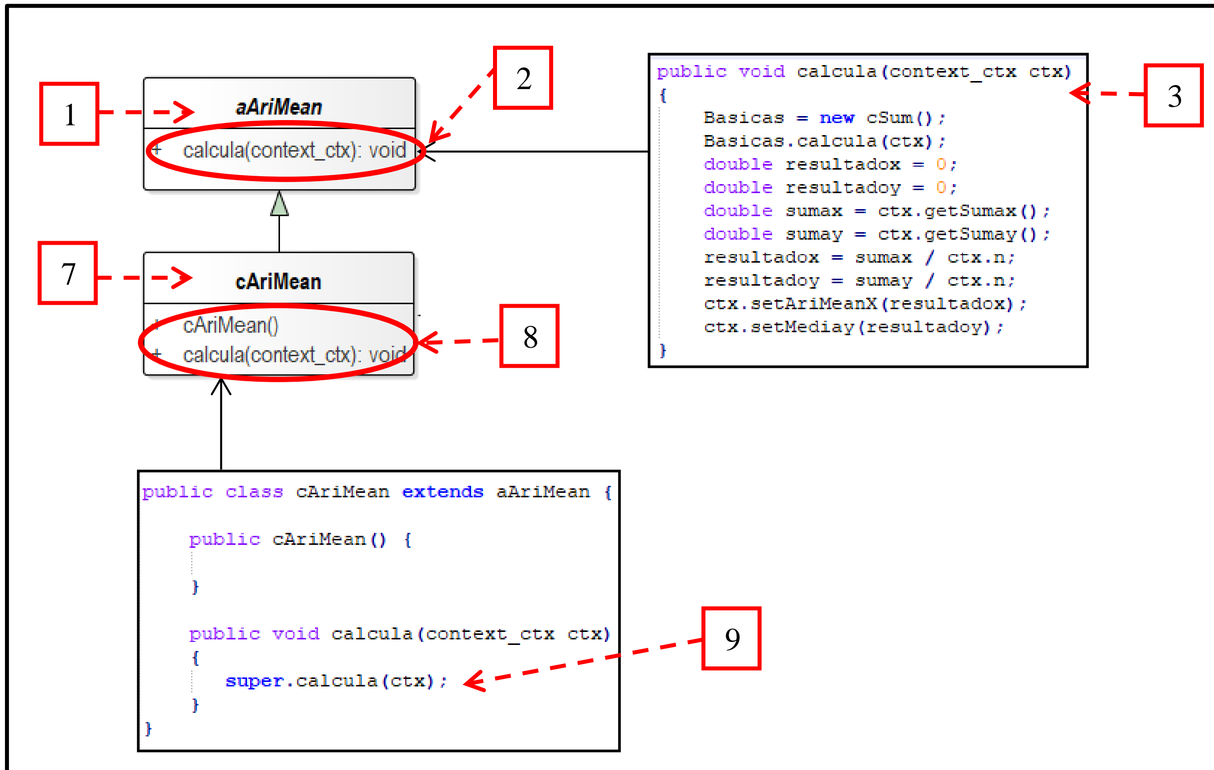


Figura 8.- Clases pertenecientes al MAOO de estadística.

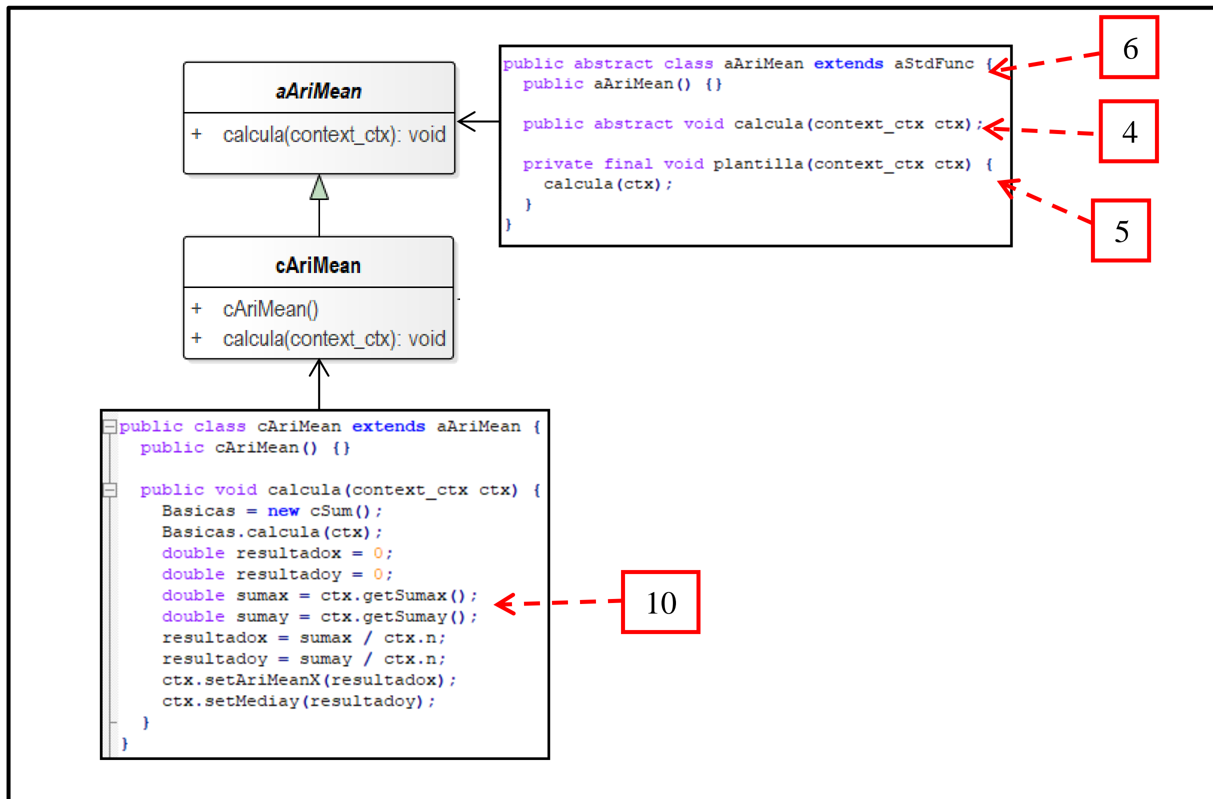


Figura 9.- Clases refactorizadas pertenecientes al MAOO estadístico.

En la Figura 10, se resalta con rojo el camino que el ejemplo anterior recorre en el subproceso **Refactoriza Clase Base y Derivada**.

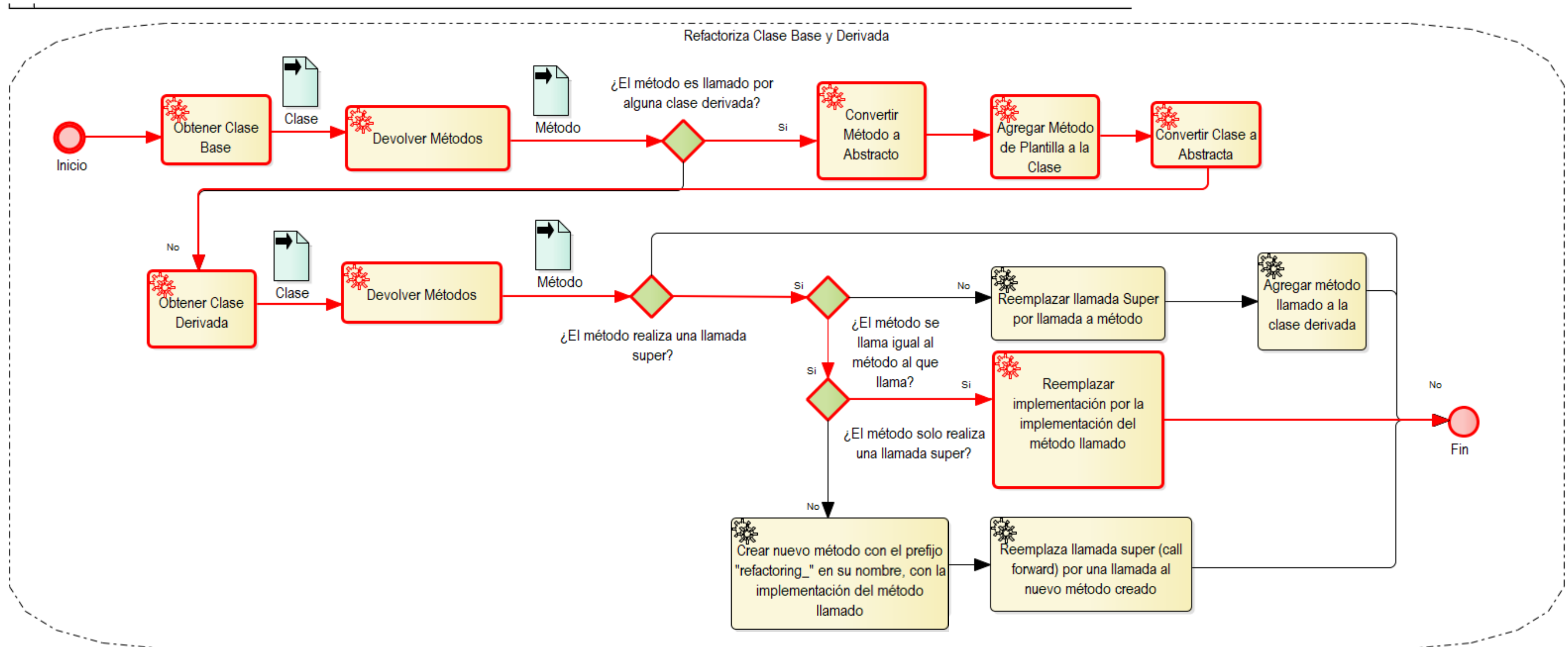


Figura 10.- Recorrido del ejemplo propuesto en el diagrama BPMN del método de refactorización.

Beneficios obtenidos por la refactorización

Los beneficios obtenidos al aplicar el método de Reducción de Herencia de Implementación, son: la disminución del acoplamiento y el aumento de la flexibilidad.

El acoplamiento disminuye a consecuencia de la inversión de dependencias que el método realiza, al invertir una *"call forward"* por una *"call back"*, la implementación de un método ya no estará dada en una clase base, sino que estará en una clase derivada. Esto genera que las clases derivadas sean independientes y no dependan de las clases base para funcionar.

La flexibilidad aumenta debido a que un método al tener su implementación en una clase derivada se puede variar su comportamiento sin modificar el código existente, solo se agregan clases con el nuevo comportamiento requerido.

Precondiciones del Método de Reducción de Herencia de Implementación

- El código del marco de aplicaciones orientado a objetos no debe contar con errores léxicos ni sintácticos.
- Solo debe de existir una llamada *"super"* en una clase derivada en una relación de herencia.
- En este trabajo de tesis el método de refactorización solo considera las relaciones de herencia de clases que estén en el mismo paquete.

Capítulo 5.- DESARROLLO DEL SISTEMA

En este capítulo se describe todo el análisis realizado para desarrollar el método de reducción de herencia de implementación. Utilizando técnicas de modelado UML como: casos de uso, diagramas de secuencias, de actividades y de clases. Así como también se describe la herramienta utilizada para realizar el análisis del código fuente.

5.1.- Análisis de casos de uso del sistema

A continuación, se presenta el análisis del sistema, modelado con diagramas de casos de uso de UML. La Figura 11 muestra el diagrama general de la herramienta, mostrando las opciones que tiene el usuario. Hay cuatro casos de uso principales: "*Manejar Archivo*" (código original), *"Análisis Sintáctico"*, *"Medición"* y *"Refactorización"*.

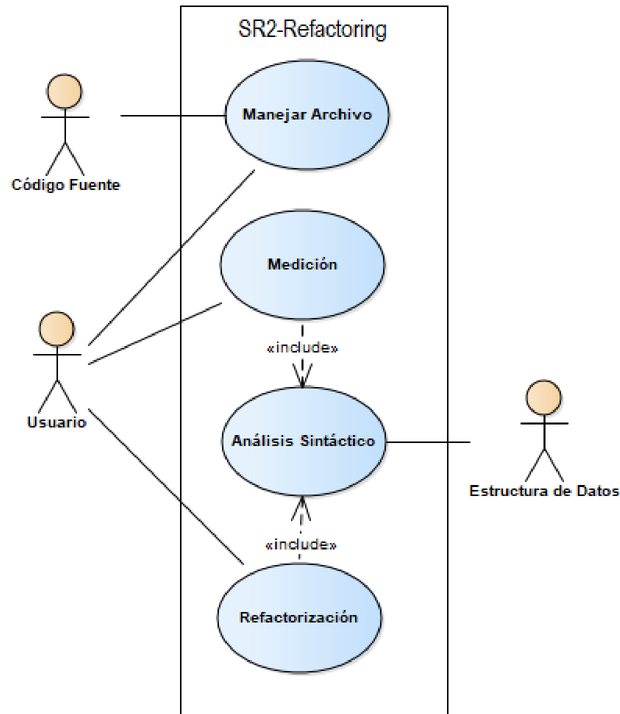


Figura 11.- Diagrama de Casos de Uso del Sistema SR2-Refactoring.

Como primer paso el usuario selecciona los archivos del marco orientado a objetos que se desea analizar y refactorizar, para después crear una copia de seguridad de dichos archivos. En este paso se aplica el caso de uso "*Manejar Archivo*" que se muestra en la Figura 12, el cual debe ser invocado antes de hacer cualquier otra operación.

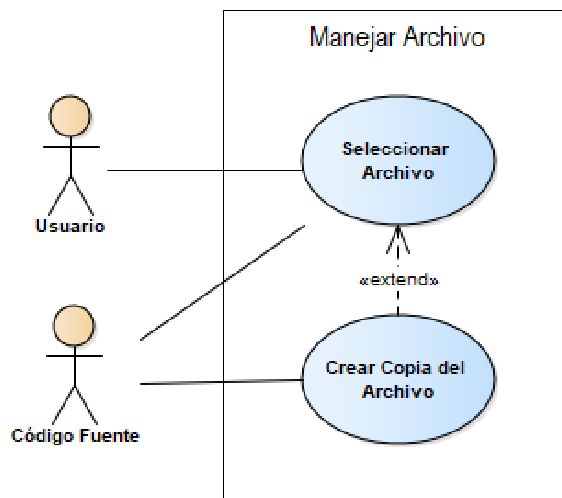


Figura 12. Diagrama del Caso de Uso Manejar Archivo.

La Figura 13 muestra el diagrama del caso de uso *"Análisis Sintáctico"*, que puede ser invocado ya sea desde la pantalla de cálculo de la métrica o la del método de refactorización. Primero se localizan las clases para obtener los métodos de cada una de ellas o se puede encontrar la cantidad de métodos implementados y no implementados en una clase. Adicionalmente, para cada método se verifica que éste realice una llamada directa de código de cliente hacia código de librerías (call forward).

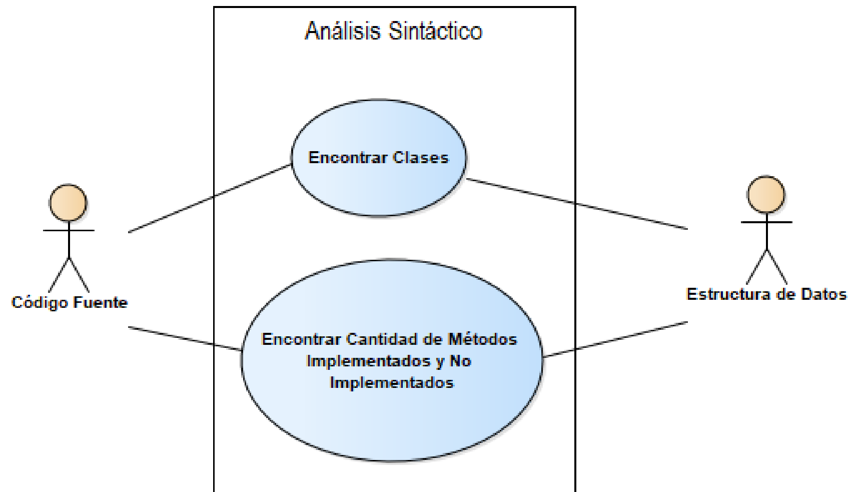


Figura 13.- Diagrama del caso de uso Análisis Sintáctico.

Después de hacer el análisis sintáctico, se procede al cálculo de métricas basándose en la información de la estructura de datos recabada en el paso anterior. La Figura 14 muestra el diagrama del caso de uso *"Medición"*, el cual es invocado después del análisis sintáctico cuando se elige la opción de Calcular Métrica.

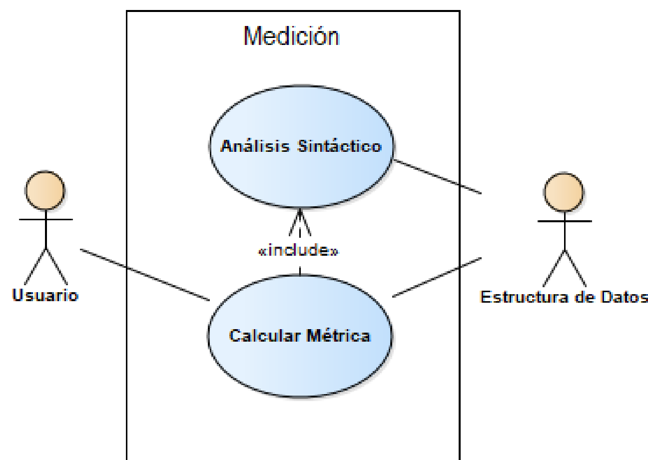


Figura 14.- Diagrama del caso de uso Medición.

La Figura 15 muestra el diagrama del caso de uso *"Refactorización"*, el cual es invocado cuando se desea realizar la refactorización. Este caso consiste en reestructurar el código, invirtiendo las llamadas directas de código de cliente a código de librerías (call forward), hacia llamadas inversas de código de librerías hacia código de cliente (call back). Este procedimiento es conducido por el patrón de diseño Template Method.

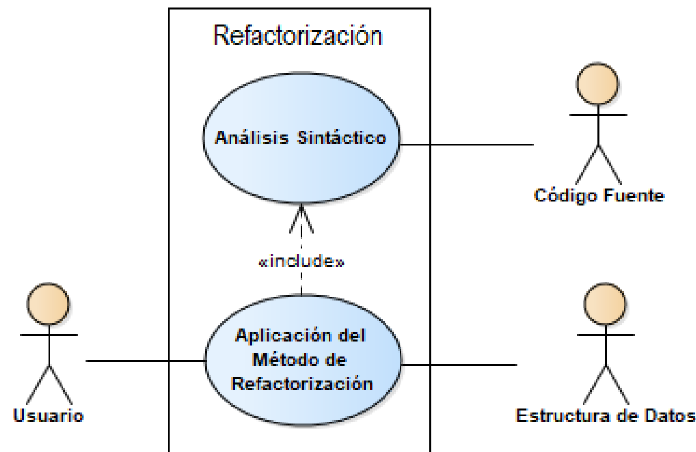


Figura 15.- Diagrama de Caso de Uso Refactorización.

5.2.- Explicación de Actores

Usuario

Representa una entidad física (persona) o una entidad lógica (sistema externo) que utiliza este método de refactorización. Existen tres acciones que puede ejecutar: Seleccionar archivos para analizarlos, ejecutando el caso de uso *"Manejar Archivo"*, solicitar el cálculo de la métrica, ejecutando el caso de uso *"Medición"*, y solicitar la ejecución del método de refactorización, ejecutando el caso de uso *"Refactorización"*.

Código Fuente

Representa a los archivos java originales que el Usuario elige abrir en el caso de uso *"Manejar Archivo"*. Son los archivos donde se encuentra el código que se desea analizar para refactorizar.

Estructura de Datos

Representa una lista de datos en memoria principal que contiene la información de los datos producidos por el caso de uso "*Análisis Sintáctico*". Esta estructura de datos será utilizada por los casos de uso "*Medición*" y "*Refactorización*".

5.3.- Diseño detallado del sistema

En esta sección se describirán a detalle las actividades encaminadas hacia el diseño del sistema que son: diagramas de secuencias, de actividades y el diseño arquitectural del sistema.

A continuación, se muestran los diagramas de secuencias de cada uno de los casos de uso, *"Manejar Archivo"*, *"Análisis Sintáctico"*, *"Medición"* y *"Refactorización"*.
En la Figura 16**,** se observa el diagrama de secuencia del caso de uso *"Manejar Archivo"*. La secuencia inicia cuando el usuario invoca el método *seleccionarArchivo* de la clase <Director>. Este método tiene la función de seleccionar los archivos originales pertenecientes al marco orientado a objetos. Para después ejecutar el método *generarCopiaSeguridad* encargado de crear una copia de seguridad de los archivos seleccionados.

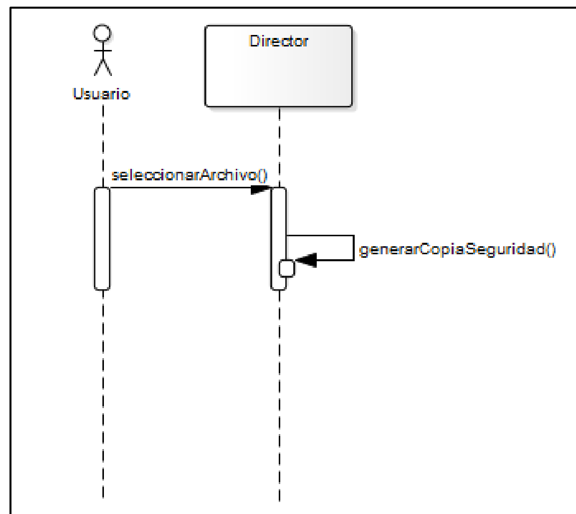


Figura 16.- Diagrama de secuencia del caso de uso Manejar Archivo.

La Figura 17 muestra el diagrama de secuencia del caso de uso *"Análisis Sintáctico"*, este caso de uso, se ejecuta después del caso de uso *"Manejar Archivo"*. Su secuencia inicia al momento que se llama a los métodos *obtenerDatos, getListaClases* y *getListaDatos* de la clase <JavaParserBaseListener>. El método "obtenerDatos" tiene la función de ejecutar el análisis sintáctico y de proveer de información a la estructura contenedora de datos. El método getListaClases, retorna una lista con la información de las clases de cada uno de los archivos previamente seleccionados. Por último, el método *getListaDatos* retorna una lista de datos esenciales para realizar el cálculo de las métricas.

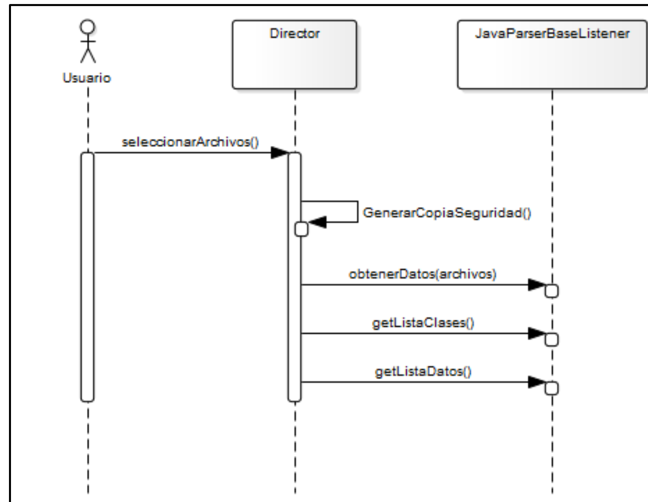


Figura 17.- Diagrama de secuencia del caso de uso Análisis Sintáctico.

En la Figura 18 se muestra el diagrama de secuencia del caso de uso *"Medición"*. Este caso debe ejecutarse después del caso de uso *"Análisis Sintáctico"*. Su comportamiento inicia al momento de llamar al método *calcular* de alguna de las siguientes clases: MetricaFHI¸ MetricaFHIJ, MetricaFHIAC, MetricaFFC o MetricaFMFAC. La función del método es la de realizar el cálculo de la métrica correspondiente y regresar el valor obtenido por dicho cálculo. El usuario asocia al objeto correspondiente el tipo de la clase según la métrica que desea calcular. La clase Sistema retorna al usuario el resultado del cálculo a través del método *mostrarResultado*.

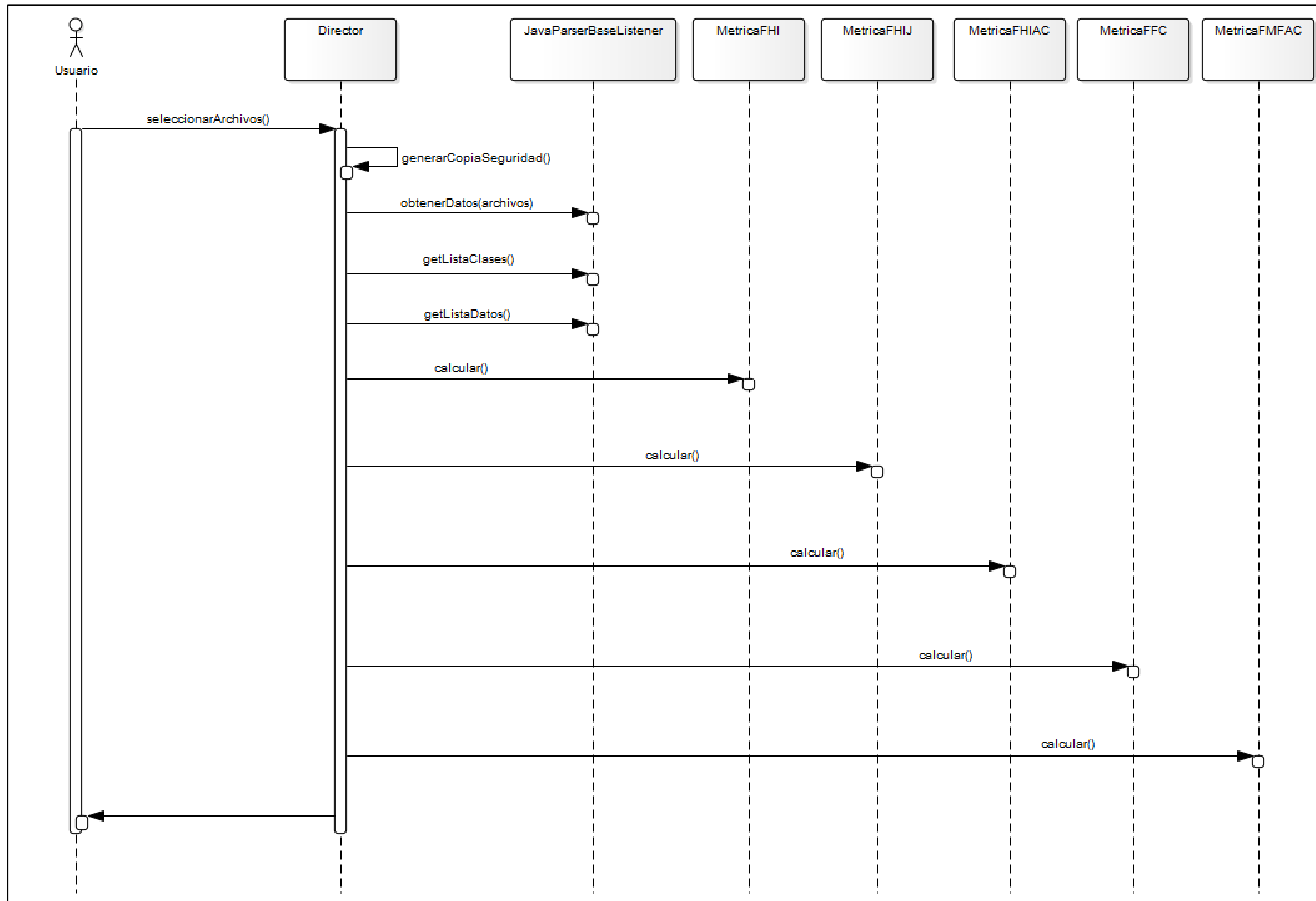


Figura 18.- Diagrama de secuencia del caso de uso Medición.

En la Figura 19 se muestra el diagrama de actividades del método “calcular” de la clase <MetricaFHI>.

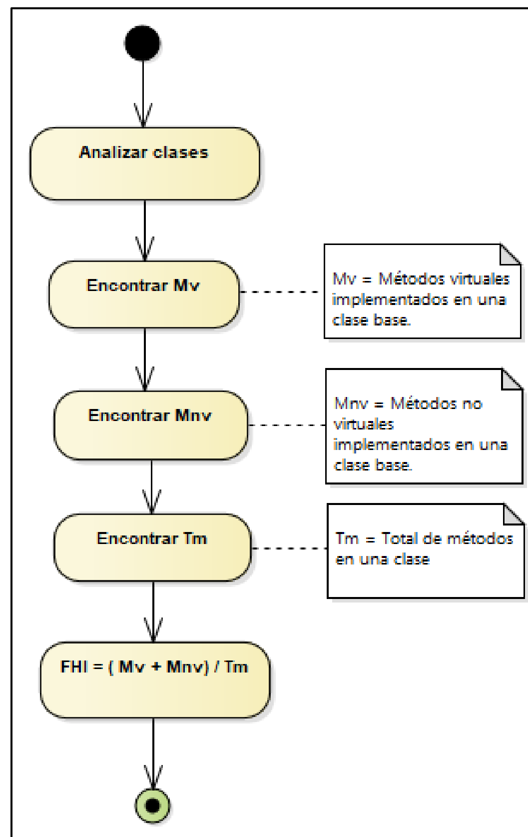


Figura 19.- Diagrama de actividades del método calcular.

En este proceso, primero se analizan las clases para contar la cantidad de métodos virtuales y no virtuales que tienen implementación, así como también, el número total de métodos que tiene esa clase, para después realizar el cálculo de la métrica FHI.

La Figura 20 muestra el diagrama de secuencia del caso de uso *"Refactorización"*. Este caso debe ser ejecutado después del caso de uso *"Análisis Sintáctico"*. Su proceso inicia cuando se llama al método "aplicarMetodoRefactorizacion" de la clase <MetodoRefactorizacion>.

Para después la clase <Director> con su método *GenerarArchivosRefactorizados* devuelva al usuario el código refactorizado.

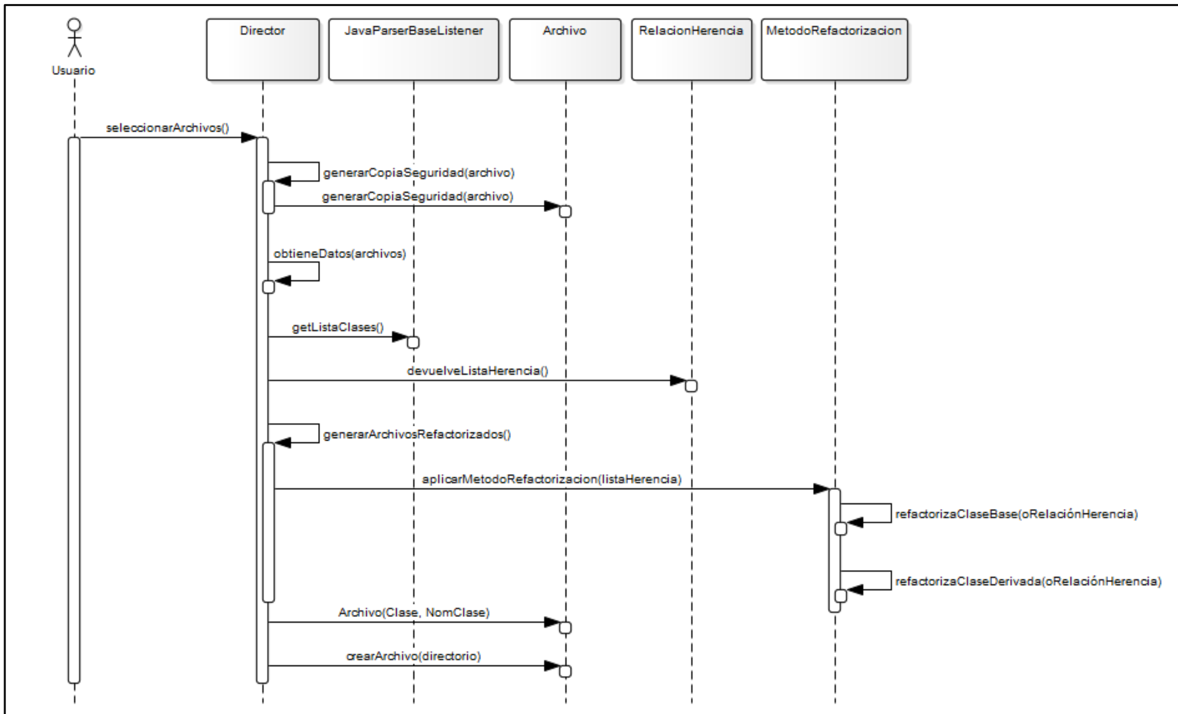


Figura 20.- Diagrama de secuencia del caso de uso Refactorización.

En la Figura 21 se muestra el diagrama de actividades del método “refactorizaClaseBase” de la clase <MetodoRefactorizacion>.

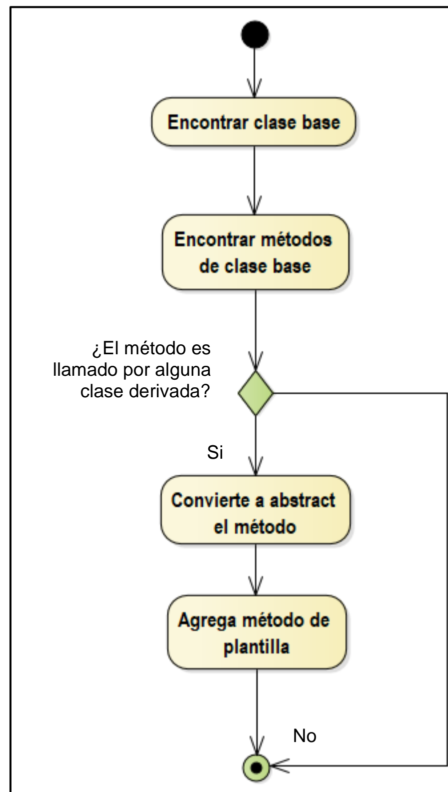


Figura 21.- Diagrama de actividades del método refactorizaClaseBase.

Para refactorizar la clase base, primero se necesita generar un objeto de tipo <Clase> perteneciente a esa clase. A continuación, se verifica si alguno de sus métodos es llamado por alguna clase derivada, siendo así, este método se convierte a abstracto y se agrega un método de plantilla a la clase base que invoca al método abstracto.

En la Figura 22 se muestra el diagrama de actividades del método "refactorizaClaseDerivada" de la clase <MetodoRefactorizacion>.

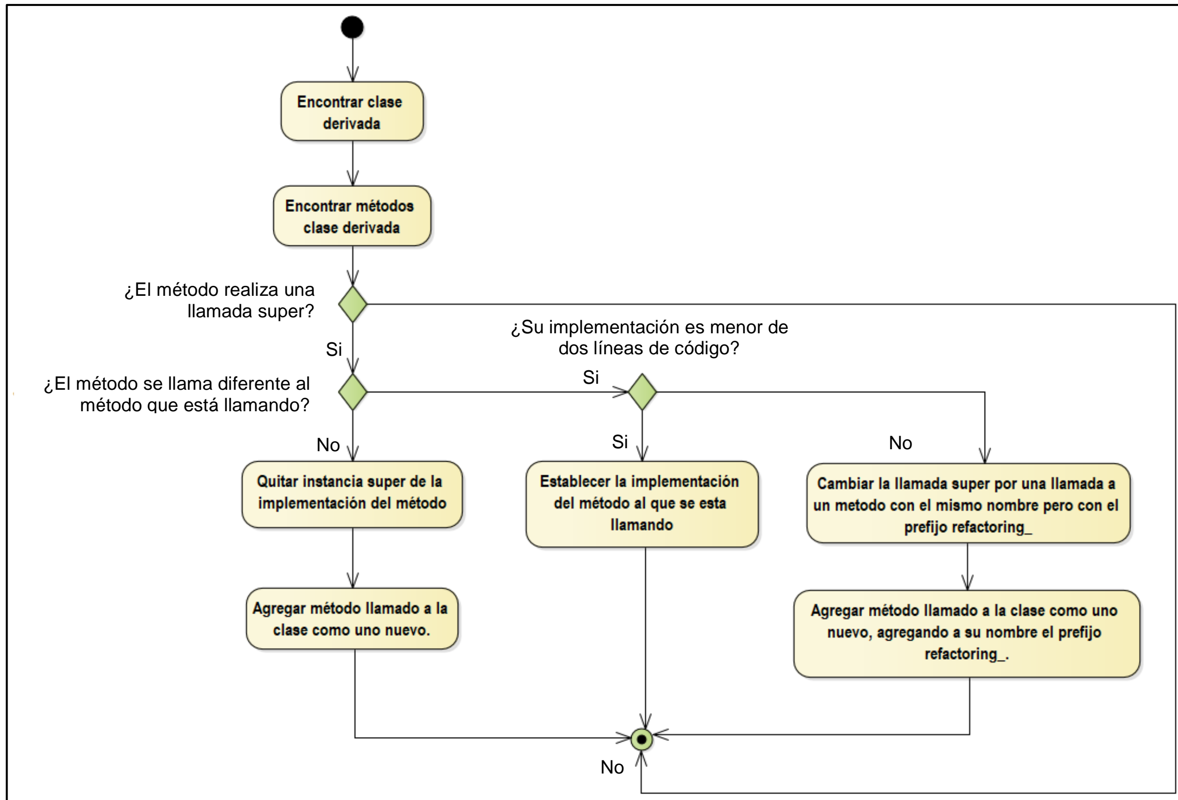


Figura 22.- Diagrama de actividades del método refactorizaClaseDerivada.

Para refactorizar la clase derivada, primero se necesita generar un objeto de tipo <Clase> perteneciente a la clase derivada. A continuación, se obtienen los métodos de esa clase, y se comprueba, si el método realiza una llamada *"super"*, se verifica si el nombre del método es igual al método que se está llamando. Si esto es falso, se quita la instancia *"super"* y se agrega el método llamado como un nuevo método. Si la verificación es cierta, se procede a saber si la cantidad de líneas de código en la implementación del método llamado es menor a dos líneas de código, si es así, se quita la llamada *"super"* y se agrega el código correspondiente a la implementación del método llamado en su lugar. Si la cantidad de líneas de código es mayor a dos líneas de código se cambia la llamada *"super"* por una llamada a un método con el mismo nombre, pero con el prefijo "refactoring_". Después, se agrega el método llamado a la clase derivada como uno nuevo, agregando a su nombre el prefijo "refactoring_".

5.4.- Diagrama de Clases del Sistema

En la Figura 23 se muestra el diagrama de clases del SR2 - Refactoring. En el diagrama se encuentran todas las clases utilizadas por los diagramas de secuencia descritos anteriormente, además de las clases entidad, utilizadas para la obtención de los datos de una clase. Las clases entidad son: <Clase>, <Metodo>, <Parametro> y <Herencia>.

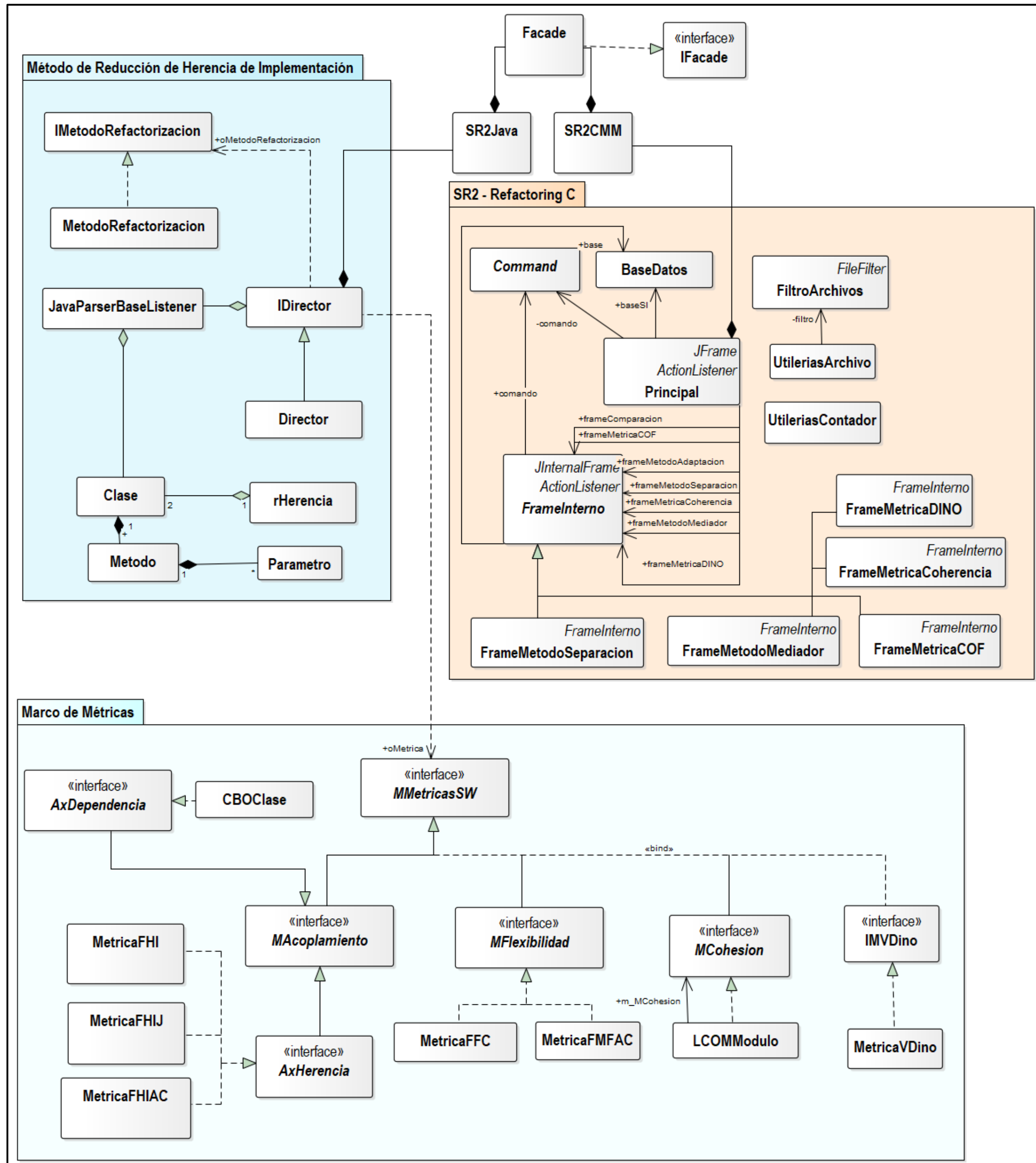


Figura 23.- Arquitectura del SR2-Refactoring.

5.5.- Descripción de las Clases Entidad

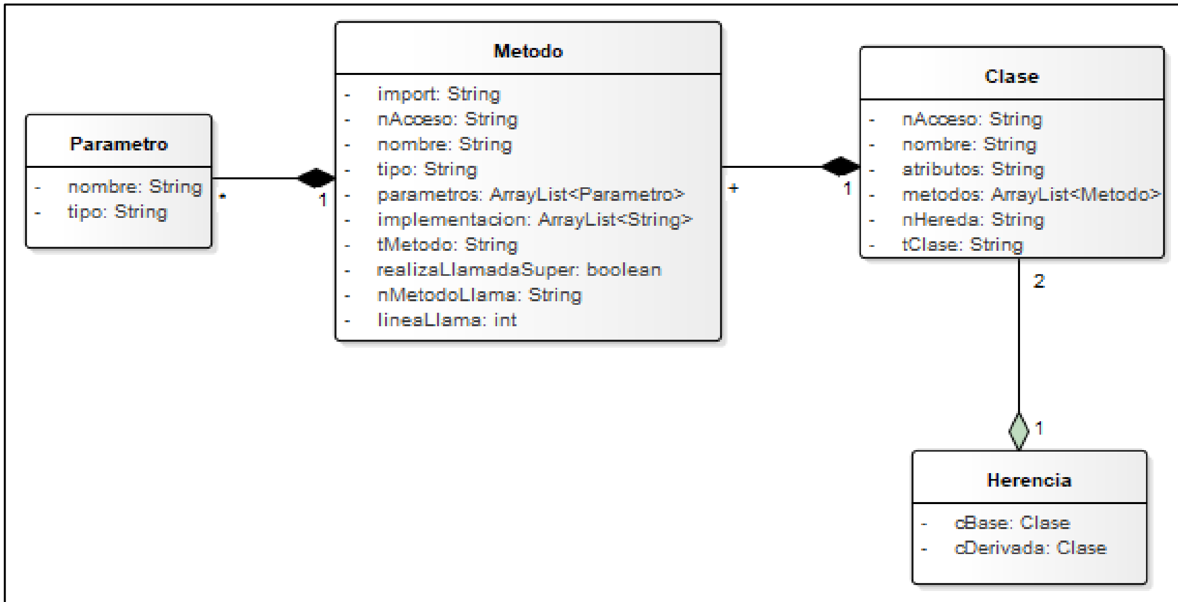


Figura 24.- Arquitectura del sistema de refactorización.

La información necesaria para que el sistema pueda aplicar el método de refactorización a un código legado o marco orientado a objetos, se modela en el diagrama de clases de la Figura 24. A continuación, se especifica cada uno de los datos que se guardarán en dichas clases, en donde, para cada uno de los datos se denota entre paréntesis el nombre del atributo seguido de su tipo de dato:

1. Para la información de clase, se utiliza la clase <Clase> y sus datos son: importaciones de librerías que tenga la clase (imports: string), nivel de acceso (nAcceso: string), nombre (nombre: string), atributos (atributos: string) (entiéndase por atributo las líneas de código que están en una clase y no representan métodos), métodos (métodos: lista<metodo>), si es el caso, el nombre de la clase a la que hereda (nHereda: string) y el tipo de clase (tClase: string).

2. Para la información de los métodos, se utiliza la clase <metodo> y sus datos son: nivel de acceso (nAcceso: string), nombre (nombre: string), tipo (tipo: string), parámetros (parametros: list<parametro>), implementación (implementacion: list<string>), cada línea de la implementación se guarda en la lista. Además, se requiere conocer el tipo de método (tMetodo: string), saber si el método está realizando una llamada *"super"* (realizaLlamadaSuper: boolean), si el caso, saber el nombre del método al que llama (nMetodoLlama: string). Así como también se requiere conocer la posición en el registro de la lista de implementación del método, que esté realizando la llamada *"super"* (lineaLlama: int).

3. El nombre y tipo de los parámetros, utilizando la clase <parametro> que está formada por el nombre (nombre: string) y el tipo (tipo: string) de un parámetro.

4. Las relaciones de herencia que existen en el marco, indicando la clase base y derivada. Estos datos se guardarán en memoria en una lista de tipo <Herencia> llamada **listaHerencia**. En donde el tipo <Herencia> está conformado por una clase base (cBase: Clase) y derivada (cDerivada: Clase).

5.6.- Análisis del Código fuente

ANTLR es una herramienta para reconocimiento de lenguajes que está escrito en java, por lo que se necesita alguna máquina virtual de java para poder ejecutarlo. Es de software libre, lo que quiere decir que al descargarlo de la página oficial (http://www.antlr.org) se obtienen tanto los archivos compilados (*.class) como el código fuente en forma de archivos (*.java). Para el reconocimiento de lenguajes cuenta con un *generador de analizadores*. A ANTLR y herramientas similares tales como JavaCC se les llama *compiladores de compiladores*, dado que ayudan a implementar compiladores.

ANTLR es capaz de generar analizadores léxicos, sintácticos o semánticos, para varios lenguajes (java, C++ y C# en su versión 2.7.2). El reconocimiento de lenguajes parte de archivos escritos en un lenguaje propio. Dicho lenguaje es básicamente una serie de reglas EBNF (Extended Backus–Naur Form) y un conjunto de construcciones auxiliares. La notación de Extended Backus–Naur Form, es un metalenguaje usado para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales (García Cota & Troyano Jiménez, 2003).

En esta investigación se realizó un estudio de la gramática del lenguaje Java desarrollado en ANTLR por (Terence & Harwell, 2018). Los archivos llamados “JavaLexer.g4” y “JavaParser.g4” fueron estudiados. Cuando se compilan estos archivos, ANTLR genera las siguientes clases:

- JavaLexer.tokens
- JavaLexer.java
- JavaLexer.interp
- JavaLexer.g4
- JavaParserListener.java
- JavaParserBaseListener.java
- JavaParser.tokens
- JavaParser.java
- JavaParser.interp
- JavaParser.g4

La clase JavaBaseParserListener, contiene dos métodos por cada regla gramatical del documento JavaParser.g4, uno para antes de analizar la regla y otro después del análisis de la regla. En dichos métodos se incorporó el código para obtener la información de la clase, para después, aplicar el método de refactorización y calcular las métricas. El método de refactorización fue integrado al sistema SR2-Refactoring como un paquete. Así como también, el método para realizar el cálculo de las métricas fue agregado a un marco orientado

a objetos desarrollado para obtener servicios de cálculos de métrica (Santaolaya Salgado, Fragoso Diaz, Ortiz Gutierrez, Bautista Juarez, & Barrera Monje, 2019). El sistema SR2-Refactoring ocupará dicho marco para obtener el valor de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC. En la Figura 23, se muestra la arquitectura del SR2-Refactoring, resaltando tres paquetes: el paquete que contiene el método de reducción de herencia de implementación, el que contiene el marco de métricas y, por último, el que contiene los métodos de refactorización para código legado escrito en lenguaje C++. El método de reducción de herencia de implementación se agregó al SR2-Refactoring como paquete, para respetar el principio de abierto-cerrado y extender su funcionalidad sin modificar el código existente.

Capítulo 6.- Pruebas

En este capítulo se describe como se planearon y realizaron las pruebas para corroborar el correcto funcionamiento del método de reducción de herencia de implementación, así como también el funcionamiento del cálculo automático de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC propuestas en este documento de tesis.

6.1.- Convención de nombres

La convención de nombres que se muestra en la Figura 25, es utilizada a través de toda la evaluación del método de reducción de herencia de implementación.

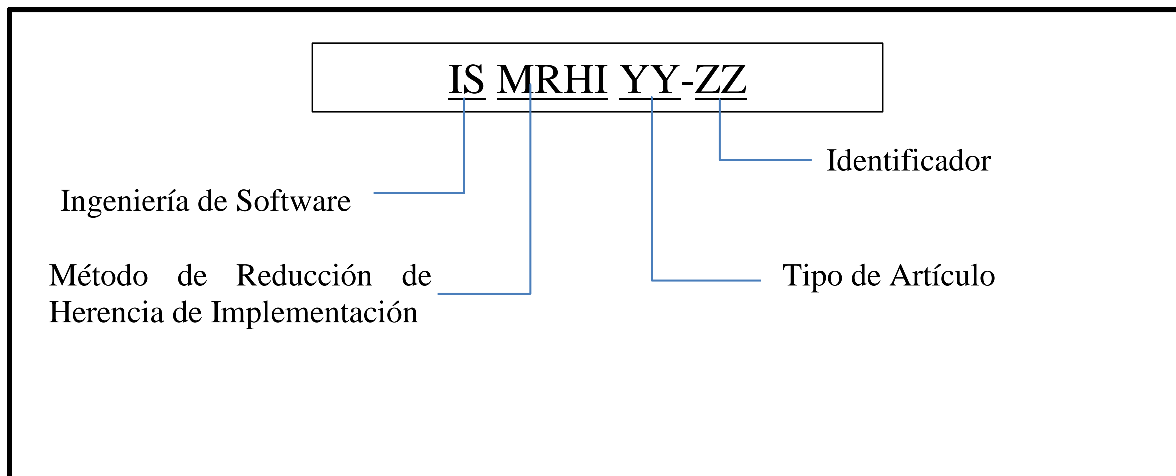


Figura 25.- Convención de nombres.

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

Módulos de programa	
ISMRHI - 0101	Módulos de programa

Programas de Control	
ISMRHI - 0201	Programas de control, utilerías, ordenadores, entre otros.

Documentación de pruebas	
ISMRHI - 0301	Plan de pruebas.
ISMRHI - 0401	Especificación de diseño de pruebas.
ISMRHI - 0501	Especificación de casos de prueba.

6.2.- Plan de pruebas

6.2.1. Plan de prueba: ISMRHI - 0301. Diseñado para el método de refactorización de reducción de herencia de implementación.

6.2.2. Introducción

El sistema de reingeniería de software para reuso (SR2-Refactoring), es un sistema que contiene la implementación automatizada de varios métodos de refactorización tanto de alta escala como de baja escala, cada uno con un objetivo y alcance específico. El método de reducción de herencia de implementación es un método de alta escala y de largo alcance, y es el que será probado a continuación.

6.2.3. Artículos de prueba. Los módulos de programa a ser probados se muestran en la

6.2.4. Tabla 2:

Tabla 2.- Módulos a probar.

Sistema	Función	No.
Método de Reducción de Herencia de Implementación	Subsistema de reestructura de código fuente.	ISMRHI - 0101
Marco de Métricas de Calidad de Arquitecturas Orientadas a Objetos	Subsistema de medición de métricas de calidad.	ISMRHI - 0102

6.2.5. Los procedimientos de control de tareas se presentan en la Tabla 3.

Tabla 3.- Características a probar.

Tipo	Función	No.
SR2-Refactoring	Identificación de relaciones de herencia	ISMRHI - 0201
SR2-Refactoring	Identificación del paquete a la que pertenece una clase	ISMRHI - 0202
SR2-Refactoring	Identificación de las librerías importadas de una clase	ISMRHI - 0203
SR2-Refactoring	Identificación de clases	ISMRHI - 0204
SR2-Refactoring	Identificación de atributos de clase	ISMRHI - 0205
SR2-Refactoring	Identificación de métodos	ISMRHI - 0206
SR2-Refactoring	Identificación de parámetros	ISMRHI - 0207
SR2-Refactoring	Identificación de sentencia *"super"*	ISMRHI - 0208
SR2-Refactoring	Identificación de la implementación de un método	ISMRHI - 0209
SR2-Refactoring	Creación del código del *"método de plantilla"*	ISMRHI - 0210
SR2-Refactoring	Reubicación de implementación de un método	ISMRHI - 0211
SR2-Refactoring	Creación del respaldo de los archivos originales	ISMRHI - 0212
SR2-Refactoring	Creación de los nuevos archivos	ISMRHI - 0213

7. Características a ser probadas

La Tabla 4 describe las características que deben ser probadas.

Tabla 4.- Características del sistema.

Diseño de prueba	Descripción
ISMRHI - 0401	Método de refactorización de código.
ISMRHI - 0402	Cálculo de métricas de calidad.

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. 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.
   - 8.3. Así mismo, no se pretende comprobar la interfaz del sistema.

9. Enfoque

La realización de los casos de prueba y la actividad de pruebas estuvieron a cargo del Ing. Orlando Ortiz Gutierrez, estudiante de maestría del Tecnológico Nacional de México / CENIDET. Esto ayudará a asegurar que las pruebas representan efectivamente el desarrollo y uso del método de reducción de herencia de implementación en sistema SR2-Refactoring completo.

9.1. Pruebas de reestructura.

La validez del método de reducción de herencia de implementación, que consta de cambiar la herencia de implementación por herencia de interfaz, se realizará mediante la ejecución del código original contra la ejecución del código refactorizado. Comprobando, que, bajo las mismas entradas (precondiciones), ambos sistemas deberán comportarse funcionalmente de la misma manera y ofrecer la misma salida (postcondiciones).

9.2. Pruebas de reducción de herencia de implementación.

Estas pruebas (de calidad) incluyen la aplicación de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC para realizar una comparación del valor obtenido de las métricas del código original contra el valor de las métricas del código refactorizado. Se espera una reducción de estos factores de calidad después de la refactorización.

10. Criterio aprobado / no-aprobado de casos de prueba.

Para los casos de prueba del proceso de refactorización, el criterio aprobado / no-aprobado, es de acuerdo con el grado de satisfacción de los resultados obtenidos en cada caso de prueba. En el caso de las pruebas de reducción de herencia de implementación, el criterio aprobado / no-aprobado será mediante la mejora de los valores de las métricas de calidad mencionadas en el punto anterior, para cada caso de prueba.

11. Criterio de suspensión y requisitos de reanudación.

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

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 ISMRHI04 – 01

1. Diseño de prueba: ISMRHI04 – 01. Método de refactorización de código.

2. Características a ser probadas
   2.1. En esta prueba lo que se evaluará es el correcto funcionamiento del método de reducción de herencia de implementación.

3. Refinamiento del enfoque

El objetivo es evaluar la reestructura del código de un sistema de software legado. En particular, que la reestructura del código, refleje una disminución en la herencia de implementación.

Antes de realizar cada caso de prueba, el sistema legado deberá ser compilado y ejecutado previamente en un compilador para el lenguaje Java, con la finalidad de corroborar que su construcción está 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.

A partir de estas estructuras de datos, el subsistema debe realizar una reestructura al código, utilizando el patrón de diseño *Template Method*.

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

   En cada caso de prueba, documentado en ISMRHI05 – XX, se especifica tanto las entradas como las salidas que el sistema requiere y arroja respectivamente. El criterio de evaluación de esta prueba se realizará tomando en cuenta las entradas y salidas del sistema original y refactorizado. Un caso de prueba debe considerarse valido cuando los resultados generados por la herramienta empaten con los resultados esperados. Para que se pase la prueba, cada característica debe pasar todos sus casos de prueba.

6.3.2.- Diseño de Prueba ISMRHI04 – 02

1. Diseño de prueba: ISMRHI04 – 02. Cálculo de métricas de calidad.

2. Características a ser probadas

   2.1. En esta prueba se evaluará el correcto funcionamiento del cálculo de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC de calidad de software.

3. Refinamiento del enfoque

El objetivo es evaluar el correcto funcionamiento del cálculo de las métricas obtenidos al evaluar un sistema de software legado.

Antes de realizar cada caso de prueba, el sistema legado deberá ser compilado y ejecutado previamente en un compilador para el lenguaje Java, con la finalidad de corroborar que la construcción de su código está 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 aprobado / no aprobado de evaluación de características

   En este caso de prueba, documentado en ISMRHI05 – XX, se verifica el resultado del valor de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC. El criterio de evaluación de esta prueba se realizará tomando en cuenta los valores de las métricas, cabe mencionar que dichos valores, son valores calculadas manualmente y verificados con los valores obtenidos automáticamente con el Marco Orientado a Objetos para el Cálculo de Métricas. Un caso de prueba debe considerarse valido cuando los resultados generados por la herramienta empaten con los resultados esperados. 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 ISMRHI05 – 01

Artículos de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje Java que tiene el objetivo de realizar automáticamente cálculos estadísticos.

2.1. Las características a probar del proceso de refactorización del método de reducción de herencia de implementación se muestran en la Tabla 5.

Tabla 5.- Características a ser probadas del proceso de reestructura.

No. de especificación del diseño de prueba	Características a ejecutar
ISMRHI04 – 01	Conversión de la clase base a abstracta.
ISMRHI04 – 01	Creación del método de la plantilla.
ISMRHI04 – 01	Desarrollo de la implementación del método de la plantilla.
ISMRHI04 – 01	Conversión a un método abstracto del método que está siendo llamado por una clase derivada.
ISMRHI04 – 01	Reubicación de la implementación del método que está siendo llamado por una clase derivada.
ISMRHI04 – 01	Creación de un método que contenga la implementación del método que está siendo llamado en la clase derivada, cuando sea necesario.
ISMRHI04 – 01	Creación de la llamada al método creado que tiene la implementación del código, cuando sea necesario.
ISMRHI04 – 01	Eliminación de la sentencia super.

3. Especificación de entrada

Las entradas al proceso de refactorización son:

1. El código fuente del marco estadístico compilado y probado previamente.

4. Especificación de salida

A la salida del proceso de refactorización se deberá tener:

1. El código fuente original refactorizado libre de la deuda técnica originada por el código desagradable que es definido como herencia de implementación en arquitecturas orientadas a objetos.

6.4.2.- Caso de Prueba ISMRHI05-02

Artículo de prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje Java que tiene el objetivo de realizar automáticamente cálculos estadísticos.

1.1. Las características a probar del proceso de cálculo de métricas de calidad se muestran en la Tabla 6.

Tabla 6.- Características a ser probadas del proceso de cálculo de métricas de calidad.

No. de especificación del diseño de prueba	Características a ejecutar
ISMRHI04 – 02	Cálculo de la métrica (FHI) Factor de Herencia de Implementación.
ISMRHI04 – 02	Cálculo de la métrica (FHIJ) factor de herencia de implementación por Jerarquía de Clase
ISMRHI04 – 02	Cálculo de la métrica (FHIAC) factor de herencia de implementación por Arquitectura de Clase
ISMRHI04 – 02	Cálculo de la métrica (FFC) factor de Flexibilidad de Clase.
ISMRHI04 – 02	Cálculo de la métrica (FMFAC) Factor Medio de Flexibilidad por Arquitectura de Clase.

3. Especificación de entrada

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

1. El código fuente 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:

1. El valor de cada una de las métricas de manera correspondiente.

6.4.3.- Caso de Prueba ISMRHI05 – 03

Artículos de Prueba: PSP Cenidet

El PSPCenidet es un sistema que mide los tiempos que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software.

2.1. Las características a probar del proceso de refactorización del método de reducción de herencia de implementación se muestran en la Tabla 7.

Tabla 7.- Características a ser probadas del proceso de reestructura.

No. de especificación del diseño de prueba	Características a ejecutar
ISMRHI04 – 01	Conversión de la clase base a abstracta.
ISMRHI04 – 01	Creación del método de la plantilla.
ISMRHI04 – 01	Desarrollo de la implementación del método de la plantilla.
ISMRHI04 – 01	Conversión a un método abstracto del método que está siendo llamado por una clase derivada.
ISMRHI04 – 01	Reubicación de la implementación del método que está siendo llamado por una clase derivada.
ISMRHI04 – 01	Creación de un método que contenga la implementación del método que está siendo llamado en la clase derivada, cuando sea necesario.
ISMRHI04 – 01	Creación de la llamada al método creado que tiene la implementación del código, cuando sea necesario.
ISMRHI04 – 01	Eliminación de la sentencia super.

3. Especificación de entrada

Las entradas al proceso de reestructura son:

1. El código fuente del marco estadístico compilado y probado previamente.

4. Especificación de salida

A la salida del proceso de reestructura se deberá tener:

1. El código fuente original refactorizado libre de la deuda técnica originada por el código desagradable que es definido como herencia de implementación en arquitecturas orientadas a objetos.

6.4.4.- Caso de Prueba ISMRHI05-04

Artículo de prueba: PSP Cenidet

El PSPCenidet es un sistema que mide los tiempos que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software.

1.1. Las características a probar del método de reducción de herencia de implementación se muestran en la Tabla 6.

Tabla 8.- Características a ser probadas del proceso de cálculo de métricas de calidad.

No. de especificación del diseño de prueba	Características a ejecutar
ISMRHI04 – 02	Cálculo de la métrica (FHI) Factor de Herencia de Implementación.
ISMRHI04 – 02	Cálculo de la métrica (FHIJ) factor de herencia de implementación por Jerarquía de Clase
ISMRHI04 – 02	Cálculo de la métrica (FHIAC) factor de herencia de implementación por Arquitectura de Clase
ISMRHI04 – 02	Cálculo de la métrica (FFC) factor de Flexibilidad de Clase.
ISMRHI04 – 02	Cálculo de la métrica (FMFAC) Factor Medio de Flexibilidad por Arquitectura de Clase.

3. Especificación de entrada

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

1. El código fuente 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:

1. El valor de cada una de las métricas de manera correspondiente.

6.5.- Ejecución de Pruebas

6.5.1.- Caso de Prueba ISMRHI05-01

Artículo de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje Java que tiene el objetivo de realizar automáticamente cálculos estadísticos.

Método de Reducción de Herencia de Implementación

Para comprobar que el método de reducción de herencia de implementación no afectó el comportamiento del marco estadístico después de ser refactorizado, se calculó la media aritmética de dos listas de números, antes y después de la refactorización. En la Tabla 9 se muestran los datos utilizados en ambas listas.

Tabla 9.- Lista de datos.

Lista 1	Lista 2
10	15
56	4
26	23
37	55
44	63
82	92
36	54
65	43
71	52
6	62

En la Tabla 10 se muestran los resultados obtenidos antes y después de la refactorización, y como se puede observar, ambos resultados fueron el mismo en ambas listas.

Tabla 10.- Resultado del cálculo de la media aritmética para la lista uno y dos.

# Lista	Media Aritmética	
	Antes de la refactorización	Después de la refactorización
Lista 1	43.3	43.3
Lista 2	46.3	46.3

Adicionalmente, como parte del proceso de comprobación del funcionamiento del marco estadístico, en las Figura 26 y Figura 27, se observa que la media aritmética obtenida del marco estadístico se conserva antes y después de la refactorización respectivamente.

```
run:
Lista 1: 43.3
Lista 2: 46.3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 26.- Captura del resultado de la media aritmética antes de la refactorización.

```
run:
Lista 1: 43.3
Lista 2: 46.3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 27.- Captura del resultado de la media aritmética después de la refactorización.

Se midió el factor de flexibilidad y el de acoplamiento por herencia de implementación del marco estadístico, antes y después de realizar la refactorización. La Tabla 11 muestra la comparación de los valores de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC.

Tabla 11.- Comparación de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC.

Métricas	Código Original	Código Refactorizado
FHI	1	0
FHIJ	0.25	0
FHIAC	0.854	0.762
FFC	0	1
FMFAC	0.196	0.239

En el Gráfico 1 se muestra que los valores de todas las métricas de acoplamiento por herencia de implementación mejoraron, ya que bajaron su valor, acercándose a 0. El acoplamiento por herencia de implementación de todo el marco estadístico se redujo de un 85.4% a un 76.2%. El acoplamiento de todo el marco estadístico se redujo en un 9.8%. Téngase en cuenta que la clase *"aStatistic"* de la Figura 32 cuenta con 17 métodos que están heredando su implementación. Dichos métodos no están siendo llamados por sus clases derivadas a través de la cláusula "*super*", así que el método de reducción de herencia de implementación no los considera para su refactorización. Como se muestra en la §fórmula (3) de la métrica de factor de herencia de implementación por arquitectura de clases, en el divisor, se toman en cuenta el número total de jerarquías para realizar el cálculo, sin embargo, no todas las jerarquías presentan herencia de implementación, lo cual incrementa el valor del divisor (NOH) en la métrica, esto produce que al momento de obtener el cociente de la métrica se obtiene un valor relativamente pequeño.

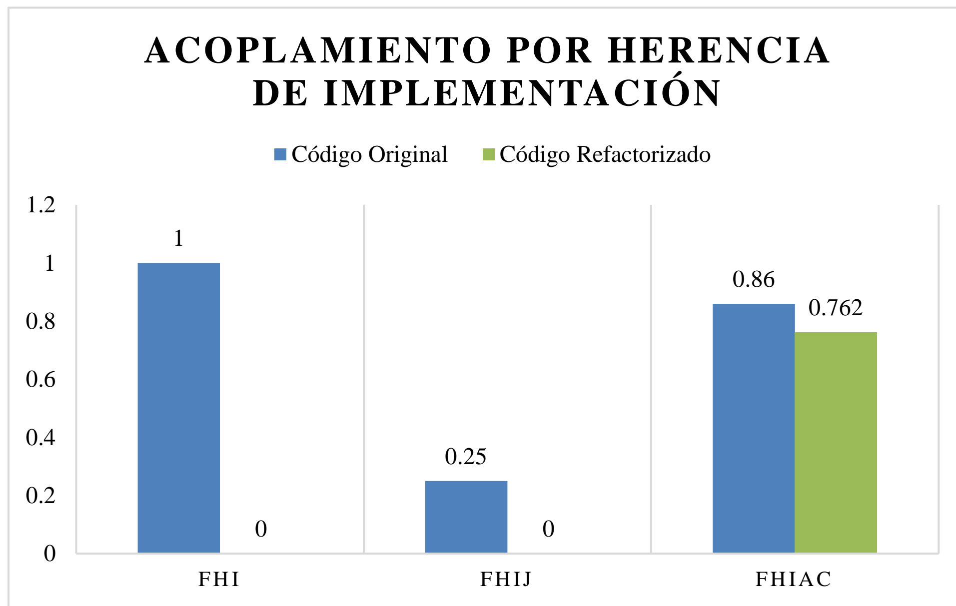


Gráfico 1.- Comparación de métricas de acoplamiento por herencia de implementación.

En el Gráfico 2 se puede observar cómo los valores de las métricas FFC y FMFAC mejoraron, al acercarse a 1. El valor de la flexibilidad de toda la arquitectura mejoró de un 19.6% a un 23.9%. Se observa que el factor medio de flexibilidad por arquitectura de clases solo presenta una mejora en un 4.3%. Esto es debido a que la §fórmula (5) de la métrica para calcular el factor medio de flexibilidad por arquitecturas de clase toma en cuenta el número total de clases en el sistema, sin embargo, no todas las clases del sistema presentan una buena medida de Factor de Flexibilidad de Clase (FFC), lo cual incrementa el valor del divisor (Tc) por lo que al momento de realizar el cociente de la métrica se obtiene un valor relativamente bajo.

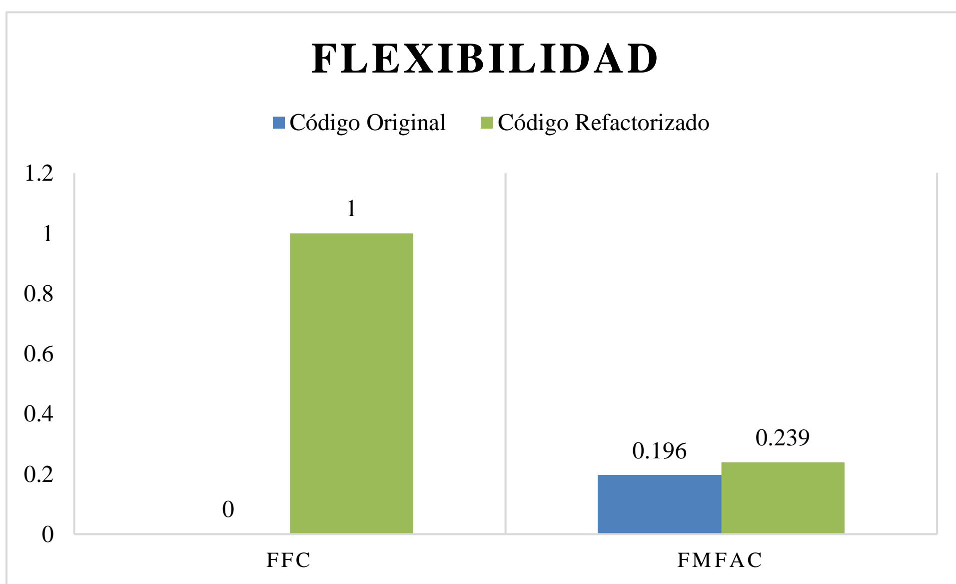


Gráfico 2.- Comparación de métricas de flexibilidad.

6.5.2.- Caso de Prueba ISMRHI05-02

Artículo de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje Java que tiene el objetivo de realizar cálculos estadísticos automáticamente.

Cálculo de la métrica (FHI) Factor de Herencia de Implementación

A manera de ejemplo se utilizaron las clases siguientes del marco estadístico en donde se realizará el cálculo de la métrica:

1. <aBetas>
2. <cGaussJordan>

La arquitectura de dichas clases se muestra en la Figura 28. En donde, los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

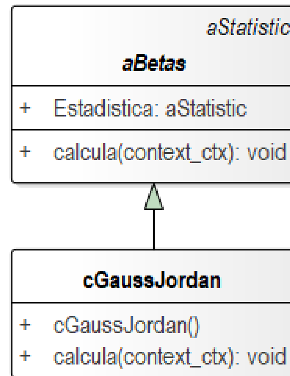


Figura 28.- Arquitectura de las clases para medir la métrica FHI.

El cálculo manual de la métrica FHI, correspondiente a la arquitectura de la Figura 28 es el siguiente:

$$FHI = \frac{\Sigma Mv + \Sigma Mnv}{\Sigma Tm} = \frac{1+0}{1} = 1 \qquad (1)$$

En donde:

Mv = Métodos virtuales implementados en una clase base.

Mnv = Métodos no virtuales implementados en una clase base.

Tm = Métodos en una clase base.

El valor óptimo de la métrica FHI es 0, que significa que no se está heredando la implementación de los métodos. En caso contrario, el valor menos deseado es el 1, el cual significa que todos sus métodos están heredando su implementación. Como se puede observar el resultado de la Formula FHI anterior calculada manualmente es 1, el cual es el valor menos deseado. Se realizó el cálculo de la métrica FHI de manera automática utilizando

el sistema SR2-Refactoring y como se puede observar en la Figura 29, el resultado también es 1.

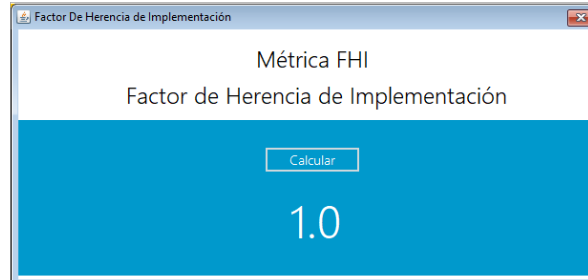


Figura 29.- Resultado de la métrica FHI utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 12, se compararon los resultados obtenidos del cálculo de la métrica FHI de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 - Refactoring pasa la prueba del cálculo de la métrica FHI.

Tabla 12.- Comparación de resultados de la métrica FHI.

Calculo Manual	Calculo Automático
1.0	1.0

Cálculo de la métrica (FHIJ) Factor de Herencia de Implementación por Jerarquía de Clase

A manera de ejemplo se utilizaron las clases siguientes del marco estadístico en donde se realizará el cálculo de la métrica:

1. <aStdFunc>
2. <aAriMean>
3. <cAriMean>

La arquitectura de dichas clases se muestra en la Figura 30. En donde, los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

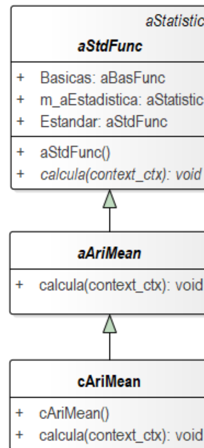


Figura 30.- Arquitectura de las clases para medir la métrica FHIJ:

El cálculo manual de la métrica FHIJ, correspondiente a la arquitectura de la Figura 30 es el siguiente:

$$FHIJ = \frac{\sum_{i}^{n}\left(\sum_{j=0}^{j=i-1} Mv \Big/ \sum_{j=0}^{j=i-1} M\right)}{n-1} = \frac{0 + 0/_1 + 1/_2}{3-1} = \frac{0+0+0.5}{2} = \frac{0.5}{2} = 0.25 \qquad (2)$$

En donde:
Mv = Cantidad de Métodos Virtuales de la Clases
M = Cantidad de Métodos Totales de las Clases
i, j =Representan el recorrido de la i, j-esima clase
n = Total de clases en la jerarquía

El valor óptimo de la métrica FHIJ es 0, lo cual significa que no se está heredando la implementación de los métodos en la jerarquía. En caso contrario, el valor menos deseado es el 1, el cual significa que todos los métodos de la clase base están heredando su implementación. Como se puede observar, el resultado del cálculo manual fue 0.25, que significa que esta jerarquía presenta herencia de implementación. Además, se realizó el cálculo de la métrica FHIJ de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 31, el resultado también fue 0.25.

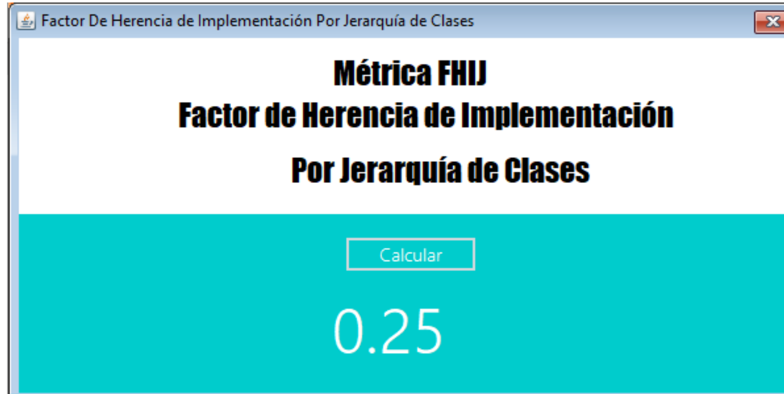


Figura 31.- Resultado de la métrica FHIJ utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 13, se compararon los resultados obtenidos del cálculo de la métrica FHIJ de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 - Refactoring pasa la prueba para el cálculo de la métrica FHIJ.

Tabla 13.- Comparación de resultados de la métrica FHIJ.

Calculo Manual	Calculo Automático
0.25	0.25

Cálculo de la métrica (FHIAC) Factor de Herencia de Implementación de una Arquitectura de Clases

Para realizar el cálculo de la métrica FHIAC se utilizó la arquitectura de clases del marco estadístico, la cual se muestra en la Figura 32. El cálculo manual de la métrica FHIAC se muestra en la Tabla 14, en donde las siglas Mv y Tm significan métodos virtuales y total de métodos respectivamente. Se excluyen las clases <Integ>, <Distribution>, <cElement> y <context_ctx> ya que son clases únicas, que no tienen ninguna relación de herencia. Así como también, no se tomó en cuenta la clase que invoca al marco y funciona como cliente.

Tabla 14.- Cálculo manual de la métrica FHIAC en el Marco Estadístico.

<table>
<tr><th>Clase base</th><th>Mv</th><th>Tm</th><th>Clase derivada</th><th>Mv</th><th>Tm</th><th>Clase derivada</th><th>Mv</th><th>Tm</th><th>Clase derivada</th><th>FHIJ</th><th>No.</th></tr>
<tr><td rowspan="4">List</td><td rowspan="4">7</td><td rowspan="4">7</td><td rowspan="4">aListMgt</td><td rowspan="4">7</td><td rowspan="4">8</td><td>cShell</td><td></td><td></td><td></td><td>0.938</td><td>1</td></tr>
<tr><td>cQuickSort</td><td></td><td></td><td></td><td>0.938</td><td>2</td></tr>
<tr><td rowspan="2">aBuble</td><td rowspan="2">7</td><td rowspan="2">9</td><td>cBubleXY</td><td>0.884</td><td>3</td></tr>
<tr><td>cBuble</td><td>0.884</td><td>4</td></tr>
<tr><td rowspan="3">aStatistic</td><td rowspan="3">17</td><td rowspan="3">17</td><td rowspan="3">aStdFunc</td><td rowspan="3">17</td><td rowspan="3">18</td><td rowspan="2">aStdDev</td><td rowspan="2">17</td><td rowspan="2">19</td><td>cDev</td><td>0.946</td><td>5</td></tr>
<tr><td>cDevY</td><td>0.946</td><td>6</td></tr>
<tr><td>cVariance</td><td></td><td></td><td></td><td>0.972</td><td>7</td></tr>
</table>

						cNormal				0.972	8
						aAriMean	18	19	cAriMean	0.964	9
			cCorrelation							1.000	10
			cRange							1.000	11
			cLinealRegress							1.000	12
			aDistribution	17	18	aDistributionX2				0.972	13
						cDistItems				0.972	14
						cDistributionX2				0.972	15
						aDistributionT	17	19	cCalt	0.946	16
									cDistribution	0.946	17
									cDistributionT	0.946	18
			aBasFunc	17	18	aSum	17	19	cSumY	0.946	19
									cSumXY	0.946	20
									cSumR	0.946	21
									cSquareSum	0.946	22
									cSum	0.946	23
			aRegression	17	18	cMultlRegress				0.972	24
			aBetas	18	18	cGaussJordan				1.000	25
aED	0	0	x							0.000	26
			xy							0.000	27
			xyz							0.000	28
									Σ =	**23.905**	
									FHIAC =	**0.854**	

Para obtener el resultado de manera manual, se obtuvo el valor de la sumatoria de las métricas FHIJ de la arquitectura y se dividió entre 28, que es el número total de jerarquías. La fórmula siguiente es la que se utilizó:

$$\text{FHIAC} = \frac{\Sigma FHIJ}{\text{NOH}} = \frac{23.905}{28} = 0.854 \qquad (3)$$

En donde:
$FHIJ$ = FHIJ de una jerarquía de clases.
NOH = Es el conteo de las jerarquías de clase.

El valor óptimo de la métrica FHIAC es 0, lo que significa que no se está heredando la implementación de métodos. En caso contrario, el valor menos deseado es el 1, el cual significa que todos los métodos están heredando implementación. Como se puede observar, el resultado del cálculo manual fue de 0.854, el cual significa que el factor de herencia de implementación de la arquitectura en la Figura 32 está cerca del valor menos deseado.

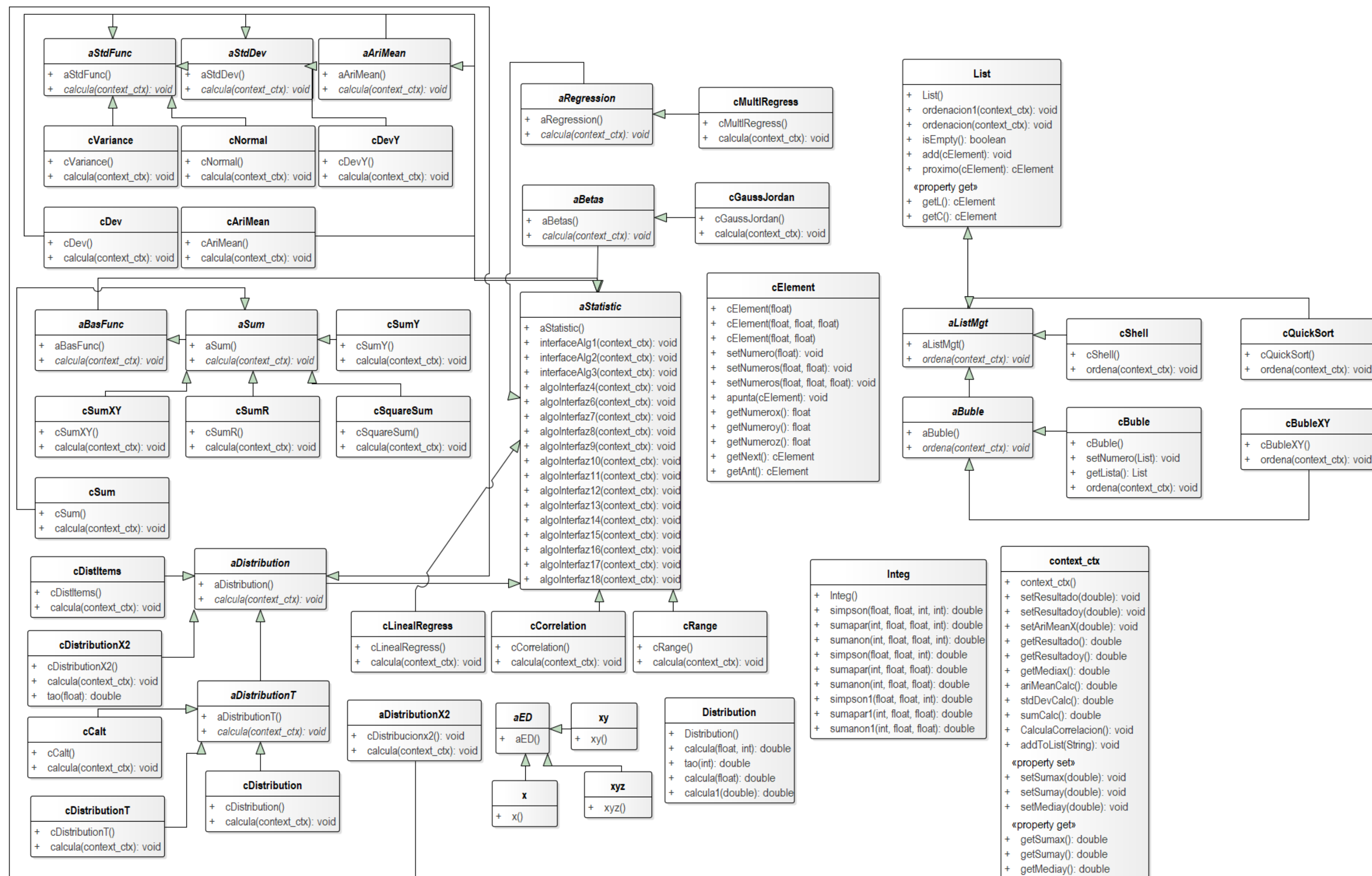


Figura 32.- Arquitectura de clases para medir la métrica FHIAC.

Se realizó el cálculo de la métrica FHIAC de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 33, el resultado fue 0.854.

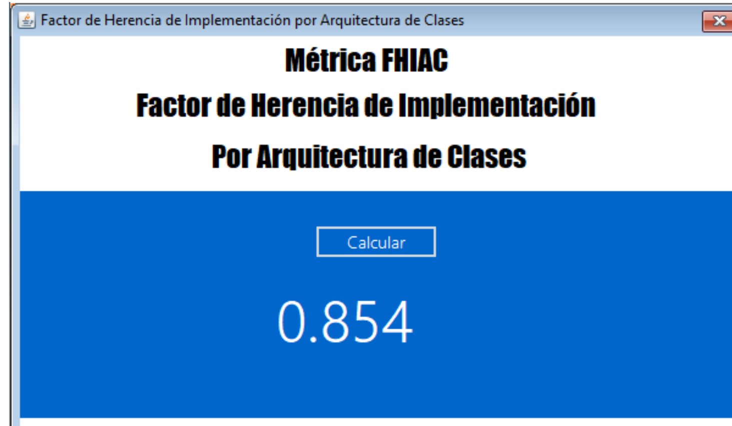


Figura 33.- Resultado de la métrica FHIAC utilizando el sistema SR2-Refactoring.

En la Tabla 15, se compararon los resultados obtenidos del cálculo de la métrica FHIAC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba del cálculo de la métrica FHIAC.

Tabla 15.- Comparación de resultados de la métrica FHIAC.

Calculo Manual	Calculo Automático
0.854	0.854

Cálculo de la métrica (FFC) Factor de Flexibilidad de Clases

Clases del marco estadístico en donde se realizará el cálculo de la métrica:

1. <aBetas>

A detalle se muestra la clase aBetas en donde los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

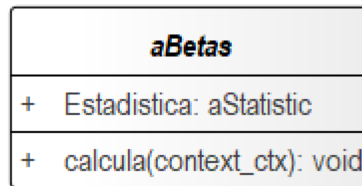


Figura 34.- Clase aBetas.

El resultado manual del cálculo de la métrica FFC, correspondiente a la clase BaseDatos es el siguiente:

$$FFC = \frac{\Sigma NOP}{\Sigma \mathrm{T}m} = \frac{0}{1} = 0 \quad (4)$$

En donde:
NOP = *Métodos virtuales que exhiben un comportamiento polimórfico.*
Tm = *Métodos en una clase base.*

El valor óptimo de la métrica FFC es 1, significa que todos los métodos de la clase son abstractos, por lo tanto, las funciones son polimórficas, lo que quiere decir que son flexibles a cambios de comportamiento. En caso contrario, el valor menos deseado es el 0, significa que ningún método es polimórfico, lo que quiere decir que no son flexibles a cambios de comportamiento.

Se realizó el cálculo de la métrica FFC de manera automática utilizando el sistema SR2-Refactoring. Como se puede observar en la Figura 35, el resultado también fue 0.

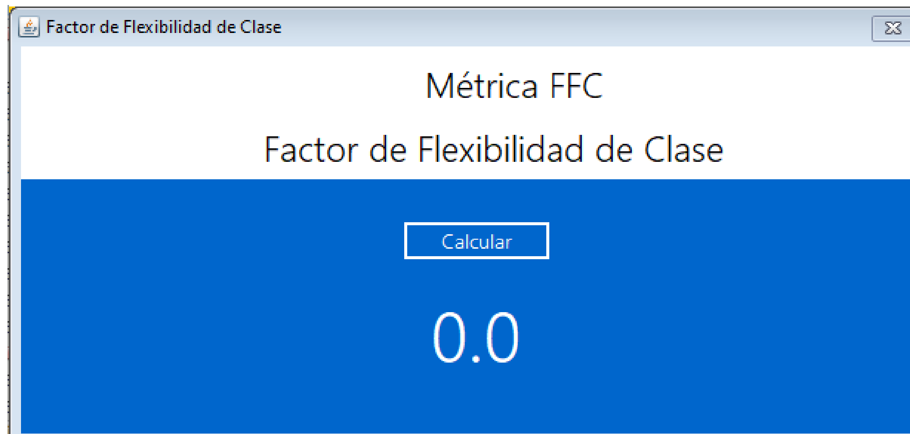


Figura 35.- Resultado de la métrica FFC utilizando el sistema SR2-Refactoring.

En la Tabla 16, se compararon los resultados obtenidos del cálculo de la métrica FFC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba del cálculo de la métrica FFC.

Tabla 16.- Comparación de resultados de la métrica FFC.

Calculo Manual	Calculo Automático
0.0	0.0

Cálculo de la Métrica (FMFAC) Factor Medio de Flexibilidad de Arquitecturas de Clases

Para el cálculo de la métrica FMFAC se utilizó la arquitectura de clases del marco estadístico, la cual se muestra en la Figura 32. El cálculo manual de la métrica FMFAC se muestra en la Tabla 17, en ella se observa el factor de flexibilidad de clase (FFC) por cada clase base.

Tabla 17.- Cálculo manual de la métrica FMFAC en el Marco Estadístico.

Clase	FFC
aAriMean	0
aBasFunc	1
aBetas	0
aBuble	1
aDistribution	1
aDistributionT	1
aED	0
aListMgt	1
aRegression	1
aStatistic	0
aStdDev	1
aStdFunc	1
aSum	1
List	0
Total =	9
FMFAC =	0.196

Para obtener el resultado de manera manual, se obtuvo el valor de la sumatoria de las métricas FFC y se dividió entre 46 que es el número total de clases en la arquitectura. La fórmula siguiente es la que se utilizó:

$$FMFAC = \frac{\Sigma FFC}{\Sigma Tc} = \frac{9}{46} = 0.196 \qquad (5)$$

En donde:

FFC = FFC de una clase que hereda su comportamiento.

Tc = Número total de clases en la arquitectura.

El valor óptimo de la métrica FMFAC es 1, significa que todos los métodos de la clase son abstractos, por lo tanto, las funciones son polimórficas, lo que quiere decir que son flexibles a cambios de comportamiento. En caso contrario, el valor menos deseado es el 0, significa que ningún método es polimórfico, lo que quiere decir que no son flexibles a cambios de comportamiento. El resultado del cálculo manual fue de 0.196, esto significa que el factor medio de flexibilidad por arquitectura de clases se encuentra muy cerca del valor menos deseado, lo cual significa que hay poca flexibilidad a cambios de comportamiento.

Se realizó el cálculo de la métrica FMFAC de manera automática utilizando el sistema SR2-Refactoring. En la Figura 36, se muestra un resultado de 0.196.

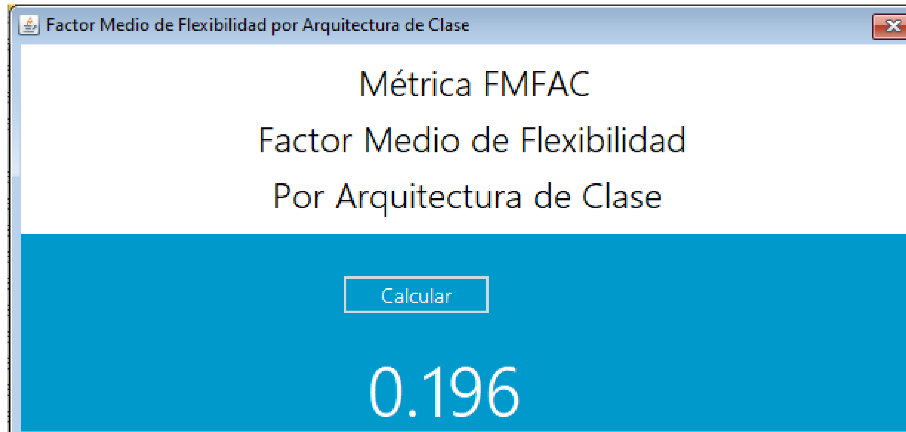


Figura 36.- Resultado de la métrica FMFAC utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 18, se compararon los resultados obtenidos del cálculo de la métrica FMFAC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba para el cálculo de la métrica FMFAC.

Tabla 18.- Comparación de resultados de la métrica FMFAC.

Calculo Manual	Calculo Automático
0.196	0.196

6.5.3.- Caso de Prueba ISMRHI05-03

Artículo de Prueba: PSP Cenidet.

El PSPCenidet es un sistema que mide los tiempos que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software.

Método de Reducción de Herencia de Implementación

Para comprobar que el método de reducción de herencia de implementación no afectó el comportamiento del PSP Cenidet después de ser refactorizado, se dio de alta un usuario y se registró una actividad, antes y después de la refactorización.

En la Figura 37 se muestran dos usuarios que existen en el sistema PSPCenidet antes de la refactorización. Los cuales son: René Santaolaya Salgado y Orlando Ortiz Gutierrez.

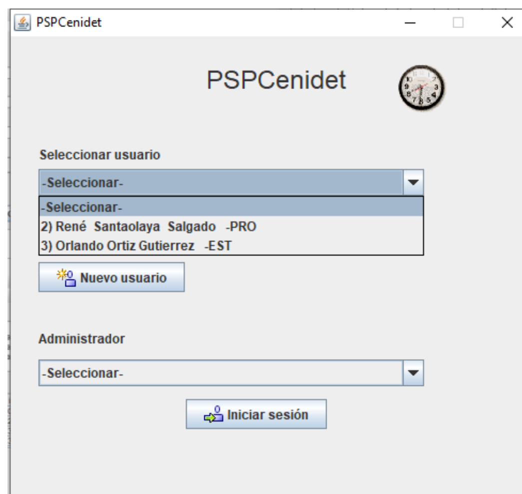


Figura 37.- Usuarios antes de la refactorización.

En la Figura 38 se muestran las cinco actividades que tiene el usuario Orlando Ortiz Gutierrez.

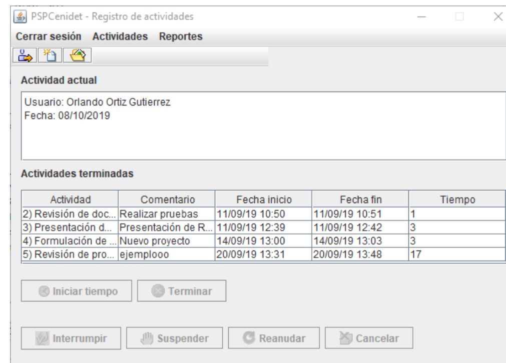


Figura 38.- Actividades del usuario Orlando Ortiz Gutierrez antes de la refactorización.

Se dio de alta el usuario Marco Valencia Vázquez después de la refactorización y en la Figura 39 se puede observar que ya se encuentra dentro de los usuarios del sistema.

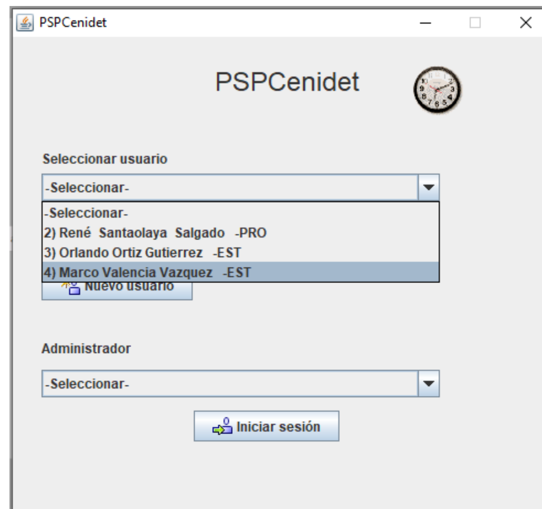


Figura 39.- Usuarios del sistema PSPCenidet.

Así como también se dio de alta una nueva actividad al usuario Orlando Ortiz Gutierrez. En la Figura 40, se puede observar que ya existe la actividad 6, que es la nueva actividad registrada.

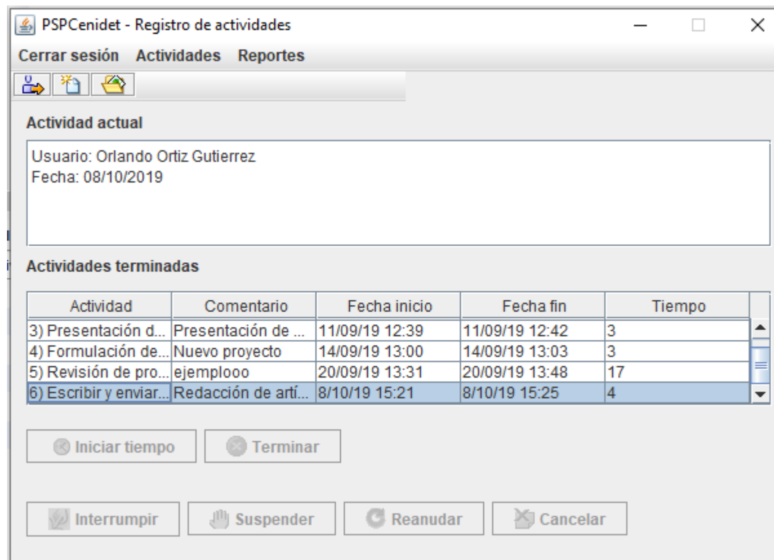


Figura 40.- Registro de actividades.

Se midió el factor de flexibilidad y el de acoplamiento por herencia de implementación del sistema PSPCenidet, antes y después de realizar la refactorización. La Tabla 19 muestra la comparación de los valores de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC.

Tabla 19.- Comparación de las métricas FHI, FHIJ, FHIAC, FFC y FMFAC.

Métricas	Código Original	Código Refactorizado
FHI	0.875	0.111
FHIJ	1	0.545
FHIAC	0.876	0.147
FFC	0.8	1
FMFAC	0.088	0.132

En el

Gráfico 3 se puede observar que los valores de todas las métricas de acoplamiento por herencia de implementación mejoraron, ya que bajaron su valor, acercándose a 0. El acoplamiento por herencia de implementación de todo el sistema PSPCenidet se redujo de un 87.6% a un 14.7%. El acoplamiento de todo el marco estadístico se redujo en un 72.9%.

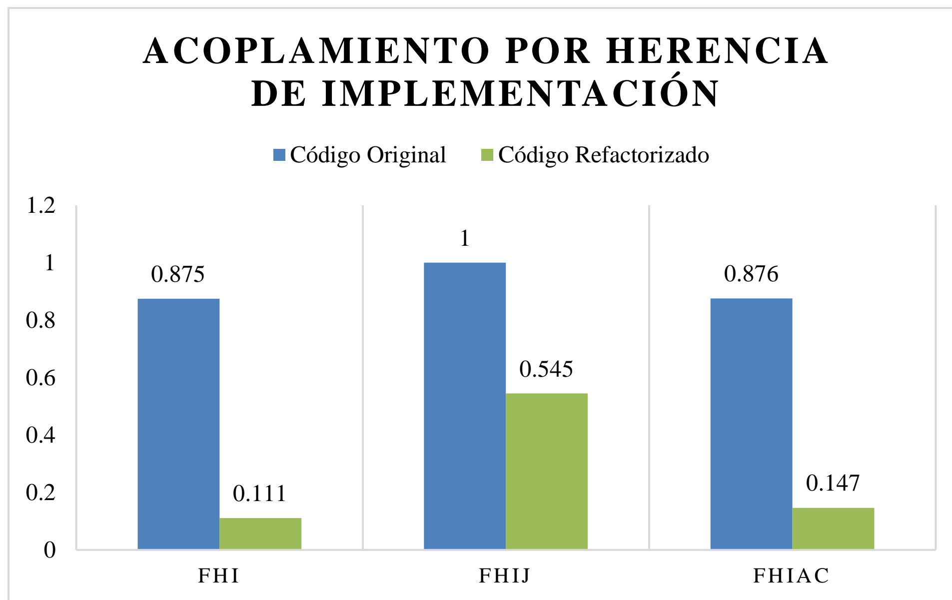


Gráfico 3.- Comparación de métricas de acoplamiento por herencia de implementación.

En el

Gráfico 4 se puede observar como los valores de las métricas FFC y FMFAC mejoraron, al acercarse a 1. El valor de flexibilidad de toda la arquitectura mejoró de un 8.8% a un 13.2%. El factor medio de flexibilidad por arquitectura de clase incrementó en un 4.4%. La baja mejora a nivel de arquitectura se explica de la misma manera que en el caso de prueba ISMRHI05-01.

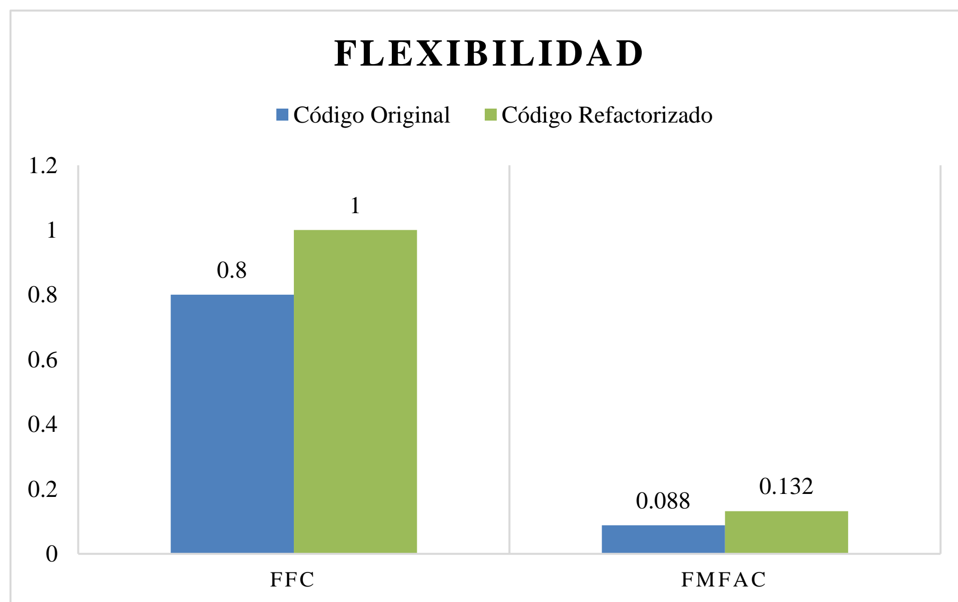


Gráfico 4.- Comparación de las métricas de flexibilidad.

6.5.4.- Caso de Prueba ISMRHI05-04

Artículo de Prueba: PSP Cenidet

El PSPCenidet es un sistema que mide los tiempos que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software.

Cálculo de la métrica (FHI) Factor de Herencia de Implementación

Clases del PSP Cenidet en donde se realizará el cálculo de la métrica:

1. <aEstadoA>
2. <DesarrolloAct_Cot>

La arquitectura de dichas clases se muestra en la Figura 41. En donde, los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

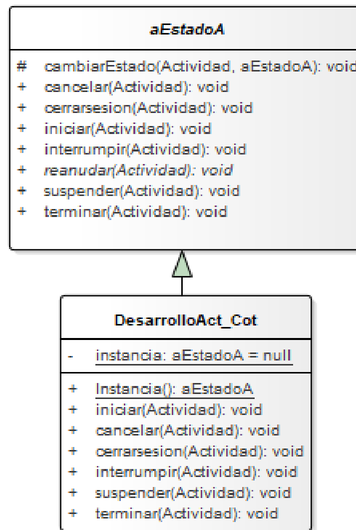


Figura 41.- Arquitectura de las clases para medir la métrica FHI.

El resultado manual del cálculo de la métrica FHI, correspondiente a la arquitectura de la Figura 41 es el siguiente:

$$FHI = \frac{\Sigma Mv + \Sigma Mnv}{\Sigma Tm} = \frac{7}{8} = 0.875 \qquad (6)$$

En donde:
Mv = *Métodos virtuales implementados en una clase base.*
Mnv = *Métodos no virtuales implementados en una clase base.*
Tm = *Métodos en una clase base.*

El valor óptimo de la métrica FHI es 0, que significa que no se está heredando la implementación de los métodos. En caso contrario, el valor menos deseado es el 1, el cual significa que todos sus métodos están heredando su implementación y como se puede observar, el resultado del cálculo manual fue 0.875, el cual es un valor cercano al menos deseado.
Se realizó el cálculo de la métrica FHI de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 42, el resultado fue 0.875.

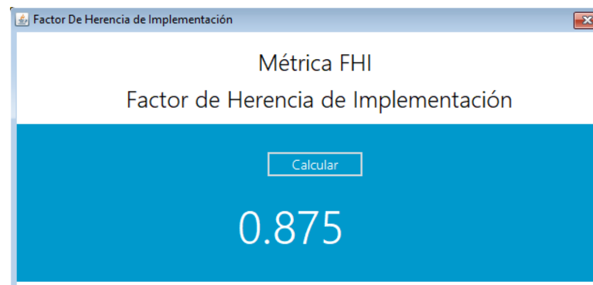


Figura 42.- Resultado de la métrica FHI utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 20, se compararon los resultados obtenidos del cálculo de la métrica FHI de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba para el cálculo de la métrica FHI.

Tabla 20.- Comparación de resultados de la métrica FHI.

Calculo Manual	Calculo Automático
0.875	0.875

Cálculo de la métrica (FHIJ) Factor de Herencia de Implementación por Jerarquía de Clase
Clases del PSP Cenidet en donde se realizará el cálculo de la métrica:

4. <EntidadPSP>
5. <Actividad>
6. <ActividadCotidiana>

La arquitectura de dichas clases se muestra en la Figura 43. En donde, los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

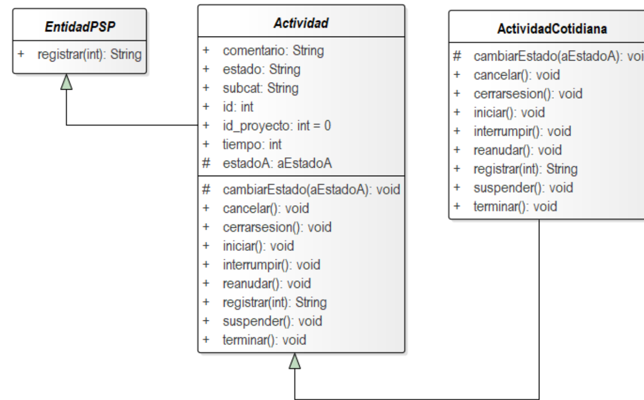


Figura 43.-Arquitectura de las clases para medir la métrica FHIJ.

El resultado manual del cálculo de la métrica FHIJ, correspondiente a la arquitectura de la Figura 43 es el siguiente:

$$FHIJ = \frac{\sum_{i}^{n} \left(\sum_{j=0}^{j=i-1} Mv \Big/ \sum_{j=0}^{j=i-1} M \right)}{n - 1} = \frac{0 + {}^{1}\!/_{1} + {}^{10}\!/_{10}}{3 - 1} = \frac{0 + 1 + 1}{2} = \frac{2}{2} = 1 \qquad (7)$$

En donde:
Mv = Cantidad de Métodos Virtuales de la Clases
M = Cantidad de Métodos Totales de las Clases
i, j =Representan el recorrido de la i, j-esima clase
n = Total de clases en la jerarquía

Asimismo, el valor óptimo de la métrica FHIJ es 0, que significa que no se está heredando la implementación de los métodos en la jerarquía. En caso contrario, el valor menos deseado es el 1, el cual significa que todos sus métodos están heredando su implementación y como se puede observar, el resultado del cálculo manual fue 1, que significa que el factor de herencia de implementación por jerarquía de clase se encuentra en su valor menos deseado.
Además, se realizó el cálculo de la métrica FHIJ de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 44, el resultado fue 1. Lo que quiere decir que todos sus métodos están heredando su comportamiento.

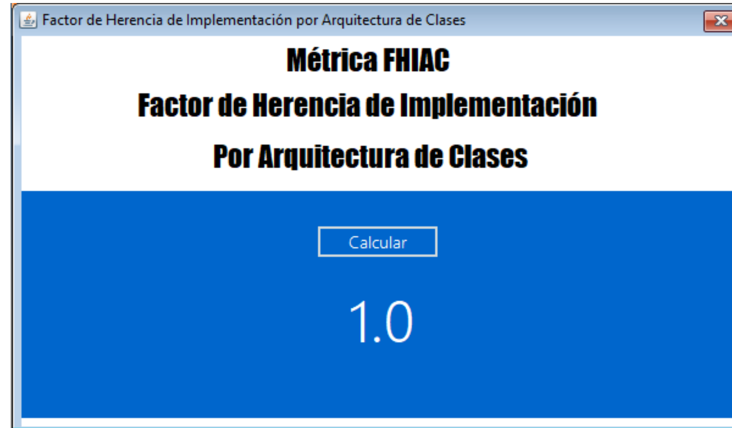


Figura 44.- Resultado de la métrica FHIJ utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 21, se compararon los resultados obtenidos del cálculo de la métrica FHI de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba para el cálculo de la métrica FHIJ.

Tabla 21.- Comparación de resultados de la métrica FHIJ.

Calculo Manual	Calculo Automático
1.0	1.0

Cálculo de la métrica (FHIAC) Factor de Herencia de Implementación de una Arquitectura de Clases

Para realizar el cálculo de la métrica FHIAC se utilizó la arquitectura de clases del PSP Cenidet, la cual se muestra en la

Figura 45 en donde las siglas Mv y Tm significan métodos virtuales y total de métodos respectivamente. El resultado manual de la métrica FHIAC se muestra en la Tabla 22, en donde se excluyen las clases Modelo y Usuario ya que son clases únicas, que no tienen ninguna relación de herencia. Así como también, no se tomó en cuenta la clase que funciona como cliente, la cual hace que el sistema se ejecute.

Tabla 22.- Cálculo manual de la métrica FHIAC en el PSP Cenidet.

Clase base	Mv	Tm	Clase derivada	Mv	Tm	Clase derivada	FHIJ	No.
EntidadPSP.java	1	1	Actividad.java	10	10	ActividadCotidiana.java	1	1
						ActividadProyecto.java	1	2
			Proyecto.java				1	3
			Interrupcion.java				1	4
aTiempo.java	0	1	cTiempo.java				0	5
aReporte.java	0	1	ReporteJasper.java				0	6
aEstadoA.java	7	8	PendienteAct_Cot.java				0.875	7

			DesarrolloAct_Proy.java				0.875	8
			DesarrolloAct_Cot.java				0.875	9
			PendienteAct_Proy.java				0.875	10
Conexion.java	1	3	ConexionMySql.java				0.333	11
			CbActSuspendidaCommand.java				1	12
			CbActSuspendidaPCommand.java				1	13
			CbCategoriaCatCommand.java				1	14
			CbCategoriaCommand.java				1	15
			CbEstadoCommand.java				1	16
			CbPerfilCatCommand.java				1	17
			CbPerfilCommand.java				1	18
			CbProyectoCommand.java				1	19
			CbReabrirActPCommand.java				1	20
			CbSubcategoriaCatCommand.java				1	21
			CbSubcategoriaCommand.java				1	22
			CbUsuarioCommand.java				1	23
Command.java	1	1	ExecuteQueryCommand.java				1	24
			ObtenerCountCommand.java				1	25
			ObtenerIdCommand.java				1	26
			ObtenerTextAreaCommand.java				1	27
			ObtenerTextFieldCommand.java				1	28
			ReporteCommand.java				1	29
			tblActividadCommand.java				1	30
			tblCategoriaCommand.java				1	31
			tblEstadoCommand.java				1	32
			tblPerfilCommand.java				1	33
			tblProyectoCommand.java				1	34
			tblSubCategoriaCommand.java				1	35
			tblUsuarioCommand.java				1	36
aEstadoR.java	3	3	EdoActivoR.java				1	37
			EdoInactivoR.java				1	38
BaseDatos.java	1	5	BaseDatosMySql.java				0.2	39
aReloj.java	0	5	cReloj.java				0.0	40
						Σ =	**35.0**	
						FHIAC =	**0.876**	

Para obtener el resultado de manera manual, se obtuvo el valor de la sumatoria de las métricas FHIJ de la arquitectura y se dividió entre 40 que es el número total de jerarquías. La fórmula siguiente es la que se utilizó:

$$\text{FHIAC} = \frac{\Sigma FHIJ}{\text{NOH}} = \frac{35}{40} = 0.876 \qquad (8)$$

En donde:

$FHIJ$ = *FHIJ de una jerarquía de clases.*
NOH = *Es el conteo de las jerarquías de clase.*

Asimismo, el valor óptimo de la métrica FHIAC es 0, que significa que no se está heredando la implementación de los métodos. En caso contrario, el valor menos deseado es el 1, el cual significa que todos sus métodos están heredando su implementación. Como se puede observar, el resultado del cálculo manual fue de 0.876, el cual significa que el factor de herencia de implementación de la arquitectura en la
Figura 45 está cerca del valor menos deseado.

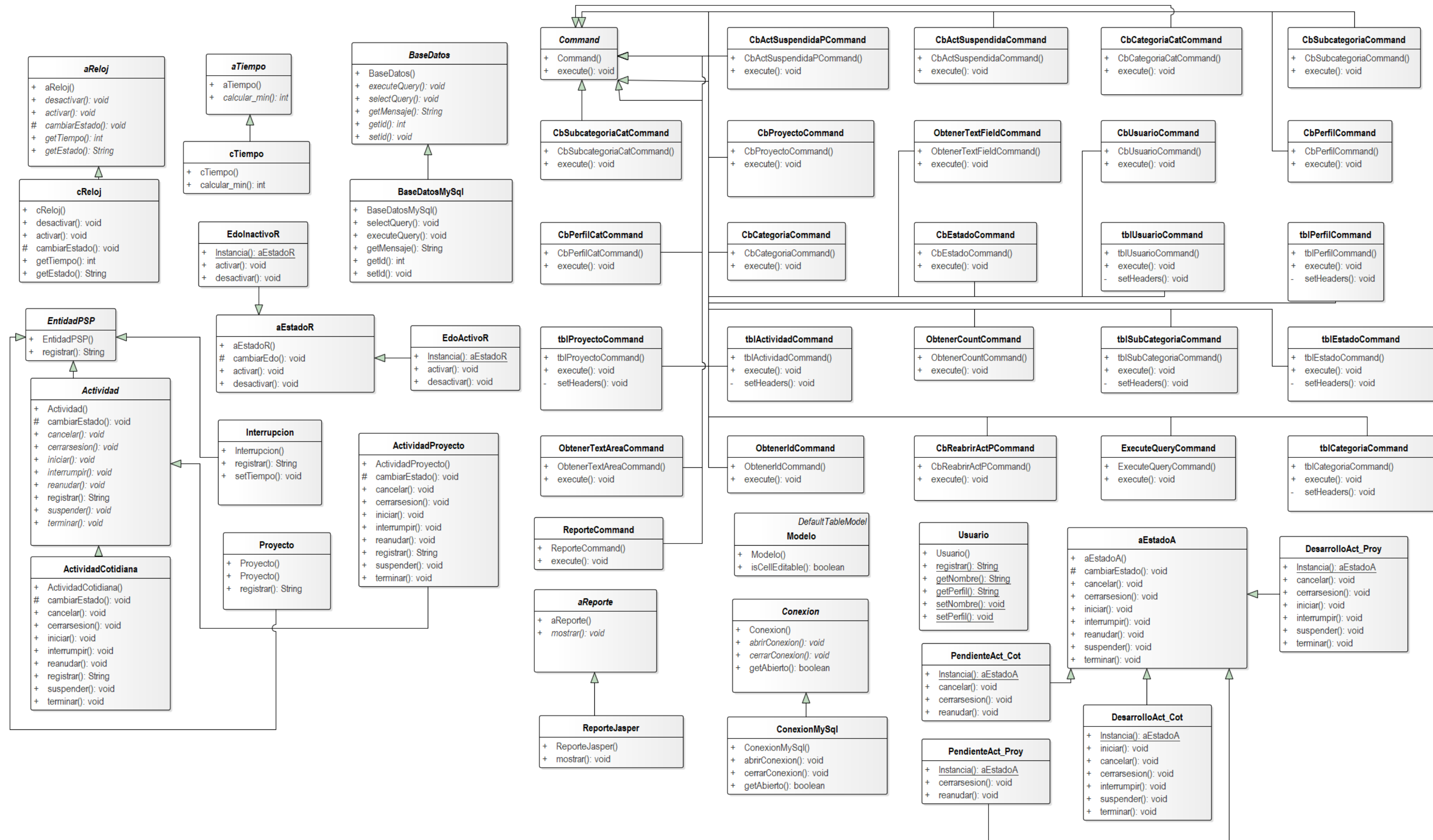


Figura 45.- Arquitectura de las clases para medir la métrica FHIAC.

Se realizó el cálculo de la métrica FHIAC de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 46, el resultado fue 0.876.

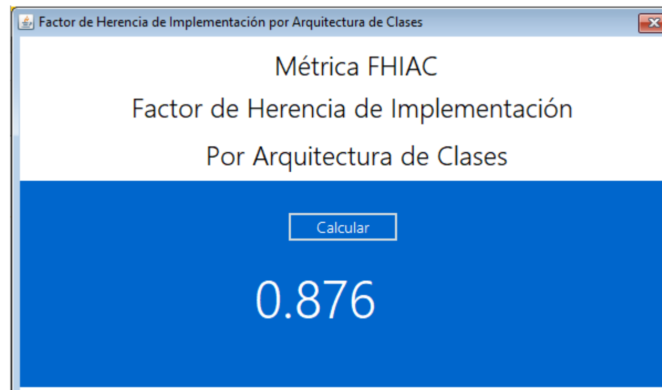


Figura 46.- Resultado de la métrica FHIAC utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 23, se compararon los resultados obtenidos del cálculo de la métrica FHIAC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba para el cálculo de la métrica FHIAC.

Tabla 23.- Comparación de resultados de la métrica FHIAC.

Calculo Manual	Calculo Automático
0.876	0.876

Cálculo de la métrica (FFC) Factor de Flexibilidad de Clases

Clases del PSP Cenidet en donde se realizará el cálculo de la métrica:

2. <BaseDatos>

En detalle se muestra la clase <BaseDatos> en la Figura 47. En donde, los nombres de los métodos que estén en cursivas, significan que son métodos abstractos, es decir, que no se toman en cuenta en la fórmula.

BaseDatos

+ dataSet: ResultSet
# mensaje: String
+ filas: int

+ executeQuery(Conexion, String): void
+ *selectQuery(Conexion, String): void*
+ *getMensaje(): String*
+ *getId(): int*
+ *setId(int): void*

Figura 47.- Clase BaseDatos.

El resultado manual del cálculo de la métrica FFC, correspondiente a la clase <BaseDatos> es el siguiente:

$$FFC = \frac{\Sigma NOP}{\Sigma \mathrm{T}m} = \frac{4}{5} = 0.8 \quad (9)$$

En donde:

NOP = *Métodos virtuales que exhiben un comportamiento polimórfico.*

Tm = *Métodos en una clase base.*

El valor óptimo de la métrica FFC es 1, que significa que todos los métodos de la clase son abstractos, por lo tanto, las funciones son polimórficas, lo que quiere decir que son flexibles a cambios de comportamiento. En caso contrario, el valor menos deseado es el 0, significa que ningún método es polimórfico, lo que quiere decir que no son flexibles a cambios de comportamiento y como se puede observar el resultado del cálculo manual de la métrica FFC fue de 0.8, el cual se acerca al valor deseado.

Se realizó el cálculo de la métrica FFC de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 48, el resultado fue 0.8.

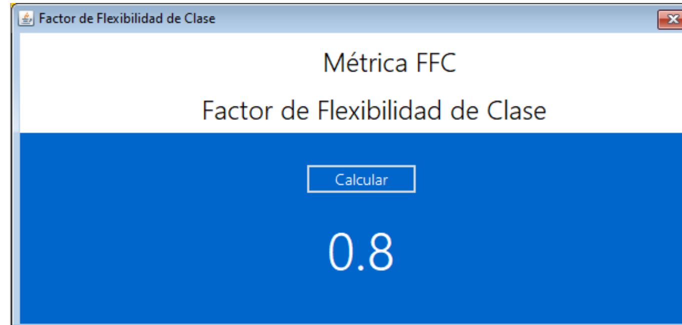


Figura 48.- Resultado de la métrica FFC utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 24, se compararon los resultados obtenidos del cálculo de la métrica FFC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que pasa la prueba.

Tabla 24.- Comparación de resultados de la métrica FFC.

Calculo Manual	Calculo Automático
0.8	0.8

Cálculo de la Métrica (FMFAC) Factor Medio de Flexibilidad de Arquitecturas de Clases

Para el cálculo de la métrica FMFAC se utilizó la arquitectura de clases del sistema PSP Cenidet, la cual se muestra en la

Figura 45. El cálculo manual de la métrica FMFAC se muestra en la Tabla 25, en ella se observa el factor de flexibilidad de clase (FFC) por cada clase base.

Tabla 25.- Cálculo manual de la métrica FMFAC en el PSP Cenidet.

Clase	FFC
Actividad	0
aEstadoA	0.125
aEstadoR	0
aReloj	1
aReporte	1
aTiempo	1
BaseDatos	0.8
Command	0
Conexion	0.66666667

EntidadPSP	0
Total=	4.592
FMFAC=	0.088

Para obtener el resultado de manera manual, se obtuvo el valor de la sumatoria de las métricas FFC y se dividió entre 52 que es el número total de clases en la arquitectura. La fórmula siguiente es la que se utilizó:

$$FMFAC = \frac{\Sigma FFC}{\Sigma Tc} = \frac{4.592}{52} = 0.088 \quad (10)$$

En donde:

FFC = FFC de una clase que hereda su comportamiento.

Tc = Número total de clases en la arquitectura.

Asimismo, el valor óptimo de la métrica FMFAC es 1, que significa que todos los métodos de la clase son abstractos, por lo tanto, las funciones son polimórficas, lo que quiere decir que son flexibles a cambios de comportamiento. En caso contrario, el valor menos deseado es el 0, significa que ningún método es polimórfico, lo que quiere decir que no son flexibles a cambios de comportamiento. Como se puede observar, el resultado del cálculo manual fue de 0.088, el cual significa que el factor medio de flexibilidad por arquitectura de clase se encuentra muy cerca del valor menos deseado.

Además, se realizó el cálculo de la métrica FMFAC de manera automática utilizando el sistema SR2-Refactoring y como se puede observar en la Figura 49, el resultado fue 0.088.

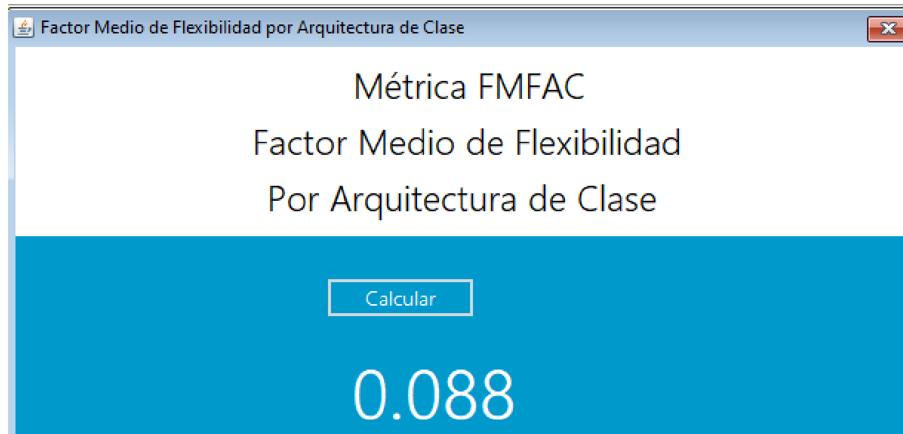


Figura 49.- Resultado de la métrica FMFAC utilizando el sistema SR2-Refactoring.

Como se puede observar en la Tabla 26, se compararon los resultados obtenidos del cálculo de la métrica FMFAC de manera manual y automática. Ambos fueron el mismo, lo que quiere decir que el sistema SR2 – Refactoring pasa la prueba para el cálculo de la métrica FMFAC.

Tabla 26.- Comparación de resultados de la métrica FMFAC.

Calculo Manual	Calculo Automático
0.088	0.088

En las Tablas Tabla 27 y Tabla 28, se muestra un resumen de los resultados de cada uno de los casos de prueba realizados. En la Tabla 27 se muestran los resultados de los casos de prueba que evalúan el método de reducción de herencia de implementación y en la Tabla 28 los que evalúan el cálculo automático de las métricas: FHI (Factor de Herencia de Implementación), FHIJ (Factor de Herencia de Implementación de una Jerarquía de Clases), FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases), FFC (Factor de Flexibilidad de Clase) y FMFAC (Factor Medio de Flexibilidad de Arquitecturas de Clases).

Tabla 27.- Resultados de las pruebas del método de refactorización.

	Prueba	***Métricas***	***Original***	***Refactorizado***	***Valor Deseado***
Marco Estadístico	ISMRHI05-01	FHI	1	0	0
		FHIJ	0.25	0	
		FHIAC	0.854	0.762	
		FFC	0	1	1
		FMFAC	0.196	0.239	
PSP CENIDET	ISMRHI05-03	FHI	0.875	0.111	0
		FHIJ	1	0.545	
		FHIAC	0.876	0.147	
		FFC	0.8	1	1
		FMFAC	0.088	0.132	

Como se puede observar en la Tabla 27, los atributos de flexibilidad y acoplamiento por herencia de implementación en ambos sistemas mejoraron, después de haber sido refactorizados por el método de reducción de herencia de implementación. Por dicha razón las pruebas ISMRHI05-01 y ISMRHI05-03 se aprobaron, ya que, además de reducir el acoplamiento y aumentar la flexibilidad, los sistemas siguen funcionando correctamente, es decir, solo se mejoró su arquitectura sin modificar su comportamiento externo.

Tabla 28.- Resultados de las pruebas del cálculo de métricas.

	Prueba	Métricas	Cálculo Manual	Cálculo Automático
Marco Estadístico	ISMRHI05-02	FHI	1	1
		FHIJ	0.25	0.25
		FHIAC	0.854	0.854
		FFC	0	0
		FMFAC	0.196	0.196
PSP CENIDET	ISMRHI05-04	FHI	0.875	0.875
		FHIJ	1	1
		FHIAC	0.876	0.876

		FFC	0.8	0.8
		FMFAC	0.088	0.088

En la Tabla 28 se muestran los resultados del cálculo manual y automático de las métricas: FHI (Factor de Herencia de Implementación), FHIJ (Factor de Herencia de Implementación de una Jerarquía de Clases), FHIAC (Factor de Herencia de Implementación de una Arquitectura de Clases), FFC (Factor de Flexibilidad de Clase) y FMFAC (Factor Medio de Flexibilidad de Arquitecturas de Clases), y como se puede observar en cada uno de los casos de prueba, los resultados fueron el mismo. Por lo cual, los resultados de los casos de prueba ISMRHI05-02 y ISMRHI05-04 fueron aprobados, ya que se comprobó que la manera en que el sistema realiza el cálculo automático de las métricas, es igual a lo estipulado en las fórmulas correspondientes a cada métrica.

Capítulo 7.- CONCLUSIONES Y TRABAJO A FUTURO

En este capítulo se muestran las conclusiones y aportaciones derivadas al termino de todo el trabajo de investigación. Asimismo, se enuncian los trabajos futuros que se pueden realizar a partir de esta investigación.

7.1.- Conclusiones

- Las llamadas directas (call forward) hacen que los sistemas orientados a objetos sean rígidos e inmóviles, ya que el efecto es que las clases derivadas dependen de sus clases base para funcionar. Esto no es deseable porque si se quiere extender el comportamiento del sistema se tendrá que modificar el código ya existente, violando el principio de "Abierto-Cerrado".

- Las métricas a nivel de clase FHI y FFC se complementan, ya que si suman sus valores de la misma clase el resultado será 1.

$$FHI(X) + FFC(X) = 1$$

- El uso de los métodos del SR2-Refactoring deben ser aplicados en un orden secuencial razonable que se muestra en la Figura 50. Es decir, la secuencia de aplicación de los métodos de refactorización no puede ser aplicados aleatoriamente, sino que tienen un orden razonable.

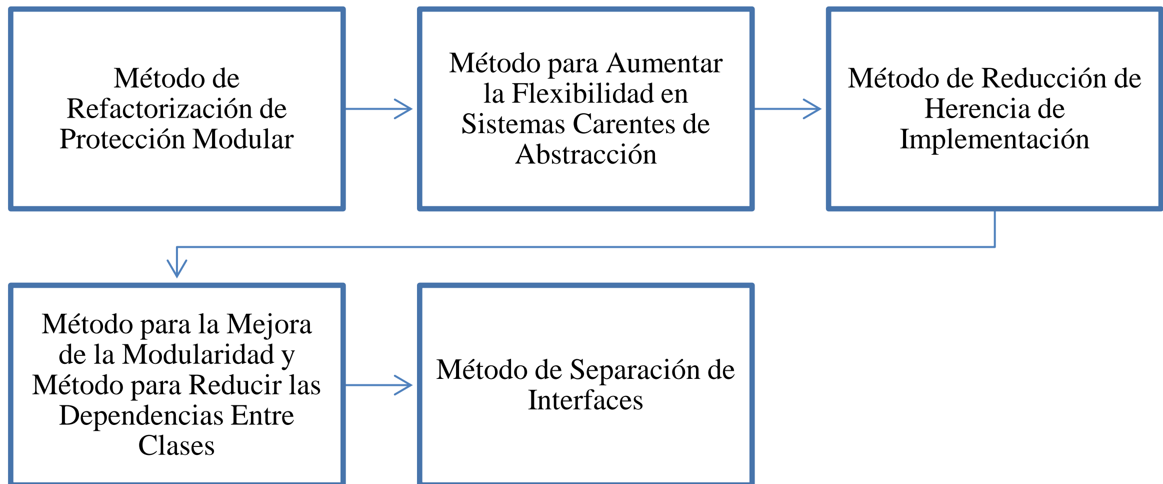


Figura 50.- Secuencia de Aplicación de los Métodos de Refactorización del SR2-Refactoring.

Al comprobar con los casos de prueba que el método de reducción de herencia de implementación funcionó correctamente, reduciendo el acoplamiento por herencia de implementación y aumentando la flexibilidad de un MAOO, se cumple con el objetivo general de esta tesis, el cual es "Mejorar el diseño de arquitecturas del software legado orientado a objetos en su flexibilidad y autonomía en relación al acoplamiento, encaminadas a lograr mayor calidad para facilitar su reuso y su mantenimiento". El método de reducción de herencia de implementación se desarrolló para ser aplicado en MAOO's programados en lenguaje Java y se implementó en el sistema SR2-Refactoring, extendiendo su funcionalidad, por lo cual, se están cumpliendo los objetivos específicos de esta tesis, los cuales son: "mejorar el diseño de arquitecturas de software legado escrito en lenguaje java" y "extender la funcionalidad del SR2-Refactoring, para dar soporte a sus métodos de re-factorización de alto impacto en código escrito en lenguaje Java".

Además de lo que se mencionó anteriormente, el aprendizaje que se obtuvo al terminar de realizar este proyecto, fue utilizar los diversos patrones de diseño de software al momento de desarrollar cualquier sistema. En (Gamma, Helm, Johnson, & Vlissides, 2002) se mencionan algunos de los patrones de diseño de software que existen, cada uno describe un problema de diseño especifico y su manera de resolverlo. En este proyecto se enfocó en el patrón de diseño del método de la plantilla, ya que, junto con él, se utiliza la herencia de interfaz para hacer que los sistemas de software sean flexibles. Además, al hacer uso del patrón de diseño del método de la plantilla se cumplen con los principios de diseño de Abierto-Cerrado y de Inversión de Dependencias. El que en este proyecto se haya enfocado solo en un patrón de diseño no quiere decir que los demás sean menos importantes, sino que es el adecuado para resolver el problema planteado. Es muy importante saber cómo se utilizan los diversos patrones de diseño, para que no se pierda la oportunidad de aplicarlos en el desarrollo de cualquier sistema de software.

7.2.- Aportaciones de la tesis

7.2.1.- Método de refactorización denominado Reducción de Herencia de Implementación

- El método tiene el objetivo de reducir el acoplamiento por herencia de implementación. Se aplica el principio de inversión de dependencias para cambiar las llamadas directas de código de cliente a código de librerías (call forward) a llamadas de código de librerías a código de cliente (call back).

- Mediante la refactorización de las cláusulas *"super"*, los sistemas rígidos e inmóviles por depender de la implementación de los métodos de las clases base, ahora serán sistemas flexibles y móviles, haciendo que su mantenimiento sea más fácil y se habilita la reusabilidad.

- Una vez que el MAOO se encuentre refactorizado, se respetará el principio de "Abierto-Cerrado" así como el principio de inversión de dependencias. Lo que provoca que la extensibilidad aumente al poder agregar nuevas funciones sin dificultad.

7.2.2.- Conjunto de métricas para medir el factor de acoplamiento por herencia de implementación y de flexibilidad

- Debido a que no existe documentación en cuanto a cómo medir y evaluar el acoplamiento por herencia de implementación y la flexibilidad tomando en cuenta métodos abstractos, se definieron las siguientes métricas:

1. FHI (Factor de Herencia de Implementación)
2. FHIJ (Factor de Herencia de Implementación por Jerarquía de Clase)
3. FHIAC (Factor de Herencia de Implementación de Arquitecturas de Clases)
4. FFC (Factor de Flexibilidad de Clase)
5. FMFAC (Factor Medio de Flexibilidad por Arquitectura de Clase)

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

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

- Además, se integró al SR2-Refactoring el cálculo automático del conjunto de métricas FHI, FHIJ, FHIAC, FFC y FMFAC.

7.3.- Trabajo a Futuro

7.3.1.- Renovar la fórmula para calcular la métrica de Flexibilidad Media por Arquitectura de Clases

Actualmente la fórmula consiste en dividir la sumatoria de la métrica FFC entre el número total de clases. Con el propósito de hacer más evidente la mejora en cuanto la flexibilidad y el acoplamiento por herencia de implementación, se podría crear una nueva versión de la fórmula dividiendo entre el número total de clases que están heredando su implementación.

$$FMFAC = \frac{\Sigma FFC}{\Sigma Tch}$$

En donde:
FFC= FFC de una clase que hereda su comportamiento.
Tch = Número total de clases en la arquitectura que heredan su implementación.

7.3.2.- Evaluar la conveniencia de la herencia de implementación lo cual produce acoplamiento contra la herencia de interfaz lo cual podría violar el principio de “No Repetición.”

Determinar el valor óptimo que equilibre la herencia de implementación vs la no-repetición funcional.

Anexo A.- Métricas FHI, FHIJ, FHIAC, FFC y FMFAC como escalas ordinales

Factor de Herencia de Implementación (FHI) como escala ordinal
Se tiene el siguiente sistema relacional empírico:

1. *P* es el conjunto de una clase base y una clase derivada.
2. •≥ es una relación empírica entre clases que describe que una clase tiene mayor o igual factor de herencia de implementación que otra.
3. ℜ denota el conjunto de los números reales.
4. ≥ *"mayor o igual que"* es una relación binaria entre números.

Entonces ((P, •≥),(ℜ,≥),FHI) es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

1. La relación binaria •≥, es de *orden débil*.
2. Clase1 •≥ Clase2 ⇔ FHI(Clase1) ≥ FHI(Clase2)

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (1) para realizar el cálculo del FHI de las clases uno, dos y tres, ubicadas en la Figura 51. En donde los métodos que tienen "= 0", significa que no son virtuales, y los que no lo tienen,

quiere decir que tienen una implementación en su cuerpo. Obteniéndose los siguientes resultados:

- FHI (Clase1) = $\frac{\Sigma Mv+\Sigma Mnv}{\Sigma Tm}$ = 2/3 = 0.666...
- FHI (Clase2) = $\frac{\Sigma Mv+\Sigma Mnv}{\Sigma Tm}$ = 2/4 = 0.5
- FHI (Clase3) = $\frac{\Sigma Mv+\Sigma Mnv}{\Sigma Tm}$ = 2/5 = 0.4

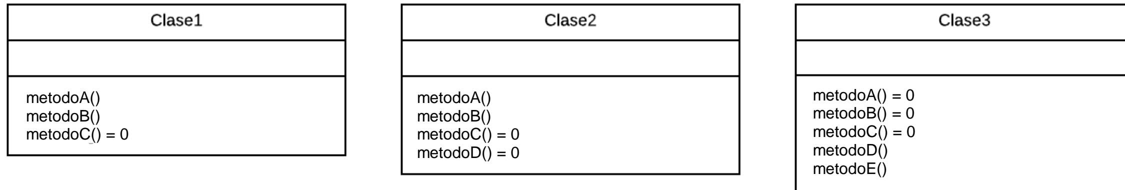


Figura 51.- Arquitectura de clases de un sistema.

Para comprobar que la relación binaria es de orden débil, tiene que cumplir con las propiedades de transitividad y completitud.

Transitividad:

Clase1 •≥ Clase2, Clase2 •≥ Clase3 → Clase1 •≥ Clase3

↔

FHI (Clase1) •≥ FHI (Clase2) & FHI (Clase2) •≥ FHI (Clase3) →
FHI (Clase1) •≥ FHI (Clase3)

Reflejado en números, se tiene que:

$$0.666\ldots \geq 0.5 \ \& \ 0.5 \geq 0.4 \rightarrow 0.666\ldots \geq 0.4$$

Como se puede observar, la clase1 tiene un factor de herencia de implementación mayor al de la clase2. Éste a su vez tiene un factor de herencia de implementación mayor al de la clase3. Concluyendo que la relación binaria •≥, cumple con la propiedad de transitividad.

Relación Total:
Si se tienen las clases uno y dos, se debe poder decir que la clase 1 *"tiene mayor o igual factor de herencia de implementación que"* la clase 2, o viceversa. Es decir:

Clase1 •≥ Clase2 o Clase2 •≥ Clase1

Calculando el FHI en las clases uno y dos de la Figura 51, siempre fue posible determinar cuando existía mayor o igual factor de herencia de implementación.

Homomorfismo:
Comprobando la segunda condición quedaría de la siguiente manera:

La clase 1 *"tiene mayor o igual factor de herencia de implementación que"* la clase 2

⇔

0.666… ≥ 0.5

Conclusión

La relación binaria *"tiene mayor o igual factor de herencia de implementación que"* de la métrica FHI, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Suze, 1992).

Factor de Herencia de Implementación por Jerarquía de Clases (FHIJ) como Escala Ordinal

Se tiene el siguiente sistema relacional empírico:

1. *P* es el conjunto de jerarquías de clases.
2. •≥ es una relación empírica entre clases que describe que una jerarquía de clase tiene mayor o igual factor de herencia de implementación que otra.
3. ℜ denota el conjunto de los números reales.
4. ≥ *"mayor o igual que"* es una relación binaria entre números.

Entonces ((P,•≥),(ℜ,≥), FHIJ) es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

1. La relación binaria •≥, es de *orden débil*.
2. JerarquiaClase1 •≥ JerarquiaClase2 ⇔ FHIJ(JerarquiaClase1) ≥ FHIJ(JerarquiaClase2).

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (2) para realizar el cálculo de la métrica FHIJ de las clases uno, dos y tres, ubicadas en la Figura 52. En donde los métodos que tienen "= 0", significa que no son virtuales, y los que no lo tienen, quiere decir que tienen una implementación en su cuerpo. Obteniéndose los siguientes resultados:

1. $\text{FHIJ(JerarquiaClase1)} = \dfrac{\sum_{i}^{n}\left(\sum_{j=0}^{j=i-1} Mv \middle/ \sum_{j=0}^{j=i-1} M\right)}{n-1} = \dfrac{(0+2/3+3/5)}{3-1} = \dfrac{1.266\ldots}{2} = 0.633\ldots$

Como se puede observar en la fórmula, primero se realizó las operaciones del numerador. A continuación, se muestra cómo se resolvieron dichas operaciones.

Cuando i = 0, FHIJ obtiene el valor de 0.

$i = 0 \quad \rightarrow \quad FHIJ = 0$

i = 1 → Se suma el total de la sumatoria anterior, más el cociente de dividir la cantidad de métodos virtuales heredados, entre la sumatoria de métodos que se tengan en las clases anteriores a la clase i, que en este caso la clase i corresponde a la clase <Clase2>. La clase <Clase1> que es la única clase anterior a la clase <Clase2>, cuenta con 2 métodos virtuales y 3 métodos en total. La sumatoria se expresa de la siguiente manera:

$$i = 1 \quad \rightarrow \quad 0 + \left(\sum_{j=0}^{j=0} Mv \Big/ \sum_{j=0}^{j=0} M \right) = 0 + \left(\frac{2}{3}\right) = 0.666$$

i = 2 → Se suma el total de la sumatoria anterior, más el cociente de dividir la cantidad de métodos virtuales heredados, entre la sumatoria de métodos que se tengan en las clases anteriores a la clase i, que en este caso la clase i corresponde a la clase <Clase3>. Las clases <Clase1> y <Clase2> son las clases anteriores a la clase <Clase3>, y juntas cuentan con un total de 3 métodos virtuales y 5 métodos en total. La sumatoria se expresa de la siguiente manera:

$$i = 2 \quad \rightarrow \quad 0.666 + \left(\sum_{j=0}^{j=1} Mv \Big/ \sum_{j=0}^{j=1} M \right) = 0.666 + \left(\frac{3}{5}\right) = 0.666 + 0.6 = 1.266$$

Una vez que se tiene el valor del numerador se prosigue a realizar la división entre el denominador.

$$FHIJ = \frac{\sum_{i=0}^{n} \left(\sum_{j=0}^{j=i-1} Mv \Big/ \sum_{j=0}^{j=i-1} M \right)}{3-1} = \frac{1.266 \ldots}{2} = 0.633 \ldots$$

Teniendo como resultado de la métrica 0.633, los demás ejemplos se realizaron utilizando la misma metodología.

2. FHIJ(JerarquiaClase2) = $\frac{\sum_{i}^{n} \left(\sum_{j=0}^{j=i-1} Mv \Big/ \sum_{j=0}^{j=i-1} M \right)}{n-1} = \frac{(0 + 2/4 + 4/7)}{3-1} = \frac{0.5 + 0.571\ldots}{2} = 0.535\ldots$

3. FHIJ(JerarquiaClase3) = $\frac{\sum_{i}^{n} \left(\sum_{j=0}^{j=i-1} Mv \Big/ \sum_{j=0}^{j=i-1} M \right)}{n-1} = \frac{(0 + 2/5 + 4/9)}{3-1} = \frac{0.844\ldots}{2} = 0.422\ldots$

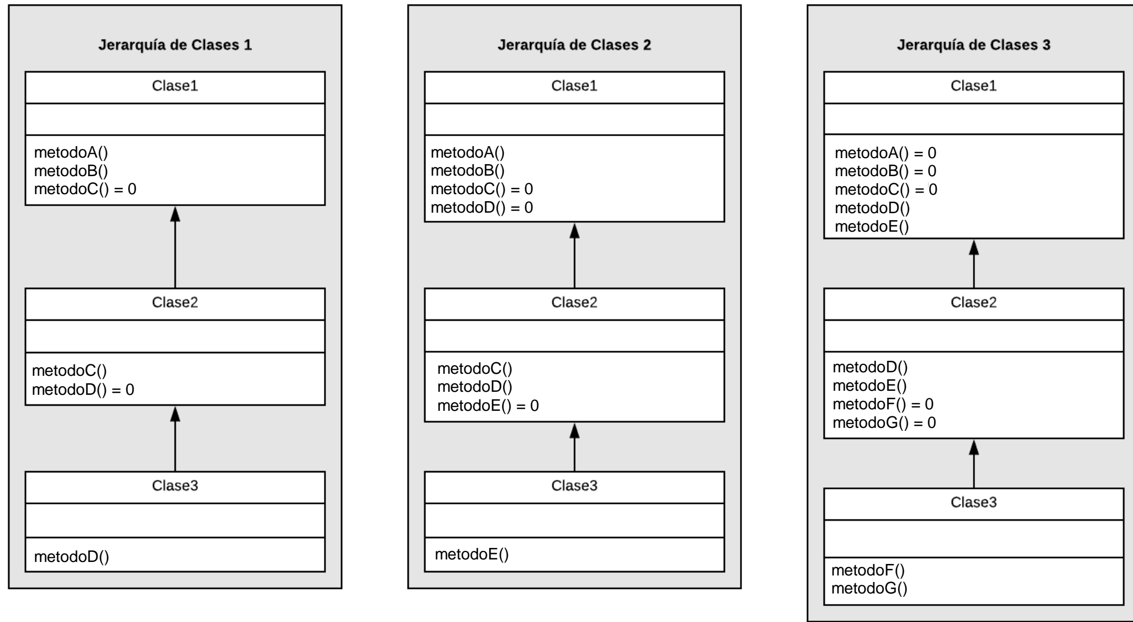


Figura 52.- Conjunto de jerarquías de clases.

Para comprobar que la relación binaria es de orden débil, tiene que cumplir con las propiedades de transitividad y completitud.
Transitividad:

JerarquiaClase1 •≥ JerarquiaClase2, JerarquiaClase2 •≥ JerarquiaClase3 → JerarquiaClase1 •≥ JerarquiaClase3

↔

FHIJ(JerarquiaClase1) •≥ FHIJ (JerarquiaClase2) & FHIJ (JerarquiaClase2) •≥ FHIJ (JerarquiaClase3) → FHIJ (JerarquiaClase1) •≥ FHIJ (JerarquiaClase3)

Reflejado en números, se tiene que:

$$0.633\ldots \geq 0.535\ldots \ \& \ 0.535\ldots \geq 0.422 \rightarrow 0.633\ldots \geq 0.422$$

Como se puede observar, la jerarquía de clase1 tiene un factor de herencia de implementación mayor o igual al de la jerarquía de clase 2. Éste a su vez tiene un factor de herencia de implementación mayor al de la jerarquía de clase 3. Concluyendo que la relación binaria •≥ cumple con la propiedad de transitividad.
Relación Total:

Si se tienen las jerarquías de clases uno y dos (Figura 52), se debe poder decir que la jerarquía de clase 1 *"tiene mayor o igual factor de herencia de implementación que"* la jerarquía de clase 2, o viceversa. Es decir:

JerarquiaClase1 •≥ JerarquiaClase2 o JerarquiaClase2 •≥ JerarquiaClase1

Calculando el FHIJ en las jerarquías de clases uno y dos de la Figura 52, siempre fue posible determinar cuando existía mayor o igual factor de herencia de implementación.

Homomorfismo:
Comprobando la segunda condición quedaría de la siguiente manera:

La jerarquía de clase 1 *"tiene mayor o igual factor de herencia de implementación que"* la jerarquía de clase 2

$$\Leftrightarrow$$

$$0.633\ldots \geq 0.535\ldots$$

Conclusión:

La relación binaria *"tiene mayor o igual factor de herencia de implementación que"* de la métrica FHIJ, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Suze, 1992).

Factor de Herencia de Implementación de una Arquitectura de Clases (FHIAC) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1. *P* es el conjunto de una clase base y una clase derivada.
2. •≥ es una relación empírica entre clases que describe que una arquitectura de clases tiene mayor o igual factor de herencia de implementación que otra.
3. $\mathfrak{R}$ denota el conjunto de los números reales.
4. ≥ *"mayor o igual que"* es una relación binaria entre números.

Entonces ((**P**, •≥), (**$\mathfrak{R}$**,≥), FHIAC) es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

1. La relación binaria •≥, de *orden débil*.
2. ArquitecturaClase1 •≥ ArquitecturaClase2 ⇔ FHIAC(ArquitecturaClase1) ≥ FHIAC(ArquitecturaClase2)

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (3) para realizar el cálculo del FHI de las clases uno, dos y tres, ubicadas en la Figura 53. En donde los métodos que tienen "= 0", significa que son abstractos, y los que no lo tienen, quiere decir que tienen una implementación en su cuerpo. Obteniéndose los siguientes resultados:

- FHIAC (ArquitecturaClases1) = $\frac{\Sigma FHIJ}{NOH} = \frac{(0.583\ldots+0.5)}{2} = \frac{1.083\ldots}{2} = 0.5416\ldots$
- FHIAC (ArquitecturaClases2) = $\frac{\Sigma FHIJ}{NOH} = \frac{(0.4+0.575)}{2} = \frac{0.975}{2} = 0.4875$
- FHIAC (ArquitecturaClases3) = $\frac{\Sigma FHIJ}{NOH} = \frac{(0.416\ldots+0.5)}{2} = \frac{0.916\ldots}{2} = 0.4583\ldots$

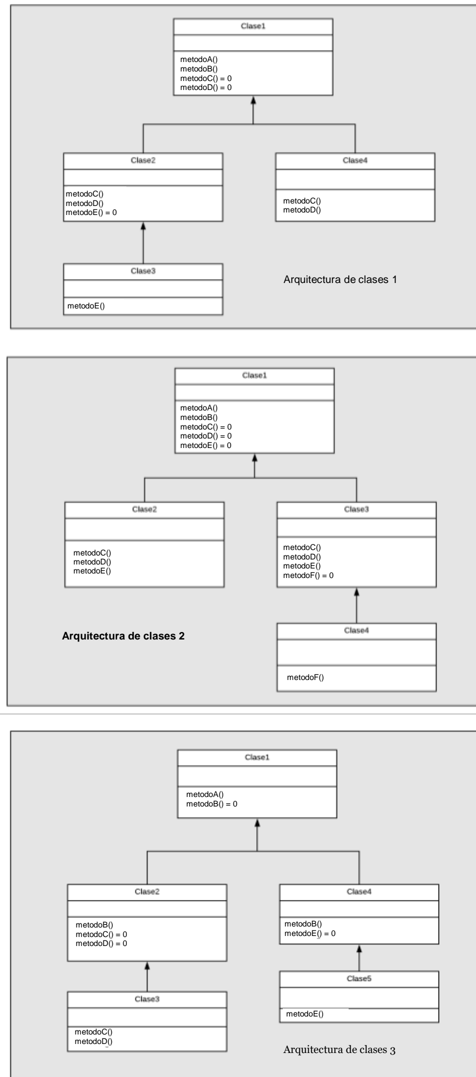


Figura 53.- Arquitecturas de clases.

Para comprobar que la relación binaria es de orden débil, tiene que cumplir con las propiedades de transitividad y completitud.

Demostración empírica de la propiedad de transitividad:

ArquitecturaClases1 •≥ ArquitecturaClases2,
ArquitecturaClases2 •≥ ArquitecturaClases3 →
ArquitecturaClases1 •≥ ArquitecturaClases3
↔
FHIAC (ArquitecturaClases1) •≥ FHIAC (ArquitecturaClases2) **&**
FHIAC(ArquitecturaClases2) •≥ FHIAC (ArquitecturaClases3) →
FHIAC (ArquitecturaClases1) •≥ FHIAC (ArquitecturaClases3)

Demostración formal de la propiedad de transitividad:

Reflejado en números, se tiene que:

$$0.5416\ldots \geq 0.4875\ldots \ \& \ 0.4875\ldots \geq 0.4583\ldots \rightarrow 0.5416\ldots \geq 0.4583\ldots$$

Como se puede observar, la arquitectura de clases 1 tiene un factor de herencia de implementación mayor al de la arquitectura 2. Éste a su vez tiene un factor de herencia de implementación mayor al de la arquitectura 3. Concluyendo que la relación binaria •≥, cumple con la propiedad de transitividad.

Demostración empírica de la propiedad de completitud:

Si se tienen las arquitecturas de clases uno y dos, se debe poder decir que la arquitectura de clase *"tiene mayor o igual factor de herencia de implementación que"* la arquitectura de clase 2, o viceversa. Es decir:

ArquitecturaClases1 •≥ ArquitecturaClases2 o ArquitecturaClases2 •≥ ArquitecturaClases1

Demostración formal de la propiedad de completitud:

Calculando el FHIAC en las arquitecturas de clases uno y dos de la Figura 53, siempre fue posible determinar cuando existía mayor o igual factor de herencia de implementación.

Demostración empírica de FHIAC es un homomorfismo:
Comprobando la segunda condición quedaría de la siguiente manera:

La arquitectura de clases 1 *"tiene mayor o igual factor de herencia de implementación que"* la arquitectura de clases 2

Formalmente:

$$0.5416\ldots \geq 0.4875\ldots$$

Conclusión:

La relación binaria "tiene mayor o igual factor de herencia de implementación que" de la métrica FHIAC, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Suze, 1992).

Factor de Flexibilidad de Clases (FFC) como escala ordinal
Se tiene el siguiente sistema relacional empírico:

1. *P* es el conjunto de clases de una arquitectura.
2. •≥ es una relación empírica binaria entre clases, que describe que una clase tiene mayor o igual factor de flexibilidad que otra.

3. $\Re$ denota el conjunto de los números reales.
4. ≥ *"mayor o igual que"* es una relación binaria entre números.

Entonces ((**P**, •≥), (**ℜ**,≥), FFC) es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

1. La relación binaria •≥, es de *orden débil*.
2. Clase1 •≥ Clase2 ⇔ FFC(Clase1) ≥ FFC(Clase2)

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (4) para realizar el cálculo del FFC de las clases uno, dos y tres, ubicadas en la Figura 54. En donde los métodos que tienen "= 0", significa que no son virtuales, y los que no lo tienen, quiere decir que tienen una implementación en su cuerpo. Obteniéndose los siguientes resultados:

- FFC(Clase1) = 3/5 = 0.6
- FFC(Clase2) = 2/4 = 0.5
- FFC(Clase3) = 1/3 = 0.333…

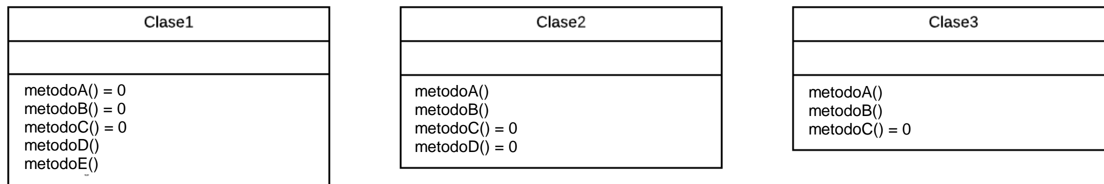


Figura 54.- Arquitectura de clases de un sistema.

Para comprobar que la relación binaria es de orden débil, tiene que cumplir con las propiedades de transitividad y completitud.

Transitividad:

Clase1 •≥ Clase2, Clase2 •≥ Clase3 → Clase1 •≥ Clase3

↔

FFC(Clase1) •≥ FFC(Clase2) & FFC(Clase2) •≥ FFC(Clase3) →

FFC(Clase1) •≥ FFC(Clase3)

Reflejado en números, se tiene que:

$$0.6 \geq 0.5 \ \& \ 0.5 \geq 0.333 \rightarrow 0.6 \geq 0.333$$

Como se puede observar, la clase 1 tiene un factor de flexibilidad de clase mayor al de la clase 2. Éste a su vez tiene un factor de flexibilidad mayor al de la clase 3, debido a la cantidad de métodos virtuales (con comportamiento polimórfico) que tienen. Concluyendo que la relación binaria •≥, cumple con la propiedad de transitividad.

Relación Total:

Si se tienen las clases uno y dos, se debe poder decir que la clase 1 “tiene mayor o igual factor de flexibilidad de clase que” la clase 2, o viceversa. Es decir:

Clase1 •≥ Clase2 o Clase2 •≥ Clase1

Calculando el FFC en las clases uno y dos de la Figura 54 siempre fue posible determinar cuando existía mayor o igual factor de flexibilidad.

Homomorfismo:

Comprobando la segunda condición quedaría de la siguiente manera:

La clase 1 *“tiene mayor o igual factor de flexibilidad que”* la clase 2
⇔
$0.6 \geq 0.5$

Demostración empírica de la propiedad de transitividad:

Clase1 •≥ Clase2, Clase2 •≥ Clase3 → Clase1 •≥ Clase3
↔

FFC(Clase1) •≥ FFC(Clase2) & FFC(Clase2) •≥ FFC(Clase3) →

FFC(Clase1) •≥ FFC(Clase3)

Demostración formal de la propiedad de transitividad:

Reflejado en números, se tiene que:

$0.6 \geq 0.5$ & $0.5 \geq 0.333 \rightarrow 0.6 \geq 0.333$

Como se puede observar, la clase 1 tiene un factor de flexibilidad de clase mayor al de la clase 2. Éste a su vez tiene un factor de flexibilidad mayor al de la clase 3, debido a la cantidad de métodos virtuales con comportamiento polimórfico que tienen. Concluyendo que la relación binaria •≥, cumple con la propiedad de transitividad.

Demostración empírica de la propiedad de completitud:

Si se tienen las clases uno y dos, se debe poder decir que la clase 1 *“tiene mayor o igual factor de flexibilidad de clase que”* la clase 2, o viceversa. Es decir:

Clase1 •≥ Clase2 o Clase2 •≥ Clase1

Demostración formal de la propiedad de completitud:

Calculando el FFC en las clases uno y dos de la Figura 54 siempre fue posible determinar cuando existía mayor o igual factor de flexibilidad.

Demostración empírica de FFC que es un homomorfismo:

Comprobando la segunda condición quedaría de la siguiente manera:

La clase 1 *"tiene mayor o igual factor de flexibilidad que"* la clase 2

Formalmente:

$$0.6 \geq 0.5$$

Conclusión:

La relación binaria *"tiene mayor o igual factor de flexibilidad que"* de la métrica FFC, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Suze, 1992).

Factor Medio de Flexibilidad de Arquitecturas de Clases (FMFAC) como escala ordinal
Se tiene el siguiente sistema relacional empírico:

1. P es el conjunto de una clase base y una clase derivada.
2. •≥ es una relación empírica entre clases que describe que una arquitectura de clases tiene mayor o igual factor medio de flexibilidad de la arquitectura de clases que otra.
3. ℜ denota el conjunto de los números reales.
4. ≥ *"mayor o igual que"* es una relación binaria entre números.

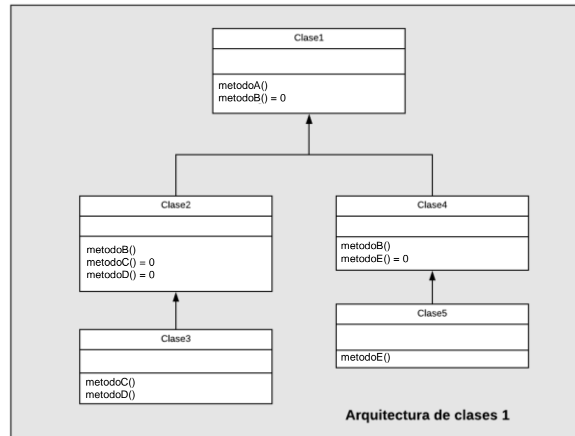
Entonces ((**P**, •≥), (ℜ,≥), FMFAC) es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

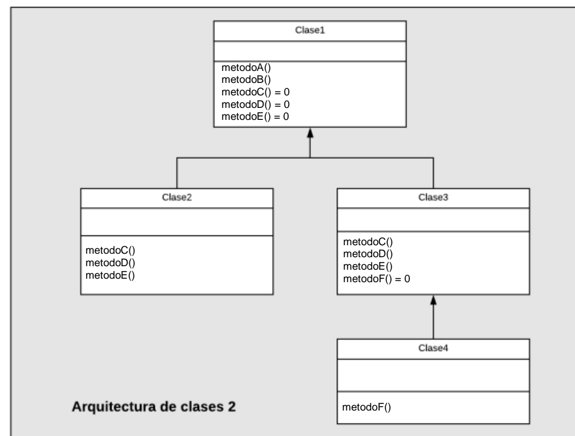
1. La relación binaria •≥, es de orden débil.
2. ArquitecturaClases1 •≥ ArquitecturaClases2 ⇔ FMFAC(ArquitecturaClases1) ≥ FMFAC(ArquitecturaClases2)

Como parte del proceso de comprobación de las condiciones anteriores, se utiliza la ecuación (5) para realizar el cálculo del FMFAC de las clases uno, dos y tres, ubicadas en la Figura 55. En donde los métodos que tienen "= 0", significa que son abstractos, y los que no lo tienen, quiere decir que tienen una implementación en su cuerpo. Obteniéndose los siguientes resultados:

- FMFAC(ArquitecturaClases1) $= \frac{\Sigma FFC}{\Sigma Tc} = \frac{1/2 + 2/3 + 1/2}{5} = \frac{0.5+0.666\ldots+0.5}{5} = \frac{1.666\ldots}{5} =$ 0.333…
- FMFAC(ArquitecturaClases2) $= \frac{\Sigma FFC}{\Sigma Tc} = \frac{3/5 + 1/4}{4} = \frac{0.6+0.25}{4} = \frac{0.85}{4} = 0.2125$

- FMFAC(ArquitecturaClases3) = $\frac{\Sigma FFC}{\Sigma \mathrm{T}c} = \frac{2/_4 + 1/_3}{4} = \frac{0.5+0.333\ldots}{4} = \frac{0.833\ldots}{4} = 0.2083\ldots$





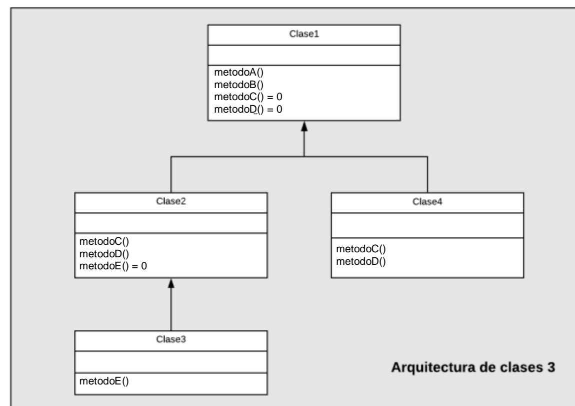


Figura 55.- Arquitectura de clases de un sistema.

Para comprobar que la relación binaria es de orden débil, tiene que cumplir con las propiedades de transitividad y completitud.

Demostración empírica de la propiedad de transitividad:

ArquitecturaClases1 •≥ ArquitecturaClases2,
ArquitecturaClases2 •≥ ArquitecturaClases3 →
ArquitecturaClases1 •≥ ArquitecturaClases3

↔

FMFAC(ArquitecturaClases1) •≥ FMFAC (ArquitecturaClases2) &
FMFAC(ArquitecturaClases2) •≥ FMFAC (ArquitecturaClases3) →
FMFAC (ArquitecturaClases1) •≥ FMFAC (ArquitecturaClases3)

Demostración formal de la propiedad de transitividad:

Reflejado en números, se tiene que:

$$0.333\ldots \geq 0.2125 \ \& \ 0.2125 \geq 0.2083\ldots \rightarrow 0.333\ldots \geq 0.2083\ldots$$

Como se puede observar, la arquitectura de clases 1 tiene un factor medio de flexibilidad mayor al de la arquitectura de clases 2. Éste a su vez tiene un factor medio de flexibilidad mayor al de la arquitectura de clases 3. Concluyendo que la relación binaria •≥, cumple con la propiedad de transitividad.

Demostración empírica de la propiedad de completitud:

Si se tienen las arquitecturas de clases uno y dos, se debe poder decir que la arquitectura de clases 1 *"tiene mayor o igual factor medio de flexibilidad de la arquitectura de clases que"* la arquitectura de clases 2, o viceversa. Es decir:

ArquitecturaClases1•≥ ArquitecturaClases2 o ArquitecturaClases2•≥ ArquitecturaClases1

Demostración formal de la propiedad de transitividad:

Calculando el FMFAC en las arquitecturas de clases uno y dos de la Figura 55 siempre fue posible determinar cuando existía mayor o igual factor medio de flexibilidad de la arquitectura de clases.

Demostración empírica de FMFAC es un homomorfismo:

Comprobando la segunda condición quedaría de la siguiente manera:

La arquitectura de clases uno “tiene mayor o igual factor medio de flexibilidad de la arquitectura de clases que” la arquitectura de clases dos

Formalmente:

$$0.333\ldots \geq 0.2125$$

Conclusión:

La relación binaria “tiene mayor o igual factor medio de flexibilidad de la arquitectura de clases” de la métrica FMFAC, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Suze, 1992).

REFERENCIAS

Al Dallal, J. (2015). Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, *58*, 231–249. https://doi.org/10.1016/j.infsof.2014.08.002

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language*.

Allen, M. J., & Jen, W. M. (1979). Introduction to Measurement Theory (pp. 6–8). Wadsworth Pub Co.

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17. https://doi.org/10.1109/32.979986

Calero, C. (2002). *Métricas para la Calidad de los Sistemas de Información*.

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

Castillo, L. E. S. (2005). *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter*. Centro Nacional de Investigación y Desarrollo Tecnológico.

Chawla, M. K., & Chhabra, I. (2013). Capturing OO Software Metrics to attain Quality Attributes – A case study. *International Journal of Scientific & Engineering Research*, *4*(6), 359–363.

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

Deitel P. J., D. H. M. (2014). *Cómo programar en Java*. *Pearson Education* (Vol. 14). https://doi.org/9702605318

E. Sweet, R. (1985). The Mesa programming environment. *SIGPLAN Notices*, *20*(7), 216–229.

Eclipse. (2015). JDeodorant. Recuperado a partir de https://marketplace.eclipse.org/content/jdeodorant

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

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). *Design Patterns – Elements of Reusable Object-Oriented Software* (Second). https://doi.org/10.1093/carcin/bgs084

García Cota, E. J., & Troyano Jiménez, J. A. (2003). *Guía práctica de ANTLR 2.7.2* (Vol. 0).

García Peñalvo, F. J., Marqués Corral, J. M., & Maudes Raedo, J. M. (1997). *Análisis y Diseño Orientado al Objeto para Reutilización*.

Gary, F. (2010). Measurement Theory for Software Engineers. Recuperado a partir de https://courses.cs.ut.ee/2010/se/uploads/Main/measurement-theory.pdf

IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*, 1–84. https://doi.org/10.1109/IEEESTD.1990.101064

Khatchadourian, R., & Masuhara, H. (2017a). 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

Khatchadourian, R., & Masuhara, H. (2017b). Defaultification refactoring: A tool for automatically converting Java methods to default. *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 984–989. https://doi.org/10.1109/ASE.2017.8115716

Kruchten, P. (2016). Refininig the definition of technical debt. *At a Dagstuhl seminar, sponsored by the Leibniz society*. Recuperado a partir de https://philippe.kruchten.com/2016/04/22/refining-the-definition-of-technical-debt/

Martin Fowler, K. B. (2009). *Refactoring: Improving the Design of Existing Code* (Vol. 12). https://doi.org/10.1007/s10071-009-0219-y

Meyer, B. (1988). *Object-Oriented Software Construction*. (P. Hall, Ed.) (Second).

Opdyke, W. F., & Johnson, R. E. (1993). Creating abstract superclasses by refactoring. https://doi.org/10.1145/170791.170804

Oracle. (2017). Oracle Java Documentation. Recuperado a partir de https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Ortiz, O. (2019). Métricas para la Medición del Factor de Flexibilidad y el Factor de Herencia de Implementación de Sistemas de Software.

Padhy, N., Singh, R. P., & Chandra, S. (2018). Software reusability metrics estimation : Algorithms , models and optimization techniques. *Computers and Electrical Engineering*, *69*, 653–668. https://doi.org/10.1016/j.compeleceng.2017.11.022

Padilla, P. (2019). *Método de Refactorización de Código Java con Interfaces y Abstracciones Incorrectas*. Centro Nacional de Investigación y Desarrollo Tecnológico.

Pinto, A. (2015). Interface inheritance vs Implementation inheritance. Recuperado a partir de https://www.linkedin.com/pulse/why-prefer-interface-inheritance-implementation-adrain-pinto

Rajnish, K., & Bhattacherjee, V. (2007). Class Inheritance Metrics-An Analytical and Empirical Approach.

Rathee, A. mit, & Chhabra, J. K. (2018). Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering. *Procedia Computer Science*, *125*, 740–746. https://doi.org/10.1016/j.procs.2017.12.095

Santaolaya Salgado, R., Fragoso Diaz, O. G., Ortiz Gutierrez, O., Bautista Juarez, C. V., & Barrera Monje, H. (2019). *Marco Orientado a Objetos para la Medición de Calidad de Arquitecturas de Software*. Cuernavaca, Morelos.

Suze, H. (1992). Properties of software measures. *Software Quality Journal*, *1*, 225–260. https://doi.org/https://doi.org/10.1007/BF01885772

Terence, P., & Harwell, S. (2018). Java Antlr Grammar. Recuperado a partir de https://github.com/antlr/grammars-v4/tree/master/java

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

Zafeiris, V. E., Poulias, S. H., Diamantidis, N. A., & Giakoumakis, E. A. (2017). Automated refactoring of super-class method invocations to the Template Method design pattern. *Information and Software Technology*, *82*, 19–35. https://doi.org/10.1016/j.infsof.2016.09.008