Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos

presentada por

Lic. Marisol Ramírez Cruz

como requisito para la obtención del grado de

Maestra en Ciencias de la computación

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Diciembre de 2022.

cenidet®
Centro Nacional de Investigación
y Desarrollo Tecnológico

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA | TECNOLÓGICO NACIONAL DE MÉXICO

Centro Nacional de Investigación y Desarrollo Tecnológico
Departamento de Ciencias Computacionales

Cuernavaca, Mor., 27/octubre/2022

OFICIO No. DCC/006/2022
Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

DR. CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de la C. MARISOL RAMÍREZ CRUZ, con número de control M19CE012, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"METODOS DE RE-FACTORIZACIÓN DE CÓDIGO JAVA PARA MEJORAR SU MODULARIDAD Y REDUCIR LAS DEPENDENCIAS ENTRE CLASES DE 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
Director de tesis

DRA. OLIVIA GRACIELA FRAGOSO DÍAZ
Revisor 1

M.C. HUMBERTO HERNÁNDEZ GARCÍA
Revisor 2

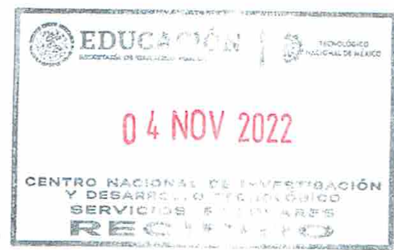
C.c.p. Depto. Servicios Escolares.
Expediente / Estudiante
JGGS/ibm

Cuernavaca, Mor., 03/noviembre/2022
No. De Oficio: SAC/156/2022
Asunto: Autorización de impresión de tesis

MARISOL RAMÍREZ CRUZ
CANDIDATO(A) AL GRADO DE MAESTRO(A) 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 **"METODOS DE RE-FACTORIZACIÓN DE CÓDIGO JAVA PARA MEJORAR SU MODULARIDAD Y REDUCIR LAS DEPENDENCIAS ENTRE CLASES DE 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®
"Educación Tecnológica al Servicio de México"

DR. CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

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

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

CMAZ/CHG

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

Dedicatorias

Dedico todo mi esfuerzo y trabajo a mis padres y a mi hermana por siempre apoyarme y por enseñarme que todo en esta vida es posible.

Agradecimientos

Expreso mis más sinceros agradecimientos:

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

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

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

A mi director de tesis Dr. René Santaolaya Salgado, por apoyarme y guiarme en este trabajo de investigación. Por su dedicación y paciencia mostrada en todo momento para terminar este trabajo. De igual forma agradezco todo su apoyo en el proceso relacionado con mi titulación.

A mis revisores de tesis Dra. Olivia Graciela Fragoso Díaz, M.C. Humberto Hernández García y que dedicaron su tiempo y paciencia en la revisión de mi trabajo y que contribuyeron con sus comentarios constructivos sobre el mismo.

A los profesores que contribuyeron en mi formación académica y profesional durante los dos años de maestría, por haber compartido conmigo su conocimiento, por sus sabios consejos y por la disposición mostrada en todo momento.

A mis padres Asunción Ramírez Martínez y Lilia Dilbar Cruz Cruz pues ellos fueron el principal cimiento para la construcción de mi vida académica y profesional, pues sentaron en mí las bases de responsabilidad y deseos de superación.

A mi hermana Gabriela Ramírez Cruz que siempre estuvo conmigo brindándome su apoyo.

A mi novio Jesús Santiago Manzano por el apoyo brindado en los momentos más difíciles.

A mis compañeros y amigos: Ricardo Federico Tello Díaz, Nélida Barón Pérez, Felipe de Jesús Rodríguez Pérez, Julio Cesar Bernabé Analco, Orlando Ortiz, Valentino Velázquez Olivera, Cruz Violeta Bautista Juárez, Iván Humberto Fuentes Chab, por compartir grandes experiencias y conocimientos conmigo.

Resumen

En la actualidad, el desarrollo de sistemas de software requiere que el código se realice de forma adecuada con la aplicación de técnicas y decisiones de diseño que se adapten a las necesidades. Sin embargo, existen diversos sistemas que no cumplen con la calidad adecuada y contienen código desagradable dentro de estos, originando deuda técnica con lo que se ocasiona que los módulos no puedan reutilizarse y el costo de mantenimiento sea elevado. Existen diversas causas que originan esta situación, por ejemplo, clases de objetos que atienden a más de una responsabilidad. Al contener más de una secuencia de métodos, estas clases contienen métodos que no participan colaborativamente para el mismo objetivo, y que no hacen uso de todos los atributos. Otro ejemplo que causa deuda técnica son arquitecturas de clases que muestran un alto grado relaciones de dependencia.

La re-factorización juega un papel importante para la reducción de deuda técnica en los sistemas de software legado, ya que esta técnica mejora la estructura interna sin que se altere el comportamiento externo del código original.

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

Para la verificación de los dos métodos de re-factorización propuestos en esta investigación se realizó lo siguiente: Para el primer método de re-factorización se utilizaron tres casos de estudio, con la división de clases de acuerdo con las responsabilidades y la división de acuerdo con la relación de los componentes de la clase (métodos y atributos). Con respecto al segundo método, dos casos de estudio donde se realizó la implantación del patrón de diseño mediator para reducir las dependencias entre clases al redirigir la comunicación de las clases a una clase mediadora. Estos casos de estudio muestran, además, como se lleva a cabo la re-factorización de forma automática y que se ha cumplido con el objetivo de la investigación.

Abstract

Currently, the development of software systems requires that the code is made in an adequate way with the application of techniques and design decisions that adapt to the needs. However, there are several systems that do not comply with the adequate quality and contain unpleasant code within them, causing technical debt, which means that the modules cannot be reused and the maintenance cost is high. There are several causes that originate this situation, for example, object classes that serve more than one responsibility. By containing more than one sequence of methods, these classes contain methods that do not participate collaboratively for the same objective, and that do not make use of all the attributes. Another example that causes technical debt are class architectures that show a high degree of dependency relationships.

Re-factoring plays an important role in reducing technical debt in legacy software systems, since this technique improves the internal structure without altering the external behavior of the original code.

In order to solve some problems, this paper describes the development of two re-factoring methods: the first one brings an advance in the search for improving modularity by increasing the coherence and cohesion of object classes, the second method with the aim of reducing dependencies between object classes in systems developed in Java language. These methods allow extending the functionality of the SR2-Refactoring system by adding new methods for refactoring code.

For the verification of the two re-factoring methods proposed in this research, the following was performed: For the first re-factoring method, three case studies were used, with the division of classes according to responsibilities and the division according to the relationship of class components (methods and attributes). Regarding the second method, two case studies where the implementation of the mediator design pattern was performed to reduce the dependencies between classes by redirecting the communication of the classes to a mediator class. These case studies also show how the re-factoring is carried out automatically and that the research objective has been met.

ÍNDICE GENERAL

Capítulo 1. Introducción **16**

1.1 Organización de la tesis 17

Capítulo 2. Antecedentes **19**

2.1 Planteamiento del problema 19

2.2 Objetivo General 19

2.2.1 Objetivos Específicos 19

2.3 Desarrollos tecnológicos sobre re-factorización en el TecNM / CENIDET 20

2.3.1 SR2-Refactoring 20

2.4 Trabajos relacionados 25

Capítulo 3. Marco Conceptual **34**

3.1 Paradigma de programación orientada a objetos 34

3.1.1 Clase 34

3.1.2 Objeto 34

3.1.3 Representación interna de una clase 35

3.2 Modularidad 36

3.2.1 Completitud 36

3.2.2 Suficiencia 38

3.2.2.2 Dependencia 39

3.2.2.3 Acoplamiento 39

3.3 Re-factorización (Refactoring) 40

3.4 Código desagradable (Smell Code) 40

3.5 Métrica de calidad 40

3.5.1 COF (Coupling Factor) 41

3.5.2 LCOM (Lack of Cohesion in Methods) 41

3.5.3 CrCU (Coherencia de caso de uso) 41

3.6 Flexibilidad 41

3.7 Extensibilidad 41

3.8 ANTLR 41

3.9 Patrón de diseño 42

3.9.1 Patrón de diseño “Mediator” 42

Capítulo 4. Materiales y métodos de solución **44**

4.1 Métrica COF (Factor de acoplamiento)......44

4.2 Métrica LCOM* (Carencia de cohesión)......45

4.3 Métrica CrCU(Coherencia de Caso de Uso)......46

4.4 Diseño de la métrica FR (Factor de responsabilidades)......46

4.5 Proceso de cada uno de los métodos de re-factorización......47

4.5.1 Proceso de re-factorización del método para mejorar la modularidad......47

4.5.2 Proceso de re-factorización para reducir las dependencias entre clases de objetos......49

Capítulo 5. Desarrollo del sistema......51

5.1 Análisis de casos de uso......51

5.2 Diagrama de secuencias del cálculo de métricas......61

5.3 Diagrama de secuencias de los métodos de re-factorización......61

5.3.1 Mejora de modularidad......62

5.3.2 Reducción de dependencia entre clases......62

5.4 Diagrama de clases del sistema......63

Capítulo 6. Pruebas de Evaluación......68

6.1 Mejora de modularidad......68

6.2 Reducción de dependencias entre clases de objetos......103

Capítulo 7. Conclusiones y trabajos futuros......122

Conclusiones:......122

Aportaciones......124

Trabajos futuros......124

Referencias......126

Anexo A. Sustento de métrica FR......129

ÍNDICE DE FIGURAS

Figura 1. Modelo conceptual de una clase. 34
Figura 2. Envío de mensaje de un objeto de la clase Cliente a un objeto “op” de la clase Operaciones para realizar la suma de dos números. 36
Figura 3. Ejemplo de una clase con baja cohesión. 37
Figura 4. Ejemplo de una clase con alta cohesión. 37
Figura 5. Secuencias de la clase A. 38
Figura 6. Secuencias de la clase A. 38
Figura 7. Comunicación entre módulos. 39
Figura 8.Diagrama BPMN correspondiente a la re-factorización de código del algoritmo del método para mejorar la modularidad 48
Figura 9. Diagrama BPMN correspondiente a la re-factorización de código del algoritmo del método para reducir las dependencias entre clases de objetos. 50
Figura 10. Diagrama de caso de uso general 51
Figura 11. Diagrama de caso de uso métrica FR 57
Figura 12. Diagrama de caso de uso métrica LCOM* 58
Figura 13. Diagrama de caso de uso métrica CrCU. 59
Figura 14. Diagrama de caso de uso métrica COF 60
Figura 15. Diagrama de secuencias del cálculo de métricas 61
Figura 16. Diagrama de secuencias del primer método de re-factorización para mejora de la modularidad 62
Figura 17. Diagrama de secuencias del segundo método de re-factorización para reducción de dependencias entre clases de objetos 63
Figura 18. Diagrama de clases general del sistema 64
Figura 19. Diagrama de clases del marco de métricas 65
Figura 20. Diagrama de clases del paquete modelo. 66
Figura 21. Diagrama detallado del paquete modelo 67
Figura 22. Convención de nombres. 68
Figura 23. Arquitectura de clases de la aplicación Marco estadístico antes de la re-factorización. 78
Figura 24. Arquitectura de clases de la aplicación Marco estadístico resultante de la re-factorización. 83
Figura 25. Arquitectura de clases de la aplicación LibraryAdministration antes de la re-factorización. 89
Figura 26. Arquitectura de clases de la aplicación LibraryAdministration después de la re-factorización. 92
Figura 27. Arquitectura de clases de la aplicación ConversionAutomata antes de la re-factorización. ..96
Figura 28. Arquitectura de clases de la aplicación Conversión Autómata después de la re-factorización. 99
Figura 29. Convención de nombres. 103
Figura 30. Arquitectura de clases de la aplicación *LibraryAdministration* antes de la re-factorización. 112
Figura 31. Arquitectura de clases de la aplicación LibraryAdministration después de la re-factorización. 114
Figura 32. Arquitectura de clases de la aplicación ConversionAutomata antes de la re-factorización. 117

Figura 33. Arquitectura de clases de la aplicación ConversionAutomata despúes de la re-factorización. 119
Figura 35. Arquitectura 1. 130
Figura 36. Arquitectura 2. 131
Figura 37. Arquitectura 3. 131

ÍNDICE DE TABLAS

Tabla 1. Comparativa de artículos relacionados....................31
Tabla 2. Descripción de caso de uso “Re-factorizar coherencia”....................52
Tabla 3. Descripción de caso de uso “Re-factorizar Cohesión”....................53
Tabla 4. Descripción de caso de uso dependencias entre clases de objetos....................55
Tabla 5. Descripción de caso de uso “Generar código”....................56
Tabla 6. Descripción de caso de uso “Calcular métrica FR”....................57
Tabla 7. Descripción de caso de uso “Calcular métrica LCOM*“....................58
Tabla 8. Descripción de caso de uso “Calcular métrica CrCU”....................59
Tabla 9. Descripción de caso de uso “Calcular métrica COF”....................60
Tabla 10. Módulos de programa....................69
Tabla 11. Control de tareas....................69
Tabla 12. Características por probar....................70
Tabla 13. Características por probar del proceso del cálculo de métricas de calidad....................73
Tabla 14. Características por probar del proceso de re-factorización....................74
Tabla 15. Características por probar del proceso del cálculo de métricas de calidad....................75
Tabla 16. Características por probar del proceso de re-factorización....................75
Tabla 17. Características por probar del proceso del cálculo de métricas de calidad....................76
Tabla 18. Características por probar del proceso de re-factorización....................77
Tabla 19. Características por probar del proceso del cálculo de métricas de calidad....................77
Tabla 20. Cálculo manual y automático de la métrica FR de Marco estadístico....................79
Tabla 21. Cálculo manual y automático de la métrica CrCU de Marco estadístico....................80
Tabla 22. Cálculo manual y automático de la métrica LCOM* de Marco estadístico....................80
Tabla 23. Características por probar del proceso de re-factorización....................81
Tabla 24. Conjunto de datos ingresados a la prueba....................84
Tabla 25. Cálculo de la media antes y después de la re-factorización de la aplicación Marco estadístico....................84
Tabla 26. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación Marco estadístico....................85
Tabla 27. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación Marco estadístico....................86
Tabla 28. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación Marco estadístico....................87
Tabla 29. Características por probar del proceso del cálculo de métricas de calidad....................88
Tabla 30. Cálculo manual y automático de la métrica FR de LibraryAdministration....................90
Tabla 31. Cálculo manual y automático de la métrica CrCU de LibraryAdministration....................90
Tabla 32. Cálculo manual y automático de la métrica LCOM* de LibraryAdministration....................91
Tabla 33. Características por probar del proceso de re-factorización....................91
Tabla 34. Datos ingresados en la prueba....................93

Tabla 35. Inserción de un usuario en la aplicación *LibraryAdministration* antes y después de la re-factorización 93
Tabla 36. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación LibraryAdministration. 94
Tabla 37. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación LibraryAdministration. 95
Tabla 38. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación LibraryAdministration 95
Tabla 39. Características por probar del proceso del cálculo de métricas de calidad 96
Tabla 40. Cálculo manual y automático de la métrica FR de ConversionAutomata 97
Tabla 41. Cálculo manual y automático de la métrica CrCU de ConversionAutomata. 97
Tabla 42. Cálculo manual y automático de la métrica LCOM* de ConversionAutomata. 98
Tabla 43. Características por probar del proceso de re-factorización 99
Tabla 44. Datos ingresados en la prueba 100
Tabla 45. conversión de una expresión infija a postfija. 100
Tabla 46. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación Conversión Autómata. 101
Tabla 47. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación Conversión Autómata. 102
Tabla 48. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación ConversionAutomata 102
Tabla 49. Módulos de programa. 105
Tabla 50. Control de tareas. 105
Tabla 51. Características por probar. 106
Tabla 52. Características por probar del proceso del cálculo de métricas de calidad. 109
Tabla 53. Características por probar del proceso de re-factorización. 109
Tabla 54. Características por probar del proceso del cálculo de métricas de calidad. 110
Tabla 55. Características por probar del proceso de re-factorización. 111
Tabla 56. Características a probar del proceso del cálculo de métricas de calidad. 111
Tabla 57. Cálculo manual y automático de la métrica COF de LibraryAdministration. 113
Tabla 58. Características por probar del proceso de re-factorización. 113
Tabla 59. Datos ingresados en la prueba 115
Tabla 60. Inserción de datos de un libro en la aplicación *LibraryAdministration* antes y después de la re-factorización 115
Tabla 61. Valores obtenidos de la métrica COF antes y después de la re-factorización de la aplicación LibraryAdministration 116
Tabla 62. Características por probar del proceso del cálculo de métricas de calidad. 116
Tabla 63. Cálculo manual y automático de la métrica COF de ConversionAutomata. 118
Tabla 64. Características por probar del proceso de re-factorización. 118
Tabla 65. Datos ingresados en la prueba 119
Tabla 66. conversión de una expresión infija a postfija. 120

Tabla 67. Valores obtenidos de la métrica COF antes y después de la re-factorización de la aplicación ConversionAutomata..120
Tabla 68. Resumen de los resultados de las pruebas..123
Tabla 69. Resumen de los resultados de las pruebas..123

Capítulo 1. Introducción

En la actualidad, cuando se desarrolla software y se realizan decisiones equivocadas o erróneas tanto de diseño como de implementación, debido a la falta de experiencia y habilidad en el desarrollo para aplicar las técnicas adecuadas, como el desconocimiento conceptual de reglas y principios de diseño, métricas internas de calidad y el uso de patrones de diseño, se origina deuda técnica en el código, debido a que se producen unidades de programa con segmentos de código desagradable (*smell code*) que resultan poco reusables y costosos en su mantenimiento (Fields et al., 2010; Khomh et al., 2009; Tufano et al., 2017).

El uso de técnicas adecuadas; la aplicación de principios, tanto de diseño como de codificación; tener presente una cultura de calidad; así como de la aplicación de patrones de diseño, permite que los sistemas de software mejoren su calidad y generen segmentos de código reutilizables y menos costosos en su mantenimiento.

Los patrones de diseño (*Design Patterns*) son soluciones comprobadas y habituales a problemas comunes en el diseño de software. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular del código (Gamma et al., 2005; R. Khatchadourian & Masuhara, 2017; Zafeiris et al., 2017).

De acuerdo a diversos estudios realizados en la literatura consultada se observa que la re-factorización mejora la reutilización de código, debido a que permite mejorar las arquitecturas existentes de software legado escritos en distintos lenguajes, mediante la automatización del proceso para transformar el código desagradable en código limpio y con un diseño simple manteniendo la funcionalidad original.

También se observa que, aunque existen diversos trabajos relacionados con la re-factorización aún no se cuenta con sistemas que automaticen la re-factorización para mejorar problemáticas como el alto grado de dependencias y que aporten en la mejora de la modularidad (Dos Santos Neto et al., 2015; R. T. Khatchadourian et al., 2016; Napoli et al., 2013; Rathee & Chhabra, 2018).

Ante la necesidad de automatizar este proceso y mejorar la problemática, en esta investigación se consideró el diseño e implementación de dos métodos de re-factorización, que tienen por objetivo mejorar el diseño de arquitecturas de software existentes en lenguaje Java, mediante la reducción de las dependencias entre clases y la mejora de la modularidad en suficiencia y completes, mediante la fragmentación de unidades de programa con mejores

cualidades de coherencia, cohesión y con la implantación del patrón de diseño Mediator para reducir las dependencias caóticas entre objetos.

1.1 Organización de la tesis

El presente trabajo de investigación se compone de seis capítulos, una sección de referencias y una sección de anexos. A continuación, se describen de forma general los capítulos que integran esta tesis.

Capítulo 2. Antecedentes

Este apartado describe el problema planteado en la investigación, los objetivos de la investigación. También se muestran los trabajos antecedentes, realizados en el CENIDET y los trabajos relacionados con la presente investigación, consultados en diversas revistas científicas relacionados con la presente investigación.

Capítulo 3. Marco Conceptual

Este apartado se integra de conceptos básicos y fundamentos teóricos necesarios para comprender el tema científico y tecnológico que se aborda en esta tesis.

Capítulo 4. Materiales y métodos de solución

En este capítulo se describen las métricas utilizadas en la investigación (FR, LCOM*, CrCU y COF). También se muestra la descripción de los procesos que componen los métodos de re-factorización con los cuales se propone una solución a la problemática planteada y así cumplir con los objetivos planteados.

Capítulo 5. Desarrollo del sistema

En este apartado se desglosa el análisis, diseño e implementación de los métodos de re-factorización que resuelven la problemática planteada en la investigación. Además, se describen los diagramas que permiten entender el problema y su solución.

Capítulo 6. Pruebas de evaluación

En este apartado se desglosa el plan de pruebas de cada método de re-factorización que incluye la especificación de diseño de casos de pruebas, especificación de casos de prueba y la ejecución de los casos de prueba. Con el fin de verificar que se cumple con los objetivos planteados en la investigación.

Capítulo 7. Conclusiones y trabajos futuros

Este capítulo muestra las conclusiones de la investigación, describe las aportaciones de la investigación y los trabajos futuros.

Capítulo 2. Antecedentes

2.1 Planteamiento del problema

En la actualidad muchos módulos de software legados no satisfacen completamente las demandas funcionales requeridas en nuevos sistemas, desarrolladas a partir de la integración, ensamble y/o composición de unidades reusables. Por ejemplo, existen unidades de software que exhiben alto grado de dependencias y modularidad incorrecta, lo cual puede redundar en insuficiencia e incompletitud en sus partes. La completitud se encuentra en función del grado de relación que se tiene entre las funciones (coherencia) y las funciones con los atributos (cohesión). Por otro lado, la suficiencia se encuentra en función del grado de dependencias (acoplamiento) existente entre las clases de las arquitecturas de software.

Una causa que origina incoherencia se debe al diseño incorrecto de arquitecturas de software, integrada por clases, las cuales atienden a más de una responsabilidad, al contener más de una secuencia de métodos y/o tienen menor o mayor funcionalidad de la necesaria para alcanzar un único objetivo o meta de valor para los clientes (Tsantalis & Chatzigeorgiou, 2009). Por otro lado, algunos de estos sistemas también muestran clases con métodos que no hacen uso de todos los atributos de las clases (baja cohesión).

Por otro lado, algunas arquitecturas de software de mayor granulación muestran problemas de insuficiencia, por lo que, para alcanzar su objetivo o meta de valor, necesitan complementar la funcionalidad desde otros módulos de programa. Estas arquitecturas muestran alto acoplamiento debido a un excesivo número de relaciones de dependencia entre los módulos (Bois et al., 2004).

2.2 Objetivo General

El objetivo de este proyecto de investigación es mejorar el diseño de arquitecturas de componentes de software existentes en lenguaje Java mediante la reducción de las dependencias entre clases y mejorar su modularidad en suficiencia y completitud.

2.2.1 Objetivos Específicos

- Reducir las dependencias de clases de sistemas legados en lenguaje Java.

- Mejorar el Diseño de Arquitecturas de Componentes de Software Existentes escritos en lenguaje Java.
- Extender la funcionalidad del SR2-Refactoring, mediante la aportación de nuevos métodos de re-factorización resultantes de esta investigación.
- Lograr mayor autosuficiencia y autonomía de entidades de software encaminadas a la mejora de la calidad para facilitar su mantenimiento y su capacidad de reutilización de las entidades de software legado.

2.3 Desarrollos tecnológicos sobre re-factorización en el TecNM / CENIDET

En apoyo a planteamientos problemáticos similares al descrito en este documento, en el CENIDET se han planteado varios proyectos de reingeniería y re-factorización para mejorar arquitecturas de software existente, escritos en lenguaje C++ y Java, los cuales son descritos a continuación.

2.3.1 SR2-Refactoring

SR2-Refactoring (Sistema de Reingeniería para Reuso) hasta ahora implementa cuatro métodos de re-factorización, y se tienen actualmente otros en desarrollo, que en conjunto mejoran la arquitectura de marcos de aplicaciones orientado a objetos de dominios, escritos en lenguaje C++ y JAVA. Actualmente el sistema implementa tres métricas para detectar problemas específicos de diseño en los marcos de aplicaciones orientados a objetos.

El sistema SR2-Refactoring fue implementado en lenguaje Java, utilizando el ambiente Eclipse, y como soporte utiliza el manejador de Base de Datos MySQL. La información técnica sobre los métodos de re-factorización y las métricas se encuentran englobadas en cuatro tesis de maestría del CENIDET que se muestran a continuación: 1) Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces, 2) Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter, 3) Re-factorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator y 4) Re-factorización de Sistemas Legados de Software, para equilibrar la Coherencia, Cohesión y el Factor de Acoplamiento de su Estructura Interna.

Son tres métodos que refactorizan código escrito en lenguaje C++:

1) "Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces", Manuel Alejandro Valdés Marrero (Valdés Marrero, 2004).

2) "Re-factorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator", Leonor Cárdenas Robledo (Cárdenas Robledo, 2004).
3) "Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter", Luis Santos Castillo (Santos Castillo, 2005)

Son tres métodos que refactorizan código escrito en lenguaje Java que aún no se incorporan al SR2-Refactoring:

1) "Método de Re-factorizar de código java con interfaces y abstracciones incorrectas", Pablo Padilla Salgado. (Padilla, 2019)
2) "Re-factorizar de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos", Orlando Ortíz Gutiérrez (Ortiz Gutierrez, 2020)
3) "Método de re-factorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente", Nélida Barón Pérez (Barón, 2020)

Además, otros seis métodos de re-factorización se encuentran actualmente de desarrollo y que serán un aporte el sistema SR2-Refactoring.

1) "Métodos de re-factorización de arquitecturas de software con carencia de abstracciones", Ricardo Tello Díaz
2) "Re-factorización de Módulos Colaborativos con Carencia de Abstracción", Fernando Sánchez Rogel
3) "Disminución de deuda técnica producida por arquitecturas de clases con más de una responsabilidad", Juan Antonio Díaz Díaz
4) "Plug-In Para Integrar Los Métodos De Re-factorización Del SR2-Refactoring al Ide-Eclipse", Leslie A. Ruíz Bustos.
5) "Reducción de la Deuda Técnica por la Fragilidad Modular de Arquitecturas de software legado", Miguel Romero Meza
6) "Tratamiento de la Deuda Técnica originada por la Carencia de Protección de Funciones Plantilla de Software Legado", Elías A. Ramírez García.

2.3.2 Método de re-factorización de marcos de aplicaciones orientados a objetos por la Separación de Interfaces (Valdés Marrero, 2004)

Esta investigación consistió en el desarrollo de un método de re-factorización para la "Separación de interfaces", cuyo algoritmo se implementó en la Herramienta SR2-Refactoring. El método realiza la re-factorización de código de marcos orientados a objetos en lenguaje C++, con el propósito de reducir las dependencias de herencias de interfaz entre clases debido a que

puedan contener implementaciones nulas de las interfaces no utilizadas en las clases derivadas, con lo cual se viola el principio de sustitución de Barbara Liskov al debilitarse las postcondiciones en las funciones de las clases derivadas. Este proyecto diseñó e implementó la métrica orientada a objetos denominada "V-DINO" para medir el grado de dependencia que existe debido a interfaces que no son utilizadas.

2.3.3 Re-factorización de marcos orientados a objetos para reducir el acoplamiento aplicando el patrón de diseño Mediator (Cárdenas Robledo, 2004).

En este trabajo se creó un método de re-factorización que reduce el acoplamiento entre clases, debido a que existen muchas relaciones de asociación y/o dependencia entre ellas. Este método está escrito en lenguaje C++ y funciona para marcos orientados escritos en el mismo lenguaje. El método de re-factorización es conducido según la intención del patrón de diseño *"Mediator",* el cual incorpora en la arquitectura resultante la estructura de clases como se describe en la solución de este patrón de diseño. La re-factorización del método funciona automáticamente y fue incorporado a la herramienta SR2-Refactoring. También implementa la métrica COF que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF).

2.3.4 Adaptación de interfaces de marcos de aplicaciones orientados a objetos, usando el patrón de diseño *Adapter* (Santos Castillo, 2005)

En este trabajo se desarrolló un algoritmo de adaptación de interfaces entre código cliente y código servidor. Con este método se resuelve el problema cuando las interfaces de comunicación entre un cliente y servidor no empatan en tipo y/o número de parámetros. El método genera de manera semiautomática adaptadores de las interfaces, para ajustarlas a las necesidades del código cliente. El método fue integrado al sistema SR2-Refactoring y aplica para código existente escrito en lenguaje C++.

2.3.5 Método de re-factorización de código java con interfaces y abstracciones incorrectas (Padilla, 2019)

El método desarrollado en la investigación busca cumplir con el principio de "Separación de interfaces" localizando estructuras de código desagradable, tales como: abstracciones incorrectas que no respetan los principios de diseño de "sustitución" y de "única responsabilidad" en aplicaciones existentes escritas en lenguaje Java, las cuales son corregidas automáticamente. El método anula implementaciones nulas de funciones en clases derivadas y balancea las jerarquías de herencia tanto en lo vertical (clases descendientes) como en lo

horizontal (clases derivadas hermanas) para distribuir las responsabilidades entre las diferentes clases de una arquitectura Orientada a Objetos.

2.3.6 Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos (Ortiz Gutierrez, 2020)

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

2.3.7 Método de re-factorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software Existentes (Barón, 2020)

El método desarrollado en esta investigación tiene como objetivo aumentar la protección modular de las arquitecturas de clases. Para lograr este objetivo es necesario que cada atributo, así como cada función integradas en las clases de objetos de arquitecturas de software, presenten el calificador de alcance correcto de acuerdo a los cuatro niveles de visibilidad: 1) los atributos y funciones privadas ("*private*"), estas son las más protegidas puesto que solamente son utilizadas por otras funciones dentro de la misma clase; 2) los atributos y funciones protegidas ("*protected*"), solo pueden ser utilizadas por otras funciones de la misma clase o de clases derivadas; 3) los atributos y funciones amistosas ("*friendly*"), son aquellas utilizadas por otras funciones de cualquier clase integradas dentro del mismo paquete; 4) los atributos y las funciones públicas ("*public*"), son las más permisivas puesto que pueden ser utilizadas por cualquier función de cualquier clase en el sistema de software. Este método fue integrado a la herramienta SR2-Refactoring y funciona para marcos de aplicaciones orientados a objetos escritos en lenguaje Java.

Finalmente, el sistema SR2-Refactoring cuenta con un menú, para realizar las acciones de re-factorización y cálculo de métricas, así como acciones adicionales, como son la selección y comparación de archivos y el manejo de usuarios. La mayoría de las opciones de los menús llevan a pantallas o cuadros de diálogo, y cada uno de ellos cuenta con elementos gráficos, como botones y cuadros de texto.

2.4 Trabajos relacionados

En esta sección se describen los trabajos consultados que guardan cierta relación con el tema que se trata en esta investigación. De acuerdo con los trabajos analizados se observa que en algunos de estos se muestran diversas técnicas y métodos de re-factorización similares al objetivo de esta investigación con el cual se busca reducir las dependencias entre clases y mejora de la modularidad en suficiencia y completitud.

2.4.1 Using Modularity Metrics to assist Move Method Refactoring of Large Systems (Napoli et al., 2013)

El objetivo de ese trabajo de investigación es mejorar la modularidad de los sistemas de software a nivel de clases y métodos, así como reducir el tiempo de cálculo de métricas en grandes sistemas de software. El enfoque es encontrar automáticamente sugerencias de re-factorización que mejoren varios componentes a la vez, lo que se hace posible al calcular varias métricas e idear un algoritmo paralelo que se ejecuta en una GPU (unidad de procesamiento gráfico) para calcular métricas de productos que consumen mucho tiempo.

Al evaluar métricas como Fan-in, Fan-out (métricas estructurales), WMC (Weighted Methods Per Class), NOC (Number of Children), DIT(Depth of Inheritance Tree), CBO (Coupling between Objects), RFC (Response For Class) y LCOM (Lack of Cohesion of Methods) que son métricas orientadas a objetos. Se aboga por la posibilidad de mejorar la modularidad de un sistema orientado a objetos mediante la combinación de las métricas CBO, LCOM, Fan-in y Fan-out. Dentro de los experimentos se proporcionan valores de varias características de la estructura (clases, métodos y atributos) y las métricas calculadas para varios sistemas de software. Estos valores son comparados con los tiempos del uso de una sola CPU y una GPU Tesla. La ganancia de tiempo obtenida por el GPU que se obtuvo es buena en comparación con la CPU.

Como conclusión, para obtener una vista adecuada sobre el efecto de los cambios para un sistema de software grande, la cantidad de valores que se calculan crece muy rápidamente. Esto resulta abrumador para una sola CPU, por lo tanto, una GPU puede emplearse como solución. También es importante considerar que las oportunidades de re-factorización en sistemas grandes pueden ser más difíciles de evaluar para el desarrollador sin la ayuda de una herramienta de re-factorización.

2.4.2 Automated refactoring of super-class method invocations to the Template Method design pattern (Zafeiris et al., 2017)

El Método de plantilla es un patrón de diseño que permite extensiones a funciones (métodos de clase) de varios pasos sin anular su implementación concreta. El objetivo de esta investigación es reducir el acoplamiento de las subclases con la clase base por la herencia de implementación que dificulta su evolución y mantenimiento. El enfoque aplicado es mediante la sustitución de la herencia de implementación por la herencia de interfaz mediante la re-factorización automática de invocación a métodos de clases base (*call super*), utilizando el patrón de diseño *"Template Method"* (método de plantilla).

Se introdujo un algoritmo para el descubrimiento de candidatos de re-factorización que se basa en un conjunto extenso de condiciones previas de re-factorización. Además, se especificó la transformación del código fuente para re-factorizar una instancia de "*call super"* a un método de plantilla.

Se evaluó la implementación del enfoque propuesto en un conjunto de proyectos Java de código abierto. Los resultados de la evaluación resaltan: a) la frecuente aparición del patrón "*call super"* entre anulaciones de método, b) el potencial proporcionado por el método para el descubrimiento y eliminación de varios las instancias no triviales de "*call super"* y c) la mejora del código resultante.

Con base en los resultados de los experimentos se concluye que el método propuesto automatiza el reemplazo de la herencia de implementación por herencia de interfaz a través de la re-factorización de instancias "*call super"* al Método de plantilla. La evaluación empírica del método respalda su aplicabilidad, solidez y eficiencia en el tiempo de ejecución.

2.4.3 AutoRefactoring: A platform to build refactoring agents (Dos Santos Neto et al., 2015)

El mantenimiento del software puede degradar la calidad de este. Una de las principales formas de reducir los efectos no deseados del mantenimiento es la re-factorización. Esta es una técnica para mejorar la calidad del código de software sin cambiar su comportamiento observable. El objetivo de esta investigación es mejorar la calidad de software en cuanto a flexibilidad, reutilización, efectividad y extensibilidad.

En este trabajo se propone una plataforma basada en agentes, llamada plataforma AutoRefactoring, que permite el desarrollo de un agente para realizar de manera autónoma la re-factorización. Para aplicar de manera segura una re-factorización, se deben considerar los siguientes pasos: 1) identificar las partes del código que deben mejorarse; 2) determinar los

cambios que deben aplicarse al código para mejorarlo; 3) evaluar los impactos de las correcciones en la calidad del código; y 4) comprobar que el comportamiento observable del software se mantendrá después de aplicar las correcciones a código como: *Data Clumps* y *Public Fields*.

En la experimentación se aplicó el enfoque a cinco proyectos Java de código abierto, a saber, Log4j, 5 SweetHome3D, 6 HSQLDB, 7 jEdit8 y Xerces.9. Se corrigió hasta el 80.56% de código smell detectado, mientras que la calidad del código evolucionó hasta el 70.54% y preservó el comportamiento del software. Como conclusión el enfoque es capaz de hacer frente a los requisitos de re-factorización mencionados: la identificación del código no deseado, la determinación de correcciones, la mejora de la calidad y la preservación del comportamiento.

2.4.4 Towards Improving Interface Modularity in Legacy Java Software through Automated Refactoring (R. T. Khatchadourian et al., 2016)

Java 8 es una de las actualizaciones más grandes para el lenguaje Java. Existen varias características nuevas y clave que pueden ayudar a que los programas sean más fáciles de leer, escribir y mantener. Esta investigación tiene como objetivo mejorar el modularidad en la dimensión de abstracción.

Tiene como enfoque la creación de una herramienta de re-factorización automatizada que aumenta la modularidad de la interfaz de Java mediante la migración automática de la definición de métodos en las clases a las interfaces de Java 8 como *"default method"*.

En la experimentación se analizaron proyectos grandes de código abierto en busca de ocurrencias comunes y variaciones del patrón *"Skeletal implementation"*. Se propone que el enfoque se pueda ampliar más adelante para tratar las variaciones de patrones. Se concluye que al aumentar la modularidad de la interfaz de Java mediante la migración automática de las definiciones de los métodos en clases a las interfaces de Java 8 hace que las interfaces Java sean más fáciles de implementar.

2.4.5 Modularity-Oriented Refactoring (Bryton & Brito e Abreu, 2008)

La re-factorización, a pesar de ser ampliamente reconocida como una de las mejores prácticas de diseño y programación orientada a objetos, aún carece de bases cuantitativas y herramientas eficientes para tareas como detectar código desagradable de mal olor. Elegir la re-factorización más adecuada o validar la bondad de los cambios. El objetivo de esta investigación es mostrar bases cuantitativas para detectar código desagradable en los sistemas de software.

El enfoque es mediante la utilización de un método de re-factorización denominado MORe para remover el código desagradable existente en los sistemas de software a través de paradigmas basada en métricas de modularidad independientes de paradigma y formalismo. El método MORe consiste en una secuencia de 7 pasos. Cada uno con su respectivo resultado que en algunos casos es la entrada del siguiente.

La evaluación del trabajo se divide en tres fases: La primera es obtener tanto conocimiento como sea posible sobre el problema a la mano, en la segunda fase, se implementan las propuestas de investigación de acuerdo a la información, y en la tercera fase, los resultados se validarán con un estudio de caso, los patrones de diseño de GoF (es decir los patrones de diseño creados por el grupo Gang of Four compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides) implementados tanto en Java como en AspectJ. Se concluye que la estrategia seguida por el método MORe puede utilizarse por métodos similares que apunten a automatizar la re-factorización, con respecto a diferentes propiedades de calidad.

2.4.6 Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering (Rathee & Chhabra, 2018)

Aumentar la calidad del diseño del software es un desafío de investigación clave en el sistema de desarrollo de software orientado a objetos. La cohesión es uno de los aspectos clave que ayuda a evaluar la calidad y la modularidad de un sistema de software a nivel de diseño. El objetivo de esta investigación es mejorar la cohesión de los elementos de software que conforman el diseño estructural de sistemas de software para aumentar su calidad.

El agrupamiento se realiza mediante el uso de un nuevo algoritmo de agrupamiento propuesto denominado FUPClust (Clustering basado en patrones de uso frecuente) basado en las interacciones FUP (Patrones de uso frecuente) entre los módulos. La experimentación se llevó a cabo con dos proyectos Java (Junit y HospitalAutomation) estándar, descargados del repositorio de código abierto GitHub. Los sistemas de software se estudian manualmente y se extraen los datos de FUP. En el caso de Junit, la cohesión general del software, calculada como el promedio de los valores de cohesión de los módulos individuales de los sistemas de software correspondientes, se incrementa en alrededor del 34%. Similar es el caso de *HospitalAutomation* en el que la mejora general en la cohesión es de alrededor del 33%.

Como conclusión, en ese trabajo se ha medido y mejorado la cohesión de un software orientado a objetos a nivel de módulo con resultados favorables por lo que la métrica puede aplicarse en trabajos futuros.

2.4.7 Performance-driven software model refactoring (Arcelli et al., 2018)

La re-factorización relacionada con los requisitos funcionales ha sido ampliamente estudiada en los últimos años. La re-factorización no funcional sigue siendo crítica, principalmente porque las características no funcionales del software son difíciles de evaluar y las acciones de re-factorización apropiadas pueden ser difíciles de identificar. Esta investigación tiene como objetivo la satisfacción de los requisitos de rendimiento de los clientes en cuanto al tiempo de ejecución de los requerimientos del cliente.

Se implementaron, dentro de la plataforma EPSILON, reglas de detección y acciones de re-factorización en modelos UML para un conjunto de anti patrones de rendimiento bien conocidos. En la experimentación se utilizó el marco en un sistema de gestión de un jardín botánico para mostrar, por un lado, que las antipartículas pueden conducir de manera efectiva la re-factorización del software hacia modelos que satisfacen los requisitos de rendimiento y, por el otro, que la automatización introducida por las sesiones basadas en EPSILON permite Inspeccionar múltiples caminos y proponer una variedad de soluciones.

Los autores concluyen que este trabajo demuestra que, la automatización de la re-factorización de modelos de software basada en el rendimiento puede ser beneficiosa, y que los anti patrones de rendimiento pueden ser instrumentos poderosos en manos de los ingenieros de software para detectar y resolver problemas de rendimiento generalmente ocultos al análisis tradicional de cuellos de botella.

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

Los métodos predeterminados de Java 8, que permiten que las interfaces contengan implementaciones de métodos (de instancia) que son útiles para el patrón de diseño *"Skeletal implementation"*. El objetivo de la investigación es mejorar la estructura del código de java heredado. Se utiliza un enfoque de re-factorización para la migración a los métodos predeterminados de la interfaz mejorada de Java 8. Se abren las clases a la herencia permitiendo que las clases hereden múltiples definiciones de interfaz.

En la experimentación se implementó el enfoque como un plug-in de Eclipse y se aplicó a 19 proyectos Java del mundo real, así como a solicitudes de extracción enviadas a los populares repositorios de GitHub. Los resultados muestran que la herramienta puede escalar y re-factorizar sistemas escritos en ese lenguaje.

En este trabajo, se concluye que de acuerdo al estudio se destaca el uso de patrones y brinda información a los diseñadores de lenguajes sobre la aplicabilidad al software existente. Como

trabajo complementario recomiendan explorar variaciones de patrones, por ejemplo, permitir métodos de entrada de clases concretas, lo que requiere analizar instancias y determinar métodos predeterminados adecuados de clases concretas. En la Tabla 1 se muestra una comparativa de los artículos relacionados al trabajo de esta investigación.

Tabla 1. Comparativa de artículos relacionados.

Articulo	Objetivo	Enfoque	Métricas	Patrones de diseño	Producto resultante
Using Modularity Metrics to assist Move Method Refactoring of Large Systems (Napoli et al., 2013)	Mejorar la modularidad de los sistemas de software a nivel de clases y métodos y reducir el tiempo de cálculo de métricas en grandes sistemas de software.	Crea un método que muestra sugerencias de re-factorización al calcular algunas métricas y se crea un algoritmo paralelo que se ejecuta en una GPU para calcular métricas de sistemas de software grandes que consumen mucho tiempo.	Fan-in, Fan-out, CBO Y LCOM	N/A	Método de sugerencias para re-factorizar y un algoritmo para reducción de tiempo de ejecución de métricas.
Automated refactoring of super-class method invocations to the Template Method design pattern (Zafeiris et al., 2017)	Reducir el acoplamiento de las subclases con la clase base en la herencia de implementación que dificulta su evolución y mantenimiento.	Sustituir la herencia de implementación por la herencia de interfaz mediante la re-factorización automática de invocación a métodos de clases base (*call super*), utilizando el patrón de diseño *"Template Method"* (método de la plantilla).	N/A	Template method	Implementación de un método para re-factorización automática.
AutoRefactoring: A platform to build refactoring agents (Dos Santos Neto et al., 2015)	Mejorar la calidad de software en cuanto a flexibilidad, reusabilidad, efectividad y extendibilidad.	Aplica los métodos de re-factorización "Encapsulate Fields" y "Extract Class" mediante una plataforma basada en agentes, llamada plataforma AutoRefactoring que resuelve problemas de ocultamiento de datos y por tener campos similares en diferentes clases que son originados por el código desagradable: Public Fields y	Métricas propietarias del trabajo: reusabilidad, flexibilidad, extensibilidad y efectividad	N/A	Implementación de un agente para re-factorización automática.

		Data Clumps presentados en algunos sistemas de software.			
Towards Improving Interface Modularity in Legacy Java Software through Automated Refactoring (R. T. Khatchadourian et al., 2016)	Mejorar el modularidad en la dimensión de abstracción.	Crea una herramienta de re-factorización automatizada que aumenta la modularidad de la interfaz de Java mediante la migración automática de las definiciones de los métodos en la clases a las interfaces de Java 8 como “default method”.	N/A	Skeletal implementation	Implementación del método de re-factorización.
Modularity-Oriented Refactoring (Bryton & Brito e Abreu, 2008)	Mostrar bases cuantitativas para detectar código desagradable en los sistemas de software.	Utiliza un método de re-factorización denominado MORe para remover el código desagradable existente en los sistemas de software a través de paradigmas basada en métricas de modularidad independientes de paradigma y formalismo. El método MORe consiste en una secuencia de 7 pasos, en cada uno se produce un resultado.	Normativas orientadas a modularidad	N/A	La herramienta MORe que implementa al método de re-factorización.
Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering (Rathee & Chhabra, 2018)	Mejorar la cohesión los elementos de software que conforman el diseño estructural de sistemas de software para aumentar su calidad.	Proponer un método que consta de 3 pasos: Extraer el patrón de uso frecuente, calcular la cohesión usando la métrica propuesta, finalmente, se utiliza el valor de cohesión calculado para agrupar elementos en elementos más cohesivos y una nueva métrica de cohesión para el software orientado a objetos calculada a nivel de modulo (UPBC).	Métrica propietaria para la medición de la cohesión UPBC	N/A	La métrica UPBC y un método implementado de re-factorización.

Performance-driven software model refactoring (Arcelli et al., 2018)	La satisfacción de los requisitos de rendimiento de los clientes en cuanto al tiempo de ejecución de los requerimientos del cliente.	Utiliza la plataforma EPSILON para identificar acciones de re-factorización en modelos UML para un conjunto de antipatrones de rendimiento	N/A	N/A	Un método implementado de re-factorización de modelos que mejora el rendimiento.
Automated Refactoring of Legacy Java Software to Default Methods (R. Khatchadourian & Masuhara, 2017)	Mejorar la estructura del código de java heredado.	Re-factorización para la migración a los métodos predeterminados de la interfaz mejorada de Java 8. Se abren las clases a la herencia permitiendo que las clases hereden múltiples definiciones de interfaz.	N/A	Skeletal implementation	Plug-in de código abierto para el IDE de Eclipse
Tesis	Mejorar el diseño de arquitecturas de componentes de software existentes en lenguaje Java mediante la reducción de las dependencias entre clases de marcos de aplicaciones y mejora de la modularidad en suficiencia y completitud.	Se utiliza el patrón de diseño Mediator para reducir el acoplamiento de clases en marcos de aplicaciones orientadas a objetos. La clase mediadora gestiona la comunicación entre las clases que colaboran.	COF LCOM Coherencia FR	Mediator	Métodos de re-factorización y su implementación integrado al SR2-Refactoring para reducir la dependencia entre clases y mejorar la modularidad en suficiencia y completitud.

Capítulo 3. Marco Conceptual

3.1 Paradigma de programación orientada a objetos

3.1.1 Clase

Una clase es una plantilla que permite representar un conjunto de objetos con características y relaciones comunes. Por ejemplo, en la Figura 1 se representa la clase Automóvil, esta es una clase de objetos caracterizados por tener motor, modelo, etc. Un Ibiza o un Jetta son objetos particulares que se obtienen instanciando la clase automóvil para esos casos particulares (Joyanes Aguilar, 2008). Esencialmente, una clase es un plan que especifica cómo construir objetos**,** en ella se definen los atributos, que denotan el estado implícito y el estado explícito; los métodos que denotan el comportamiento con los cuales los objetos responden a mensajes externos; así como mensajes comunes a todos los objetos que son instancias de la clase o tipo.

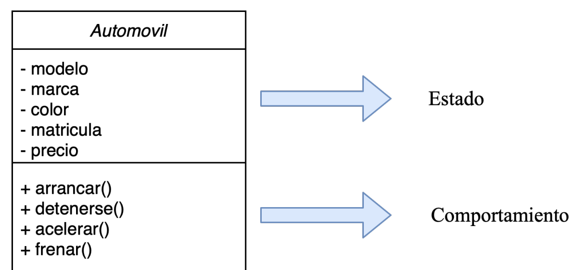


Figura 1. Modelo conceptual de una clase.

3.1.2 Objeto

El objeto es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intenta descubrir e implementar los objetos que juegan un rol en el dominio de un problema, y en consecuencia programa. Un objeto es cualquier cosa tangible o intangible que se pueda imaginar, definida frente al exterior mediante unos atributos y las operaciones que permiten modificar dichos atributos. Un objeto automóvil se ensambla de partes como, el motor, carrocería, puertas o puede ser descrito utilizando propiedades como la velocidad, el kilometraje o el fabricante. Estos atributos indican el objeto. De modo similar, una persona también se puede ver como un objeto, del cual se dispone de diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc. (Joyanes Aguilar, 2008). Un objeto tiene identidad propia, encapsula su propio estado y su propio comportamiento.

3.1.3 Representación interna de una clase

3.1.3.1 Atributos

Los atributos son las características individuales que diferencian a un objeto de otro(s) y determinan su apariencia, estado u otras cualidades. Los atributos representan variables, y cada objeto particular puede tener valores distintos para estas variables. El estado de un objeto puede ser implícito o explícito. El estado implícito lo definen los valores que asumen cada uno de los atributos en cierto momento. El estado explícito lo definen variables que, explícitamente, denotan un cierto estado del objeto a través de la ejecución de uno o varios casos de uso.

Los atributos o características de los objetos son, por ejemplo: en las personas, su edad, su profesión, su domicilio, etc.; en un auto, la potencia, el número de matrícula, el precio, número de puertas, etc.; en una casa, la superficie, el precio, el año de construcción, la dirección, etc. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; tienen un valor específico, tal como 200 metros cuadrados, 20.000 dólares, cinco puertas, etc. (Joyanes Aguilar, 2008).

3.1.3.2 Métodos

El comportamiento de una clase se define con la creación de métodos funcionales. El comportamiento son las acciones que ejecutan los objetos del mundo real como respuesta a estímulos determinados. En programación las acciones se ejecutan como respuesta a mensajes o peticiones de servicio desde agentes externos. El conjunto de los métodos de un objeto determina el comportamiento general de los objetos. Por ejemplo, si se pisan los frenos en un auto, el auto se detiene; si acelera, el auto aumenta su velocidad, entre otros. El comportamiento, en esencia, es una función: se llama a una función para hacer algo (por ejemplo, visualizar la nómina de los empleados de una empresa) (Joyanes Aguilar, 2008).

3.1.3.3. Mensaje

El mensaje es el fundamento de una relación de comunicación que enlaza dinámicamente los objetos que fueron separados en el proceso de descomposición de un módulo. En la práctica, un mensaje es una comunicación entre objetos, en los que un objeto (cliente) solicita a otro objeto (proveedor o servidor) hacer o ejecutar alguna acción (Joyanes Aguilar, 2008). En respuesta al mensaje, el receptor se comportará de una determinada forma. Cada mensaje consta de tres partes:

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

En la Figura 2, se puede observar un ejemplo de envío de un mensaje, en donde un objeto de clase Cliente solicita un servicio al objeto "*op*" que es de clase Operaciones para realizar la suma de los números 4 y 3.

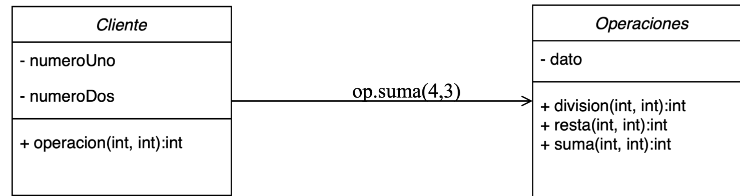


Figura 2. Envío de mensaje de un objeto de la clase Cliente a un objeto "op" de la clase Operaciones para realizar la suma de dos números.

3.2 Modularidad

Modularidad, es la propiedad que permite subdividir una aplicación en partes más pequeñas llamadas módulos, los cuales deben ser autónomos, tanto como sea posible, de la aplicación en sí y de las partes restantes (Meyer, 1997). La modularidad se aplica tanto a la especificación del diseño como a la implementación.

3.2.1 Completitud

Grado en el que los datos asociados con una entidad tienen valores para todos los atributos esperados relacionadas en un contexto de uso específico (Calabrese et al., 2019). Se encuentra en función del grado de relación que se tiene entre las funciones (coherencia) y las funciones con los atributos (cohesión).

3.2.1.1 Cohesión

El término cohesión tiene su origen en el diseño estructural y se refiere a la relación entre los diversos elementos de un módulo dado. Es un indicador importante de la calidad del diseño de software y su modularidad (Rathee & Chhabra, 2018). Un modelo cuyas partes están fuertemente relacionadas con cada uno de los otros se dice que es fuertemente cohesivo. Un modelo cuyas partes no están relacionadas con otras se dice que es débilmente cohesivo (Joyanes Aguilar, 2008).

Cohesión de clase

En el caso de una clase, esta se encuentra compuesta de atributos y métodos por lo tanto estos son los elementos que componen el módulo (clase). Una clase altamente cohesiva indicaría una fuerte relación entre sus métodos debido a que comparten los mismos atributos. Por otro lado, una clase débilmente cohesiva (baja cohesión) indica que los atributos y métodos no se encuentran fuertemente relacionados (C. Martin, 2012).

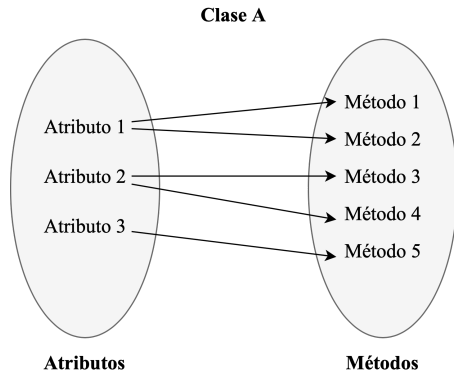


Figura 3. Ejemplo de una clase con baja cohesión.

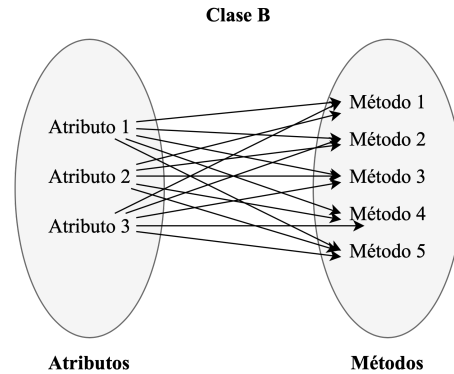


Figura 4. Ejemplo de una clase con alta cohesión.

Supóngase que se tiene una clase A (Figura 3), la cual cuenta con tres atributos y cinco métodos, se observa que los atributos no están relacionados completamente con todos los métodos. Es decir, que no todos los métodos ocupan todas las variables, por lo tanto, se puede decir que la clase A es débilmente cohesiva (baja cohesión). Por otro lado, se observa en la clase B (Figura 4) que todos los métodos hacen uso de todas las variables por lo que se dice que la clase B es altamente cohesiva, debido a que todos los métodos están relacionados porque comparten las mismas variables.

3.2.1.2 Coherencia

Para fines de la investigación de esta tesis, se conceptualiza a la coherencia de clase como el grado de relación funcional de una responsabilidad en la clase, es decir la cantidad de funciones que interactúan para satisfacer una responsabilidad en relación al número total de funciones en la clase que los contiene, y se refiere a una secuencia interactiva de métodos implicados para cumplir con un caso de uso (responsabilidad o meta de valor).

3.2.1.3 Responsabilidad de clase

Este concepto surge directamente del principio de una única responsabilidad conocido como *"Single Responsibility Principle"* (SRP), el cual indica que: "una clase o módulo debe tener uno y sólo un motivo para cambiar" (C. Martin & Martin, 2006). El número de secuencias de métodos iniciadas por un método *"public"* existentes en una clase, indican el número de metas de valor o motivos para cambiar que contiene la clase. En caso de existir más de una secuencia de métodos en la clase indicaría que se está rompiendo el principio de única responsabilidad.

La Figura 5 muestra dos secuencias encontradas en la clase A. Esto indica que la clase A tiene dos responsabilidades. Mientras que la Figura 6 muestra la clase B en la cual se cuenta con sólo una secuencia de métodos (esto es, una única responsabilidad).

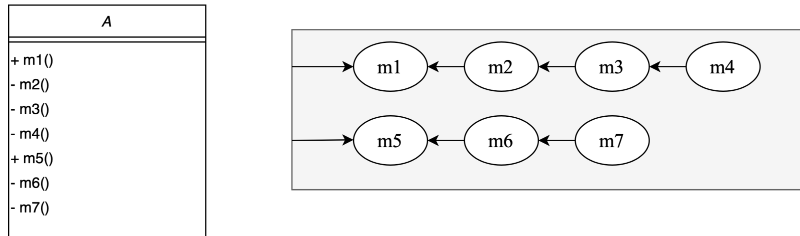


Figura 5. Secuencias de la clase A.

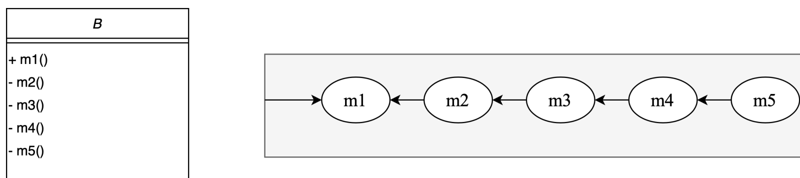


Figura 6. Secuencias de la clase A.

3.2.2 Suficiencia

Los Módulos, Componentes, Paquetes, Clases, o Servicios, que son suficientes se definen como aquellas unidades de software que contienen toda la información y la funcionalidad, necesarias (ni más- ni menos) para realizar un objetivo o meta de valor para un usuario, sin la necesidad de requerir de otros módulos o unidades de programa para poder llevar a cabo las tareas.

La suficiencia se encuentra en función del grado de dependencias (acoplamiento) existente en las clases de las arquitecturas de software y en función de que existan pocas interfaces en los módulos (Padilla Salgado, 2019).

3.2.2.1 Pocas interfaces

El concepto de pocas interfaces se refiere a que cada módulo debe comunicarse con el menor número posible de otros módulos. La regla de pocas interfaces restringe el número total de canales de comunicación entre módulos en una arquitectura de software. La comunicación puede ocurrir entre módulos de diversas formas. Los módulos pueden llamarse entre sí, o compartir estructuras de datos, entre otros. La regla de pocas interfaces limita el número de tales conexiones.

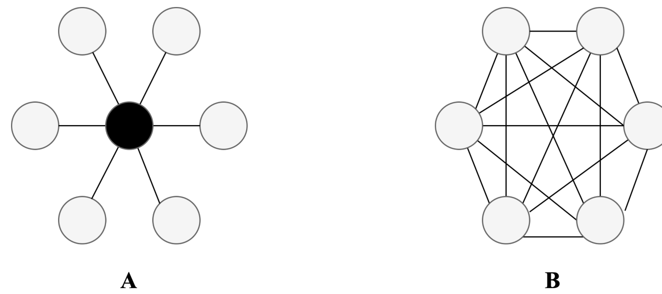


Figura 7. Comunicación entre módulos.

Si un sistema está compuesto por ´n´ módulos, entonces el número de conexiones entre módulos debe permanecer mucho más cerca del mínimo, n – 1, mostrado como A en la Figura 7, que, del máximo, n (n - 1) / 2, mostrado como B en la misma figura.

Esta regla se deriva en particular de los criterios de continuidad y protección (Meyer, 1997): es decir, si hay demasiadas relaciones entre módulos, entonces el efecto de un cambio o de un error este puede propagarse a un gran número de módulos. También está relacionado con la composibilidad (si desea que un módulo sea utilizable por sí mismo en un nuevo entorno, entonces no debería depender de muchos otros), la comprensibilidad y la descomposibilidad (Meyer, 1997).

3.2.2.2 Dependencia

La relación más débil que puede existir entre dos clases es una relación de *dependencia* (Vázquez Escudero et al., 2001). Una dependencia entre clases significa que una clase utiliza, o tiene conocimiento de otra clase, o dicho de otro modo “lo que una clase necesita conocer de otra clase para utilizar objetos de esa clase” y así cumplir con su meta de valor u objetivo.

3.2.2.3 Acoplamiento

El acoplamiento se refiere al grado de dependencia entre módulos. El grado de acoplamiento se puede utilizar para evaluar la calidad del diseño de un sistema. Es preciso minimizar el acoplamiento entre módulos, esto es, minimizar su interdependencia. El criterio de acoplamiento es una medida para evaluar cómo un sistema ha sido modularizado. Este criterio sugiere que un sistema bien modularizado es aquel en que los interfaces sean claras y sencillas (Joyanes Aguilar, 2008). Es decir, el modo en que un módulo está siendo afectado por la estructura interna de otro módulo

3.3 Re-factorización (*Refactoring*)

La re-factorización es el proceso de cambiar un sistema de software de tal manera que mejore su estructura interna, pero que no se altere el comportamiento externo del código. Es una forma disciplinada de limpiar el código para minimizar las posibilidades de introducir errores. En esencia, cuando se refactoriza, se está mejorando el diseño del código después de que ha sido escrito.

Un método de re-factorización puede tomar un mal diseño, incluso el caos, y transformarlo en un código mejor diseñado. Por ejemplo, mover un campo de una clase a otra clase, extraer algo de código de un método para convertirlo en su propio método y mover algún código hacia arriba o hacia abajo en una jerarquía de herencia. Sin embargo, el efecto acumulativo de estos pequeños cambios puede mejorar radicalmente el diseño (Fields et al., 2010; Fowler Martin, 1994).

3.4 Código desagradable (*Smell Code*)

El código desagradable engloba malas opciones de diseño y/o implementación, opuestas a los modismos y, en cierta medida, a los patrones de diseño, en el sentido de que pertenecen a la implementación, mientras que los patrones de diseño pertenecen al diseño. Son la implementación "pobre" de soluciones a problemas recurrentes. En la práctica, el código desagradable se encuentra entre el diseño y la implementación: pueden referirse al diseño de una clase, pero concretamente, se manifiestan en el código fuente como clases con implementación específica (Khomh et al., 2009). Por lo general, se revelan a través de valores métricos particulares (Marinescu, 2004).

3.5 Métrica de calidad

Una métrica de software es una forma estándar de medir algunos atributos, tanto del producto como del proceso de desarrollo de software. Los factores de calidad son normalmente atributos de un alto nivel de abstracción como "fiabilidad", "facilidad de uso", "facilidad de mantenimiento", etc.

El modelo de calidad del software de McCall define tres aspectos o características importantes de un producto: características operacionales, de modificación y de transición, descomponiendo cada una en factores de calidad de alto nivel que determinan la calidad en cada una de ellas. Estos son a su vez descompuestos en criterios de calidad. Por último, estos son asociados a un conjunto de atributos directamente mensurables denominados métricas de calidad. Esto permite definir unos atributos en términos de otros más fáciles de medir.

3.5.1 COF (Coupling Factor)

Es la proporción entre el número real de acoplamientos y el máximo número posible de acoplamientos, no imputables a herencia, en el sistema. Es decir, indica la comunicación entre clases (Rodríguez & Harrison, n.d.; Vázquez Escudero et al., 2001). La descripción completa y la formula se describen en el Capítulo 4.

3.5.2 LCOM (Lack of Cohesion in Methods)

Indica la calidad de la abstracción hecha en la clase. Usa el concepto de grado de similitud de métodos. Si no hay atributos comunes, el grado de similitud es cero (Rodríguez & Harrison, n.d.). Una baja cohesión incrementa la complejidad y por tanto la facilidad de cometer errores durante el proceso de desarrollo (Rodríguez & Harrison, n.d.; Vázquez Escudero et al., 2001). La descripción completa y la formula se describen en el Capítulo 4.

3.5.3 CrCU (Coherencia de caso de uso)

Esta métrica mide la cantidad de funciones que interactúan para satisfacer una responsabilidad en relación al número total de funciones del caso de uso que los contiene, y se refiere a una secuencia interactiva de métodos implicados para cumplir con un caso de uso (Gallardo Vera, 2018).

3.6 Flexibilidad

Es la facilidad con la que un sistema o componente puede modificar su comportamiento para su uso en aplicaciones o entornos distintos de aquellos para los que fue diseñado específicamente (Dos Santos Neto et al., 2015), es decir, sin que se afecte la arquitectura original, lo que satisface el criterio de continuidad modular.

3.7 Extensibilidad

Es la facilidad con la que un sistema o componente puede aumentar su capacidad funcional o de almacenamiento (Dos Santos Neto et al., 2015) sin modificar su definición actual, es decir, sin que se afecte la arquitectura original, lo que satisface el criterio de continuidad modular.

3.8 ANTLR

ANTLR (ANother Tool for Language Recognition) es un generador de analizadores para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. Este es utilizado para

construir lenguajes, herramientas y marcos. A partir de una gramática, ANTLR genera un analizador que puede construir y recorrer árboles de análisis sintáctico. (Terence, 2013).

3.9 Patrón de diseño

Un patrón de diseño describe un problema, el cuál ocurre una y otra vez en un entorno, y luego describe el núcleo de la solución a ese problema, de tal manera que se puede utilizar esta misma solución un millón de veces, sin tener que hacer lo mismo dos veces (Gamma et al., 2005). En general, en lenguaje alexandriano, un patrón tiene cuatro elementos esenciales:

1. **El nombre del patrón** es un identificador, de una palabra o dos, que se usa para describir un problema de diseño, su solución y consecuencias en una palabra o dos. Nombrar un patrón aumenta inmediatamente el vocabulario. Permite el diseño a un mayor nivel de abstracción.

2. **El problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Podría describir problemas de diseño específicos, como lo es representar algoritmos como objetos. Podría describir estructuras de clase u objeto 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 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 nuestro caso), que se puede ajustar a la medida de un contexto particular o específico.

4. **Las consecuencias** son los resultados y las compensaciones de aplicar el patrón. Aunque las consecuencias a menudo pasan desapercibidas cuando describimos las decisiones de diseño, son fundamentales para evaluar las alternativas de diseño y para comprender los costos y beneficios de aplicar el patrón.

3.9.1 Patrón de diseño *"Mediator"*

Es un patrón de diseño de comportamiento que permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador (Gamma et al., 2005). A continuación, se describe el patrón de diseño *"Mediator"*:

Problema

El diseño orientado a objetos fomenta la distribución del comportamiento entre los objetos. Tal distribución puede resultar en una estructura de objetos con muchas conexiones entre estos. En el peor de los casos, todos los objetos están enterados de todos los demás.

Aunque dividir un sistema en muchos objetos generalmente mejora la reutilización, la proliferación de interconexiones tiende a reducirlo nuevamente. Muchas interconexiones disminuyen la probabilidad que un objeto pueda funcionar sin el apoyo de otros; el sistema actúa como si fuera monolítico. Además, puede ser difícil cambiar el comportamiento del sistema de manera significativa, ya que el comportamiento se distribuye entre muchos objetos. Como resultado, uno puede verse obligado a definir muchas subclases para personalizar el comportamiento del sistema (Gamma et al., 2005).

Solución

El patrón Mediator sugiere detener la comunicación directa entre los componentes que se quiere sean independientes entre sí. En lugar de ello, estos componentes colaboraran indirectamente, invocando un objeto mediador especial que realice las llamadas a los componentes adecuados. Como resultado, los componentes dependen únicamente de una sola clase mediadora, en lugar de estar acoplados a muchos de sus colegas (Gamma et al., 2005).

Consecuencias

El patrón mediator tiene los siguientes beneficios e inconvenientes:

- Limita el subclaseo.
- Desacopla a los colegas (clases que interactúan).
- Simplifica el protocolo entre objetos interactuantes.

Capítulo 4. Materiales y métodos de solución

En este trabajo de tesis se realizó una investigación en busca de métricas cuyo propósito fuera medir la modularidad en las dimensiones de suficiencia y completitud de arquitecturas de software orientadas a objetos. En la literatura no se reporta la existencia de métricas que midan la modularidad en esos niveles. Por lo que en este trabajo se utilizaron tres métricas utilizadas en anteriores investigaciones del CENIDET. Estas son: Factor de acoplamiento (COF), Carencia de cohesión (LCOM*) y Coherencia de Caso de Uso (CrCU). Además, en esta tesis, se propone una nueva métrica para medir el promedio de responsabilidades por clase (FR).

Para propósito de mostrar que se cumple con el objetivo de la investigación, que es la mejora de la modularidad y reducción de dependencias entre clases de objetos, a continuación, se presenta una descripción detallada de cada una de las métricas utilizadas, para medir la calidad modular de arquitecturas orientadas a objetos antes y después de haberles aplicado los métodos de re-factorización creados.

4.1 Métrica COF (Factor de acoplamiento)

Esta métrica calcula el factor de acoplamiento de un sistema debido a múltiples relaciones de dependencia entre clases, incluyendo las llamadas a funciones, la creación de objetos y destrucción de objetos. Para las pruebas de esta tesis, sólo se consideraron las llamadas a métodos o funciones que ocurren en primera instancia, es decir, llamadas a funciones de otras clases de manera directa o a través de clases abstractas, excepto mensajes entre clases hermanas o entre clases base con sus clases derivadas (Brito e Abreu et al., 1995; Vázquez Escudero et al., 2001).

La expresión matemática se muestra a continuación:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} es_cliente(C_i, C_j)\right]}{TC^2 - TC} \quad (1)$$

Donde:
TC es el total de clases
TC^2 - TC es el máximo número de acoplamientos en un sistema con TC clases.

$$es_cliente(Cc,Cs) = \begin{cases} 1 \text{ si y solo si } Cc \Rightarrow Cs \ \wedge Cc \neq Cs \\ 0 \text{ en caso contrario} \end{cases}$$

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

Valoración:

Los diseñadores de software deberían evitar valores muy altos de COF. Sin embargo, para una aplicación determinada, las clases deben cooperar de alguna manera para ofrecer algún tipo de funcionalidad. Por lo tanto, se espera que el valor de COF sea bajo lo más bajo posible. Como resultado de experimentación en (Brito e Abreu et al., 1995) y (Aivosto, n.d.) (Aivosto, n.d.) se sugiere que el factor de acoplamiento (COF) en un sistema no exceda de 12% o un 17.7% respectivamente.

El acoplamiento que se mide con esta métrica, en esta investigación es aquel que se da por la comunicación entre la clase cliente y la clase servidora al invocar un método (llamada a método). Es decir, no se toman en cuenta otras relaciones, por ejemplo, la declaración de objetos. No se toma en cuenta la cantidad de métodos de una clase que son invocados por otra clase, sino que todos cuentan como una sola dependencia, lo único que importa es contar las clases que tienen dependencias con otras clases en particular.

4.2 Métrica LCOM* (Carencia de cohesión)

LCOM es una medida de la cohesión de una clase, midiendo el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase. Un valor alto de LCOM implica falta de cohesión, es decir, escasa relación entre los métodos de una clase. Esto puede indicar que la clase está compuesta de elementos no relacionados, incrementando la complejidad y la probabilidad de errores durante el desarrollo. Es deseable una alta cohesión en los métodos dentro de una clase ya que esto fomenta la encapsulación (Jain & Gupta, 2015; Rodríguez & Harrison, n.d.).

La expresión matemática es la siguiente:

$$\mathrm{LCOM}* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m} \tag{2}$$

Donde
a es el número de atributos.
m el número total de métodos.
$\mu(A_j)$ es el número de métodos que acceden al atributo A_j

Esta métrica sólo puede calcularse cuando $m > 1$.

Cuando todos los métodos acceden a todos los atributos, entonces Σμ(Aj) = ma (todos los métodos acceden a todos los atributos), y por lo tanto LCOM* = 0. Esto indica una cohesión perfecta. Valores cercanos a 0 indican que la mayoría de los métodos accede a la mayoría de las variables de instancia. Por el contrario, cuando cada método accede sólo a un atributo, entonces Σ μ(Aj) = a y, por lo tanto, LCOM* = 1, lo cual indica una carencia total de cohesión.

4.3 Métrica CrCU(Coherencia de Caso de Uso)

Esta métrica mide la cantidad de métodos que interactúan para satisfacer una responsabilidad en relación al número total de métodos del módulo que los contiene, y se refiere a una secuencia interactiva de métodos implicados para cumplir con un caso de uso (Gallardo Vera, 2018).

$$CrCU = \frac{n}{m} \tag{3}$$

Donde:
n = Total de métodos en la secuencia.
m = Total de métodos del módulo o paquete.

4.4 Diseño de la métrica FR (Factor de responsabilidades)

De acuerdo al principio de única responsabilidad descrito en el apartado de marco conceptual se propone la siguiente fórmula para la métrica que permite obtener el promedio de responsabilidades por clase.

$$FR = \frac{1}{\frac{\sum_{i=1}^{NC} Resp_i}{NC}} \tag{4}$$

Donde:

FR = Factor de responsabilidades
NC = Número total de clases concretas en la arquitectura
Resp = Responsabilidad

La métrica fue normalizada con el objetivo de que los valores obtenidos se encuentren dentro del rango del 0 al 1. A medida que esta razón tienda a 0 indica que en promedio hay más de una responsabilidad por clase. Una medida que tienda al 1 indica que en promedio hay una responsabilidad por clase. Este es el valor deseable de acuerdo al principio de única responsabilidad. La métrica fue sustentada en la teoría de la medición y se hicieron tres pruebas

con arquitecturas diferentes. El sustento teórico completo del diseño de esta métrica se encuentra descrito a detalle en el Anexo A.

4.5 Proceso de cada uno de los métodos de re-factorización

Los dos métodos de re-factorización para mejorar de la modularidad y reducir las dependencias se encuentran conformados por varios procesos importantes que describen el funcionamiento de estos. En estos dos métodos de re-factorización se encuentran tres procesos importantes: A) Analizar código, B) Calcular métricas y C) Generar código.

En el proceso de Analizar código se realiza el análisis sintáctico del código de entrada para obtener la información necesaria para realizar el cálculo de métricas y para el proceso de re-factorización del código. Esta información es retenida en memoria mediante una lista enlazada compleja.

Para el cálculo de métricas se aplican las fórmulas de las métricas para obtener el resultado de estas antes y después de la re-factorización. Con respecto al proceso de Generar código, como entrada se tiene la información de la estructura de datos de la lista enlazada, la cual fue modificada con información de las clases resultante de la aplicación del algoritmo de re-factorización, con esta información se llenan las plantillas *String Template* y se generan los archivos que conformarán el proyecto refactorizado.

4.5.1 Proceso de re-factorización del método para mejorar la modularidad

El proceso en la (Figura 8) ilustra el método para mejorar la modularidad. Este se encuentra dividido en dos partes: a) la identificación de responsabilidades y b) las relaciones entre métodos y atributos existentes en cada clase. A continuación, se describen los pasos que conforman el algoritmo que implementa este método de re-factorización:

1. Por cada clase que conforman la arquitectura del código del sistema de origen:
    a. Obtener el número de responsabilidades (número de secuencias de métodos).
    b. Separar las responsabilidades, a través de la creación de nuevas clases, de forma que cada clase sólo contenga una única responsabilidad. Estas clases recién creadas se integran a la estructura de la lista enlazada. En caso de que sólo hubiera una responsabilidad, la clase correspondiente no es dividida.
2. Por cada clase que conforma la lista creada en el paso 1 b)
    a. Obtener el número de relaciones entre atributos y métodos.
    b. Separar en subconjuntos los métodos y datos relacionados. Para cada subconjunto, crear una nueva clase de objetos e integrar los elementos de ese subconjunto de manera apropiada. En caso de que todos los métodos hagan uso

de todas las variables, esta clase no es dividida y conserva los elementos de origen.

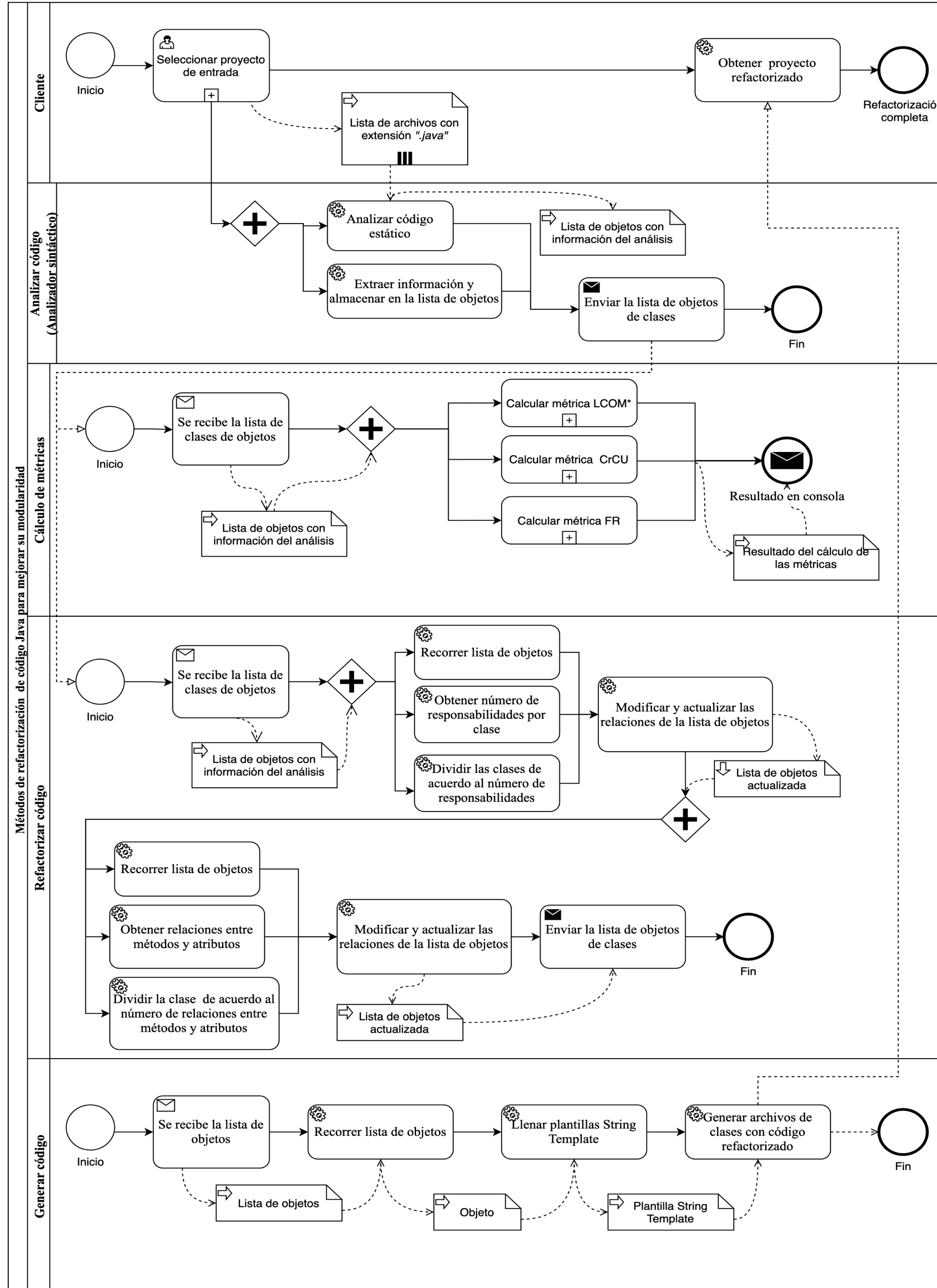


Figura 8.Diagrama BPMN correspondiente a la re-factorización de código del algoritmo del método para mejorar la modularidad

4.5.2 Proceso de re-factorización para reducir las dependencias entre clases de objetos

Este proceso consiste en la creación de un algoritmo que permita la implementación del patrón de diseño "*Mediator*" en el código existente. Con el uso de este patrón de diseño se realiza la reducción de las dependencias entre clases del sistema original, ver (Figura 9).

A continuación, se describen los pasos para la introducción del patrón de diseño "*Mediator*":

1. Por cada conjunto de clases colega
   1.1 Generar una clase colega abstracta.
      a. Establecer su nombre como *"ColegaAbstracto"*
      b. Agregar los métodos participantes en la comunicación sin cuerpo, es decir, como métodos abstractos.
   1.2 Generar una clase mediadora
      a. Establecer su nombre como *"Mediador"*
      b. Agregar los métodos donde se realizarán las llamadas hacia el mediador y que participan en la comunicación.
      c. Agregar como parámetro a estos métodos un objeto de la clase colega a la cual pertenece cada método y establecerlos como métodos abstractos.
   1.3 Generar la clase mediadora concreta
      a. Establecer su nombre como *"MediadorConcreto"*
      b. Agregar los métodos que participan en la comunicación
      c. Agregar la implementación de las llamadas desde el mediador hacia las clases colega de la siguiente forma:
      objColex = new Clase();
      objColex = objeClase;
      objColex.métodos(parámetros);
   1.4 Modificar las clases colegas
      a. Si las clases *"colega"* no derivan de alguna clase base o son abstractas, entonces se establece la derivación de la clase colega abstracta creada en el punto 1.1. Si heredan de otra clase se establece la derivación de la clase de la cual heredan.
      b. Crear un objeto del tipo de la clase *"Mediador"* y otro objeto de la clase *"Colega Abstracto"*.
      c. Modificar las llamadas de los métodos de las clases colegas que participan en la comunicación. El objeto de la clase mediadora hará la llamada del método a ejecutar y se enviará como parámetro el objeto del tipo de la clase *"Colega Abstracto"*.

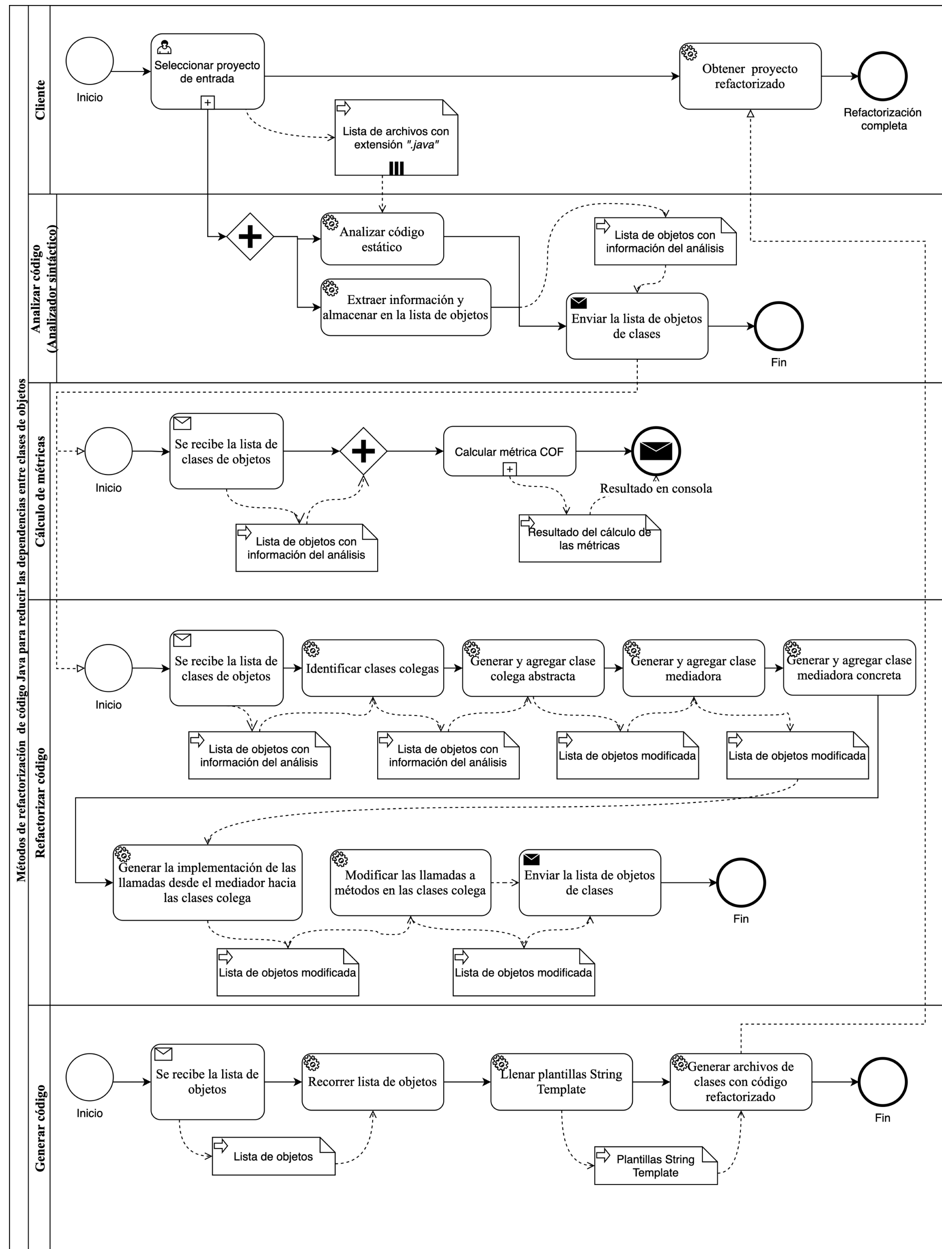


Figura 9. Diagrama BPMN correspondiente a la re-factorización de código del algoritmo del método para reducir las dependencias entre clases de objetos.

En el capítulo siguiente se describe la implementación de los métodos de re-factorización, conforme con la descripción de los procesos descritos en este capítulo.

Capítulo 5. Desarrollo del sistema

En este capítulo se describe tanto el análisis como el desarrollo realizado para la creación de los dos métodos de re-factorización, a) Método para mejorar la modularidad y b) Método para la reducción de dependencias entre clases de objetos. Se utilizó el lenguaje unificado de modelado (UML) para la construcción de diagramas que describen el proceso detalladamente.

5.1 Análisis de casos de uso

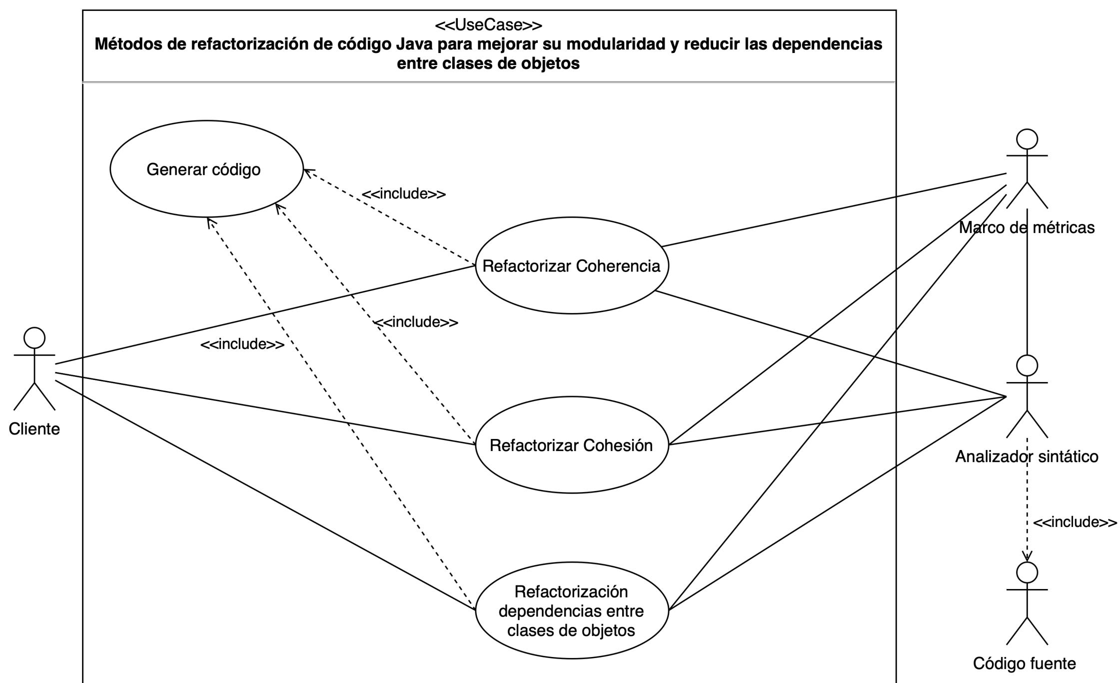


Figura 10. Diagrama de caso de uso general

Cliente

Representa una entidad física (persona) o una entidad lógica (programa) que utiliza el sistema de re-factorización. En la descripción de los diagramas cuando se hable de un una entidad física se le denominará "Usuario" y cuando se refiera a un cliente lógico se denominará "Cliente".

Código Fuente

Representa el código original, contenido en los archivos con extensión Java. Es el código fuente que se desea analizar tanto para el cálculo de métricas como para el proceso de re-factorización.

Analizador sintáctico

Representa el subsistema correspondiente al proceso de análisis sintáctico. Este recibe como entrada un proyecto de software escrito en lenguaje Java y como salida devuelve la estructura de datos con la información necesaria ya sea para el cálculo de métricas o para la re-factorización de código.

Marco de métricas

Representa el subsistema para el cálculo de métricas. De este subsistema se hace uso de las métricas correspondientes a la medición de modularidad (CrCU, LCOM*, FR) y a la medición del nivel de acoplamiento (COF). A continuación, se describen cada uno de los casos de uso mostrados en el diagrama anterior de la Figura 10.

Tabla 2. Descripción de caso de uso "Re-factorizar coherencia"

ID:	**CU-1**
Nombre del caso de uso:	Re-factorizar Coherencia
Actores	Usuario, Analizador sintáctico y Marco de métricas
Descripción	Consiste en la división de clases de acuerdo a las responsabilidades detectadas
Precondiciones	• Contar con la información necesaria para realizar el proceso de re-factorización, contenida en una estructura de datos contenedora, como una lista enlazada. • El cálculo de métricas debería haberse efectuado antes del proceso de re-factorización.
Excepciones:	
Postcondiciones	• Se obtiene una lista que incluye información de las nuevas clases creadas, que se integrarán al código refactorizado del proyecto original. • El sistema activa la aplicación de métricas para evaluar la mejora en la modularidad de la arquitectura resultante después del proceso de re-factorización.

Escenario principal de éxito:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El sistema ejecuta el subsistema “Analizador sintáctico” para realizar el análisis del código de entrada 3. El sistema recibe la lista enlazada obtenida del subsistema “Analizador sintáctico”, contenida en una estructura contenedora de datos (lista enlazada) 4. El sistema analiza la información contenida en la lista enlazada para identificar qué clases contienen más de una responsabilidad. 5. El sistema realiza la división de las clases de acuerdo a las responsabilidades contenidas en cada una 6. El sistema sustituye las clases que se dividieron y agrega las clases creadas de la división de acuerdo a las responsabilidades. 7. El sistema actualiza las relaciones existentes en otras clases de las clases divididas, por ejemplo, llamadas a métodos, tipos de datos, entre otras 8. El sistema genera código refactorizado (CU-4)
Escenario de fracaso 1:	1. El Usuario selecciona incorrectamente la ruta del proyecto de software de entrada 2. El sistema concluye su funcionamiento y se envía un mensaje indicando que la re-factorización no se llevó a cabo.
Escenario de fracaso 2:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El código del proyecto de software seleccionado no se encuentra escrito en lenguaje Java 3. El sistema concluye el proceso y envía un mensaje, indicando que el código no se encuentra escrito en lenguaje Java

Tabla 3. Descripción de caso de uso “Re-factorizar Cohesión”

ID:	**CU-2**
Nombre del caso de uso:	Re-factorizar Cohesión
Actores	Usuario, Analizador sintáctico y Marco de métricas
Descripción	Este método de re-factorización consiste en la división de clases de acuerdo a la relación entre atributos y métodos de la clase
Precondiciones	• Contar con la información necesaria para realizar el proceso de re-factorización, contenida en una estructura de datos contenedora, como una lista enlazada.

	• El cálculo de métricas debería haberse efectuado antes del proceso de re-factorización.
Excepciones:	
Postcondiciones	• Se obtiene una lista que incluye información de las nuevas clases creadas que se integrarán al código refactorizado del proyecto original. • El sistema activa la aplicación de métricas para evaluar la mejora en la modularidad de la arquitectura resultante después del proceso de re-factorización.
Escenario principal de éxito:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El sistema invoca al subsistema "Analizador sintáctico" para realizar el análisis sintáctico del código de entrada 3. El sistema recibe la lista enlazada obtenida del subsistema "Analizador sintáctico", contenida en una estructura contenedora de datos (lista enlazada) 4. El sistema analiza la información contenida en la lista enlazada para identificar en qué clases no todos los métodos hacen uso de todas los atributos de clase (baja cohesión) 5. El sistema realiza la división de las clases que tienen baja cohesión de manera apropiada, de modo que cada clase creada tenga mayor cohesión. 6. El sistema sustituye las clases que se dividieron y agrega las clases creadas en el paso 5. 7. El sistema actualiza las relaciones existentes en las clases de manera apropiada, por ejemplo, llamadas a métodos, tipos de datos, entre otras 8. El sistema genera una nueva lista de clases con las clases creadas y e integra las clases generadas en el punto 5 con las clases que no fueron divididas. 9. El sistema genera código refactorizado (CU-4).
Escenario de fracaso 1:	1. El Usuario selecciona incorrectamente la ruta del proyecto de entrada 2. El sistema concluye su funcionamiento y se envía un mensaje indicando que la re-factorización no se llevó a cabo
Escenario de fracaso 2:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El código del proyecto de software seleccionado no se encuentra en escrito en lenguaje Java

	3. El sistema concluye el proceso y envía un mensaje, indicando que el código no se encuentra escrito en lenguaje Java

Tabla 4. Descripción de caso de uso dependencias entre clases de objetos

ID:	**CU-3**
Nombre del caso de uso:	Re-factorizar dependencias entre clases de objetos
Actores	Usuario, Analizador sintáctico y Marco de métricas
Descripción	Consiste en la reducción de las relaciones la dependencia entre clases de objetos mediante la implantación del patrón de diseño mediator
Precondiciones	• Contar con la información necesaria contenida en la estructura contenedora de datos, como una lista enlazada. La información es resultado del proceso del subsistema Analizador sintáctico. • El cálculo de métricas debería haberse efectuado antes del proceso de re-factorización
Postcondiciones	• Se obtiene una lista que incluye la información de las nuevas clases creadas que son integradas al código refactorizado del proyecto original. • El sistema activa la aplicación de métricas para evaluar la mejora en la modularidad de la arquitectura resultante después del proceso de re-factorización
Escenario principal de éxito:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El sistema ejecuta el subsistema "Analizador sintáctico" para realizar el análisis del código de entrada 3. El sistema recibe la lista enlazada obtenida del subsistema "Analizador sintáctico", contenida en una estructura contenedora de datos (lista enlazada) 4. El sistema identifica las clases colegas del sistema de software de entrada 5. El sistema genera la clase colega abstracta 6. El sistema genera la clase base mediadora 7. El sistema genera la subclase mediadora concreta, relacionada por herencia a la clase base mediadora 8. El sistema modifica las llamadas originales entre clases colega y las redirige apropiadamente hacia la clase base mediadora

	9. El sistema genera una nueva lista e integra las clases generadas en los puntos 5 a 7 con las clases ya existentes 10. El sistema genera código refactorizado (CU-4).
Escenario de fracaso 1:	1. El Usuario selecciona incorrectamente la ruta del proyecto de entrada 2. El sistema no refactoriza el proyecto deseado.
Escenario de fracaso 2:	1. El Usuario selecciona la ruta del proyecto de software de entrada 2. El código del proyecto de software seleccionado no se encuentra en escrito en lenguaje Java 3. El sistema concluye el proceso y envía un mensaje, indicando que el código no se encuentra escrito en lenguaje Java

Tabla 5. Descripción de caso de uso "Generar código"

ID:	**CU-4**
Nombre del caso de uso:	Generar código
Actores	Marco de métricas
Descripción	Consiste en la generación de código refactorizado con el llenado de plantillas ST para el modelado de las partes que componen un sistema
Precondiciones	• Contar con la lista de la información necesaria para llenar las plantillas ST
Excepciones:	
Postcondiciones	• Se obtiene la carpeta correspondiente al proyecto de software refactorizado • El sistema activa la aplicación de métricas para evaluar la mejora en la modularidad de la arquitectura resultante después del proceso de re-factorización
Escenario principal de éxito:	1. El sistema realiza el llenado de cada una de las partes que componen al sistema refactorizado, por ejemplo, las variables, los métodos, las sentencias, los atributos, las clases, entre otros 2. Se realiza el llenado de la plantillas *"String Template"* 3. El sistema genera las ruta y carpetas correspondientes al proyecto refactorizado 4. El sistema devuelve la ruta donde se ubica el proyecto refactorizado

A continuación, se describe el caso de uso de cada una de las métricas utilizadas en la investigación.

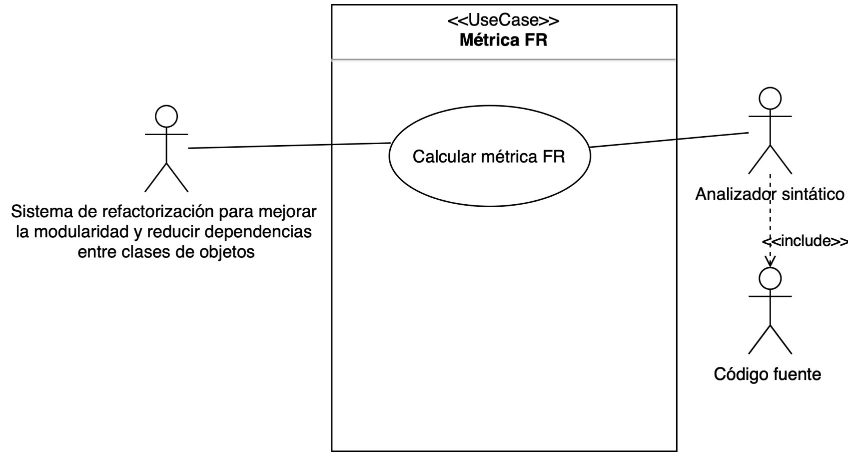


Figura 11. Diagrama de caso de uso métrica FR

Tabla 6. Descripción de caso de uso "Calcular métrica FR"

ID:	**CU-5**
Nombre del caso de uso:	Calcular métrica FR
Actores	Cliente (Sistema de re-factorización), Analizador sintáctico, Código fuente
Descripción	El cliente selecciona la métrica FR para realizar el cálculo.
Precondiciones	• Contar con la lista de la información necesaria para realizar el cálculo de las métrica
Excepciones:	
Postcondiciones	• Se obtiene el resultado de la métrica FR
Escenario principal de éxito:	1. El sistema invoca al marco de métricas y selecciona la métrica FR 2. El subsistema correspondiente al cálculo de la métrica del promedio por responsabilidades existente en las clases aplicando la fórmula mostrada en el apartado 4.4 3. El subsistema muestra el resultado obtenido del cálculo de la métrica

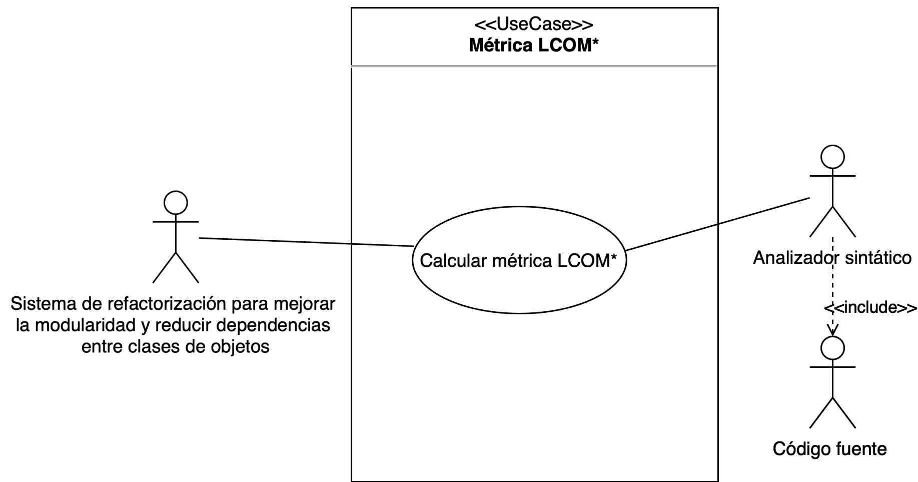


*Figura 12. Diagrama de caso de uso métrica LCOM**

Tabla 7. Descripción de caso de uso "Calcular métrica LCOM"*

ID:	**CU-6**
Nombre del caso de uso:	Calcular métrica LCOM*
Actores	Cliente (Sistema de re-factorización), Analizador sintáctico, Código fuente
Descripción	El cliente selecciona la métrica LCOM* para realizar el cálculo
Precondiciones	• Contar con la lista de la información necesaria para realizar el cálculo de las métrica
Excepciones:	
Postcondiciones	• Se obtiene el resultado de la métrica LCOM*
Escenario principal de éxito:	4. El sistema invoca al marco de métricas y selecciona la métrica LCOM* 5. El subsistema correspondiente al cálculo de la métrica identifica si todos los método de la clase hacen uso de todos los atributos aplicando la formula mostrada en el apartado 4.2 6. El subsistema muestra el resultado obtenido del cálculo de la métrica

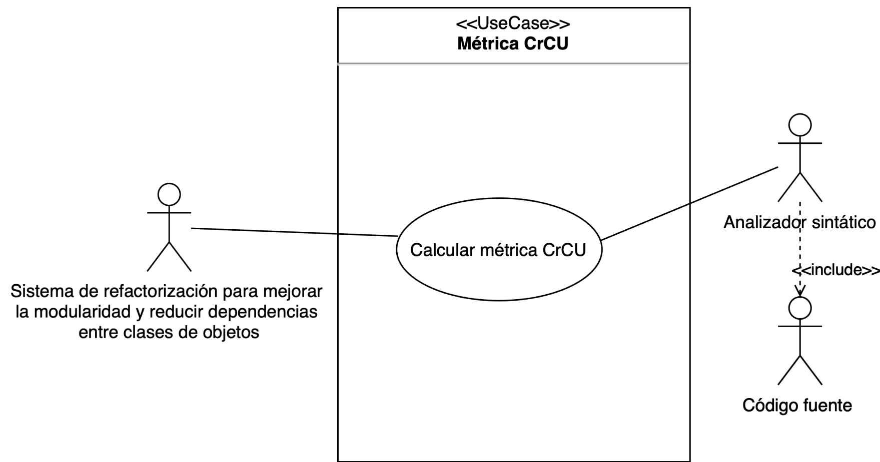


Figura 13. Diagrama de caso de uso métrica CrCU.

Tabla 8. Descripción de caso de uso "Calcular métrica CrCU"

ID:	**CU-7**
Nombre del caso de uso:	Calcular métrica CrCU
Actores	Cliente (Sistema de re-factorización), Analizador sintáctico, Código fuente
Descripción	El cliente selecciona la métrica CrCU para realizar el cálculo
Precondiciones	• Contar con la lista de la información necesaria para realizar el cálculo de las métrica
Excepciones:	
Postcondiciones	• Se obtiene el resultado de la métrica CrCU
Escenario principal de éxito:	1. El sistema invoca al marco de métricas y selecciona la métrica CrCU 2. El subsistema correspondiente cuenta la cantidad de funciones que interactúan para satisfacer una responsabilidad en relación al número total de funciones del módulo que los contiene aplicando la formula mostrada en el apartado 4.3 3. El subsistema muestra el resultado obtenido del cálculo de esta métrica

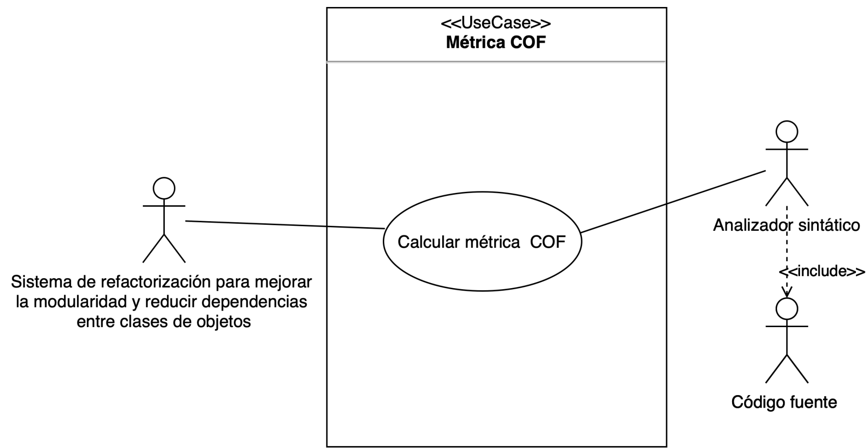


Figura 14. Diagrama de caso de uso métrica COF

Tabla 9. Descripción de caso de uso "Calcular métrica COF"

ID:	**CU-8**
Nombre del caso de uso:	Calcular métrica COF
Actores	Cliente (Sistema de re-factorización), Analizador sintáctico, Código fuente
Descripción	El cliente selecciona la métrica COF para realizar el cálculo.
Precondiciones	• Contar con la lista de la información necesaria para realizar el cálculo de las métrica
Excepciones:	
Postcondiciones	• Se obtiene el resultado de la métrica COF
Escenario principal de éxito:	4. El sistema invoca al marco de métricas y selecciona la métrica COF 5. El subsistema correspondiente al cálculo de la métrica mide el acoplamiento por paso de mensajes (llamadas a métodos), ya que únicamente se miden los canales de comunicación entre clases aplicando la formula mostrada en el apartado 4.1 6. El subsistema muestra el resultado obtenido del cálculo de la métrica

5.2 Diagrama de secuencias del cálculo de métricas

En la Figura 15 se muestra el diagrama de secuencias correspondiente al cálculo de las métricas utilizadas en la investigación.

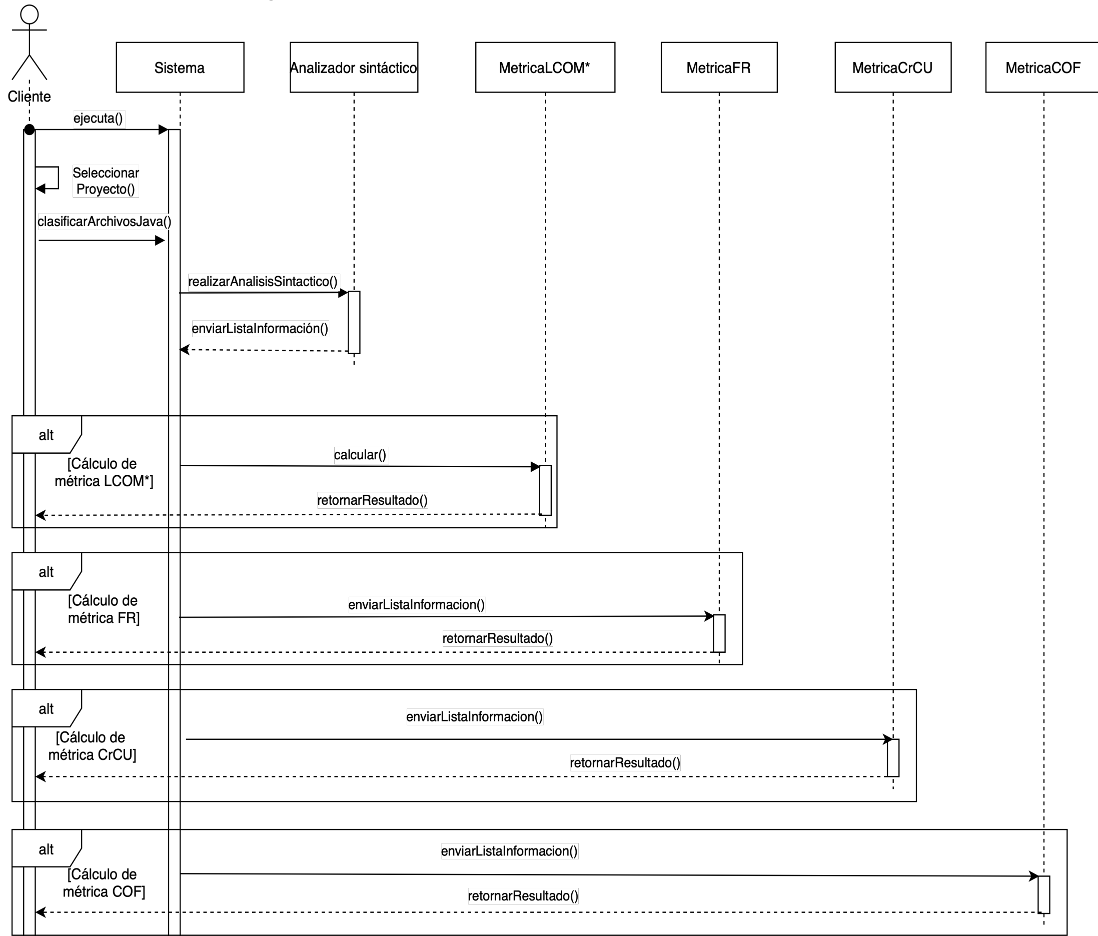


Figura 15. Diagrama de secuencias del cálculo de métricas

5.3 Diagrama de secuencias de los métodos de re-factorización

En esta sección se describen los diagramas de secuencia de los métodos de re-factorización para mejora de la modularidad y reducción de las dependencias entre clases. Con la mejora de la modularidad se busca el aumento de la coherencia mediante la identificación de responsabilidades por clase y aumento de cohesión mediante la identificación de relaciones entre métodos y atributos de las clases. Con respecto a la reducción de las dependencias entre clases, implica la implantación del patrón de diseño *Mediator* para redirigir la comunicación y con esto reducir los canales de comunicación existentes entre las clases.

5.3.1 Mejora de modularidad

A continuación, se muestran los diagramas de los métodos de re-factorización para mejora de modularidad correspondientes al aumento de la coherencia y al aumento de cohesión en las clases, (Figura 16).

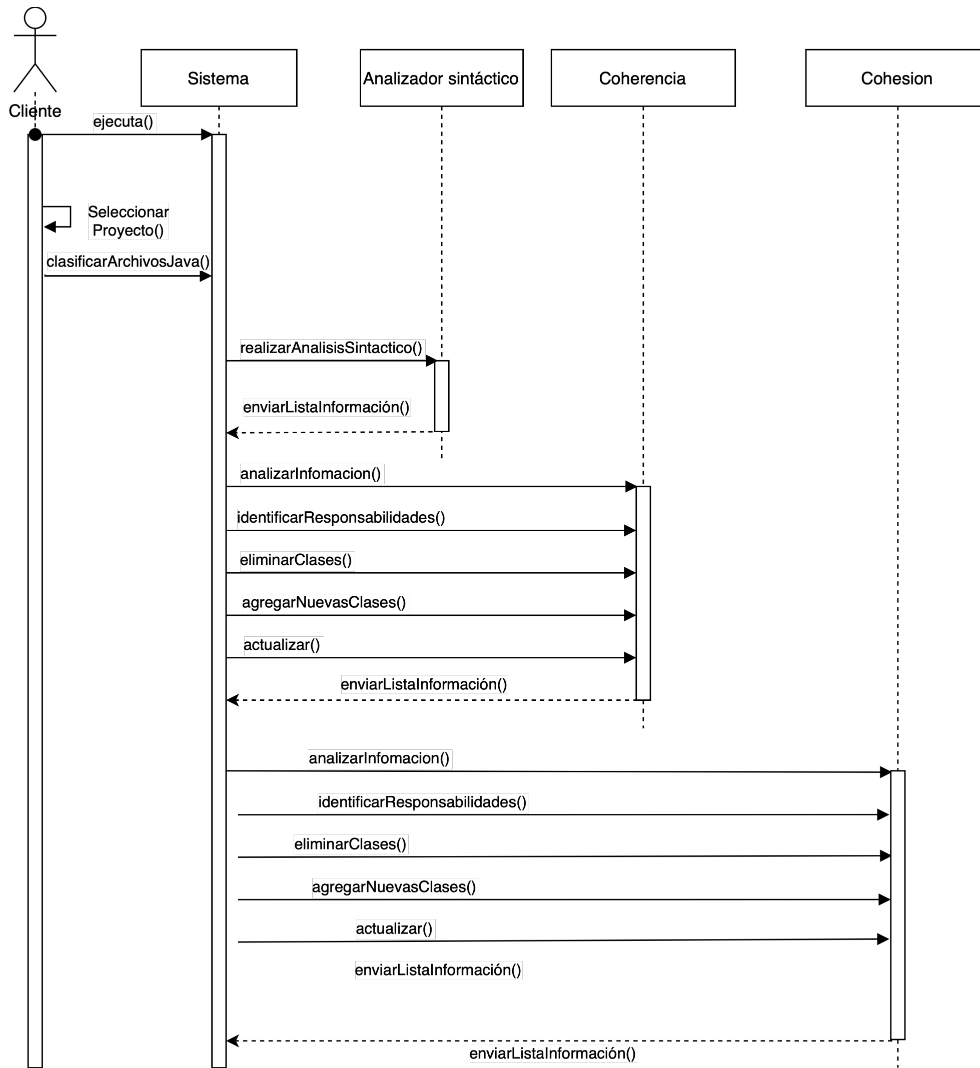


Figura 16. Diagrama de secuencias del primer método de re-factorización para mejora de la modularidad

5.3.2 Reducción de dependencia entre clases

La Figura 17 muestra el diagrama de secuencias correspondiente a la reducción de dependencias entre clases.

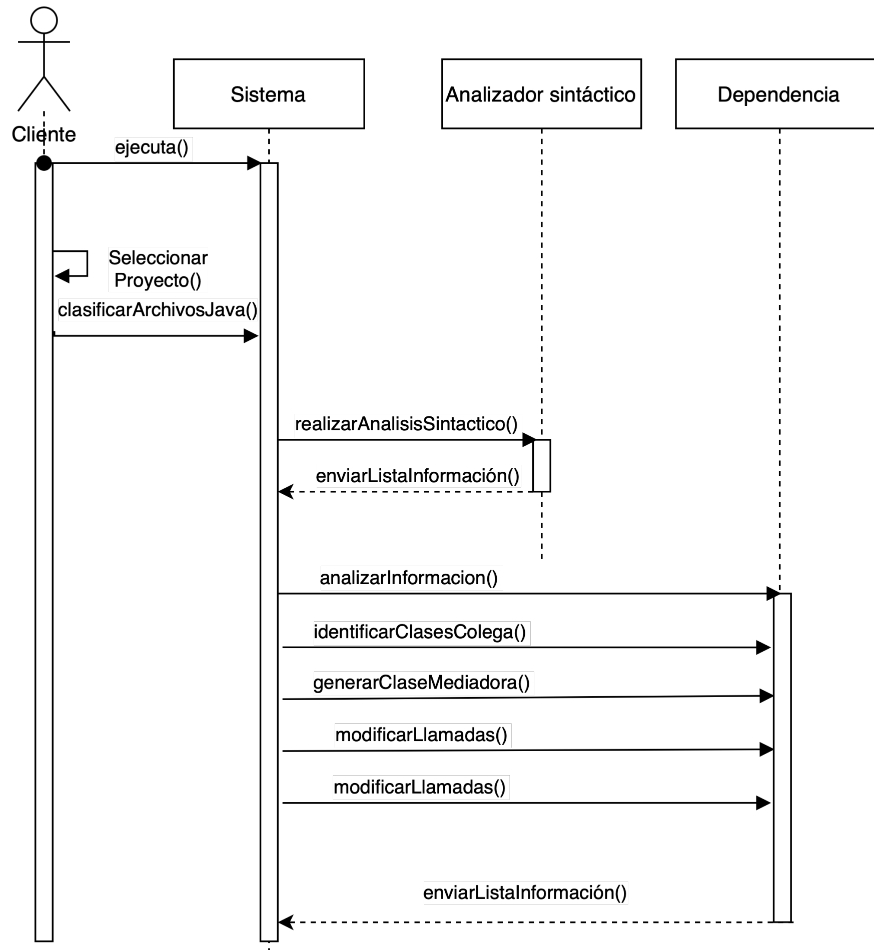


Figura 17. Diagrama de secuencias del segundo método de re-factorización para reducción de dependencias entre clases de objetos

5.4 Diagrama de clases del sistema

En este apartado se muestra el diagrama, (Figura 18), de clases correspondiente al sistema donde se encuentran encapsulados los métodos de re-factorización, así como la interacción con otros módulos, que en conjunto trabajan para cumplir con el objetivo de la investigación. El módulo correspondiente al analizador sintáctico permitió analizar del código para extraer la información necesaria para el cálculo de las métricas y para la re-factorización.

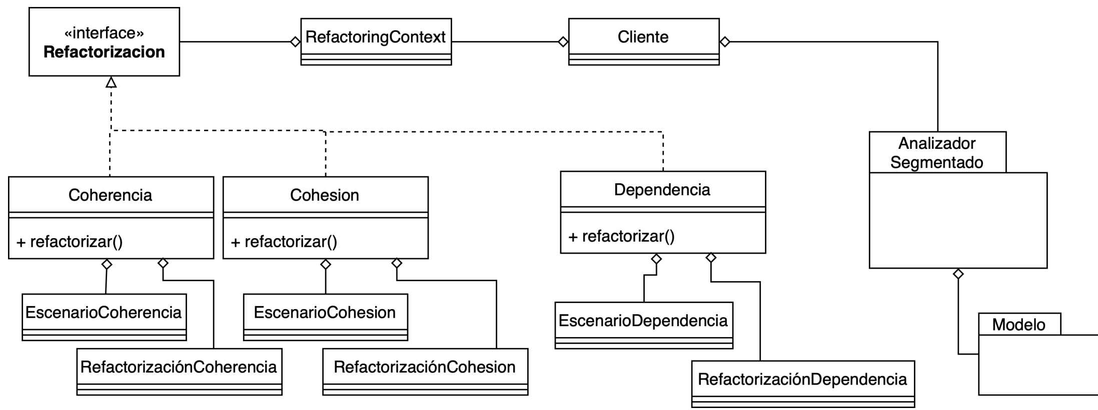


Figura 18. Diagrama de clases general del sistema

La Figura 19 se presenta el diagrama de clases correspondiente al marco de métricas. La métrica creada en esta investigación se encuentra resaltada con color rojo y las métricas existentes utilizadas en la investigación en color azul.

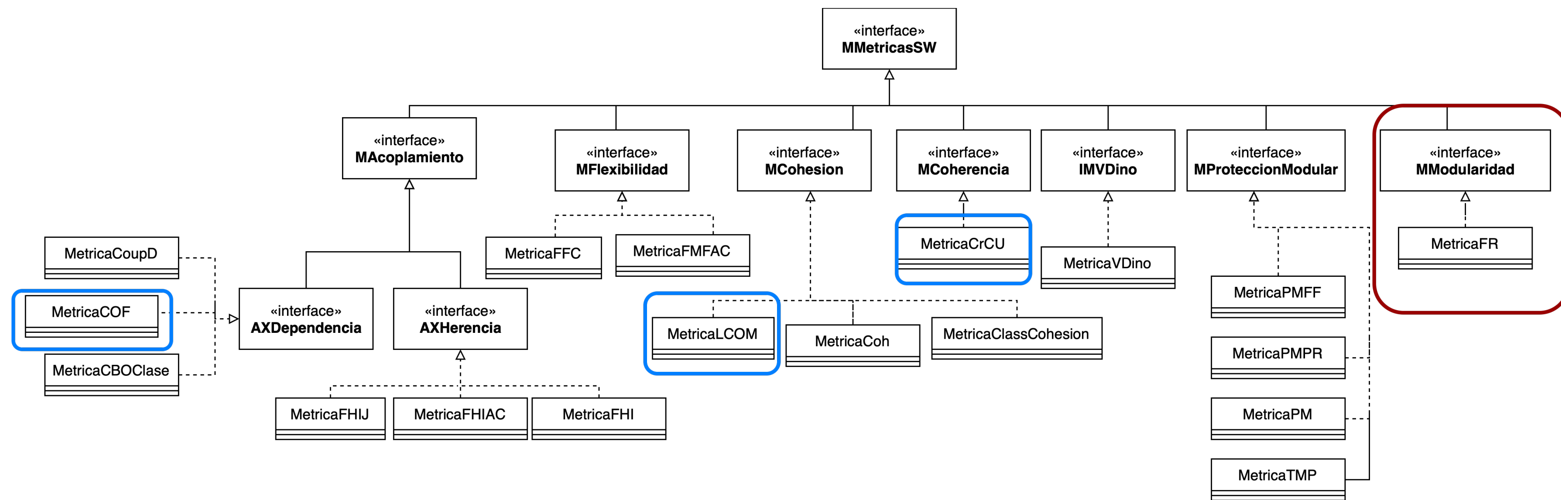


Figura 19. Diagrama de clases del marco de métricas

En la Figura 18 correspondiente al diagrama de clases general del sistema, se hace uso del paquete correspondiente a los modelos utilizados para guardar la información necesaria para el cálculo de métricas y la re-factorización. Estas clases son utilizadas durante el proceso de análisis sintáctico. A continuación, en la Figura 20 se muestra una pequeña descripción de cada una de las clases que conforman el paquete correspondiente al modelo, así como la representación del diagrama de clases y en la Figura 21 se observa un esquema detallado del diagrama de la Figura 20.

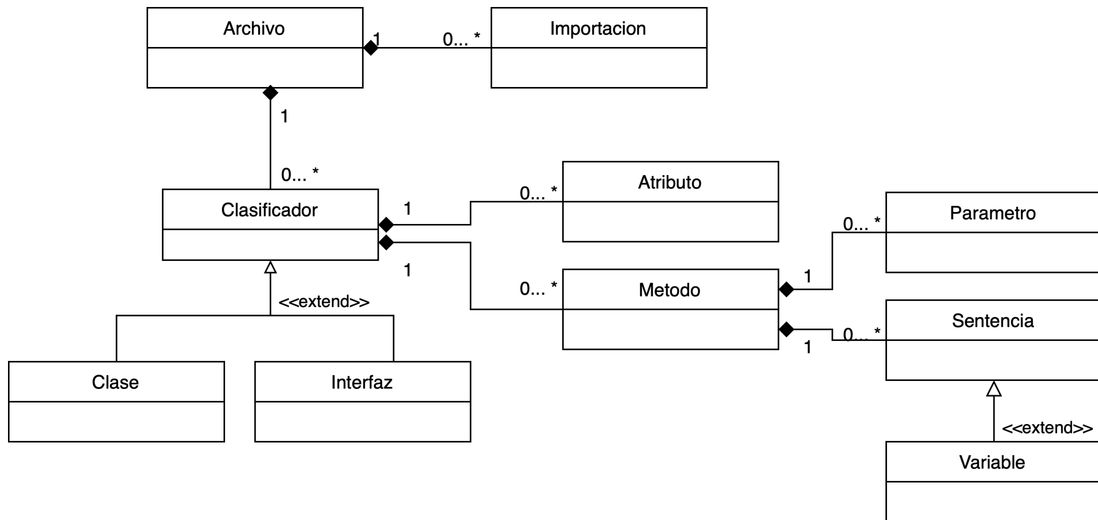


Figura 20. Diagrama de clases del paquete modelo.

Las clases correspondientes al modelo, permiten crear objetos para guardar la información de las clases del proyecto de entrada. Estas clases permiten que se modele un sistema para guardar la información. La clase Archivo contiene información de las importaciones y una o una colección de clasificadores, ya que, como se sabe, un archivo puede contener más de una clase.

En la clase *"Importación"* se guarda la información correspondiente a una importación.

La clase *"Clasificador"* modela las clases e interfaces, esta clase al ser una clase abstracta hereda características que comparten a las clases hijas que son: *"Clase"* e *"Interfaz"*.

La clase *"Atributo"* contiene información que describe a un atributo. En la gramática se indica que un atributo tiene la declaración y posiblemente la inicialización de ésta. Los atributos pueden ser de un tipo de clase existente en el sistema.

La clase *"Parámetro"* contiene información similar a un atributo la diferencia es que el atributo contiene más información descriptiva que un parámetro.

Por otro lado, la clase *"Método"* contiene información que describe a un método, este modelo tiene relación con otros modelos como el de parámetro, sentencia y variable.

La clase *"Sentencia"* es una abstracción de los diferentes tipos de líneas de código que puede tener un método, es posible iterar en sus clases hijas de una forma más sencilla.

La clase *"Variable"* almacena directamente un atributo y así se reutiliza el modelo que requiere para la información que puede presentarse. Una característica es que la variable al ser una sentencia representa una línea de código de un método.

Toda la información de este modelo de objetos se almacena en estructuras de datos dinámicas de tipo lista. En la Figura 21 se ilustra un ejemplo de cómo se representa en memoria la información recabada por el analizador sintáctico.

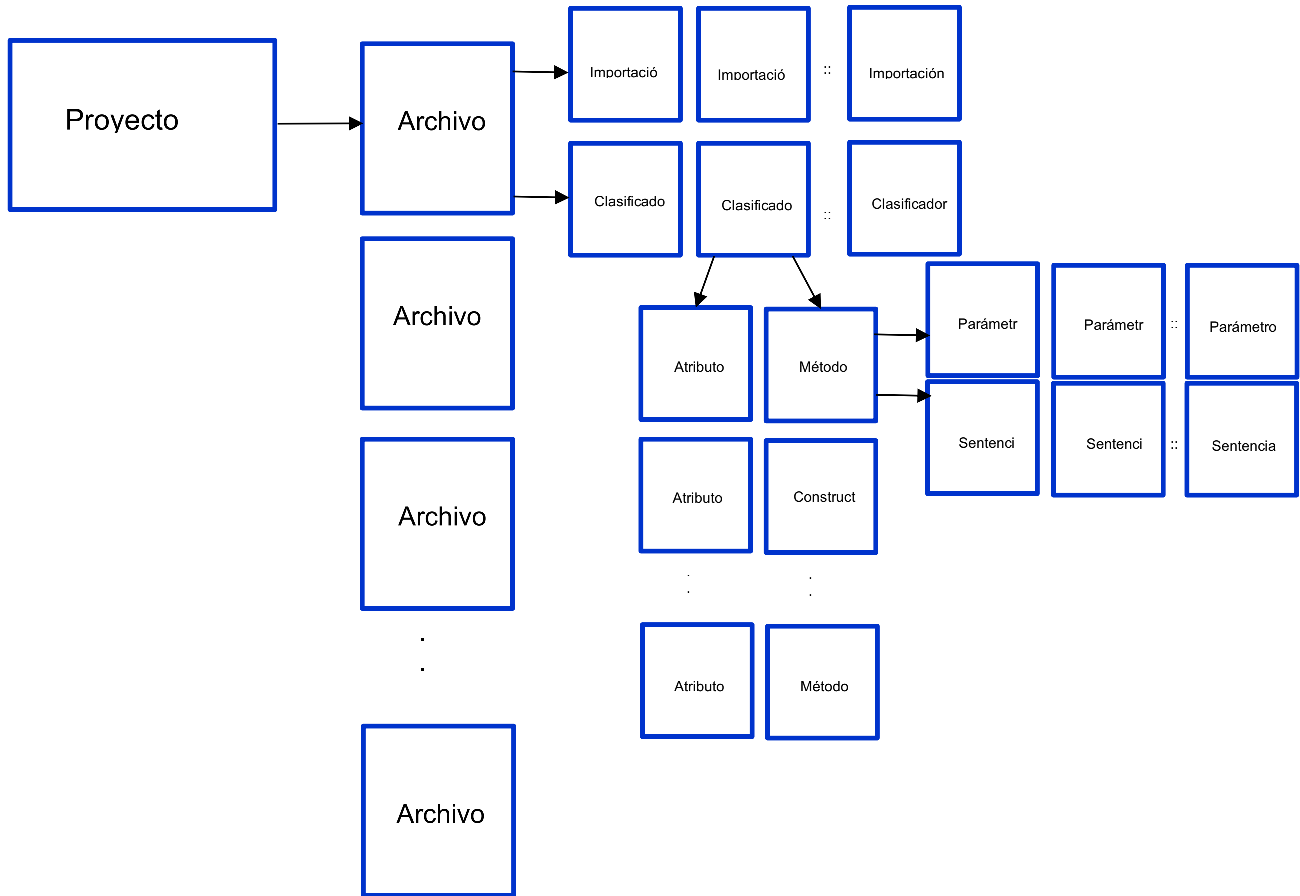


Figura 21. Diagrama detallado del paquete modelo

Capítulo 6. Pruebas de Evaluación

6.1 Mejora de modularidad

Convención de nombres

La convención de nombres que se muestra en la Figura 22, es utilizada a través de toda la evaluación del método para mejorar la modularidad.

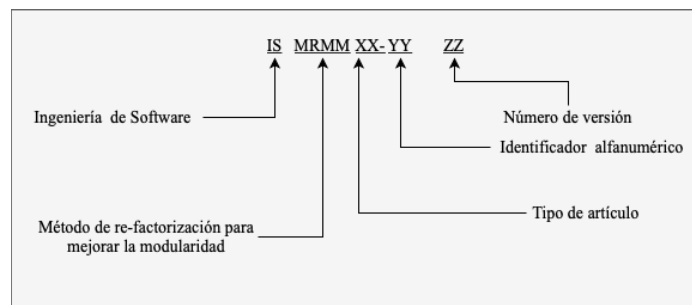


Figura 22. Convención de nombres.

Tipo de Artículo

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

Programas de Control

ISMRMM01 - YYZZ — Módulos de programa

Módulos de programa

ISMRMM02 - YYZZ — Programas de control, utilerías, ordenadores, entre otros.

Documentación de pruebas

ISMRMM03 - YYZZ — Plan de pruebas.

ISMRMM04 - YYZZ — Especificación de diseño de pruebas.

Plan de pruebas

1.- Plan de prueba: ISMRMM03 – 01: Plan de pruebas para la validación del correcto funcionamiento del método de re-factorización para la mejora de la modularidad.

2.- Introducción
El método de re-factorización de arquitecturas de marcos orientados a objetos para mejorar la modularidad tiene la función de analizar las secuencias de métodos iniciadas por un método con calificador de alcance *"public"* y analizar las relaciones entre métodos y atributos de una clase, con el fin de verificar que cada clase tenga una responsabilidad y que además cuente con las relaciones adecuadas entre métodos y atributos, de lo contrario dividir las clases de acuerdo al número de responsabilidades y relaciones.

3.- Artículos de prueba. Los artículos a ser probados son los siguientes:

3.1.- Módulos de programa. Los módulos de programa a ser probados serán identificados como se muestra a continuación:

Tabla 10. Módulos de programa.

Sistema	Función	No.
Método de Re-factorización para mejora de la modularidad	Subsistema de evaluación de secuencias de métodos de una clase, *y* de las relaciones entre métodos y atributos de clases, iniciadas por un método con calificador de alcance *"public"*.	ISMRMM01 - 01
Marco de Métricas de Calidad de Arquitecturas Orientadas a Objetos	Cálculo de métricas de modularidad (LCOM*, CrCu, FR).	ISMRMM01 - 02

3.2 Procedimientos de control de tareas.

Tabla 11. Control de tareas.

Sistema	Función	No.
Programa de aplicación	Localización del paquete al que pertenece la clase.	ISMRMM02 - 01
Programa de aplicación	Localización de las importaciones de la clase.	ISMRMM02 - 02

Programa de aplicación	Localización del nombre de la clase.	ISMRMM02 - 03
Programa de aplicación	Localización de la clase padre de la clase (si existiera).	ISMRMM02 - 04
Programa de aplicación	Localización de atributos de la clase.	ISMRMM02 - 05
Programa de aplicación	Localización de métodos de la clase	ISMRMM02 - 06
Programa de aplicación	Localización de la información de los métodos de clase, tales como: calificador de alcance, si es abstracto, si es estático, nombre, parámetros, sentencias del cuerpo del método.	ISMRMM02 - 07
Programa de aplicación	Conteo de las secuencias iniciadas por un método con calificador de alcance *"public"* existentes en las clases.	ISMRMM02 - 08
Programa de aplicación	Conteo del número de relaciones entre métodos y atributos existentes en las clases.	ISMRMM02 - 09
Programa de aplicación	Cálculo de la métrica CrCU	ISMRMM02 – 10
Programa de aplicación	Cálculo de la métrica LCOM*	ISMRMM02 – 11
Programa de aplicación	Cálculo de la métrica FR	ISMRMM02 - 12
Programa de aplicación	Método de re-factorizar para mejorar la modularidad.	ISMRMM02 - 13
Programa de aplicación	Llenado de las plantillas ST con código refactorizado.	ISMRMM02 - 14
Programa de aplicación	Compilación del código refactorizado.	ISMRMM02 - 15
Programa de aplicación	Verificación del funcionamiento del código refactorizado.	ISMRMM02 – 16

4.- Características a ser probadas.

La siguiente lista describe las características que deben ser probadas.

Tabla 12. Características por probar.

Diseño de Prueba No. de especificación	Descripción
ISMMM04 - 01	Cálculo de métricas de mejora en la modularidad
ISMRMM04 - 02	Método de re-factorizar para mejorar la modularidad

5.- Características a no ser probadas.

5.1. Los casos de prueba no incluirán todas las posibles construcciones sintácticas y combinaciones de éstas, para código escrito en lenguaje "Java".

5.2. El método no indicará si el código de entrada se encuentra libre de errores.

5.3. No se pretende comprobar que la totalidad del proceso de reingeniería para reuso es automática. Es necesario cierto nivel de intervención del experto en el dominio o el experto en programación.

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

6.- Enfoque.
La realización de los casos de prueba y la actividad de pruebas estará asistida por la alumna sustentante de esta tesis de maestría, Marisol Ramírez Cruz. Esto ayudará para asegurar que las pruebas representan efectivamente el desarrollo y uso del sistema SR2 completo.

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

6.2 Pruebas de re-factorización. La validación del método de re-factorización de arquitecturas de marcos orientados a objetos con modularidad incorrecta, que consiste en dividir las clases en caso de tener más de una responsabilidad o no tener relación entre atributos y métodos. La validación se realizará mediante la ejecución del código original contra la ejecución del código refactorizado. Comprobando, que, bajo las mismas entradas, ambos sistemas deberán comportarse de la misma manera y ofrecer los mismos resultados.

6.3 Pruebas de calidad. Las pruebas de calidad incluyen la aplicación de métricas de modularidad CrCU, LCOM* y FR para la medición comparativa del código original contra el código orientado a objetos obtenido del proceso de re-factorización.

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

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

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

8.- Criterio de suspensión y requisitos de reanudación.

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

9.- 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.

Especificación del diseño de pruebas

1.- Diseño de Prueba: ISMRMM04 – 01. Método de re-factorización para mejorar la modularidad

2.- Características a ser probadas.

2.1.- En esta prueba se evaluará el correcto funcionamiento del método de re-factorización al dividir las clases de acuerdo con el número de responsabilidades y de acuerdo con las relaciones entre métodos y atributos.

3.- Refinamiento del enfoque.

El objetivo es evaluar la correcta ejecución del método de re-factorización al dividir las clases de acuerdo con el número de responsabilidades y a las relaciones entre métodos y atributos existentes en estas.

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 escrito. Posteriormente, el subsistema de re-factorización de código tomará el o los archivos y realizará un reconocimiento léxico y sintáctico, generando las estructuras de datos en memoria con la información necesaria para efectos de la re-factorización.

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

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

1.- Diseño de Prueba: ISMRMM04 – 02. Cálculo de métricas de calidad.

2.- Características a ser probadas.

2.1.- En esta prueba se evaluará el cálculo correcto de las métricas de modularidad (LCOM*, CrCU, FR).

3.- Refinamiento del enfoque.

El objetivo es evaluar el correcto funcionamiento de las métricas de modularidad al evaluar un sistema.

Antes de realizar cada caso de prueba, éste será compilado y ejecutado previamente en un compilador para el lenguaje Java, con objeto de corroborar que su construcción está o no correctamente escrita. Posteriormente, para el cálculo de métricas se tomará el código bajo estudio y realizará un reconocimiento léxico y sintáctico, para generar la estructura de datos con la información requerida para estos cálculos.

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

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

Especificación de casos de prueba

1.- Caso de Prueba: ISMRMM05 - 01.

2.- Artículos de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje java, está conformado por 47 clases y tiene como objetivo realizar cálculos estadísticos automáticamente.

a) Características a probar, del proceso de cálculo de métricas de calidad.

Tabla 13. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	Característica Por Ejercitar
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

3.- Especificación de entrada

Las entradas al proceso de cálculo de métricas de calidad son:

1. Las clases pertenecientes a la aplicación de entrada compilado y probado.

4.- Especificación de salida

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

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

1.- Caso de Prueba: ISMRMM05 - 02.

2.- Artículos de Prueba: Marco estadístico

El marco estadístico es un sistema desarrollado en lenguaje java está conformado por 47 clases y tiene como objetivo realizar cálculos estadísticos automáticamente.

a) Características a probar, del método de Re-factorizar para mejorar la modularidad.

Tabla 14. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización
ISMRMM04 – 01	Cálculo de las métricas para verificar si hubo una mejora de la modularidad.

3.- Especificación de entrada

Las entradas al proceso de re-factorizar son:

1. Las clases pertenecientes a la aplicación marco estadístico compilado y probado.

4.- Especificación de salida

A la salida del proceso de reestructura se deberá tener:

1. Las clases pertenecientes a la aplicación del código bajo estudio refactorizadas con reducción de deuda técnica originado por código desagradable que se caracteriza por tener una modularidad incorrecta en suficiencia y completitud en diferentes entidades de software orientadas a objetos.

1.- Caso de Prueba: ISMRMM05 - 03.

2.- Artículos de Prueba: *LibraryAdministration*

LibraryAdministration es un sistema desarrollado en lenguaje Java que tiene como objetivo realizar la administración automática de una librería.

a) Características a probar, del proceso de cálculo de métricas de calidad.

Tabla 15. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	**Característica a Ejercitar**
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

3.- Especificación de entrada

Las entradas al proceso de cálculo de métricas de calidad son:

2. Las clases pertenecientes a la aplicación *LibraryAdministration,* compilado y probado.

4.- Especificación de salida

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

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

1.- Caso de Prueba: ISMRMM05 - 04.

2.- Artículos de Prueba: *LibraryAdministration*

LibraryAdministration es un sistema desarrollado en lenguaje Java que tiene como objetivo realizar la administración automática de una librería.

a) Características a probar, del método de re-factorizar para mejorar la modularidad.

Tabla 16. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	**Característica por Ejercitar**
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización
ISMRMM04 – 01	Comprobar la mejora de la modularidad en la aplicación de prueba.

3.- Especificación de entrada

Las entradas al proceso de re-factorización son:

2. Las clases pertenecientes a la aplicación *LibraryAdministration* compilado y probado.

4.- Especificación de salida

A la salida del proceso de re-factorizar se deberá tener:

3. El valor obtenido de cada una de las métricas de manera independiente.
4. Las clases pertenecientes a la aplicación marco estadístico refactorizadas libre de deuda técnica originado por código desagradable que se caracteriza por tener más de una responsabilidad por clase y además de no tener una correcta modularidad en nivel de suficiencia y completitud.

1.- Caso de Prueba: ISMRMM05 - 05.

2.- Artículos de Prueba: Conversión Autómata

El sistema Conversión Autómata es un sistema desarrollado en lenguaje java que tiene como objetivo convertir una expresión regular a un autómata finito determinista.

a) Características a probar, del proceso de cálculo de métricas de calidad.

Tabla 17. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

3.- Especificación de entrada

Las entradas al proceso de cálculo de métricas de calidad son:

3. Las clases pertenecientes a la aplicación Conversión Autómata compilado y probado.

4.- Especificación de salida

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

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

1.- Caso de Prueba: ISMRMM05 - 06.

2.- Artículos de Prueba: Conversión Autómata

El sistema Conversión Autómata es un sistema desarrollado en lenguaje java que tiene como objetivo convertir una expresión regular a un autómata finito determinista.

a) Características a probar, del método re-factorizar para mejorar la modularidad.

Tabla 18. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización
ISMRMM04 – 01	Comprobar la mejora de la modularidad en la aplicación bajo estudio.

3.- Especificación de entrada

Las entradas al proceso de re-factorizar son:

1. Las clases pertenecientes a la aplicación Conversión Autómata compilado y probado.

4.- Especificación de salida

A la salida del proceso de re-factorizar se deberá tener:

1. El valor obtenido de cada una de las métricas de manera independiente.
2. Las clases pertenecientes a la aplicación marco estadístico refactorizadas libre de deuda técnica originado por código desagradable que se caracteriza por tener más de una responsabilidad por clase y además de no tener una correcta modularidad en nivel de suficiencia y completitud.

Ejecución del plan de pruebas

1.- Caso de Prueba: ISMRMM05 - 01.

2.- Artículos de Prueba: Marco estadístico

En la Tabla 19 se enlistan las características a probar, del proceso de cálculo de métricas de calidad.

Tabla 19. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

Dada la Figura 23 denominada Marco estadístico, que representa un sistema desarrollado en lenguaje java conformado por 47 clases, el cual tiene como objetivo realizar cálculos estadísticos automáticamente. Se procede a realizar el caso de prueba **ISMRCA05 - 01**.

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

Cálculo correcto de las métricas de modularidad (FR, LCOM* Y CrCU)

Se somete la aplicación del caso de prueba al cálculo de métricas de modularidad para comprobar que el cálculo se realiza de manera correcta. Primero se realiza el cálculo manual y posteriormente se realiza un cálculo automático para comparar ambos resultados, los cuales deberán ser iguales.

Cálculo de la métrica FR (Factor de Responsabilidad)

$$FR = \frac{1}{\frac{\sum_{i=1}^{NC} Resp_i}{NC}}$$

Donde:
FR = Factor de responsabilidades
NC = Número total de clases concretas en la arquitectura
Resp = Responsabilidad
FR = Resultado del factor de responsabilidad

El cálculo manual y automático de la métrica FR, correspondiente a la arquitectura de la Figura 23 se muestra en Tabla 20.

Tabla 20. Cálculo manual y automático de la métrica FR de Marco estadístico.

Cálculo manual	Cálculo automático
$FR = \frac{1}{\frac{63}{31}} = \frac{1}{2.032}$ =0.4920	FR = 0.4920

En la Tabla 20 se observa que el valor de la métrica FR obtenido de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se llevó de forma correcta.

Cálculo de la métrica Coherencia de caso de uso (CrCU)

$$CrCU = \frac{n}{m}$$

Donde:
n *= Total de métodos en la secuencia.*
m *= Total de métodos del módulo o paquete.*
El cálculo manual y automático de la métrica CrCU, correspondiente a la arquitectura de la Figura 23 se muestra en Tabla 21.

Tabla 21. Cálculo manual y automático de la métrica CrCU de Marco estadístico.

Cálculo manual	Cálculo automático
$CrCU = \frac{4}{19} = 0.2105$	CrCU: 0.2105

En la Tabla 21 se observa que el valor de la métrica CrCU obtenido de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

Cálculo de la métrica LCOM* (Carencia de cohesión en los métodos)

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j-1}^{a}\mu(A_j)\right) - m}{1 - m}$$

El cálculo manual y automático de la métrica LCOM*, correspondiente a la arquitectura de la Figura 23 se muestra en Tabla 22.

Tabla 22. Cálculo manual y automático de la métrica LCOM* de Marco estadístico.

Cálculo manual		Cálculo automático
Statistic	1.0625	aStatistic = 1.0625
aBuble	0.0	aBuble = 0.0
Integ	0.53571426	Integ = 0.5357142686843872
cDev	0.0	cDev = 0.0
cGaussJordan	0.0	cGaussJordan = 0.0
cSumY	0.0	cSumY = 0.0
cDevY	0.0	cDevY = 0.0
aED	0.0	aED = 0.0
cAriMean	0.0	cAriMean = 0.0
cShell	0.0	cShell = 0.0
cVariance	0.0	cVariance = 0.0
cLinealRegress	0.0	cLinealRegress = 0.0
cDistributionX2	0.0	cDistributionX2 = 0.0
List	0.60	List = 0.6000000238418579
aDistribution	0.0	aDistribution = 0.0
cSquareSum	0.0	cSquareSum = 0.0
cBubleXY	0.0	cBubleXY = 0.0
cSumR	0.0	cSumR = 0.0
xy	0.0	xy = 0.0
aBasFunc	0.0	aBasFunc = 0.0
cBuble	1.0	cBuble = 0.0
cCorrelation	0.0	cCorrelation = 0.0
		aRegression = 0.0
		xyz = 0.0
		aSum = 0.0
		cDistributionT = 0.0
		cElement = 0.8199999928474426
		aDistributionX2 = 0.0
		Cliente = 0.0
		context_ctx = 1.013888955116272
		cCalt = 0.0
		aBetas = 0.0
		aStdFunc = 0.0

aRegression	0.0	cDistribution = 0.0 aDistributionT = 0.0 cDistItems = 0.0 aAriMean = 0.0 cSumXY = 0.0 cMultlRegress = 0.0 cNormal = 0.0 x = 0.0 Distribution = 1.3333 cQuickSort = 0.0 cSum = 0.0
xyz	0.0	
aSum	0.0	
cDistributionT	0.0	
cElement	0.8199	
aDistributionX2	0.0	
Cliente	0.0	
context_ctx	1.013	
cCalt	0.0	
aBetas	0.0	
aStdFunc	0.0	
aListMgt	0.0	
cRange	0.0	
aStdDev	0.0	
cDistribution	0.0	
aDistributionT	0.0	
cDistItems	0.0	
aAriMean	0.0	
cSumXY	0.0	
cMultlRegress	0.0	
cNormal	0.0	
x	0.0	
Distribution	1.333333	
cQuickSort	0.0	
cSum	0.0	

En la Tabla 22 se observa que los valores obtenidos de la métrica LCOM* por cada clase del sistema de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

1.- Caso de Prueba: ISMRMM05 - 02.
2.- Artículos de Prueba: Marco estadístico

En la Tabla 23 se enlistan las características a probar del proceso de re-factorización en la aplicación Marco estadístico.

Tabla 23. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización

ISMRMM04 – 01	Comprobar la mejora de la modularidad en la aplicación Marco Estadístico.

Una vez realizado el proceso de re-factorización automática al sistema marco estadístico se obtuvo como resultado la Figura 23, se tiene como resultado la arquitectura mostrada en la Figura 24.

Figura 24. Arquitectura de clases de la aplicación Marco estadístico resultante de la re-factorización.

Comportamiento del sistema después de la re-factorización

Para comprobar que la aplicación del Marco estadístico mantiene el mismo comportamiento después de ser sometida al método de re-factorización se realizaron pruebas de las funciones proporcionadas por el sistema del marco estadístico. A continuación, se muestra una de las pruebas realizadas que consiste en calcular la media de dos conjuntos de números (listas de números) antes y después de la re-factorización. En la Tabla 24 se muestra los números que conforman cada conjunto.

Tabla 24. Conjunto de datos ingresados a la prueba

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

La Tabla 25 muestra los resultados obtenidos por la aplicación Marco estadístico antes y después de la re-factorización, se puede observar que el comportamiento se mantuvo, por lo que se comprueba que el método de re-factorización no altera el compartimento de la aplicación.

Tabla 25. Cálculo de la media antes y después de la re-factorización de la aplicación Marco estadístico.

Resultados antes de la re-factorización	Resultados después de la re-factorización
Media de la lista X 42.3 Media de la lista Y 30.4	Media de la lista X 42.3 Media de la lista Y 30.4

Comprobar la mejora de la modularidad en la aplicación Marco Estadístico

Para comprobar la mejora de la modularidad de la aplicación Marco estadístico, dicha aplicación se somete al cálculo de las métricas FR, CrCU y LCOM* antes y después de la re-factorización, con el fin de comparar su grado de mejora con base en estas métricas. Para el cálculo de las métricas se compararon los valores resultantes del cálculo de las métricas, las cuales resultaron con un aumento de mejora de la modularidad, algunas en menor porcentaje, debido a la división realizada en las clases misma que afecta en los valores resultantes de las métricas. Con el fin de demostrar la mejora de la modularidad a continuación se muestra una de las pruebas realizadas al sistema. Esta consiste el cálculo de la media de dos conjuntos de datos (Tabla 24). Las Tabla 26 muestra la comparación los valores obtenidos de las métricas FR, CrCU.

Tabla 26. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación Marco estadístico.

Métrica	Valor antes de la re-factorización	Valor después de la re-factorización
FR	0.4920	0.5555
CrCU	0.2105	0.2105

Con respecto a las métricas FR y CrCU el Grafico 1 muestra los valores obtenidos antes y después de la re-factorización de la aplicación del marco estadístico. El promedio de responsabilidades por clase (FR) aumento un 6.35%. La coherencia de caso de uso (CrCU) mantuvo sus valores.

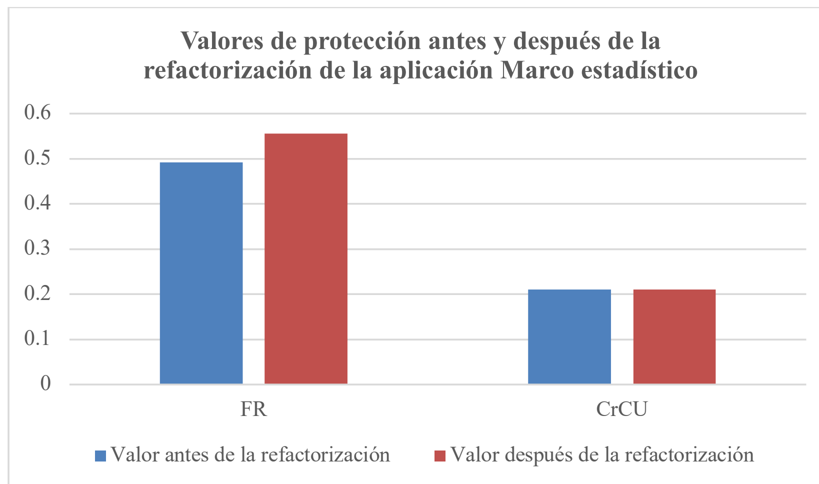


Gráfico 1. Valores de las métricas FR y CrCU antes y después de la re-factorización.

Con respecto a la métrica LCOM* en la Tabla 27 se muestran los valores obtenidos antes de la re-factorización y se observa que el 85.1% del total de las clases se encuentran dentro del valor deseado de la métrica que en este caso es 0. La Tabla 28 muestra los valores obtenidos de la métrica LCOM* después de la re-factorización, en esta se puede observar que el 86.2 % del total de las clases cumplen con el valor deseado de la métrica LCOM*. Con estos valores se concluye que se aumentó la cohesión en un 1.1 %.

De esta manera se demuestra que los valores de las métricas mejoraron métricas debido a que la mayoría de las clases de la aplicación del Marco estadístico después de la re-factorización cuentan con una responsabilidad y la correcta división entre atributos y métodos.

Tabla 27. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación Marco estadístico.

Clase	**Valor antes de la re-factorización**
Statistic	1.0625
aBuble	0.0
Integ	0.53571426
cDev	0.0
cGaussJordan	0.0
cSumY	0.0
cDevY	0.0
aED	0.0
cAriMean	0.0
cShell	0.0
cVariance	0.0
cLinealRegress	0.0
cDistributionX2	0.0
List	0.60
aDistribution	0.0
cSquareSum	0.0
cBubleXY	0.0
cSumR	0.0
xy	0.0
aBasFunc	0.0
cBuble	1.0
cCorrelation	0.0
aRegression	0.0
xyz	0.0
aSum	0.0
cDistributionT	0.0
cElement	0.8199
aDistributionX2	0.0
context_ctx	1.013
cCalt	0.0
aBetas	0.0
aStdFunc	0.0
aListMgt	0.0

cRange	0.0
aStdDev	0.0
cDistribution	0.0
aDistributionT	0.0
cDistItems	0.0
aAriMean	0.0
cSumXY	0.0
cMultlRegress	0.0
cNormal	0.0
x	0.0
Distribution	1.333333
cQuickSort	0.0
cSum	0.0

Tabla 28. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación Marco estadístico

Clase	Valor antes de la re-factorización
aStatistic	1.0625
aBuble	0.0
cDev	0.0
cGaussJordan	0.0
cSumY	0.0
cDevY	0.0
aED	0.0
Coher_Integ3	0.0
cAriMean	0.0
Coher_Integ2	0.0
cShell	0.0
cVariance	0.0
cLinealRegress	0.0
Coher_Distribution3	0.0
cDistributionX2	0.0
List	0.60
aDistribution	0.0
cSquareSum	0.0
cBubleXY	0.0
cSumR	0.0
xy	0.0
aBasFunc	0.0
Coher_Distribution2	0.0
cBuble	0.0
cCorrelation	0.0
aRegression	0.0
Coher_Distribution1	0.0
xyz	0.0
aSum	0.0
cDistributionT	0.0
cElement	0.819999
aDistributionX2	0.0
Cliente	0.0
context_ctx	1.01388
cCalt	0.0

aBetas	0.0
aStdFunc	0.0
aListMgt	0.0
cRange	0.0
aStdDev	0.0
cDistribution	0.0
aDistributionT	0.0
Coher_Integ1	0.7142
cDistItems	0.0
aAriMean	0.0
cSumXY	0.0
cMultlRegress	0.0
cNormal	0.0
x	0.0
cQuickSort	0.0
cSum	0.0

1.- Caso de Prueba: ISMRMM05 - 03.

2.- Artículos de Prueba: *LibraryAdministration*
En la Tabla 29 se enlistan las características a probar, del proceso de cálculo de métricas de calidad.

Tabla 29. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

Dada la Figura 25 denominada *LibraryAdministration*, que representa un sistema desarrollado en lenguaje Java, el cual tiene como objetivo realizar la administración automática de una librería. Se procede a realizar el caso de prueba **ISMRCA05 - 03**.

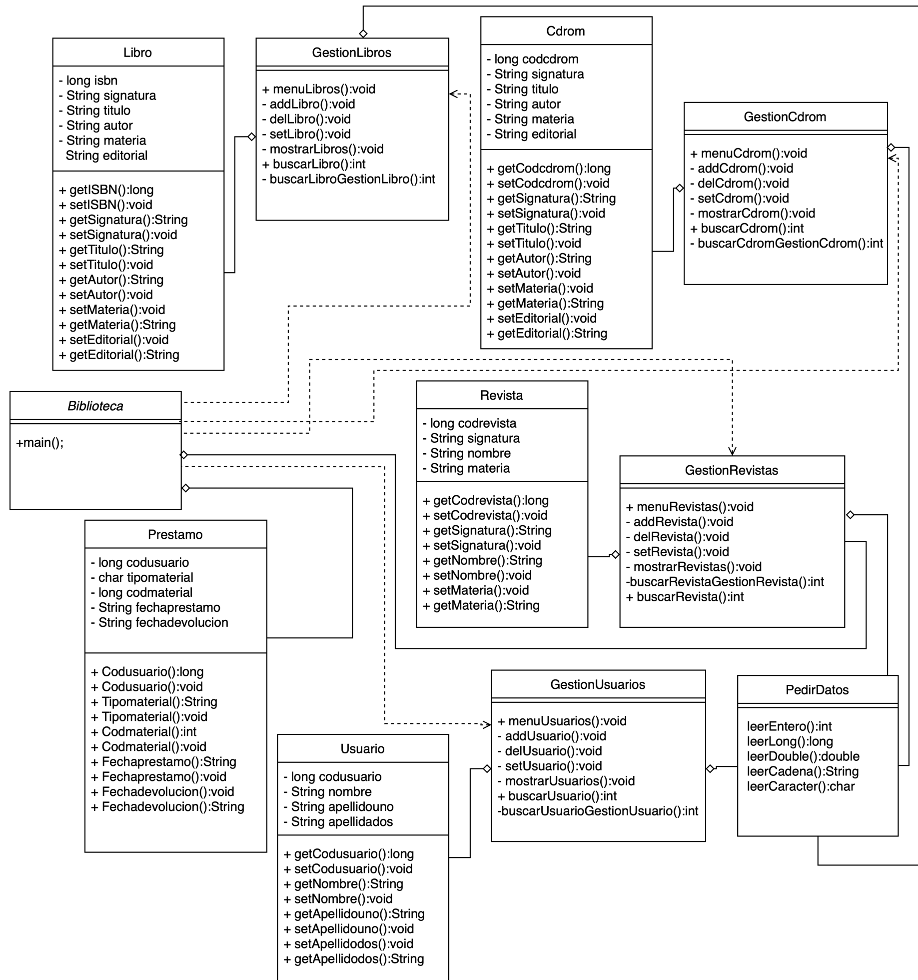


Figura 25. Arquitectura de clases de la aplicación *LibraryAdministration* antes de la re-factorización.

Cálculo correcto de las métricas de modularidad (FR, LCOM* y CrCU)

Cálculo de la métrica FR (Factor de Responsabilidad)

$$FR = \frac{1}{\frac{\sum_{i=1}^{NC} Resp_i}{NC}}$$

Donde:
NC = Número total de clases concretas en la arquitectura
Resp = Responsabilidad

El cálculo manual y automático de la métrica FR, correspondiente a la arquitectura de la Figura 25 se muestra en Tabla 30.

Tabla 30. Cálculo manual y automático de la métrica FR de *LibraryAdministration.*

Cálculo manual	Cálculo automático
$FR = \frac{1}{\frac{63}{10}} = \frac{1}{6.3} = 0.15873$	FR = 0.15873

En la Tabla 30 se observa que el valor de la métrica FR obtenido tanto de manera manual como de manera automática fue el mismo, por lo que el cálculo automático de la métrica se efectuó de forma correcta.

Cálculo de la métrica Coherencia de caso de uso (CrCU)

$$CrCU = \frac{n}{m}$$

Donde:
n *= Total de métodos en la secuencia.*
m *= Total de métodos del módulo o paquete.*

El cálculo manual y automático de la métrica CrCU, correspondiente a la arquitectura de la Figura 25 se muestra en Tabla 31.

Tabla 31. Cálculo manual y automático de la métrica CrCU de *LibraryAdministration.*

Cálculo manual	Cálculo automático
$CrCU = \frac{28}{33} = 0.8484$	CrCU: 0.8484

En la Tabla 31 se observa que el valor de la métrica CrCU obtenido tanto de manera manual como de manera automática fue el mismo, por lo que el cálculo automático de la métrica se efectuó de forma correcta .

Cálculo de la métrica LCOM* (Carencia de cohesión en los métodos)

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j-1}^{a}\mu(A_j)\right) - m}{1 - m}$$

Donde
a *es el número de atributos.*
m *el número de métodos.*
$\mu(A_j)$ *es el número de métodos que acceden al atributo* A_j

Esta métrica sólo puede calcularse cuando $m > 1$.
Cuando todos los métodos acceden a todos los atributos, entonces Σμ(Aj) = ma, y por lo tanto LCOM = 0. Esto indica una cohesión perfecta. Valores cercanos a 0 indican que la mayoría de los métodos accede a la mayoría de las instancias. Por el contrario, cuando cada método accede solo a un atributo, entonces Σ μ(Aj) = a y, por lo tanto, LCOM* = 1, lo cual indica falta de cohesión.*

El cálculo manual y automático de la métrica LCOM*, correspondiente a la arquitectura de la Figura 25 se muestra en Tabla 32.

Tabla 32. Cálculo manual y automático de la métrica LCOM* de *LibraryAdministration.*

Cálculo manual		Cálculo automático
Prestamo	0.8888889	Prestamo = 0.8888888955116272
Libro	0.90909093	Libro = 0.9090909361839294
GestionLibros	0.0	GestionLibros = 0.0
GestionRevistas	0.0	GestionRevistas = 0.0
Biblioteca	0.0	Biblioteca = 0.0
GestionCdrom	0.0	GestionCdrom = 0.0
GestionUsuarios	0.0	GestionUsuarios = 0.0
Usuario	0.66666686	Usuario = 0.6666666865348816
PedirDatos	0.0	PedirDatos = 0.0
Cdrom	0.90909093	Cdrom = 0.9090909361839294
Revista	0.6666668	Revista = 0.6666666865348816

En la Tabla 32 se observa que los valores obtenidos de la métrica LCOM* por cada clase del sistema de manera manual y automática fue el mismo por lo que el cálculo automático de la métrica se efectuó de manera correcta.

1.- Caso de Prueba: ISMRMM05 - 04.

2.- Artículos de Prueba: *LibraryAdministration*

En la Tabla 33 se enlistan las características a probar del proceso de re-factorización en la aplicación *LibraryAdministration*.

Tabla 33. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización
ISMRMM04 – 01	Comprobar la mejora de la modularidad en la aplicación *LibraryAdministration*

Una vez realizado el proceso de re-factorización automática al sistema de *LibraryAdministration* en el que se tuvo como entrada la arquitectura mostrada en la Figura 25, se tiene como resultado la arquitectura mostrada en la Figura 26.

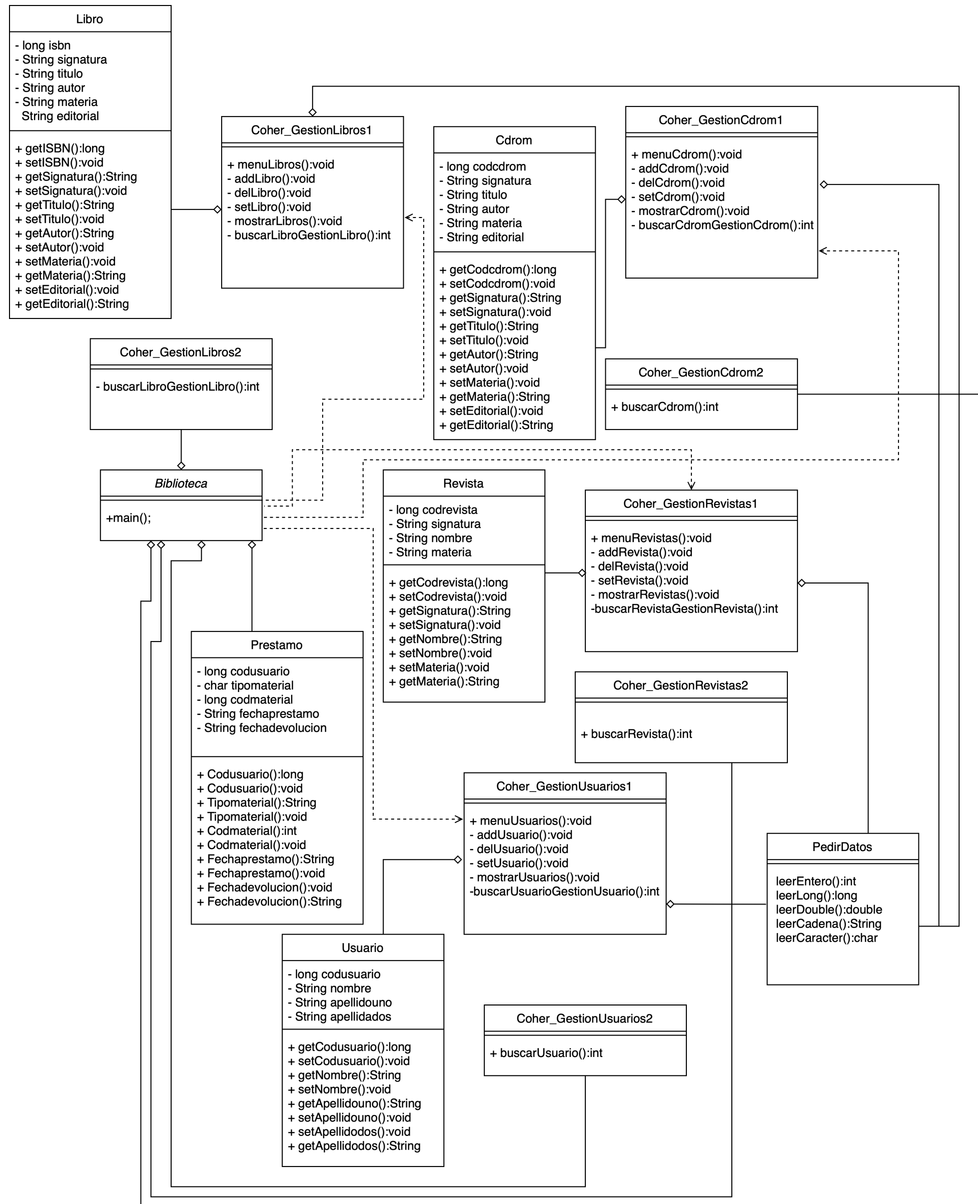


Figura 26. Arquitectura de clases de la aplicación *LibraryAdministration* después de la re-factorización.

Comportamiento del sistema después de la re-factorización

Para comprobar que la aplicación del *LibraryAdministration* mantiene el mismo comportamiento después de ser sometida al método de re-factorización se realizaron pruebas de todas las funcionalidades proporcionadas por el sistema. A continuación, se muestra una prueba, ésta consiste en la inserción de los datos de un usuario al sistema antes y después de la re-factorización. La Tabla 34 muestra los datos ingresados a la prueba.

Tabla 34. Datos ingresados en la prueba

Código usuario	001
Nombre	Jesús
Primer Apellido	Santiago
Segundo apellido	Manzano

La Tabla 35 muestra los resultados obtenidos por la aplicación *LibraryAdministration* antes y después de la re-factorización. Se puede observar que el comportamiento se mantuvo al hacer uso de una de las funcionalidades que proporciona el sistema, por lo que se comprueba que el método de re-factorización no altera el compartimento de la aplicación.

Tabla 35. Inserción de un usuario en la aplicación *LibraryAdministration* antes y después de la re-factorización

Resultados antes de la re-factorización	Resultados después de la re-factorización
GESTIÓN DE USUARIOS ------------------- 1. Añadir usuario. 2. Eliminar usuario. 3. Modificar usuario. 4. Mostrar usuarios. 0. Volver al menú principal. ¿Qué desea hacer? 1 Introduzca el código del usuario que desea añadir. 001 Introduzca el nombre del usuario. Jesus Introduzca el primer apellido del usuario. Santiago Introduzca el segundo apellido del usuario. Manzano El usuario con el código 1 ha sido añadido correctamente. GESTIÓN DE USUARIOS ------------------- 1. Añadir usuario. 2. Eliminar usuario. 3. Modificar usuario. 4. Mostrar usuarios. 0. Volver al menú principal. ¿Qué desea hacer?	GESTIÓN DE USUARIOS ------------------- 1. Añadir usuario. 2. Eliminar usuario. 3. Modificar usuario. 4. Mostrar usuarios. 5. Eliminar Usuarios. 0. Volver al menú principal. ¿Qué desea hacer? 1 Introduzca el código del usuario que desea añadir. 001 Introduzca el nombre del usuario. Jesus Introduzca el primer apellido del usuario. Santiago Introduzca el segundo apellido del usuario. Manzano El usuario con el código 1 ha sido añadido correctamente. GESTIÓN DE USUARIOS ------------------- 1. Añadir usuario. 2. Eliminar usuario. 3. Modificar usuario. 4. Mostrar usuarios. 5. Eliminar Usuarios. 0. Volver al menú principal. ¿Qué desea hacer?

Comprobar la mejora de la modularidad en la aplicación *LibraryAdministration*

Para comprobar que hubo una mejora en la modularidad de la aplicación *LibraryAdministration*, dicha aplicación se somete al cálculo de las métricas FR, CrCU y LCOM* antes y después de la re-factorización, con el fin de comparar el grado de mejora con base en a estas métricas. Con

el fin de demostrar la mejora de la modularidad a continuación se muestra una de las pruebas realizadas del cálculo de las métricas al sistema. Esta consiste en añadir un usuario al sistema antes y después de la re-factorización (Tabla 32). La Tabla 36 muestra la comparación de los valores de las métricas FR, CrCU.

Tabla 36. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación *LibraryAdministration.*

Métrica	Valor antes de la re-factorización	Valor después de la re-factorización
FR	0.1587	0.22222
CrCU	0.8484	0.9655

Como se indicó anteriormente uno de los objetivos principales que se tiene en este trabajo de investigación, es mejorar la modularidad en suficiencia y completitud de la aplicación que se someta al método de re-factorización.

Con respecto a las métricas FR y CrCU el Gráfico 2 muestra los valores obtenidos antes y después de la re-factorización de la aplicación del *LibraryAdministration*, . Se observa que el valor resultante aumentó. El factor de responsabilidad (FR) aumento 6.35%. La coherencia de caso de uso (CrCU) aumento en un 11.71%.

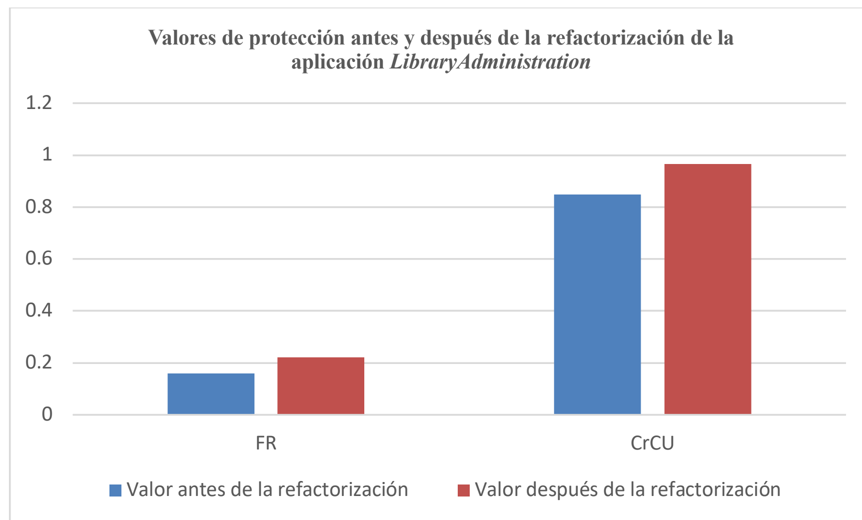


Gráfico 2. Valores de las métricas FR y CrCU antes y después de la re-factorización.

Referente a la métrica LCOM* en la Tabla 37 se muestran los valores obtenidos de la métrica LCOM* antes de la re-factorización, en ésta se observa que el 54.54 % del total de las clases se encuentran dentro del valor deseado de la métrica que en este caso es 0. La Tabla 38 muestra los valores obtenidos de la métrica LCOM* después de la re-factorización, en ésta se puede

observar que el 66.66% del total de las clases cumplen con el valor deseado de la métrica LCOM*. Con esto se concluye que se disminuyó la carencia de cohesión en un 12.12%.

Tabla 37. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación *LibraryAdministration.*

Clase	Valor antes de la re-factorización
Prestamo	0.8888889
Libro	0.90909094
GestionLibros	0.0
GestionRevistas	0.0
Biblioteca	0.0
GestionCdrom	0.0
GestionUsuarios	0.0
Usuario	0.6666668
PedirDatos	0.0
Cdrom	0.90909093
Revista	0.6666668

Tabla 38. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación *LibraryAdministration*

Clase	Valor antes de la re-factorización
Prestamo	0.8888889
Libro	0.90909094
Biblioteca	0.0
Coher_GestionUsuarios2	0.0
Coher_GestionCdrom1	0.0
Coher_GestionRevistas2	0.0
Coher_GestionLibros2	0.0
Usuario	0.66666668
PedirDatos	0.0
Cdrom	0.9090909
Coher_GestionLibros1	0.0
Coher_GestionCdrom2	0.0
Coher_GestionRevistas1	0.0
Revista	0.66666
Coher_GestionUsuarios1	0.0

De esta manera se demuestra que los valores de las métricas mejoraron en la arquitectura de salida, esto es debido a que la mayoría de las clases de la aplicación del *LibraryAdministration* después de la re-factorización cuentan con una responsabilidad y la correcta división entre atributos.

1.- Caso de Prueba: ISMRMM05 - 05.

2.- Artículos de Prueba: *ConversionAutomata*
En la Tabla 39 se enlistan las características a probar, del proceso de cálculo de métricas de calidad.

Tabla 39. Características por probar del proceso del cálculo de métricas de calidad

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRMM04 – 02	Cálculo correcto de las métricas FR, LCOM* y CrCU.

Dada la Figura 27 denominada *ConversiónAutómata*, que representa un sistema desarrollado en lenguaje Java, el cual tiene como realizar la conversión de una expresión regular a un autómata finito determinista minimizado. Se procede a realizar el caso de prueba **ISMRCA05 - 05**.

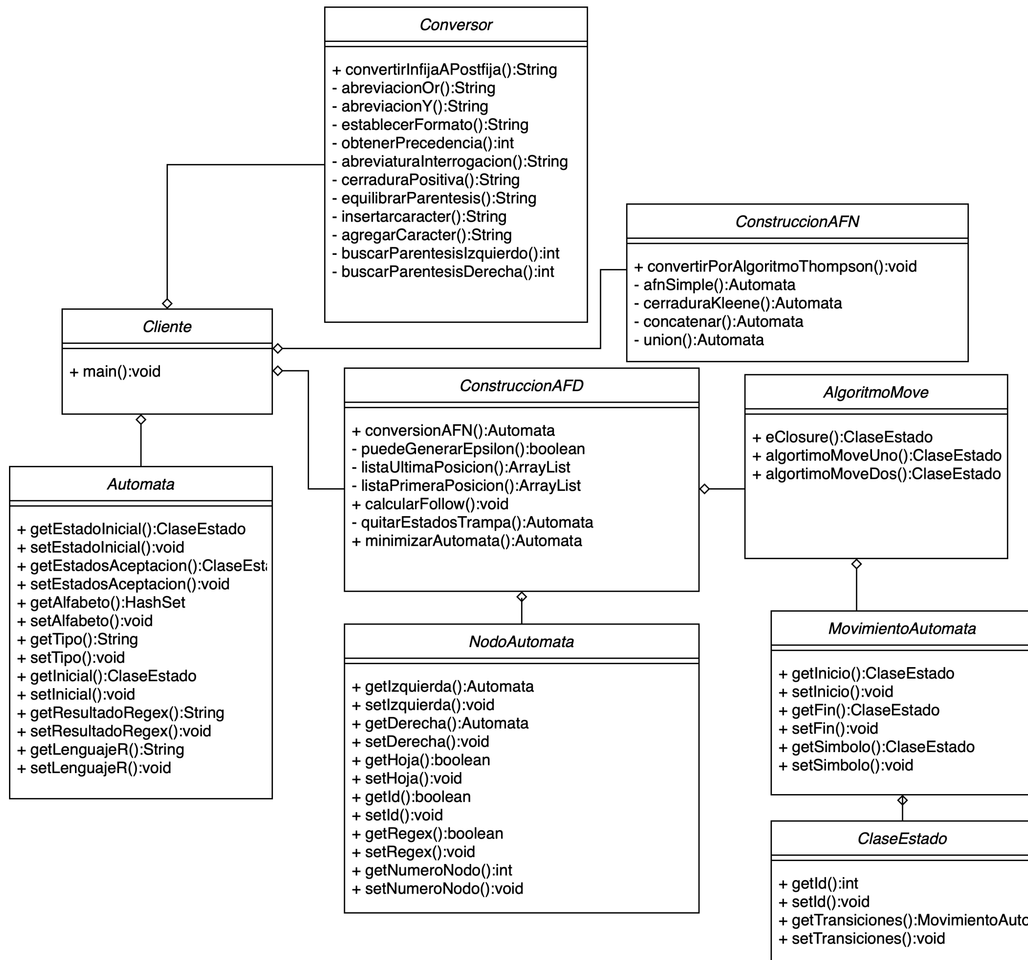


Figura 27. Arquitectura de clases de la aplicación *ConversionAutomata* antes de la re-factorización.

Cálculo correcto de las métricas de modularidad (FR, LCOM* Y CrCU)

Cálculo de la métrica FR (Factor de Responsabilidad)

$$FR = \frac{1}{\frac{\sum_{i=1}^{NC} Resp_i}{NC}}$$

Donde:
FR = Factor de responsabilidades
NC = Número total de clases concretas en la arquitectura
Resp = Responsabilidad
FR = Resultado del factor de responsabilidad

El cálculo manual y automático de la métrica FR, correspondiente a la arquitectura de la Figura 27 se muestra en Tabla 40.

Tabla 40. Cálculo manual y automático de la métrica FR de *ConversionAutomata*.

Cálculo manual	Cálculo automático
$FR = \frac{1}{\frac{55}{9}} = \frac{1}{6.11} = 0.1636$	FR = 0.1636

En la Tabla 40 se observa que el valor de la métrica FR obtenido de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

Cálculo de la métrica FR (Factor de Responsabilidad)

$$CrCU = \frac{n}{m}$$

Donde:
n *= Total de métodos en la secuencia.*
m *= Total de métodos del módulo o paquete.*

El cálculo manual y automático de la métrica CrCU, correspondiente a la arquitectura de la Figura 27 se muestra en Tabla 41.

Tabla 41. Cálculo manual y automático de la métrica CrCU de *ConversionAutomata*.

Cálculo manual	Cálculo automático
$CrCU = \frac{13}{13} = 1.0$	---------- CrCU: 1.0 ----------

En la Tabla 41 se observa que el valor de la métrica CrCU obtenido de manera manual, así como el obtenido de manera automática es el mismo, por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

Cálculo de la métrica LCOM* (Carencia de cohesión en los métodos)

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j-1}^{a}\mu(A_j)\right) - m}{1 - m}$$

Donde
a *es el número de atributos.*
m *el número de métodos.*
$\mu(A_j)$ *es el número de métodos que acceden al atributo* A_j

Esta métrica sólo puede calcularse cuando $m > 1$.
Cuando todos los métodos acceden a todos los atributos, entonces Σμ(Aj) = ma, y por lo tanto LCOM = 0. Esto indica una cohesión perfecta. Valores cercanos a 0 indican que la mayoría de los métodos accede a la mayoría de las instancias. Por el contrario, cuando cada método accede solo a un atributo, entonces Σ μ(Aj) = a y, por lo tanto, LCOM* = 1, lo cual indica una falta de cohesión.*

El cálculo manual y automático de la métrica LCOM*, correspondiente a la arquitectura de la Figura 27 se muestra en Tabla 42.

Tabla 42. Cálculo manual y automático de la métrica LCOM* de *ConversionAutomata*.

Cálculo manual		Cálculo automático
NodoAutomata	1.0500	NodoAutomata = 1.0500000715255737
ConstruccionAFD	0.9333	ConstruccionAFD = 0.93333333730697
MovimientoAutomata	0.571428	MovimientoAutomata = 0.57142859697
ConstruccionAFN	0.0	ConstruccionAFN = 0.0
Conversor	0.0	Conversor = 0.0
Cliente	0.0	Cliente = 0.0
Automata	0.8646	Automata = 0.864661693572998
ClaseEstado	0.625	ClaseEstado = 0.625
AlgoritmoMove	0.0	AlgoritmoMove = 0.0

En la Tabla 42 se observa que los valores obtenidos de manera manual y automática, de la métrica LCOM* por cada clase del sistema fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

1.- Caso de Prueba: ISMRMM05 - 06.

2.- Artículos de Prueba: *ConversionAutomata*

En la Tabla 43 se enlistan las características a probar del proceso de re-factorización en la aplicación *ConversionAutomata.*

Tabla 43. Características por probar del proceso de re-factorización

Diseño de Prueba No. de especificación	**Característica por Ejercitar**
ISMRMM04 – 01	Comportamiento del sistema después de la re-factorización
ISMRMM04 – 01	Comprobar la mejora de la modularidad en la aplicación Marco Estadístico.

Una vez realizado el proceso de re-factorización automática al sistema de *ConversionAutomata* en el que se tuvo como entrada la arquitectura mostrada en la Figura 27, se tiene como resultado la arquitectura mostrada en la Figura 28.

Figura 28. Arquitectura de clases de la aplicación Conversión Autómata después de la re-factorización.

Comportamiento del sistema después de la re-factorización

Para comprobar que la aplicación del sistema *ConversionAutomata* mantiene el mismo comportamiento después de ser sometida al método de re-factorización, se realizaron pruebas de las funciones proporcionadas por el sistema.A continuación, se muestra una prueba que

consiste en la conversión de una expresión infija a postfija. La Tabla 44 muestra la expresión regular ingresada.

Tabla 44. Datos ingresados en la prueba

Expresión regular
(a|b)*abb

La Tabla 45 muestra los resultados obtenidos por la aplicación *ConversionAutomata* antes y después de la re-factorización, se puede observar que el comportamiento se mantuvo, por lo que se comprueba que el método de re-factorización no altera el compartimento de la aplicación.

Tabla 45. conversión de una expresión infija a postfija.

Resultados antes de la re-factorización
Expresion ingresada (a|b)*abb Infija a Postfija: ab|*a.b.b.
Resultados después de la re-factorización
Expresion ingresada (a|b)*abb Infija a Postfija: ab|*a.b.b.

Comprobar la mejora de la modularidad en la aplicación Conversión Autómata

Para comprobar que la mejora de la modularidad de la aplicación *ConversionAutomata* se aumentó, dicha aplicación se somete al cálculo de las métricas FR, CrCU y LCOM* antes y después de la re-factorización, con el fin de comparar su grado de mejora con base en a estas métricas. Se realizaron varios cálculos de las métricas para comprobar que el sistema entregará el resultado correcto de las cuales resultó con aumento de mejora de la modularidad, algunas en menor porcentaje, debido a la división realizada en las clases misma que afecta en los valores resultantes de las métricas. Con el fin de demostrar la mejora de la modularidad a continuación se muestra una de las pruebas realizadas al sistema que consiste en la conversión de una expresión infija a una expresión postfija (Tabla 44). La Tabla 46 muestra la comparación de los valores de las métricas FR, CrCU.

Tabla 46. Valores obtenidos de las métricas FR y CrCU antes y después de la re-factorización de la aplicación Conversión Autómata.

Métrica	Valor antes de la re-factorización	Valor después de la re-factorización
FR	0.1636	0.2881
CrCU	1.0	1.0

Con respecto a las métricas FR y CrCU, el Gráfico 3 muestra los valores obtenidos antes y después de la re-factorización de la aplicación del Conversión Autómata. Se puede observar que el factor de responsabilidad (FR) aumento 12.45%. La coherencia de caso de uso (CrCU) mantuvo su valor deseable en 1 por lo que con la re-factorización no se vio afectado.

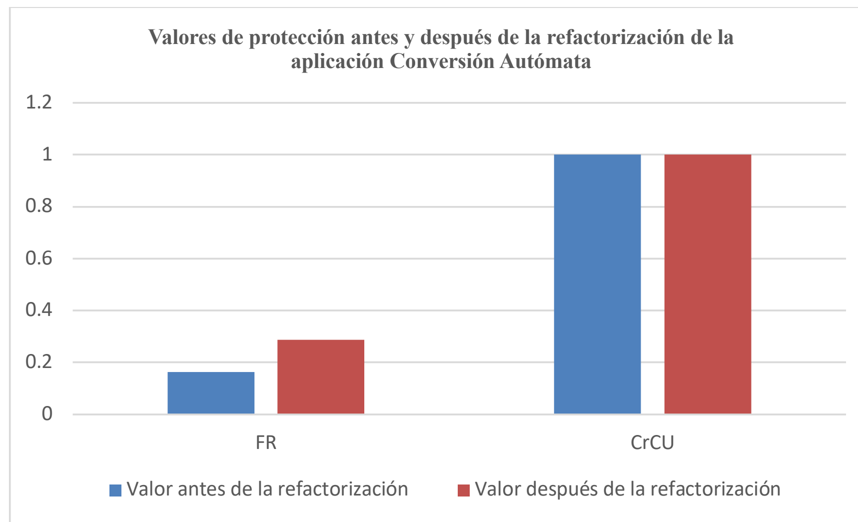


Gráfico 3. Valores de las métricas FR y CrCU antes y después de la re-factorización.

Con respecto a la métrica LCOM* la Tabla 47 muestra los valores obtenidos de la métrica LCOM* antes de la re-factorización. En esta se observa que el 44.4% del total de las clases se encuentran dentro del valor deseado de la métrica que en este caso es 0. La Tabla 48 muestra los valores obtenidos de la métrica LCOM* después de la re-factorización. En el resultado se puede observar que el 76.4% del total de las clases cumplen con el valor deseado de la métrica LCOM*. Con esto se concluye que se disminuyó la carencia de cohesión en un 32%.

Tabla 47. Valores obtenidos de la métrica LCOM* antes de la re-factorización de la aplicación Conversión Autómata.

Clase	Valor antes de la re-factorización
NodoAutomata	1.0500
ConstruccionAFD	0.9333
MovimientoAutomata	0.5714286
ConstruccionAFN	0.0
Conversor	0.0
Cliente	0.0
Automata	0.8646
ClaseEstado	0.625
AlgoritmoMove	0.0

Tabla 48. Valores obtenidos de la métrica LCOM* después de la re-factorización de la aplicación *ConversionAutomata*

Clase	**Valor después de la re-factorización**
Cohes_Coher_ConstruccionAFD24	0.0
NodoAutomata	0.9027
Cohes_Coher_ConstruccionAFD32	0.0
Coher_AlgoritmoMove2	0.0
Coher_AlgoritmoMove3	0.0
MovimientoAutomata	0.57142859
Cohes_Coher_ConstruccionAFD22	0.0
ConstruccionAFN	0.0
Cohes_Coher_ConstruccionAFD23	0.0
Conversor	0.0
Cliente	0.0
Automata	0.8646
Cohes_Coher_ConstruccionAFD21	0.0
ClaseEstado	0.625
Coher_ConstruccionAFD1	0.0
Coher_AlgoritmoMove1	0.0
Cohes_Coher_ConstruccionAFD31	0.0

De esta manera se demuestra que todas las métricas mejoraron su valor debido a que la mayoría de las clases de la aplicación del *ConversionAutomata* después de la re-factorización cuentan con una responsabilidad y la correcta división entre atributos y métodos.

6.2 Reducción de dependencias entre clases de objetos

Convención de nombres

La convención de nombres que se muestra en la Figura 29, es utilizada a través de toda la evaluación del método para reducir las dependencias entre clases de objetos.

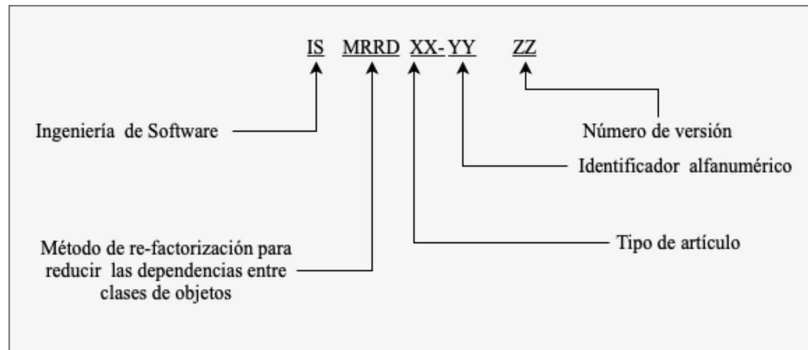


Figura 29. Convención de nombres.

Tipo de Artículo

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

Programas de Control

ISMRRD01 - YYZZ Módulos de programa

Módulos de programa

ISMRRD02 - YYZZ Programas de control, utilerías, ordenadores, entre otros.

Documentación de pruebas

ISMRRD03 - YYZZ Plan de pruebas.
ISMRRD04 - YYZZ Especificación de diseño de pruebas.

Plan de pruebas

1.- Plan de prueba: ISMRRD03 – 01: Plan de pruebas para la validación del correcto funcionamiento del método de re-factorizar para reducir las dependencias entre clases de objetos.

2.- Introducción

El método de re-factorización de arquitecturas de software legado con dependencias entre clases tiene la función de analizar si existen dependencias entre clases, con el fin de verificar que se tenga un valor deseable del factor de acoplamiento (COF). De lo contrario, realizar el proceso de re-factorización mediante la implantación del patrón de diseño *Mediator* para reducir los canales de comunicación y, por lo tanto, reducir la dependencia entre clases.

3.- Artículos de prueba. Los artículos para probar son los siguientes:

3.1.- Módulos de programa. Los módulos de programa a ser probados serán identificados como se muestra a continuación:

Tabla 49. Módulos de programa.

Sistema	Función	No.
Método de Re-factorización para reducir las dependencias entre clases de objetos.	Subsistema de evaluación de dependencia entre clases, mediante la localización de canales de comunicación de éstas.	ISMRRD01 - 01
Marco de Métricas de Calidad de Arquitecturas Orientadas a Objetos	Cálculo de la métrica de acoplamiento COF	ISMRRD01 - 02

3.2 Procedimientos de control de tareas.

Tabla 50. Control de tareas.

Sistema	Función	No.
Programa de aplicación	Localización del paquete al que pertenece la clase.	ISMRRD02 - 01
Programa de aplicación	Localización de las importaciones de la clase.	ISMRRD02 - 02
Programa de aplicación	Localización del nombre de la clase.	ISMRRD02 - 03
Programa de aplicación	Localización de la clase padre de la clase (si existiera).	ISMRRD02 - 04
Programa de aplicación	Localización de atributos de la clase.	ISMRRD02 - 05
Programa de aplicación	Localización de métodos de la clase	ISMRRD02 - 06
Programa de aplicación	Localización de la información de los métodos de clase, tales como: calificador de alcance, si es abstracto, si es estático, nombre, parámetros, cuerpo.	ISMRRD02 - 07
Programa de aplicación	Verificación de existencia de relaciones de dependencia entre clases.	ISMRRD02 - 08
Programa de aplicación	Cálculo de la métrica COF	ISMRRD02 – 09

Programa de aplicación	Método de re-factorizar para reducir las dependencias entre clases.	ISMRRD02 - 10
Programa de aplicación	Llenado de las plantillas ST con código refactorizado.	ISMRRD02 - 11
Programa de aplicación	Compilación del código refactorizado.	ISMRRD02 – 12
Programa de aplicación	Verificación del funcionamiento del código refactorizado.	ISMRRD02 – 13

4.- Características a ser probadas.
La siguiente lista describe las características que deben ser probadas.

Tabla 51. Características por probar.

Diseño de Prueba No. de especificación	Descripción
ISMRD04 - 01	Cálculo de métricas de acoplamiento.
ISMRRD04 – 02	Método de re-factorizar para reducir las dependencias entre clases de objetos.

5.- Características a no ser probadas.

5.1. Los casos de prueba no incluirán todas las posibles construcciones sintácticas y combinaciones de éstas, para código escrito en lenguaje "Java".
5.2. El método no indicará si el código de entrada se encuentra libre de errores.
5.3. No se pretende comprobar que la totalidad del proceso de reingeniería para reuso es automática. Es necesario cierto nivel de intervención del experto en el dominio o el experto en programación.
5.4. Así mismo, no se pretende comprobar la interfaz del sistema.

6.- Enfoque.
La realización de los casos de prueba y la actividad de ejecución de pruebas estará asistida por la alumna sustentante de esta tesis de maestría, Marisol Ramírez Cruz. Esto ayudará para asegurar que las pruebas representan efectivamente el desarrollo y uso del sistema SR2 completo.

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

6.2 Pruebas de re-factorización. La validación del método de re-factorización de arquitecturas de software legado con dependencia entre clases, es decir que no se tengan

valores de la métrica COF cercanos a los deseados y de ser necesario reducir la dependencia mediante la implantación del patrón de diseño *Mediator*. La validación se realizará mediante la ejecución del código original contra la ejecución del código refactorizado. Comprobando, que, bajo las mismas entradas, ambos sistemas deberán comportarse de la misma manera y ofrecer los mismos resultados.

6.3 Pruebas de calidad. Las pruebas de calidad incluyen la aplicación de la métrica COF para la medición comparativa del código original contra el código orientado a objetos obtenido del proceso de re-factorización.

7.- Criterio Pasa / No-pasa de casos de prueba.
Para los casos de prueba del proceso de análisis del código fuente en java, el criterio pasa / no-pasa se realizará mediante la comparación del análisis y la obtención de información manual contra el análisis y la obtención de información de manera automática. En ambos casos se deberá obtener la misma información, comprobándose además la generación correcta de la lista de archivos y de clasificadores (clases e interfaces) requeridas por el proceso de re-factorización.

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

Para los casos de prueba de reducción de dependencia entre clases, el criterio pasa / no-pasa será mediante la comparación del resultado obtenido por cálculo manual de métricas contra el resultado obtenido de manera automática, ambos deberán ser los mismos.

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

9.- 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.

Especificación del diseño de pruebas

1.- Diseño de Prueba: ISMRRD04 – 01. Método de re-factorización para reducir las dependencias entre clases de objetos.

2.- Características a ser probadas.

2.1.- En esta prueba se evaluará el correcto funcionamiento del método de re-factorización al implantar el patrón de diseño mediator para reducir las dependencias entre clases de objetos.

3.- Refinamiento del enfoque.

El objetivo es evaluar la correcta ejecución del método de re-factorización al implantar el patrón de diseño mediator para reducir las dependencias entre clases de objetos.

Antes de realizar cada caso de prueba, el sistema bajo estudio 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 re-factorización de código tomará el o los archivos respectivos y realizará un reconocimiento léxico y sintáctico, generando las estructuras de datos en memoria con la información necesaria para efectos de la re-factorización.

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

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

1.- Diseño de Prueba: ISMRRD04 – 02. Cálculo de métricas de calidad

2.- Características a ser probadas.

2.1.- En esta prueba se evaluará el correcto cálculo de la métrica COF).

3.- Refinamiento del enfoque.

El objetivo es evaluar el correcto funcionamiento de la métrica COF al evaluar un sistema de software de entrada. Antes de realizar cada caso de prueba, este sistema) será compilado y ejecutado previamente en un compilador para el lenguaje Java, con objeto de corroborar que su construcción está o no correctamente escrita. Posteriormente, el Marco Orientado a Objetos para el cálculo de métricas tomará este código y realizará un reconocimiento léxico y sintáctico, para generar la estructura de datos con la información requerida para estos cálculos.

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

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

Especificación de casos de prueba

1.- Caso de Prueba: ISMRRD05 - 01.

2.- Artículos de Prueba: *LibraryAdministration*

LibraryAdministration es un sistema desarrollado en lenguaje Java que tiene como objetivo realizar la administración automática de una librería.

a) Características a probar, del proceso de análisis.

Tabla 52. Características por probar del proceso del cálculo de métricas de calidad.

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRRD04 – 02	Cálculo correcto de la métrica COF.

3.- Especificación de entrada

Las entradas al proceso de cálculo de métrica de calidad son:

1. Las clases pertenecientes a la aplicación *LibraryAdministration* compilado y probado.

4.- Especificación de salida

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

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

1.- Caso de Prueba: ISMRRD05 - 02.

2.- Artículos de Prueba: *LibraryAdministration*

LibraryAdministration es un sistema desarrollado en lenguaje Java que tiene como objetivo realizar la administración automática de una librería.

a) Características a probar, del método de re-factorizar para mejorar la modularidad.

Tabla 53. Características por probar del proceso de re-factorización.

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRRD04 – 01	Comportamiento del sistema después de la re-factorización
ISMRRD04 – 01	Cálculo de las métricas para verificar si hubo una mejora de la modularidad.

3.- Especificación de entrada

Las entradas al proceso de re-factorización son:

1. Las clases pertenecientes a la aplicación *LibraryAdministration* compilado y probado.

4.- Especificación de salida

A la salida del proceso de reestructura se deberá tener:

1. Las clases pertenecientes a la aplicación *LibraryAdministration* refactorizadas con reducción de deuda técnica originado por código desagradable que se caracteriza por tener una modularidad incorrecta en suficiencia y completitud en diferentes entidades de software orientadas a objetos.

1.- Caso de Prueba: ISMRRD05 - 03.

2.- Artículos de Prueba: *ConversionAutomata*

El sistema *ConversionAutomata* es un sistema desarrollado en lenguaje java que tiene como objetivo convertir una expresión regular a un autómata finito determinista.

a) Características a probar, del proceso de análisis.

Tabla 54. Características por probar del proceso del cálculo de métricas de calidad.

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRRD04 – 02	Cálculo correcto de la métrica COF.

3.- Especificación de entrada

Las entradas al proceso de cálculo de métrica de calidad son:

1. Las clases pertenecientes a la aplicación *ConversionAutomata* compilado y probado.

4.- Especificación de salida

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

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

1.- Caso de Prueba: ISMRRD05 - 04.

2.- Artículos de Prueba: *ConversionAutomata*

El sistema *ConversionAutomata* es un sistema desarrollado en lenguaje java que tiene como objetivo convertir una expresión regular a un autómata finito determinista.

a) Características a probar, del método de re-factorización para mejorar la modularidad.

Tabla 55. Características por probar del proceso de re-factorización.

Diseño de Prueba No. de especificación	Característica por Ejercitar
ISMRRD04 – 01	Comportamiento del sistema después de la re-factorización
ISMRRD04 – 01	Cálculo de las métricas para verificar si hubo una mejora de la modularidad.

3.- Especificación de entrada

Las entradas al proceso de re-factorización son:

1. Las clases pertenecientes a la aplicación *ConversionAutomata* compilado y probado.

4.- Especificación de salida

A la salida del proceso de reestructura se deberá tener:

1. Las clases pertenecientes a la aplicación *ConversionAutomata* refactorizadas con reducción de deuda técnica originado por tener alto grado de dependencias entre clases.

Ejecución del plan de pruebas

1.- Caso de Prueba: ISMRRD05 - 01.

2.- Artículos de Prueba: *LibraryAdministration*

En la Tabla 56 se enlistan las características a probar, del proceso de cálculo de métricas de calidad.

Tabla 56. Características a probar del proceso del cálculo de métricas de calidad.

Diseño de Prueba No. de especificación	**Característica a Ejercitar**
ISMRRD04 – 02	Cálculo correcto de la métrica COF.

Dada la Figura 30 denominada del *LibraryAdministration*, que representa un sistema desarrollado en lenguaje java, el cual tiene como objetivo realizar la administración de una biblioteca. Se procede a realizar el caso de prueba **ISMRRD05 - 01**.

Figura 30. Arquitectura de clases de la aplicación *LibraryAdministration* antes de la re-factorización.

Cálculo correcto de las métricas de Factor de acoplamiento COF

Cálculo de la métrica COF (Factor de Acoplamiento)

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} es_cliente\left(C_i, C_j\right)\right]}{TC^2 - TC}$$

Donde:

TC es el total de clases
TC2 - TC es el máximo número de acoplamientos en un sistema con TC clases.

$$es_cliente(Cc,Cs) = \begin{cases} 1 \text{ si y solo si } Cc \Rightarrow Cs \ \wedge Cc \neq Cs \\ 0 \text{ en caso contrario} \end{cases}$$

La relación cliente-servidor (Cc => Cs) significa que Cc (la clase cliente) contiene al menos una referencia no basada en la herencia a un método de la clase Cs (clase servidora). El cálculo manual y automático de la métrica COF, correspondiente a la arquitectura de la Figura 30 se muestra en Tabla 57.

Tabla 57. Cálculo manual y automático de la métrica COF de *LibraryAdministration*.

Cálculo manual	Cálculo automático
$COF = \frac{4}{210} = 0.019$	COF: 0.019

En la Tabla 57 se observa que el valor de la métrica COF obtenido de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

1.- Caso de Prueba: ISMRRD05 - 02.

2.- Artículos de Prueba: *LibraryAdministration*

En la Tabla 58 se enlistan las características a probar del proceso de re-factorización en la aplicación *LibraryAdministration*.

Tabla 58. Características por probar del proceso de re-factorización.

Diseño de Prueba No. de especificación	**Característica por Ejercitar**

ISMRRD04 – 01	Comportamiento del sistema después de la re-factorización
ISMRRD04 – 01	Cálculo de las métricas para verificar si hubo una mejora de la modularidad.

Una vez realizado el proceso de re-factorización automática al sistema de *LibraryAdministration* en el que se tuvo como entrada la arquitectura mostrada en la Figura 30, se tiene como resultado la arquitectura mostrada en la Figura 31.

Figura 31. Arquitectura de clases de la aplicación *LibraryAdministration* después de la re-factorización.

Comportamiento del sistema después de la re-factorización

Para comprobar que la aplicación del *LibraryAdministration* mantiene el mismo comportamiento después de ser sometida al método de re-factorización se realizaron pruebas de las funciones proporcionadas por el sistema. A continuación, se muestra una prueba que consiste

en la inserción de datos de un libro al sistema antes y después de la re-factorización. La Tabla 59 muestra los datos ingresados a la prueba.

Tabla 59. Datos ingresados en la prueba

ISBN	7075502
Signatura	978
Titulo	ALGEBRA BALDOR
Autor	Aurelio Baldor
Materia	Matemáticas
Editorial	Patria

La Tabla 60 muestra los resultados obtenidos por la aplicación *LibraryAdministration* antes y después de la re-factorización, se puede observar que el comportamiento se mantuvo, por lo que se comprueba que el método de re-factorización no altera el compartimento de la aplicación.

Tabla 60. Inserción de datos de un libro en la aplicación *LibraryAdministration* antes y después de la re-factorización

Resultados antes de la re-factorización	Resultados después de la re-factorización
GESTIÓN DE LIBROS ----------------- 1. Añadir libro. 2. Eliminar libro. 3. Modificar libro. 4. Mostrar libros. 0. Volver al menú principal. ¿Qué desea hacer? 1 Introduzca el ISBN del libro que desea añadir. 6075502 Introduzca la signatura. 978 Introduzca el título. ALGEBRA BALDOR Introduzca el autor. Aurelio Baldor Introduzca la materia. Matemáticas Introduzca la editorial. Patria El libro con el ISBN 6075502 se ha creado correctamente. GESTIÓN DE LIBROS ----------------- 1. Añadir libro. 2. Eliminar libro. 3. Modificar libro. 4. Mostrar libros. 0. Volver al menú principal. ¿Qué desea hacer?	GESTIÓN DE LIBROS ----------------- 1. Añadir libro. 2. Eliminar libro. 3. Modificar libro. 4. Mostrar libros. 0. Volver al menú principal. ¿Qué desea hacer? 1 Introduzca el ISBN del libro que desea añadir. 6075502 Introduzca la signatura. 978 Introduzca el título. ALGEBRA BALDOR Introduzca el autor. Aurelio Baldor Introduzca la materia. Matemáticas Introduzca la editorial. Patria El libro con el ISBN 6075502 se ha creado correctamente. GESTIÓN DE LIBROS ----------------- 1. Añadir libro. 2. Eliminar libro. 3. Modificar libro. 4. Mostrar libros. 0. Volver al menú principal. ¿Qué desea hacer?

Comprobar la reducción de las dependencias entre las clases de objetos

Para comprobar que hubo reducción en las dependencias entre clases de objetos *LibraryAdministration* , dicha aplicación se somete al cálculo de la métrica COF antes y después de la re-factorización, con el fin de comparar su grado de mejora con base en a estas métricas. A continuación, se muestra una de las pruebas realizadas al sistema. Esta consiste en la inserción de datos de un libro al sistema antes y después de la re-factorización (Tabla 59). La Tabla 61 muestra la comparación de los valores resultantes del cálculo de la métrica.

Tabla 61. Valores obtenidos de la métrica COF antes y después de la re-factorización de la aplicación *LibraryAdministration*.

Métrica	**Valor antes de la re-factorización**	**Valor después de la re-factorización**
COF	0.019	0.007

La implantación del patrón de diseño *Mediator* reduce las dependencias entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador. En la Tabla 61 se observa que se redujo el acoplamiento en el sistema refactorizado con respecto al sistema de entrada. El nuevo valor de la métrica indica que se redujo un 1.2 % la dependencia entre clases esto debido a que la mayor parte de la comunicación ahora fluye a través de la clase mediadora.

De esta manera se demuestra que toda la arquitectura redujo la dependencia entre clases. Esto es debido a que la mayoría de las clases de la aplicación del *LibraryAdministration* después de la re-factorización cuentan con un mediador de la comunicación entre clases, por lo tanto, la se redujo la dependencia del sistema.

1.- Caso de Prueba: ISMRRD05 - 03.

2.- Artículos de Prueba: *ConversionAutomata*

En la Tabla 62 se enlistan las características a probar, del proceso de cálculo de métricas de calidad.

Tabla 62. Características por probar del proceso del cálculo de métricas de calidad.

Diseño de Prueba No. de especificación	**Característica por Ejercitar**
ISMRRD04 – 02	Cálculo correcto de la métrica COF.

Dada la Figura 32 denominada del *ConversionAutomata*, que representa un sistema desarrollado en lenguaje java, el cual tiene como objetivo realizar la administración de una biblioteca. Se procede a realizar el caso de prueba **ISMRRD05 - 01**.

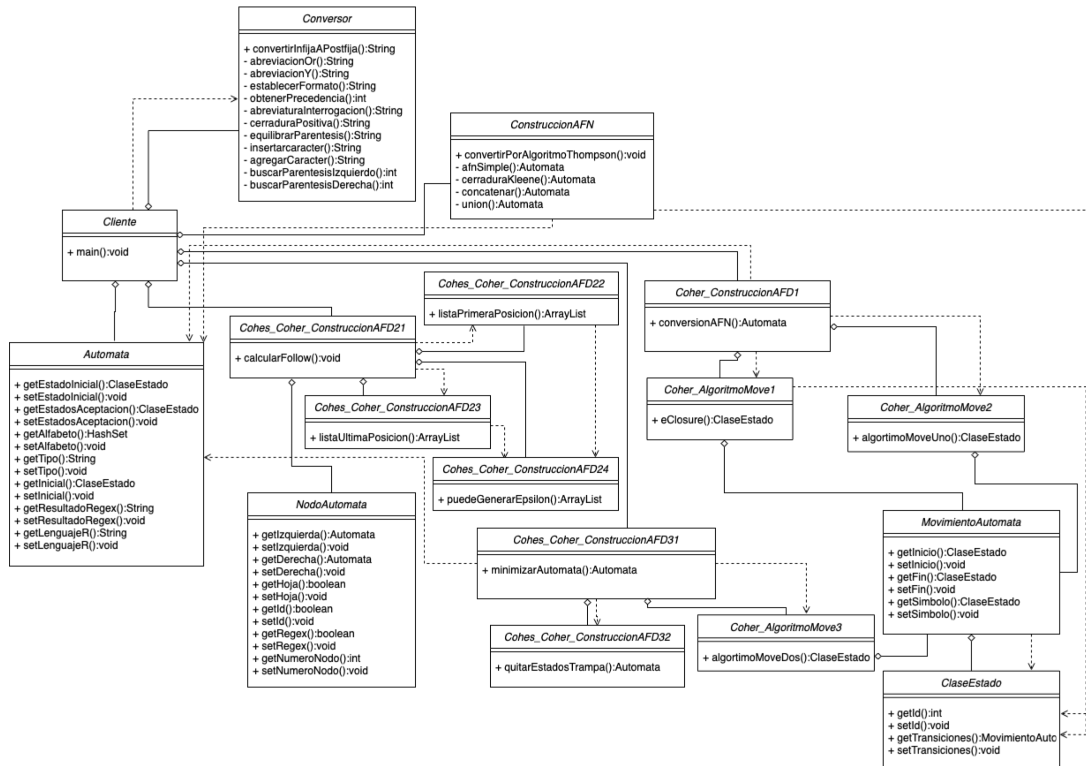


Figura 32. Arquitectura de clases de la aplicación *ConversionAutomata* antes de la re-factorización.

Cálculo correcto de las métricas de Factor de acoplamiento COF

Cálculo de la métrica COF (Factor de Acoplamiento)

$$COF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} es_cliente(C_i, C_j) \right]}{TC^2 - TC}$$

Donde:

TC es el total de clases
TC2 - TC es el máximo número de acoplamientos en un sistema con TC clases.

$$es_cliente(Cc,Cs) = \begin{cases} \text{1 si y solo si } Cc \Rightarrow Cs \ \wedge Cc \neq Cs \\ \text{0 en caso contrario} \end{cases}$$

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

El cálculo manual y automático de la métrica COF, correspondiente a la arquitectura de la Figura 32 se muestra en Tabla 63.

Tabla 63. Cálculo manual y automático de la métrica COF de *ConversionAutomata*.

Cálculo manual	Cálculo automático
$COF = \frac{15}{272} = 0.055$	COF: 0.055

En la Tabla 63 se observa que el valor de la métrica COF obtenido de manera manual y automática fue el mismo por lo que el cálculo de la métrica de forma automática se efectuó de forma correcta.

1.- Caso de Prueba: ISMRRD05 - 04.

2.- Artículos de Prueba: *ConversionAutomata*

En la Tabla 64 se enlistan las características a probar del proceso de re-factorización en la aplicación *ConversionAutomata*.

Tabla 64. Características por probar del proceso de re-factorización.

Diseño de Prueba No. de especificación	**Característica por Ejercitar**
ISMRRD04 – 01	Comportamiento del sistema después de la re-factorización
ISMRRD04 – 01	Cálculo de las métricas para verificar si hubo una mejora de la modularidad.

Una vez realizado el proceso de re-factorización automática al sistema de *ConversionAutomata* en el que se tuvo como entrada la arquitectura mostrada en la Figura 32, se tiene como resultado la arquitectura mostrada en la Figura 33.

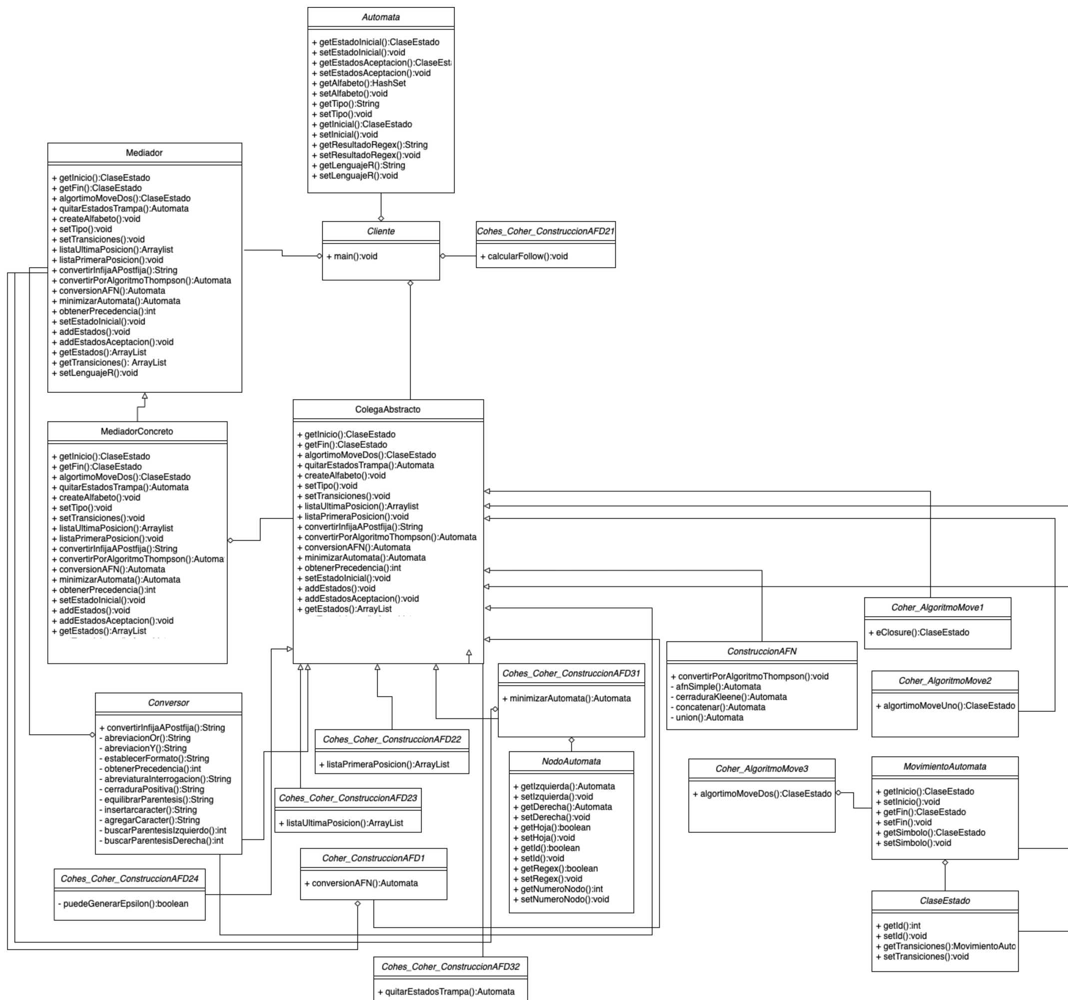


Figura 33. Arquitectura de clases de la aplicación *ConversionAutomata* despúes de la re-factorización.

Comportamiento del sistema después de la re-factorización

Para comprobar que la aplicación del *ConversionAutomata* mantiene el mismo comportamiento después de ser sometida al método de re-factorización se realizaron pruebas de las funciones proporcionadas por el sistema. A continuación, se muestra una prueba que consiste en la inserción de datos de un libro al sistema antes y después de la re-factorización. La Tabla 65 muestra los datos ingresados a la prueba.

Tabla 65. Datos ingresados en la prueba

Expresión regular
ab*ba

La Tabla 66 muestra los resultados obtenidos por la aplicación *ConversionAutomata* antes y después de la re-factorización, se puede observar que el comportamiento se mantuvo, por lo que se comprueba que el método de re-factorización no altera el compartimento de la aplicación.

Tabla 66. conversión de una expresión infija a postfija.

Resultados antes de la re-factorización
Expresion ingresada ab*ba Infija a Postfija: ab*.b.a.
Resultados después de la re-factorización
Expresion ingresada ab*ba Infija a Postfija: ab*.b.a.

Comprobar la reducción de las dependencias entre las clases de objetos

Para comprobar que hubo reducción en las dependencias entre clases de objetos *ConversionAutomata* , dicha aplicación se somete al caculo de la métrica COF antes y después de la re-factorización, con el fin de comparar su grado de mejora con base en a estas métricas. A continuación, se muestra una de las pruebas realizadas al sistema. Esta consiste en la inserción de datos de un libro al sistema antes y después de la re-factorización (Tabla 65). La Tabla 67 muestra la comparación de los valores resultantes del cálculo de la métrica.

Tabla 67. Valores obtenidos de la métrica COF antes y después de la re-factorización de la aplicación *ConversionAutomata*.

Métrica	Valor antes de la re-factorización	Valor después de la re-factorización
COF	0.055	0.042

La implantación del patrón de diseño *Mediator* reduce las dependencias entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador. En la Tabla 67 se observa que se redujo el acoplamiento en el sistema refactorizado con respecto al sistema de entrada. El nuevo valor de la métrica indica que se redujo un 1.3 % la dependencia entre clases esto debido a que la mayor parte de la comunicación ahora fluye a través de la clase mediadora.

De esta manera se demuestra que toda la arquitectura redujo la dependencia entre clases. Esto es debido a que la mayoría de las clases de la aplicación del *LibraryAdministration* después de la re-factorización cuentan con un mediador de la comunicación entre clases, por lo tanto, se redujo la dependencia del sistema.

Capítulo 7. Conclusiones y trabajos futuros

En este capítulo se muestran las conclusiones, los objetivos alcanzados, las aportaciones realizadas, los trabajos futuros, así como algunas recomendaciones referentes al trabajo realizado en el desarrollo de este trabajo de investigación.

Conclusiones:

El campo de la re-factorización es muy amplio y con oportunidad para agregar más investigaciones ya que es una forma de garantizar la mejora de la calidad del código y la reducción de costos en mantenimiento. Con los métodos de re-factorización realizados en esta investigación se garantiza que los sistemas de entrada mejoren la modularidad en suficiencia y completitud y que se reduzcan las dependencias entre clases de objetos. Estos métodos pueden aplicarse de forma independiente en caso de que el usuario así lo requiera.

Con base en la literatura consultada se observó que se han realizado diversas investigaciones con respecto a la identificación de código desagradable y algunos referentes a la re-factorización de código. Sin embargo, en algunos estudios el enfoque se centra en la identificación, y algunos pocos implementan métodos de re-factorización. No se encontró investigación que implementara algún método automático para la mejora de la modularidad y reducción de dependencias entre clases.

Dentro del marco conceptual se mostraron los fundamentos teóricos y conceptuales que fueron la base para la investigación de este trabajo de tesis, del cual se observa que los patrones de diseño pueden ser aplicables para solucionar diversos problemas que existen en el desarrollo de software. Un aspecto importante para considerar la aplicación de algún patrón de diseño es el análisis del problema para poder identificar qué patrón de diseño aplicar y cómo hacerlo para no caer en la generación de más problemas de código desagradable en los sistemas y así garantizar la reutilización de los módulos del que están compuestos.

Con relación a las herramientas utilizadas para el uso de métricas en el proceso de re-factorización, se tiene un valor que ofrece una pauta para conocer la calidad en que se encuentra un sistema en los diferentes niveles en que este sistema puede ser evaluado. En esta investigación las métricas utilizadas permiten la medición de los sistemas de entrada y salida para comprobar los resultados y verificar si hubo mejoras después de realizar el proceso de re-factorización.

El análisis y el diseño determinaron los elementos que intervendrían en la solución del problema de esta investigación y los elementos que serían necesarios para la implementación de

los métodos de re-factorización como la implementación del patrón de diseño *"Mediator",* el flujo que siguieran los algoritmos creados para realizar el proceso de re-factorización y con esto cumplir con el objetivo de la investigación.

Con respecto a las pruebas del sistema para la mejora de su modularidad, en la Tabla 68 se muestra el resumen de los resultados obtenidos en las pruebas mostradas en este documento. Se observa que en algunos casos se obtuvo un mínimo porcentaje debido a que el sistema de mejora de modularidad consistió en dos procesos: el aumento de la coherencia y el aumento de la cohesión, mismos que en la reestructuración afectan el valor resultante de las métricas, sin embargo, los valores de las métricas mejoraron en comparación al primer cálculo de la métrica.

También en algunos valores se observó que no hubo mejora, esto fue debido a que el valor de la métrica se mantuvo y que la re-factorización no afectó estos valores debido a que el valor de las métricas fue el esperado. En general se obtuvo el comportamiento deseado por parte del sistema y en la mayoría de los casos los valores de las métricas mejoraron por lo que se comprobó que el sistema para mejorar la modularidad en suficiencia y completitud es capaz de mejorar sistema de software existentes escritos en lenguaje Java.

Tabla 68. Resumen de los resultados de las pruebas

	Arquitectura 1	**Arquitectura 2**	**Aquitectura 3**
LCOM*	1.1 %	12.12 %	32%
CrCU	0	11.7 %	0
FR	6.35 %	6.35 %	12.45 %

En lo que se refiere a las pruebas realizadas del método de reducción de dependencias entre clases se tienen los resultados que se muestran en la Tabla 69. De esta tabla se observa que en ambas arquitecturas se redujo la dependencia en la arquitectura ingresada. El porcentaje de mejora depende del número de dependencias detectadas en el sistema de entrada. En todas las pruebas realizadas se obtuvo una reducción de las dependencias entre clases de las arquitecturas por lo que se concluye que el sistema es capaz de reducir las dependencias entre clases.

Tabla 69. Resumen de los resultados de las pruebas

	ARQUITECTURA 1	**ARQUITECTURA 2**
COF	1.2	1.3

Con respecto al sistema Sr2-refactoring para obtener una mejora completa del sistema de entrada y hacer uso de los métodos de re-factorización que este contiene, los métodos deben aplicarse en el siguiente orden: 1) Método de re-factorización para mejorar la protección

modular, 2) Método para aumentar la flexibilidad en sistemas carentes de abstracción, 3) Método de reducción de herencia de implementación, 4) Método para la mejora de la modularidad 5) Método para reducir las dependencias entre clases y 6) Método de separación de interfaces.

Aportaciones

En esta sección se describen las aportaciones que se obtuvieron con el desarrollo y cumplimiento de este trabajo de investigación

Métodos de re-factorización para mejorar su modularidad y reducir las dependencias entre clases de objetos de Java.

- El primer método de re-factorización es capaz de mejorar la modularidad en suficiencia y completitud de sistemas escritos en lenguaje Java mediante la división de las clases por responsabilidades y la relación correcta entre los componentes de una clase, es decir, que todos los métodos de una clase hagan uso de todos los atributos de esta.

- El segundo método es capaz de reducir la dependencia entre clases de objetos mediante la implantación del patrón de diseño *Mediator* que permite dirigir los canales de comunicación de las clases.

Métrica

- Se diseñó e implemento en lenguaje Java la métrica FR para medir el factor de responsabilidades por clase

Extensión a la herramienta de re-factorización denominada SR2-Refactoring

Dos métodos de re-factorización para extender la herramienta y agregar funcionamiento para solucionar tres problemáticas referentes a la mejora de código desagradable: mejorar la modularidad (suficiencia y completitud) y reducir las dependencias entre clases de objetos.

Trabajos futuros

De esta investigación se derivan los trabajos futuros que se mencionan a continuación:

- Mejora del algoritmo para la identificación de parámetros de los métodos, ya que estos pueden enviarse de diversas formas o en su caso ampliar el analizador sintáctico para identificar las formas de envío de parámetros.

- Identificar las responsabilidades a nivel de arquitectura debido a que actualmente en la investigación se tratan las responsabilidades a nivel de clases, y así complementar esta investigación.

- Ampliar a las características de JavaBeans o beans, que son clases simples de Java, y que cumplen con ciertas normas con los nombres de sus propiedades y métodos.

Referencias

Aivosto. (n.d.). *MOOD and MOOD2 metrics*. Aivosto. Retrieved August 6, 2019, from https://www.aivosto.com/project/help/pm-oo-mood.html

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

Barón, N. (2020). *Método de re-factorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existentes*. 121.

Bois, B. Du, Demeyer, S., & Verelst, J. (2004). Refactoring - Improving coupling and cohesion of existing code. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 144–151. https://doi.org/10.1109/WCRE.2004.33

Brito e Abreu, F., Goulão, M., & Esteves, R. (1995). Toward the design quality evaluation of object-oriented software systems. *Proceedings 5th Int'ernatianal Conference Software Quality*, *October*, 44–57. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.6468&rep=rep1&type=pdf

Bryton, S., & Brito e Abreu, F. (2008). Modularity-oriented refactoring. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 294–297. https://doi.org/10.1109/CSMR.2008.4493330

C. Martin, R. (2012). *Código Limpio Manual de Estilo para el Desarrollo Ágil de Software* (p. 463). ANAYA.

C. Martin, R., & Martin, M. (2006). *SWA Agile Principles, Patterns, and Practices in CSharp*. Prentice Hall.

Calabrese, J., Esponda, S., Pasini, A., Boracchia, M., & Pesado, P. (2019). Guía para evaluar calidad de datos basada en ISO/IEC 25012. *XXV Congreso Argentino de Ciencias de La Computación*, 694–706. https://core.ac.uk/download/pdf/301104068.pdf

Cárdenas Robledo, L. A. (2004). *Re-factorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator*. Centro Nacional de Investigación Y Desarrollo Tecnológico.

Dos Santos Neto, B. F., Ribeiro, M., Da Silva, V. T., Braga, C., Pereira De Lucena, C. J., & De Barros Costa, E. (2015). AutoRefactoring: A platform to build refactoring agents. *Expert Systems with Applications*, *42*(3), 1652–1664. https://doi.org/10.1016/j.eswa.2014.09.022

Fields, J., Harvie, S., & Fowler, M. (2010). *Refactoring*. Pearson.

Fowler Martin, B. K. (1994). Refactoring: Improving the Design of Existing Code. In *Lecture Notes in Computer Science: Vol. 857 LNCS*. https://doi.org/10.1007/bfb0020422

Gallardo Vera, S. (2018). *Generador de Servicios Web desde Marcos Orientados a Objetos con el Enfoque de Clases Internas*. Centro Nacional de Investigación y Desarrollo Tecnológico.

Gamma, E., Helm, R., Ralph, J., & Vlissides, J. (2005). Design Patterns. In *Riparia*. https://doi.org/10.1016/b978-012663315-3/50005-8

Jain, V., & Gupta, A. (2015). Lack of conceptual cohesion of methods : A new alternative to Lack of Cohesion of methods. *ACM International Conference Proceeding Series*, *18-20-Febr*, 110–119. https://doi.org/10.1145/2723742.2723753

Joyanes Aguilar, L. (2008). *Fundamentos de Programación. Algoritmos, estructura de datos y*

objetos (Cuarta edi). McGraw Hill.
Khatchadourian, R., & Masuhara, H. (2017). Automated Refactoring of Legacy Java Software to Default Methods. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, 82–93. https://doi.org/10.1109/ICSE.2017.16
Khatchadourian, R. T., Moore, O., & Masuhara, H. (2016). *Towards improving interface modularity in legacy Java software through automated refactoring*. 104–106. https://doi.org/10.1145/2892664.2892681
Khomh, F., Di Penta, M., & Guéhéneuc, Y. G. (2009). An exploratory study of the impact of code smells on software change-proneness. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 75–84. https://doi.org/10.1109/WCRE.2009.28
Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. *IEEE International Conference on Software Maintenance, ICSM*, 350–359. https://doi.org/10.1109/ICSM.2004.1357820
Meyer, B. (1997). Object-oriented software construction. In *Science of Computer Programming* (Segunda ed, Vol. 12, Issue 1). ISE Inc. https://doi.org/10.1016/0167-6423(89)90034-8
Napoli, C., Pappalardo, G., & Tramontana, E. (2013). Using modularity metrics to assist move method refactoring of large systems. *Proceedings - 2013 7th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2013*, 529–534. https://doi.org/10.1109/CISIS.2013.96
Ortiz Gutierrez, O. (2020). *Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos*. Centro Nacional de Investigación y Desarrollo Tecnológico.
Padilla Salgado, P. (2019). *Método de Re-factorización de código java con interfaces y abstracciones incorrectas*. Centro Nacional de Investigación y Desarrollo Tecnológico.
Rathee, A., & 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
Rodríguez, D., & Harrison, R. (n.d.). Medición en la Orientación a Objetos. *Medición Para La Gestión En La Ingeniería de Software*, 1–16.
Santos Castillo, L. E. (2005). *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter*. Centro Nacional de Investigación Y Desarrollo Tecnológico.
Terence, P. (2013). *The definitive ANTLR 4 Reference*.
Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, *35*(3), 347–367. https://doi.org/10.1109/TSE.2009.1
Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. Di, De Lucia, A., & Poshyvanyk, D. (2017). When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering*, *43*(11), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105
Valdés Marrero, M. A. (2004). *Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces*. Centro Nacional de Investigación Y Desarrollo Tecnológico.
Vázquez Escudero, P. J., Moreno García, M. N., & García Peñalvo, F. J. (2001). Métricas Orientadas a Objetos. *Departamento de Informática y Automática*, 2–4.
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

Anexo A. Sustento de métrica FR

Factor de responsabilidades (FR) como escala ordinal

Se ha sugerido ha sugerido que la teoría de la medición debería servir de base para desarrollar, razonar y aplicar métricas de ingeniería de software (Briand, El Emam, & Morasca, 1996). Para clasificar una métrica como de escala ordinal, se deben cumplir los requisitos del axioma de orden débil, los cuales son: que •≥ sea una relación binaria total y transitiva. Las propiedades de transitividad e completitud, son las siguientes (Zuse, 1992):

1. Transitividad: P •≥ P', P' •≥ P'' => P •≥ P''

2. Completitud •≥ P'oP'•≥ P

para todo P', P'' Ɛ P, donde P es un conjunto y •≥ es una relación empírica binaria como "igual o más compleja que".

Supóngase que (P, •≥) es un sistema relacional empírico, donde P es un conjunto contable no vacío y •≥ es una relación binaria en P. Luego existe una función μ: P → ℜ, con: P' •≥ P'' ↔ μ(P') ≥ μ(P''),

para todo P', P'' Ɛ P, sí y sólo sí, •≥ es de orden débil. Si tal homomorfismo existe, entonces, ((P, •≥), (ℜ,≥) μ) es 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. Con respecto a la métrica propuesta se tiene el siguiente sistema relacional empírico:

1. P es el conjunto de funciones con calificador de alcance "public" en una arquitectura de clases.
2. ●≥ es una relación empírica entre arquitecturas de clases la cual describe si existe mayor o igual Factor de responsabilidades.
3. ℜ denota el conjunto de números reales.
4. ≥ "mayor o igual que" es una relación binaria entre números.

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

1. La relación binaria •≥, es de orden débil.

2. P1 •≥ P2 ⇔ FR(Arquitectura1) ≥ FR(Arquitectura2)

Como parte del proceso de comprobación, se aplica la métrica mencionada anteriormente para realizar el cálculo del Factor de Responsabilidades (FR) de las arquitecturas 1, 2 y 3 mostradas en las Figuras 34, 35 y 36 respectivamente. Donde el valor deseable es 1 y el peor valor es 0.

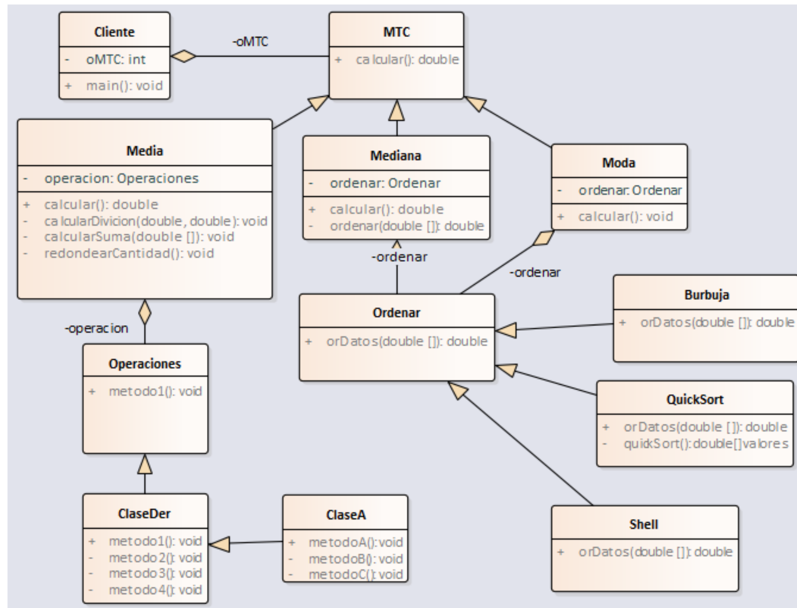


Figura 34. Arquitectura 1.

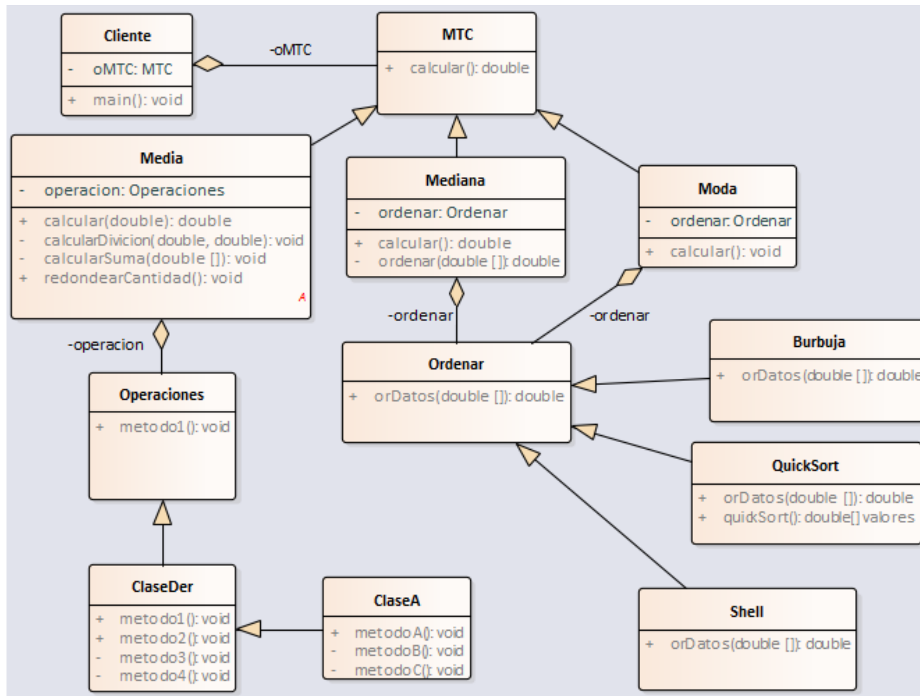


Figura 35. Arquitectura 2.

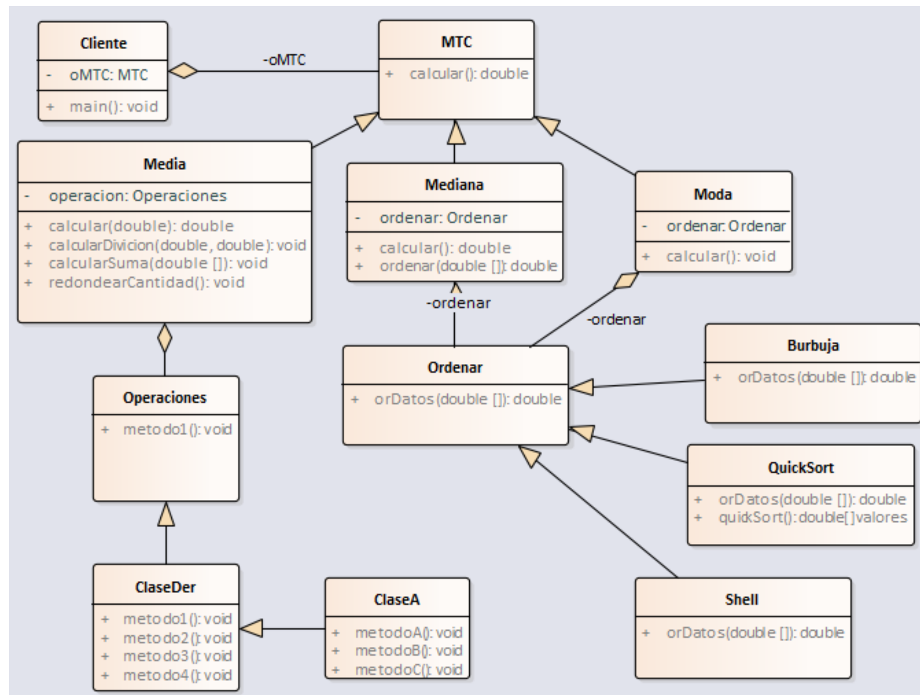


Figura 36. Arquitectura 3.

Del cálculo se obtienen los siguientes resultados:

	Resultados manuales	Resultados implementando la métrica
Arquitectura 1	1.0	<terminated> Cliente (3) [Java Application] C:\ `Factor de responsabilidades: 1.0`
Arquitectura 2	0.75	<terminated> Cliente (3) [Java Application] C:\Pr `Factor de responsabilidades: 0.75`
Arquitectura 3	0.47	<terminated> Cliente (3) [Java Application] C:\ `Factor de responsabilidades: 0.47`

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

Transitividad

P1 •≥ P2, P2 •≥ P3 → P1 •≥ P3
↔
FR (Arquitectura1) •≥ FR (Arquitectura2) & FR (Arquitectura2) •≥ FR(Arquitectura3) → FR (Arquitectura1) •≥ FR (Arquitectura3)

Reflejado en números, se tiene que:

1.0 ≥ 0.75 & 0.75 ≥ 0.47 → 1.0 ≥ 0.4

Como se puede observar, la Arquitectura1 tiene un Factor de responsabilidades mayor a la Arquitectura 2. Este a su vez tiene un Factor de Responsabilidades mayor al de la Arquitectura 3. Concluyendo que la relación binaria •≥ cumple con la propiedad de transitividad.

Relación Total

Si se tienen las arquitecturas uno y dos, se debe poder decir que la Arquitectura 1 "tiene mayor o igual Factor de responsabilidades que" la Arquitectura 2, o viceversa. Es decir:

Arquitectura1 •≥ Arquitectura2 o Arquitectura2 •≥ Arquitectura1
Calculando el FR en las Arquitecturas uno y dos de la Figura 6 y 7 respectivamente, siempre fue posible determinar cuando existía mayor o igual Factor de responsabilidades.

Homomorfismo

Comprobando la segunda condición quedaría de la siguiente manera:

La Arquitectura 1 “tiene mayor o igual Factor de responsabilidades que” la Arquitectura 2

$$\Leftrightarrow$$

$$1 \geq 0.75$$

Conclusión

La relación binaria “tiene mayor o igual Factor de responsabilidades” de la métrica R, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual, se concluye que la métrica es de escala ordinal (Zuse, 1992).