SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Secretaría Académica, de Investigación e Innovación
Dirección de Posgrado, Investigación e Innovación

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

Centro Nacional de Investigación y Desarrollo Tecnológico

Subdirección Académica

Departamento de Ciencias Computacionales

TESIS DE MAESTRÍA EN CIENCIAS

Refactorización de Sistemas Legados de Software, para Equilibrar la Coherencia y Cohesión de su Estructura Interna

presentada por
Lic. Sandro Geovani Vázquez Díaz

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

Director de tesis
Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Julio de 2016.

Cuernavaca, Morelos a 20 de junio del 2016
OFICIO No. DCC/158/2016

Asunto: Aceptación de documento de tesis

C. DR. GERARDO V. GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial del **Lic. Sandro Geovani Vázquez Díaz**, con número de control M14CE021, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis profesional titulado **"Refactorización de sistemas legados de software, para equilibrar la coherencia y cohesión de su estructura interna"** y hemos encontrado que se han realizado todas las correcciones y observaciones que se le indicaron, por lo que hemos acordado aceptar el documento de tesis y le solicitamos la autorización de impresión definitiva.

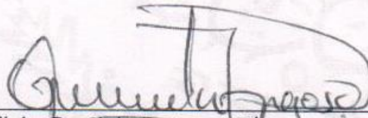
DIRECTOR DE TESIS

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

REVISOR 1

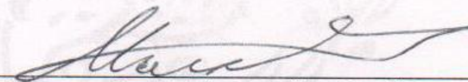
M.C. Humberto Hernández García
Maestro en Ciencias con Especialidad en Sistemas Computacionales
7573641

REVISOR 2



Dra. Olivia Graciela Fragoso Díaz
Doctora en Ciencias en Ciencias de la Computación
7420199

REVISOR 3



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

C.p. Lic. Guadalupe Garrido Rivera - Jefa del Departamento de Servicios Escolares.
Estudiante
Expediente

AMR/lmz

Cuernavaca, Mor., 23 de junio de 2016
OFICIO No. SAC/231/2016

Asunto: Autorización de impresión de tesis

LIC. SANDRO GEOVANI VÁZQUEZ DÍAZ
CANDIDATO AL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
PRESENTE

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"Refactorización de Sistemas Legados de Software, para Equilibrar la Cohesión de su Estructura Interna"**, 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.

ATENTAMENTE
"CONOCIMIENTO Y TECNOLOGÍA AL SERVICIO DE MÉXICO"

DR. GERARDO VICENTE GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO

C.p. Lic. Guadalupe Garrido Rivera.- Jefa del Departamento de Servicios Escolares.
Expediente

GVGR/mcr

Agradecimientos

Agradezco al Consejo Nacional de Ciencia y Tecnología CONACYT por el apoyo económico que recibí, así como la oportunidad para realizar el estudio de posgrado.

Al Centro Nacional de Investigación y Desarrollo Tecnológico CENIDET, por la experiencia y conocimientos adquiridos, por brindarme la oportunidad de terminar mis estudios de maestría.

A mi director de tesis Dr. René Santaolaya Salgado, por su apoyo, consejos, orientación y paciencia brindada durante la realización de esta investigación y mi estancia en el CENIDET.

A mi revisora de tesis Dra. Olivia G. Fragoso Díaz y revisores: Dr. Moisés González García, y M.C. Humberto Hernández García por su orientación y consejos para obtener un buen resultado de esta investigación.

Al Dr. Joaquín Pérez Ortega y todos los profesores del CENIDET con quienes tuve la oportunidad de tomar clases, por transmitírteme sus diversos conocimientos y experiencias.

A todo el personal administrativo del CENIDET quienes siempre estuvieron con su mejor disposición para apoyarme en todos los trámites desde el ingreso hasta la terminación de la maestría.

A mi familia por su apoyo constante e incondicional en las decisiones de mi vida, por todos los buenos momentos que hemos pasado y especialmente quiero agradecer a mi mamá por todos sus consejos y por ser un gran ejemplo a seguir.

A todos mis amigos y amigas del CENIDET, así como amigos y amigas que conocí en Cuernavaca, Morelos, específicamente en las clases de natación e inglés, por brindarme su amistad y confianza, gracias por los buenos momentos que compartimos juntos.

Resumen

Los sistemas Orientados a Objetos demandan del desarrollador gran capacidad de *imaginación, abstracción y creatividad*, para plantear la solución correcta a problemas prácticos de aplicaciones computacionales. Sin embargo estas habilidades son difíciles de ejercer y más difícil de utilizarlas en conjunto. Lo que resulta en arquitecturas de software o unidades de programas que exhiben características de fragilidad, rigidez a cambios o extensiones de requerimientos y baja reusabilidad. Esta situación se manifiesta en altos costos por mantenimiento y por el desarrollo de nuevas aplicaciones. Como parte de la solución a este problema, algunos trabajos de investigación utilizan métodos de refactorización de software de legado. Sin embargo, estos trabajos no mejoran el grado de coherencia o no equilibran los factores de calidad de coherencia y cohesión de las unidades de programa.

Esta tesis presenta un proceso de refactorización y una herramienta automática que implementa este proceso para lograr el equilibrio entre los factores de cohesión y coherencia de software legado desarrollado en lenguaje C ++. Así mismo, se propone una métrica para medir la coherencia utilizando las secuencias interactivas entre los métodos de una clase, y una métrica para medir la cohesión que refleja el grado de relación entre los atributos de una clase con un método único en esa clase.

El proceso de refactorización de esta tesis incluye dos métodos que se aplican en la siguiente secuencia. El objetivo del primero de estos métodos es mejorar la cohesión, se busca que los elementos de la clase estén relacionados para cumplir con un objetivo o meta de valor para un usuario. El segundo método verifica la coherencia y tiene el propósito que las clases tengan una única responsabilidad.

Se realizaron cinco casos de prueba, caso uno: el grado de coherencia no fue mejorado, caso dos: se mejoró el grado de cohesión y coherencia, caso tres: reubicar secuencias interactivas de métodos, caso cuatro: crear herencia por atributos compartidos, caso cinco: cinco clases, una clase es abstracta y no se proporciona la clase cliente. Las pruebas realizadas al proceso de refactorización muestran que es posible equilibrar automáticamente el grado de cohesión y coherencia en las arquitecturas de software legado, lo que mejora la modularidad, la consecución de un mejor nivel de reuso, reducir los tiempos de mantenimiento y por lo tanto los costos de software.

Abstract

Object-oriented systems demand to developers higher capacities of imagination, abstraction and creativity to bring the right solution to practical problems of computing applications. However, these skills are difficult to exercise and even more difficult to use them together. Resulting in software architectures that exhibit characteristics of fragility, rigidity to changes or extend to the requirements, and low reusability. This situation is reflected in high costs for maintenance and development of new applications. As part of the solution to this problem, some research works use legacy software refactoring methods. However, these works do not improve the degree of coherence or they do not balance quality factors of coherence and cohesion of the program units.

This thesis presents a refactoring process and an automatic tool that implements this process to achieve balance between the factors of cohesion and coherence of legacy software developed in C ++ language. Likewise, a metric is proposed to measure the coherence using the interactive sequences between methods of a class, and a metric to measure the cohesion that reflects the degree of relationship between the attributes of a class with a single method in that class.

The refactoring process of this thesis includes two methods, which are applied in the following sequence. The aim of the first of these methods is to improve cohesion, it is intended that the elements of the class are related to meet a goal or target value for a user. The second method verifies the coherence and is intended that classes have a unique responsibility.

Five test cases were performed, case one: the degree of coherence was not improved, case two: the degree cohesion and coherence were improved, case three: relocate interactive sequences of methods, case four: create inheritance shared attributes, case five: five classes, a class is abstract and customer class is not provided. The tests carried to the refactoring process show that it is possible to automatically balance the degree of cohesion and coherence in legacy software architectures, which improves modularity, achieving a better level of reuse, reduce maintenance times and so both software costs

Contenido

Lista de figuras XIV
Lista de tablas............ XVII
1. INTRODUCCIÓN............ XIX
 1.1. Descripción del problema 21
 1.2. Objetivos 23
  1.2.1. Objetivo general 23
  1.2.2. Objetivos específicos............ 23
 1.3. Estado del arte 23
  1.3.1. Antecedentes............ 23
  1.3.2. Trabajos relacionados 27
 1.4. Organización de esta tesis............ 37
2. DESCRIPCIÓN DEL MÉTODO DE SOLUCIÓN 39
 2.1. Coherencia. 40
  2.1.1. Métrica para medir coherencia (Cr). 40
  2.1.2. Ejemplo aplicando métrica de coherencia Cr. 41
 2.2. Cohesión 41
  2.2.1. Métrica para medir cohesión LCOM4............ 42
  2.2.2. Ejemplo aplicando métrica cohesión LCOM4............ 42
  2.2.3. Métrica para medir cohesión LCOM4 –A 44
  2.2.4. Ejemplo aplicando métrica LCOM4-A 45
 2.3. Proceso del método de refactorización. 46
  2.3.1. Método de refactorización para lograr alta cohesión (MRACS). 46
  2.3.2. Método de refactorización para lograr alta coherencia (MRACR) 53
  2.3.3. Método de refactorización de ordenamiento de clases (MROCL). 65
3. DESCRIPCIÓN DE LA HERRAMIENTA 69
 3.1. Diagrama de componentes. 70
 3.2. Interfaz de la herramienta 71
  3.2.1. Pantalla Método de Equilibrio (Alta cohesión y Alta coherencia). 72
  3.2.2. Pantallas: Método para lograr Alta Cohesión y Método para lograr Alta Coherencia. 74
4 PRUEBAS E INTERPRETACIÓN DE RESULTADOS 75
 4.1. Caso de prueba 1: ejemplo de una clase. 76

4.2. Caso de prueba 2: clase con baja cohesión y baja coherencia. 81
4.3. Caso de prueba 3: clase con dos secuencias interactivas. 84
4.4. Caso de prueba 4: tres archivos, una clase, alta cohesión y baja coherencia. 87
4.5. Caso de prueba 5: cuatro clases, una clase abstracta y sin método *main()*. 90
4.6. Resumen de las pruebas. 94
5. CONCLUSIONES Y TRABAJO FUTURO 97
5.1. Aportaciones de la tesis. 98
5.2. Conclusiones. 99
5.3. Trabajo futuro. 100
REFERENCIAS 103
A. ANEXO A ESTUDIO DE MÉTRICAS DE COHESIÓN 107
B. ANEXO B PRUEBAS 139

Lista de figuras

Figura 2.1 Ejemplo para aplicar métrica Cr 41
Figura 2.2 Ejemplo para aplicar métrica LCOM4........ 43
Figura 2.3 Ejemplo 2 para aplicar LCOM4 43
Figura 2.4 Ejemplo para aplicar LCOM4-A........ 45
Figura 2.5 Ejemplo 2 para aplicar LCOM4-A........ 45
Figura 2.6 Diagrama de actividades del MRACS 47
Figura 2.7. Ejemplo de programa en C++ 48
Figura 2.8 Matriz relación de métodos por atributo 49
Figura 2.9 Matriz de métodos relacionados y matriz de atributos de la clase 50
Figura 2.10 Archivo de la nueva clase creada........ 51
Figura 2.11 Ejemplo de archivo creado con la función main() 52
Figura 2.12 Diagrama de actividades del MRACR 54
Figura 2.13 Diagrama de secuencia para aplicar MRACR. 55
Figura 2.14 Ejemplo de matriz de secuencia interactiva de métodos 58
Figura 2.15 Diagrama de clases (representación de invocación). 59
Figura 2.16 Método main() donde se realizan llamadas externas a métodos que se mueven a nuevas clases 60
Figura 2.17 Ejemplo de nueva clase 62
Figura 2.18 Ejemplo de las modificaciones realizadas al método main()........ 65
Figura 2.19 Diagrama de actividades del MROCL 66
Figura 3.1 Diagrama de componentes de la herramienta desarrollada en este trabajo de investigación. 70
Figura 3.2 Pantalla de autenticación del SR2 refactoring. 71
Figura 3.3 Pantalla principal del SR2 Refactoring, resaltando las nuevas pantallas agregadas. 71
Figura 3.4 Pantalla para seleccionar archivos. 72
Figura 3.5 Pantalla Método de Equilibrio (Alta cohesión y Alta coherencia). 72
Figura 3.6 Pantalla Método de Equilibrio después de pulsar en el botón Analizar código 73
Figura 3.7 Pantalla después de aplicar el proceso de refactorización. 73
Figura 3.8 Pantalla del Método para lograr Alta Cohesión. 74
Figura 3.9 Pantalla de Método para lograr Alta Coherencia. 74
Figura 4.1. Pantalla método de refactorización equilibrio. 77
Figura 4.2. Representación de valores de cohesión y coherencia........ 78
Figura 4.3 Diagrama de secuencia del caso de prueba 1 78
Figura 4.4. Resultados de medir cohesión y coherencia 79
Figura 4.5. Resultados después de aplicar el proceso de refactorización. 80
Figura 4.6 Diagrama de secuencia del caso de prueba 2. 82
Figura 4.7 Resultados después de aplicar el proceso de refactorización. 82
Figura 4.8 Diagrama de secuencia después de pasar por el proceso de refactorización. 83

Figura 4.9 Diagrama de secuencia del caso de prueba 3. 85
Figura 4.10 Resultado después de aplicar MRACR 86
Figura 4.11 Diagrama de secuencia después de pasar por el método de refactorización MRACR. 86
Figura 4.12 Diagrama de clases del caso de prueba 4. 88
Figura 4.13 Resultados después de aplicar el proceso de refactorización. 88
Figura 4.14 Diagrama de clases después de pasar el proceso de refactorización. 89
Figura 4.15 Diagrama de clases del caso de prueba 5. 91
Figura 4.16 Resultados después de aplicar el proceso de refactorización. 92
Figura 4.17 Diagrama de clases representando el código después del proceso de refactorización. 93
Figura A.1 Ejemplo de clases 109
Figura B.1 Código completo del caso de prueba 1 140
Figura B.2 Archivo de entrada del caso de prueba 1 140
Figura B.3 Código refactorizado del archivo main.cpp 141
Figura B.4 Código refactorizado del archivo Programming.h 141
Figura B.5 Código refactorizado del archivo Programming.cpp 142
Figura B.6 Archivos generados después de pasar por el proceso de refactorización. 142
Figura B.7 Código completo del caso de prueba 2. 143
Figura B.8 Archivo de entrada del caso de prueba 2 143
Figura B.9 Código refactorizado del archivo main.cpp. 144
Figura B.10 Código refactorizado del archivo Temp.h. 144
Figura B.11 Código refactorizado del archivo Temp.cpp 145
Figura B.12 Código refactorizado del archivo Cohesion82_Temp.h 145
Figura B.13 Código refactorizado del archivo Temp.cpp. 145
Figura B.14 Archivos generados después de pasar por el proceso de refactorización. 146
Figura B.15 Código completo del caso de prueba 3. 148
Figura B.16 Archivo de entrada del caso de prueba 3. 148
Figura B.17 Código refactorizado del archivo main.cpp 149
Figura B.18 Código refactorizado del archivo Operaciones.h. 149
Figura B.19 Código refactorizado del archivo Operaciones.cpp 150
Figura B.20 Código refactorizado del archivo Cohesion710_Operaciones.h 150
Figura B.21 Código refactorizado del archivo Cohesion710_Operaciones.cpp 151
Figura B.22 Archivos generados después de pasar por el proceso de refactorización. 151
Figura B.23 Código completo del archivo main.cpp 152
Figura B.24 Código completo del archivo Figura.h. 152
Figura B.25 Código completo del archivo Figura.cpp 153
Figura B.26 Archivos de entrada del caso de prueba 4. 153
Figura B.27 Código refactorizado del archivo main.cpp 154
Figura B.28 Código refactorizado del archivo Figura.h. 154
Figura B.29 Código refactorizado del archivo Figura.cpp 155
Figura B.30 Código refactorizado del archivo CrareaRectangulo1Base 155

Figura B.31 Código refactorizado del archivo CrareaRectangulo1.h 156
Figura B.32 Código refactorizado del archivo CrareaRectangulo1.cpp 156
Figura B.33 Código refactorizado del archivo CrareaIsosceles2.h 157
Figura B.34 Código refactorizado del archivo CrareaIsosceles2.cpp 157
Figura B.35 Archivos generados después de pasar por el proceso de refactorización. 158
Figura B.36 Código del archivo PruebaConstructor.h 159
Figura B.37 Código del archivo PruebaConstructor.cpp 160
Figura B.38 Archivos de entrada del caso de prueba5 161
Figura B.39 Código refactorizado del archivo Resta.h 161
Figura B.40 Código refactorizado del archivo Resta.cpp 161
Figura B.41 Código refactorizado del archivo Suma.h. 162
Figura B.42 Código refactorizado del archivo Suma.cpp 162
Figura B.43 Código refactorizado del archivo Multiplica.h. 162
Figura B.44 Código refactorizado del archivo Multiplica.cpp 163
Figura B.45 Código refactorizado del archivo Cohesion35_Multiplica.h 163
Figura B.46 Código refactorizado del archivo Cohesion35_Multiplica.cpp 163
Figura B.47 Código refactorizado del archivo Operaciones.h 164
Figura B.48 Código refactorizado del archivo PruebaConstructor.h 164
Figura B.49 Código refactorizado del archivo PruebaConstructor.cpp 165
Figura B.50 Archivos generados después de pasar por el proceso de refactorización. 165

Lista de tablas

Tabla 1.1 Resumen de trabajos relacionados 35
Tabla 4.1 Caso de prueba 1 76
Tabla 4.2 Caso de prueba 2 81
Tabla 4.3 Caso de prueba 3 84
Tabla 4.4 Caso de prueba 4 87
Tabla 4.5 Caso de prueba 5 90
Tabla 4.6 Resumen de los casos de prueba antes de aplicar el proceso de refactorización. 94
Tabla 4.7 Resumen de los casos de prueba después de aplicar el proceso de refactorización. 95
Tabla A.1: Relación de métricas. 135

Capítulo 1

INTRODUCCIÓN

En este capítulo se presenta una perspectiva general del contenido del documento esta tesis incluyendo: introducción al tema de tesis, descripción del problema, objetivo general y objetivos específicos, así como una breve descripción de la organización de la tesis.

Los sistemas de software legado tienen un valor importante para las empresas, ya que han soportado procesos críticos durante su ciclo de vida, además pueden considerarse como la memoria de la organización, al contener de manera implícita la experiencia y evolución que ha tenido a lo largo de su vida. Lamentablemente, la mayoría de este software, al desarrollarse en sus inicios, no fue pensado para ser reusable ni extensible.

El software legado inevitablemente presentará cambios motivados por nuevas exigencias del negocio que generarán cambios en el ambiente, correcciones a fallos encontrados en el funcionamiento, nuevos requerimientos, actualización de los requerimientos existentes, entre otros. Estas necesidades han llevado al concepto de evolución del software, la cual aborda la problemática de la funcionalidad de los sistemas cuando cambia su ambiente. En este sentido, Lehman estableció un conjunto de hipótesis concernientes a cambios de los sistemas [Lehmanand, 1985], de las cuales se desprende que el tiempo de vida de un sistema de software puede ser extendido, manteniéndolo o reestructurándolo.

Un sistema legado no puede ser ni reemplazado ni actualizado excepto a un alto costo. Como respuesta a esto, es que surge el desarrollo del concepto de refactorización que se define como: “el proceso de mejorar el diseño de código existente, cambiando su estructura interna sin alterar su comportamiento externo” [Fowler, 1999]. El objetivo de la refactorización es reducir la complejidad del sistema legado lo suficiente como para ser usado y adaptado a un costo razonable [Demeyer, 2002].

En este documento de tesis, se presenta una investigación en la que se desarrolló una herramienta que tiene el objetivo de medir y equilibrar los valores de cohesión, y coherencia de sistemas legados de software orientado a objetos. Mediante un proceso de refactorización el sistema genera módulos o componentes con autosuficiencia para su reuso y dando como resultado que su mantenimiento sea más fácil y con un menor costo.

1.1. Descripción del problema

El desarrollo de software orientado a objetos se considera un enfoque competente para lograr mayores beneficios de productividad y reuso. Desafortunadamente se dan ejemplos de sistemas Orientados a Objetos (OO) rígidos, frágiles, con poca mantenibilidad o donde el grado de reusabilidad es mínimo.

Los sistemas OO demandan del ser humano una gran capacidad de *imaginación, abstracción y creatividad*, para plantear la solución correcta a problemas prácticos de aplicaciones informáticas. Para el ser humano, estas capacidades son difíciles de ejercer y aún más difícil de usarlas en conjunto.

Cuando el desarrollador de Software carece de experiencia y habilidad en el desarrollo, suelen producirse unidades de programa que resultan poco reusables y costosas en su mantenimiento, debido a que los sistemas de software exhiben, poca coherencia, escasa cohesión, y un alto acoplamiento, dando origen a dependencias entre los módulos y bajo nivel de reuso. Estas dependencias se manifiestan debido a la carencia de un efectivo encapsulamiento.

El encapsulamiento de unidades de programa depende del grado de relación entre funciones y datos (cohesión) y entre las funciones (coherencia). Cuando los atributos o datos de una clase son accedidos desde otras clases, ya sea de manera directa o indirecta, o bien cuando una meta de valor u objetivo del cliente es alcanzado por la interacción de funciones esparcidas en varias clases, entonces tiene un diseño carente de cohesión y carente de coherencia, esto es un diseño con un incorrecto encapsulamiento.

Cuando un módulo no es suficientemente cohesivo, le falta capacidad funcional o no cuenta con los datos suficientes para alcanzar el objetivo requerido. Esto quiere decir que no es auto-suficiente o auto-contenido para alcanzar una meta u objetivo. Es entonces, cuando el módulo necesita de la interacción con otros módulos, tanto para acceder a datos que están ubicados en otros módulos como para extender su capacidad funcional desde otros módulos. Esto incrementa el nivel de acoplamiento entre los módulos involucrados y por tanto las dependencias entre éstos.

Cuando se re-factoriza un módulo para ganar en cohesión, es necesario reorganizar las funciones y los datos entre módulos. De este modo se pueden obtener módulos de grano grueso o módulos de grano fino. Los módulos de grano grueso pueden estar más sobrados que suficientes, tanto en su capacidad funcional como en los datos con los que cuenta, intercambiando datos innecesarios al ser accedidos, esto significa que el módulo atiende a más de una responsabilidad, por lo tanto, su funcionalidad no es coherente. Los módulos de grano fino son incompletos y no suficientes para alcanzar una meta de valor para el usuario, porque necesitan de la interacción con otros módulos para alcanzar la suficiencia funcional, aumentando así el acoplamiento entre módulos y por lo tanto sus dependencias.

Un buen diseño arquitectónico, debe mantener en equilibrio este nivel de granulación, logrando beneficios en costos por mantenimiento y reuso de los módulos o componentes. Un buen diseño de módulos debe lograr auto-suficiencia en estos, entendiéndose que son completos y suficientes, es decir, que no tienen más ni menos capacidad funcional y los datos necesarios para atender un objetivo o meta de valor para un usuario. Se busca que una petición de servicio sea atendida con auto-suficiencia. Un módulo de programa suficientemente cohesivo y coherente cuenta con la capacidad funcional y los datos necesarios para atender una meta u objetivo, sin requerir de la capacidad funcional de otros módulos. Esto es, alta coherencia, alta cohesión y bajo acoplamiento. El acoplamiento entre clases y la carencia de cohesión son dos factores relacionados que influyen en el equilibrio del nivel de granularidad o el encapsulamiento, y existen métricas para medir su influencia en buenos diseños arquitecturales.

El problema radica en que relativamente al grado de conocimientos, experiencia, creatividad, imaginación y abstracción del desarrollador; se pueden obtener malos diseños de arquitecturas de software orientadas a objetos. Lo cual se verá reflejado en su carencia de reuso y dificultades en su mantenimiento.

1.2. Objetivos

1.2.1. Objetivo general

Desarrollar un método para mejorar la modularidad y el nivel de reuso de módulos o unidades de programa orientados. Mediante el balance entre los valores de coherencia y cohesión de módulos y una herramienta que automatice el método.

1.2.2. Objetivos específicos

- Mejorar la estructura interna de software legado con problemas de autosuficiencia equilibrando la coherencia y cohesión.
- Medir el grado de coherencia y cohesión de sistemas de software legado.
- Extender la funcionalidad del “SR2 Refactoring” para dar soporte a un nuevo método de refactorización.

1.3. Estado del arte

1.3.1. Antecedentes

En apoyo a este planteamiento, en el CENIDET se han planteado varios proyectos de refactorización para mejorar arquitecturas de software existente. Entre estos proyectos se cuenta con el desarrollo de herramientas para adaptar interfaces lógicas que no empatan a las necesidades de los clientes, medir el problema de interfaces no utilizadas en marcos de aplicaciones orientados a objetos, y refactorizar para separar esas interfaces no utilizadas, medir el factor de acoplamiento, que produce dependencias, y refactorizar para reducir este factor al mínimo indispensable. Todos estos reunidos en un desarrollo tecnológico denominado SR2-Refactoring, el cual se describe a continuación.

SR2-Refactoring.

El principal desarrollo que antecede a esta tesis es el SR2-Refactoring (Sistema de Reingeniería para Reuso). Este sistema implementa tres métodos de refactorización que permiten mejorar la arquitectura de un software orientado a objetos, escritos en lenguaje C++. El sistema implementa dos 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 refactorización y las métricas se encuentra englobada en tres tesis de maestría del CENIDET, las cuales son:

- Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces. [Valdés, 2004]

- Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter. [Santos, 2005].

- Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator. [Cárdenas, 2004].

El sistema cuenta con seis funciones, los cuales realizan las acciones de refactorización y cálculo de métricas, así como acciones adicionales a los métodos de refactorización, como son la selección y comparación de archivos y el manejo de usuarios.

Sin embargo esta herramienta no cubre la medición del grado de cohesión ni la coherencia de las entidades de software, tales como módulos de programa o componentes, ni tampoco contempla su refactorización. La investigación que se presenta en esta tesis aporta en este sentido, midiendo, evaluando y refactorizando las arquitecturas de software para atender de manera equilibrada sus niveles de cohesión y coherencia, con el objetivo de lograr una mejor autonomía de estas entidades de software auto-suficientes.

Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces [Valdés, 2004].

Una situación que frecuentemente surge en el diseño de marcos orientados a objetos, es el problema de dependencias de interfaces producido por la herencia de interfaz cuando las subclases en realidad no ocupan dichas interfaces. Cuando un marco de aplicaciones orientado a objetos tiene el problema de dependencias de interfaces, su funcionalidad no puede ser reusada de manera separada y tampoco puede ser extendida sin violar otros principios orientados a objetos, como el principio de abierto / cerrado.

En [Valdés, 2004] se definió y creó una métrica orientada a objetos que mide el grado de interfaces que no se ocupan. Así mismo, para resolver este problema, este método de refactorización fue implementado satisfactoriamente en la herramienta SR2-refactoring, aplicando la refactorización de manera automática.

Se presentan casos de estudio para mostrar cómo esta métrica ayuda a detectar cuándo los marcos tienen el problema de dependencias por herencia de interfaces. Con esta información se puede tomar una decisión cuantitativa para encargarse de la resolución del problema. Se muestra en diferentes casos de estudio cómo se lleva a cabo la refactorización de forma automática, utilizando la herramienta implementada y las ventajas que se obtienen al aplicar el proceso sobre marcos orientados a objetos que exhiben este problema.

Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter [Santos, 2005].

En comparación con las técnicas basadas en bibliotecas para reuso, los marcos de aplicaciones orientados a objetos enfatizan el reuso de diseños o micro-arquitecturas de software que dan solución a problemas recurrentes del dominio de aplicaciones, lo cual no impide el reuso de código. El reuso es fomentado por medio de la estabilidad de las interfaces provistas por el marco, al definir componentes genéricos que pueden ser reusados para crear nuevas aplicaciones.

En [Santos, 2005] se trata el problema que se presenta cuando las interfaces de los marcos orientados a objetos no son conformes con las necesidades de los clientes, o como las nuevas aplicaciones las requieren, por lo que es necesario adaptarlas convirtiendo la interfaz del componente legado en la interfaz que el cliente espera. Para la conversión de dicha interfaz se aplica el patrón de diseño Adapter el cual permite que dos interfaces trabajen juntas aunque no sean compatibles en tipo (ya que a veces una interfaz no puede ser reusada debido a que no concuerda con el tipo que está usando la aplicación) o número de parámetros de la interfaz. Una vez implementado el patrón en el método de refactorización de interfaces incompatibles, en el SR2-Refactoring, los clientes llaman a las operaciones por medio de un Adaptador, el cual llama a las operaciones de la interfaz adaptada para ser usadas por el cliente, mejorando de esta forma el reuso de componentes de software.

Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator [Cárdenas, 2004].

Numerosas empresas cuentan con software legado que contiene años de conocimiento, reglas de negocio y experiencias obtenidas al trabajar en aplicaciones de dominios específicos. Este software representa recursos valiosos que podrían utilizarse a futuro para construir nuevas aplicaciones. Los marcos de aplicaciones orientados a objetos se consideran un enfoque competente para lograr mayores beneficios de productividad y reuso. Desafortunadamente muchos marcos exhiben un alto grado de acoplamiento debido a muchas interfaces o canales de comunicación entre clases, lo cual impide su reuso y dificulta su mantenimiento. Para resolver el problema, en el proyecto de [Cárdenas. 2004], se presenta un método de refactorización para reducir el acoplamiento entre clases de marcos de aplicaciones orientados a objetos, incorporando a su arquitectura una estructura dirigida por el patrón de diseño 'Mediator'. La idea es que en lugar de que las clases estén acopladas porque se comunican entre sí, se comuniquen a través de un módulo intermediario, cuya finalidad sea la de realizar dicha comunicación de manera indirecta.

Este algoritmo fue implementado en el sistema SR2-refactoring. El SR2-refactoring mide y evalúa el problema y realiza la refactorización de manera automática. La implementación incluye el cálculo de la métrica para medir los niveles de acoplamiento (Métrica del Factor de

Acoplamiento COF) entre las clases, antes y después de haber aplicado el proceso de refactorización. Se presenta una evaluación experimental utilizando distintos casos de prueba de marcos de aplicaciones orientados a objetos que se caracterizan por tener el problema de acoplamiento de clases debido a canales de comunicación. Con estas pruebas se demuestra que con el método de refactorización de reducción de acoplamiento, basado en el patrón de diseño 'Mediator' es posible disminuir el número de dependencias debido a llamadas a métodos.

Adicionalmente a los trabajos desarrollados en el laboratorio de ingeniería de software del CENIDET enmarcados en esta sección de Antecedentes, a continuación se presentan trabajos que con el enfoque diferente tratan el problema planteado contra los que compite la investigación de esta tesis.

1.3.2. Trabajos relacionados

Restructuring Conditional Code Structures Using Object Oriented Design Patterns, [Santaolaya S, 2003].

El objetivo de la investigación [Santaolaya S, 2003] es Transformar fragmentos de código en lenguaje C, hacia código en lenguaje C++, basado en los patrones de diseño State y Strategy.

Se realizó la herramienta SR2, que es un software basado en el proceso de reingenería en el cual se introduce código escrito en lenguaje C y genera como salida un framework orientado a objetos en C++. SR2 inicia un proceso de análisis de código fuente apoyado por un analizador léxico y sintáctico El analizador contiene la máquina para el reconocimiento de estructuras condicionales en un programa. El SR2 contiene un conjunto de acciones semánticas que son incluidas en la especificación de la gramática del lenguaje C, escrita en el lenguaje ANTLR. Estas acciones son ejecutadas cuando las estructuras buscadas son encontradas (la presencia de una relación, datos relacionados, funciones, estructuras de datos, etc.) y estos son necesarios para llenar la información de las tablas que pueden servir como base para el proceso de reestructuración.

Identification of refactoring opportunities introducing polymorphism, [Tsantalis, 2009].

El objetivo del trabajo en [Tsantalis, 2009] es Identificar y eliminar problemas correspondientes a estatutos condicionales en proyectos Java. Los autores desarollaron una técnica que extrae sugerencias para aplicar refactorización introduciendo polimorfismo como solución a los estatutos condicionales. La técnica propuesta tiene un enfoque interactivo, en el sentido de que el diseñador del programa es quien acepta o no la sugerencia de refactorización. Hacen uso de la asociación dinámica de tipos, aportada por la propiedad del polimorfismo y los patrones State/Strategy para sugerir oportunidades de refactorización, logrando con esto módulos reusables.

A Two-Step Technique For Extract Class Refactoring, [Bavota, 2010].

El objetivo de la investigación presentada en [Bavota, 2010] es extraer clases con métodos no relacionados, que tienen baja cohesión, que necesitan ser refactorizadas mediante la distribución de algunas de sus responsabilidades a nuevas clases, lo que reduce su complejidad y mejora su cohesión.

El enfoque propone una nueva heurística, que utiliza dos etapas para obtener clases con varias responsabilidades, que serán candidatas para aplicar refactorización y lograr clases con alta cohesión. La función de la heurística es identificar subconjuntos de datos y subconjuntos de métodos que tienen responsabilidades similares y que tienen alta cohesión. Tras la heurística propuesta se proponen dos etapas para extraer clases que podrán ser refactorizadas:

Construcción de matriz método por método: Dada una clase que va ser refactorizada se propone calcular una medida de cohesión (referencias de atributos, llamadas a métodos y contenido semántico) entre todos los posibles pares de métodos en la clase.

Método de extracción de secuencia de métodos (cadenas). En la matriz método por método se identifican las relaciones más importantes entre métodos, para esto se filtra la matriz método por método considerando un umbral de cohesión, en donde todas aquellas menor a este umbral serán igual a 0. La cohesión entre cadenas C_i y C_j se calcula por el promedio de cohesión entre todos los posibles pares de métodos de C_i y C_j.

Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm, [Alkhalid, 2010].

El objetivo del trabajo de [Alkhalid, 2010] es proporcionar soporte automatizado para identificar clases mal estructuradas o de poca cohesión y presentar sugerencias de refactorización.

El enfoque para la refactorización de software a nivel método/función utilizando el algoritmo de agrupamiento (clustering). Se crea un nuevo algoritmo: Adaptive K-Nearest Neighbor (A-KNN) que realiza agrupamiento con diferentes pesos de los atributos.

El enfoque propuesto consta de tres fases:

Fase 1: Fase de preprocesamiento, el código es ordenado a un estilo estándar; esto facilita el trabajo del analizador.

Fase 2: El análisis, en esta fase se genera la matriz entidad-atributo. En esta matriz las filas representan líneas de código, las columnas representan los atributos, que son en este caso variables de las líneas de código, variables que pueden ser variables de datos o de control. Entendiéndose por variable de dato la que es usada directamente en una declaración y las variables de control nos muestran la dependencia de relación entre entidades.

Fase 3: Clusters, se utiliza la matriz entidad-atributo para calcular los coeficientes de semejanza entre las líneas de código, después se usa el algoritmo aglomerativo jerárquico (A-KNN) para producir el árbol de agrupamiento. Este árbol de agrupamiento ayuda al diseñador de software para tomar la decisión en base a su experiencia y conocimientos de como será aplicada la refactorización.

The Impact of Service Cohesion on the Analyzability of Service-Oriented Software, [Perepletchikov, 2010].

El obejtivo de [Perepletchikov, 2010] es mejorar la capacidad de mantenimiento a nivel de sistema en la etapa de diseño.

En el trabajo de [Perepletchikov, 2010] se presenta un conjunto de métricas basadas en el concepto de cohesión de software en diseños orientados al servicio (SO), validadas teóricamente y evaluadas empíricamente para predecir la capacidad de mantenimiento de servicios orientados al diseño basado en productos de software.

Proponen cinco métricas, la cuales son:

Service Interface Data Cohesion (SIDC): Esta métrica mide el grado de correspondencia entre los tipos de parámetros y tipo de retorno a través de cada uno de los métodos de una clase orientada a objetos.

Service Interface Usage Cohesion (SIUC): Esta métrica mide la totalidad de operaciones de servicio que son invocados por todos los clientes. Los clientes son generalmente otros servicios en el sistema, pero en teoría, cualquier pieza de software ejecutable puede ser considerado como un cliente.

Service Interface Implementation Cohesion (SIIC): Cubre todas las características de implementación de las operaciones de servicio. Un servicio se considera con implementación cohesiva cuando la totalidad de sus operaciones de servicio se implementan por los mismos elementos de implementación.

Service Interface Sequential Cohesion (SISC): Un servicio se considera que es secuencialmente cohesivo cuando la totalidad de sus operaciones de servicio tienen dependencias secuenciales, donde una postcondición/salida de una determinada operación satisface una precondición/entrada de la siguiente operación.

Total Interface Cohesion of a Service (TICS): Esta métrica cubre todos los aspectos posibles de cohesión de interfaz de servicio capturado por las métricas previamente definidas, cuantificando de esta manera el total de cohesión (general) de un servicio

Assessing Package Organization in Legacy Large Object-Oriented Software, [Abdeen H, 2011].

El objetivo en [Abdeen H, 2011] es evaluar los aspectos de modularidad en sistemas grandes de software legado orientados a objetos.

Se realizaron 3 nuevas métricas que evalúan algunos principios de modularidad (1.- Ocultamiento de datos y encapsulamiento. 2.- mutabilidad, mantenibilidad y reusabilidad. 3.- Comunalidad (estado de características o atributos compartidos) por meta vs similitud por proposito) para paquetes en sistemas grandes de software legado orientados a objetos, donde las APIs no esten predefinidas. Dos métricas se ocupan del acoplamiento de paquetes y las otras dos con la cohesión del paquete, todas toman un valor entre 0 y 1, donde 1 es el valor óptimo y 0 el peor caso.

Se definen tres principios de modularidad:

Principio 1 (Ocultamiento de Información y Encapsulamiento): La comunicación entre paquetes debe de ser tan poco como sea posible.

Principio 2 (Mutabilidad, Mantenibilidad y Reusabilidad): La conectividad entre paquetes debe ser tan poco como sea posible.

Principio 3 (Comunalidad por meta vs similitud por propósito): Un paquete debe proporcionar sólo un servicio al resto del software, si no es así, el objetivo de cada interfaz de paquete debería ser lo más consistente posible.

JDeodorant: Identification and Application of Extract Class Refactorings, [Fokaefs, 2011].

El objetivo del trabajo de [Fokaefs, 2011] es reconocer las oportunidades de obtener clases con alta cohesión de clases llamadas “God Classes” y aplicar refactorización para mejorar la comprensibilidad del código. En la investigación se desarrolló una herramienta automática que extrae clases para ser refactorizadas, mediante la aplicación de un algoritmo de agrupamiento jerárquico. Utilizan la distancia Jaccard como la métrica de distancia. La distancia entre los atributos y métodos de una clase es calculada para comparar la similitud

de sus conjuntos de entidades. El conjunto de entidades de una entidad (es decir cada atributo y método) contiene todos los miembros de una clase que utiliza o son utilizados por la entidad que se trate. Dada una clase, el algoritmo comienza calculando los conjuntos de entidades de cada clase y la colocación de cada entidad individual en un grupo aparte. Luego, en cada paso, el algoritmo decide fusionar dos grupos (conjunto de entidades) que están más cerca entre sí de acuerdo con el criterio de un solo vínculo. El proceso se detiene, cuando no hay más grupos que puedan fusionarse ya que son todos más distantes que el umbral de distancia.

Evolution of Legacy System Comprehensibility through Automated Refactoring, [Griffith, 2011].

El objetivo del trabajo de [Griffith, 2011] es reducir siginificativamente el tiempo dedicado a un proyecto de software en la parte de mantenimiento durante su ciclo de vida. Se desarrolló un sistema automatizado que utliza algoritmos evolutivos para manipular refactorizaciones correctamente sin necesidad de una comprensión subyacente del software.

Dividen el sistema en tres secciones importantes: procesamiento de entrada, refactorización del subsistema, salida del subsistema.

Procesamiento de entrada: Usan JavaCC para generar un árbol de análisis sintáctico y éste es convertido en un diagrama de flujo que representa el código fuente.

Refactorización del subsistema: En el proceso de refactorizar, se hace uso de un algoritmo genético que primero selecciona un individuo aleatoriamente (que representa un miembro de la población en la iteración actual de la búsqueda) después es pasado al subsistema de medición de métricas en donde se calculan las longitudes de las cadenas (individuos que pueden ser refactorizados) y por último se aplica un conjunto de nociones cualitativas que ayudan a indicar cuándo es necesaria una refactorización o cuando dejar de aplicar refactorización.

Salida del subsistema: Una vez que el subsistema de refactorización ha completado sus tareas, los resultados son pasados al componente SystemController que invoca el componente OutputDirector para generar la salida (diagrama de clases UML).

Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics, [Safwat M, 2012].

En el trabajo de [Safwat M, 2012] se porpociona un nuevo criterio de evaluación para medir la calidad de un diseño de software. Los métodos y atributos heredados son considerados para la evaluación, proporcionando una guía para la elección de clases nominadas para la refactorización.

Los autores realizaron una herramienta llamada *Cohesion Measure Tool* (CMT) que examina la calidad de un diseño de software orientado a objetos. Permitiendo al diseñador personalizar y configurar las opciones para cada métrica de cohesión. Estas son las opciones: se puede elegir entre incluir o excluir el acceso de métodos y constructores para el análisis. También entre incluir o excluir la herencia de los atributos y métodos. Utilizan métricas que miden la cohesión para identificar las clases que serán sometidas a la refactorización. Las pruebas experimentales realizadas a más de 35 clases de de 16 proyectos demuestran que el nuevo criterio de evaluación identifica clases que deben ser refactorizadas especialmente cuando se usan métricas como *Loose Cohesion* (LCC y LCC_1)

In medio stat virtus: Extract Class Refactoring Through Nash Equilibria, [Bavota, 2014].

El objetivo del trabajo [Bavota, 2014] es encontrar la particion de métodos de clases grande y complejas que denominan: “clases Blob”, para maximizar la cohesión y limitar al mismo tiempo el aumento de acoplamiento.

Se desarrolló un enfoque basado en la teoría de juegos para identificar soluciones de refactorización (Extract Class Refactoring (ECR)) que proporcionan una solución de compromiso entre el incremento deseado de la cohesión y el incremento no deseado del acoplamiento.

Algoritmo Extract Class Refactoring (ECR):

Cada jugador (clase que se extrae) inicialmente posee un método principal L (método semilla). Los n jugadores restantes pueden contener L – n métodos de la clase a ser refactorizada. La construcción de las clases es incremental. En cada iteración cada jugador puede tomar al menos uno de los métodos no asignados de la clase original (dos jugadores no pueden tener el mismo método). Si un jugador no toma algún método, se dice que este toma un movimiento nulo . Ejemplo Consideremos el caso de dos jugadores P1 y P2, uno de los siguientes pares de movimientos está permitido durante una iteración;

P1 toma m_i y m_j deja a P2.

P1 toma m_i mientras P2 juega el movimiento nulo.

P1 juega el movimiento nulo, mientras P2 toma m_j.

La combinación de movimientos a realizar para los n jugadores durante una iteración del algoritmo es elegida para encontrar el punto de equilibrio Nash en la matriz de pagos. El algoritmo termina cuando cada método de la clase original es asignado a uno de los n jugadores.

Tabla 1.1 Resumen de trabajos relacionados

ARTÍCULO	***OBJETIVO***	***MEDIO***	***HERRAMIENTA O ENFOQUE***	***MÉTRICAS USADAS****	***ENTRADA***	***SALIDA***
[Santaolaya S, **2003**] Restructuring Conditional Code Structures Using Object Oriented Design Patterns.	Obtener marcos orientados a objetos en lenguaje C++.	Refactorizando mediante el uso de patrones de diseño *State/Strategy*	SR2	LOC y CC para revisar resultados	Código en lenguaje C.	Código en lenguaje C++.
[Tsantalis, **2009**] Identification of refactoring opportunities introducing polymorphism.	Eliminar estatutos condicionales.	Refactorizando mediante el uso de *Polimorfismo* y patrones de diseño *State/Strategy*	Una técnica para extraer sugerencias de refactorización.	LCC	Código en lenguaje Java	Código en lenguaje Java
[Bavota, **2010**] A Two-Step Technique For Extract Class Refactoring.	Lograr nuevas clases menos complejas y con mayor cohesión.	Refactorizando haciendo uso de la nueva heurística desarrollada.	Nueva heurística llamada Responsibility-based	ClassCoh	Código en lenguaje Java	Código en lenguaje Java
[Alkhalid, **2010**] Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm.	Identificar funciones mal estructuradas o de poca cohesión y presentar sugerencias de refactorización.	Refactorizando utilizando el algoritmo de agrupamiento	Nuevo algoritmo: Adaptive K-Nearest Neighbor (A-KNN)	LCOM	Código en lenguaje Java	Código en lenguaje Java
[Fokaefs, **2011**] JDeodorant: Identification and Application of Extract Class Refactorings.	Obtener clases con alta cohesión de clases llamadas "God Classes"	Refactorizando mediante la aplicación de un algoritmo de agrupamiento jerárquico	JDeodorant	MPC, CBO y LCOM	Código en lenguaje Java	Código en lenguaje Java
[Griffith, **2011**] Evolution of Legacy System Comprehensibility through Automated Refactoring.	Reducir siginificativamente el tiempo dedicado a un proyecto en la parte de mantenimiento.	utliza algoritmos evolutivos para manipular refactorizaciones correctamente	Un sistema automatizado (No mencionan el nombre).	DIT, LCOM, NOC, WMC, CBO, RFC.	Código en lenguaje Java	Código en lenguaje Java

[Safwat M, **2012**] Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics.	Medir la calidad (Cohesión) de un diseño de software	Configurar las opciones para cada métrica de cohesión	Cohesion Measure Tool (CMT)	LCCI, LCCD, LCC, TCC, CC**, LCOM3	Código en lenguaje Java	Código en lenguaje Java
[Bavota, **2014**] In medio stat virtus: Extract Class Refactoring Through Nash Equilibria.	Encontrar la partición de métodos de una clase Blob para maximizar la cohesión, limitando al mismo tiempo el aumento de acoplamiento	Refactorizando aplicando un nuevo algoritmo.	Un algoritmo basado en la teoría de juegos	LCOM2, CBO Y MPC.	Código en lenguaje Java	Código en lenguaje Java
En este trabajo de investigación	Obtener módulos auto-suficientes.	Agregando al SR2-Refactoring un proceso que logre el equilibrio entre los valores de cohesión y coherencia.	Un método de refactorización posiblemente una nueva métrica de cohesión basada en la interacción de métodos, y una herramienta que implementa las métricas y el método de refactorización.	Valores de cohesión y coherencia.	Código en lenguaje C++	Código en lenguaje C++

* LOC = Lines Of Code, CC = Cyclomatic Complexity, LCC = Lose Class Cohesion, LCOM Lack of Cohesion Of Methods, MPC = Message Passing Coupling, CBO = Coupling Between Objects, DIT = Depth Of The Inheritance Tree, NOC = Number Of Children, WMC = Weighted Methods Per Class, RFC = Response For a Class, LCCD = Lack Of Cohesion in the Class-Direct, LCCI = Lack Of Cohesion in the Class-Indirect, TCC = Tight Class Cohesion, **CC = Class Cohesion.

1.4. Organización de esta tesis

La estructura restante de este documento se compone de cuatro capítulos más, los cuales se describen a continuación.

Capítulo 2

En este capítulo se describen las métricas de cohesión, coherencia y métodos de refactorización que fueron desarrollados en esta investigación, para dar solución al problema presentado en (§1.1).

Capítulo 3.

En este capítulo se presentan los diagramas y la implementación de la herramienta desarrollada en esta tesis y todo el trabajo realizado para lograr el equilibrio entre los factores de cohesión y coherencia.

Capítulo 4.

En este capítulo se describen las pruebas que se realizaron a los métodos de refactorización utilizando distintos casos de prueba y los resultados a partir de ellas.

Capítulo 5.

En este capítulo se encuentran las conclusiones y los trabajos futuros a los que se llegaron después de haber realizado este trabajo.

DESCRIPCIÓN DEL MÉTODO DE SOLUCIÓN

Capítulo 2

En este capítulo se describe el método de solución que fue implementado en la herramienta desarrollada en esta tesis. Se presentan las métricas y el proceso de refactorización que fue desarrollado para lograr el equilibrio entre los factores de cohesión y coherencia.

2.1. Coherencia.

La coherencia se define como el grado de relación funcional para completar una responsabilidad en una unidad de programa. La coherencia se basa en el principio de una única responsabilidad *Single Responsibility Principle (SRP*) el cual: "indica que una clase o módulo debe tener uno y sólo un motivo para cambiar" [Martin, 2002].

En términos generales una clase está compuesta por elementos que pueden ser métodos y atributos. En el contexto de la coherencia, una responsabilidad se refiere a una secuencia interactiva de métodos implicados para cumplir o alcanzar una meta o un objetivo de valor para un usuario.

2.1.1. Métrica para medir coherencia (Cr).

En esta investigación se diseñó una métrica para medir la coherencia a nivel de clase de objetos denominada Cr (**C**ohe**r**encia). Cr revisa la secuencia interactiva de métodos que participan para resolver una única responsabilidad, es decir, busca la llamada de un método a otro método y de ahí sigue la secuencia hasta que el último método no llame a algún método subsiguiente, lo cual indica la terminación de la secuencia interactiva de métodos.

La métrica Cr necesita saber el **t**otal de **m**étodos (*tm*) en la clase y la cantidad de métodos de clase que participan en cada **s**ecuencia **i**nteractiva de métodos (*si*) para resolver una única responsabilidad. Se divide (*si*) entre (*tm)* para calcular el grado de coherencia de la clase. Los valores van de 0 a 1, donde 0 significa incoherencia total y 1 significa alta coherencia. La fórmula de Cr es la siguiente:

$$Cr = \frac{si}{tm}$$

Donde: *si* = secuencia interactiva de métodos.
tm = total de métodos de la clase.

2.1.2. Ejemplo aplicando métrica de coherencia Cr.

En la Figura 2.1 tenemos un ejemplo de una clase *A* que contiene cuatro métodos en total. Se observa una secuencia interactiva de métodos que inicia con el método *m1* invocando al método *m4* y éste a su vez invocando al método *m3,* siendo *m3* el final de la secuencia interactiva de métodos.

Aplicando la métrica Cr, tenemos que el total de métodos de la clase *A* es de *tm* = 4 y la secuencia interactiva de métodos *si* = *3,* por lo tanto, *Cr* = 3/4 = 0.75.

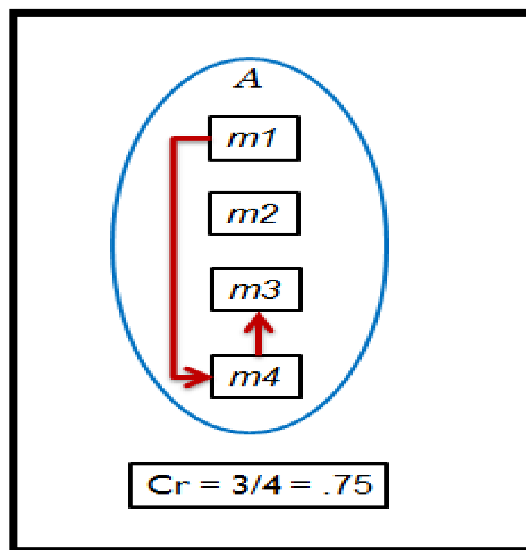


Figura 2.1 Ejemplo para aplicar métrica Cr

2.2. Cohesión

La cohesión se define como el grado de relación entre los miembros de una clase [Chidamber, 1992]. La cohesión es el concepto donde se ve cómo los métodos de una clase están estrechamente relacionados entre sí. Si el módulo es altamente cohesivo la complejidad de las clases se reduce, el módulo es reusable y fácil de mantener [Yadav, 2014].

2.2.1. Métrica para medir cohesión LCOM4

Para seleccionar la métrica de cohesión se realizó un análisis de 12 métricas que miden cohesión (§Anexo A). Como resultado del análisis de las métricas y basándose en el concepto de cohesión se determinó que la métrica apropiada para este trabajo de investigación es *Lack of Cohesion of Methods (*LCOM4*).*

La métrica LCOM4 considera un grafo no dirigido *G*, donde los vértices son los métodos de una clase y existe una arista entre dos vértices si los métodos *Mi* y *Mj* correspondientes comparten al menos un atributo o si *Mi* invoca a *Mj* o viceversa. El número de conjuntos que se formen en el grafo indica el valor de cohesión en la clase.

Si LCOM4 = 1 indica una clase con alta cohesión.

Si LCOM4 >= *2* indica un problema (baja cohesión), la clase debe dividirse en otras clases más pequeñas.

Si LCOM4 = 0 no hay métodos en esa clase, también son consideradas clases mal diseñadas.

2.2.2. Ejemplo aplicando métrica cohesión LCOM4.

En la Figura 2.2 se observa que la clase *classA* tiene tres métodos y dos atributos, en este caso el método *m1* tiene relación con el método *m2* porque comparten el atributo *a1* y el método *m2* se encuentra relacionado con el método *m3* porque existe una invocación entre ellos.

Aplicando la métrica LCOM4 se dibujan los tres métodos *m1, m2* y *m3* que son los vértices del grafo *G* y se traza una línea entre *m1* y *m2* porque comparten un atributo y la línea entre el vértice *m2* y *m3* se dibuja por la invocación que existe de *m2* a *m3.* En este grafo se forma solo un conjunto de métodos por eso el valor de LCOM4 es 1, lo que quiere decir que la clase tiene alta cohesión porque todos los métodos están relacionados.

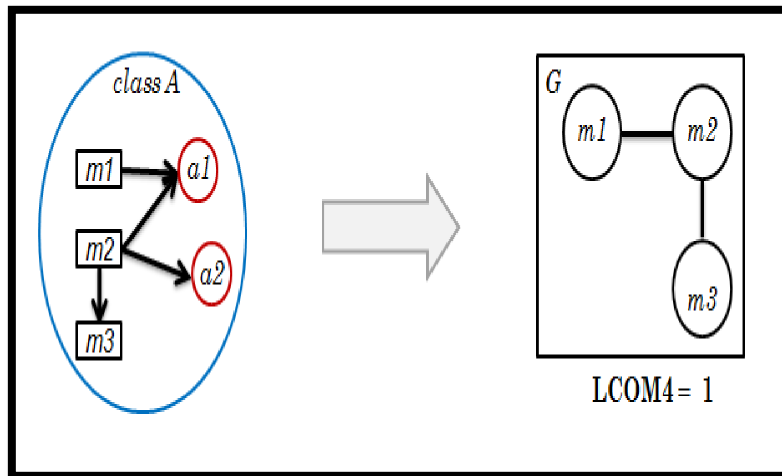


Figura 2.2 Ejemplo para aplicar métrica LCOM4

En la Figura 2.3 tenemos la clase *classB* que contiene cuatro métodos y dos atributos. Se puede ver que los métodos *m3* y *m4* están relacionados porque ambos utilizan el atributo *a2*, los métodos restantes no comparten algún atributo y no se invocan entre ellos.

Aplicando la métrica LCOM4 se dibujan cuatro vértices correspondientes a los métodos de la clase y sólo se dibuja una línea entre el método *m3* y el método *m4,* en este grafo se forman tres conjuntos de métodos, conjunto uno: *m3* y *m4,* conjunto dos: *m1* y conjunto tres: *m2,* por lo tanto, LCOM4 = 3, lo que significa que esta clase debe ser dividida.

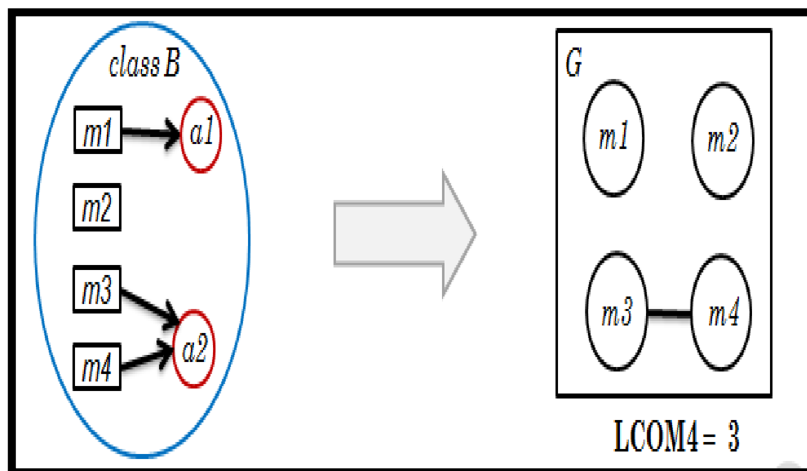


Figura 2.3 Ejemplo 2 para aplicar LCOM4

2.2.3. Métrica para medir cohesión LCOM4 –A

Esta métrica forma parte del resultado del análisis de métricas de cohesión. En el análisis se encontró que la mayoría de las métricas no miden la cohesión de clases que tienen un único método, es decir, al encontrar una clase que únicamente declara un método en automático es considerado como una clase con alta cohesión. Basados en la definición de cohesión no se puede decir que una clase con un único método se considere con alta cohesión sin tomar en cuenta los posibles atributos de la clase (Anexo A).

LCOM4 – A (*Lack of Cohesion of Methods - Attribute*) es una métrica que está basada en LCOM4 y se propone para medir las clases que contienen un único método. LCOM4-A representa los atributos como vértices del grafo y se trazan las aristas cuando se están utilizando los atributos en el método, de este modo obtenemos el grafo que nos indica el grado de cohesión para las clases que tienen un único método.

La definición para la métrica **LCOM4 – A** es la siguiente: se considera un grafo G donde los atributos de la clase a_i y a_j son considerados como los vértices del grafo G y se trazan las aristas entre dos vértices si atributos son utilizados en el método de la clase. En otras palabras se puede decir que **LCOM4 – A** mide el grado de relación entre atributos y un método de la clase.

Si **LCOM4 – A** = 1 indica una clase con alta cohesión.

Si **LCOM4 – A** >= 2 indica un problema, la clase tiene variables que posiblemente puedan ser eliminadas de la clase.

Si **LCOM4 – A** = 0 indica una clase con alta cohesión, debido a que no existen variables a nivel clase.

2.2.4. Ejemplo aplicando métrica LCOM4-A

En la Figura 2.4 se puede ver la clase *classA* que contiene solo un método *m1* declarado y dos atributos *a1* y *a2.* Dibujamos en el grafo *G* los dos atributos, en este caso *a1* es utilizado en el método *m1 pero a2* no se usa, por lo tanto, no se dibuja una arista entre los dos atributos y se forman dos conjuntos de atributos (conjunto 1: *a1* y conjunto 2: *a2*), por este motivo el resultado de LCOM4-A = 2, que significa baja cohesión.

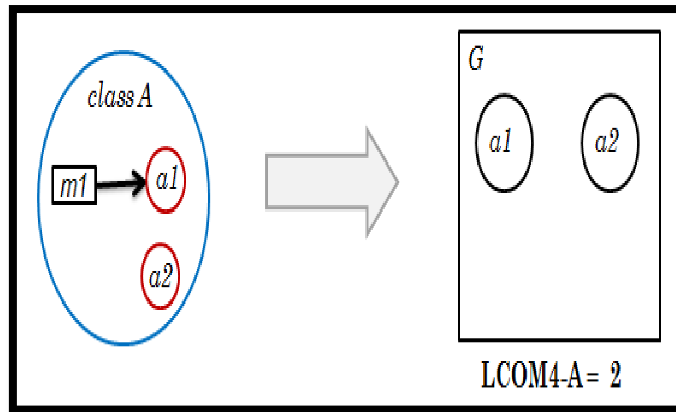


Figura 2.4 Ejemplo para aplicar LCOM4-A

En la Figura 2.5 se presenta una clase *classA* únicamente con un método declarado *m1* y dos atributos *a1* y *a2*. Aplicando LCOM4-A los dos vértices son los atributos *a1* y *a2* y se dibuja una arista entre ellos porque ambos atributos son utilizados en el método *m1,* en este caso sólo se forma un conjunto (conjunto 1: *a1* y *a2*) obteniendo como resultado LCOM4-A = 1, significa que la clase tiene alta cohesión.

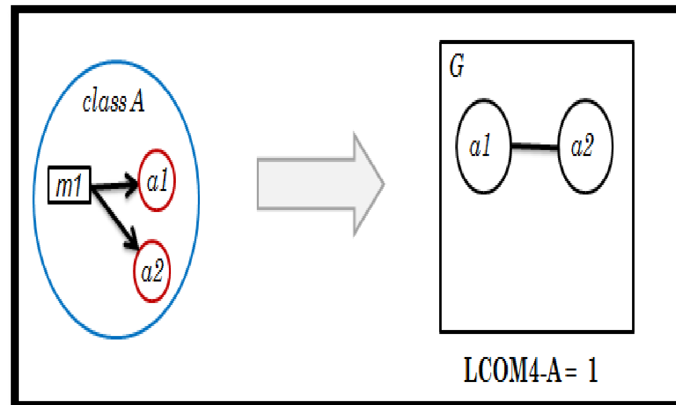


Figura 2.5 Ejemplo 2 para aplicar LCOM4-A

2.3. Proceso del método de refactorización.

Para cumplir con el objetivo (§1.2.1) se desarrolló una herramienta que utiliza un proceso de refactorización denominado: "*proceso de refactorización equilibrio*", el cual fue desarrollado en esta tesis. El *proceso de refactorización equilibrio* está conformado por tres métodos de refactorización, los cuales son: **m**étodo de **r**efactorización para lograr **a**lta **c**ohe**s**ión (**MRACS**), **m**étodo de **r**efactorización para lograr **a**lta **c**ohe**r**encia (**MRACR**) y el **m**étodo de **r**efactorización de **o**rdenamiento de **cl**ases (**MROCL**).

En este trabajo de tesis fueron desarrollados los tres métodos de refactorización: MRACS, MRACR y MROCL. A partir de este punto cuando nos refiramos al proceso de refactorización estamos hablando del *proceso de refactorización equilibrio* y cuando se menciona al método de refactorización estamos hablando de uno de los métodos (MRACS, MRACR o MROCL) desarrollados en esta tesis. A continuación se describen los tres métodos de refactorización desarrollados entes trabajo de tesis.

2.3.1. Método de refactorización para lograr alta cohesión (MRACS).

MRACS logra clases con alta cohesión, es decir, crea clases con todos sus elementos relacionados. Este método consiste de ocho pasos ver Figura 2.6. Para explicar el método MRACS se hará uso del ejemplo de un programa orientado a objetos en C++ que se encuentra en la Figura 2.7, el cual consiste de un archivo con el método *main()* y una clase con el nombre *Temp*. La clase contiene dos atributos y dos funciones. En el método *main()* se crean dos objetos de clase (*Temp*) para llamar a sus respectivos métodos.

En el código de la Figura 2.7 se puede observar que el atributo *data1* es utilizado en el método *int_data(int d)*, también vemos que el atributo *data2* es utilizado en el método *float_data()*. Por lo tanto, no existe relación entre los métodos por medio de algún atributo de la clase y tampoco tienen una invocación entre los métodos. Aplicando la métrica LCOM4 (§2.2.1) el resultado es de dos, lo que significa que la clase tiene baja cohesión y debe ser dividida.

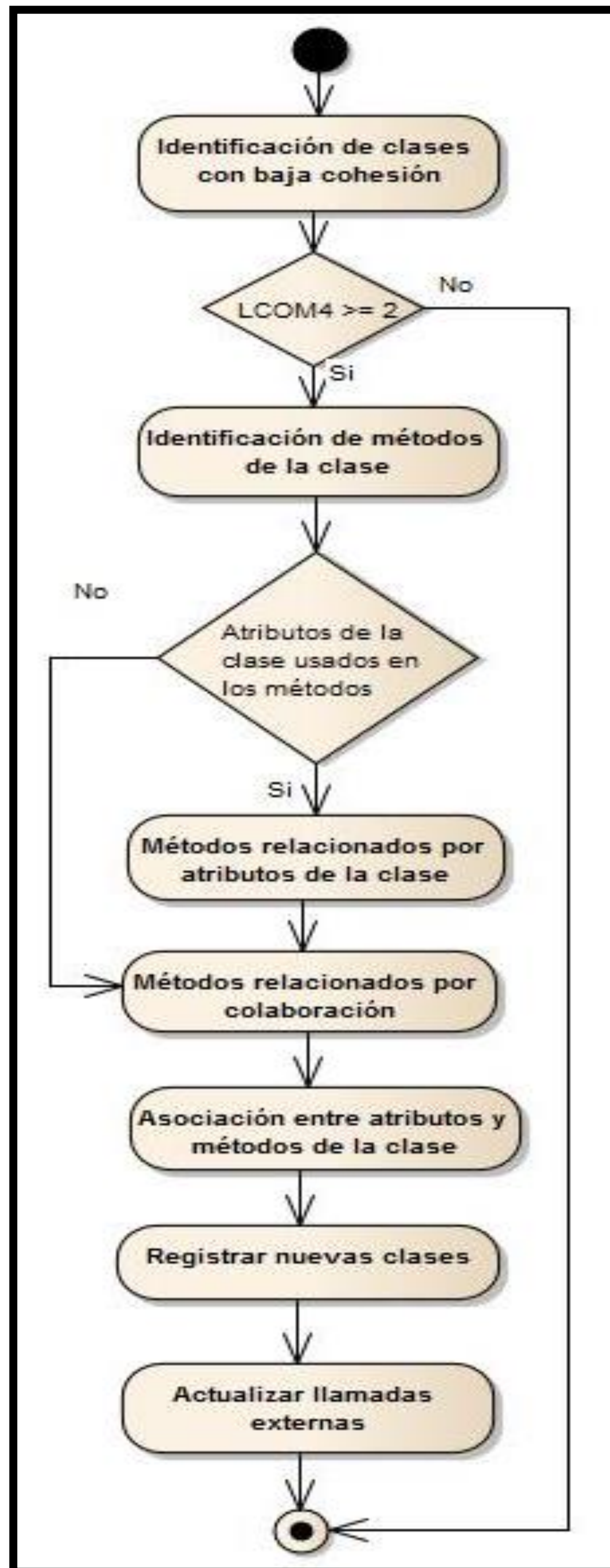


Figura 2.6 Diagrama de actividades del MRACS

```
4   #include <iostream>
5   class Temp
6   {
7       private:
8           int data1;
9           float data2;
10      public:
11          void int_data(int d)
12          {
13              data1=d;
14              std::cout<<"Number: "<<data1;
15          }
16          float float_data()
17          {
18              std::cout<<"\nEnter data: ";
19              std::cin>>data2;
20              return data2;
21          }
22  };
23  int main()
24  {
25          Temp *obj1 = new Temp();
26          Temp *obj2 = new Temp();
27          obj1->int_data(12);
28          std::cout<<"You entered "<<obj2->float_data();
29          return 0;
30  }
```

Figura 2.7. Ejemplo de programa en C++

2.3.1.1. Paso 1: identificación de clases con baja cohesión.

El primer paso del método de refactorización MRACS es encontrar las clases que tienen baja cohesión como es el caso del código de la Figura 2.7. En este momento inicia el método de refactorización MRACS.

2.3.1.2. Paso 2: identificación de métodos de la clase.

El segundo paso es identificar los métodos que tiene la clase, en el ejemplo de la Figura 2.7 los dos métodos encontrados serian *int_data(int d) y float_data()*.

2.3.1.3. Paso 3: identificación de atributos de la clase usados en los métodos.

En el tercer paso se encuentran los atributos de la clase que se usan en los métodos, en el método *int_data(int d)* se utiliza el atributo *data1,* en el método *float_data()* se utiliza el atributo *data2.* Cuando no se encuentran atributos de la clase se continúa en el paso 5.

2.3.1.4. Paso 4: métodos relacionados por atributos de la clase.

Se crea una matriz donde las filas son clases y las columnas son métodos. Por cada método de la clase se comparan sus atributos de la clase usados en el método contra los atributos de la clase usados en otro método de la misma clase. Si coinciden con usar el mismo atributo los dos métodos se guardan en la misma fila, de lo contrario, cada método se guarda en una fila diferente. En el caso del código de la Figura 2.7 ninguno de los dos métodos comparte un atributo, por lo tanto, se guardan en diferentes filas como se observa en la Figura 2.8.

int_data	
float_data	

Figura 2.8 Matriz relación de métodos por atributo

2.3.1.5. Paso 5: Métodos relacionados por colaboración.

En este paso se revisan las llamadas internas de los métodos de la clase, es decir, se buscan las llamadas que pueden existir entre métodos que se encuentran dentro de la clase, en este punto la matriz de métodos relacionados por variables es utilizada para que ver que métodos se encuentran separados (diferentes filas) y se busca si existe una invocación entre estos métodos.

Al final de este paso se regresa una matriz donde quedan ordenados los métodos en sus respectivas filas si tienen relación por un atributo o por alguna invocación. En el ejemplo de

la Figura 2.7 los métodos no tienen invocación, por lo tanto, la matriz final es igual al de la Figura 2.8.

2.3.1.6. Paso 6: Asociación entre atributos y métodos de la clase.

En esta parte del método de refactorización ya se conocen como quedan distribuidos los métodos en nuevas clases. Cada fila de la matriz que se logró en el paso anterior representa una clase y las columnas los métodos de cada clase. En este paso se crea una matriz con los atributos de la clase que deben ir en cada fila, de manera que, la fila cero de la matriz de métodos relacionados debe tener las variables que tiene la matriz de atributos de la clase de la fila cero, ver Figura 2.9.

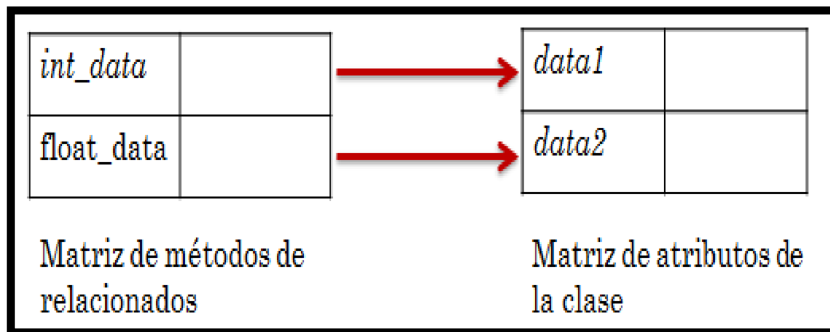


Figura 2.9 Matriz de métodos relacionados y matriz de atributos de la clase

2.3.1.7. Paso 7: Registrar nuevas clases

En este paso el método de refactorización necesita de las dos matrices que se obtienen en los pasos (§2.3.1.5) y (§2.3.1.6) para registrar las nuevas clases. El método inicia creando el nombre que debe llevar la nueva clase, el cual se forma con la palabra *cohesión* más el último *identificador* (el identificador se refiere al número que funciona como llave primaria en la tabla CLASS de la base de datos) de la tabla CLASS, más un *guion bajo* y por último se agrega el *nombre de la clase original* (por clase original nos referimos a la clase que tiene baja cohesión y que se encuentra en el proceso de refactorización). En este caso estamos creando el archivo con extensión *.h*, el nombre del archivo se forma con el nombre de la nueva clase más la extensión *.h*.

Posteriormente se re-ubican los métodos y atributos de la clase con el identificador de la nueva clase de manera correspondiente. Se revisa si la clase original tiene herencia. En

caso de tenerla registramos la herencia en la nueva clase, por último se copian los archivos *include* de la clase original a la nueva clase creada. En la Figura 2.10 se observa el archivo con extensión *.h* de la nueva clase creada.

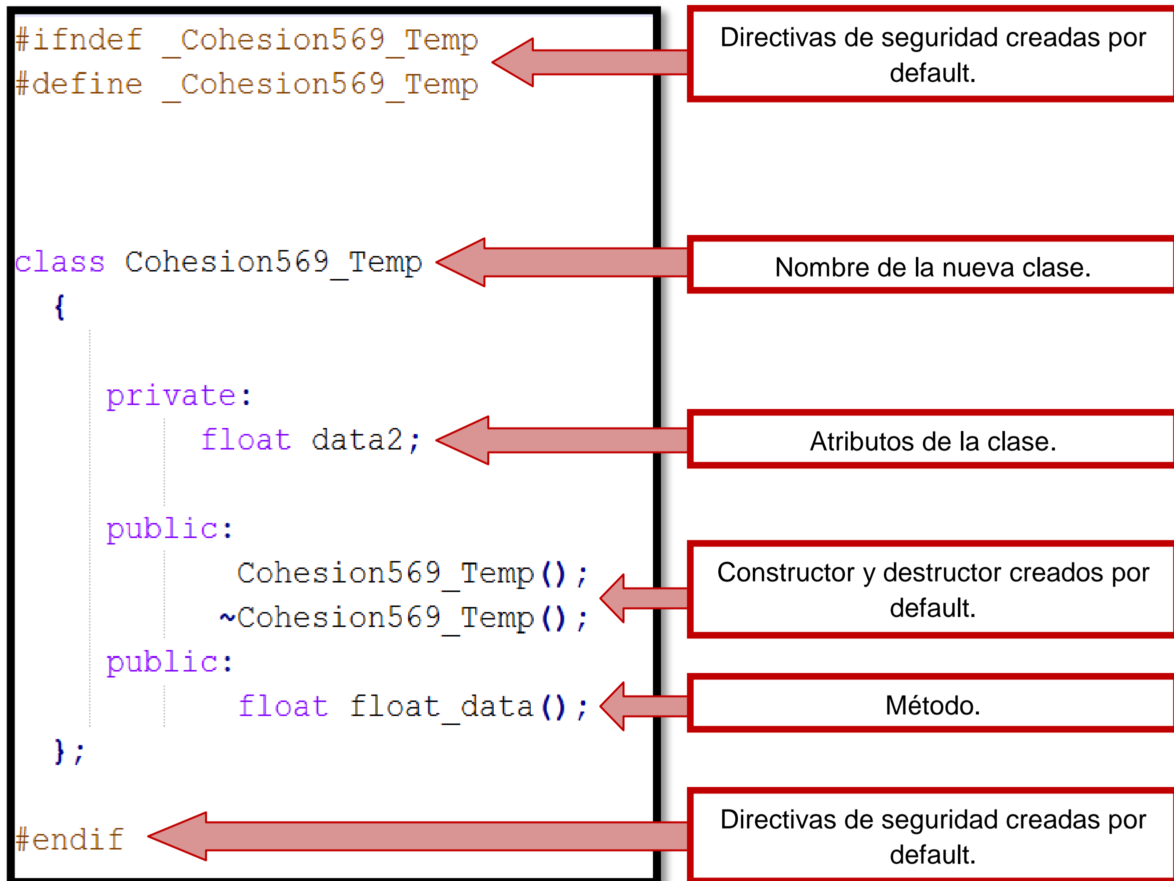


Figura 2.10 Archivo de la nueva clase creada

2.3.1.8. Paso 8: Actualizar llamadas externas.

Se revisan las llamadas externas de los métodos que se movieron a nuevas clases. Si existe alguna llamada externa hacia los métodos que se encuentran en clases nuevas, se crea una variable de instancia de la clase donde se realiza la llamada. Esta variable se declara a nivel clase, su tipo e instancia que se crea son del tipo de la clase nueva en donde se encuentra el método y al nombre de la variable se le antepone el asterisco (*) lo que significa que es un puntero.

Cuando el método de refactorización encuentra que la clase donde se debe declarar la variable es donde se encuentra el método *main()*, en ese momento cambia el nivel de declaración de la variable a nivel método ver Figura 2.11. El nombre de la variable es el mismo que el de la nueva clase, con la diferencia de que se declara en minúsculas en la Figura 2.11 se encuentra un ejemplo del nombre de una nueva variable de instancia.

Como último paso en el proceso del método de refactorización se reconstruyen las llamadas de manera adecuada de los métodos que se movieron a una nueva clase. Se identifica dónde se realiza la llamada y se cambia el atributo utilizado por la nueva variable creada y de ser necesario se cambia el operador punto (.) por el operador flecha (>).También se agrega el archivo include de la nueva clase creada, ésto por la nueva variable que se agrega. En la Figura 2.11 se puede observar los nuevos elementos que fueron agregados al momento de crear el archivo para el método *main()*.

```
#include <iostream>
#include "Temp.h"
#include "Cohesion569_Temp.h"


int main()
{
    Cohesion569_Temp *cohesion569_temp = new Cohesion569_Temp();
     Temp *obj1 = new Temp();

     obj1->int_data(12);
         std::cout<<"You entered "<<cohesion569_temp->float_data();
     return 0;
}
```

Archivos incluidos

Nueva variable declarada de tipo puntero

Modificación del objeto con el que se realiza la llamada

Figura 2.11 Ejemplo de archivo creado con la función main()

2.3.2. Método de refactorización para lograr alta coherencia (MRACR)

El MRACR tiene el objetivo de lograr clases con alta coherencia, es decir, clases que contengan una única secuencia interactiva de métodos, sin embargo, no en todos los casos es posible lograr alta coherencia. En la Figura 2.12 se presentan los pasos que sigue el MRACR. Para explicar el método propuesto MRACR se hará uso del ejemplo de un programa orientado a objetos en C++ del cual se presenta el diagrama de secuencia en la Figura 2.13.

El ejemplo de la Figura 2.13 contiene una clase con el nombre *Operaciones*, dentro de la clase *Operaciones* se encuentran cinco métodos implementados: *operacionSimple, operacionCompuesta, operacionDos, operacionTres y operacionUno*. La clase no tiene atributos declarados. Dentro del mismo archivo se encuentra el método *main()*, en el *main()* se crean dos variables de referencia del tipo *Operaciones*: *operación* y *calcular*. Se invoca al método *operacionCompuesta* utilizando la variable de referencia *operación* y el método operacionUno se invoca con la variable de referencia *calcular*.

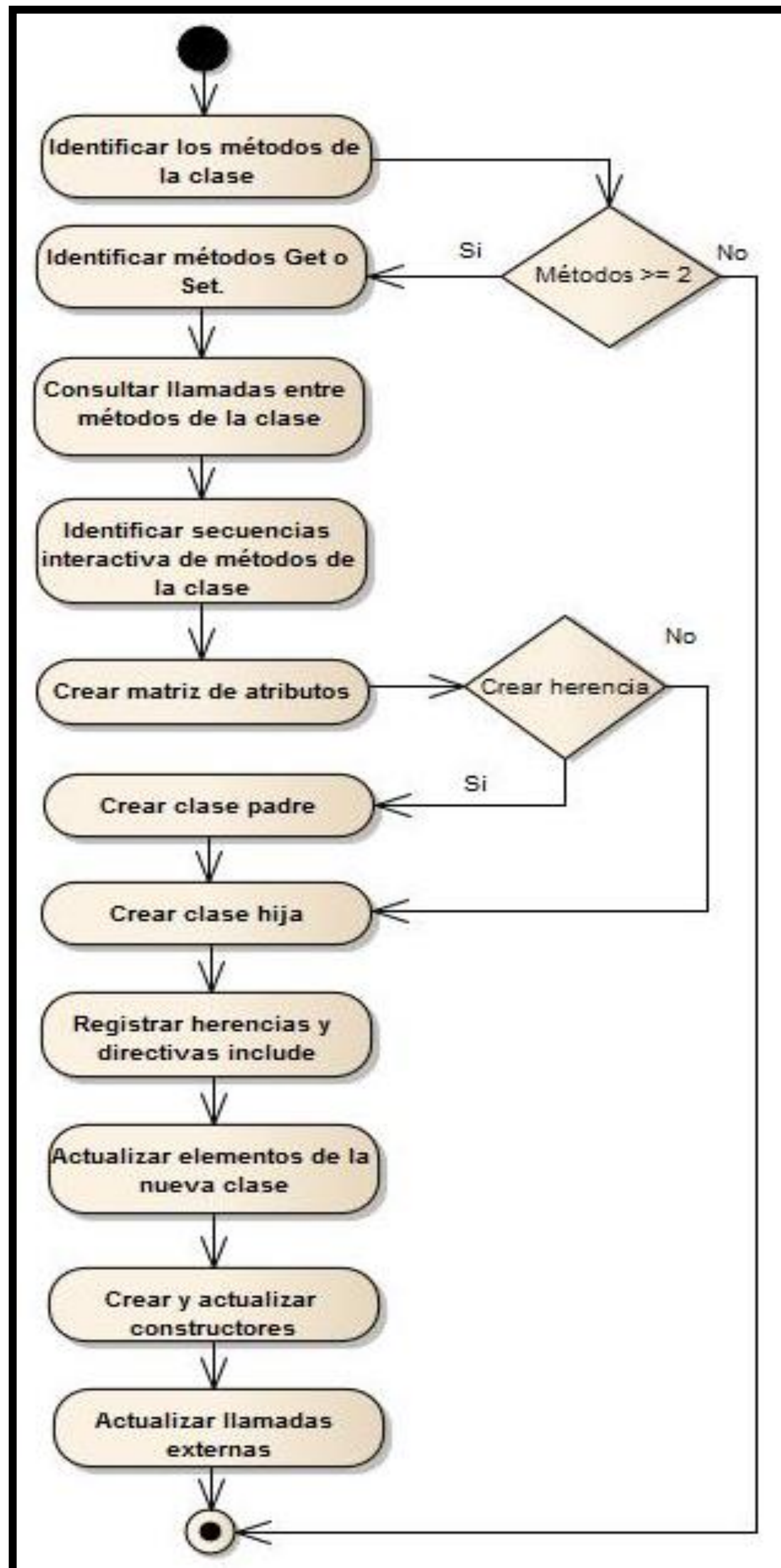


Figura 2.12 Diagrama de actividades del MRACR

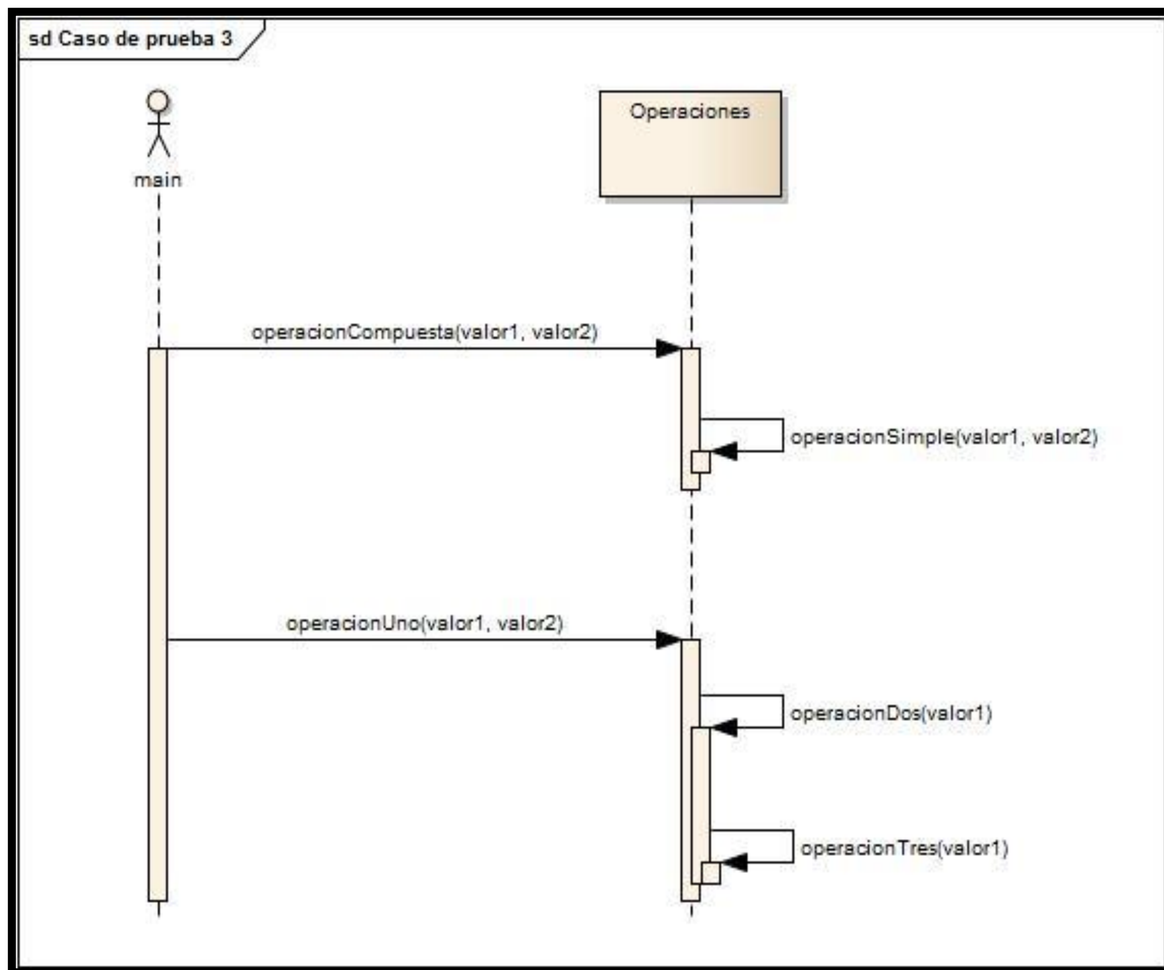


Figura 2.13 Diagrama de secuencia para aplicar MRACR.

2.3.2.1. Paso 1: identificar los métodos de la clase.

MRACR inicia buscando los métodos implementados de la clase que se está refactorizando, en el ejemplo de la Figura 2.13 se encuentran cinco métodos implementados, los cuales son; *operacionSimple, operacionCompuesta, operacionDos, operacionTres y operacionUno.* El método continúa su proceso de refactorización si al menos encuentra dos métodos implementados. De existir solo un método en la clase, se concluye el método de refactorización para esa clase, debido a que no se pueden formar secuencias interactivas con un único método.

2.3.2.2. Paso 2: identificar métodos Get o Set.

Con la información de los métodos de la clase se revisan si cumplen con las características básicas para determinar si es un método Get o Set. Debido a que coherencia busca la llamada de un método a otro y entre los métodos Get o Set no existe una invocación entre ellos, el resultado sería separar los métodos y formar una clase por cada uno de ellos. Esto estaría afectando el patrón Modelo Vista Controlador (MVC) en su versión de 4 capas, donde la cuarta capa se refiere a clases con métodos Get o Set. Por tal motivo, este paso se vuelve fundamental en MRACR.

Las características que debe cumplir un método para ser considerado como Get o Set son: para un método Get;

Debe tener un tipo de retorno (*int, char, string*, etc.).
Su firma debe ser vacía, es decir, no debe tener parámetros.
Su modificador de acceso debe ser *public*.
El nombre del método debe iniciar con Get.
Utiliza una variable a nivel clase.

Para un método Set;

Debe ser de tipo *void.*
Su firma debe ser de uno, es decir, recibir un parámetro.
Su modificador de acceso debe ser *public*.
El nombre del método debe iniciar con Set.
Utiliza una variable a nivel clase.

Los métodos que se identifican como Get o Set se ingresan en una lista, si todos los métodos cumplen con las características de Get o Set el método de refactorización termina. En el caso de la Figura 2.13 ninguno de los métodos cumple con las características mencionadas anteriormente, por lo tanto, el método de refactorización MRACR continua en el paso 3.

2.3.2.3. Paso 3: Consultar llamadas entre métodos de la clase.

Se consultan las llamadas de los métodos que se encuentran dentro de la clase original (clase original: se refiere a la clase que se encuentra en ese momento en el proceso de refactorización). Sólo se consultan las llamadas internas (llamadas internas: se refiere a las llamadas que se realizan entre métodos de la misma clase), de los métodos que no son Get o Set.

Cuando no existen llamadas internas, el método de refactorización crea una matriz insertando cada método de la clase en una fila diferente, de este modo sabemos que cada método debe ir en una nueva clase, continua en el paso 8. En el ejemplo de la Figura 2.13, si existen llamadas internas, por lo tanto, el método de refactorización MARCR continua en el paso 4.

2.3.2.4. Paso 4: identificar secuencias interactivas de métodos de la clase.

En este paso se analizan las llamadas internas de la clase original, en el caso de la Figura 2.13 las llamadas internas de la clase *Operaciones* son: *operacionCompuesta* manada a llamar al método *operacionSimple,* el método *operacionUno* manda a llamar al método *operacionDos* y *operacionTres* es invocado por el método *operacionDos.*

Con la primera llamada interna del método *operacionCompuesta* invocando al método *operacionSimple* inicia la creación de la secuencia interactiva de métodos, después se busca si *operacionSimple* invoca a otro método, en este ejemplo *operacionSimple* no invoca a otro método, por lo tanto, esta secuencia interactiva finaliza sólo con dos métodos (*operacionCompuesta* y *operacionSimple*), se repite este proceso con todas las llamadas internas. En el caso del ejemplo que está analizando la segunda secuencia interactiva de métodos quedaría conformada por *operacionUno, operacionDos* y *operacionTres.*

El resultado final del paso cuatro es crear una matriz, donde se ingresan las secuencias interactivas formadas y la lista de métodos Get o Set si existen en la clase. En la Figura 2.14 se observa la matriz para el caso de ejemplo que está siendo analizando.

operacionCompuesta	operacionSimple	
operacionUno	operacionDos	operacionTres

Figura 2.14 Ejemplo de matriz de secuencia interactiva de métodos

2.3.2.5. Paso 5: crear matriz de atributos.

Cada fila de la matriz que se logró en el paso (§2.3.2.4) representa una clase y en las columnas se encuentran los métodos que debe tener cada clase.

En este paso se crea una matriz con los atributos de la clase que deben ir en cada fila, de manera que la fila cero de la matriz de secuencia interactiva de métodos debe tener los atributos que contiene la matriz de atributos de la clase de la fila cero.

En el caso del ejemplo analizado la clase *Operaciones* no tiene declarados atributos de la clase, por esta razón, no se crea la matriz de atributos de la clase.

2.3.2.6. Paso 6: validar herencia.

En este paso se revisa si es posible crear una herencia para las nuevas clases que se pueden crear. La primera validación es revisar si los métodos de la secuencia interactiva de métodos comparten algún atributo de la clase con los métodos de la clase original.

Si comparten atributos de la clase, se revisa que las llamadas externas de los métodos que se mueven a un nueva clase, se realicen con un objeto diferente, es decir, que se utilicen diferentes instancias para la llamada de cada método (por llamadas externas nos referimos a: llamadas de métodos que se encuentran en otras clases a los métodos que se mueven a una nueva clase).

Cuando no se cumple con las dos validaciones mencionadas anteriormente (compartir atributos de la clase y llamadas de diferentes instancias) significa que no es posible separar la secuencia interactiva de métodos con los métodos de la clase original, por consiguiente, finaliza el método de refactorización.

Existen casos donde no se declaran atributos de la clase y las llamadas externas de los métodos que se mueven a una nueva clase se reconstruyen conforme a las instancias correspondientes. Como es el caso del ejemplo que se está analizando.

En la Figura 2.16 se puede ver el método main(), quien es el encargado de realizar la llamada externa al método que se movió a una nueva clase. Se observa que existen dos variables de referencia *operación* y *calcular*. En el método main() se invoca al método operaciónCompuesta del objeto (variable de referencia) operación. También se invoca al método *operacionUno* del objeto (variable de referencia) calcular.

En el diagrama de clases se representa la invocación que se realiza en el método main() (clase cliente) al método *operacionUno* que se encuentra en la clase *Operaciones,* ver Figura 2.15.

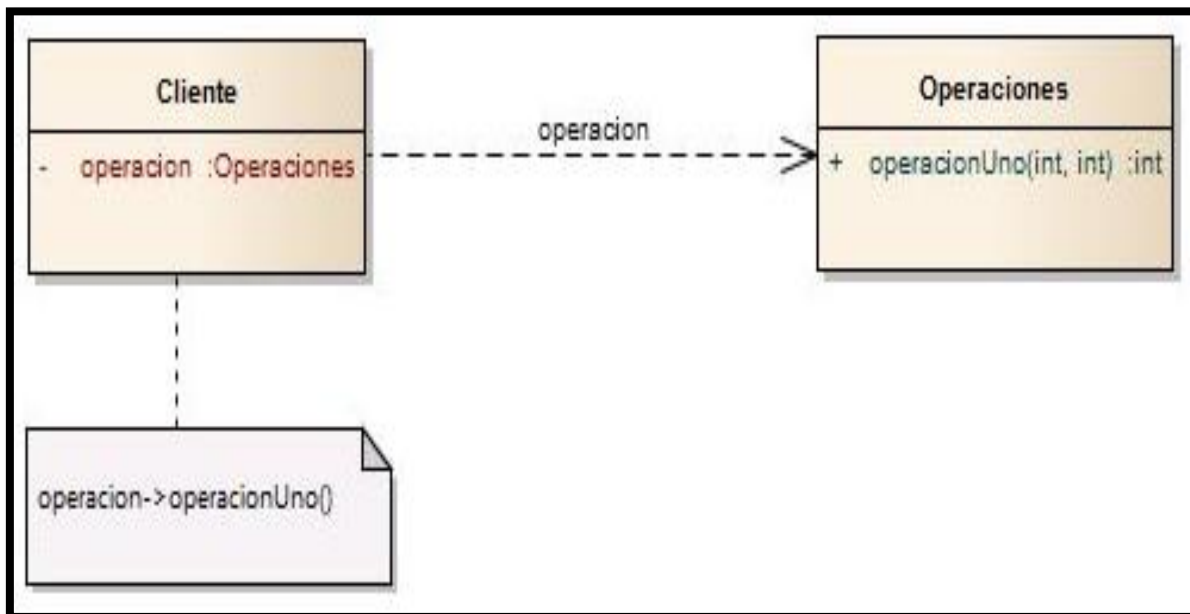


Figura 2.15 Diagrama de clases (representación de invocación).

```
int main()
{
    Operaciones *operacion = new Operaciones();
    int resultado = 0;

    resultado = operacion->operacionCompuesta(5,10);
    std::cout<<"Resultado: "<<resultado<<std::endl;

    Operaciones *calcular = new Operaciones();

    resultado = calcular->operacionUno(6,2);
    std::cout<<"Resultado: "<<resultado;

    int n;
    std::cin>>n;
    return 0;
}
```


Figura 2.16 Método main() donde se realizan llamadas externas a métodos que se mueven a nuevas clases

Cuando se cumplen las validaciones anteriores se puede crear la herencia y se continúa con el paso 7, porque necesitamos crear la clase padre. En caso del ejemplo que estamos analizando no es necesario crear una herencia, es decir no se necesita crear una clase padre, por esta razón, el método de refactorización continua en el paso 8.

2.3.2.7. Paso 7: Crear clase padre.

Se obtiene el nombre del primer método que contiene la secuencia interactiva de métodos, le agregamos la palabra *Base* y con esto formamos el nombre de la clase padre. El nombre del archivo es el mismo que el nombre de la clase padre, en este caso solo debemos agregar al final .h para especificar que se crea un archivo de tipo *header*.

2.3.2.8. Paso 8: Crear clase hija.

En este paso se crea la clase hija o simplemente una clase. Primero se consulta el identificador de la clase a la que pertenecen los elementos (métodos o variables) que se desean mover a una nueva clase, se consulta si en esa clase existen otros elementos diferentes a los que se desean mover a nueva clase, si existen otros elementos el método continua con la creación de la nueva clase, de lo contrario los elementos se pueden quedar en esa clase y el método de refactorización MRACR termina su proceso.

Para crear una nueva clase el nombre se forma con la siguiente estructura, primero inicia con las letras *Cr (derivado del nombre* ***C****ohe***r***encia),* luego se agrega el nombre del primer método de la secuencia interactiva de métodos, más un número entero que indica la cantidad de clases creadas para la clase original. El nombre del archivo en el que se almacena la clase tiene el mismo nombre que se formó para la nueva clase, con la diferencia que se agrega al final *.h,* ver Figura 2.17.

```
#ifndef _CroperacionCompuesta1
#define _CroperacionCompuesta1

class CroperacionCompuesta1
  {

    public:
            CroperacionCompuesta1();
           ~CroperacionCompuesta1();

    public:
            int operacionCompuesta ( int valor1 , int valor2 );
            int operacionSimple ( int valor1 , int valor2 );

  };

#endif
```


Figura 2.17 Ejemplo de nueva clase

2.3.2.9. Paso 9: registrar herencias y directivas *inlcude.*

Este paso es ejecutado sólo cuando se crea una clase padre. Se registra el nuevo *include* en la clase hija, se crea el *include* iniciando con la palabra "*#include*" más el nombre de la clase padre. Este paso es importante para la creación de nuevos archivos, de este modo, cuando se crea un nuevo archivo ya se tienen los *inlcude* que deben estar en esa clase.

Si fue creada una clase padre, por consiguiente, se debe actualizar la clase hija con la derivación de la clase padre. En este caso obtenemos el nombre de la clase padre y se ingresa la nueva herencia en la clase hija.

2.3.2.10. Paso 10: actualizar elementos de la nueva clase

Los elementos que se pueden actualizar en este paso son: métodos, atributos de la clase, constructores y los archivos *include.* Primero se actualizan los métodos. Se revisa la matriz de secuencia interactiva de métodos y se obtienen los identificadores de los métodos que estarán en la nueva clase. En el caso de ejemplo que estamos analizando sólo dos métodos (*operacionCompuesta* y *operacionSimple*) son reubicados en una nueva clase.

Se busca en la matriz de atributos de la clase los identificadores de los atributos que se mueven a la nueva clase. En este ejemplo la clase *Operaciones* no tiene atributos declarados en la clase.

Por último en este proceso se actualizan los *include* que tiene la clase original a la nueva clase que fue creada.

2.3.2.11. Paso 11: crear y actualizar constructores.

Primero se busca si la clase original tiene declarado algún constructor. De no tener constructor, el método de refactorización continúa en el paso 12.

De lo contrario, cuando la clase original tiene constructor se revisan los parámetros que recibe y los atributos de la clase que son inicializados. Se busca si alguno de los atributos inicializados en el constructor original corresponde a uno de los atributos de la clase que se movieron a una nueva clase.

Si coincide algún atributo de la nueva clase con los atributos inicializados en el constructor original, entonces debemos crear un nuevo constructor en la nueva clase creada, pasando la línea de la variable inicializada del constructor original al nuevo constructor y pasar su respectivo parámetro.

Al final se debe actualizar el constructor de la clase original, eliminando los parámetros y variables inicializadas que se movieron al nuevo constructor.

2.3.2.12. Paso 12: actualizar las llamadas externas.

En este paso se revisa si existen llamadas externas a los métodos que se movieron a una nueva clase. Por cada método de la secuencia interactiva de métodos se buscan sus llamadas externas de manera correspondiente.

Si se encuentran llamadas externas, se obtiene la variable de referencia que se utiliza para realizar la llamada y se busca si es utilizada para invocar a otros métodos, esto indica si podemos eliminar esta variable de referencia y cambiarla por una nueva o debemos dejar esa variable de referencia y se crea una nueva variable de referencia.

Antes de crear la nueva variable de referencia se revisa si existe una variable de referencia del tipo y la instancia de la nueva clase. De existir una, sólo actualizamos la llamada de los métodos con el nombre de la variable de referencia encontrada. De no existir una variable con el tipo y la instancia de la nueva clase debemos crear una nueva variable de referencia.

Para crear una nueva variable referencia se revisa si la clase donde se realiza la llamada se encuentra el método main(), si se encuentra el método main(), la nueva variable de referencia que debemos crear debe ser a nivel método, de lo contrario la antigua variable.

El nombre de la variable de referencia es el mismo que el de la clase nueva, con la diferencia que ésta se encuentra en minúsculas, se busca si debe llevar argumentos, esto lo podemos saber si la clase nueva tiene declarado un constructor con parámetros. De existir el constructor y tener parámetros, se crea el argumento que debe llevar la nueva variable de referencia y actualiza el argumento de la variable antigua.

Registramos la nueva variable de referencia en su nivel correspondiente y se reconstruyen las llamadas externas con la nueva variable de referencia que fue creada. En el ejemplo que esta siendo analizado se crea una nueva variable de referencia y se reconstruye la llamada a uno de los métodos que se movieron a la nueva clase. En la Figura 2.18 se observan las modificaciones que se le realizaron al método *main().*

```
#include <iostream>
#include "Operaciones.h"
#include "CroperacionCompuesta1.h"

int main()
{

    CroperacionCompuesta1 *croperacioncompuesta1 = new CroperacionCompuesta1();

    int resultado = 0;

        resultado = croperacioncompuesta1->operacionCompuesta(5,10);
    std::cout<<"Resultado: "<<resultado<<std::endl;

    Operaciones *calcular = new Operaciones();

    resultado = calcular->operacionUno(6,2);
    std::cout<<"Resultado: "<<resultado;

    int n;
    std::cin>>n;
    return 0;
}
```

Archivos incluidos

Nueva variable de referencia declarada de tipo puntero

Modificación del objeto con el que se realiza la llamada

Figura 2.18 Ejemplo de las modificaciones realizadas al método main()

2.3.3. Método de refactorización de ordenamiento de clases (MROCL).

Este método de refactorización es ejecutado cuando terminan los métodos de refactorización anteriores (MRACS y MRACR). El objetivo de este método es buscar las clases que se encuentran en un mismo archivo, en la Figura 2.19 se presentan los pasos que sigue MROCL.

Para explicar de mejor manera este método de refactorización se retomará el ejemplo de la Figura 2.7. El cual consiste en un archivo que contiene una clase con el nombre *Temp* y dentro del mismo archivo se encuentra el método *main().*

Antes de iniciar la descripción del proceso de MROCL, es importante mencionar que la función *main()* debe ser considerada como una clase, por lo que, se crea una clase fantasma con el nombre *main,* que contiene sólo a esta función, esto se hace para evitar conflictos al momento de aplicar la métrica LCOM4 o la métrica CR (Coherencia).

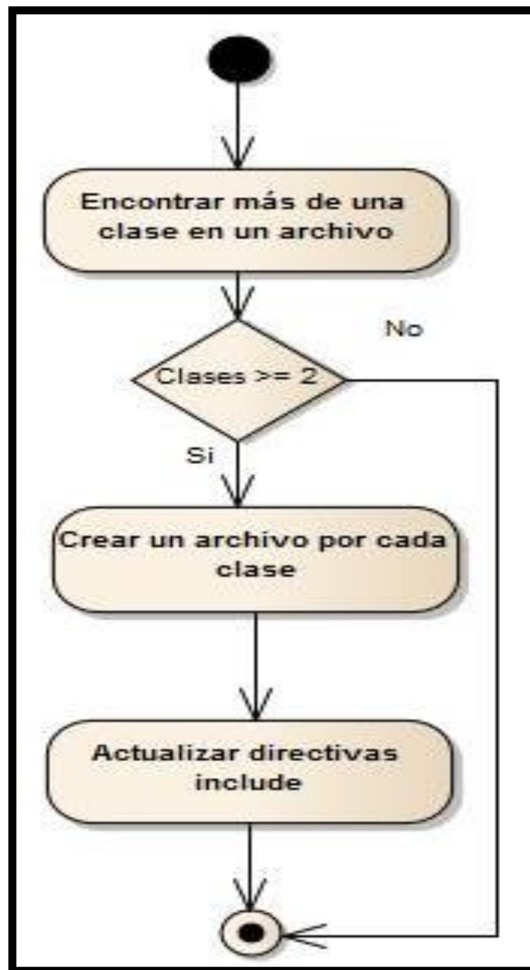


Figura 2.19 Diagrama de actividades del MROCL

2.3.3.1. Paso 1: encontrar más de una clase en un archivo.

MROCL consulta todos los archivos que se están analizando y por cada archivo se analiza si existe más de una clase declarada dentro del mismo archivo. Cuando se encuentra más de una clase dentro de un mismo archivo, en ese momento las clases son separadas en archivos diferentes.

En el caso del ejemplo de la Figura 2.7 se encuentran dos clases en el mismo archivo, la clase *Temp* y la clase fantasma *Main* para el método main(). Se obtiene el nombre del archivo sin la extensión y se determina si coincide con alguna de las clases que se encuentran dentro del mismo archivo.

Cuando se encuentra una clase con el nombre del archivo, significa que esa clase debe permanecer en el archivo. De no encontrar una clase que coincida con el nombre del archivo, la primera clase que se encuentre declarada es la que se queda en el archivo original y las otras clases son las que se deben mover a un nuevo archivo. En el ejemplo analizado sería la clase *Temp* la que se quedaría en el archivo con el nombre *Temp* y la clase *Main* se mueve a una nueva clase.

2.3.3.2. Paso2: crear un archivo por cada clase.

En este paso se crean los archivos para cada una de las clases que se encuentran en el mismo archivo. El nombre de cada archivo nuevo se forma con el nombre de la clase y se le agrega la extensión *.h.* Cuando se encuentra que la clase es con el nombre *Main* la extensión del archivo cambia a *.cpp,* porque en esta clase sólo se encuentra el método *main()* y es un archivo con métodos implementados, por esta razón debe terminar con la extensión *.cpp.*

2.3.3.3. Paso 3: actualizar las directivas *include.*

Se copian las directivas *include* que tiene el archivo de la clase original que se encuentra en el método de refactorización y se llevan al nuevo archivo creado. Se busca si en otras clases existen variables de referencia con el tipo o instancia de la clase que se movió a un nuevo archivo, si existen se inserta la directiva *inlcude* del nuevo archivo en la clase donde se crea una variable de tipo o instancia de la clase que ahora se encuentra en un nuevo archivo.

DESCRIPCIÓN DE LA HERRAMIENTA

Capítulo 3

En este capítulo se presenta el diagrama de componentes y las pantallas de la herramienta desarrollada que permite el equilibrio entre los valores de cohesión y coherencia.

3.1. Diagrama de componentes.

La herramienta desarrollada en el trabajo de tesis se encuentra integrada al sistema SR2 “Sistema de Reingeniería de Software Legado para Reuso”. En la Figura 3.1 se presenta el diagrama de componentes de la herramienta desarrollada en esta tesis, que se compone de cinco módulos:

- Analizador.
- Base de datos.
- Métricas.
- Métodos de refactorización.
- Generador de código.

La implementación de la herramienta fue desarrollada en el lenguaje de programación Java, se utilizó JavaCC para la generación del analizador y las tablas de información son gestionadas por el sistema manejador de base de datos MySQL.

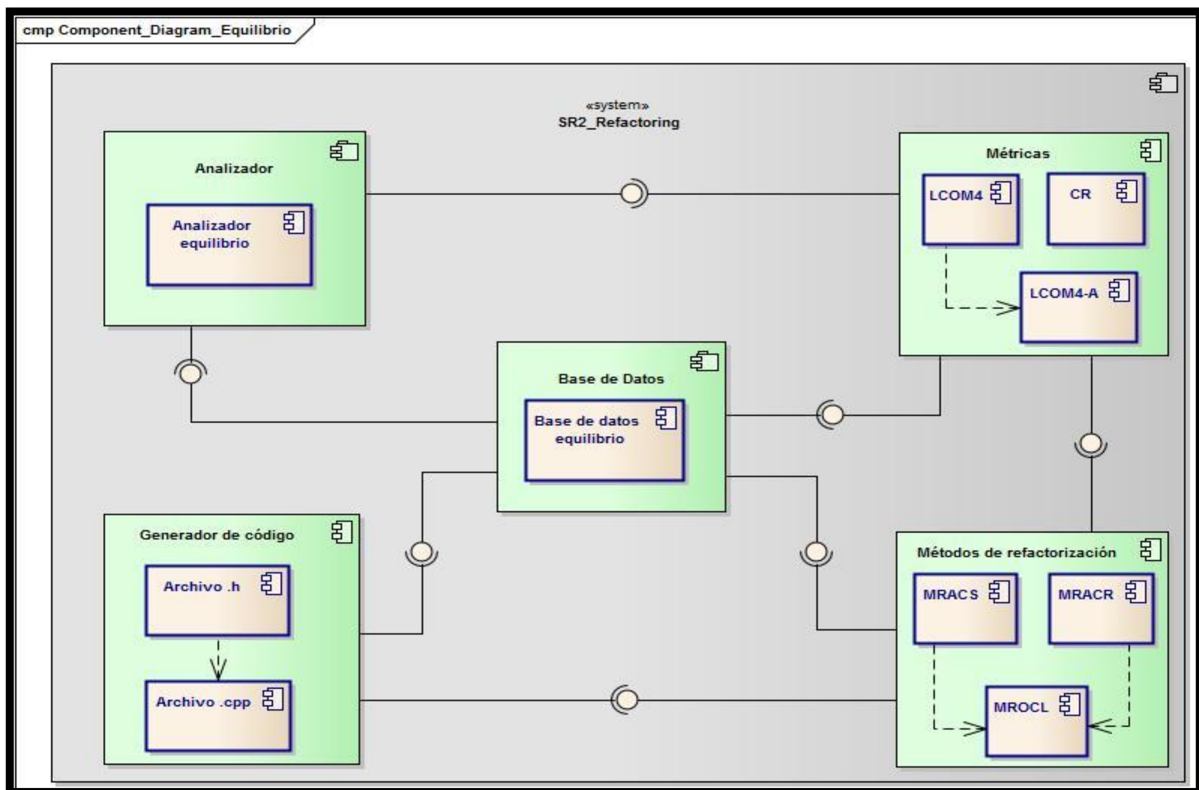


Figura 3.1 Diagrama de componentes de la herramienta desarrollada en este trabajo de investigación.

3.2. Interfaz de la herramienta

En esta sección se describen las interfaces graficas de la herramienta desarrollada y su funcionamiento. En la Figura 3.2 se presenta la pantalla cuando se inicia el sistema SR2 Refactoring. En ella el usuario debe ingresar sus datos correctos para poder acceder a la funcionalidad que brinda el SR2 Refactoring.

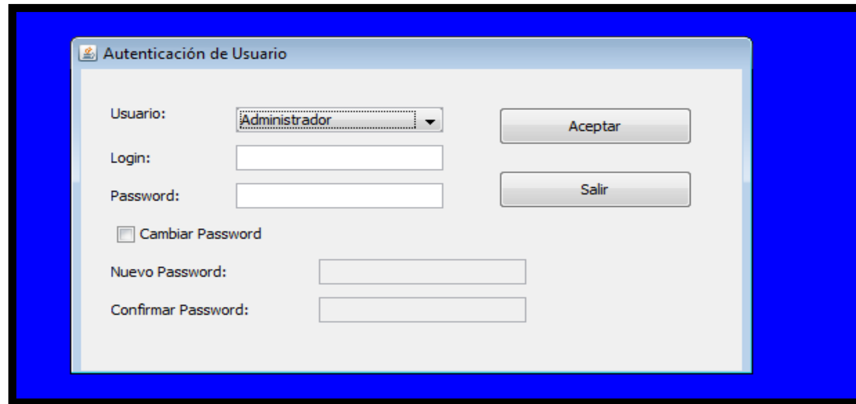


Figura 3.2 Pantalla de autenticación del SR2 refactoring.

En el menú de refactorización se resaltan las opciones desarrolladas en este trabajo de investigación y agregadas al SR2. Las pantallas son: Método de Equilibrio (Alta cohesión y Alta Coherencia), Método para lograr Alta Cohesión y Método para lograr Alta Coherencia, ver Figura 3.3.

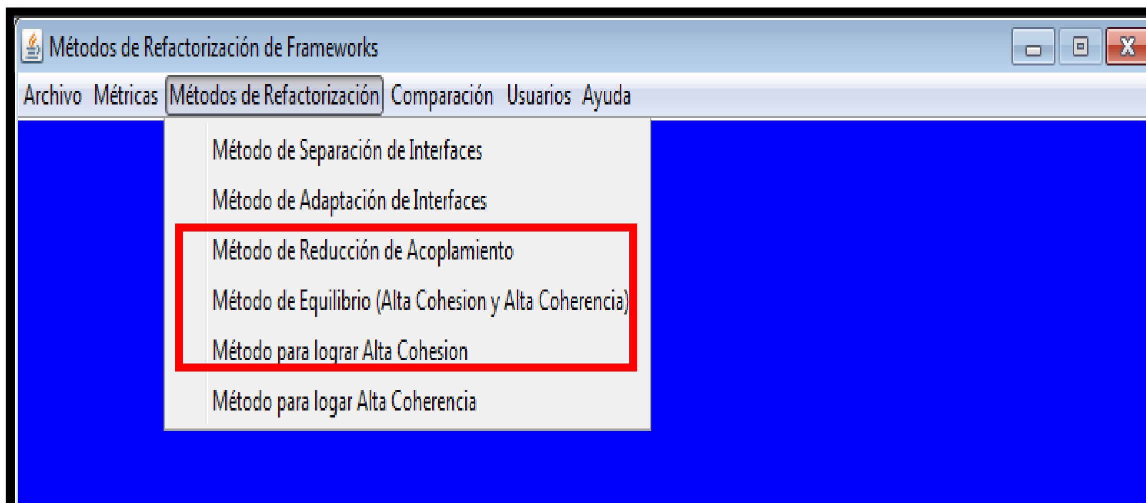


Figura 3.3 Pantalla principal del SR2 Refactoring, resaltando las nuevas pantallas agregadas.

Para utilizar cualquiera de las nuevas opciones primero en el menú de SR2 Refactoring pulsar la opción *Archivo* después pulsar en *Seleccionar Archivos Originales* y finalmente seleccionar los archivos que se quieren analizar ver Figura 3.4.

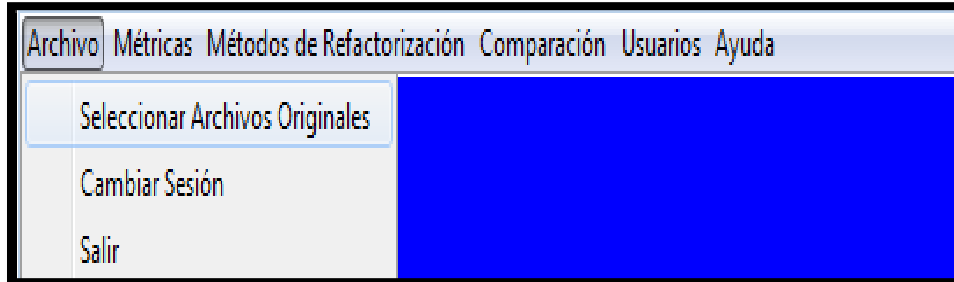


Figura 3.4 Pantalla para seleccionar archivos.

3.2.1. Pantalla Método de Equilibrio (Alta cohesión y Alta coherencia).

El siguiente paso es seleccionar cualquiera de las nuevas opciones desarrolladas en el trabajo de tesis. En la Figura 3.5 se presenta la pantalla de Método de Equilibrio (Alta Cohesión y Alta Coherencia), la pantalla tiene tres secciones: Bitácora (Sección 1); indica la ruta y nombre de los archivos analizados, Tabla 1 (Sección 2); se presentan los resultados de aplicar las métricas LCOM4, LCOM4-A y CR, Tabla 2 (Sección 3); se presentan los resultados después de aplicar el proceso de refactorización.

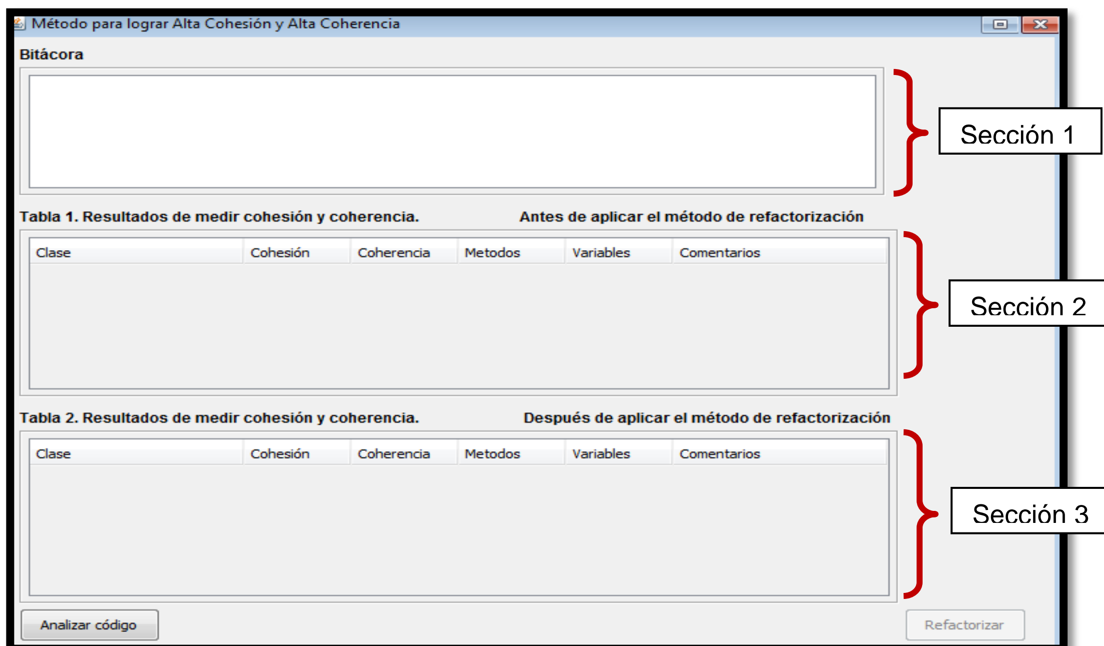


Figura 3.5 Pantalla Método de Equilibrio (Alta cohesión y Alta coherencia).

La Figura 3.6 presenta la pantalla después de pulsar en el botón Analizar código. Se observa los valores de cohesión y coherencia, una celda de color verde significa que es un nivel deseado, celda de rojo siginifica que se obtuvo un nivel no deseado y celda de amarillo: significa que la clase no implementa métodos. Cuando se encuentra un nivel no deseado de cohesión o coherencia automaticamente sugiere aplicar el proceso de refactorización.

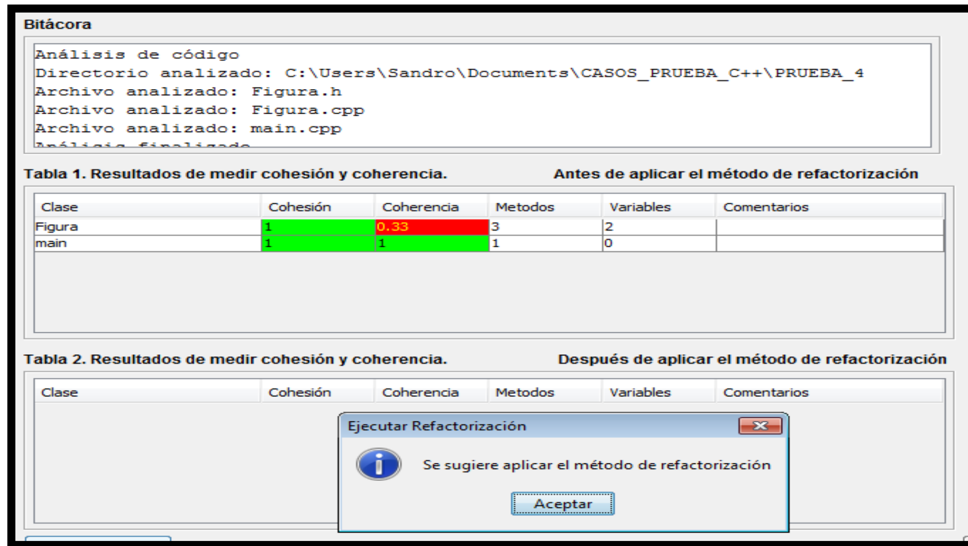


Figura 3.6 Pantalla Método de Equilibrio después de pulsar en el botón Analizar código

La Figura 3.7 presenta la pantalla después de aplicar el proceso de refactorización.

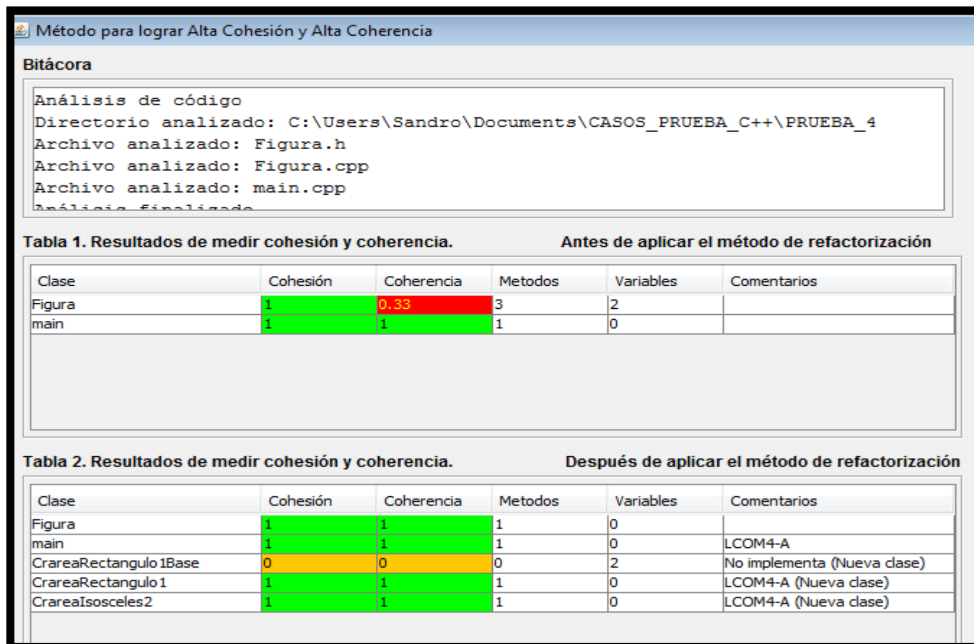


Figura 3.7 Pantalla después de aplicar el proceso de refactorización.

3.2.2. Pantallas: Método para lograr Alta Cohesión y Método para lograr Alta Coherencia.

A continuación se presentan las otras dos opciones que se agregaron al menu de métodos de refactorización. La primer pantalla se encarga de medir cohesión y aplicar el método de refactorización para logar alta cohesión. La segunda pantalla se encarga de medir coherencia y aplicar el método de refactorización para logar alta coherencia. Ambas pantallas tienen las mismas secciones mencionadas en (§3.2.1).

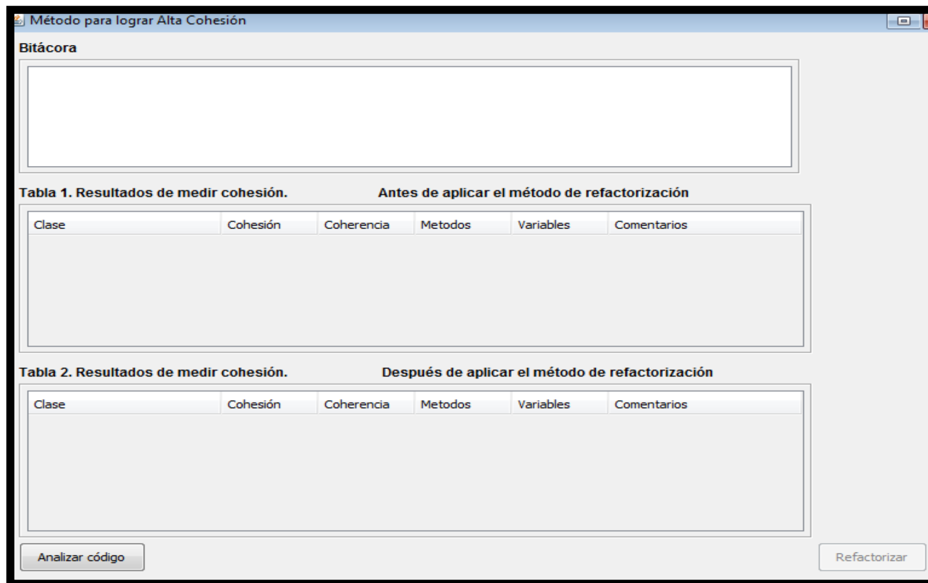


Figura 3.8 Pantalla del Método para lograr Alta Cohesión.

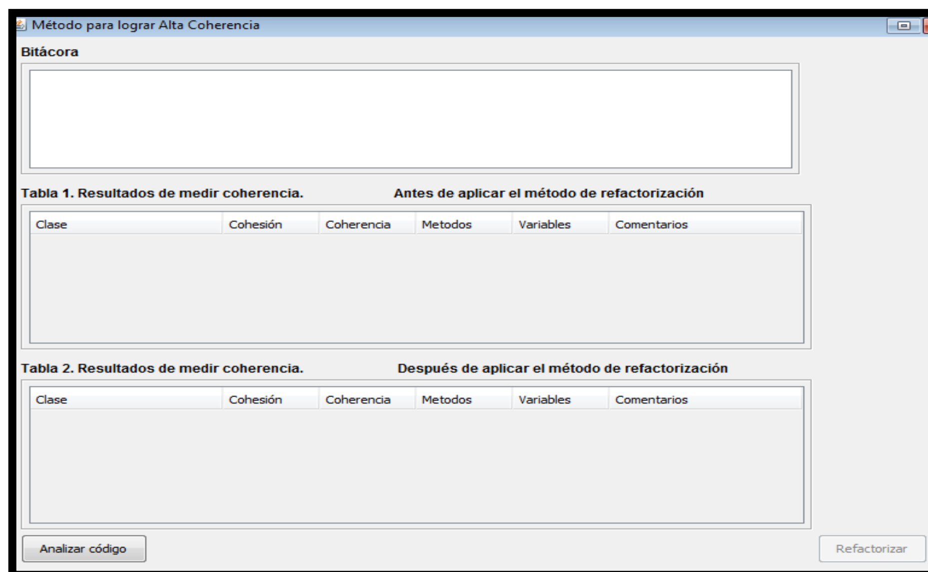


Figura 3.9 Pantalla de Método para lograr Alta Coherencia.

PRUEBAS E INTERPRETACIÓN DE RESULTADOS

Capítulo 4

En este capítulo se presentan los casos de prueba utilizados para evaluar el funcionamiento de la herramienta que logra el equilibrio entre los valores de cohesión y coherencia.

4.1. Caso de prueba 1: ejemplo de una clase.

La Figura 4.3 presenta el diagrama de secuencia del caso de prueba 1 el cual fue encontrado en la web [Walia, 2016] desarrollado en lenguaje C++. En un archivo se encuentra el método *main()* y una clase con el nombre *Programming*. La clase contiene un atributo y dos métodos. En el main se crea un objeto de tipo clase (*Programming*) para llamar a sus métodos.

Tabla 4.1 Caso de prueba 1

<table>
<tr><td>Objetivo:</td><td colspan="3">Probar funcionalidad básica de la herramienta desarrollada (ver Figura 4.1).</td></tr>
<tr><td>Características a probar:</td><td colspan="3">Cálculo de las métricas cohesión y coherencia (Botón: Analizar Código).
Presentación de los resultados de las tablas (Tabla 1 y Tabla 2).
Proceso de refactorización (Botón: Refactorizar).</td></tr>
<tr><td>Entrada:</td><td colspan="3">Un archivo con una clase y método main().</td></tr>
<tr><td>Resultados esperados:</td><td colspan="3"></td></tr>
<tr><td rowspan="3"></td><td></td><td>Antes de refactorizar</td><td>Después de refactorizar</td></tr>
<tr><td>Grado de cohesión</td><td>1</td><td>1</td></tr>
<tr><td>Grado de coherencia</td><td>0.5</td><td>0.5</td></tr>
<tr><td>Procedimiento de prueba:</td><td colspan="3">Seleccionar archivo a refactorizar
Pulsar en la opción Método para lograr alta cohesión y alta coherencia.
Pulsar botón Analizar código.
Pulsar botón Refactorizar.</td></tr>
<tr><td colspan="4">Resultados después de refactorizar</td></tr>
<tr><td>Resultado:</td><td colspan="3"></td></tr>
<tr><td rowspan="2"></td><td>Grado de cohesión:</td><td colspan="2">1</td></tr>
<tr><td>Grado de coherencia:</td><td colspan="2">0.5</td></tr>
<tr><td>Estado del caso de prueba:</td><td colspan="3">Exitoso</td></tr>
<tr><td>Observaciones:</td><td colspan="3">El grado de coherencia no fue mejorado porque el MRACR no tiene el comportamiento adecuado para este caso de prueba.</td></tr>
</table>

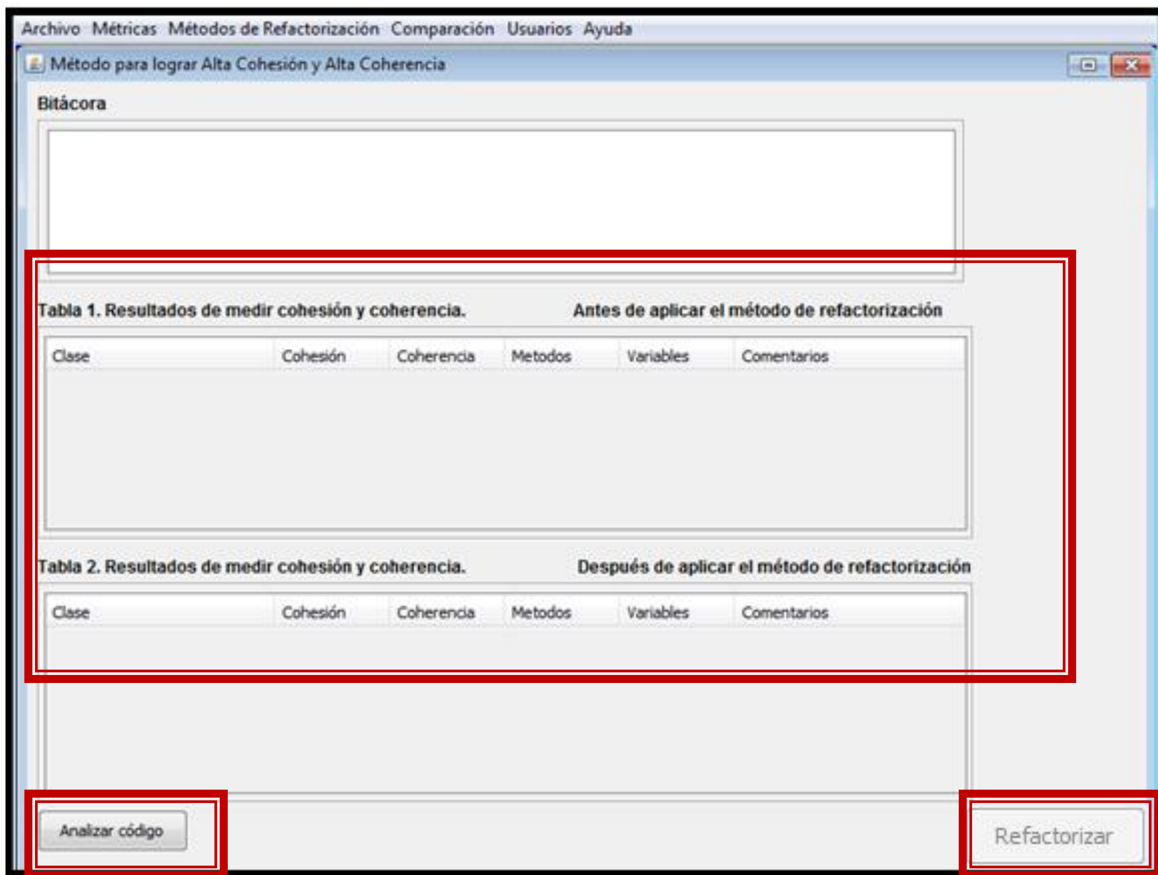


Figura 4.1. Pantalla método de refactorización equilibrio.

La Figura 4.2 se observa como son representados los valores de cohesión y coherencia, los colores de las celdas tienen un significado: el color verde; significa que la clase tiene alta cohesión o alta coherencia dependiendo en la columna que se muestre, el color rojo; significa que la clase tiene valores no deseados de cohesión y de coherencia respectivamente, el color amarillo; significa que la clase no tiene métodos implementados, esto quiere decir, que la herramienta encontró una clase abstracta y por lo tanto, no tiene métodos implementados o puede ser una clase que no implementa métodos y no puede ser considerada para medir la cohesión y/o coherencia.

cohesión y coherencia. Antes de aplicar el método de refactorización

	Cohesión	Coherencia	Metodos	Variables	Comentarios
	0	0	0	0	Metodos no implementados
	1	1	1	4	
	2	0.5	2	6	
	1	1	1	2	
	1	0.5	2	6	

Figura 4.2. Representación de valores de cohesión y coherencia

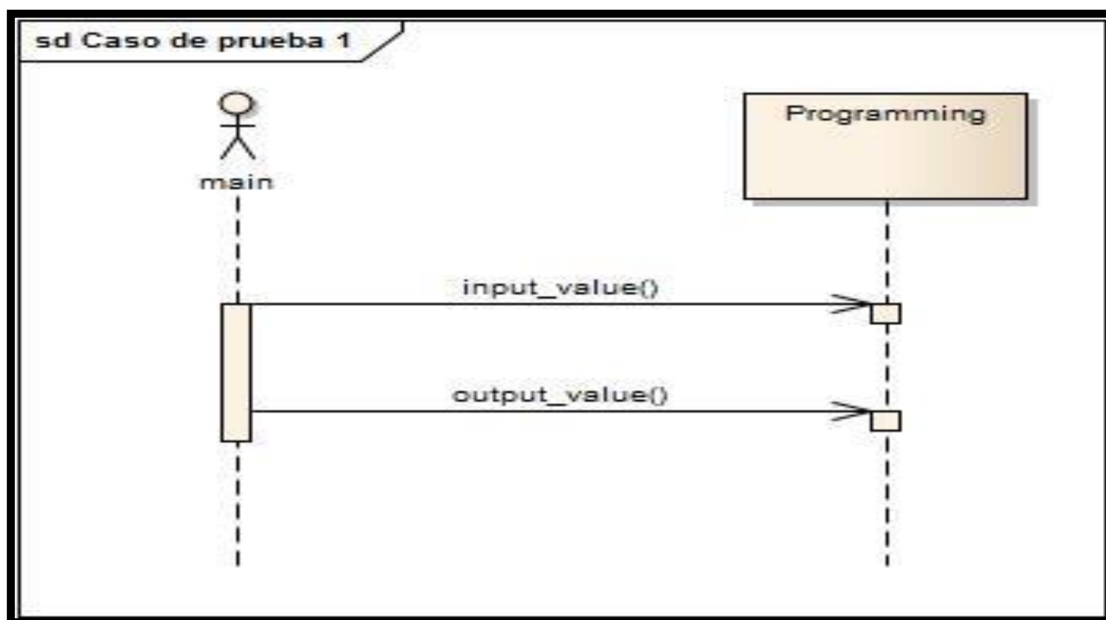


Figura 4.3 Diagrama de secuencia del caso de prueba 1

La Figura 4.4 presenta la pantalla de la herramienta después de seleccionar el archivo del caso de prueba 1 y pulsar en el botón *Analizar código*. Se observa una tabla donde se presentan los nombres de las clases, grado de cohesión y coherencia, métodos, variables y comentarios. En el caso de prueba 1 la herramienta da como resultado 1 en ambas clases por parte de cohesión, en la columna de coherencia se observa una celda de color rojo indicando que no existe coherencia en esa clase. De acuerdo a estos resultados la herramienta sugiere que se debe aplicar el método de refactorización para mejorar el grado de coherencia.

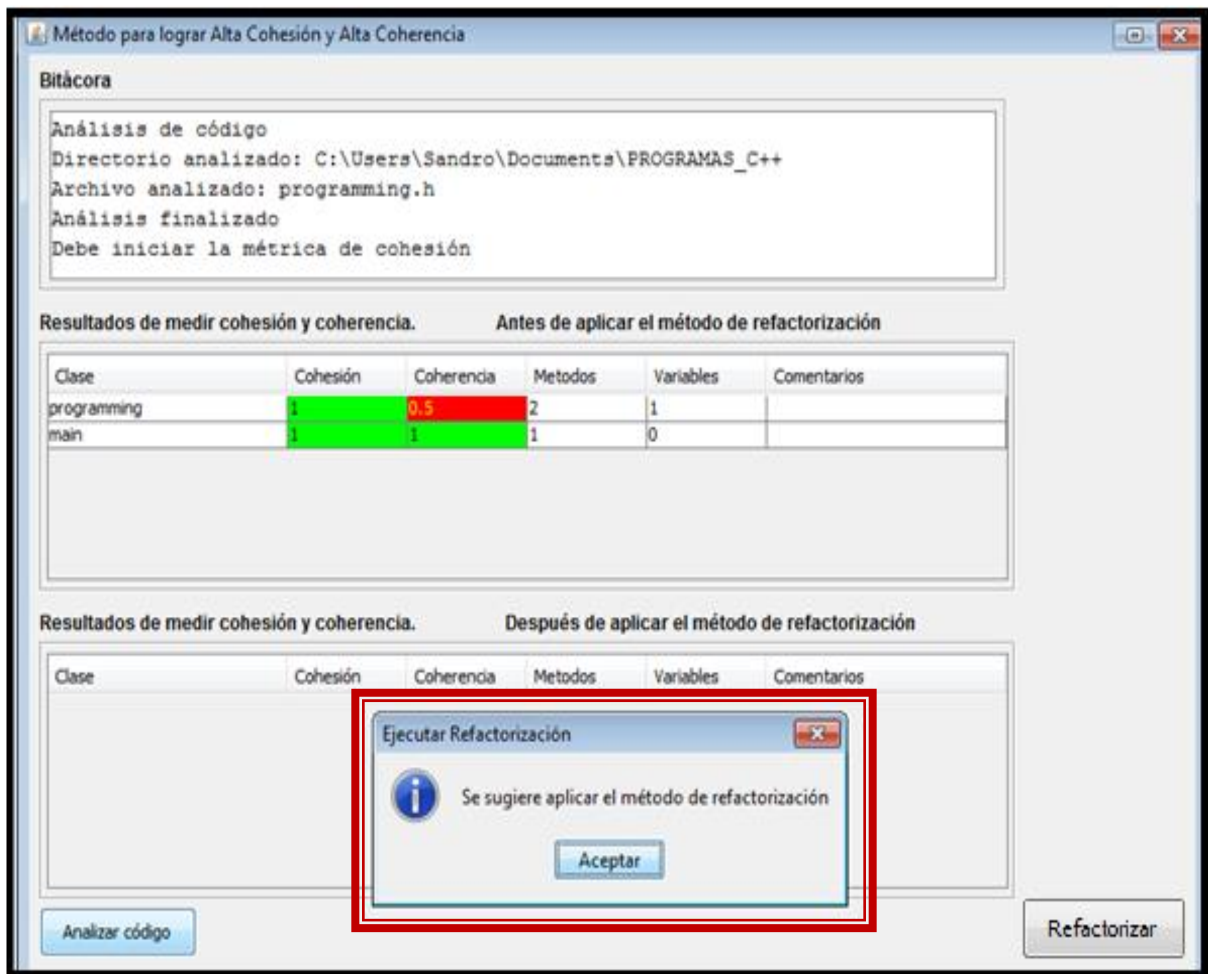


Figura 4.4. Resultados de medir cohesión y coherencia

La Figura 4.5 presenta los resultados después de aplicar el proceso de refactorización. Se observa que no existen diferencias entre la tabla 1 y la tabla 2. La clase *Programming* sigue con el grado de coherencia de 0.5. y contiene el mismo número de métodos y variables.

El último método en el proceso de refactorización es el método de refactorización de ordenamiento de clases (MROCL). Iniciamos con el paso 1 (§2.3.3.1), en este caso de prueba 1 encontramos que se encuentran dos clases en un mismo archivo, la explicación del porque el método *main()* es considerado como una clase se encuentra en (§2.3.3). Se ejecuta el paso 2 (§2.3.3.2) y paso 3 (§2.3.3.3) para finalizar con MROCL.

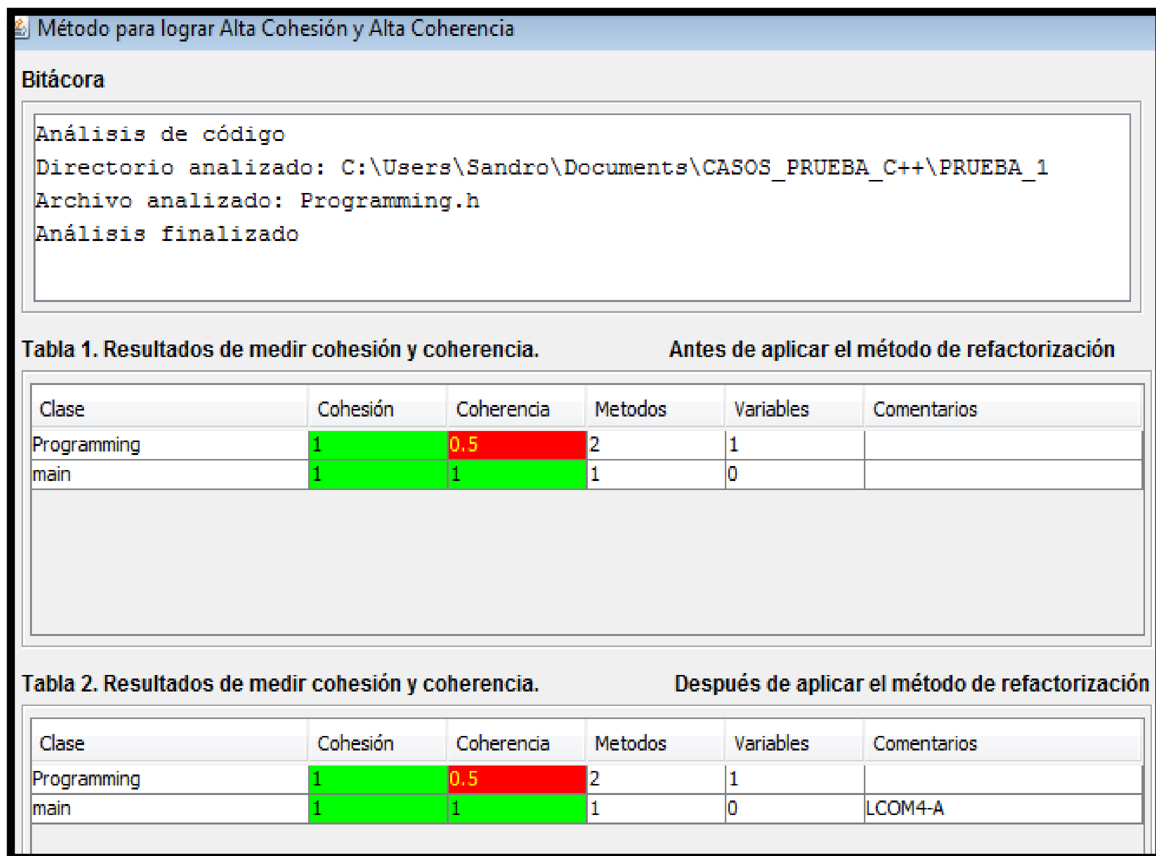


Figura 4.5. Resultados después de aplicar el proceso de refactorización.

En el anexo B (§Figura B.6) se presentan los archivos que son generados por MROCL al finalizar el proceso de refactorización. El caso de prueba 1 tenía solo un archivo que contenía una clase y el main. Ahora en la (§Figura B.6) se observa que tenemos tres archivos, el método *main()* se encuentra en un archivo y la clase *Programming* fue dividida en dos archivos: uno con la extensión *.h;* donde se encuentran declarada la variable *number* y los prototipos de los dos métodos *input_value()* y *output_value()*, el segundo archivo es con la extensión *.cpp*; su contenido es la implementación de los dos métodos *input_value()* y *output_value().*

4.2. Caso de prueba 2: clase con baja cohesión y baja coherencia.

Tabla 4.2 Caso de prueba 2

Objetivo:	Validar y verificar el comportamiento del proceso de refactorización al encontrarse con un programa que tiene baja cohesión y baja coherencia.		
Características a probar:	• Cálculo de las métricas cohesión y coherencia • Presentación de los resultados de las tablas. • Proceso de refactorización.		
Entrada:	Un archivo con una clase y método *main()*.		
Resultados esperados:			
		Antes de refactorizar	**Después de refactorizar**
	Grado de cohesión	2	1
	Grado de coherencia	0.5	1
Procedimiento de prueba:	1. Seleccionar archivo a refactorizar 2. Pulsar en la opción Método para lograr alta cohesión y alta coherencia. 3. Pulsar botón Analizar código. 4. Pulsar botón Refactorizar.		
Resultados después de refactorizar			
Resultado:			
	Grado de cohesión:	1	
	Grado de coherencia:	1	
Estado del caso de prueba:	**Exitoso**		
Observaciones:	Se crea una nueva clase y un total de 5 archivos.		

El código completo del programa del caso de prueba 2 se encuentra en el anexo B (§Figura B.7). El objetivo de este programa es imprimir el número que el usuario ingrese desde la pantalla y consiste en un archivo con la clase *Temp* y el método *main().* La clase *Temp* contiene dos variables *data1* y *data2*, y dos métodos *int_data(int d)* y *float_data()*. En el método *main()* se crean dos objetos de tipo clase (*Temp*) y el método main() invoca a un método de cada uno de los objetos. El diagrama de secuencia del caso de prueba 2 se presenta en la Figura 4.6.

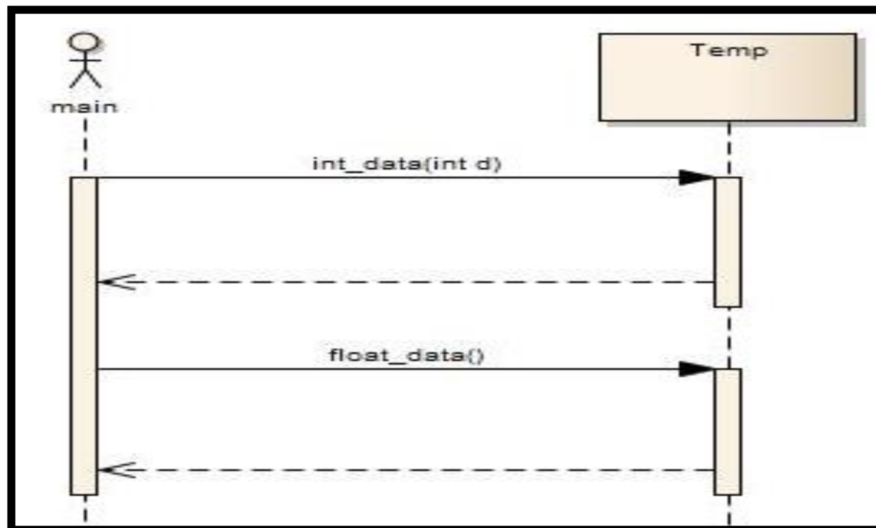


Figura 4.6 Diagrama de secuencia del caso de prueba 2.

En la tabla 2 de la Figura 4.7 presenta los resultados después de pulsar en el botón Refactorizar, Se observa que el grado de cohesión y coherencia mejoró, incluso se obtuvo los valores más altos de amabas métricas (LCOM4 y CR).

Método para lograr Alta Cohesión y Alta Coherencia

Bitácora

```
Análisis de código
Directorio analizado: C:\Users\Sandro\Documents\CASOS_PRUEBA_C++\PRUEBA_2
Archivo analizado: Temp.h
Análisis finalizado
```

Tabla 1. Resultados de medir cohesión y coherencia. **Antes de aplicar el método de refactorización**

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Temp	2	0.5	2	2	
main	1	1	1	0	

Tabla 2. Resultados de medir cohesión y coherencia. **Después de aplicar el método de refactorización**

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Temp	1	1	1	1	
main	1	1	1	0	LCOM4-A
Cohesion82_Temp	1	1	1	1	Nueva clase

Analizar código Refactorizar

Figura 4.7 Resultados después de aplicar el proceso de refactorización.

Cuando inicia el proceso de refactorización en el código del caso de prueba 2 primero aplica el método MRACS el cual ejecuta los ocho pasos que se describen en (§2.3.1) para lograr clases con alta cohesión. El siguiente método que se debe aplicar es MRACR aplicando los pasos del uno al tres (§2.3.2.3) y finaliza sin crear nuevas clases, esto porque al aplicar MRACS el problema de coherencia es resuelto. En la Figura 4.8 se observa el diagrama de secuencia después de pasar por el proceso de refactorización.

Por último el método MROCL se encarga de separar las clases en diferentes archivos como se muestra en la (§Figura B.14). Se puede observar como cada clase tiene su archivo con extensión *.h* y *.cpp* claro esto a excepción del método *main().*

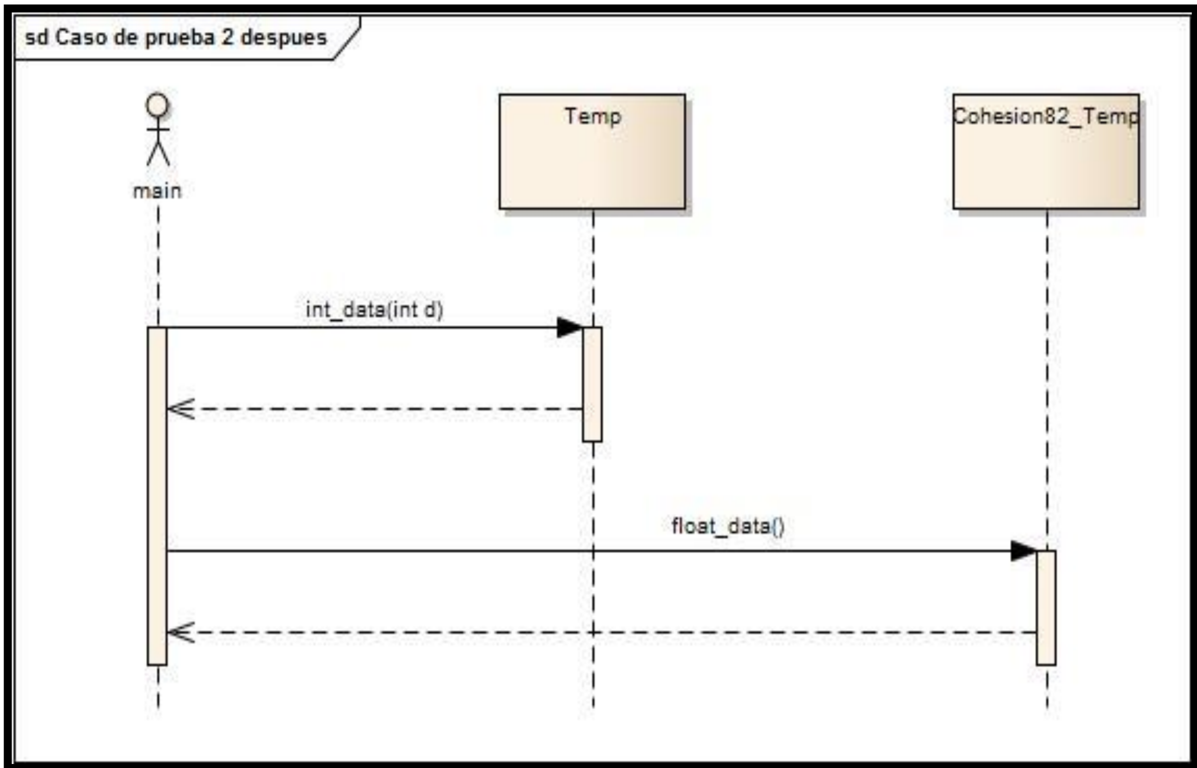


Figura 4.8 Diagrama de secuencia después de pasar por el proceso de refactorización.

4.3. Caso de prueba 3: clase con dos secuencias interactivas.

Tabla 4.3 Caso de prueba 3

Objetivo:	Observar el comportamiento del método de refactorización MRACR cuando encuentra secuencias interactivas		
Características a probar:	• Cálculo de las métricas cohesión y coherencia • Presentación de los resultados de las tablas. • Proceso de refactorización.		
Entrada:	Un archivo con una clase, método *main()* y dos secuencias interactivas de métodos.		
Resultados esperados:			
		Antes de refactorizar	**Después de refactorizar**
	Grado de cohesión	2	1
	Grado de coherencia	0.4	1
Procedimiento de prueba:	1. Seleccionar archivo a refactorizar 2. Pulsar en la opción Método para lograr alta coherencia. 3. Pulsar botón Analizar código. 4. Pulsar botón Refactorizar.		
Resultados después de refactorizar			
Resultado:			
	Grado de cohesión:	1	
	Grado de coherencia:	1	
Estado del caso de prueba:	**Exitoso**		
Observaciones:	Se crea una nueva clase y un total de 5 archivos.		

El caso de prueba 3 es un programa realizado con la finalidad de crear secuencias interactivas dentro de una clase. La clase Operaciones tiene cinco métodos *operacionSimple, operacionCompuesta, operacionDos, operacionTres* y *operacionUno* y no tiene atributos. Las invocaciones entre los métodos esta de la siguiente manera: *operacionCompuesta* manda a llamar al método *operacionSimple, operacionUno* invoca al método *operacionDos* y este manda a llamar al método *operacionTres.* Ver Figura 4.9, el código completo se puede encontrar en el anexo B (§ Figura B.15).

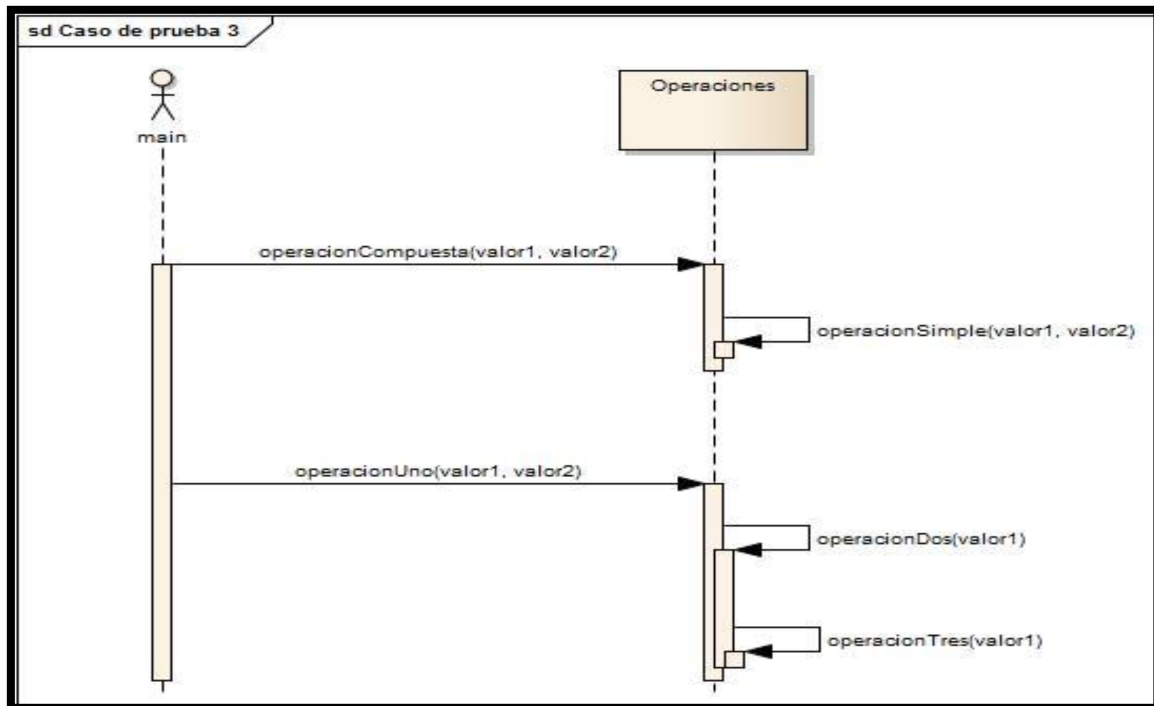


Figura 4.9 Diagrama de secuencia del caso de prueba 3.

En el caso de prueba 3 el método *main()* tiene dos variables de referencia tipo Operaciones, el método main() invoca al método *operacionCompuesta* y *operacionUno* con su respectiva variable de referencia (*operación* y *calcular*).

Como el propósito del caso de prueba es ver el comportamiento del método MRACR (§2.3.2), Se decide seleccionar en el menú métodos de refactorización el método para lograr alta coherencia. La Figura 4.10 presenta que el grado de coherencia fue mejorado y se crea una nueva clase. Se puede ver en la tabla 1 de la Figura 4.10 en la columna métodos que la clase *Operaciones* tenía cinco métodos, después de aplicar MRACR en la tabla 2 de la misma figura se puede ver que la clase *Operaciones* en la columna Métodos tiene ahora solo 2 y la nueva clase creada tiene 3 métodos, la responsabilidad de la clase fue dividida.

Tabla 1. Resultados de medir coherencia. **Antes de aplicar el método de refactorización**

Clase	Coherencia	Metodos	Variables	Comentarios
Operaciones	0.4	5	0	
main	1	1	0	

Tabla 2. Resultados de medir coherencia. **Después de aplicar el método de refactorización**

Clase	Coherencia	Metodos	Variables	Comentarios
Operaciones	1	3	0	
main	1	1	0	LCOM4-A
CroperacionCompuesta1	1	2	0	Nueva clase

Figura 4.10 Resultado después de aplicar MRACR

La Figura 4.11 presenta el diagrama de secuencia después pasar por el proceso de refactorización, se observa la nueva clase creada *Cohesion22_Operaciones* para separar las responsabilidades, es decir, cada clase tiene una secuencia interactiva de métodos (una única responsabilidad).

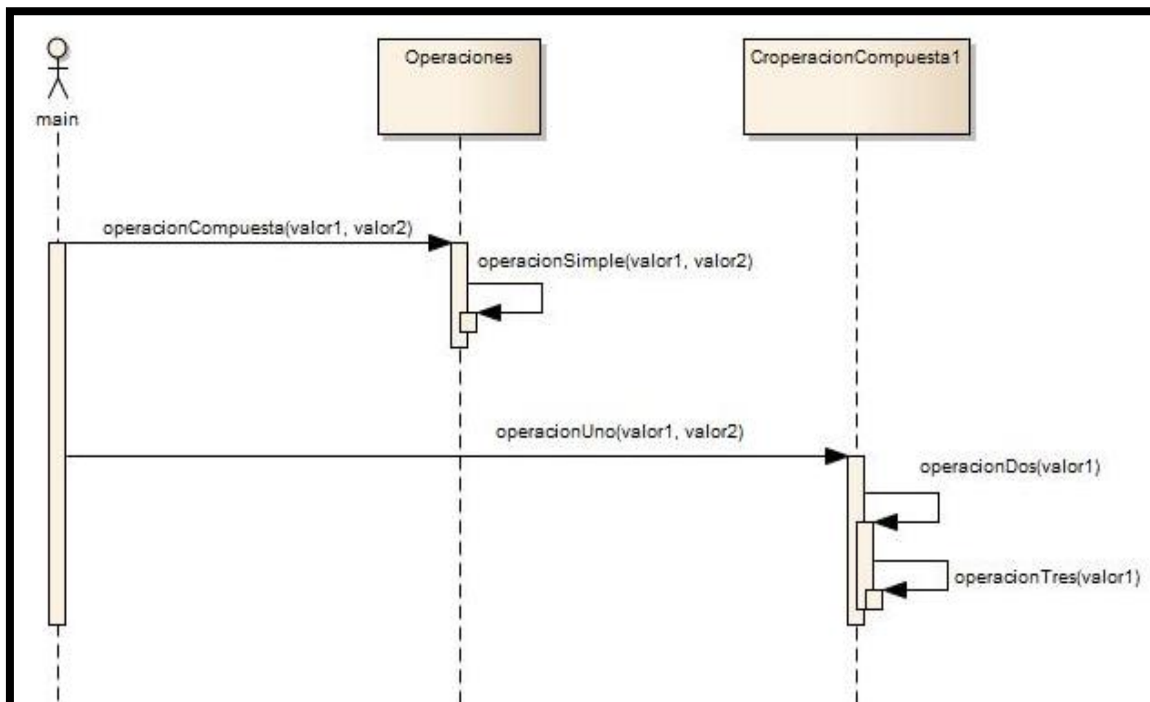


Figura 4.11 Diagrama de secuencia después de pasar por el método de refactorización MRACR.

4.4. Caso de prueba 4: tres archivos, una clase, alta cohesión y baja coherencia.

Tabla 4.4 Caso de prueba 4

Objetivo:	Observar el comportamiento del método de refactorización MRACR cuando encuentra métodos que comparten atributos de la clase y no existe invocación entre los métodos.		
Características a probar:	• Cálculo de las métricas cohesión y coherencia • Presentación de los resultados de las tablas. • Proceso de refactorización.		
Entrada:	Tres archivos con una clase y método *main()*.		
Resultados esperados:			
		Antes de refactorizar	**Después de refactorizar**
	Grado de cohesión	1	1
	Grado de coherencia	0.33	1
Procedimiento de prueba:	1. Seleccionar archivo a refactorizar 2. Pulsar en la opción Método para lograr alta cohesión y alta coherencia. 3. Pulsar botón Analizar código. 4. Pulsar botón Refactorizar.		
Resultados después de refactorizar			
Resultado:			
	Grado de cohesión:	1	
	Grado de coherencia:	1	
Estado del caso de prueba:	**Exitoso**		
Observaciones:	Se crean tres clases nuevas y un total de 8 archivos.		

En el programa del caso de prueba 4 la clase *Figura* tiene declarado dos atributos de tipo double: *height* y *width.* También tiene tres métodos: *areaRectangulo, areaIsosceles* y *areaCylinder.* La Figura 4.12 presenta el diagrama de clases y en el anexo B (§ Figura B.23, Figura B.24 y Figura B.25) se encuentra el código completo del caso de prueba 4, el cual consta de tres archivos: *main.cpp, Figura.h y Figura.cpp.*

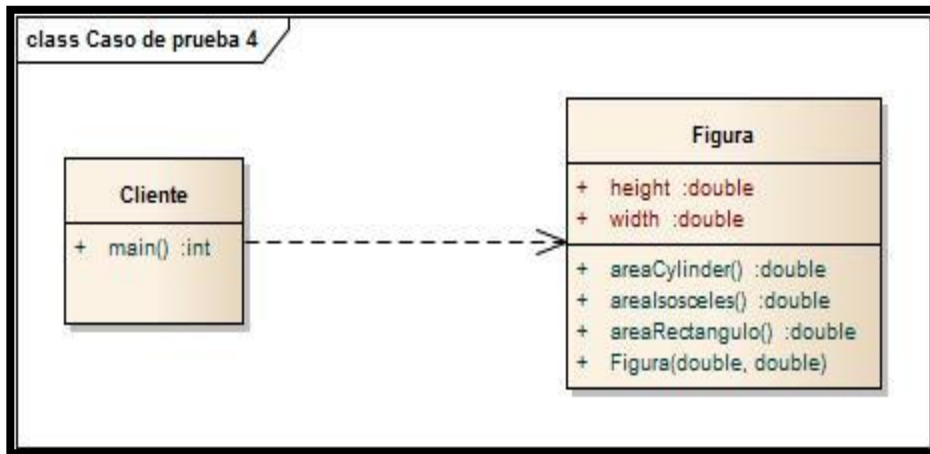


Figura 4.12 Diagrama de clases del caso de prueba 4.

La Figura 4.13 presenta los resultados después de aplicar el proceso de refactorización. Se inicia aplicando el MRACS, en el paso 1 (§2.3.1.1) el método de refactorización termina porque no encuentra clases con baja cohesión. El siguiente método es el MRACR, aquí hacemos hincapié en el paso 6 (§2.3.2.6) porque es donde se decide si es posible crear clases heredadas, los pasos 11 (§2.3.2.11) y 12 (§2.3.2.12) son importantes porque ahí se crean los nuevos constructores de las nuevas clases y es donde se reconstruyen las llamadas a los métodos que se movieron a una nueva clase.

Bitácora

```
Análisis de código
Directorio analizado: C:\Users\Sandro\Documents\CASOS_PRUEBA_C++\PRUEBA_4
Archivo analizado: Figura.h
Archivo analizado: Figura.cpp
Archivo analizado: main.cpp
```

Tabla 1. Resultados de medir cohesión y coherencia. **Antes de aplicar el método de refactorización**

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Figura	1	0.33	3	2	
main	1	1	1	0	

Tabla 2. Resultados de medir cohesión y coherencia. **Después de aplicar el método de refactorización**

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Figura	1	1	1	0	
main	1	1	1	0	LCOM4-A
CrareaRectangulo1Base	0	0	0	2	No implementa (Nueva clase)
CrareaRectangulo1	1	1	1	0	LCOM4-A (Nueva clase)
CrareaIsosceles2	1	1	1	0	LCOM4-A (Nueva clase)

Figura 4.13 Resultados después de aplicar el proceso de refactorización.

Los archivos generados al terminar el proceso de refactorización se encuentran en el anexo B (§Figura B.35) resaltando que ahora tenemos un total de 8 archivos y el caso de prueba original tenía solo tres archivos. La Figura 4.14 presenta el diagrama de clases que representa el código generado por el proceso de refactorización.

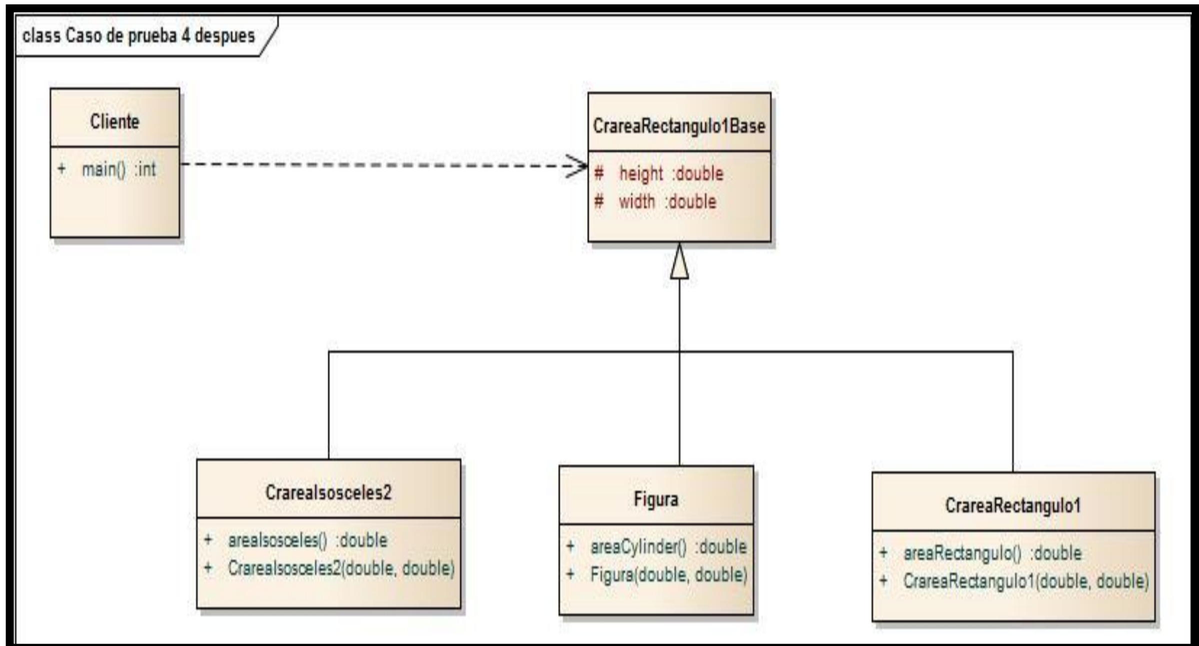


Figura 4.14 Diagrama de clases después de pasar el proceso de refactorización.

4.5. Caso de prueba 5: cuatro clases, una clase abstracta y sin método *main()*.

Tabla 4.5 Caso de prueba 5

<table>
<tr><td>Objetivo:</td><td colspan="11">Observar el comportamiento del proceso de refactorización al encontrarse con una clase abstracta, no encontrar el método main() y un total de cinco clases.</td></tr>
<tr><td>Características a probar:</td><td colspan="11">• Cálculo de las métricas cohesión y coherencia
• Presentación de los resultados de las tablas.
• Proceso de refactorización.</td></tr>
<tr><td>Entrada:</td><td colspan="11">Dos archivos con cinco clases y sin método main().</td></tr>
<tr><td>Resultados esperados:</td><td colspan="11"></td></tr>
<tr><td rowspan="4"></td><td></td><td colspan="5">Antes de refactorizar</td><td colspan="5">Después de refactorizar</td></tr>
<tr><td>No. de clase</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td>Grado de cohesión</td><td>0</td><td>1</td><td>2</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Grado de coherencia</td><td>0</td><td>1</td><td>0.5</td><td>1</td><td>0.5</td><td>0</td><td>1</td><td>1</td><td>0.5</td><td>1</td></tr>
<tr><td>Procedimiento de prueba:</td><td colspan="11">5. Seleccionar archivo a refactorizar
6. Pulsar en la opción Método para lograr alta cohesión y alta coherencia.
7. Pulsar botón Analizar código.
8. Pulsar botón Refactorizar.</td></tr>
</table>

<table>
<tr><td colspan="7">Resultados después de refactorizar</td></tr>
<tr><td>Resultado:</td><td>No. de clase</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td rowspan="2"></td><td>Grado de cohesión:</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Grado de coherencia:</td><td>0</td><td>1</td><td>1</td><td>0.5</td><td>1</td></tr>
<tr><td>Estado del caso de prueba:</td><td colspan="6">Exitoso</td></tr>
<tr><td>Observaciones:</td><td colspan="6">Se crea una nueva clase y un total de 11 archivos.</td></tr>
</table>

El caso de prueba 5 tiene cuatro clases con métodos implementados, tiene una clase abstracta y en esta prueba se simula que el usuario no selecciona el archivo donde se encuentra el método *main()*. La clase *Operaciones* es la clase abstracta y tiene solamente un método virtualmente puro que se llama *calcular()*.

La clase *Suma* tiene de padre a la clase *Operaciones,* tiene cuatro tributos de la clase y la implementación del método *calcular()*. La clase *Multiplica* igual tiene de padre a la clase *Operaciones,* tiene seis atributos de la clase y dos métodos: la implementación del método *calcular* y el método *calcularAreaTrapecio().* La clase Resta tiene de padre a la clase *Operaciones,* tiene dos atributos de la clase y la implementación del método *calcular().* La clase *PruebaConstructor* tiene cinco atributos de la clase y una de ellas es de tipo *Suma* y tiene dos métodos: *resultadCalculo* y *soloPrueba().*

La Figura 4.15 presenta el diagrama de clases del caso de prueba 5, en el anexo B (§Figura B.36 y Figura B.37) se encuentra el código completo.

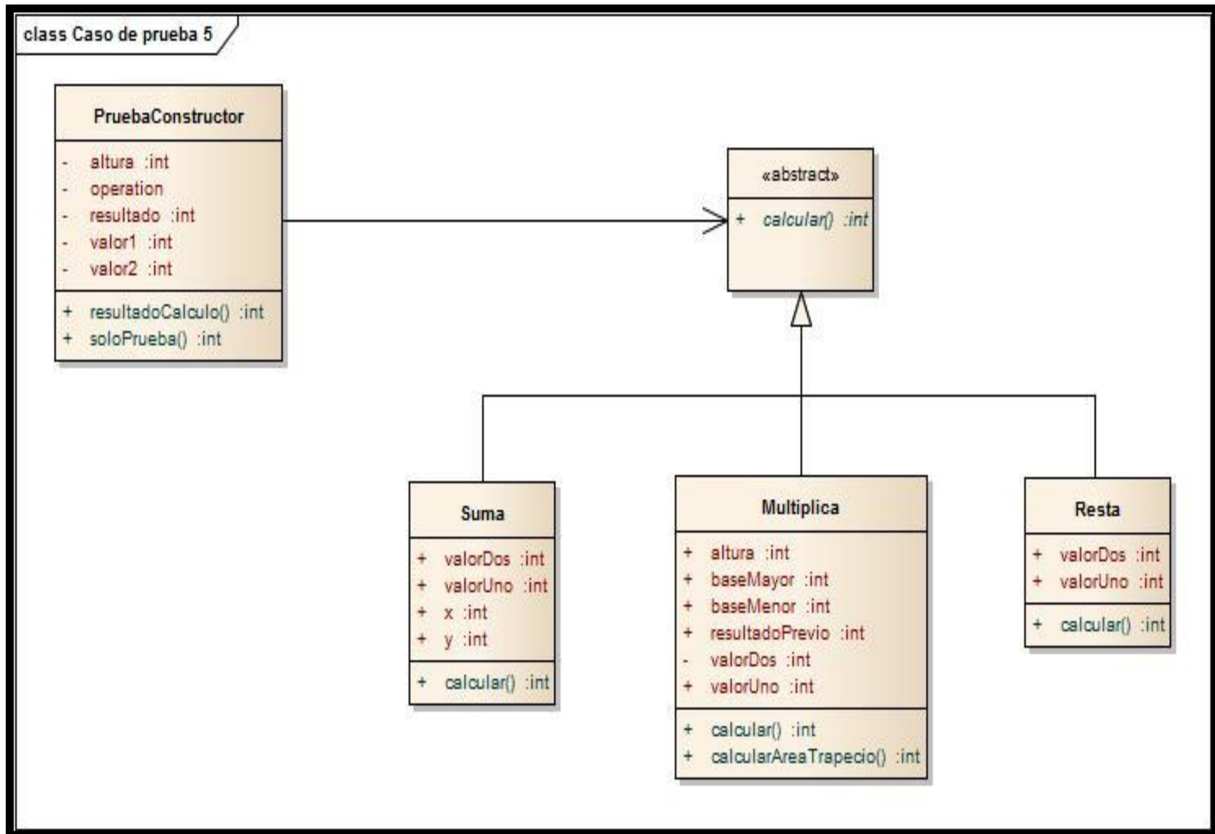
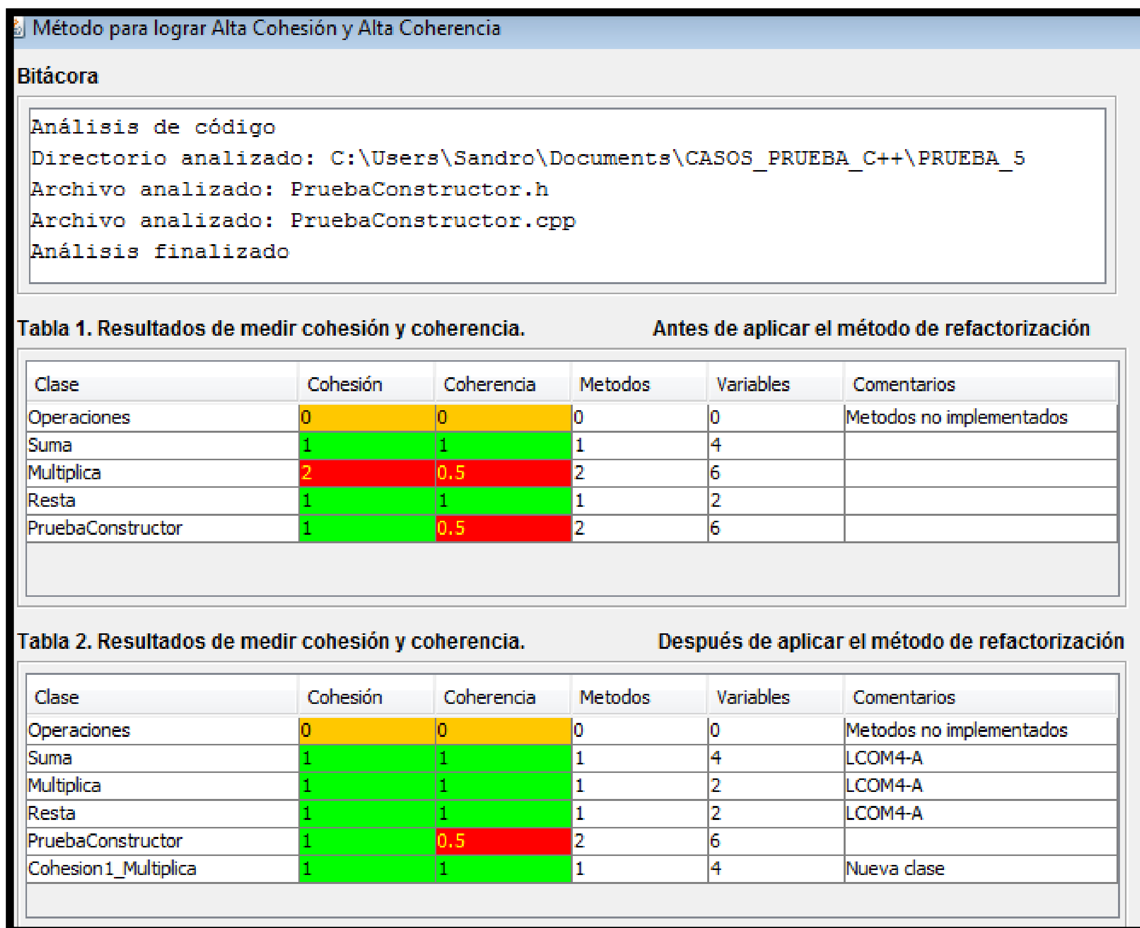


Figura 4.15 Diagrama de clases del caso de prueba 5.

La Figura 4.16 presenta los resultados después de aplicar el proceso de refactorización, se observa que la clase nueva fue creada por el MRACS, la clase *Operaciones* continua con el mismo grado de coherencia, esto porque el MRACR en el paso 6 (§2.3.2.6) verificó si era posible crear la herencia y no se cumplieron las validaciones necesarias, por lo tanto, la clase continua con los dos métodos y su grado de coherencia no fue mejorado.



Método para lograr Alta Cohesión y Alta Coherencia

Bitácora

```
Análisis de código
Directorio analizado: C:\Users\Sandro\Documents\CASOS_PRUEBA_C++\PRUEBA_5
Archivo analizado: PruebaConstructor.h
Archivo analizado: PruebaConstructor.cpp
Análisis finalizado
```

Tabla 1. Resultados de medir cohesión y coherencia. Antes de aplicar el método de refactorización

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Operaciones	0	0	0	0	Metodos no implementados
Suma	1	1	1	4	
Multiplica	2	0.5	2	6	
Resta	1	1	1	2	
PruebaConstructor	1	0.5	2	6	

Tabla 2. Resultados de medir cohesión y coherencia. Después de aplicar el método de refactorización

Clase	Cohesión	Coherencia	Metodos	Variables	Comentarios
Operaciones	0	0	0	0	Metodos no implementados
Suma	1	1	1	4	LCOM4-A
Multiplica	1	1	1	2	LCOM4-A
Resta	1	1	1	2	LCOM4-A
PruebaConstructor	1	0.5	2	6	
Cohesion1_Multiplica	1	1	1	4	Nueva clase

Figura 4.16 Resultados después de aplicar el proceso de refactorización.

Los archivos generados al terminar el proceso de refactorización se pueden observar en el anexo B (§Figura B.50). La mayoría de estos archivos son generados por el MROCL (§2.3.3).

La Figura 4.17 presenta el diagrama de clases del código refactorizado después de terminar el proceso de refactorización, se puede ver la nueva clase generada *Cohesion35_Multiplica,* esta clase fue generada a partir de dividir las responsabilidades de la clase *Multiplica.*

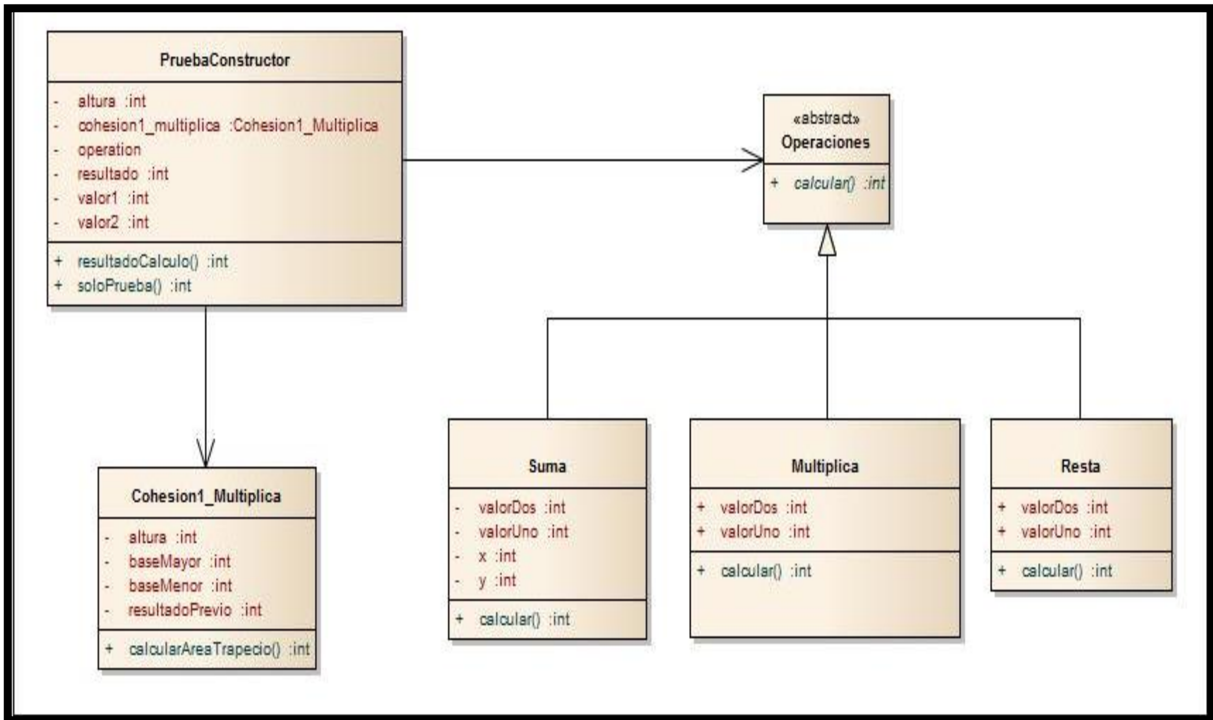


Figura 4.17 Diagrama de clases representando el código después del proceso de refactorización.

4.6. Resumen de las pruebas.

A continuación se presenta el resumen de los casos de prueba en dos tablas, los datos de las tablas presentan el antes y después de aplicar el proceso de refactorización, en las columnas cohesión y coherencia se utilizan los colores: rojo; valor no deseado, verde; valor deseado, amarillo: la clase no implementa métodos.

Una vez que se ejecutaron todos los casos de prueba se presentan los resultados de los casos de prueba antes de aplicar el proceso de refactorización ver Tabla 4.6.

Por cada caso de prueba se incluye: el total de archivos y total de clases. Por cada clase del caso de prueba se muestra: nombre de la clase, métodos, variables, grado de cohesión y grado de coherencia, esta información es la misma para ambas tablas.

Tabla 4.6 Resumen de los casos de prueba antes de aplicar el proceso de refactorización.

Caso de prueba	Total Archivos	Total Clases	Nombre de la clase	Métodos	Variables	Cohesión	Coherencia
1	1	1					
			Programming	2	1	**1**	**0.5**
2	1	1					
			Temp	2	2	**2**	**0.5**
3	1	1					
			Operaciones	5	0	**2**	**0.4**
4	3	1					
			Figura	3	2	**1**	**0.33**
5	2	5					
			Operaciones	0	0	**0**	**0**
			Suma	1	4	**1**	**1**
			Multiplica	2	6	**2**	**0.5**
			Resta	1	2	**1**	**1**
			PruebaConstructor	2	6	**1**	**0.5**

Se presentan todos los resultados de los casos de prueba después de aplicar el proceso de refactorización ver Tabla 4.7.

Tabla 4.7 Resumen de los casos de prueba después de aplicar el proceso de refactorización.

Caso de prueba	Total Archivos	Total Clases	Nombre de la clase	Métodos	Variables	Cohesión	Coherencia
1	3	1					
			Programming	2	1	**1**	**0.5**
2	5	2					
			Temp	1	1	**1**	**1**
			Cohesion2_Temp	1	1	**1**	**1**
3	5	2					
			Operaciones	3	0	**1**	**1**
			CroperacionCompuesta1	2	0	**1**	**1**
4	8	4					
			CrareaRectangulo1Base	0	2	**0**	**0**
			CrareaRectangulo1	1	0	**1**	**1**
			CrareaRectangulo2	1	0	**1**	**1**
			Figura	1	0	**1**	**1**
5	11	6					
			Operaciones	0	0	**0**	**0**
			Suma	1	4	**1**	**1**
			Multiplica	1	2	**1**	**1**
			Resta	1	2	**1**	**1**
			PruebaConstructor	2	6	**1**	**0.5**
			Cohesion35_Multiiplica	1	4	**1**	**1**

En la columna Total Archivos de la Tabla 4.6 y la Tabla 4.7, se observa como el número de archivos se incrementa después de pasar por el proceso de refactorización, esto sucede en todos los casos de prueba.

En la columna Total Clases se refleja que después de pasar por el proceso de refactorización y mejorar el grado de cohesión o coherencia se incrementa el número de clases, esto sucede en todos los casos de prueba a excepción del caso de prueba 1, donde el grado de coherencia no fue mejorado.

Se analiza el comportamiento de la columna Métodos: después de aplicar el proceso de refactorización y mejorar el grado de cohesión o coherencia se puede decir que existe una reubicación de métodos en las clases. Es decir, se busca que en las clases sus elementos estén relacionados (alta cohesión) y que exista una única secuencia interactiva de métodos (alta coherencia), esto se observa en el caso de prueba 2, 3 y 5, en este último caso de prueba específicamente nos referimos a la clase *Multiplica*.

En la columna cohesión siempre fue mejorado su grado de cohesión después de aplicar el proceso de refactorización, esto lo observamos en el caso de prueba 2, 3 y 5, este último específicamente en la clase *Multiplica*. En el caso de la columna coherencia su grado fue mejorado en los casos de prueba 2, 3, 4 y 5, este último específicamente en la clase *Multiplica.* El detalle a observar es que en el caso de prueba 1 y 5 (*PruebaConstructor*) el grado de coherencia no fue posible mejorarlo.

CONCLUSIONES Y TRABAJO FUTURO Capítulo 5

En este capítulo se mencionan las conclusiones a las que se ha llegado después de terminar este trabajo de investigación, las aportaciones y trabajo futuro que pueden complementar o extender este trabajo.

5.1. Aportaciones de la tesis.

En este trabajo de investigación se desarrolló un proceso de refactorización de factores de calidad de arquitectura de software orientado a objetos que está compuesto por tres métodos de refactorización:

- MRACS; Método que logra clases con todos los elementos (métodos y atributos) relacionados, es decir, alta cohesión (§2.3.1).
- MRACR; Método que logra clases con una única responsabilidad (alta coherencia) (§2.3.2).
- MROCL; Método que ordena las clases, de manera que cada clase se encuentre en una archivo (.h y .cpp) (§2.3.3).

Un estudio y análisis de las métricas de cohesión:

- Con la ayuda de este estudio se seleccionó la métrica LCOM4.

Se proponen dos métricas:

- LCOM4-A; mide la cohesión de clases con un solo método (§2.2.3).
- CR; mide la coherencia a nivel clase basado en el principio de una única responsabilidad (§2.1.1).

Se desarrolló una herramienta:

- Que implementa los tres métodos de refactorización, las dos métricas propuestas (LCOM4-A y CR) y una métrica que mide la cohesión (LCOM4). Todas las métricas aplicadas a nivel clase (§3.1).

5.2. Conclusiones.

En este trabajo de investigación se concluye que:

Se pueden lograr clases con alta cohesión y alta coherencia de manera automática aplicando MRACS y MRACR en este orden, de lo contrario se puede dar el caso de crear herencias incorrectamente debido a que los métodos comparten atributos.

Usando las métricas de este trabajo de investigación se deduce que una clase puede tener alta cohesión pero no alta coherencia, sin embargo, cuando una clase contiene más de un método y tenga alta coherencia se concluye que la clase tiene alta cohesión.

Cuando se mejora el nivel de cohesión y/o de coherencia se incrementa el número de clases en colaboración por lo que se incrementa los canales de comunicación y las dependencias.

Cuando se mejora el nivel de cohesión y/o de coherencia se mejora la modularidad, logrando con esto que el nivel de reuso sea mayor debido a que los módulos de programa son completos y suficientes para alcanzar un objetivo o meta de valor para el usuario, sin embargo, para alcanzar una meta de mayor alcance se incrementa las dependencias.

Si una clase tiene alta cohesión y/o alta coherencia es más fácil de entender, realizarle pruebas, su complejidad disminuye y su mantenimiento se vuelve menos complicado, debido a que los elementos de una clase están relacionados y cumplen con una única responsabilidad, quedan encapsulados y con pocas dependencias.

Basados en que un sistema no es más que un conjunto de partes (clases) desarrollados por separado y que lo importante es cómo interactúan entre sí, todos los elementos que contiene una clase deben estar relacionados (alta cohesión) y solo debe tener una secuencia interactiva de métodos, es decir, una única responsabilidad (alta coherencia) y la interacción entre las clases debe ser en el menor grado posible (bajo acoplamiento) y con esto lograr que una clase sea completa y suficiente (auto-suficiencia).

Después de aplicar el método de refactorización se pueden lograr clases que tengan alta cohesión y alta coherencia. Sin embargo, si el código de entrada tiende defectos de funcionalidad, entonces estos defectos serán trasladados a la arquitectura de salida.

5.3. Trabajo futuro.

Después de concluir esta tesis surge uno o varios proyectos nuevos. En un proyecto la idea es aplicar cohesión y coherencia a nivel micro-arquitecturas de clases donde la meta de valor es cumplir con un requerimiento específico. En la coherencia se deben revisar las llamadas entre métodos pero de diferentes clases, eso sería coherencia externa. En la coherencia externa se formarían secuencias interactivas de métodos que involucren llamadas internas (dentro de la clase) y llamas externas (fuera de la clase), buscando cumplir con el principio de una única responsabilidad pero sin formar clases de grano grueso. Esto nos llevaría a definir nuevas métricas a nivel micro-arquitectura.

Otro proyecto de investigación es aplicar cohesión y coherencia a nivel métodos para que cada método tenga una única operación: seria por cada método realizar conjuntos de variables y sentencias de código que estén relacionados, si existe más de un conjunto en un método, significa que ese método tiene baja cohesión (a nivel método) y se debe dividir. En el caso de la coherencia sería formar las secuencias de estatutos.

Un trabajo de investigación se puede encargar de unir dos proyectos. Un proyecto sería el de cohesión y coherencia a nivel métodos y el otro proyecto sería el desarrollado en esta investigación (cohesión y coherencia a nivel clase) realizando pruebas para determinar el orden correcto de cada proyecto. Se considera que primero se mejoré la cohesión y coherencia a nivel métodos (separar métodos) y posteriormente mejorar la cohesión y coherencia a nivel clase (para unir los métodos con una única responsabilidad) en una unidad de programa llamada módulo, paquete o componente.

En lo que respecta a este trabajo de investigación, se deben realizar más pruebas a la herramienta desarrollada para observar el comportamiento del proceso de refactorización y agregar posibles casos que no fueron considerados en las pruebas a esta herramienta.

Una mejora a la herramienta desarrollada es agregar el comportamiento que debe adoptar el MRACR cuando se encuentra con métodos que comparten una variable o más pero no existe invocación entre los métodos. Una posible solución sería pasar el valor que necesita el otro método por medio de parámetros.

En la herramienta desarrollada se deben mejorar los nombres asignados a las nuevas clases. Una posible solución podría ser permitirle al usuario que agregue los nombres deseados a las nuevas clases que se están creando.

Otra mejora a la herramienta desarrollada es crear clases con métodos virtualmente puros, es decir, clases abstractas. Esto permitiría separar las clases concretas del acceso del cliente, favoreciendo la modularidad y el nivel de reuso.

De la herramienta desarrollada mejorar el movimiento de los archivos *include*, debido a que en este momento se copian todos los existentes a las nuevas clases generadas, esto puede ocasionar errores a las nuevas clases creadas. Otro punto es agregar los include por default que debe llevar cada clase.

Completar la gramática de C++ generada con JavaCC para que reconozca la totalidad del lenguaje C++. De este modo, en posteriores proyectos no se consuma tiempo en el analizador y así cada investigación sólo tendría que realizar las actualizaciones correspondientes a los nuevos elementos agregados al lenguaje C++ y enfocarse solo en el método de refactorización.

Para el proyecto SR2, cuando se complete la gramática de C++, se debe trabajar en la construcción de una base de datos donde de manera ordenada se almacenen todos los datos del código analizado. Esto permitiría que los métodos de refactorización existentes en el SR2 se comuniquen sólo a una base de datos y con algunas tablas exclusivas por cada método de refactorización. Facilitando la creación de nuevos métodos de refactorización al tener una base de datos con la información necesaria y talvez solo agregar alguna(s) tabla(s) para el nuevo método de refactorización, entonces el nuevo proyecto de investigación podría enfocar la mayor parte de su tiempo solo al nuevo método refactorización.

REFERENCIAS

[Lehmanand, 1985] Program Evolution: processes of software change, M. M. Lehmanand L. A. Belady.. Academic Press Professional, Inc., (1985).

[Fowler, 1999] Refactoring: Improving the Design of Existing Code, Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Addison-Wesley, ISBN: 0201485672 páginas: 431, (2002).

[Demeyer, 2002] Object Oriented Reengineering Patterns, S. Demeyer, P. DucasseandO. Nierstrasz. Morgan Kaufmann Publishers Inc., San Francisco, CA, (2002).

[Misic, 2001] Misic, V.B. Cohesion is Structural, Coherence is Functional: Different Views, Different Measures. Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International. Page(s): 135 – 144. IEEE.

[Bustamante, 2003] Reestructuración de código legado a partir del comportamiento para la generación de componentes reutilizables. Cesar Bustamante Laos. (2003). Centro Nacional de Investigación y Desarrollo Tecnológico. PP. 1-121.

[Santaolaya, 2003] Restructuring Conditional Code Structures Using Object Oriented Design Patterns. Rene Santaolaya S., Olivia G. Fragoso D., Joaquín Pérez O. Lorenzo Zambrano S. ICCSA (2003), LNCS 2667, pp. 704-713.

[Cárdenas, 2004] Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator. Leonor Adriana Cárdenas Robledo. (2004) Centro Nacional de Investigación y Desarrollo Tecnológico. P. 1 – 132.

[Valdés, 2004] Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la separación de interfaces. Manuel Alejandro Valdés Marrero. (2004) Centro Nacional de Investigación y Desarrollo Tecnológico. P. 1 – 173.

[Santos, 2005] Adaptación de interfaces de Marcos de Aplicaciones Orientados a Objetos, usando el Patrón de Diseño Adapter. Luis Esteban Santos Castillo. (2005) Centro Nacional de Investigación y Desarrollo Tecnológico. P. 1 – 97.

[Tsantalis, 2009] Identification of refactoring opportunities introducing polymorphism. Nikolaos Tsantalis, Alexander Chatzigeorgiou. (2009). P14. The Journal of System and Software.

[Bavota, 2010] A Two-Step Technique For Extract Class Refactoring. Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto. (2010) Proceedings of the IEEE/ACM international conference on Automated software engineering, Pages 151-154.

[Alkhalid, 2010] Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm. Abdulaziz Alkhalid, Mohammad Alshayeb, Sabri Mahmoud. Advances in Engineering Software, ISSN 0965-9978, (2010), Volumen 41, Número 10, pp. 1160 – 1178.

[Perepletchikov, 2010] The Impact of Service Cohesion on the Analyzability of Service-Oriented Software. Mikhail Perepletchikov, Member, IEEE, Caspar Ryan, Zahir Tari. IEEE Transactions on Services Computing, VOL. 3, No. 2, APRIL-JUNE (2010).

[Abdeen, H, 2011] Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software. Abdeen, H.; Ducasse, S.; Sahraoui, H. (2011) Working Conference Reverse Engineering (WCRE). Pages: 394 – 398.

[Fokaefs, 2011] JDeodorant: Identification and Application of Extract Class Refactorings. Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, Alexander Chatzigeorgiou. Proceedings of the 33rd International Conference on Software Engineering. ISBN: 978-1-4503-0445-0. (2011). Pp. 1037-1039.

[Griffith, 2011] Evolution of Legacy System Comprehensibility through Automated Refactoring. Isaac Griffith, Scott Wahl, Clemente Izurieta. Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering. (2011). Pages 35-42.

[Safwat M., 2012] Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics. Ibrahim, Safwat M., Salem, Sameh A., Ismail, Manal A., Eladawy, Mohamed. (2012). Academic, Vol. 9 Issue 2, P68. International Journal of Computer Science Issues.

[Bavota, 2014] In medio stat virtus: Extract Class Refactoring Through Nash Equilibria. Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Andrian Marcus, Yann-Gaël Guéhéneuc, Giuliano Antoniol. (2014) Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 214 - 223

[Walia, 2016] Dipesh walia., Jonny., Rinku Goel., Yogesh., (16/05/2016), “Programing Simplified”, Haryana, India, Recuperado de http://www.programmingsimplified.com.

[Martin, 2002] Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

[Yadav, 2014] Review of Object-Oriented Coupling and Cohesion Metrics, Sushma Yadav, Dr. Sunil Sikka, Uttpal Shrivastava, A International Journal of Computer Science Trends and Technology (IJCST) – Volume 2 Issue 5, Sep-Oct (2014).

[Chidamber, 1992] A Metrics Suite for Object Oriented Design Shyam R. Chidamber, Chris F. Kemerer, Center for Information Systems Research, December (1992).

ANEXO A ESTUDIO DE MÉTRICAS DE COHESIÓN

En este anexo se presenta el estudio de las métricas de cohesión, se muestra los ejemplos que fueron utilizados para aplicar cada métrica y se resume en una tabla para comparar todos los resultados de las diferentes métricas de cohesión.

Introducción

Las métricas de software son usadas para revisar y evaluar varios aspectos de la complejidad de un producto de software, acoplamiento y cohesión son considerados como atributos más importantes. Para mantener alta calidad en el software los desarrolladores siempre eligen bajo acoplamiento y alta cohesión en el diseño. Uno de los principales objetivos detrás del análisis y diseño orientado a objetos es implementar un sistema de software donde las clases tienen alta cohesión y bajo acoplamiento [1].

Las clases son el concepto fundamental en el paradigma orientado a objetos, estas son la unidad básica de los programas orientados a objetos y sirven como la unidad de encapsulación que promueve la modificabilidad y la reutilización de las clases. Las entidades en un dominio de aplicación son capturadas como clases y las aplicaciones son construidas con objetos que son instanciados desde estas clases [2].

La cohesión es altamente deseada por los desarrolladores de software porque es asociada con varios rasgos deseados del software incluyendo robustez, fiabilidad, capacidad de reutilización y la comprensión, mientras que baja cohesión se asocia con rasgos indeseables tales como ser difícil de mantener, difícil de probar, difícil de volver a utilizar, e incluso difícil de entender. [4]

En esta investigación analizamos 12 métricas con el objetivo de seleccionar la que mejor cumpla con el concepto de cohesión (se refiere al grado en que los elementos de un módulo están relacionados). Realizamos una breve descripción de cada una de las métricas y utilizamos los ejemplos de clases para determinar el valor de cohesión que tienen con respecto a la métrica que se utiliza, describimos los elementos necesarios para realizar el cálculo de la métrica y obtenemos su valor de cohesión. Al final de la descripción de métricas tenemos una tabla que muestra los diferentes valores de cohesión por cada ejemplo de las clases, la tabla nos permite generar un análisis de las métricas y concluir con la métrica seleccionada.

Ejemplo de clases

Figura A.1 Ejemplo de clases

LACK OF COHESION OF METHODS (LCOM (1991)):

LCOM establece en qué medida los métodos hacen referencia a los atributos. 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 [5].

> Considere una clase C_1 con los métodos $M_1, M_2 \ldots, M_n$. Se considera $\{I_i\}$ = al conjunto de variables de instancia usados por el método M_i. Estos son tales conjuntos $\{I_1\}, \ldots, \{I_n\}$.

LCOM = El número de conjuntos disjuntos formados por la intersección de los *n* conjuntos [3], [4]. LCOM es una medida de cohesión inversa. Un valor alto de LCOM indica baja cohesión y de lo contrario un menor valor de LCOM indica alta cohesión [5].

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Pares de métodos: (m1, m2)
Conjuntos disjuntos: 0.
LCOM = 0

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3= {}
Pares de métodos: (m1, m2). (m1, m3), (m2, m3)
Conjuntos disjuntos: (m1, m3), (m2, m3)
LCOM = 2

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Pares de métodos: No hay pares de métodos
Conjuntos disjuntos: No hay conjuntos disjuntos
LCOM = No se puede calcular.

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6).
Conjuntos disjuntos: (m1, m2), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6).
LCOM = 14

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Pares de métodos: (m1, m2)
Conjuntos disjuntos: 0
LCOM = 0

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
Conjuntos disjuntos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4).
LCOM = 6

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
Conjuntos disjuntos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4).
LCOM = 6

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m4, m5).
Conjuntos disjuntos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m4, m5).
LCOM = 9

LCOM2 (1994):

Se define como la diferencia entre el número de pares de métodos que no comparten atributos en común y aquellos pares que si comparten atributos en común [6], Un valor de LCOM cero especifica una clase cohesiva [1].

Dados los métodos n $M_1,\ M_2, \dots, M_n$ contenidos en una clase C1, que también contiene un conjunto de variables de instancia $\{I_i\}$. Entonces para cualquier método M_i podemos definir el conjunto de particiones [1]:

P = {(Ii, Ij)| $(Ii \cap Ij = 0)$} y Q = {(Ii, Ij)|$Ii \cap Ij \neq 0$}
Entonces LCOM2 $= if\ |P| > |Q|\ entonces\ |P| - |Q|,\ de\ lo\ contrario\ LCOM2 = 0$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Pares de métodos: (m1, m2)
Conjuntos nulos: ***P = 0***
Conjuntos no vacíos: (m1, m2): ***Q = 1***
LCOM2 = 0

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = { }
Pares de métodos: (m1, m2), (m1, m3), (m2, m3)
Conjuntos nulos: (m1, m3), (m2, m3): ***P = 2***
Conjuntos no vacíos: (m1, m2): ***Q = 1***
LCOM2 = 1

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}.
Pares de métodos: No hay pares de métodos
Conjuntos nulos: 0
Conjuntos no vacíos: 0
LCOM2 = No se puede calcular

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = {}, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6)
Conjuntos nulos: (m1, m2), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6): ***P = 15***
Conjuntos no vacíos: (m1, m3) ***Q = 1***
LCOM2 = 14

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Pares de métodos: (m1, m2)
Conjuntos nulos: 0: ***P = 0***
Conjuntos no vacíos: (m1, m2): ***Q = 1***
LCOM2 = 0

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {}, m3 = {}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
Conjuntos nulos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4): ***P = 6***
Conjuntos no vacíos: ***Q = 0***
LCOM2 = 6

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = {}, m3 = {}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
Conjuntos nulos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4): ***P = 6***
Conjuntos no vacíos: ***Q = 0***
LCOM2 = 6

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m3, m5), (m4, m5)
Conjuntos nulos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m3, m5), (m4, m5): ***P = 10***
Conjuntos no vacíos: ***Q = 0***
LCOM2 = 10

LCOM3 (1993):

Considera un grafo no dirigido G, donde los vértices son los métodos de una clase y existe una arista entre dos vértices si los métodos correspondientes comparten al menos un atributo [4].

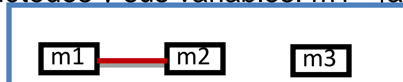
Si LCOM3 = 1 indica una clase cohesiva. Si LCOM3 >= 2 indica un problema, la clase debe dividirse en otras clases más pequeñas. Si LCOM3 = 0 no hay métodos en esa clase, también son llamadas malas clases.

LCOM3 = | Número de componentes en G |

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}

LCOM3 = 1

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = { }

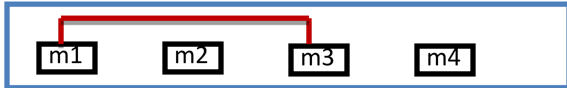


LCOM3 = 2

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}

LCOM3 = No aplica con clases que tienen un método

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = {}, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}



LCOM3 = 3

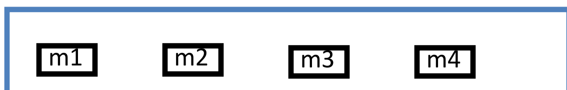
***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}

LCOM3 = 1

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {}, m3 = {}, m4 = {}

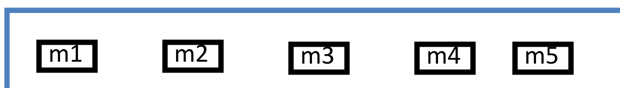
LCOM3 = 4

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {}, m4 = { }



LCOM3 = 4

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a3, a4}, m5 = {a6}

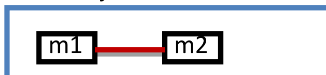


LCOM3 = 5

LCOM4 (1995):

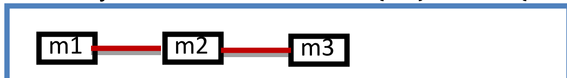
La métrica LCOM4 es parecida a LCOM3, donde al grafo G se le agregan aristas entre los vértices representados por los métodos $Mi\ y\ Mj$ si $Mi\ invoca\ a\ Mj$ o viceversa.
Si LCOM4 = 1 indica una clase cohesiva. Si LCOM4 >= 2 indica un problema, la clase debe dividirse en otras clases más pequeñas. Si LCOM4 = 0 no hay métodos en esa clase, también son llamadas malas clases [4].

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a1, a2, a3}



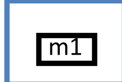
LCOM4 = 1

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {**m2**}



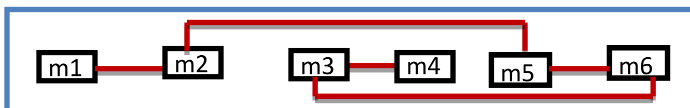
LCOM4 = 1

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}



LCOM4 = No aplica con clases que tienen un método.

***Clase D*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m5**}, m3 = {a2, **m4**}, m4 = {a3}, m5 = {**m6**}, m6 = {**m3**}.

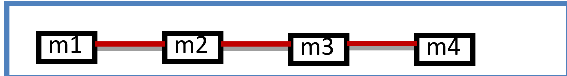


LCOM4 = 1

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a3}

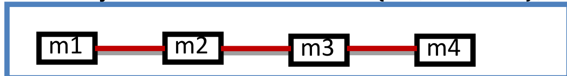
LCOM4 = 1

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}



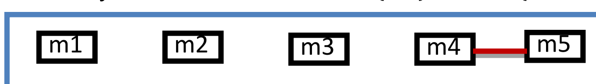
LCOM4 = 1

***Clase G*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = { }



LCOM4 = 1

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5, **m5**}, m5 = {a6}



LCOM4 = 4

Tight Class Cohesion (TCC (1995)):

Es la relación normalizada entre el número de métodos directamente conectados con otros métodos a través de una variable de instancia y el número total de posibles conexiones entre los métodos [8], [9] y [10]. TCC toma valores entre 0 y 1, métodos que son privados, constructores y destructores son ignorados. Un valor alto de TCC indica una clase cohesiva, un valor menor a 0.5 indica una clase no cohesiva, si se obtiene un valor de 1 significa alta cohesión en la clase: todos los métodos están conectados [7].

NDC = número de conexiones directas entre métodos públicos.
NP = número máximo de pares de métodos públicos = $N*(N-1)2$
Donde N = número de métodos.

$$TCC = NDC/NP$$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Pares de métodos: (m1, m2)
NDC = (m1, m2) = **1**
NP = 2*(2-1)/2 = **1**
TCC = 1/1 = 1

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {}
Pares de métodos: (m1, m2), (m1, m3), (m2, m3)
NDC = (m1, m2) = **1**
NP = 3*(3-1)/2 = **3**
TCC = 1/3 = 0.33

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Pares de métodos: No hay pares de métodos
NDC = **0**
NP = 1*(1-1)/2 = **0**
TCC = No se puede calcular

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6).
NDC = (m1, m3) = **1**
NP = 6*(6-1)/2 = **15**
TCC = 1/15 = 0.066

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Pares de métodos: (m1, m2)
NDC = (m1, m2) = **1**
NP = 2*(2-1)/2 = **1**
TCC = 1/1 = 1

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
NDC = **0**
NP = 4*(4-1)/2 = **6**
TCC = 0/6 = 0

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
NDC = **0**
NP = 4*(4-1)/2 = **6**
TCC = 0/6 = 0

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m4, m5).
NDC = **0**
NP = 4*(4-1)/2 = **6**
TCC = 0/6 = 0

Loose Class Cohesion (LCC (1995))

Es la relación normalizada entre el número de métodos de relación directa o indirecta con otros métodos a través de una variable de instancia y el número total de posibles conexiones entre los métodos. Indirectamente se da cuando un método *m2* invoca a otro método *m1* y el método *m1* usa una variable de la clase *a1*, entonces se dice que *m2* indirectamente usa la variable *a1*.

Cuanto mayor es el valor de LCC, más cohesiva es la clase. Un valor de LCC menor de 0.5 significa que la clase no es cohesiva. LCC = 0.8 es considerada bastante cohesiva, un valor de 1 es el máximo valor de una clase: todos sus métodos están conectados, una clase altamente cohesiva.
[7], [9] y [10].

NIC = número de conexiones indirectas entre métodos públicos.
NDC = número de conexiones directas entre métodos públicos.
NP = número máximo de pares de métodos públicos = $N * \frac{(N-1)}{2}$
Donde N = número de métodos públicos [6], [8] y [9].

$$LCC = NDC + NIC/NP$$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a1, a2, a3}
Pares de métodos: (m1, m2)
NIC = **0**
NDC = (m1, m2) = **1**
NP = 2*(2-1)/2 = **1**
LCC = (1+ 0)/1 = 1/1 = 1

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {**m2**}
Pares de métodos: (m1, m2), (m1, m3), (m2, m3)
NIC = (m1, m3), (m2, m3) = **2**
NDC = (m1, m2) = **1**
NP = 3*(3-1)/2 = **3**
LCC = (1 +2)/6 = 3/3 = 1

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Pares de métodos: No hay pares de métodos
NIC = **0**
NDC = **0**
NP = 1*(1-1)/2 = **0**
LCC = No se puede calcular

***Clase D*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m5**}, m3 = {a2, **m4**}, m4 = {a3}, m5 = {**m6**}, m6 = {**m3**}.
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6).
NIC = (m1, m2), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6) = **14**
NDC = (m1, m3) = **1**
NP = 6*(6-1)/2 = **15**
LCC = (14 +1)/6 = 15/15 = 1

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a3}
Pares de métodos: (m1, m2)
NIC = (m1, m2) = **1**
NDC = (m1, m2) = **1**
NP = 2*(2-1)/2 = **1**
LCC = 1/1 = 1

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
NIC = (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4) = **6**
NDC = **0**
NP = 4*(4-1)/2 = **6**
LCC = 0 + 6/6 = 1

***Clase G*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
NIC = (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4) = **0**
NDC = **0**
NP = 4*(4-1)/2 = **6**
TCC = 0 +6/6 = 1

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5, **m5**}, m5 = {a6}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m4, m5).
NIC = (m4, m5) **1**
NDC = **0**
NP = 4*(4-1)/2 = **6**
TCC = 1/6 = 0.166

LCOM5 (1996):

$$LCOM5 = \frac{NOM - \frac{\sum m \in M\ NOAcc\ (m)}{NOA}}{NOM - 1}$$

Donde M es el conjunto de métodos de la clase, NOM el número de métodos, NOA el número de atributos y $\sum NOA_{cc}\ (m)$ es el número de atributos de la clase que acceden a los métodos. LCOM5 puede regresar un valor de 2, cuando en una clase solo hay dos métodos y ningún acceso a los atributos. LCOM5 varia en el rango de 0 a 1, donde cero significa que cada método hace referencia a todos los atributos de la clase (alta cohesión), el valor de 1 significa que cada método de la clase hace referencia solo a un atributo de la clase. [7], [11].

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
NOM = 2.
NOA = 3.
$\sum \boldsymbol{NOA_{cc}}\ (\boldsymbol{m})$ = 6.
LCOM5 = (2 - (6/3)) / (2-1) = (2 – 2) / 1 **= 0**

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {**m2**}
NOM = 3.
NOA = 2.
$\sum \boldsymbol{NOA_{cc}}\ (\boldsymbol{m})$ = 3.
LCOM5 = (3 - (3/2)) / (3-1) = (3 – 1.5) / 2 **=** 1.5/2 **= 0.75**

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
NOM = 1.
NOA = 4.
$\sum NOA_{cc}(m)$ = 2.
LCOM5 = (1 - (2/4)) / (4-1) = (1 – 0.5) / 3 **=** 0.5/3 **= 0.166**

***Clase D*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m5**}, m3 = {a2, **m4**}, m4 = {a3}, m5 = {**m6**}, m6 = {**m3**}.
NOM = 6.
NOA = 3.
$\sum NOA_{cc}(m)$ = 4.
LCOM5 = (6 - (4/3)) / (6-1) = (6 – 1.33)/5 **=** 4.67/5 **= 0.934**

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a3}
NOM = 2.
NOA = 6.
$\sum NOA_{cc}(m)$ = 4.
LCOM5 = (2 - (4/6)) / (2-1) = (2 – 0.66)/1 **=** 1.34/1 **= 1.34**

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
NOM = 4.
NOA = 6.
$\sum NOA_{cc}(m)$ = 3.
LCOM5 = (4 - (3/6)) / (4-1) = (4 – 0.5)/3 **=** 3.5/3 **= 1.16**

***Clase G*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
NOM = 4.
NOA = 2.
$\sum NOA_{cc}(m)$ = 2.
LCOM5 = (4 - (2/2)) / (4-1) = (4 – 1)/3 **=** 3/3 **= 1**

Ejemplo H:
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5, **m5**}, m5 = {a6}
NOM = 5.
NOA = 6.
$\sum NOA_{cc}(m)$ = 6.
LCOM5 = (5 - (6/6)) / (5-1) = (5 – 1)/4 **=** 4/4 **= 1**

Coh (1998):

Es una variación a la métrica LCOM5 [8], Coh calcula la cohesión como la relación entre el número de atributos distintos que acceden a métodos de una clase [13].
Su fórmula es la siguiente [12], [14]:

$Coh = \frac{a}{kl}$ Donde: $l = número\ de\ atributos$

$k = número\ de\ métodos$

$a = la\ suma\ del\ número\ de\ atributos\ distintos\ que\ acceden\ a\ cada\ método\ en\ la\ calse$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Variables de la clase = {a1, a2, a3}
l = 3.
k = 2.
a = 6.
Coh = 6/(2*3) = 6/6 **= 1**

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {}
Variables de la clase = {a1, a2}
l = 2.
k = 3.
a = 3.
Coh = 3/(3*2) = 3/6 **= 0.5**

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Variables de la clase = {a1, a2, a3, a4}
l = 4.
k = 1.
a = 2.
Coh = 2/(4*1) = 2/4 **= 0.5**

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Variables de la clase = {a1, a2, a3}
l = 3.
k = 6.
a = 4.
Coh = 4/(6*3) = 4/18 **= 0.222**

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Variables de la clase = {a1, a2, a3, a4, a5, a6}
l = 6.
k = 2.
a = 4.
Coh = 4/(2*6) = 4/12 **= 0.333**

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
Variables de la clase = {a1, a2, a3, a4, a5, a6}
l = 6.
k = 4.
a = 3.
Coh = 3/(4*6) = 3/24 **= 0.125**

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
Variables de la clase = {a1, a2}
l = 2.
k = 4.
a = 2.
Coh = 2/(4*2) = 2/8 **= 0.25**

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Variables de la clase = {a1, a2, a3, a4, a5, a6}
l = 6.
k = 5.
a = 6.
Coh = 6/ (5*6) = 6/30 **= 0.2**

Class Cohesion (CC (2006))

CC se basa en la noción de similitudes entre los métodos de una clase. El grado de similitud entre dos métodos puede ser medido para encontrar el conjunto común de variables de instancia usadas por ambos métodos, relacionados por el número total (unión) de variables de instancias usadas por los dos métodos. La similitud entre dos métodos varía desde 0 a 1 donde el valor de 0 en *MS* indica la no similitud (no hay variables de instancia en común compartidas por los métodos) y *MS* valor de 1 indica alto grado de similitud (ambos métodos usan las mismas variables de instancia.) [15].

Fórmula para medir la similitud entre métodos:

$$MS = \frac{|IV|_c}{|IV|_t}$$

$|IV|_c = \{IV_i\} \cap \{IV_{i+1}\}$, en el cual $\{IV_i\}$ es el conjunto de variables utilizadas un método M_i

$|IV|_t = \{IV_i\} \cup \{IV_{i+1}\}$, en el cual $\{IV_i\}$ es el conjunto de variables utilizadas un método M_i

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t}\, i$$

n = el número de métodos de la clase.

***Clase A*:**

Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}

IV_C = *m1* ∩ *m2* = {a1, a2, a3}.

IV_t = *m1* ∪ *m2* = {a1, a2, a3}.

MS = *3/3 = 1*

n = 2.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t}\, i = \frac{2(2-2)!}{2!} * \sum 1 = \frac{2(0)!}{2!} * 1 = 1$$

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {}
IV_C = *m1 ∩ m2* = {a1}, *m1 ∩ m3* = {}, *m2 ∩ m3* = {}.
IV_t = *m1 ∪ m2* = {a1, a2}, *m1 ∪ m3* = {a1}, *m2 ∪ m3* = {a1, a2}.
MS = *1/2 + 0/1 + 0/2 = 0.5*
n = 2.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t}\, i = \frac{2(2-2)!}{2!} * \sum 0.5 = \frac{2(0)!}{2!} * 0.5 = 0.5$$

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
IV_C = *No se puede calcular*.
IV_t = *No se puede calcular*.
MS = *No se puede calcular*
n = 1.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t}\, i = No\ se\ puede\ calcular$$

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
IV_C = *m1 ∩ m2* = {}, *m1 ∩ m3* = {a2}, *m1 ∩ m4* = {}, *m1 ∩ m5* = {}, *m1 ∩ m6* = {}, *m2 ∩ m3* = { }, *m2 ∩ m4* = {}, *m2 ∩ m5* = {}, *m2 ∩ m6* = {}, *m3 ∩ m4* = {}, *m3 ∩ m5* = {}, *m3 ∩ m6* = {}, *m4 ∩ m5* = {}, *m4 ∩ m6* = {}, *m5 ∩ m6* = {}.
IV_t = *m1 ∪ m2* = {a1, a2}, *m1 ∪ m3* = {a1, a2}, *m1 ∪ m4* = {a1, a2, a3}, *m1 ∪ m5* = {a1, a2}, *m1 ∪ m6* = {a1, a2}, *m2 ∪ m3* = {a2}, *m2 ∪ m4* = {a3}, *m2 ∪ m5* = {}, *m2 ∪ m6* = {}, *m3 ∪ m4* = {a2, a3}, *m3 ∪ m5* = {a2}, *m3 ∪ m6* = {a2}, *m4 ∪ m5* = {a3}, *m4 ∪ m6* = {a3}, *m5 ∪ m6* = {}.
MS = *0/2 + 1/2 + 0/3 + 0/2+ 0/2 + 0/2 + 0/1 + 0/1 + 0/0 + 0/0 + 0/2 + 0/1+ 0/1 + 0/1 + 0/1 + 0/0 = 0.5*
n = 6.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t}\, i = \frac{2(6-2)!}{6!} * \sum 0.5 = \frac{2(4)!}{6!} * 0.5 = 0.03$$

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
IV_C = *m1* ∩ *m2* = {a3}.
IV_t = *m1* ∪ *m2* = {a1, a2, a3}.
MS = *1/3 = 0.33*
n = 2.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t} i = \frac{2(2-2)!}{2!} * \sum 0.33 = \frac{2(0)!}{2!} * 0.33 = 0.33$$

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
IV_C = *m1* ∩ *m2* = {}, *m1* ∩ *m3* = {}, *m1* ∩ *m4* = {}, *m2* ∩ *m3* = {}, *m2* ∩ *m4* = {}, *m3* ∩ *m4* = {}.
IV_t = *m1* ∪ *m2* = {a1, a2, a3}, *m1* ∪ *m3* = {a1, a2, a3}, *m1* ∪ *m4* = {a1, a2, a3}, *m2* ∪ *m3* = {}, *m2* ∪ *m4* = {}, *m3* ∪ *m4* = {}.
MS = *0/3 + 0/3 + 0/3 + 0/0 +0/0 + 0/0 = 0*
n = 4.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t} i = \frac{2(4-2)!}{4!} * \sum 0 = \frac{2(2)!}{2!} * 0 = 0$$

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
IV_C = *m1* ∩ *m2* = {}, *m1* ∩ *m3* = {}, *m1* ∩ *m4* = {}, *m2* ∩ *m3* = {}, *m2* ∩ *m4* = {}, *m3* ∩ *m4* = {}.
IV_t = *m1* ∪ *m2* = {a1, a2}, *m1* ∪ *m3* = {a1, a2}, *m1* ∪ *m4* = {a1, a2}, *m2* ∪ *m3* = {}, *m2* ∪ *m4* = {}, *m3* ∪ *m4* = {}.
MS = *0/3 + 0/3 + 0/3 + 0/0 +0/0 + 0/0 = 0*
n = 4.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t} i = \frac{2(4-2)!}{4!} * \sum 0.33 = \frac{2(2)!}{2!} * 0 = 0$$

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
IV_C = *m1* ∩ *m2* = {}, *m1* ∩ *m3* = {}, *m1* ∩ *m4* = {}, *m1* ∩ *m5* = {}, *m2* ∩ *m3* = {}, *m2* ∩ *m4* = {}, *m2* ∩ *m5* = {}, *m3* ∩ *m4* = {}, *m3* ∩ *m5* = {}, *m4* ∩ *m5* = {}.
IV_t = *m1* ∪ *m2* = {a1, a2, a3}, *m1* ∪ *m3* = {a1, a4}, *m1* ∪ *m4* = {a1, a4}, *m1* ∪ *m5* = {a1, a6}, *m2* ∪ *m3* = {a2, a3, a4}, *m2* ∪ *m4* = {a2, a3, a5}, *m2* ∪ *m5* = {a2, a3, a6}, *m3* ∪ *m4* = {a4, a5}, *m3* ∪ *m5* = {a4, a6}, *m4* ∪ *m5* = {a5, a6}.
MS = *0/3 + 0/2 + 0/2 + 0/2 +0/2 + 0/3 + 0/3+ 0/3 + 0/2 + 0/2 + 0/2= 0*
n = 5.

$$CC = \frac{2(n-2)!}{n!} * \sum_{i=1}^{\frac{2(n-2)!}{n!}} \frac{|IV|_c}{|IV|_t} i = \frac{2(5-2)!}{5!} * \sum 0.33 = \frac{2(3)!}{5!} * 0 = 0$$

Transitive LCOM (TLCOM (2010))

Transitive LCOM (TLCOM) es una extensión de LCOM y se define como el número de pares de métodos que directamente o transitivamente comparten un atributo común. Un método transitivo hace referencia a un atributo cuando el método directamente o indirectamente llama a otro método que directamente hace referencia al atributo. Un par de métodos transitivos comparten un atributo común cuando el atributo común es referenciado transitivamente por ambos métodos o la referencia transitiva se realiza por uno de los métodos y directamente por el otro método. Si TLCOM es igual a cero la clase tiene alta cohesión, de lo contrario un valor alto de TLCOM significa que la clase puede ser dividida porque tiene baja cohesión. TLCOM no considera las clases que tiene menos de dos métodos [16]. **P**ares de **M**étodos que **N**o **C**omparten **A**tributos en **C**omún **T**ransitivamente o **D**irectamente (PMNCACTD)

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Pares de métodos: (m1, m2)
PMNCACTD: 0
TLCOM = 0

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {**m2**}
Pares de métodos: (m1, m2). (m1, m3), (m2, m3)
PMNCACTD: 0
TLCOM = 0

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Pares de métodos: No hay pares de métodos
PMNCACTD: 0
TLCOM = No se puede calcular, TLCOM no considera la clases con un solo método.

***Clase D*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m5**}, m3 = {a2, **m4**}, m4 = {a3}, m5 = {**m6**}, m6 = {**m3**}.
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m1, m6), (m2, m3), (m2, m4), (m2, m5), (m2, m6), (m3, m4), (m3, m5), (m3, m6), (m4, m5), (m4, m6), (m5, m6).
PMNCACTD: 0
TLCOM = 0

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {a3}
Pares de métodos: (m1, m2)
PMNCACTD: 0
TLCOM = 0

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
PMNCACTD: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4).
TLCOM = 6

***Clase G*:**
Métodos y sus variables: m1= {a1, a2, **m2**}, m2 = {**m3**}, m3 = {**m4**}, m4 = {}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4)
PMNCACTD: (m1, m2), (m1, m3), (m1, m4), (m2, m3), (m2, m4), (m3, m4).
TLCOM = 6

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5, **m5**}, m5 = {a6}
Pares de métodos: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4), (m4, m5).
PMNCACTD: (m1, m2), (m1, m3), (m1, m4), (m1, m5), (m2, m3), (m2, m4), (m2, m5), (m3, m4).
TLCOM = 8

Count Cohesion (CC (2014))

La métrica CC se encuentra normalizada en un rango de 0 a 1. Considera que una clase *C* puede consistir de un conjunto de variables globales $V = \{V_1, V_2, V_3\ldots V_n\}$ y de un conjunto de métodos $M = \{m_1, m_2, m_3\ldots m_n\}$. Para evaluar CC, primero calculamos el valor de cohesión de una variable global *i* de una clase (CV_i). El valor de CV_i es definido cuando *V* o *M* no son conjuntos vacíos, de estar vacíos el valor de CV_i es cero [13]:

$$CV_I = \frac{No.\,de\ funciones\ que\ comparten\ la\ variable\ i\ de\ una\ clase}{No.\,total\ de\ funciones\ de\ la\ clase}$$

Para CC de una clase de *n* variables globales para una clase *C* esta se calcula como:

$$CC = \frac{\sum_1^n CV_i}{n}$$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Variables = {a1, a2, a3}
$\mathbf{CV_i} = 2/2, 2/2, 2/2.$
$\mathbf{n} = 3.$

$$CC = \frac{(\frac{2}{2}+\frac{2}{2}+\frac{2}{2})}{3} = \frac{3}{3} = 1$$

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {}
Variables = {a1, a2}
$\mathbf{CV_i} = 2/3, 1/3.$
$\mathbf{n} = 2.$

$$CC = \frac{(\frac{2}{3}+\frac{1}{3})}{2} = \frac{1}{2} = 0.5$$

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Variables = {a1, a2, a3, a4}
$\mathbf{CV_i} = 1/1, 1/1, 0/1, 0/1.$
$\mathbf{n} = 4.$

$$CC = \frac{(\frac{1}{1}+\frac{1}{1}+\frac{0}{1}+\frac{0}{1})}{4} = \frac{2}{4} = 0.5$$

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Variables = {a1, a2, a3}
$\boldsymbol{CV_i}$ = *1/6, 2/6, 1/6*.
n = 3.

$$CC = \frac{(\frac{1}{6}+\frac{2}{6}+\frac{1}{6})}{3} = \frac{4}{18} = 0.22$$

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Variables = {a1, a2, a3, a4, a5, a6}
$\boldsymbol{CV_i}$ = *1/2, 1/2, 2/2, 0/2, 0/2, 0/2*.
n = 6.

$$CC = \frac{(\frac{1}{2}+\frac{1}{2}+\frac{2}{2}+\frac{0}{2}+\frac{0}{2}+\frac{0}{2})}{6} = \frac{2}{6} = 0.33$$

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
Variables = {a1, a2, a3, a4, a5, a6}
$\boldsymbol{CV_i}$ = *1/4, 1/4, 1/4, 0/4, 0/4, 0/4*.
n = 6.

$$CC = \frac{(\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{0}{4}+\frac{0}{4}+\frac{0}{4})}{6} = \frac{2}{6} = 0.125$$

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
Variables = {a1, a2}
$\boldsymbol{CV_i}$ = *1/4, 1/4*.
n = 2.

$$CC = \frac{(\frac{1}{4}+\frac{1}{4})}{2} = \frac{1}{4} = 0.25$$

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Variables = {a1, a2, a3, a4, a5, a6}
$\boldsymbol{CV_i}$ = *1/5, 1/5, 1/5, 1/5, 1/5, 1/5*.
n = 6.

$$CC = \frac{\left(\frac{1}{5}+\frac{1}{5}+\frac{1}{5}+\frac{1}{5}+\frac{1}{5}+\frac{1}{5}\right)}{6} = \frac{2}{6} = 0.2$$

MALCOM (2014)

Esta es una nueva métrica que cubre las dificultades de LCOM. Para una clase *C* con un número de métodos, *M1, M2,… Mn,* Consideramos *{Ii}* el conjunto de variables de instancia utilizadas por el método *Mi*. La métrica MALCOM es entonces determinada por contar el número de conjuntos los cuales no son diferentes de cero por la unión de los *n* conjuntos.

Si (Number of Variables present in Two Methods (**TNAV**) - Number of Variables present after union (**TNUV**) <= 0)

Incrementar *R*

De lo contrario

Incrementar *Q*

Entonces realizar ***Q – R***

Si el resultado es positivo este representa alta cohesión, cero representa nivel medio de cohesión y un valor negativo representa baja cohesión en la clase [17].

$$MALCOM = Q - R$$

***Clase A*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a1, a2, a3}
Variables = {a1, a2, a3}
TNUV = *m1* ∪ *m2* = {3}.
TNAV = *m1* + *m2* = {6}.
Q = 1
P = 0
$MALCOM = Q - R = 1 - 0 = \mathbf{1}$

***Clase B*:**
Métodos y sus variables: m1= {a1}, m2 = {a1, a2}, m3 = {}
Variables = {a1, a2}
TNUV = *m1* ∪ *m2* = {2}, *m1* ∪ *m3* = {1}, *m2* ∪ *m3* = {2}.
TNAV = *m1*+ *m2* = {3}, *m1*+ *m3* = {1}, *m2* + *m3* = {2}.
Q = 1
P = 2
$MALCOM = Q - R = 1 - 2 = -\mathbf{1}$

***Clase C*:**
Métodos y sus variables: m1= {a1, a2}
Variables = {a1, a2, a3, a4}
TNUV = *0*.
TNAV = *0*.
Q = 0
P = 0
$MALCOM = Q - R = No\ se\ puede\ calcular$

***Clase D*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = {a2}, m4 = {a3}, m5 = {}, m6 = {}
Variables = {a1, a2, a3}
TNUV = *m1* ∪ *m2* = {2}, *m1* ∪ *m3* = {2}, *m1* ∪ *m4* = {3}, *m1* ∪ *m5* = {2}, *m1* ∪ *m6* = {2}, *m2* ∪ *m3* = {1}, *m2* ∪ *m4* = {1}, *m2* ∪ *m5* = {}, *m2* ∪ *m6* = {}, *m3* ∪ *m4* = {2}, *m3* ∪ *m5* = {1}, *m3* ∪ *m6* = {1}, *m4* ∪ *m5* = {1}, *m4* ∪ *m6* = {1}, *m5* ∪ *m6* = {}.
TNAV = *m1* + *m2* = {2}, *m1* + *m3* = {3}, *m1* + *m4* = {3}, *m1* + *m5* = {2}, *m1* + *m6* = {2}, *m2* + *m3* = {1}, *m2* + *m4* = {1}, *m2* + *m5* = {}, *m2* + *m6* = {}, *m3* + *m4* = {2}, *m3* + *m5* = {1}, *m3* + *m6* = {1}, *m4* + *m5* = {1}, *m4* + *m6* = {1}, *m5* ∩ *m6* = {}.
Q = 1
P = 14
$MALCOM = Q - R = 1 - 14 = \mathbf{-13}$

***Clase E*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = {a3}
Variables = {a1, a2, a3, a4, a5, a6}
TNUV = *m1* ∪ *m2* = {3}.
TNAV = *m1* + *m2* = {4}.
Q = 1
P = 0
$MALCOM = Q - R = 1 - 0 = \mathbf{1}$

***Clase F*:**
Métodos y sus variables: m1= {a1, a2, a3}, m2 = { }, m3 = { }, m4 = { }
Variables = {a1, a2, a3, a4, a5, a6}
TNUV = *m1* ∪ *m2* = {3}, *m1* ∪ *m3* = {3}, *m1* ∪ *m4* = {3}, *m2* ∪ *m3* = {}, *m2* ∪ *m4* = {}, *m3* ∪ *m4* = {}.
TNAV = *m1* + *m2* = {3}, *m1*+ *m3* = {3}, *m1* + *m4* = {3}, *m2* + *m3* = {}, *m2* + *m4* = {}, *m3* + *m4* = {}.
Q = 0
P = 6
$MALCOM = Q - R = 0 - 6 = \mathbf{-6}$

***Clase G*:**
Métodos y sus variables: m1= {a1, a2}, m2 = { }, m3 = { }, m4 = { }
Variables = {a1, a2}
TNUV = *m1* ∪ *m2* = {2}, *m1* ∪ *m3* = {2}, *m1* ∪ *m4* = {2}, *m2* ∪ *m3* = {}, *m2* ∪ *m4* = {}, *m3* ∪ *m4* = {}.
TNAV = *m1* + *m2* = {2}, *m1* + *m3* = {2}, *m1* + *m4* = {2}, *m2* + *m3* = {}, *m2* + *m4* = {}, *m3* + *m4* = {}.
Q = 0
P = 6
$MALCOM = Q - R = 0 - 6 = \mathbf{-6}$

***Clase H*:**
Métodos y sus variables: m1= {a1}, m2 = {a2, a3}, m3 = {a4}, m4 = {a5}, m5 = {a6}
Variables = {a1, a2, a3, a4, a5, a6}
TNUV = *m1* ∪ *m2* = {3}, *m1* ∪ *m3* = {2}, *m1* ∪ *m4* = {2}, *m1* ∪ *m5* = {2}, *m2* ∪ *m3* = {3}, *m2* ∪ *m4* = {3}, *m2* ∪ *m5* = {3}, *m3* ∪ *m4* = {2}, *m3* ∪ *m5* = {2}, *m4* ∪ *m5* = {2}.
TNAV = *m1* + *m2* = {3}, *m1* + *m3* = {2}, *m1* + *m4* = {2}, *m1* + *m5* = {2}, *m2* + *m3* = {3}, *m2* + *m4* = {3}, *m2* + *m5* = {3}, *m3* + *m4* = {2}, *m3* + *m5* = {2}, *m4* + *m5* = {2}.
Q = 0
P = 10
$MALCOM = Q - R = 0 - 10 = \mathbf{-10}$

Relación de métricas

Tabla A.1: Relación de métricas.

MÉTRICA	VALOR DE COHESIÓN								Rango	Alta cohesión	Baja cohesión	Tipo de relación
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*				
LCOM (1991)	0	2	N.A.	14	0	6	6	9	[0, max]	0	> 0	AC
LCOM2 (1994)	0	1	N.A.	14	0	6	6	10	[0, max]	0	> 0	AC
LCOM3 (1993)	1	2	N.A.	3	1	4	4	5	[0, max]	1	= 0 ó >= 2	AC
LCOM4 (1995)	1	1	N.A.	1	1	1	1	4	[0, max]	1	= 0 ó >= 2	AC y IM
TCC (1995)	1	0.33	N.A.	0.06	1	0	0	0	[0, 1]	1	< 0.5	AC
LCC (1995)	1	1	N.A.	1	1	1	1	0.16	[0, 1]	1 ó >= 0.8	< 0.5	AC y IM
LCOM5 (1996)	0	0.75	0.16	0.93	1.34	1.16	1	1	[0, 1]	0	1	AC
COH (1998)	1	0.5	0.5	0.22	0.33	0.12	0.25	0.2	[0, 1]	1	0	AC
CC* (2006)	1	0.5	N.A.	0.03	0.33	0	0	0	[0, 1]	1	0	AC
TLCOM (2010)	0	0	N.A.	0	0	6	6	8	[0, max]	0	> 0	AC y IMAC
CC (2014)	1	0.5	0.5	0.22	0.33	0.12	0.25	0.2	[0, 1]	1	0	AC
MALCOM (2014)	1	-1	N.A.	-13	1	-6	-6	-10	[- , max]	>= 1	<= 0	AC

AC: Atributo en común. IM: Invocación a método. IMAC: Invocación a método con atributo en común. CC*: Class Cohesion. N. A.: No

Conclusión y análisis de las métricas

En la tabla 1 se pueden ver los resultados de las 12 métricas que se estudiaron con los ocho ejemplos propuestos, se puede ver el rango de valores de cada métrica, el valor que se debe obtener para decir que la clase tiene alta cohesión y el valor con el cual se considera baja cohesión, también se puede ver el tipo de relación que utiliza para considerar la cohesión de una clase.

El análisis de las métricas y sus resultados se basa en la definición de cohesión el cual **se refiere al grado en que los elementos de un módulo están relacionados**. Las métricas LCOM, LCOM2, LCOM3, LCOM4, TCC, LCC, Class Cohesion, TLCOM y MALCOM no consideran las clases que tiene un solo método.

Todas las métricas consideran que la *clase A* tiene alta cohesión, del lado contrario todas las métricas consideran que la *clase H* tiene baja cohesión. Las *clases B, D, F* y *G* son considerados con baja cohesión por parte de las métricas LCOM, LCOM2, LCOM3, TCC, LCOM5, COH, Class Cohesion, CC, MALCOM. En el caso de la *clase F* a excepción de los atributos que no son utilizados, todos los demás elementos de la clase están relacionados, el método *m1* utiliza los atributos *a1, a2, a3*, *m1* invoca a *m2*, *m2* invoca a *m3* y *m3* invoca a *m4*, por lo tanto esta clase debería ser considerada como una clase con alta cohesión.

El caso la *clase G* es muy similar a la *clase F* debido a que todos sus métodos tiene interacción por la invocación de uno con otro método y los métodos no tiene relación por parte de atributos en común, solo el método uno utiliza dos atributos, por lo tanto esta clase también debe ser considera con alta cohesión y en este caso en particular no tiene atributos demás (atributos que no se estén utilizando).

La métrica TLCOM considera que las clases *D* y *E* tienen alta cohesión, sin embargo consideran con baja cohesión las *clases F* y *G,* la cuales deben ser consideradas con alta cohesión ya que existe interacción entre métodos, es decir, están relacionados ya que se necesitan entre sí para resolver el objetivo o meta de la clase. Las métricas LCOM4 y LCC consideran con alta cohesión las *clases A, B, D, E, F, G* lo cual cumple con el concepto de

cohesión ya que todos sus elementos están relacionados ya sea por un atributo en común o por invocación a un método, considerando la excepción de que las *clases F* y *G* tiene atributos demás.

A través de este estudio de métricas podemos concluir que una métrica aceptable para medir el grado de cohesión de una clase es LCOM4 o LCC, basándonos en la definición de cohesión que se refiere al grado en que los elementos de una clase están relacionados, entonces podemos decir que tenemos que ver los atributos en común que son utilizados en los métodos y como los métodos se necesitan el uno al otro, LCOM4 y LCC cumplen con estas condiciones.

Otra conclusión que podemos ver, es que todas las métricas a excepción de COH y CC no logran medir la cohesión de clases con un solo método. Consideramos que no es una buena decisión decir que las clases con un solo método automáticamente tengan alta cohesión. Se tiene que medir que ese único método en la clase utilice (tenga relación) los atributos de la clase y entonces se puede decir que esa clase tiene alta cohesión.

Para resolver el problema de las clases con un solo método, proponemos dos posibles soluciones, 1) Utilizar una de las dos métricas COH o CC solo para los casos en que la clase solo tenga un método, 2) La segunda solución es una modificación o extensión a la métrica LCOM4, si LCOM4 considera los vértices como los métodos de la clase y las aristas el atributo en común que utilizan los métodos o la llamada de un método al otro método. En el caso donde solo tenemos un método, invertimos los papeles, es decir, los vértices serían los atributos y las aristas serian si se están utilizando los atributos en el método, de este modo tendríamos el grafo que nos indicaría el grado de cohesión para las clases que solo tiene un método.

Ahora tenemos una nueva métrica **LCOM4 – A** generada a partir de la extensión de LCOM4, la cual la definimos como:

Se considera un grafo G donde los vértices son representados por los atributos de la clase a_i y a_j y las aristas se crean si los atributos son utilizados en el método de la clase. En otras palabras se puede decir que **LCOM4 – A** mide el grado de relación entre variables y un método de la clase.

Si **LCOM4 – A** = 1 indica una clase cohesiva.
Si **LCOM4 – A** >= 2 indica un problema, la clase tiene variables que posiblemente puedan ser eliminadas de la clase.

La diferencia entre LCOM4 –A es el resultado que obtenemos ya que, si el valor es mayo o igual a 2 solo tenemos que restarle uno al valor que se obtuvo y de eso modo sabremos el número de variables que no se están usando.

ANEXO B PRUEBAS

En este anexo se presentan los códigos utilizados en cada caso de prueba, también se presentan los archivos generados después de pasar por el proceso de refactorización y el código refactorizado.

Código completo del caso de prueba 1.

```
#include<iostream>
class Programming
{
    private:
        int number;

    public:
        void input_value()
        {
            std::cout << "In function input_value, Enter an integer\n";
            std::cin >> number;
        }

        void output_value()
        {
            std::cout << "Variable entered is ";
            std::cout << number << "\n";
        }
};
int main()
{
    Programming *object = new Programming();

    object->input_value();
    object->output_value();

     int n;
     std::cin>>n;
    return 0;
}
```

Figura B.1 Código completo del caso de prueba 1

Archivo de entrada del caso de prueba 1

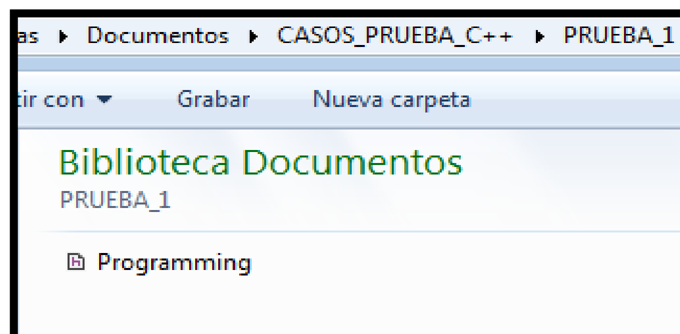


Figura B.2 Archivo de entrada del caso de prueba 1

Código refactorizado del caso de prueba 1.

```
#include<iostream>
#include "Programming.h"

int main()
{
    Programming *object = new Programming();

    object->input_value();
    object->output_value();

     int n;
     std::cin>>n;
    return 0;
}
```

Figura B.3 Código refactorizado del archivo main.cpp

```
#ifndef _Programming
#define _Programming

class Programming
    {

        private:
                int number;

        public:
                 Programming();
                ~Programming();
        public:
                 void output_value();
                 void input_value();
    };

#endif
```

Figura B.4 Código refactorizado del archivo Programming.h

```
#include<iostream>
#include "Programming.h"

void Programming::input_value()
    {
        std::cout << "In function input_value, Enter an integer\n";
        std::cin >> number;
    }

void Programming::output_value()
    {
        std::cout << "Variable entered is ";
        std::cout << number << "\n";
    }
```

Figura B.5 Código refactorizado del archivo Programming.cpp

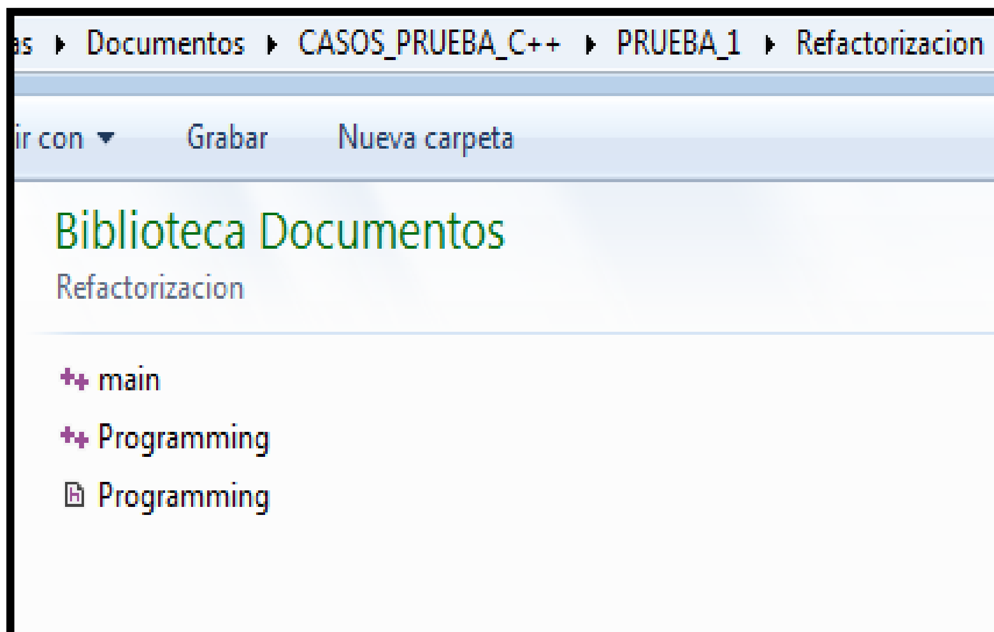


Figura B.6 Archivos generados después de pasar por el proceso de refactorización.

Código completo del caso de prueba 2.

```
#include <iostream>
class Temp
{
    private:
        int data1;
        float data2;
    public:
       void int_data(int d)
       {
          data1=d;
         std::cout<<"Number:  "<<data1;
       }
       float float_data()
       {
           std::cout<<"\nEnter data: ";
           std::cin>>data2;
           return data2;
       }
};
 int main()
 {
      Temp *obj1 = new Temp();
      Temp *obj2 = new Temp();
      obj1->int_data(12);
      std::cout<<"You entered "<<obj2->float_data();
      return 0;
 }
```

Figura B.7 Código completo del caso de prueba 2

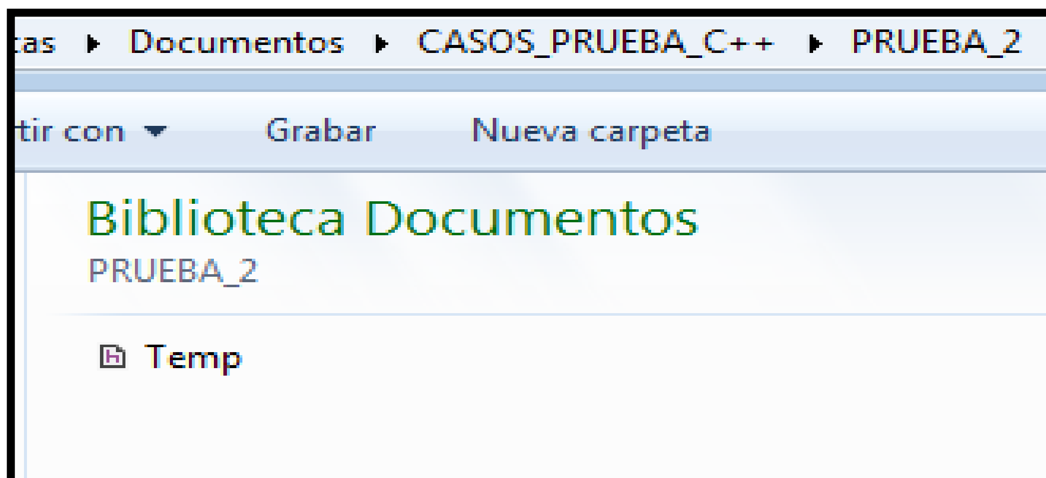


Figura B.8 Archivo de entrada del caso de prueba 2

Código refactorizado del caso de prueba 2.

```
#include <iostream>
#include "Temp.h"
#include "Cohesion82_Temp.h"

int main()
{
    Cohesion82_Temp *cohesion82_temp = new Cohesion82_Temp();
    Temp *obj1 = new Temp();

    obj1->int_data(12);
        std::cout<<"You entered "<<cohesion82_temp->float_data();
    return 0;
}
```

Figura B.9 Código refactorizado del archivo main.cpp.

```
#ifndef _Temp
#define _Temp

class Temp
  {

     private:
             int data1;

     public:
            Temp();
           ~Temp();
     public:
            void int_data ( int d );
  };
#endif
```

Figura B.10 Código refactorizado del archivo Temp.h

```
#include <iostream>
#include "Temp.h"

void Temp::int_data ( int d )
        {
           data1=d;
         std::cout<<"Number: "<<data1;
        }
```

Figura B.11 Código refactorizado del archivo Temp.cpp

```
#ifndef _Cohesion82_Temp
#define _Cohesion82_Temp
class Cohesion82_Temp
  {

     private:
          float data2;

     public:
              Cohesion82_Temp();
             ~Cohesion82_Temp();
     public:
              float float_data();
  };
#endif
```

Figura B.12 Código refactorizado del archivo Cohesion82_Temp.h

```
#include <iostream>
#include "Cohesion82_Temp.h"
float Cohesion82_Temp::float_data()
        {
            std::cout<<"\nEnter data: ";
            std::cin>>data2;
            return data2;
        }
```

Figura B.13 Código refactorizado del archivo Temp.cpp.

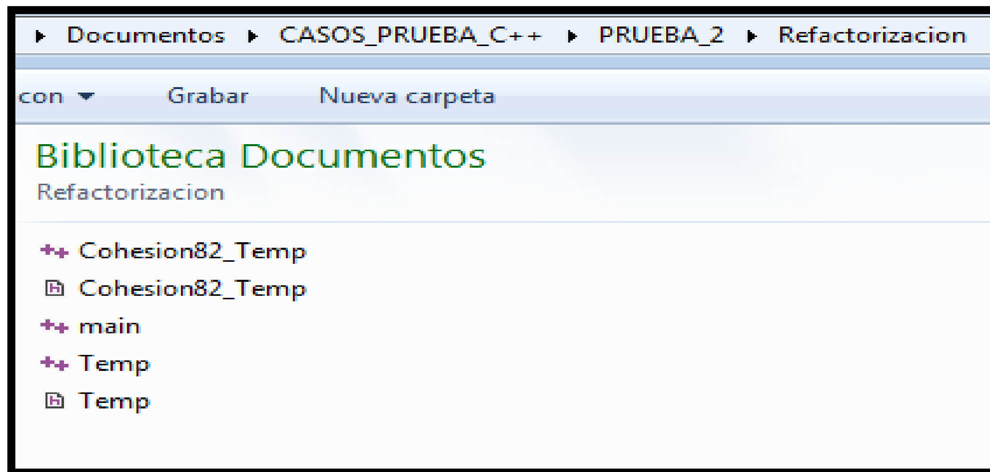


Figura B.14 Archivos generados después de pasar por el proceso de refactorización.

Código completo del caso de prueba 3.

```
#include "stdafx.h"
#include <iostream>
class Operaciones
{
public:
    int operacionSimple(int valor1, int valor2)
    {
        int resultado = 0;

        resultado = valor1 + valor2;

        return resultado;
    }
    int operacionCompuesta(int valor1, int valor2)
    {
        int resultado = 0;

        resultado = operacionSimple(valor1, valor2);

        resultado = resultado * valor2;

        return resultado;
    }
     int operacionDos(int valor1)
     {
         int resultado = 0;
         resultado = operacionTres(valor1);
         resultado = resultado - valor1;

         return resultado;
     }

     int operacionTres(int valor1)
     {
         int resultado = 0;

         resultado = valor1 * valor1;

         return resultado;
     }
```

```
int operacionUno(int valor1, int valor2)
{
    int resultado = 0;

    resultado = operacionDos(valor1);
    resultado = resultado / valor2;

    return resultado;
}

int main()
{
    Operaciones *operacion = new Operaciones();
    int resultado = 0;

    resultado = operacion->operacionCompuesta(5,10);
    std::cout<<"Resultado: "<<resultado<<std::endl;

    Operaciones *calcular = new Operaciones();

    resultado = calcular->operacionUno(6,2);
    std::cout<<"Resultado: "<<resultado;

    int n;
    std::cin>>n;
    return 0;
}
```

Figura B.15 Código completo del caso de prueba 3.

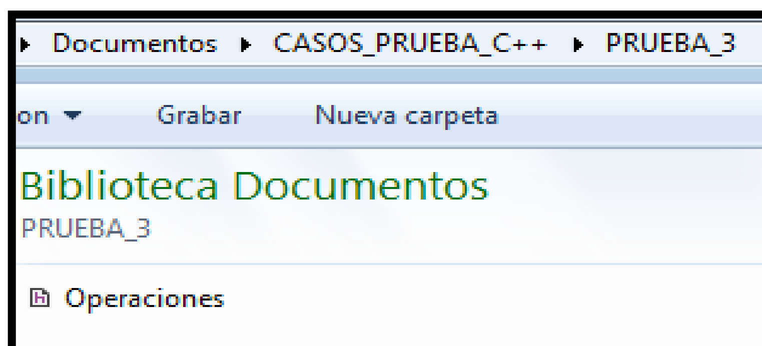


Figura B.16 Archivo de entrada del caso de prueba 3

Código refactorizado del caso de prueba 3.

```
#include "main.h"
#include <iostream>
#include "Operaciones.h"
#include "Cohesion710_Operaciones.h"
int main()
{
    Cohesion710_Operaciones *cohesion710_operaciones = new Cohesion710_Operaciones();
    Operaciones *operacion = new Operaciones();
    int resultado = 0;

    resultado = operacion->operacionCompuesta(5,10);
    std::cout<<"Resultado: "<<resultado<<std::endl;

        resultado = cohesion710_operaciones->operacionUno(6,2);
    std::cout<<"Resultado: "<<resultado;

    int n;
    std::cin>>n;
    return 0;
}
```

Figura B.17 Código refactorizado del archivo main.cpp

```
#ifndef _Operaciones
#define _Operaciones
class Operaciones
   {
     public:
            Operaciones();
           ~Operaciones();
     public:
            int operacionCompuesta ( int valor1 , int valor2 );
            int operacionSimple ( int valor1 , int valor2 );
   };
#endif
```

Figura B.18 Código refactorizado del archivo Operaciones.h

```cpp
#include "Operaciones.h"
#include <iostream>
int Operaciones::operacionSimple ( int valor1 , int valor2 )
    {
        int resultado = 0;

        resultado = valor1 + valor2;

        return resultado;
    }

int Operaciones::operacionCompuesta ( int valor1 , int valor2 )
    {
        int resultado = 0;

        resultado = operacionSimple(valor1, valor2);

        resultado = resultado * valor2;

        return resultado;
    }
```

Figura B.19 Código refactorizado del archivo Operaciones.cpp

```cpp
#ifndef _Cohesion710_Operaciones
#define _Cohesion710_Operaciones
class Cohesion710_Operaciones
  {
     public:
              Cohesion710_Operaciones();
             ~Cohesion710_Operaciones();
     public:
              int operacionUno ( int valor1 , int valor2 );
              int operacionTres ( int valor1 );
              int operacionDos ( int valor1 );
  };
#endif
```

Figura B.20 Código refactorizado del archivo Cohesion710_Operaciones.h

```cpp
#include "Cohesion710_Operaciones.h"
#include <iostream>
int Cohesion710_Operaciones::operacionDos ( int valor1 )
    {
        int resultado = 0;
        resultado = operacionTres(valor1);
        resultado = resultado - valor1;

        return resultado;
    }

int Cohesion710_Operaciones::operacionTres ( int valor1 )
    {
        int resultado = 0;

        resultado = valor1 * valor1;

        return resultado;
    }

int Cohesion710_Operaciones::operacionUno ( int valor1 , int valor2 )
    {
        int resultado = 0;

        resultado = operacionDos(valor1);
        resultado = resultado / valor2;

        return resultado;
    }
```

Figura B.21 Código refactorizado del archivo Cohesion710_Operaciones.cpp

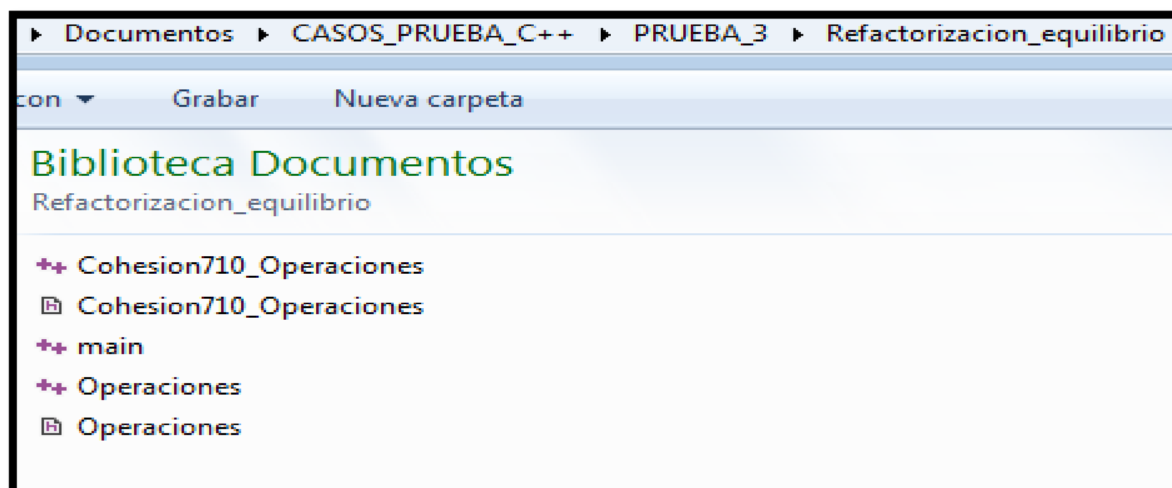


Figura B.22 Archivos generados después de pasar por el proceso de refactorización.

Código completo del caso de prueba 4.

```
#include "stdafx.h"
#include "Figura.h"
#include <iostream>
int main()
{
    double result = 0;
    Figura *rect = new Figura(10.0, 5);
    result = rect->areaRectangulo();
    std::cout << "areaRectangulo: "  << result  <<std::endl;

    Figura *trian = new Figura(4.0, 6.0);
    result = trian->areaIsosceles();
    std::cout << "areaIsosceles: "  << result  <<std::endl;

    Figura *cylin = new Figura(3.0, 4.0);
    result = cylin->areaCylinder();
    std::cout << "areaCylinder: "   << result   <<std::endl;

     int n;
    std::cin>>n;
}
```

Figura B.23 Código completo del archivo main.cpp

```
#pragma once
class Figura
{
    public:
        Figura(double height, double width);
        ~Figura();
        double height;
        double width;

    public:
        double areaRectangulo();
        double areaIsosceles();
        double areaCylinder();
};
```

Figura B.24 Código completo del archivo Figura.h.

```cpp
#include "stdafx.h"
#include "Figura.h"
Figura::Figura(double height, double width)
{
    this->height = height;
    this->width = width;
}
double Figura::areaRectangulo()
{
    return this->height * this->width;
}
double Figura::areaIsosceles()
{
    return 0.5 * width * height;
}
double Figura::areaCylinder()
{
    return (2 * 3.1416 * (width/2) * (width/2)) + (3.1416 * width * height);
}
```

Figura B.25 Código completo del archivo Figura.cpp

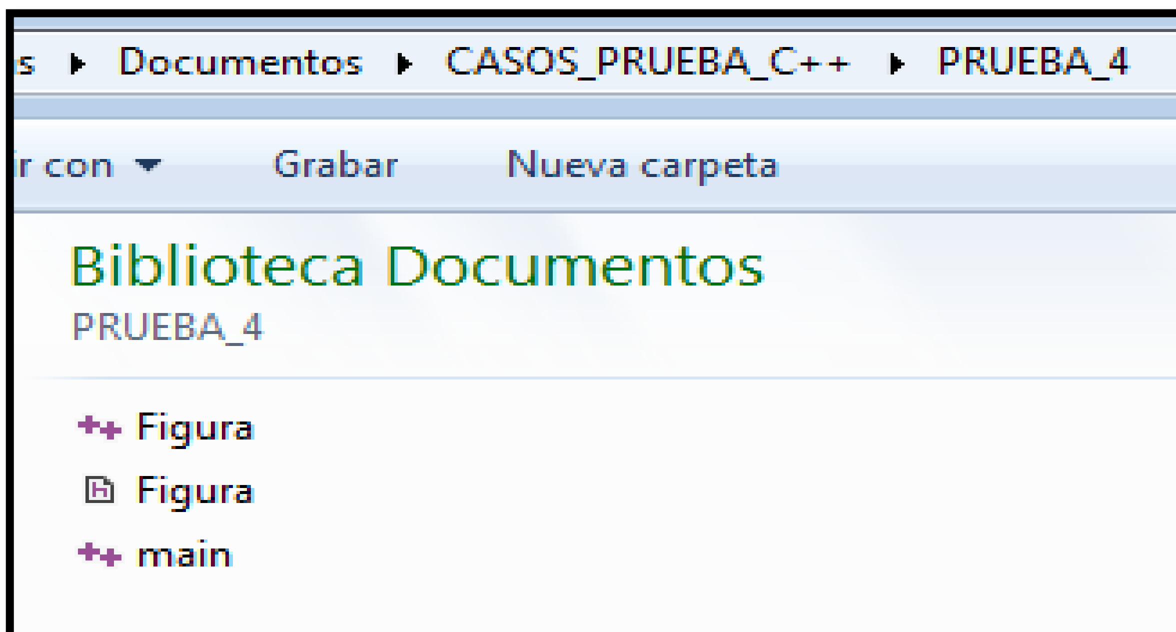


Figura B.26 Archivos de entrada del caso de prueba 4.

Código refactorizado del caso de prueba 4.

```
#include "stdafx.h"
#include "Figura.h"
#include <iostream>
#include "main.cpp"
int main()
{
    CrareaRectangulo1 *crarearectangulo1 = new CrareaRectangulo1( 10.0 ,  5  );
    CrareaIsosceles2 *crareaisosceles2 = new CrareaIsosceles2( 4.0 ,  6.0  );
    double result = 0;

        result = crarearectangulo1->areaRectangulo();
    std::cout << "areaRectangulo: "  << result  <<std::endl;

        result = crareaisosceles2->areaIsosceles();
    std::cout << "areaIsosceles: "  << result  <<std::endl;

    Figura *cylin = new Figura(3.0, 4.0);
    result = cylin->areaCylinder();
    std::cout << "areaCylinder: "   << result   <<std::endl;
     int n;
    std::cin>>n;
}
```

Figura B.27 Código refactorizado del archivo main.cpp

```
#ifndef _Figura
#define _Figura
#include "CrareaRectangulo1Base"
class Figura: public CrareaRectangulo1Base
  {
     public:
           Figura(double height, double width);
          ~Figura();
     public:
           double areaCylinder();
  };
#endif
```

Figura B.28 Código refactorizado del archivo Figura.h

```
#include "stdafx.h"
#include "Figura.h"

Figura::Figura(double height, double width)
{
    this->height=height;
    this->width=width;
}
double Figura::areaCylinder()
{
    return (2 * 3.1416 * (width/2) * (width/2)) + (3.1416 * width * height);
}
```

Figura B.29 Código refactorizado del archivo Figura.cpp

```
#ifndef _CrareaRectangulo1Base
#define _CrareaRectangulo1Base
 class CrareaRectangulo1Base
  {

    protected:
        double height;
        double width;

    public:
        CrareaRectangulo1Base();
        ~CrareaRectangulo1Base();
  };
#endif
```

Figura B.30 Código refactorizado del archivo CrareaRectangulo1Base

```
#ifndef _CrareaRectangulo1
#define _CrareaRectangulo1
#include "CrareaRectangulo1Base"
class CrareaRectangulo1: public CrareaRectangulo1Base
  {
    public:
          CrareaRectangulo1(double height, double width);
         ~CrareaRectangulo1();
    public:
          double areaRectangulo();
  };
#endif
```

Figura B.31 Código refactorizado del archivo CrareaRectangulo1.h

```
#include "CrareaRectangulo1.h"
CrareaRectangulo1::CrareaRectangulo1(double height, double width)
{
    this->height=height;
    this->width=width;
}
double CrareaRectangulo1::areaRectangulo()
{
     return this->height * this->width;
}
```

Figura B.32 Código refactorizado del archivo CrareaRectangulo1.cpp

```
#ifndef _CrareaIsosceles2
#define _CrareaIsosceles2
#include "CrareaRectangulo1Base"
class CrareaIsosceles2: public CrareaRectangulo1Base
  {
    public:
           CrareaIsosceles2(double height, double width);
          ~CrareaIsosceles2();
    public:
           double areaIsosceles();
  };
#endif
```

Figura B.33 Código refactorizado del archivo CrareaIsosceles2.h

```
#include "CrareaIsosceles2.h"

CrareaIsosceles2::CrareaIsosceles2(double height, double width)
{
    this->height=height;
    this->width=width;
}

double CrareaIsosceles2::areaIsosceles()
{
    return 0.5 * width * height;
}
```

Figura B.34 Código refactorizado del archivo CrareaIsosceles2.cpp

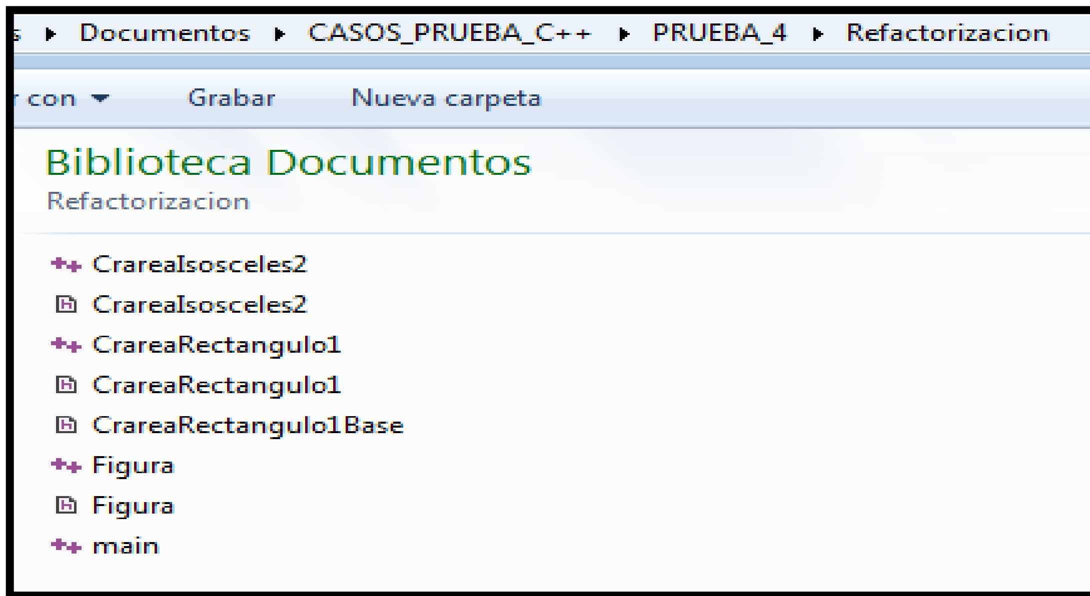


Figura B.35 Archivos generados después de pasar por el proceso de refactorización.

Código completo del caso de prueba 5.

```
#ifndef __PRUEBACONSTRUCTOR_H__
#define __PRUEBACONSTRUCTOR_H__
#include<iostream>
class Operaciones
 {
    public:
        Operaciones();
        virtual int calcular()=0;
};
class Suma: public Operaciones{
    public:
        int x;
        int y;
        int valorUno;
        int valorDos;
    public:
        Suma(int valorUno, int valorDos);
        int calcular();
};
class Multiplica: public Operaciones{
    public:
        int baseMayor;
        int baseMenor;
        int resultadoPrevio;
        int altura;
        int valorUno;
        int valorDos;
    public:
        Multiplica(int valorUno, int valorDos, int baseMayor, int baseMenor, int altura);
        int calcular();
        int calcularAreaTrapecio();
};
class Resta: public Operaciones{
    public:
        int valorUno;
        int valorDos;
    public:
        Resta(int valorUno, int valorDos);
        int calcular();
};
class PruebaConstructor
 {
   private:
     int resultado;
     int valor1;
     int valor2;
     int altura;
    Operaciones *operation;
    operation = new Suma(valor1, valor2);
   public:
     PruebaConstructor (int valor1, int valor2, int altura);
     int resultadoCalculo(int typeOperation);
     int soloPrueba();
 };
```

Figura B.36 Código del archivo PruebaConstructor.h

```
#include "PruebaConstructor.h"
Suma::Suma(int valorUno, int valorDos)
{
    this->valorUno = valorUno;
    ::valorDos = valorDos;
    this->x = 0;
    y = 0;
}
int Suma::calcular()
{
    return this->valorUno + this->valorDos;
}
Resta::Resta(int valorUno, int valorDos)
{
    this->valorUno = valorUno;
    this->valorDos = valorDos;
}
int Resta::calcular()
{
    return this->valorUno - this->valorDos;
}
Multiplica::Multiplica(int valorUno, int valorDos, int baseMayor, int baseMenor, int altura)
{
    this->valorUno = valorUno;
    this->valorDos = valorDos;
    this->baseMayor = baseMayor;
    this->baseMenor = baseMenor;
    this->altura = altura;
}
int Multiplica::calcular()
{
    return this->valorUno * this->valorDos;
}
int Multiplica::calcularAreaTrapecio()
{
    this->resultadoPrevio = (this->baseMayor + this->baseMenor)/2;
    return this->resultadoPrevio * this->altura;
}
PruebaConstructor::PruebaConstructor(int valor1, int valor2, int altura)
{
    this->valor1 = valor1;
    this->valor2 = valor2;
    this->altura = altura;
}
int PruebaConstructor::resultadoCalculo(int typeOperation)
{
    if(typeOperation == 1)
    {
        this->resultado = operation->calcular();
    }
    else
    {
        if(typeOperation == 2)
        {
            operation = new Resta(valor1, valor2);
            this->resultado = operation->calcular();
        }
        else
        {
            operation = new Multiplica(valor1, valor2, valor1, valor2, altura);
            if(typeOperation == 3)
            {
                this->resultado = operation->calcular();
            }
            else
            {
                this->resultado = operation->calcularAreaTrapecio();
            }
        }
    }
    return this->resultado;
}
int PruebaConstructor::soloPrueba()
{
    int numPar = 0;
    numPar = valor1;
    return 2;
}
```

Figura B.37 Código del archivo PruebaConstructor.cpp

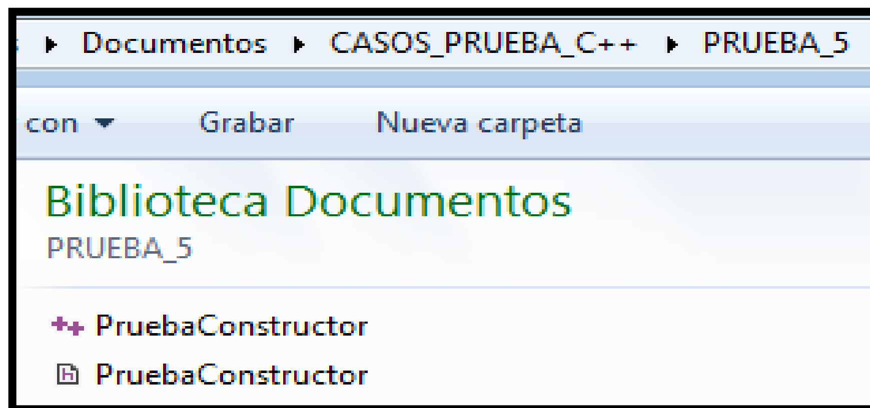


Figura B.38 Archivos de entrada del caso de prueba5

Código refactorizado del caso de prueba 5.

```cpp
#ifndef _Resta
#define _Resta
class Resta: public Operaciones
  {
     public:
            int valorUno;
            int valorDos;
     public:
            Resta(int valorUno, int valorDos);
           ~Resta();
     public:
            int calcular();
  };
#endif
```

Figura B.39 Código refactorizado del archivo Resta.h

```cpp
#include<iostream>
#include "Resta.h"
Resta::Resta(int valorUno, int valorDos)
{
    this->valorUno=valorUno;
    this->valorDos=valorDos;
}
int Resta::calcular()
{
    return this->valorUno - this->valorDos;
}
```

Figura B.40 Código refactorizado del archivo Resta.cpp

```
#ifndef _Suma
#define _Suma
class Suma: public Operaciones
  {
    public:
           int x;
           int y;
           int valorUno;
           int valorDos;
    public:
           Suma(int valorUno, int valorDos);
          ~Suma();
    public:
           int calcular();
  };
#endif
```

Figura B.41 Código refactorizado del archivo Suma.h.

```
#include<iostream>
#include "Suma.h"
Suma::Suma(int valorUno, int valorDos)
{
    this->valorUno=valorUno;
    this->valorDos=valorDos;
    this->x=0;
    this->y=0;
}
int Suma::calcular()
{
    return this->valorUno + this->valorDos;
}
```

Figura B.42 Código refactorizado del archivo Suma.cpp

```
#ifndef _Multiplica
#define _Multiplica
class Multiplica: public Operaciones
  {
    public:
           int valorUno;
           int valorDos;
    public:
           Multiplica(int valorUno, int valorDos);
          ~Multiplica();
    public:
           int calcular();
  };
#endif
```

Figura B.43 Código refactorizado del archivo Multiplica.h.

```
#include<iostream>
#include "Multiplica.h"
Multiplica::Multiplica(int valorUno, int valorDos)
{
	this->valorUno=valorUno;
	this->valorDos=valorDos;
}
int Multiplica::calcular()
{
	return this->valorUno * this->valorDos;
}
```

Figura B.44 Código refactorizado del archivo Multiplica.cpp

```
#ifndef _Cohesion35_Multiplica
#define _Cohesion35_Multiplica
class Cohesion35_Multiplica
  {
     private:
          int baseMayor;
          int baseMenor;
          int resultadoPrevio;
          int altura;
     public:
            Cohesion35_Multiplica(int baseMayor, int baseMenor, int altura);
           ~Cohesion35_Multiplica();
     public:
            int calcularAreaTrapecio();
  };
#endif
```

Figura B.45 Código refactorizado del archivo Cohesion35_Multiplica.h

```
#include<iostream>
#include "Cohesion35_Multiplica.h"
  Cohesion35_Multiplica::Cohesion35_Multiplica(int baseMayor, int baseMenor, int altura)
{
    this->baseMayor=baseMayor;
    this->baseMenor=baseMenor;
    this->altura=altura;
}
int Cohesion35_Multiplica::calcularAreaTrapecio()
{
    this->resultadoPrevio = (this->baseMayor + this->baseMenor)/2;
    return this->resultadoPrevio * this->altura;
}
```

Figura B.46 Código refactorizado del archivo Cohesion35_Multiplica.cpp

```
#ifndef _Operaciones
#define _Operaciones
class Operaciones
  {
     public:
            Operaciones();
           ~Operaciones();
     public:
            virtual int calcular()= 0;
  };
#endif
```

Figura B.47 Código refactorizado del archivo Operaciones.h

```
#ifndef _PruebaConstructor
#define _PruebaConstructor
#include<iostream>
#include "Operaciones.h"
#include "Suma.h"
#include "Cohesion35_Multiplica.h"
class PruebaConstructor
  {
     private:
Cohesion35_Multiplica *cohesion35_multiplica = new Cohesion35_Multiplica(valor1,valor2,valor1);
        int resultado;
        int valor1;
        int valor2;
        int altura;
        Operaciones *operation;
        operation = new Suma(valor1, valor2);
     public:
            PruebaConstructor(int valor1, int valor2, int altura);
           ~PruebaConstructor();
     public:
            int soloPrueba();
            int resultadoCalculo ( int typeOperation );
  };
#endif
```

Figura B.48 Código refactorizado del archivo PruebaConstructor.h

```
#include "Resta.h"
PruebaConstructor::PruebaConstructor(int valor1, int valor2, int altura)
{
    this->valor1=valor1;
    this->valor2=valor2;
    this->altura=altura;
}
int PruebaConstructor::resultadoCalculo ( int typeOperation )
{
    if(typeOperation == 1)                                            //Su
    {
        this->resultado = operation->calcular();
    }
    else
    {
        if(typeOperation == 2)                                        //Re
        {
            operation = new Resta(valor1, valor2);
            this->resultado = operation->calcular();
        }
        else
        {
    operation = new Multiplica( valor1 ,  valor2  );
            if(typeOperation == 3)                                    // M
            {
                this->resultado = operation->calcular();
            }
            else                                                      // C
            {
                    this->resultado = cohesion35_multiplica->calcularAreaTrapecio();
            }
        }
    }
    return this->resultado;
}
int PruebaConstructor::soloPrueba()
{
    int numPar = 0;
    numPar = valor1;
    return 2;
}
```

Figura B.49 Código refactorizado del archivo PruebaConstructor.cpp

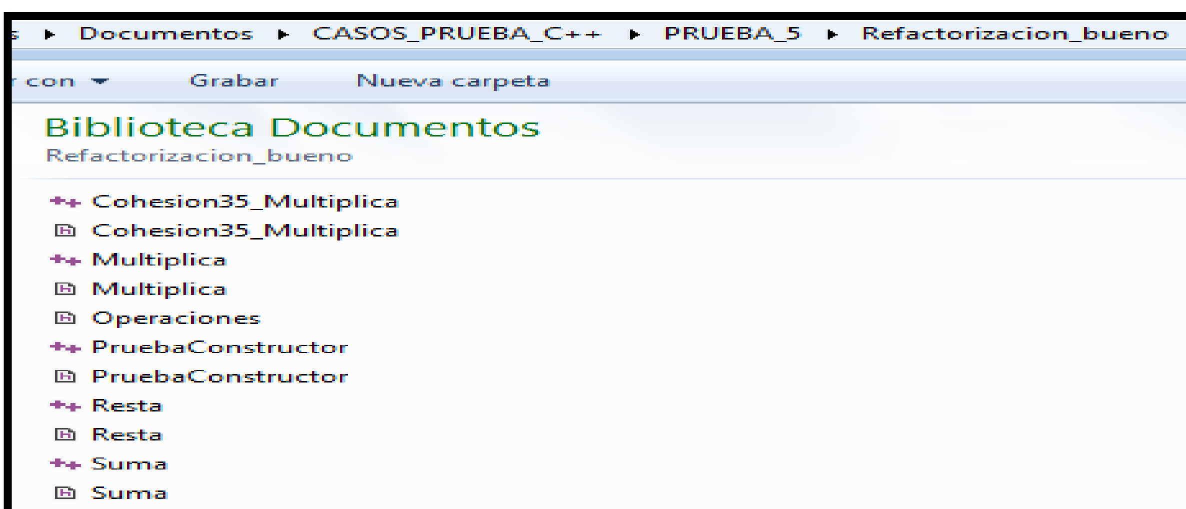


Figura B.50 Archivos generados después de pasar por el proceso de refactorización.