SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Método de Re-factorización de código java con interfaces y abstracciones incorrectas

presentada por

Ing. Pablo Padilla Salgado

como requisito para la obtención del grado de

Maestro en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Codirector de tesis

Dra. María Clara Gómez Álvarez

Cuernavaca, Morelos, México. Julio de 2019.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

"2019, Año del Caudillo del Sur, Emiliano Zapata"

Cuernavaca, Morelos a 24 de junio del 2019
OFICIO No. DCC/067/2019

Asunto: Aceptación de documento de tesis

DR. GERARDO V. GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial del Ing. Pablo Padilla Salgado, con número de control M17CE039, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis profesional titulado "Métodos de re-factorización de código java con interfaces y abstracciones incorrectas" 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

CO-DIRECTOR DE TESIS

Dra. María Clara Gómez Álvarez
Doctora en Ingeniería Énfasis en Sistemas e Informática

REVISOR 1

M. Guillén.

M.C. Mario Guillén Rodríguez
Maestro en Ciencias con Especialidad en Sistemas Computacionales
7573768

REVISOR 2

Dr. Juan Carlos Rojas Pérez
Doctor en Ciencias en Ciencias de la Computación
6099372

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

NACS/lmz

"2019, Año del Caudillo del Sur, Emiliano Zapata"

Cuernavaca, Mor., 26/junio/2019
No. de Oficio: SAC/234/2019
Asunto: Autorización de impresión de Tesis

ING. PABLO PADILLA SALGADO
CANDIDATO AL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"Métodos de re-factorización de código java con interfaces y abstracciones incorrectas"**, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

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

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

SEP TecNM
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SUBDIRECCIÓN ACADÉMICA

DR. GERARDO VICENTE GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO

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

GVGR/mcr

Dedicatorias

*Con mucho **amor:***

*A **Dios** por su infinita misericordia y por permitirme llegar a disfrutar este día.*

*A mis padres **Pablo Marcos** y **Juana** por guiarme con su sabiduría y siempre estar para mí cuando más lo necesito. Estoy muy orgulloso de ser su hijo*

*A mi novia **Mariana Díaz** por estar siempre a mi lado apoyándome durante este proceso, por su confianza y su cariño*

.

*A mis hermanos **Iván** y **Miriam** por compartirme sus alegrías, por su confianza y cuidarnos creciendo juntos.*

*A mi abuela **Victoria** por ser mi ejemplo de fortaleza.*

La educación es el arma más poderosa que puedes usar para cambiar el mundo.

(Nelson Mandela)

Agradecimientos

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

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

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

A la Universidad de Medellín por brindarme la oportunidad de realizar una estancia en el mes de octubre del 2018, por el espacio, experiencia y el tiempo para completar mis actividades de la maestría.

A mi director de tesis, Dr. René Santaolaya Salgado, por compartir su conocimiento y dirigirme con su supervisión en el desarrollo de esta investigación, por su paciencia y valiosa confianza, pero sobre todo por su motivación, integridad y el apoyo que me brindo día a día durante esta etapa de mi vida profesional y académica.

A mi Co-directora Dra. María Clara Gómez Álvarez, por compartir su conocimiento y dirigirme con su supervisión en el desarrollo de esta investigación, por su amabilidad y confianza, sobre todo por la motivación, integridad y el apoyo que me brindo durante mi estancia en la Universidad de Medellín en esta etapa de mi vida profesional y académica.

A mis revisores, Dr. Juan Carlos Rojas Pérez y MC. Mario Guillen Rodríguez por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación

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

A la M.A Ernestina Anguiano Bello docente del Tecnológico de Iguala la cual fue la persona que me emotivo para estudiar la maestría, por el apoyo y confianza que tuvo en mí, para poder seguir preparándome académicamente.

A mis compañeros que estuvieron conmigo en todo el recorrido de mi carrera, por la convivencia que tuvimos, confianza: Damián, Luis, Iván, Neri, Rebeca, Orlando, Violeta, Juan Carlos, Vitervo, Heidi y Xico.

RESUMEN

La programación orientada a objetos es uno de los paradigmas actuales que sirven para producir software de calidad, siempre y cuando se respeten y cumplan los principios del diseño orientado a objetos. Uno de estos, es la Separación de Interfaces, que establece que los clientes de cualquier software no deben ser forzados a depender de interfaces que no ocupan.

Un marco de aplicaciones orientado a objetos es un conjunto de clases abstractas que trabajan juntas para ofrecer soluciones a muchos problemas específicos u ofrecer servicios dentro de un mismo dominio de aplicaciones. Los marcos deben ofrecer métodos para extender su funcionalidad y dar facilidades para reusar partes del marco de manera independiente.

Si un marco de aplicaciones orientado a objetos cumple con sus principios orientados a objetos, puede ser reusado y extendido sin ningún problema. Si se logra el reuso de código, se reduce el tiempo de programación de una nueva aplicación y, por lo tanto, se reduce la posibilidad de defectos, ya que se está usando código previamente probado.

Cuando los marcos no están bien diseñados no se cuenta con estas características, por lo que es necesario obtenerlas mejorando la arquitectura del marco. La refactorización consiste en modificar la arquitectura del marco sin afectar a la funcionalidad del mismo.

Una situación que con frecuencia surge en el diseño de un marco es el problema de dependencia 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 dependencia 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.

Para resolver este problema se implementa el método de refactorización denominado Separación de Interfaces realizado por [Valdez, 2004], cuyo algoritmo fue implementado satisfactoriamente en una herramienta que hace refactorización de manera automática.

Aunque se sabe que este problema existe, no se tiene una forma para medir hasta qué grado este problema afecta a los marcos. En esta tesis se implementa una métrica orientada a objetos que mide el grado de dependencia debido a interfaces que no se ocupan.

Se presentan casos de estudio para mostrar cómo esta métrica ayuda a detectar cuando los marcos tienen un problema grave de dependencia de interfaces. Con esta información se puede tomar una decisión cuantitativa para encargarse del problema.

Se muestra además para estos casos de estudio cómo se lleva a cabo la refactorización de forma automática, utilizando la herramienta diseñada, y las ventajas que se obtienen al hacer este proceso sobre un marco.

Abstract

Object-Oriented programming is one of the current paradigms for producing quality software, providing that the object-oriented design principles are respected and fulfilled. One principle is the interface separation, which states that clients of any given software must not be forced to depend on interfaces they do not need.

Object-Oriented frameworks are sets of classes designed to work together in order to offer generic solutions to many specific problems or offer services within the same application domain. Frameworks must offer methods to extend its functionality and give facilities for reusing sections of the framework in an independent manner.

If an object-oriented framework complies with object-oriented principles, it can be reused and extended without any trouble. If code reuse is achieved, we reduce the time of programming a new application, and also reduce the possibility of defects, since we are using already-proven code.

When frameworks are not well-architectured they do not have these qualities, so it is necessary to obtain them improving the framework architecture. Refactoring is used to change the architecture, which consist on modifying the framework architecture without affecting its functionality.

A situation that often arises from the design of a framework is the interface dependency problem produced by interface inheritance when subclasses do not really need those interfaces.

When a framework has the interface dependency problem, its functionality cannot be reused separately, and cannot be extended either without violating other object-oriented principles, like the open / close principle.

To solve this problem, the method of refactoring called Interface Separation performed by [Valdez, 2004] is implemented, whose algorithm was implemented successfully in a tool that automatically refactorizes.

Although we know that this problem exists, we do not have a way to measure to what extent this problem affects frameworks. In this dissertation an object-oriented metric to measure the degree of dependency due to unused interfaces is proposed.

Case studies are presented in order to show how this metric helps to detect when frameworks have a serious interface dependency problem. With this information a quantitative decision can be made to take care of the problem.

For these case studies we also show how to realize the refactoring in an automatic manner, using the designed refactoring tool, and the benefits acquired when this process is done in a framework.

TABLA DE CONTENIDO

LISTADO DE FIGURAS

LISTADO DE TABLAS

GLOSARIO DE TÉRMINOS

ACOPLAMIENTO — Es una medida del grado de dependencia entre módulos. Es decir, el modo en que un módulo está siendo afectado por la estructura interna de otro módulo [38].

CLASE ABSTRACTA — Es una clase que al menos tiene un método abstracto o más interfaces, las cuales no tienen implementación dentro de la clase, sino que son implementadas en las clases derivadas de la clase abstracta [49].

CLASE DERIVADA — Es una clase que hereda de una clase base. Esta clase debe implementar todos los métodos abstractos o interfaces que tenga definidas la clase base [50].

C-NOC — La métrica C-NOC Chidamber – Number of Children, Número de Hijos. Es una métrica definida en [1] que representa el número de subclases inmediatas subordinadas a una clase en la jerarquía de clases. Es decir, es una medida de cuántas subclases van a heredar los métodos de la clase padre.

COMMAND — Comando, patrón de diseño 'Command' definido en el catálogo de patrones de diseño de Gamma [2]. Su intención es encapsular un mensaje como un objeto, con lo que permite parametrizar a los clientes, gestionar colas de mensajes y deshacer operaciones.

COMPONENTE — Aunque en [3] se establece que no existe un consenso generalizado entre la comunidad de Ingenieros del Software del mundo para definir lo que se entiende por componente, aquí se trata de esclarecer este concepto. [4] define a un componente como: "una unidad de composición de software, que puede ser desarrollado, adquirido y utilizado independientemente, y que define las interfaces por medio de las cuáles

	puede ser ensamblado con otros componentes para proveer y usar servicios".
CONFIABILIDAD	Es la capacidad del hardware o software de computación para realizar lo que el usuario espera y hacerlo de manera consistente, sin fallas ni comportamiento errático.
EFICIENCIA	Es una medida del rendimiento, expresada en tiempo de respuesta como en tiempo de procesamiento, del software y la cantidad de recursos utilizados, bajo ciertas condiciones.
EXTENSIBILIDAD	Es una medida de la capacidad que tiene un componente o un sistema de software para aceptar nuevos comandos definidos por el usuario o desarrollador de aplicaciones.
FLEXIBILIDAD	Es el número de opciones que un programador tiene para determinar el uso del componente, también llamado "generalidad".
FUNCIONALIDAD	Es el grado de funciones que posee el software necesaria y suficientemente para satisfacer las necesidades de un usuario.
MANTENIBILIDAD	Es el grado del esfuerzo requerido para modificar, mejorar o corregir errores en el software con la intención de incrementar la eficiencia o funcionamiento del software.
MAOO	Marco de Aplicaciones Orientado a Objetos. Es un conjunto de clases en colaboración que abarca un diseño abstracto para dar soluciones a una familia de problemas relacionados.
MÉTRICA DE SOFTWARE	Es una forma estándar de medir algunos atributos de calidad del software. Ejemplos de estos atributos son: cohesión, acoplamiento, tamaño, complejidad, etc.

NFNO	Número de Funciones que No se Ocupan. Es el número de interfaces con código nulo. Sólo existen por cumplir un requisito del compilador, no por ser necesarias.
NFV	Número de Funciones Virtuales. Es el número de métodos abstractos o interfaces que contiene una clase abstracta.
PATRÓN DE DISEÑO	Es una unidad de información que captura la estructura esencial y la comprensión de una familia de soluciones exitosas probadas para un problema recurrente que ocurre dentro de cierto contexto y sistema de fuerzas.
PORTABILIDAD	Es una medida de la facilidad con que un programa determinado podrá funcionar en un ambiente de computación diferente, como una marca de computadora o un sistema operativo distinto.
REFACTORIZAR	Martin Fowler en [Fowl99] proporciona dos definiciones del término dependiendo del contexto: Como nombre o sustantivo: refactorización es “un cambio hecho a la estructura interna de software para hacerlo más fácil de entender y más barato de modificar sin cambiar su comportamiento observable”. [5] Como verbo: refactorizar es la “reestructura de software aplicando una serie de refactorizaciones sin cambiar su comportamiento observable”. [5] En la misma referencia, Fowler reúne ambas definiciones en una sola quedando el termino refactorización como: “el proceso de cambiar un sistema de software en tal forma que no se altere el comportamiento externo del código, aunque mejore su estructura interna”.
REUSABILIDAD	Es la habilidad que los productos de software tienen para su reuso en nuevas aplicaciones, de manera total o en parte. El reuso es cualquier procedimiento que produce o ayuda a producir un nuevo componente de software a partir de un

	componente ya existente sin modificar este último.
SINGLETON	Patrón de diseño ‘Singleton’ definido en el catálogo de patrones de diseño de Gamma [2]. Su intención es asegurar que sólo exista una única instancia de una determinada clase y proporciona la forma de acceder a ella globalmente.
SR2	Acrónimo de “Sistema de Reingeniería de Software Legado para Reuso”, proyecto del que este tema de tesis forma parte.
STRATEGY	Estrategia, patrón de diseño ‘Strategy’ definido en el catálogo de patrones de diseño de Gamma [2]. Su intención es definir una familia de algoritmos, encapsula cada uno y los hace intercambiables. Deja que los algoritmos varíen independientemente de los clientes que los utilizan.
TEMPLATE METHOD	Método plantilla, patrón de diseño ‘Template Method’ definido en el catálogo de patrones de diseño de Gamma [2]. Su intención es definir el esqueleto de un algoritmo en una operación, difiriendo algunos pasos a cada subclase cliente. Deja que las subclases implanten ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.
USABILIDAD	Es la medida que establece la facilidad de uso de los componentes por el usuario o desarrollador de aplicaciones.
V-DINO	Valdés - Dependencia por Interfaces que No se Ocupan. Es una métrica definida por primera vez en [6], que representa la proporción entre el número real de implementaciones de interfaces con código nulo y el máximo número posible de implementaciones de interfaces en una jerarquía de clases que inicia con una clase abstracta.

ABSTRACCIÓN INCORRECTA

Se entiende por abstracción incorrecta cuando una clase base abstracta atiende a más de una única responsabilidad, en que todas las clases heredadas deben implementar la totalidad de responsabilidades definidas en su clase base. Esto puede tener como efecto que las clases derivadas presenten incoherencias funcionales, y/o que no todas las clases derivadas atienden a todas las responsabilidades, de modo que algunas funciones quedarán vacías. Además, que la clase base con este problema exhibe un gran número de clases hijas, lo cual puede producir un desbalance entre la métrica NOC y la métrica DIT. Un número relativamente grande de responsabilidades es causa de una abstracción incorrecta de la clase base y puede influir en un uso incorrecto de la herencia.

AUTO-SUFICIENCIA

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

Capítulo 1) INTRODUCCIÓN

Producir software orientado a objetos demanda del ser humano una gran capacidad de imaginación, abstracción y creatividad, para plantear una correcta solución 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, suele producir sistemas que exhiben características de *"código desagradable"* (smell code) [46]. El *"código desagradable"* consiste de ciertas estructuras que violan principios fundamentales e impactan negativamente en algunas dimensiones de calidad de su diseño, tales como la modularidad, autonomía, fragilidad, flexibilidad, extensibilidad y movilidad, lo que dificulta su mantenimiento e impide su reuso.

Un caso particular de "*código desagradable",* que atiende esta tesis, es aquel con *abstracciones incorrectas* que, en esta tesis, se entiende como aquellas clases que atienden a más de una responsabilidad, lo que contradice al *"principio de diseño de única responsabilidad"*. En este sentido, abstracciones incorrectas exhiben incoherencias en sus partes o módulos y producen un excesivo número (mayor de 4) de clases hijas.

Otro caso de *"código desagradable"*, que atiende esta tesis, es aquel que no es conforme con el *"principio de diseño de sustitución"* debido a que contiene

interfaces (funciones abstractas) cuya implementación es nula en algunas clases derivadas. El efecto es que los clientes no obtienen la funcionalidad o el resultado esperado, por la ejecución de estas funciones nulas, lo cual viola las poscondiciones de la clase base. Adicionalmente se viola el *"principio de abierto-cerrado"*, cuando se requiere de extender el código a nuevas funcionalidades debido a nuevos requerimientos, porque obliga a modificar la definición actual de las clases que conforman la estructura arquitectural.

El "*principio de única responsabilidad"* establece que *"no debe existir más que una razón para que una clase cambie".* Por tanto, si una clase tiene más de una responsabilidad, puede tener varios motivos por los que podría cambiar, por consiguiente, deberían dividirse las responsabilidades, de manera que cada una de ellas tuviera una única responsabilidad. La aplicación de este principio incide directamente en dos aspectos vinculados al diseño de aplicaciones: La cohesión y el acoplamiento [46].

El enunciado del "*principio de sustitución (LSP)"*. es: "Si para cada objeto o1 de tipo S hay un objeto o2 de tipo T, tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o1 es sustituido por o2, entonces S es un subtipo de T" [47]. Existe una relación estrecha entre LSP y el Diseño por Contratos (Design by Contract DbC) expuesto por Bertrand Meyer. El diseño por contratos establece que en los métodos se declaran las precondiciones y las poscondiciones; las precondiciones deben ser cierta antes de ejecutar el método y tras de su ejecución, mientras que el propio método debe garantizar que las poscondiciones se cumplan.

El "*principio de abierto-cerrado"* establece que las entidades de software deben ser cerradas a modificaciones y abiertas a las extensiones. De esta manera este principio habilita la extensibilidad de nuevas funciones o comportamientos funcionales sin modificar la definición actual del código y el diseño estructural de las arquitecturas de software [48].

El *"principio de separación de interfaces"* establece que los clientes no deben ser forzados a depender de interfaces que no utilizan, es decir, que las clases que implementen una interfaz o una clase abstracta no deberían estar obligadas a tener partes que no van a utilizar [46].

Estos problemas se manifiestan debido al mal uso de la herencia de interfaces y por la carencia de un efectivo encapsulamiento. El encapsulamiento está en función del grado de relación que tienen las funciones dentro de los módulos o unidades de software (tales como clases) para atender una única responsabilidad o meta de valor para un usuario, así mismo entre las relaciones de herencia entre clases o módulos que implementan las responsabilidades de las clases base.

En esta tesis se formuló e implementó un método de refactorización que localiza las estructuras del código desagradable descrito y automáticamente corrige estas estructuras. El método aplica el *"principio de separación de interfaces"* para dividir las responsabilidades, anulando las implementaciones nulas y balancea las jerarquías de herencia en lo vertical (clases descendientes) y horizontal (clases derivadas hermanas).

Este método de refactorización ayuda a limpiar esta clase de código desagradable en sistemas existentes de software que potencialmente podrían

presentar defectos de calidad. En la solución se considera que el programa a corregir posee cierto comportamiento, y que al realizar la refactorización se debe preservar dicho comportamiento.

Adicionalmente, para efectos de prueba, se plantea la utilización de métricas para la medición de las dimensiones que se pretende mejorar que son el número de clases hermanas (NOC), el grado de profundidad de herencia (DIT), dimensiones que contribuyen en la mejora de la flexibilidad y de la extensibilidad.

La métrica V-Dino mide el problema de dependencia de interfaces por contar con interfaces que no se utilizan. Ésta se basa en la métrica denominada C-NOC (Number Of Children), que cuenta el número de clases derivadas directamente de una clase padre; así como del conteo de las funciones abstractas y de las implementaciones vacías; aplicándose a cada una de las clases abstractas de la arquitectura. El valor óptimo es 0, lo que significa que el problema no existe en ese árbol, y 1 es el peor caso (inalcanzable en casos prácticos), lo que significa que la totalidad de interfaces no son necesarias.

El concepto de número de hijos de una clase base (NOC) es una medida que indica una mala abstracción de la clase base, las cuáles proliferan en número cuando la clase base atiende a más de una única responsabilidad o meta de valor para el usuario; en tales casos es necesario balancear tanto verticalmente como horizontalmente la jerarquía de clases, dividiendo las responsabilidades de las clases base entre varias clases intermedias, con lo cual se incrementa la profundidad de la herencia (DIT), pero se disminuye el número de clases hijas (NOC) en los árboles jerárquicos de arquitecturas orientadas a objetos.

Organización de este documento de tesis:

Para describir el trabajo que se realizó para sustentar, diseñar, para la construcción de la herramienta de Re-factorización, como también para el diseño e implementación de la métrica de Coherencia de esta tesis a continuación, se detallara la organización del documento.

Dentro del capítulo 2, en la sección 2.1, se da un resumen de los trabajos relacionados más representativos que se enfocan en la misma línea de este trabajo de tesis, desde diferentes perspectivas y con diferentes criterios de refactorización.

En la sección 2.2 se da una justificación del estudio de la refactorización de MAOO con el problema de dependencia de interfaces y desde el punto de vista de la necesidad de contar con una métrica de coherencia que mida dicho problema.

En la sección 2.3 se plantea el problema de dependencia de interfaces por interfaces que no se utilizan, y se explican las consecuencias de este problema, además se menciona la estrategia o enfoque propuesta e implementada para resolverlo.

En la sección 2.4 se define el objetivo a perseguir en la solución al problema planteado de este trabajo de tesis.

Por ultimo en la sección 2.5 se mencionan los alcances y limitaciones que conlleva esta tesis.

En el capítulo 3 se exponen, en general, el marco teórico en los que se apoya la solución planteada en esta tesis.

Se expone un resumen de la programación orientada a objetos y los marcos de aplicaciones orientados a objetos, los principios del diseño orientado a objetos y de refactorización, se ofrece también el fondo matemático sobre las métricas orientadas a objetos en general y las bases de la métrica diseñada, y por último sobre patrones de diseño y se especifican las definiciones de los patrones de diseño que en esta tesis son considerados muy importantes, los patrones de diseño 'Template Method' y 'Strategy', que fueron empleados en el método de refactorización, incluyendo también la descripción de los patrones 'Command' y 'Singleton' que fueron utilizados en la elaboración de la interfaz de la herramienta.

En el capítulo 4, en la sección 4.1, se explica el método de refactorización, basado en el principio de Separación de Interfaces, a base de un ejemplo paso a paso.

En la sección 4.2 se explica el algoritmo de solución en "Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces" [19]. Para resolver el problema planteado en esta tesis, denominado Separación de Interfaces.

En el capítulo 5 se explica todo el ciclo de vida del sistema creado para implementar el método de refactorización y la métrica. Este sistema cuenta con características para contener otros métodos y métricas.

En el capítulo 6 se describe un conjunto de documentos de prueba que están asociados con los aspectos dinámicos de las pruebas del sistema. Aquí se da una descripción de cómo se efectuó la investigación experimental. Se distinguen las características de los casos de estudio, el diseño, y la definición de los casos de estudio, para la prueba del proceso de análisis de código fuente, del cálculo de la métrica, y del proceso de refactorización, así como los resultados obtenidos como producto del análisis de los datos de salida del proceso de evaluación

En el capítulo 7, se derivan las conclusiones a que se llega después de analizar, diseñar y probar el sistema. Aquí se realiza un análisis de las implicaciones del estudio, y si se cumplieron parcial o totalmente los objetivos planteados en la propuesta del proyecto de tesis.

Como un punto final se da un resumen de recomendaciones para otras investigaciones en la misma dirección de este proyecto de tesis y para complementar el sistema SR2 con más métodos de refactorización, métricas orientadas a objetos y tecnología de Servicios Web.

Capítulo 2) ANTECEDENTES

2.1 Estado del Arte y Trabajos Relacionados.

La tesis que se desarrollo es parte de un proyecto que se denominó SR2: Reingeniería de Software Legado para Reuso, propuesto por el área de Ingeniería de Software del Tecnológico Nacional de México / CENIDET. [7]

El enfoque que se ha dado a este proyecto es la obtención de marcos de aplicaciones orientados a objetos a partir de la reingeniería de software legado escrito en lenguaje Java, lo cual permite ampliar el tiempo de vida útil de este software y también como obtener más reuso del código.

El SR2 consiste básicamente en plantear un modelo que contempla un proceso de reingeniería de software legado de tres etapas para la creación y mantenimiento automatizado de MAOO [8].

La primera etapa consiste en el análisis estático de código fuente para recabar información y poder reestructurar el código legado. La segunda consiste en la reestructura del código fuente original hacia un MAOO's, escritos en lenguaje Java
mejorando su diseño y obteniéndose arquitecturas reusables. Por último, la tercera etapa consiste de un sistema de refactorización, para el refinamiento sucesivo de los MAOO's obtenidos desde el proceso de reestructura [8]. El modelo de este proyecto se muestra en la Figura 1.

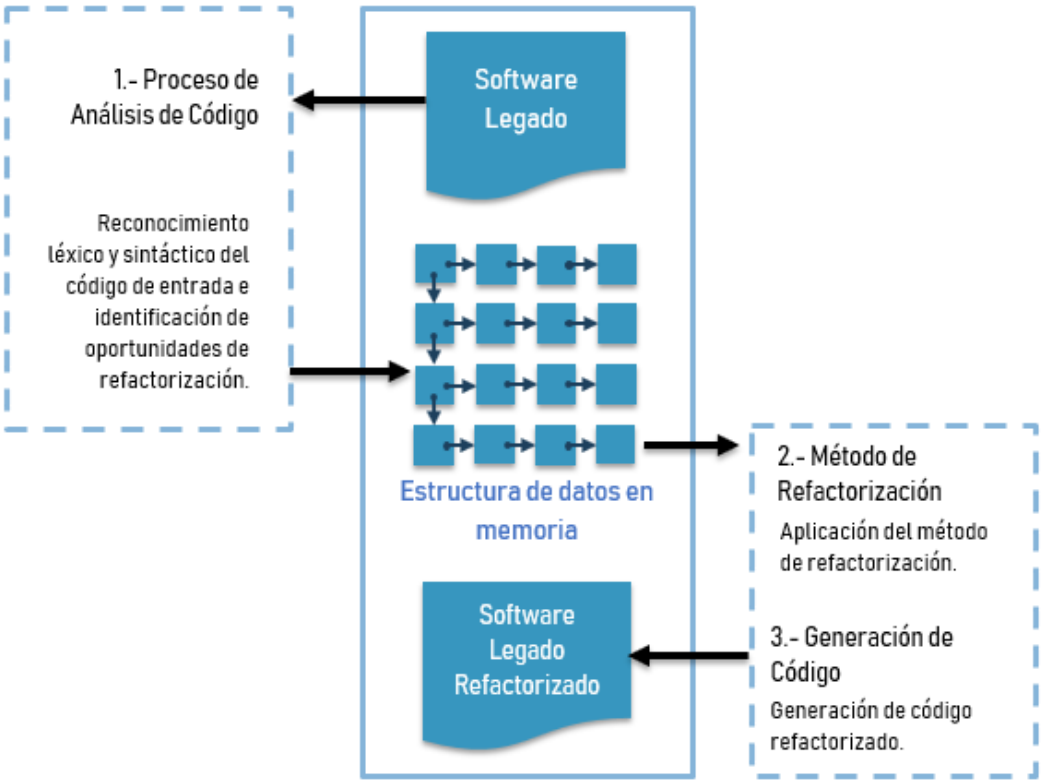


Figura 1 Modelo del SR2: Reingeniería de Software Legado para Reusó

Para cumplir las tres etapas del proyecto SR2 se han realizado y se continúan realizando un conjunto de tesis desarrolladas en el CENIDET, tanto de maestría así como, de doctorado. La tesis que se propone en este documento forma parte de la última etapa del modelo que presenta el SR2.

A continuación, se mencionan las tesis que han sido derivadas de este proyecto hasta la fecha, incluyendo la que se propone aquí, indicando quién la realiza, de qué institución es, y a qué grado pertenece.

1. “Modelo de Representación de Patrones de Código para la Construcción de Componentes Reusables”. René Santaolaya Salgado, CIC-IPN, tesis de doctorado ya terminada. [7]
2. ” Identificación de Funciones Recurrentes en Software Legado”. Anel Sheydi Zamudio López, CENIDET, tesis de maestría ya terminada.[9]
3. “Concepción de un Modelo para el Aseguramiento de Calidad de Componentes Reusables de Software”. Javier Santa Olalla Salgado, CIC-IPN, tesis de maestría ya terminada. [10]
4. ” Sistema de Pruebas de Calidad de Componentes Reusables”. Blanca R. Olascoaga Vergara, CIC-IPN, tesis de maestría terminada. [11]

5. ”IPADIC++: Sistema para Identificación de Patrones de Diseño en Código C++”. Agustín F. Castro Espinoza, CENIDET, tesis de maestría ya terminada. [12]
6. “Factorización de Funciones en Métodos de Plantilla”. Laura Alicia Hernández Moreno, CENIDET, tesis de maestría ya terminada. [13]
7. “Reconocimiento de Patrones de Diseño de Gamma a partir de la Forma Canónica definida en el IPADIC++”. Patricia Zavaleta Carrillo, CENIDET, tesis de maestría ya terminada. [14]
8. “Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator”. Leonor Adriana Cárdenas Robledo, CENIDET, tesis de maestría ya terminada. [15]
9. “Reestructuración de Software Escrito por Procedimientos Conducido por Patrones de Diseño Composicionales”. Armando Méndez Morales, CENIDET, tesis de maestría ya terminada. [16]
10. “Integración de la Funcionalidad de Frameworks Orientados a Objetos”. Juan José Rodríguez Gutiérrez, CENIDET, tesis de maestría ya terminada. [17]
11. “Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, Utilizando el Patrón de Diseño Adapter”. Luis Esteban Santos Castillo, CENIDET, tesis de maestría ya terminada. [18]
12. “Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces”, Manuel Alejandro Valdés Marrero, CENIDET,tesis de maestría ya terminada. [19]
13. “Refactorización de Sistemas Legados de Software, para equilibrar la Coherencia, Cohesión y el Factor de Acoplamiento de su estructura interna” Sandro Geovani Vázquez Díaz, CENIDET, tesis de maestría ya terminada. [20]
14. Método de Re-factorización de código java con interfaces y abstracciones incorrectas. Pablo Padilla Salgado, CENIDET, la presente tesis de maestría.

La refactorización es el proceso de cambiar un sistema de software para mejorar su estructura interna de tal manera que no se altere el comportamiento externo del código [5].

La refactorización no cambia el comportamiento de un programa, esto es, si el programa es ejecutado dos veces (antes y después de la refactorización) con la misma entrada, la salida será la misma. Las refactorizaciones preservan el comportamiento para que, cuando las precondiciones de la refactorización sean cumplidas, no hagan que falle el programa [21].

A continuación, se mencionan algunos trabajos sobre refactorización de MAOO’s aplicando patrones de diseño creacionales, estructurales y de comportamiento en la arquitectura de MAOO’s, que resuelven problemas similares a este trabajo de tesis, pero con enfoques diferentes.

Identification and Application of Extract Class Refactorings in object-oriented systems. [22]
El objetivo de esta investigación presentada es reconocer oportunidades para simplificar clases grandes, complejas, difíciles de manejar y menos cohesivas utilizando la técnica “Extract Class"

En la investigación se desarrolló una herramienta automática que Refactoriza extractos de clases, mediante la aplicación de un algoritmo de agrupamiento jerárquico. Utilizan la distancia Jaccard como la métrica de distancia. En este trabajo, se desarrolló un método y una herramienta, implementados como un plugin de Eclipse, para calcular la distancia entre los atributos y métodos de una clase para comparar la similitud de sus conjuntos de entidades. El método descrito consta de tres pasos: (a) reconocimiento de las oportunidades de extracción de clases, (b) clasificación de las oportunidades identificadas en términos de la mejora que se espera que produzca cada una en el diseño del sistema, y (c) aplicación totalmente automatizada de la refactorización elegido por el desarrollador.

El primer paso se basa en un algoritmo de agrupación aglomerativa, que identifica conjuntos cohesivos de miembros dentro de las clases del sistema. El segundo paso se basa en la métrica de ubicación de la entidad como una medida de la calidad del diseño. A través de un conjunto de experimentos, se ha demostrado que la herramienta puede identificar y extraer nuevas clases que los desarrolladores reconocen como "conceptos coherentes" y mejorar la calidad del diseño del sistema subyacente.

Manual Refactoring Changes with Automated Refactoring Validation. [23]

El objetivo de la investigación presentada es detectar refactorizaciones realizadas manualmente y comprobar automáticamente su corrección.

En la investigación se desarrolló una técnica automática de análisis estático llamada GhostFactor que separa la transformación de la verificación de corrección: permite al desarrollador transformar el código manualmente, y automáticamente comprueba la corrección de su transformación. Esta técnica fue implementada en un plug-in de código abierto para visual Studio. Este complemento, también llamado Ghost-Factor, notifica de inmediato a los desarrolladores cuando Re-factorizan incorrectamente y sugiere maneras de corregir el error.

Automated refactoring of super-class method invocations to the Template Method design pattern. [24]
El objetivo de la investigación presentada se centra en la refactorización automatizada de las instancias del patrón de código “Call Super”, cambiando su implementación hacia la estructura del patrón diseño *´Template Method´* para sustituir la herencia de implementación por la herencia de la interfaz.

"Call Super" es un patrón de código que emplea la herencia de implementación para extender el comportamiento de un método concreto. En "Call Super" el método predominante remplaza un fragmento de código por la palabra clave "super". En la clase base se ubica este fragmento de código en un método, el cual es invocado desde el método predominante a través de la palabra reservada "super". Este trabajo estudia una implementación típica del patrón de código "Call Super", donde el método sobre escrito declara una sentencia única de SuperInvocation en su cuerpo.
Los autores usan el término "Call Super" para referirse a tales instancias del patrón de código.

Los autores proponen un algoritmo para identificar patrones de llamadas directas desde clases derivadas a su clase base "Call Super" que son candidatas a re-factorización, y un método que automatiza la re-factorización de estos patrones de llamadas hacia llamadas inversas utilizando el patrón de diseño "Template Method".

El algoritmo de identificación de re-factorización recibe como entrada el código de un programa y genera un conjunto de instancias R de "Call Super"que son candidatas para Re-factorizar hacia el Método de plantilla. Cada candidato de re-factorización está representado por una instancia de la clase RefactoringCandidate.

La identificación de candidatos es un procedimiento de dos etapas que se describe formalmente en un algoritmo. La primera etapa de la operación del algoritmo, produce un conjunto inicial de oportunidades de re-factorización analizando cada clase individualmente. La segunda etapa del proceso en el algoritmo implica un análisis adicional de candidatos a re-factorización que pertenecen a la misma jerarquía de clases.

Como resultado obtenido en este desarrollo están:
- Un Método de re-factorización
- Un algoritmo que implementa el método de re-factorización
- Un plug-in para integrarla al algoritmo en eclipse. Este plug-in se denomina JDeodorant

Refactoring Sequential Java Code for Concurrency via Concurrent Libraries[25]

El objetivo de la investigación presentada es remplazar procesos secuenciales por procesos paralelos.

En la investigación se desarrolló una herramienta llamada CONCURRENCER, que permite a los programadores Re-factorizar el código secuencial en un código paralelo que usa tres java.util.concurrent (j.u.c.) servicios concurrentes.
CONCURRENCER no requiere ninguna anotación de programa. Sus

transformaciones abarcan múltiples declaraciones de programa no adyacentes. Una herramienta de búsqueda y reemplazo no puede realizar tales transformaciones, lo que requiere un análisis de programa. La evaluación empírica muestra que los refactores CONCURRENCER codifican de manera efectiva: CONCURRENCER identifica y aplica correctamente las transformaciones que algunos desarrolladores de código abierto pasaron por alto, y el código convertido muestra una aceleración en su rendimiento.

En este artículo los autores proponen CONCURRENCER, que automatiza tres refactorizaciones para convertir campos enteros en AtomicInteger, para convertir mapas hash a ConcurrentHashMap, y para paralelizar parallelizing divideand-conquer algorithms.

La experiencia que se presentó con CONCURRENCER muestra que es más eficaz que un desarrollador humano en la identificación y aplicación de dichas transformaciones, y el código paralelo muestra una buena aceleración.

A systematic review on search-based Re-factoring [26]

El objetivo de esta investigación presentada es proporcionar una visión general de los enfoques de Re-factorización basada en búsqueda (SBR) existentes, presentando sus características comunes e identificar tendencias y oportunidades de investigación.

Los autores de este artículo presentan las características comunes de todos los enfoques analizados e identifica tendencias y oportunidades de investigación. De esta manera, se muestra la mejor secuencia de re-factorizaciones que se aplica en un artefacto de software.

Se realizó una revisión sistemática siguiendo un plan que incluye la definición de preguntas de investigación, criterios de selección, una cadena de búsqueda y la selección de motores de búsqueda. Se seleccionaron 71 estudios primarios, publicados en los últimos dieciséis años. Se clasificaron considerando las dimensiones relacionadas con los principales elementos SBR, tales como los artefactos dirigidos, la codificación, la técnica de búsqueda, las métricas utilizadas, las herramientas disponibles y la evaluación realizada.

Como resultado de este artículo se obtuvo un reporte de las características de 71 documentos de refactorización de software.

Automated refactoring of legacy Java software to enumerated types[27]

El objetivo de la investigación presenta un enfoque de preservación de la semántica aumenta la seguridad del tipo, produce un código que es más fácil de comprender, elimina la complejidad innecesaria y elimina los problemas de fragilidad debido a la composición por separado.

En la investigación se desarrolló un algoritmo de inferencia de tipo inter-procedural que rastrea el flujo de valores enumerados. El algoritmo se implementó como un plug-in de Eclipse de código abierto, disponible para el público y se evaluó experimentalmente en 17 grandes puntos de referencia de Java.

Como resultado de este artículo le indica que el costo del análisis es práctico y el algoritmo puede Re-factorizar con éxito una cantidad sustancial de campos a los tipos enumerados. Este trabajo es un paso importante para proporcionar soporte automatizado de herramientas para migrar el software heredado de Java a las tecnologías Java modernas.

Model-Driven Java Code Refactoring [28]

El objetivo de la investigación presenta un enfoque basado en modelos donde las funciones de refactorización, como la representación de códigos, el análisis y la transformación, adoptan modelos como artefactos de primera clase. Se va explorar el valor de la transformación del modelo y la generación de código al formalizar refactorizaciones y desarrollar herramientas de soporte.

El enfoque presentado se aplica a la refactorización del código de Java utilizando una implementación prototípica basada en Eclipse Modeling Framework, un banco de trabajo de lenguaje, un metamodelo de Java y un conjunto de estándares de OMG.

Automated refactoring to the "*NULL OBJECT*" design pattern [29]

El objetivo de la investigación es hacer un nuevo método para la refactorización automatizada a "*NULL OBJECT*" que elimina los condicionales de verificación nula asociados con los campos de clase opcionales. Es decir, campos que no se inicializan en todas las instancias de clase y, por lo tanto, su uso debe ser protegido para evitar referencias nulas.

En la investigación se desarrolló un algoritmo para el descubrimiento automático de oportunidades de refactorización para "*NULL OBJECT*". Además, especifican el procedimiento de transformación del código fuente y un amplio conjunto de condiciones previas de actualización para Re-factorizar de forma segura un campo opcional y sus condiciones de verificación nula asociadas al patrón de diseño "*NULL OBJECT*". El método se implementa como un complemento de Eclipse y se evalúa en un conjunto de proyectos de código abierto de Java.

CRat: A refactoring support tool for Form Template Method [30]

El objetivo de la investigación es detectar y sugerir automáticamente los métodos candidatos que requieren ser Re-factorizados.
En la investigación se desarrolló una herramienta llamada CRat, la cual requiere como entrada, el código fuente de los sistemas de software como entrada y mediante un análisis del código fuente crear Programa Gráfico de dependencia

(PDG'S) con el uso de las métricas propuestas, para detectar candidatos de refactorización. Así mismo detecta clones de código en PDGs con el detector de clones existente. "*CRat*" identifica a los candidatos de refactorización con información sobre clones de código y detecta procesos comunes y únicos para cada candidato de refactorización. Visualiza cada candidato resaltando procesos comunes y únicos. Cabe resaltar que "*CRat*" no modifica el código fuente automáticamente, si no que los usuarios deben modificar sin que, el código fuente por sí mismos. "*CRat*" sugiere un par de métodos como un candidato de refactorización si cumplen los siguientes requisitos:

- Los dos métodos se definen en diferentes clases
- Los dos métodos tienen la misma clase base
- Existe al menos un par de clones (un par de fragmentos de código duplicados) entre los dos métodos.

"*CRat*" tiene una función de filtrado basada en métricas de pares de métodos candidatos. Todas las métricas se calculan para cada par de métodos. Los usuarios pueden especificar los umbrales superior e inferior para cada métrica y, a continuación, "*CRat*" sugiere candidatos cuyas métricas se encuentran en los umbrales especificados. Las métricas son las siguientes:

- **SIM**: El grado de similitud entre los dos métodos.
- **CN**: El número de instrucciones que se pueden extraer en la clase base.
- **DN**: El número de sentencias que deben permanecer en cada clase derivada.
- **LOC**: El número de líneas de código.
- **DG**: El número de nuevos métodos que se deben crear para la aplicación del Tamplate Method.
- **DOI**: La profundidad de herencia de la clase base común a las clases propietarias de los dos métodos.

J. Kerievsky
Industrial Logic, Inc., USA. [31]

Este trabajo es un libro en progreso apoyado por Industrial Logic Inc., en donde se explora la relación que existe entre la refactorización y los patrones, y detalla varias refactorizaciones de manera manual que muestran cómo llegar a los patrones de diseño siguiendo un algoritmo. Utiliza los patrones de diseño del catálogo de Gamma: 'Singleton', 'Factory Method', 'Strategy', 'Composite', 'Builder', 'Decorator', 'Proxy', 'Observer', 'Adapter', 'State', 'Template Method', 'Visitor' y 'Flyweight'.

Este trabajo si contempla el problema de dependencia de interfaces por tener interfaces que no se utilizan, pero intentan solucionar de una manera diferente, utilizando el patrón 'Adapter'.

Una evaluación de su método de solución muestra que no resuelven completamente el problema, pero si lo aligeran en las clases derivadas, ya que suben el problema de implementaciones vacías a un adaptador que colocan antes de la clase base original.

La publicación que trata sobre este trabajo es:
1. “Refactoring to Patterns” [31]

La diferencia que existe entre esta investigación y lo propuesto en esta tesis es que se toma un enfoque diferente para resolver el problema. Aunque su método de refactorización tiene menos precondiciones que el propuesto aquí, no es efectivo, ya que sólo disfraza el problema de las implementaciones vacías, pero no lo elimina. Además, sólo proponen un algoritmo y la forma manual de resolución, pero no hay intenciones de construir una herramienta que haga la refactorización automática.

En la Tabla 1 se muestra un resumen comparativo entre todos los grupos de trabajo y herramientas presentadas anteriormente, donde se muestra si la herramienta trabaja de manera automatizada, qué lenguaje es el que refactoriza, si implementa patrones de diseño y cuántos, y si utiliza los patrones de diseño ‘Strategy’ y ‘Template Method’.

Herramienta	**Automática**	**Lenguaje**	**Patrones Diseño**	**Strategy y Template Method**	**Separación de Interfaces**
Identification and Application of Extract Class	Si	Java	No	No	No
Manual Refactoring Changes with Automated Refactoring Validation	Semi-Automática	Java	No	No	No
Automated refactoring of super-class method invocations to the Template Method design pattern	Si	Java	Si(1)	Si	No
Refactoring Sequential Java Code	Si	Java	No	No	No
A systematic review on search-based Re-factoring	No	Java	No	No	No
Automated refactoring of legacy Java software	Semi-Automática	Java	No	No	No
Model-Driven	Si	Java	No	No	No
Automated refactoring “NULL OBJECT” design pattern	Si	Java	Si(1)	No	No
Crat	Si	Java	Si(1)	Si	No

Kerievsky	No	Java	Si(13)	Si	Si
Esta tesis	Sí	Java	Sí (2)	Sí	Sí

Tabla 1 Comparación de Herramientas

2.2 Justificación

Específicamente, se requiere mejorar la generalidad, modularidad, flexibilidad y complejidad de arquitecturas de clases de objetos que exhiben en su estructura alto grado de acoplamiento y encapsulamiento incorrecto, que no satisfacen los principios del modelo orientado a objetos de abierto-cerrado y de única responsabilidad, lo cual impacta en su nivel de suficiencia y completitud para alcanzar sus metas u objetivos; mediante su re-factorización arquitectural.

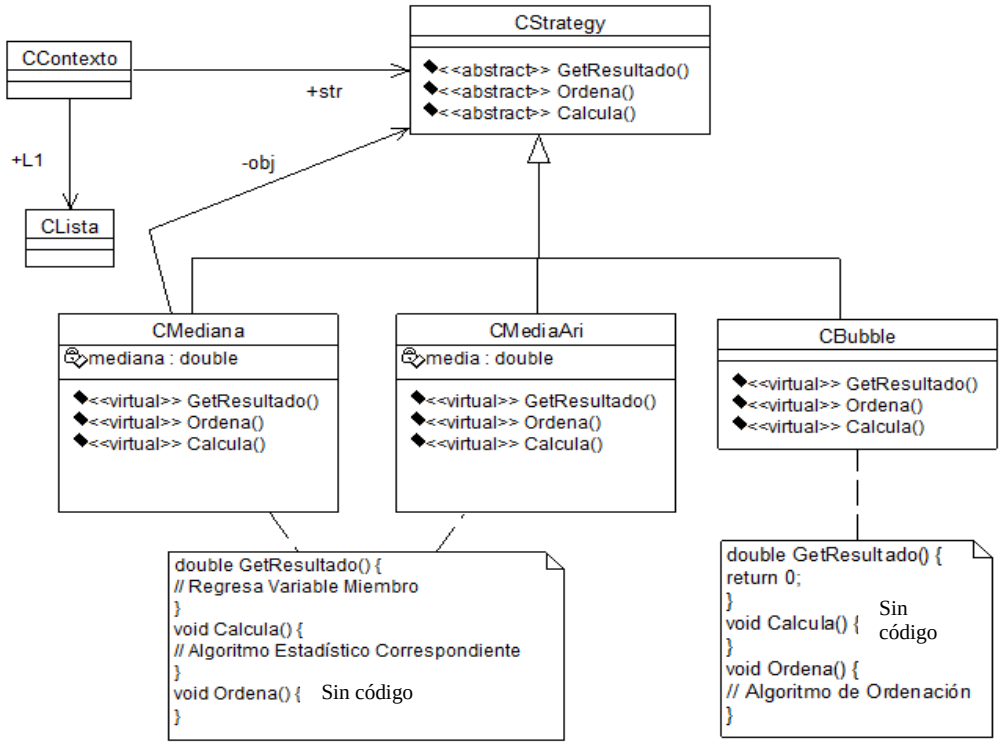
Se espera que el sistema o ambiente de modelado desarrollado, sea capaz de mejorar la abstracción y el encapsulamiento modular y el nivel de reuso de diseños de arquitecturas de software con problemas de flexibilidad y autosuficiencia.

2.3 Planteamiento del problema

El problema radica en que el Software de Aplicaciones que no logra satisfacer las demandas de calidad para el control de su evolución, ni la productividad de los desarrolladores por el reuso de componentes legados. Esto es debido a que las entidades de Software Legadas no logran cumplir con las mejores especificaciones de calidad, generalidad, flexibilidad y robustez; y no llegan a satisfacer con plenitud y suficiencia las demandas funcionales requeridas por nuevas aplicaciones desarrolladas a partir de éstas, por mecanismo de integración, ensamble y/o composición.

Ejemplo del problema de Dependencia de Interfaces

Para que este problema sea más claro, se ilustrará con una parte de un MAOO del dominio de la estadística, mostrándose su diagrama de clases bajo el estándar UML (Unified Modeling Lenguage) en la figura 2.



Nota: *Que el método o función GetResultado no está vacío en ninguna clase, no tiene que ser separado.*

Figura 2. Parte del MAOO de Estadística con Tres Operaciones

Este MAOO proporciona algunas funciones para cálculo estadístico, utilizando una lista doblemente ligada (***L1***) para almacenar la serie de números sobre los que operan las funciones. La funcionalidad total que tiene el MAOO completo es el cálculo de medidas de tendencia central, dispersiones, distribuciones, regresiones y correlaciones, y como auxiliares se tienen ordenaciones de datos y solución de sistemas de ecuaciones lineales.

La parte del MAOO que se muestra en la Figura 2 tiene las operaciones de cálculo de la operación del método de media aritmética y mediana, y como auxiliar se tiene una ordenación por burbuja. La media aritmética, implementada en "***CMediaAri"***, consiste en la suma de los elementos de la lista dividida entre el total de elementos; la mediana, implementada en la clase "***CMediana"***, consiste en el elemento central de la lista ordenada de datos, de ahí la necesidad del algoritmo de ordenación, implementado en la clase "***CBubble"***.

Para llamar a la funcionalidad del MAOO, el cliente interactúa con la clase "***CContexto"***, *y* esta clase a la vez delega sus operaciones a las clases derivadas. Utiliza las interfaces *GetResultado ()*, *Ordena ()* y *Calcula ()* de la clase *CStrategy*.

Observa que la clase, "***CBubble"*** implementa la función *Calcula( )* sin código y a *GetResultado( )* con código nulo (sólo para cumplir con el valor de retorno).

Asimismo, la clase ***"CMediaAri"*** como la clase ***"CMediana"*** implementan la función *Ordena ()* también sin código, como se muestra en la Figura 2. Aquí es donde se tiene el problema de dependencia de interfaces que en realidad no se ocupan.

El problema surge porque fueron mal planificadas las interfaces, que no son ocupadas por todas las clases concretas, y por cumplir con los requerimientos del compilador se tienen que implementar todas las funciones abstractas en las clases concretas, aunque sea vacías o con código nulo.

Si un cliente diferente no necesitara la funcionalidad de cálculo, sino sólo la de ordenación, no será posible separar sólo la parte de ordenación para esa aplicación. Esto es debido a que la clase ***"CBubble"*** tiene una interfaz *Calcula ()* que no tiene razón de existir en ese nuevo contexto.

Si se agregara otra funcionalidad diferente a ordenar y calcular, se tendría que modificar a la clase ***"CStrategy"*** para agregar otra función virtual. Además, se agregará la implementación de esta nueva función en las clases derivadas ***"CBubble"***, ***"CMediaAri"*** y ***"CMediana"*** para agregar la nueva interfaz, y a la nueva clase se le tendrían que poner las tres interfaces existentes. Esta abstracción incorrecta viola el principio de Abierto / Cerrado por lo que se pierde flexibilidad para extensiones de funcionalidad.

Con esta arquitectura se violan los principios de Diseño Orientado a Objetos de: "Única responsabilidad", "Sustitución" y de "Abierto/Cerrado".

La propiedad de reuso del MAOO se ve disminuida por el problema de inmovilidad, es decir, no se pueden llevar ciertas partes del MAOO de manera independiente hacia otros contextos. La propiedad de flexibilidad se ve disminuida por el problema de rigidez, es decir, no se puede cambiar el comportamiento de la funcionalidad del MAOO sin modificarlo, y, por supuesto, esto no es correcto ni deseable en cualquier MAOO, cuyo objetivo ideal es precisamente facilitar el reuso, la flexibilidad y la extensibilidad a nuevas operaciones.

El problema resulta grave en el MAOO completo, ya que se tienen demasiadas funciones que no se ocupan y esto puede confundir a cualquier usuario del MAOO, porque viola el principio de sustitución. Como se puede ver, este MAOO es bastante completo y ha requerido mucha inversión de tiempo y esfuerzo, y se busca entonces una manera de mejorar su diseño de manera automática.

Solución del Problema

Para resolver este problema se plantea utilizar el algoritmo creado por [19] denominado 'Separación de Interfaces', basado en el principio de diseño orientado a objetos del mismo nombre. Este principio tiene como objetivo el evitar que los clientes dependan de interfaces que en realidad no ocupan [Mart96], entendiéndose por cliente a cualquier sistema que utiliza el MAOO.

El método de refactorización propuesto para separar interfaces se basa en dos patrones de diseño propuestos por Gamma et al [2], denominados 'Strategy'

y ‘Template Method’. De estos patrones se utiliza sólo su estructura, ya que la intención de ellos no es resolver el problema de dependencia de interfaces, pero se pueden adaptar para que lo hagan.

La refactorización del MAOO consiste en establecer interfaces con nombres genéricos, que se van especializando en las clases derivadas. Es decir, se sustituyen las interfaces afines por una sola. Esta interfaz genérica tendrá su implementación en clases intermedias nuevas.

Se crean clases intermedias entre la clase base abstracta y las clases derivadas para mostrar las diferentes estrategias a seguir. Cada estrategia se considera que es una especialización del MAOO. Estas clases intermedias también deberían tener nombres genéricos y son especificados por el experto del dominio, usuario de la herramienta de refactorización.

Cada clase intermedia declara a una de las interfaces originales, y esta clase debe servir como clase base de las clases derivadas originales que si ocupan la interfaz.

La interfaz genérica es implementada con el patrón de diseño ‘Template Method’ en las clases intermedias. La implementación de este método de plantilla sólo es una llamada hacia la implementación de la interfaz de cada una de las estrategias o especializaciones, y esta interfaz ya se encuentra implementada por las clases concretas derivadas de las clases intermedias.

Como resultado, cada clase derivada tiene que implementar a una única interfaz, eliminando las interfaces que no se ocupan. Así el MAOO resultante se podrá extender sin romper el principio de abierto/cerrado.

Los beneficios principales que se obtienen con esta refactorización es que los programas que eran rígidos y frágiles por depender de interfaces que no ocupan ahora son flexibles y robustos, su mantenimiento es más fácil. Asimismo, se consigue que éstos sean más reusables.

Como resultado de la refactorización, se obtienen MAOO’s que no requieren de modificaciones cuando se quiera agregar nueva funcionalidad, respetando así el principio de Abierto / Cerrado, y las nuevas clases sólo declararán interfaces que ocupan, respetando así el principio de Sustitución de diseño orientado a objetos.

2.4 Objetivos

2.4.1 Objetivo general

El objetivo principal de esta tesis es: lograr mayor calidad de las entidades de software legado para facilitar su mantenimiento y aumentar su capacidad de reuso. Mediante la mejora de la generalidad y la flexibilidad; así como una mejor autosuficiencia y autonomía de entidades de software.

2.4.2 Objetivo específicos

- Mejorar el Diseño de Arquitecturas de Componentes de Software Existentes escritos en lenguaje Java.

- Extender la funcionalidad del “SR2-Refactoring”, para dar soporte a sus métodos de re-factorización de alto nivel en código escrito en lenguaje Java.
- El método de refactorización debe integrar las reglas sintácticas y semánticas para reconocer el problema de dependencia de interfaces, localizar las clases que intervienen en el problema, generar clases intermedias con interfaces genéricas que solucionen el problema, y finalmente refactorizar el código original, siendo la arquitectura resultante de esta refactorización correspondiente a la de los patrones de diseño ‘Strategy’ y ‘Template Method’.

2.5 Límites y Alcances del Estudio

Para resolver el problema de dependencia de interfaces se creó una herramienta de refactorización que detecta el problema y lo soluciona de manera automática solicitando pocos datos al usuario, y para ello se llevaron a cabo diferentes módulos.

El presente trabajo de tesis involucró la integración de reglas semánticas con las reglas sintácticas del lenguaje Java. Estas reglas semánticas fueron expresadas en el lenguaje Java y la especificación léxica y sintáctica de la gramática del lenguaje de aplicaciones que se explora está expresada en el metalenguaje ANTLR. Tanto las reglas léxicas, sintácticas como semánticas fueron necesarias para localizar y cuantificar el problema de dependencia de interfaces.

Con la información recabada se crean nuevas clases de estrategia para separar las interfaces, con nombres e interfaces genéricas, utilizando los patrones de diseño ‘Strategy’ y ‘Template Method’. El módulo encargado de esto es el que lleva a cabo la refactorización automática, solicitando sólo los nombres genéricos.

Finalmente se realizó un módulo para generar el código de los archivos fuente re-factorizados, copiando el código fuente original y agregando las clases genéricas de estrategia. El objetivo final es que estos nuevos archivos funcionen exactamente igual que los archivos originales, pero la arquitectura ahora es extensible y reusable.

Como valor agregado a la tesis se realizó el diseño e implementación de una métrica de Coherencia orientada a objetos capaz de detectar el problema de dependencia de interfaces descrito en esta propuesta. Esta métrica sirve para medir la cantidad de funciones que interactúan para satisfacer una responsabilidad en relación al número total de funciones en la entidad de software que los contiene.

Adicionalmente la herramienta tiene una interfaz genérica que permite que otras herramientas de refactorización sean agregadas, usando la extensión ya que se emplearon las técnicas de programación orientada a objetos para dicha interfaz empleando patrones de diseño. En este momento, en el sistema se encuentran integrados tres métodos de refactorización del SR2.

La herramienta cuenta con limitaciones, las cuales pueden ser consideradas como trabajos futuros. Primeramente, la refactorización que se lleva a cabo es a nivel de código y dicho código debe estar en el lenguaje Java con el enfoque orientado a objetos. También este código debe estar libre de errores léxicos y sintácticos, ya que de lo contrario no funcionará el código Re-factorizado como lo marca los principios de la refactorización [21].

La refactorización llevada a cabo por la herramienta sólo resuelve el problema de dependencia de interfaces, en donde los clientes son forzados a depender de interfaces que no utilizan; otro tipo de problemas de diseño, estructura o codificación que presenten los MAOO's no fueron resueltos, ya que existen o existirán otras herramientas del SR2 que se encarguen de esos otros problemas.

Capítulo 3) MARCO TEÓRICO

3.1 Marcos de Aplicaciones Orientados a Objetos (MAOO)

El reuso del software es una forma de aumentar la calidad del producto final. Si las piezas de software previamente probadas se aplican el reuso en un nuevo proyecto, es más probable que estén libres de errores que las desarrolladas recientemente, debido al uso y las pruebas repetidas. "*Esto reduce la tasa de fallas en general, por lo tanto, aumenta la calidad del artefacto del software"* [2]. La razón aquí es bastante simple: más usos permiten el descubrimiento y eliminación de más errores.

La reutilización del software ha recibido casi tantas definiciones como el número de autores que han escrito sobre esta. Para los fines de este documento, utilizaremos la definición ofrecida por el Estándar IEEE para procesos de reutilización de procesos de ciclo de vida de sistemas y sistemas de tecnología de la información, que es el estándar actual en el momento de este estudio:

"El reuso del software implica capitalizar el software y los sistemas existentes para crear nuevos productos".

En la definición anterior, la palabra "capitalizar" implica una cosecha de beneficios potenciales de la reutilización del software. Algunas actividades clave deben incluirse en el ciclo de vida del desarrollo de software **(SDLC) "Systems Development Life Cycle**" para producir nuevos sistemas. Al considerar estas

actividades, el estándar se refiere a la reutilización sistemática y lo define como "la práctica de reutilización de acuerdo con un proceso consistente y repetible".

Un componente de software reutilizable es algo que se creó con la intención de ser reutilizado, siguiendo un diseño y estructura que permita el uso del componente en varios contextos sin que tenga que ser modificado. Estos componentes tienen diferentes niveles de abstracción, que pueden ser desde una especificación de requisitos hasta una biblioteca de funciones o un MAOO que enfatiza el reuso de código y del diseño [32].

Uno de los objetivos principales de la Ingeniería de Software es el reuso, esperando que no sólo se llegue al reuso de componentes completos de código, sino que se llegue al nivel de poder reusar el diseño de sistemas, de tal manera que se puedan generar marcos de componentes reusables de diferentes ámbitos de aplicaciones y se facilite la tarea de construir nuevos sistemas para resolver nuevos problemas [3].
Los desarrollos orientados a objetos empiezan a dominar sobre los otros paradigmas de programación y son utilizados para construir software de calidad en cualquier contexto, desde aplicaciones simples de información a grandes sistemas financieros distribuidos, o desde los applets de páginas Web hasta los sistemas operativos modernos [32].

El paradigma Orientado a Objetos

El paradigma Orientado a Objetos está basado en la creación de clases que utilizan las propiedades de: Abstracción de Datos, Polimorfismo, Ocultamiento de Datos y Herencia. La clase proporciona los mecanismos de encapsulación, abstracción y ocultamiento de datos, y es el componente elemental del reuso.
La clase proporciona mecanismos de reuso en dos niveles: como representación de una abstracción de diseño, que se puede extender, y como una plantilla de objetos que comparten la estructura de datos y comportamiento definido por la clase [32].

Abstracción de Datos.

Es un modelo de software que empaqueta una estructura de datos junto con un conjunto de operaciones asociadas a ésta. Esta propiedad es la que habilita la ocultación y encapsulado de datos. En este sentido, las abstracciones están representadas por las clases, los objetos que son instancias de las clases y los mensajes o funciones de clase**.**

Polimorfismo.

Se refiere a que un objeto efectúa ciertas acciones cuando recibe un mensaje. Ante un mensaje, el método que se ejecutará no sólo depende del nombre del mensaje, sino también del tipo de objeto en particular. Esto da la facilidad de que un objeto actúe de distintas maneras dependiendo del subtipo de la clase que responda a un mensaje.

Ocultación de Datos.

Se refiere a que debe estar protegida la implementación interna de los objetos, que da los detalles de su funcionamiento. Esto significa que el usuario no necesita conocer la información oculta para usar la abstracción y, que no se le permite al usuario hacer uso o manipular la información oculta aún cuando lo desee. Es decir, los datos ocultos del objeto, únicamente deben ser accesibles para los propios métodos del objeto.

Herencia.

Es la propiedad que permite que nuevas clases puedan compartir la estructura y el comportamiento de clases existentes. Esta propiedad es la que permite la extensión del software y con ello su reuso. Mediante esta característica, se pueden construir nuevos elementos de software sobre una base de elementos existentes, para no tener que iniciar desde el principio de un sistema [32]. Asi mismo, esta característica habilita las propiedades de invalidación y reemplazo, asociación dinámica de tipos y el polimorfismo.

Las dos técnicas más comunes para reusar la funcionalidad en sistemas orientados a objetos son: la herencia de clases y la composición de objetos [2].

Herencia de Clases.

Permite definir la implementación de una clase en términos de otras. Al reuso por herencia se le conoce como “reuso de caja blanca”, debido a que el interior de las clases bases por lo general es visible para sus clases derivadas. La herencia es definida de manera estática (tiempo de compilación) y es fácil de implementar. La herencia permite modificar la implementación que se está heredando, pero, aun así, la implementación de una clase derivada es fuertemente dependiente de la implementación de su clase base. Una manera de solucionar lo anterior es heredar de clases base abstractas, ya que ellas tienen muy poco código que heredar o incluso ninguno, sino que sólo ofrecen interfaces.

Composición de Objetos.

Es una alternativa a la herencia. Con ella se consigue nueva funcionalidad ensamblando objetos para obtener una funcionalidad más compleja. La composición de objetos requiere que los objetos que están siendo compuestos tengan interfaces bien definidas. Al reuso por composición se le conoce como “reuso de caja negra”, debido a que los detalles internos de los objetos no son visibles.

La composición es definida de manera dinámica (tiempo de ejecución) a través de objetos que tienen referencias a otros objetos. La composición requiere que los objetos respeten cada una de las interfaces de otros objetos, lo que lleva a que se tengan que diseñar cuidadosamente las interfaces, para que sirvan para conectar al objeto con múltiples objetos.
Por lo general los diseños son más simples y reutilizables si dependen más de la composición de objetos que de la herencia. Debido a esto, muchos patrones de diseño fomentan el uso de composición.
Con la herencia y la composición se pueden crear dos tipos principales de componentes reusables orientados a objetos: las jerarquías de herencia y los MAOO basados en dominios de aplicaciones [32].

Jerarquías de Herencia.

Son un conjunto de clases asociadas por herencia de implementaciones, de interfaz y/o de clase, donde existen clases base y clases derivadas o subclases. Una clase abstracta tiene como objetivo ser una plantilla para subclases más específicas, y por lo general no tienen instancias y no están completamente definidas, ya que les falta la implementación de métodos que será realizada en las subclases, es decir que, las clases abstractas tienen métodos o funciones virtuales, o funciones abstractas (interfaces).

La jerarquía de herencia

Presenta el problema de dependencias entre las subclases con sus clases base, lo que las hace poco reusables, presentan el problema de romper el encapsulamiento de datos y aumentan su complejidad.

Marcos de Aplicaciones Orientados a Objetos ó Frameworks*.*

Son un conjunto de clases en colaboración que incorporan un diseño genérico y pueden ser adaptados a problemas específicos. Estos MAOO's están dirigidos a unidades de negocios y/o dominios en particular, de forma que las clases del MAOO no pueden salir de contextos del ámbito de su dominio. Los MAOO's enfatizan el reuso del diseño o las arquitecturas que dan solución a problemas recurrentes del dominio de aplicaciones. Un MAOO debe dar facilidades para que se extienda su funcionalidad, ya que un dominio siempre está en constante maduración y se desea que los MAOO's, mientras vayan evolucionando, lo cubran por completo [Sant02].
La instanciación de un MAOO es el ensamble de objetos que se necesite que trabajen juntos para solucionar un problema en particular. Un MAOO debe ocultar las partes de diseño que son comunes a todas las instancias, y dejar accesibles las partes que deben ser especializadas. Al reusar un MAOO sólo se deben realizar dos cosas: definir las nuevas clases que se necesitan y configurar un conjunto de objetos existentes.
Los MAOO son generalmente aplicaciones como interfaces de usuario, o ambientes de trabajo multimedia.
Los MAOO traen múltiples beneficios, como son: la modularidad, el reuso, la extensión y la inversión de control [7].
• La modularidad se logra por el encapsulado de los detalles de implementación, y

se evitan los cambios al MAOO.
• El reuso se debe a las interfaces provistas por el MAOO, que son muy estables y permiten extender al MAOO para crear nuevas funcionalidades.
• La extensión se mejora ya que el MAOO cuenta con métodos gancho que permiten extender la funcionalidad y desacoplar las interfaces de los métodos e invertir el control de los procesos.
• La inversión de control se refiere a que el MAOO toma el control de la aplicación y él decide qué métodos invocar de acuerdo a los mensajes que llegan del exterior.
Como se puede observar, el principal objetivo de un MAOO es permitir el reuso de su diseño y esto se logra utilizando la herencia y la instanciación. Otro objetivo de un MAOO es facilitar la extensión de su funcionalidad, con el fin de cumplir con ciertos requerimientos necesarios para un problema particular del dominio; esto también es logrado utilizando la herencia, agregando clases concretas que se conectan a las clases abstractas, las cuales ofrecen las interfaces del MAOO. Sin embargo, no siempre es posible lograr el reuso de software en los MAOO, debido a que algunos MAOO no maduros presentan problemas de diseño, y ello impide que sus objetos sean reusados de manera independiente, porque no fueron bien diseñados o por haber sido obtenidos fríamente por procesos de reingeniería automatizados. Uno de estos problemas es el de dependencia de interfaces debido a interfaces no utilizadas, al cual se enfoca esta tesis.

3.2 Principios de Diseño Orientado a Objetos

Para lograr el reuso en un ambiente orientado a objetos, el software debe cumplir con los principios de diseño orientado a objetos.

Existen varios principios del diseño orientado a objetos, sobre los que está soportada la programación orientada a objetos, de los cuales dos se relacionan directamente a esta tesis. Estos son: El Principio de Abierto / Cerrado y sustitución.

El Principio de Abierto / Cerrado surgió debido al pensamiento que “todos los sistemas cambian durante su ciclo de vida. Esto debe tenerse en mente cuando se pretende que los sistemas desarrollados perduren más de la primera versión” [33]. Esto quiere decir que se deben prever extensiones de flexibilidad y cambios de comportamiento de un sistema original, y se debe planear cómo hacer estas extensiones y flexibilidades.

El Principio enuncia: “Las entidades de software (clases, módulos, funciones.) deben estar abiertas para su extensión, pero cerradas para su modificación” [34].

Los beneficios que se obtienen de este principio es el diseño de módulos cuyo comportamiento puede ser alterado o adaptado para nuevos usos, pero sin realizar modificaciones al código fuente de estos módulos. Con esto se da soporte a la facilidad de mantenimiento, de reuso y de verificación del software.

Cuando un sistema no acata este principio, un cambio que se realice en alguna parte del sistema afecta a una gran cantidad de módulos dependientes, que deben ser modificados, esto trae como consecuencia que el sistema sea frágil, rígido, impredecible y por lo tanto no reusable. Por eso el principio nos dice que cuando se cambien los requisitos o se agreguen nuevos requisitos, se cambia o la extiende el comportamiento de los módulos añadiendo nuevo código mediante herencia, pero nunca se cambia el código que ya existe y se sabe que funciona [32].

Se dice que un sistema es rígido porque cada cambio afecta a muchos otras partes del sistema, es frágil porque cuando se efectúan cambios, se ocasionan fallas en partes inesperadas del sistema, y es difícil su reuso en otras aplicaciones debido a que los módulos del sistema no pueden ser fácilmente desintegrados de la aplicación actual [7].

Para lograr el objetivo del Principio de Abierto / Cerrado se utiliza la abstracción, ya que se crean abstracciones o clases base que son fijas y representan un grupo ilimitado de posibles comportamientos, y estos comportamientos se dan gracias a la herencia con clases derivadas y el polimorfismo. De esta manera al reusar las clases abstractas se cierra el código a modificaciones y se extiende el código con nuevas clases derivadas [32].

3.3. Refactorización

Re-factorizar es el proceso de cambiar un sistema de software de tal que mejore su estructura interna pero no se altere el comportamiento externo del código. Esta es una forma disciplinada de ordenar el código para minimizar las oportunidades de introducir errores.

Para asegurarse que esto no ocurra, una refactorización siempre cuenta con precondiciones y pos condiciones que se deben de cumplir antes y después de aplicar una transformación. Por tanto, la refactorización no ayuda a un sistema actual sino a sistemas futuros que se extiendan del sistema actual [21].

Una herramienta que ofrece refactorización automatizada debe garantizar que sus operaciones preservan el comportamiento del sistema de software original. Una de las razones porque los programas no son Re-factorizados es debido a que al hacer un cambio se corre el riesgo de introducir errores al programa. “Por eso es importante respetar las precondiciones y pos condiciones de una refactorización, para asegurar que no se afecte el comportamiento del programa [21]”. Dependiendo de la complejidad, las refactorizaciones se pueden hacer de manera automática o semiautomática, pero lo que deben de hacer automáticamente es la validación de las precondiciones y pos condiciones.

Una herramienta de refactorización debe ayudar al diseñador y al administrador del marco de aplicaciones orientado a objetos, ofreciéndoles las refactorizaciones adecuadas a un problema y asegurando que cada refactorización

será bien realizada, pero no puede tomar la decisión final. Por tanto, en ese sentido, la refactorización no puede ser completamente automatizada [21].

La refactorización es una herramienta que puede y debe ser usada para muchos propósitos
[Fowl99]:

- Mejorar el diseño del software.
- Hacer el software más fácil de entender.
- Ayudar a encontrar errores.
- Ayudar a desarrollar código más rápidamente.
- Ayudar a que el software sea más flexible, robusto y reusable.

Algunas refactorizaciones son muy simples como cambiar el nombre de una variable o método; otras son más complejas y dependen de un lenguaje en particular; y las refactorizaciones más avanzadas están compuestas por refactorizaciones simples, e involucran el uso de los patrones de diseño orientados a objetos. De este último tipo es la refactorización que se presenta en esta tesis, utilizando patrones de diseño y está enfocada a marcos escritos en el lenguaje de programación Java.

3.4. Métricas Orientadas a Objetos

Una métrica de software es una forma estándar de medir algunos atributos del proceso de desarrollo de software. Ejemplos de estos atributos son: Tamaño, costos, defectos, comunicaciones, dificultad y ambiente. Aplicar las métricas nos lleva a un mejor entendimiento e incrementa lo predecible del proceso [35].

El producto a medir debe ser interpretado en un sentido amplio. Por ejemplo, las métricas pueden ser aplicadas a cualquier artefacto construido durante el desarrollo, incluyendo no sólo código sino también modelos de análisis y diseño, así como sus componentes. La meta de las métricas de software es la identificación medida de los parámetros esenciales que afectan el desarrollo del software [Cham97].

En este trabajo de tesis se realizó un estudio de algunas métricas orientadas a objetos específicamente la de coherencia**.**

La coherencia se define como el grado de relación funcional de 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” (R. C. Martin, 2002).

En términos generales, una clase está compuesta por elementos que pueden ser métodos y atributos. En el contexto de este trabajo de tesis, la coherencia mide la cantidad de funciones que interactúan para satisfacer una única responsabilidad en relación al número total de funciones en la entidad de software que los contiene.

Para calcular la métrica de coherencia de un caso de uso se propone la siguiente formula:

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

Donde:

***CrCU* =** Coherencia de caso de uso

***n* =** total de métodos en la secuencia

***m* =** total de métodos del módulo o paquete

La métrica CrCU necesita saber el total de métodos (n) participantes en la entidad de software y la cantidad de métodos de la secuencia interactiva (m) que participan para resolver una única responsabilidad. Los valores van de 0 a 1, donde 0 significa incoherencia total y 1 significa alta coherencia.

En la siguiente figura se muestra el diagrama de clases de caso de uso.

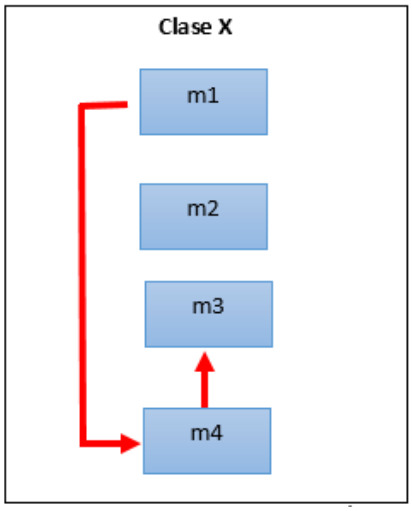


Figura 3: Diagrama de clases de caso de uso

Aplicando la fórmula de coherencia se obtiene el siguiente resultado:

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

$$Cr = \frac{3}{4}$$

$$Cr = 0.75$$

El resultado indica que el caso de uso atiende a más de una responsabilidad, ya que el valor ideal es CrCU = 1.

Métrica V-DINO (Dependencia por Interfaces que No se Ocupan)

Se implementó una métrica orientada a objetos para medir el problema de dependencia de interfaces por contar con interfaces que no se utilizan. Esta métrica entra en la clasificación de "métrica de árbol" en el marco de métricas orientadas a objetos propuesto en [35] y es una escala de tipo ordinal [35].

La métrica está basada en otra métrica ya existente, denominada C-NOC (Number Of Children) [1], que cuenta el número de clases derivadas directamente de una clase padre. También está basada en otros conteos de las funciones abtractas y de las implementaciones vacías.

Esta métrica debe ser aplicada a cada una de las clases abstractas que se tengan en un MAOO, y no hay una manera para combinar dichos valores, ya que cada árbol de clases se considera independiente. Por clase abstracta entendemos en este contexto que es una clase que tiene una o más funciones abstractas y que además tiene una o más clases derivadas que implementan dichas funciones abstractas.

Los elementos que intervienen son el Número de Funciones Virtuales (NFV), el Número de Funciones No Ocupadas (NFNO), y el número de subclases (C-NOC).

- *NFV:* Este número representa a las funciones abstractas declaradas en una clase abstracta. Este mismo número representa al número de abstracciones que deben ser implementadas en cada una de las subclases derivadas de la clase abstracta.

- *NFNO:* Este número representa a las funciones abstractas no utilizadas en todas las subclases de cierto árbol. Por funciones abstractas no utilizada se entiende una funcion que tiene código vacío o nulo sólo para cumplir los requerimientos del lenguaje de programación, pero desde el punto de vista lógico no tiene razón de existir porque no significa nada en el contexto del dominio del MAOO.

- *C-NOC:* Este número representa el número de subclases inmediatas subordinadas a una clase en la jerarquía de clases. Para este propósito se utiliza la métrica C-NOC (Chidamber – Number of Children) [1]. Es una medida de cuántas subclases van a heredar los métodos de la clase padre.

La métrica se llama V-DINO (Dependencia por Interfaces que No se Ocupan) y su expresión matemática se muestra en la Ecuación 1:

$$V - DINO = \frac{NFNO}{C - NOC \; x \; NFV} \qquad (1)$$

La expresión C-NOC x NFV representa el número total de abstracciones en todas las subclases, ya sea que se necesiten o no, y NFNO representa sólo a las interfaces no utilizadas. Debido a lo anterior, la condición NFNO ≤ (C-NOC x NFV) siempre será verdadera para cada MAOO, y por tanto, 0 ≤ V-DINO ≤ 1.

Esta métrica sólo puede ser usada para clases abstractas que tengan al menos una clase hija, para que C-NOC ≠ 0 y NFV≠ 0. Como se mencionó, esta métrica es una métrica de árbol, por lo que el análisis debe ser hecho de forma separada para cada árbol del MAOO. Este árbol debe empezar con una clase base abstracta.

Esta métrica está dentro de la escala ordinal, por lo que sus valores sólo pueden ser comparados, no sumados o usados para una media aritmética. Debido a la normalización, los valores de la métrica no dependen directamente del tamaño del MAOO o de sus árboles. El valor óptimo es 0, que significa que el problema no existe en ese árbol, y 1 es el peor caso (inalcanzable en casos prácticos), lo que significa que la totalidad de interfaces no son necesarias.

Para el MAOO de la Figura 2, se tiene que NFNO = 4, C-NOC = 3 y NFV = 3 para la clase abstracta *aStrategy*. Por tanto, la métrica V-DINO = 4 / (3 x 3) = 0.444. Esta jerarquía de clases corresponde a un MAOO más completo que cuenta con NFNO = 71, C-NOC = 29 y NFV = 4, es decir, un valor de V-DINO = 71 / (29 x 4) = 0.612.

En el capítulo de “Evaluación Experimental” se muestra éste y otros MAOO’s con el problema de dependencia de interfaces, con su cálculo de la métrica V-DINO.

De acuerdo a los casos de estudio y otras pruebas, se considera que el problema es grave cuando la métrica arroja un valor mayor o igual a 0.5, basándose en el caso más común mostrado a continuación.

El caso más común del problema de dependencia de interfaces se presenta en MAOO’s donde cada clase derivada utiliza la implementación de una abstracción diferente y todas las otras interfaces no se necesitan. En estos casos C-NOC = NFV y NFNO = (C-NOC x NFV) – NFV. Para estos casos, la métrica V-DINO está definida como se muestra en la ecuación 2.

$$V-DINO = \frac{(C-NOC\ x\ NFV)-NFV}{C-NOC\ x\ NFV} = \frac{C-NOC-1}{C-NOC} \qquad (2)$$

Por tanto, de acuerdo a la ecuación 2, un MAOO con dos clases tiene un valor de V-DINO = (2 – 1)/2 = 1/2 = 0.500, con tres clases tiene un valor de V-DINO = (3 – 1) / 3 = 2/3 = 0.666, con cuatro clases tiene un valor de V-DINO = (4 – 1) / 4 = 3/4 = 0.75, y así sucesivamente.

La Tabla 2 muestra el comportamiento de V-DINO en el caso donde cada implementación de interfaz es mutuamente exclusiva en todas las subclases.

C-NOC	V-DINO
2	1/2 = 0.500
3	2/3 = 0.666
4	3/4 = 0.750
5	4/5 = 0.800
6	5/6 = 0.833

Tabla 2: Caso más Común del Problema de Dependencia de Interfaces

Como se puede ver en la Tabla 1, V-DINO tiene un valor mayor o igual a 0.500 y nunca alcanzará el valor de 1.

En la métrica no se está considerando el caso de que las implementaciones de las abstracciones sean excluyentes. Debido a esto, para tomar la decisión de realizar o no la refactorización se considerarán dos análisis: Primero que V-DINO >= 0.5 y segundo que una abstracción a sustituir no esté implementada en alguna clase derivada junto con otra de las abstracciones que se van a sustituir, es decir, que sean mutuamente excluyentes, si no lo son, no se puede aplicar el método.

Cuando V-DINO < 0.500 no se puede asegurar que el problema se pueda resolver debido a que existen pocas interfaces que no se utilizan y las interfaces pueden no ser mutuamente excluyentes.

Por ejemplo, en el MAOO mostrado en la Figura 10, se tiene que C-NOC = 4, NFV = 3, y NFNO = 4, dando un resultado de V-DINO = 4 / (4 x 3) = 0.333. Ya que V-DINO < 0.500 se concluye que el MAOO mostrado en la Figura 9 tiene un problema de dependencia de interfaces menos severo que no necesita ser resuelto por el método de Separación de Interfaces. No se considera un problema ya que ambas interfaces se necesitan, por tanto, esas interfaces no pueden ser separadas.

La métrica está implementada en la herramienta y es calculada automáticamente generando los datos de la fórmula directamente de la base de datos que contiene el análisis del código fuente de un MAOO cualquiera.

3.5. Patrones de Diseño

Para la solución del problema resuelto por esta tesis se utilizaron los patrones de diseño. Estos engloban un conjunto de componentes que tienen un comportamiento conocido con mejores estructuras, usando para ello la experiencia de expertos y las prácticas de diseño [36].

A los patrones de diseño se les considera como una herramienta para análisis y diseño que prevé el uso futuro de los componentes de software.

Con los patrones de diseño se logra una mejor comunicación con el personal que trabaja en un diseño, ya que incrementan el reuso y productividad en el desarrollo de software. Con patrones de diseño es posible hacer reingeniería a

sistemas legados, que serían costosos de mantener en su estructura y diseño original, y con la reingeniería basada en patrones de diseño se obtienen nuevos sistemas con buen diseño y estructura que son más fáciles se comprender, mantener y reusar [36].

Un patrón de diseño nombra, abstrae, e identifica los aspectos clave de una estructura común de diseño que la hace útil para crear un diseño orientado a objetos reusable. Estos patrones identifican las clases participantes y sus instancias, sus funciones y colaboraciones, y la distribución de sus responsabilidades. Cada patrón de diseño está especializado en un problema particular de diseño orientado a objetos, por lo tanto, no todos los patrones de diseño se pueden utilizar en todos los casos, y se debe observar cuándo se pueden aplicar, y qué consecuencias traerá su uso [7].

Existen varios patrones de diseño, organizados en catálogos de patrones de diseño. Cada patrón tiene un objetivo específico, y al utilizarlos se adquieren sus ventajas y desventajas. En esta tesis se utilizan cuatro patrones de diseño del catálogo de Gamma, llamados ‘Strategy’, ‘Template Method’, ‘Command’ y ‘Singleton’.

Se utiliza la estructura de los patrones de diseño ‘Strategy’ y ‘Template Method’ para realizar el método de refactorización. Esto es porque la intención de estos patrones es diferente al uso que se les da en el método propuesto en esta tesis. Los patrones ‘Command’ y ‘Singleton’ fueron usados para realizar la interfaz de la herramienta.

Para describir un patrón de diseño, se utiliza la forma ‘Alexandriana’, también referida como ‘GOF’ o forma ‘canónica’ [7], donde:

1. El nombre, es una palabra o frase corta para referirse al patrón, y el conocimiento y la estructura que describe.
2. El problema, describe su intención, las metas y objetivos que quiere alcanzar dentro del contexto y problemática encontrada, donde por lo general, los problemas se oponen a los objetivos.
3. El contexto, describe las precondiciones bajo las cuales el problema y su solución parecen recurrir, y para lo que es deseable la solución, y con esto se puede observar si se puede aplicar el patrón o no a cierto problema.
4. Las fuerzas, o problemática describen las restricciones relevantes y cómo interactúan aun en conflicto con cada otra y con las metas que se desean.
5. La solución, representa a las reglas de relación tanto estáticas como dinámicas, que describen cómo obtener el resultado deseado. La descripción puede utilizar figuras y diagramas para identificar la estructura del patrón, sus participantes y sus colaboraciones para mostrar cómo se resuelve el problema. La estructura estática indica la forma y la organización del patrón, pero con frecuencia es el comportamiento dinámico lo que hace que el patrón viva.

6. Los ejemplos, describen las condiciones posteriores al uso del patrón y los efectos laterales, buenos y malos, del patrón.
7. El razonamiento, nos dice cómo trabaja el patrón actualmente y por qué es bueno. A diferencia de la solución, el razonamiento proporciona el conocimiento de la naturaleza interna de la estructura y los mecanismos clave que están bajo la superficie del sistema.
8. Los Patrones relacionados, muestran las relaciones estáticas y dinámicas entre un patrón y otros. Existen patrones que comparten las fuerzas, otros comparten los contextos iniciales y finales, unos dan una solución alterna al mismo problema, incluso algunos pueden aplicarse simultáneamente.

Patrón de Diseño '*Strategy'* [2]

Intención: Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. 'Strategy' deja que los algoritmos varíen independientemente de los clientes que los utilizan.

Problemática: Si los clientes, potencialmente, tienen algoritmos embebidos en ellos, es difícil: reusar estos algoritmos, intercambiar los algoritmos, desacoplar diferentes capas de funcionalidad, y variar la elección de la política a seguir en tiempo de ejecución. Estos mecanismos de políticas embebidas rutinariamente se manifiestan así mismos como expresiones condicionales múltiples.

Participantes:

Clase Contexto (*Context*): Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. El cliente lo configura con una estrategia específica mediante una referencia a la estrategia requerida. Contexto puede definir una interfaz que permita a la estrategia el acceso a sus datos en caso de que fuese necesario el intercambio de información entre el contexto y la estrategia. En caso de no definir dicha interfaz, el contexto podría pasarse a sí mismo (this) a la estrategia como parámetro.

Estrategia (*Strategy*): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.

EstrategiaConcreta (*ConcreteStrategy*): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

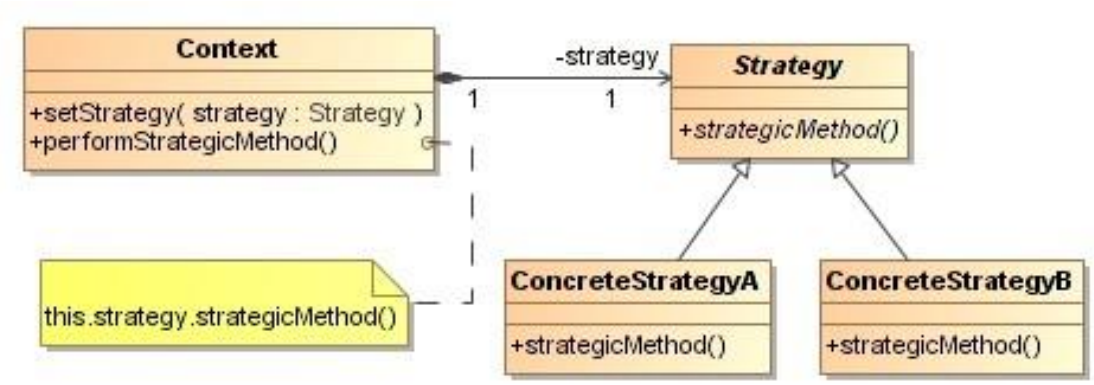


Figura 4: Estructura genérica del Diseño '*Strategy*'

Patrón de Diseño '*Template Method*' [2]

Intención: Define el esqueleto de un algoritmo en una sola operación, difiriendo algunos pasos a cada subclase cliente. Deja que las subclases implanten ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

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

Motivación: Cuando se construyen jerarquías de clases complejas para una aplicación, a menudo se duplican distintas partes del código. Esa situación no es deseable, porque la intención es reutilizar todo el código que sea posible. La refactorización de código para que los métodos comunes estén en una superclase es un paso en la dirección correcta. El problema es que algunas veces una operación que ha sido Refactorizada confía en la información específica que solamente está disponible en una subclase. Debido a esto, los desarrolladores a menudo deciden no Refactorizar y aceptar la presencia de código duplicado en distintas clases.

Participantes:

Clase Abstracta (AbstractClass)
Define operaciones primitivas abstractas que las clases concretas definen para implementar los pasos de un algoritmo. Esta clase implementa un método de plantilla que define el esqueleto de un algoritmo. El método de plantilla llama operaciones primitivas, así como operaciones definidas en la clase abstracta o las de otros objetos.

Clase concreta (ConcreteClass)
Implementa las operaciones primitivas para llevar a cabo los pasos específicos de subclase del algoritmo.

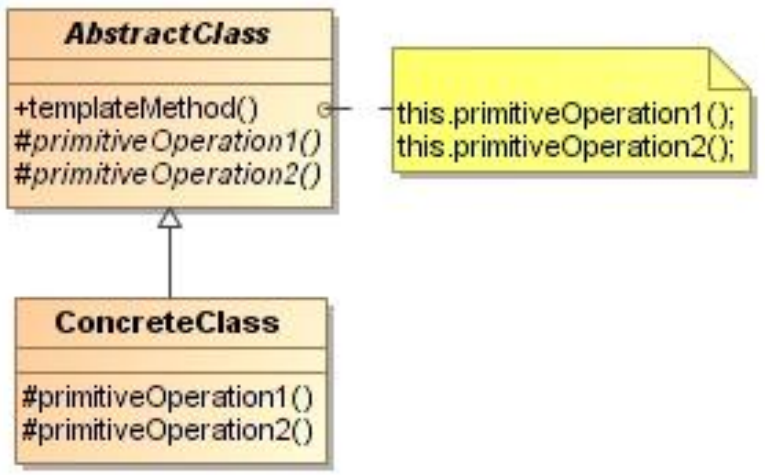


Figura 5: Estructura genérica del Diseño '*Template Method*'

Patrón de Diseño 'Command' [2]

Intención: Trata una petición como un objeto encapsulado, deja parametrizar al cliente con peticiones diferentes, encola las peticiones y soporta operaciones no revertibles.

Problemática: En ocasiones es necesario enviar peticiones a objetos sin saber nada acerca de la operación solicitada ó del receptor de la petición. Por ejemplo, las herramientas de desarrollo de interfaces de usuario incluyen objetos como botones y menús que llevan a cabo una solicitud en respuesta a entradas del usuario. Pero la herramienta no puede implementar la petición explícitamente en el botón o menú, porque sólo las aplicaciones que usen la herramienta de desarrollo saben que debe realizarse en cada objeto. 'Command' permite que los objetos de la herramienta de desarrollo hagan peticiones a objetos no especificados de la aplicación convirtiendo la propia petición en un objeto. Este objeto puede ser almacenado y transportado como cualquier otro objeto.

Motivación: El patrón de diseño "Command" permite a los objetos del kit de herramientas hacer peticiones de objetos de aplicaciones no especificadas transformando la petición misma en un objeto. Este objeto puede ser almacenado y pasado a otros objetos.

La clave de este patrón es una clase abstracta "Command", la cual declara una interfaz para ejecutar operaciones. En su forma más simple esta interfaz incluye una operación abstracta "Execute".

Las subclases concretas “Command” especifican un par “receptor-acción” almacenando al receptor como una variable de instancia e implementando “Execute” para invocar la petición.

Participantes:

Clase comando (Command).
Declara una interfaz para ejecutar una operación. La clase comando concreta (ConcreteCommand).
Define una liga entre un objeto “Receiver” y una acción. Implementa “Execute” con la invocación de la operación correspondiente del objeto “oReceptor”.

Clase cliente (Client).
Crea un objeto “ConcretCommand” y establece su variable de instancia “oReceptor”. Invocador. Pide al comando efectuar una operación. Receiver Sabe cómo efectuar la operación asociada con una petición. Cualquier clase puede servir como “Receiver”.

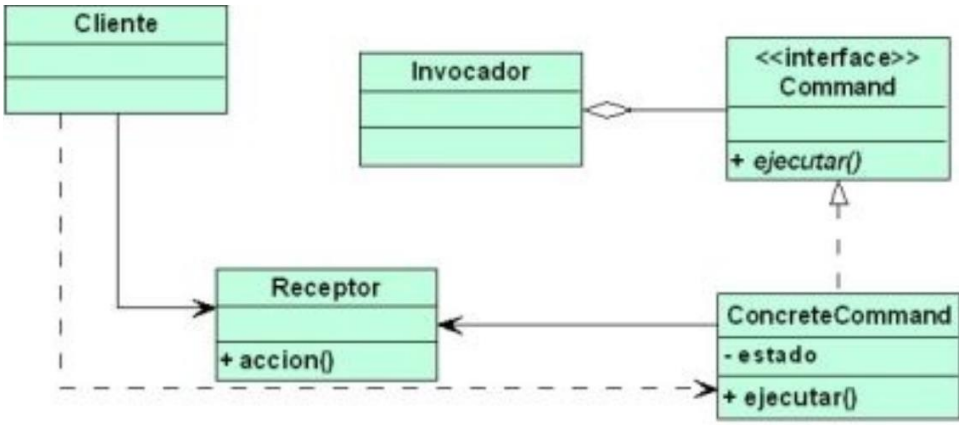


Figura 6: Estructura genérica del Diseño ‘Command’

Patrón de Diseño ‘Singleton’ [2]

Intención*:* Asegura que sólo exista una única instancia de una determinada clase y proporciona la forma de acceder a ella.
Problemática*:* Es importante para algunas clases tener exactamente una instancia y tener fácil acceso a ella. Por ejemplo, aunque puede haber muchas impresoras en un sistema, sólo puede haber una cola de impresión. Una variable global hace a un objeto accesible, pero no previene de hacer múltiples instancias. Una mejor solución es hacer que la propia clase sea responsable de llevar el control de su instancia única. La clase puede asegurar que no se crearán más instancias, interceptando peticiones para crear nuevos objetos, y puede proveer una forma de acceder a la instancia.

Casos de Aplicación:
• Se utiliza cuando debe de existir sólo una instancia de una clase, y debe estar

accesible a los clientes desde un punto conocido de acceso.
• Cuando la instancia única debería ser extensible por subclases, y los clientes deberían poder usar una instancia extendida sin modificar su código.
Estructura: La clase que necesita una única instancia, ***Singleton***, declara un método de clase, ***Instanciar( )***, que permite acceder a la única instancia, ***instanciaUnica***, y puede ser el responsable de crear dicha instancia.

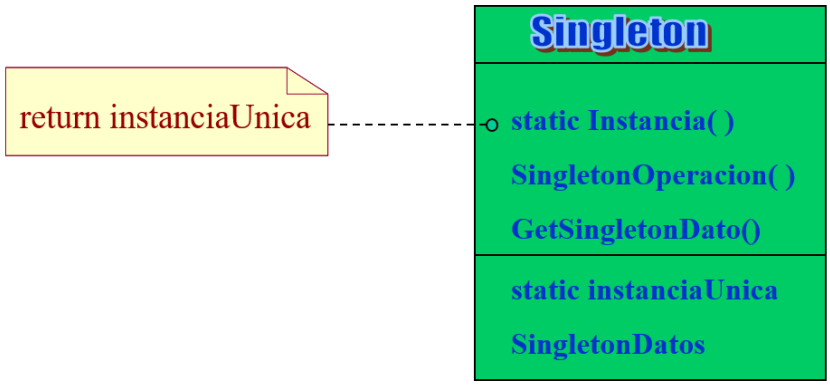


Figura 7: Estructura genérica del Diseño ‘Singleton’

Capítulo 4) MÉTODO DE SOLUCIÓN

4.1. Refactorización por el Método de Separación de Interfaces

La refactorización del MAOO consiste en establecer funciones abstractas con nombres genéricos, que se van especializando en clases derivadas. Es decir, se sustituyen las interfaces duplicadas por una sola.

Esta interfaz genérica juega el papel de ***Algoritmo Interfaz ()*** del patrón de diseño 'Strategy', que tendrá su implementación en nuevas clases intermedias. La clase base original es la clase ***Estrategia*** de un patrón de diseño.

Siguiendo el patrón 'Strategy' de la Figura (Patron Strategy capítulo 3), se crearán clases intermedias entre la clase base abstracta y las clases derivadas para mostrar las diferentes estrategias a seguir. Se considera que cada estrategia es una especialización del MAOO. Estas clases intermedias también tienen nombres genéricos que pueden ser especificados por el experto del dominio y representan las clases ***EstrategiaConcretaX*** del patrón 'Strategy'.

Cada clase intermedia declara a una de las interfaces originales duplicadas, y esta clase debe servir como clase base de las clases derivadas originales que si ocupan la interfaz.

Aquí es donde entra el patrón 'Template Method', ya que la interfaz genérica es implementada como un método de plantilla en las clases intermedias. La implementación de este método de plantilla sólo es una llamada hacia la interfaz de cada una de sus estrategias o especializaciones, y esta interfaz ya se encuentra implementada por las clases concretas derivadas de las clases intermedias.

Siguiendo la estructura del patrón '*Template Method*' de la Figura 5, la función abstracta ***AlgoritmoInterfaz ()*** juega el papel de ***Template Method ()***, las clases intermedias son las ***ClaseAbstracta*** de la figura 5 y las clases con la implementación son las ***ClaseConcretaX*** de la figura 5, siendo la interface original de la ***OperacionPrimitivaX ()***.

Como resultado, cada clase derivada tiene que implementar a solo una y única interfaz, eliminando las interfaces que no se ocupan. Así el MAOO resultante se podrá extender sin modificar nada.

En la Figura 8 siguiente se concluye el ejemplo mostrado anteriormente en el "Planteamiento del Problema", utilizando la metodología ya descrita. Esta figura muestra el mismo MAOO de la Figura 2 refactorizado con los patrones de diseño 'Strategy' y 'Template Method' utilizados de la forma que se mencionó anteriormente.

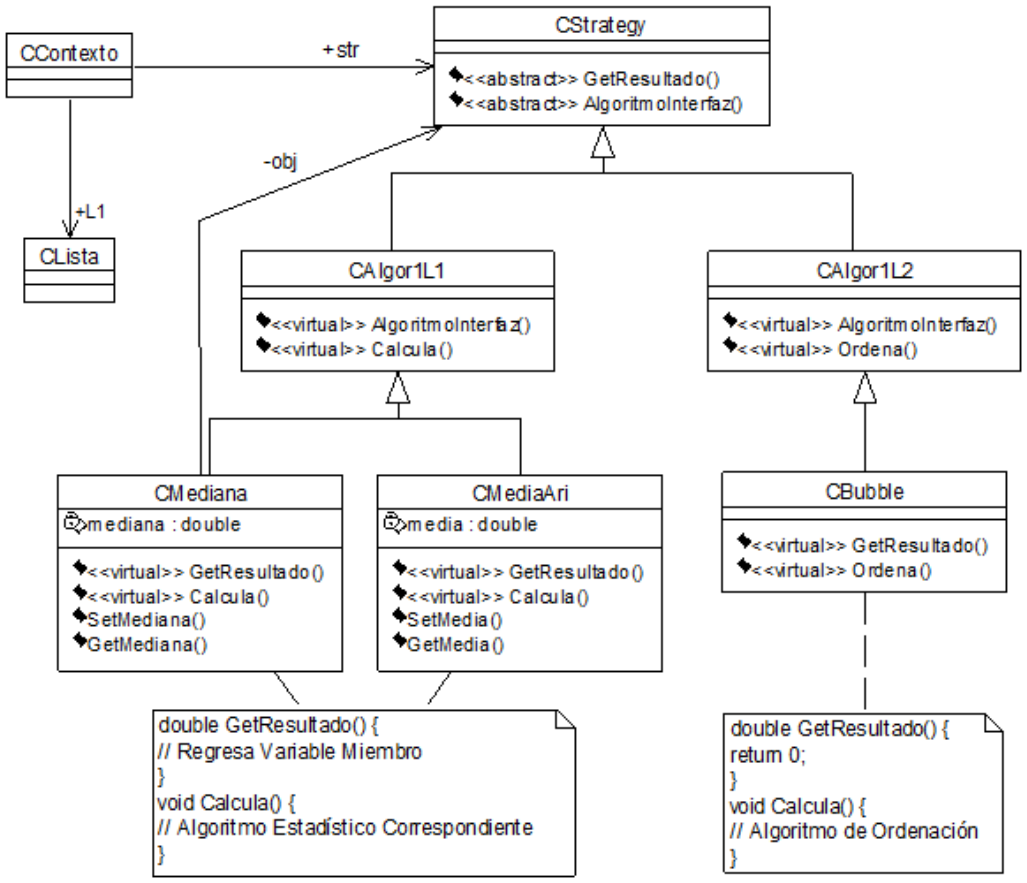


Figura 8: MAOO Re factorizado Usando el Método de Separación de Interfaces, el cual elimina las interfaces no usadas o vacias.

Nótese que la clase *CStrategy* sólo tiene una interfaz genérica *AlgoritmoInterfaz()*, se han creado las clases intermedias *CAlgor1L1* y *CAlgor1L2* para indicar las estrategias de Ordenación y Cálculos, en donde cada una implementa *AlgoritmoInterfaz( )* con una llamada hacia su función de interfaz apropiadamente.

Las clases derivadas originales *CBubble*, *CMediaAri* y *CMediana* implementan la única interfaz que ocupan. *CBubble* implementa a *Ordena ()*, *CMediaAri* y *CMediana* implementan a *Calcula ()*. Todas siguen implementando a *GetResultado ()*.

De acuerdo con la estructura genérica de los patrones; *CStrategy* es la clase 'Strategy', *AlgoritmoInterfaz()* es la interfaz genérica del 'StrategyMethod' y las interfaces *TemplateMethod( )* del patrón de diseño 'Template Method', *CAlgor1L1* y *CAlgor1L2.* A la vez, estas son las clases derivadas *ConcretStrategyA y ConcretStrategyB* del 'Strategy' y las clases derivadas *ConcreteClass* del 'Template Method',son *CMediaAri*, *CMediana* y *CBubble*, mientras que los métodos *Ordena( )* y *Calcula( )* son las interfaces del 'Template Method' *OperacionPrimitivaX( )*.

El MAOO en total tiene cuatro interfaces problemáticas y por lo que se requiere de dos clases abstractas intermedias para resolver el problema. Esta transformación del marco estadístico se puede ver en el capítulo de "Evaluación Experimental". Como se observará, cuando este método de refactorización se aplique al MAOO completo de estadística, del cual forma parte esta pequeña jerarquía de clases, se eliminarán 58 funciones no utilizadas de 29 clases derivadas, lo cual mejorará considerablemente el diseño del MAOO y reducirá el código del mismo.

Beneficios Obtenidos por la Refactorización

Básicamente existen dos beneficios que se logran con la separación de interfaces, uno es la facilidad de reusar parte del MAOO en otras aplicaciones y el segundo es la facilidad de extensión de la funcionalidad del MAOO.

El reuso se logra debido a que las interfaces ya no son dependientes entre sí, por lo que se puede transportar parte del MAOO a otra aplicación sin necesidad de modificar nada, y sin que haya interfaces que obstaculicen esta transportación.

En el ejemplo de la Figura 8 se podría reusar sólo la funcionalidad de ordenamientos a otras aplicaciones, transportando las clases *CStrategy*, *CAlgor1L2* y *CBubble* uniéndolos a otro cliente diferente de *CContexto*. En estas clases se puede notar que al haber separado estas clases no queda ningún rastro de que existiera la interfaz *Calcula ()*. Lográndose la independencia de las clases de cálculo.

La mejora en la cualidad de extensión se logra debido a que cuando se necesita agregar nueva funcionalidad al MAOO, no se requiere modificar nada de lo ya existente, solo se agregarían las nuevas clases de manera correspondiente.

Si la nueva funcionalidad ocupa las interfaces que ya están definidas, sólo se agrega por herencia las nuevas clases a las clases intermedias. Si la nueva funcionalidad ocupa de una interfaz diferente a las existentes, pero tiene la misma firma que las anteriores, sólo se tiene que agregar una nueva clase intermedia que herede de la clase abstracta, y agregar las nuevas clases con funcionalidad diferente a esa clase intermedia.

Ambos casos se ilustran en la Figura 9, donde 1) se está agregando un nuevo algoritmo de ordenación de datos, Shell. 2) Otro cálculo estadístico, desviación media, y 3) la funcionalidad de resolver sistemas de ecuaciones, método de Gauss Jordan.

El nuevo método de ordenación por Shell está implementado en la clase *CShell*, que hereda de *CAlgor1L2*, la cual cuenta con la interfaz *Ordena()*. La desviación media está implementada en la clase *CDesvMed*, heredando de *CAlgor1L1* que ofrece la interfaz *Calcula()*. Para la nueva funcionalidad de resolución de ecuaciones se agrega la clase intermedia *CAlgor1M1* que ofrece la interfaz *Resuelve ()* y la clase *CGaussJor* tiene implementado el algoritmo de resolución.

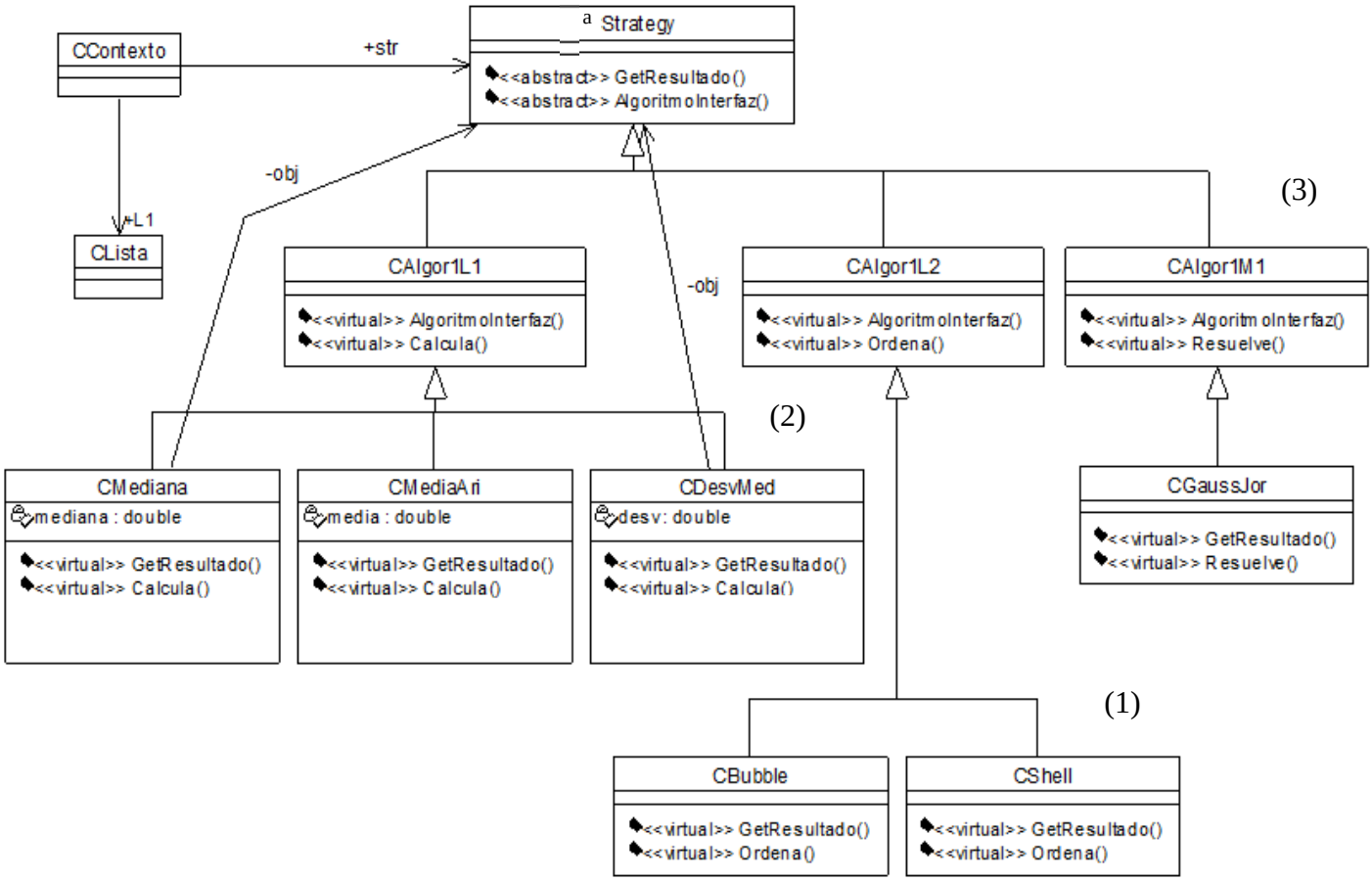


Figura 9: MAOO con Funcionalidad Extendida

Todas estas nuevas funcionalidades pueden ser invocadas mediante la interfaz genérica *AlgoritmoInterfaz ()* utilizando para ello el polimorfismo sobre un objeto de la clase abstracta *CStrategy*.

Se puede observar que, ni *aStrategy* clases derivadas específicas. Por lo tanto, al haber aplicado el principio de "separación de interfaces" se consigue una mejor estructura en el diseño. Así se logra el respeto de los principios de diseño fundamentales de "Abierto / Cerrado" y el de "Sustitución".

4.2. Algoritmo de Separación de Interfaces

Para implementar el método de Separación de Interfaces se propuso el algoritmo que se describe a continuación. En cada paso del algoritmo se mencionan los nombres de los objetos del MAOO de ejemplo (Figuras 2 y 8) involucrados en dicho proceso.

1. Lo primero es encontrar funciones abstractas, para así determinar que clases son abstractas o interfaces. En el caso del ejemplo, sería la clase *CStrategy*.

2. En cada clase abstracta o interfaz se determinan sus clases derivadas. En el caso del ejemplo, serían las clases *CBubble*, *CMediaAri* y *CMediana*.

3. En cada clase derivada se busca la implementación de las funciones abtractas. El objetivo es buscar funciones vacías o que no se utilicen. Aparecen vacías si su valor de retorno es void. También para las funciones con valor de retorno se encuentran implementaciones con una línea "return 0 o null". En el ejemplo se tienen vacías las funciones *Calcula ()* y *GetResultado ()* en la clase *CBubble* y *Ordena ()* en las otras dos clases derivadas.

4. En la clase abstracta *´CMediana´ y* se *´CMediaAri´* determina que funciones abstractas son las problemáticas. Se consideran problemáticas si alguna clase derivada implementó la función vacía. En el ejemplo las tres interfaces *Calcula ()*, *Ordena ()* y *GetResultado ()* se consideran problemáticas.

5. Se coloca en la clase abstracta una nueva función de interfaz más abstracta, denominada *AlgoritmoInterfaz ()*, y se eliminan de esta clase las interfaces problemáticas originales que pueden ser sustituidas por la nueva interfaz. Se crean tantas nuevas clases intermedias de interfaz como se requiera. En el ejemplo se sustituyen las interfaces *Ordena ()* y *Calcula ()* por *AlgoritmoInterfaz ()*. Para hacer las nuevas interfaces se toman las consideraciones mencionadas en las precondiciones.

6. Se agrupan las clases derivadas originales por las interfaces que implementan. En una clase abstracta intermedia que implementa a la interfaz genérica *Algoritmo Interfaz ()* como un 'Template Method'. Esta

implementación lo que deberá hacer es invocar a la implementación de las interfaces originales de manera adecuada.

7. En cada clase intermedia se coloca la función abstracta original que sí es implementada por las clases derivadas del grupo. Del ejemplo anterior las funciones *Ordena ()* y *Calcula ()* serían invocadas desde las nuevas clases intermedias, que denominaremos *CAlgor1L1* y *CAlgor1L2*, respectivamente.

8. Se cambia la relación de herencia de las clases derivadas hacia la clase intermedia, que a su vez hereda de la clase abstracta original.

9. Se eliminan las funciones con implementaciones vacías en cada una de las clases derivadas originales.

10. Para asegurarse que la refactorización no afecte al MAOO, dentro del código de todo el sistema se buscan las llamadas a las funciones abstractas originales, y se cambian por los nuevos nombres genéricos nuevos. Esta es la única post-condición del método de Separación de Interfaces.

Cabe aclarar que cuando un MAOO tiene varias clases abstractas, este proceso se sigue para cada una de las clases abstractas de manera independiente, ya que el método sólo afecta a la jerarquía de herencia de esa clase abstracta no al sistema completo. Este algoritmo fue implementado con éxito en la herramienta SR2 Refactoring que será explicada a detalle en el próximo capítulo.

Precondiciones del Método de Separación de Interfaces

Como todo método de refactorización, para ser ejecutado, debe validar ciertas precondiciones para que sea posible y factible la transformación del código fuente y llevar a cabo un rediseño exitoso. Todas las precondiciones son verificadas de manera automática por la herramienta antes de tratar de llevar a cabo la refactorización. Las precondiciones del método de Separación de Interfaces son:

- Las interfaces que son seleccionadas para sustituirse por una nueva interfaz deben contar con los mismos parámetros y el mismo valor de retorno.

- Para que la implementación de una interfaz con un valor de retorno se considere vacía se debe de considerar lo siguiente: La única instrucción que debe haber, si el valor de retorno es de tipo numérico, es “{return 0; }”, si el valor es de tipo boleano, “{ return false; }”, si el valor es de tipo cadena, “{ return “”; }”, y si el valor es referencia a un objeto, “{ return null; }”. Obviamente, si la función no tiene valor de retorno (void) la función estará libre de implementación de código, es decir, sólo estarán las llaves de apertura y cerradura “{}”.

- Todos los nombres de las clases intermedias que se agregan no deben estar siendo usados por otra clase; la misma condición aplica para los nombres de las interfaces genéricas agregadas.

- La consideración que se debe hacer para la refactorización es que las implementaciones de las funciones abstractas que se sustituyen deben ser mutuamente excluyentes, es decir, no debe haber una clase derivada que implemente a más de una función abstracta.

En el caso del MAOO de la Figura 2 no se puede hacer nada con la interfaz *GetResultado ()* que no es requerida en la clase *CBubble*, ya que no existe otra interfaz con su misma firma.

Se puede sustituir a las interfaces *Ordena ()* y *Calcula ()* por una sola interfaz, ya que ambas funciones tienen la misma firma y son mutuamente excluyentes.

Por mutuamente excluyente se entiende que cuando se agrupen las clases derivadas por las interfaces que implementen, una clase no debe estar en dos grupos, es decir, que no tenga que derivarse de dos clases intermedias. Si en el ejemplo de la Figura 2 se tuviera otra clase derivada, *ClaseX*, que implementara con código a las interfaces *Ordena ()* y *Calcula ()*, entonces se diría que *Ordena ()* y *Calcula ()* no son excluyentes. En ese caso no se puede aplicar el método de Separación de Interfaces, ya que la interfaz genérica no sabría hacia cuál de los dos métodos dirigirse. Este ejemplo de interfaces que no son mutuamente excluyentes se muestra en la Figura 10.

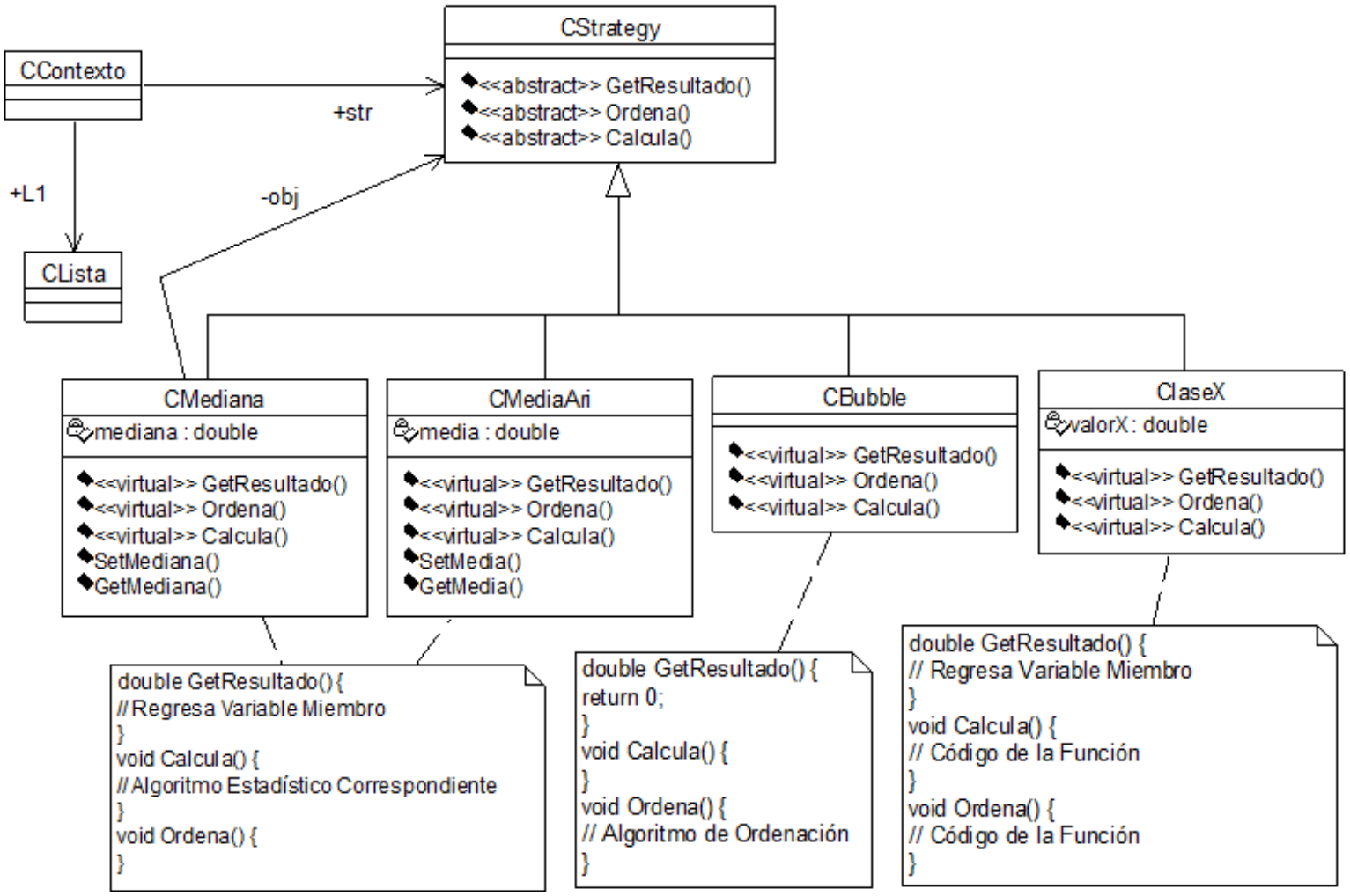


Figura 10: Ejemplo de un MAOO con Interfaces que no son Mutuamente Excluyentes

La clase *ClaseX* implementa con código a *Calcula ()* y *Ordena ()*, por tanto, no son mutuamente excluyentes estas interfaces, y no se pueden sustituir por una sola ya que ambas se necesitan.

Como consecuencias negativas de la arquitectura resultante se tiene que se aumenta en uno el nivel de profundidad del árbol de herencia, y también se añaden más llamadas a funciones para llegar a la implementación real.

Capítulo 5) DESARROLLO DEL SISTEMA

5.1. Análisis del Sistema

A continuación, se presenta un extracto del documento de análisis del sistema, modelado con diagramas de casos de uso de UML.

La Figura 11 muestra el diagrama de casos de uso de la herramienta, mostrando las opciones que tiene el usuario.

Hay cuatro casos de uso principales: Manejar Archivo, Análisis Sintáctico, Medición y Re-factorización. Los actores son: Usuario, la persona que usa la herramienta; Archivo Original, archivos del MOO que escoge el usuario y que serán Re-factorizados; Archivo Fuente, copia de seguridad de los archivos originales; Archivo Final, los archivos originales modificados usando los métodos de refactorización.

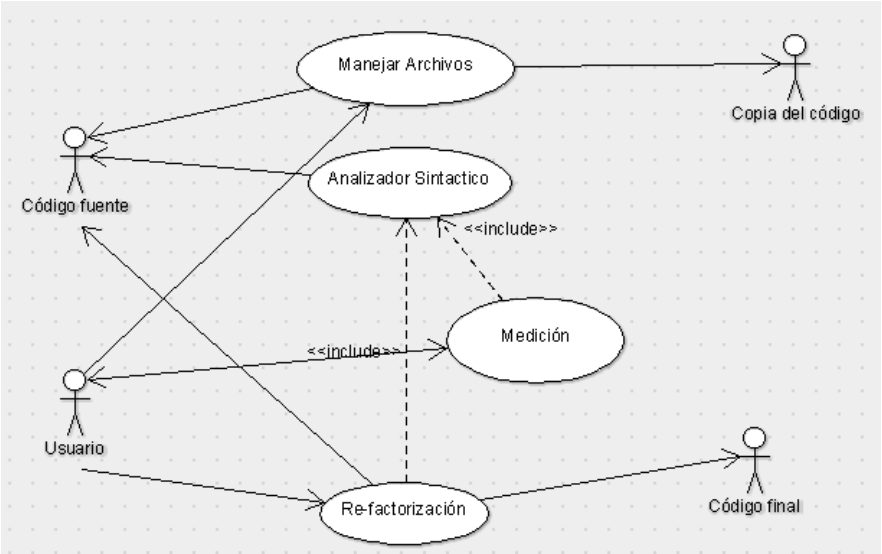


Figura 11: Diagrama de Casos de Uso Principal.

La Figura 12 muestra el diagrama del caso de uso Manejar Archivo, el cuál debe ser invocado antes de hacer cualquier otra operación.

Primero se seleccionan todos los archivos del MOOs que se desea analizar y Re-factorizar, y se crea una copia de seguridad de estos archivos.

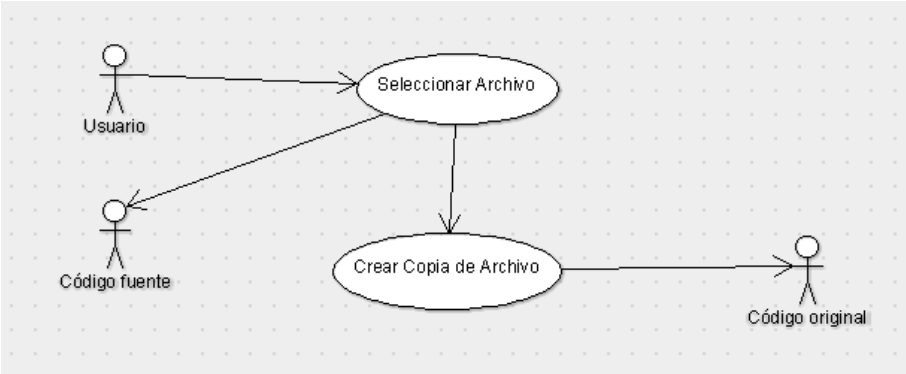


Figura 12. Diagrama del Caso de Uso Manejar Archivo.

La Figura 13 muestra el diagrama del caso de uso Análisis Sintáctico, que puede ser invocado ya sea desde la pantalla de la métrica o desde el método de refactorización.

Primero se encuentran las clases abstractas y sus funciones abstractas, posteriormente se encuentran las clases derivadas de las abstractas y la implementación de las interfaces.

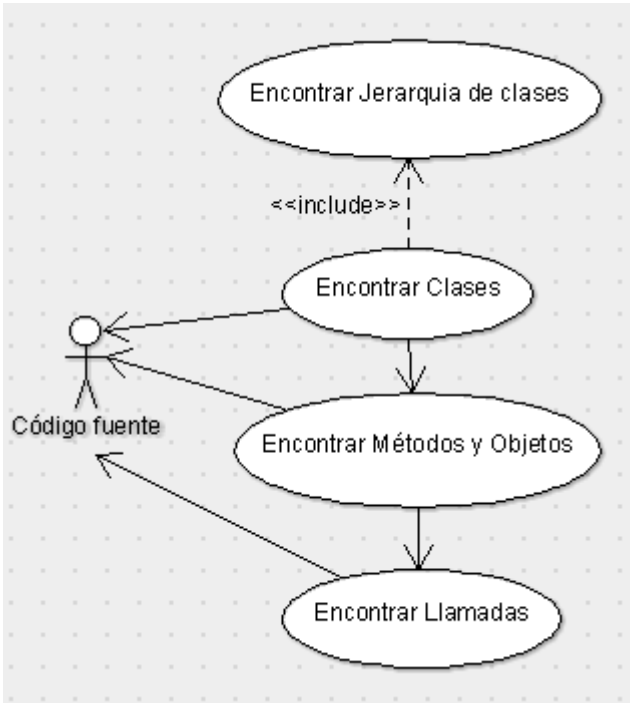


Figura 13: Diagrama del caso de uso Análisis Sintáctico

La Figura 14 muestra el diagrama del caso de uso Calculo de *Métrica*, que es invocado después del análisis sintáctico cuando se elige la opción de Calcular Métrica.

Primero se debió de hacer el análisis sintáctico, y se calcula la métrica basada en la información de la base de datos. De acuerdo al valor de la métrica se emite una interpretación del resultado, con la acción recomendada a seguir.

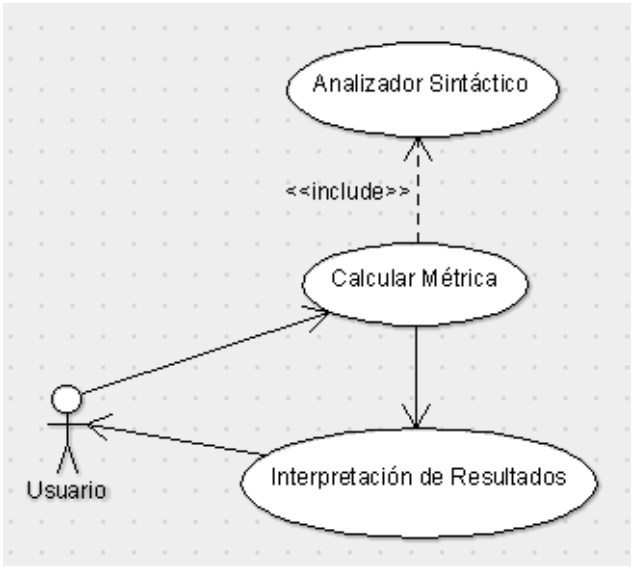


Figura 14: Diagrama del caso de uso Calculo de Métrica

La Figura 15 muestra el diagrama del caso de uso Re-factorización, que es invocado cuando se requiere de la refactorización. Este caso consiste en crear las nuevas clases intermedias para aplicar el método de refactorización.

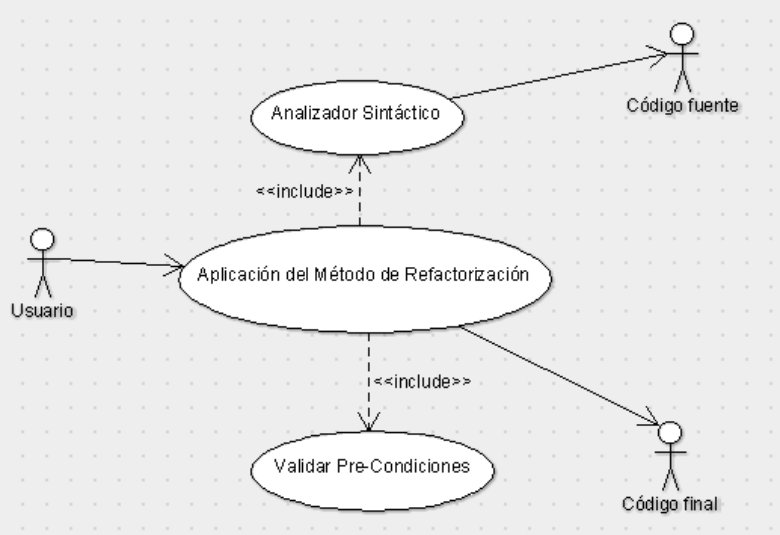


Figura 15: Diagrama del caso de uso del método de Re-factorización

5.2. Diseño del Sistema

Para diseñar el sistema se utilizaron varios patrones de diseño, con el fin de darle facilidades de extensión a la herramienta. Como se mencionó anteriormente, esta arquitectura permite la integración de otro método de refactorización al actual y se podrán agregar más en el futuro.

Se utiliza el patrón de diseño 'Singleton' para manejar las pantallas internas que tiene el sistema, de tal manera que sólo exista una instancia por pantalla y todas ellas están contenidas en la pantalla principal.

La estructura de pantallas se muestra en la Figura 16, además se muestran otras clases que dan soporte a la aplicación.

En la Figura 16 se muestra la vista de la jerarquía de pantallas, con cada pantalla implementando el patrón ‘Singleton’.

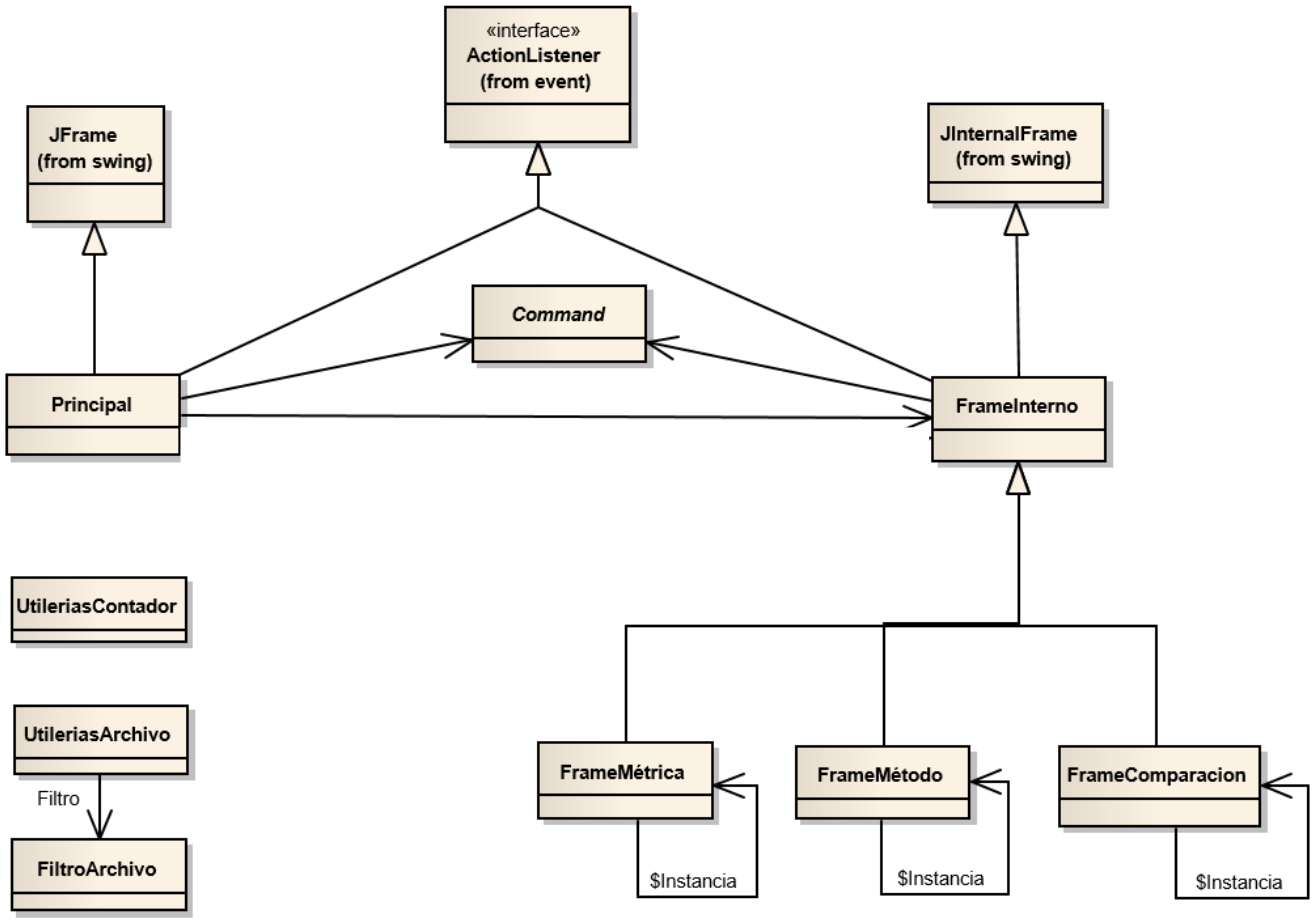


Figura 16: Arquitectura del Sistema, Jerarquía de Pantallas

La pantalla base del sistema, *Principal*, es la que contiene todas las instancias de las pantallas específicas. Funciona como un contenedor para las pantallas internas, las cuales son accedidas a través de un conjunto de objetos de tipo menú.

La pantalla Principal cuenta con un objeto comando, de clase *Command*, ya que existe una jerarquía especial de objetos del patrón de diseño *“Command”*, subclases de *CommandPrincipal*, que sirven para invocar a cada pantalla, utilizando el método ofrecido por el patrón ‘Singleton’ para crear y acceder al único objeto de tipo pantalla.

La jerarquía de pantallas internas inicia con la clase *FrameInterno* y un objeto comando, de clase *Command*, de tal manera que todas las pantallas internas tengan acceso a la jerarquía de comandos. Las pantallas específicas

FrameX especifican el *FrameInterno*, y reciben como parámetros en su construcción específica que requiere la pantalla y los archivos originales seleccionados por el usuario que conforman al MAOO para re-factorizar. Cada pantalla *FrameX* tiene una jerarquía de comandos *CommandX*, para las acciones de objetos gráficos de la pantalla. Cada clase *FrameX* es la encargada de controlar su única instancia, utilizando el patrón de diseño 'Singleton', contando con un método *instanciar ()* para crear y acceder a dicho objeto.

La clase *UtileriasArchivo* es para manejar la apertura de archivos, así como la generación de archivos de respaldo. *FiltroArchivos* es una clase auxiliar a *UtileriasArchivo* para manejar los tipos de archivos a abrir. Finalmente, la clase *UtileriasContador* sirve para manejar la comparación de archivos en la pantalla de *Comparación*, para observar las diferencias entre los archivos fuente originales y los archivos Re-factorizados, y funciones adicionales para análisis de código fuente.

Se utiliza el patrón de diseño '*Command'* para manejar los eventos generados en las pantallas por los botones, menús y listas desplegables. Se maneja una clase de tipo '*Command'* por acción. Todos los comandos de una pantalla tienen un '*Command'* genérico que implementa el patrón 'Template Method'. Además, cada clase *Command* genérica cuenta con una instancia de la pantalla a la que corresponde, para acceder de manera directa a sus elementos y poderlos modificar

La estructura de comandos se muestra en la Figura 17, con las agregaciones que tienen cada uno de los comandos genéricos de pantalla.

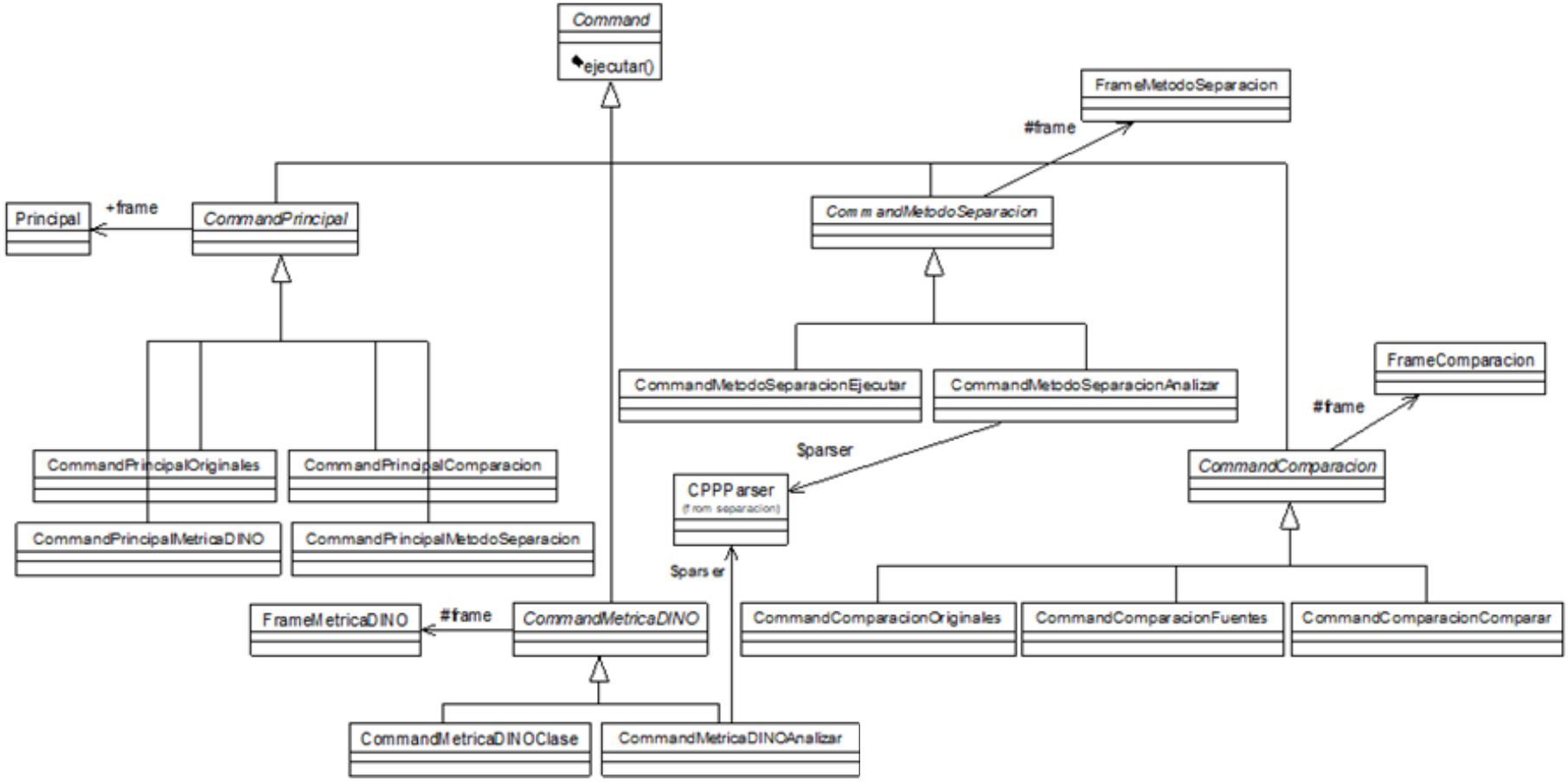


Figura 17. Arquitectura del Sistema, Jerarquía de Comandos

En la Figura 17 se muestra la vista de la jerarquía de comandos implementando el patrón de diseño "*Command*". Cada clase representa una acción de un elemento de la interfaz gráfica. Para ejecutar dicha acción, se utiliza la interfaz *ejecutar ()* declarada en la clase *Command* e implementada por todas las subclases.

Ademas se muestra una de las clases generadas por ANTLR para hacer el análisis sintáctico, *´CPPParser´*, invocado desde distintos comandos. Cada método de refactorización tiene su propio analizador, con su propio archivo *´CPPParser´* por ejemplo, incluido en un paquete con el nombre del método.

Adicionalmente se agregó la funcionalidad para manejar usuarios del sistema, debido a que es necesario controlar los cambios que se le hagan a un MAOO, los cuales sólo deben ser hechos por el administrador del MAOO y notificados a todos los usuarios del MAOO.

En la Figura 18 se muestra la vista de la jerarquía de clases que se encargan de hacer este control. Primeramente, se tienen tres pantallas, "*FrameAccesos"*, "*FrameUsuarios"* y "*FramePassword"*, para manejar los tipos de accesos al sistema, los usuarios y para autentificar al usuario, respectivamente. Estas pantallas también se derivan de la clase *FrameInterno*.

Los tipos de acceso sirven para especificar que menús pueden ser utilizados por los usuarios, por ejemplo, se podría restringir el acceso a usuarios para que sólo calculen las métricas, o que puedan Re-factorizar, pero no dar de alta usuarios. Los usuarios tienen especificado un tipo de acceso y nombre de usuario, la clave debe ser especificada en la pantalla *FramePassword*.

Cada pantalla tiene su jerarquía de comandos para realizar las diferentes acciones, las cuales tienen como superclase a *Command* y cada jerarquía tiene una clase padre "*CommandX"*, y también una clase *CommandPrincipalX* para mandar llamar la pantalla desde la clase *Principal*. Las mismas reglas de los patrones 'Command' y 'Singleton' se manejan en estas clases.

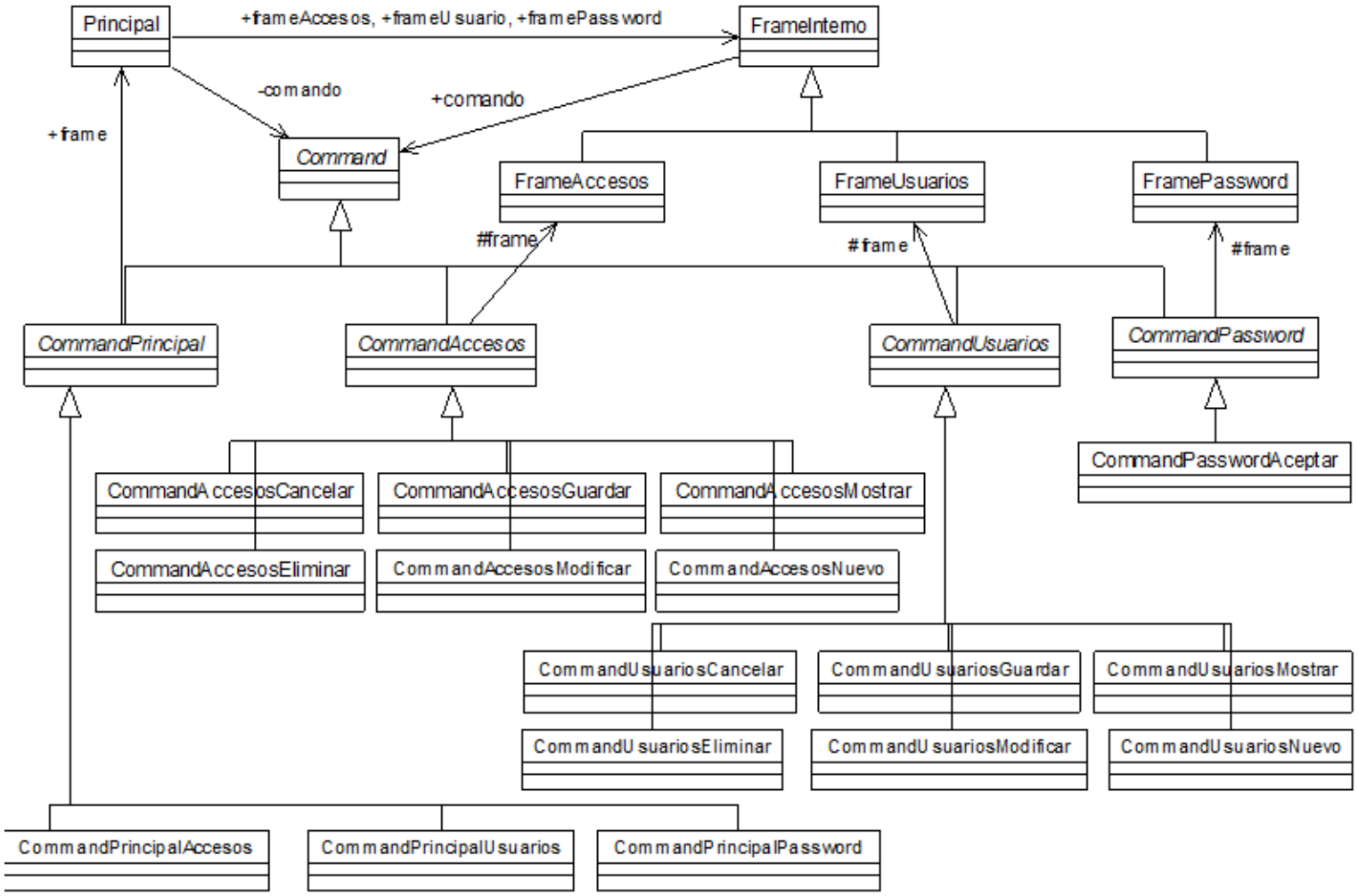


Figura 18. Arquitectura del Sistema, Manejo de Usuarios y Accesos

Con esta estructura se puede apreciar qué fácilmente se puede extender el sistema, para agregar nuevas pantallas con sus respectivos comandos. Para realizar esto, simplemente se tienen que extender por herencia la clase '*FrameInterno*' y colocar el '*FrameX*' específico, que implemente el patrón 'Singleton'. Los esqueletos de estas pantallas son iguales en todas las clases '*FrameX*'.

También se puede extender por herencia la clase '*Command*', agregando una clase '*CommandX*' que tenga incluido el '*FrameX*'. A esta clase '*CommandX*' se le agregan por herencia cada clase '*CommandXAccion*' específica de los objetos gráficos de la pantalla.

Finalmente, al cliente *Principal* se le tiene que agregar una instancia para la nueva pantalla '*FrameX*', así como el menú para acceder a ella, utilizando un objeto '*CommandPrincipalX*'.

5.3. Diseño Detallado del Sistema

Para mostrar cómo funciona el sistema se muestran a continuación tres diagramas de secuencia, para indicar el flujo de mensajes principales entre las distintas clases de la aplicación.

En la Figura 19 se muestra la intención funcional de las llamadas que ocurre cuando se seleccionan los archivos originales de un MAOO para que estén disponibles dentro de la aplicación.

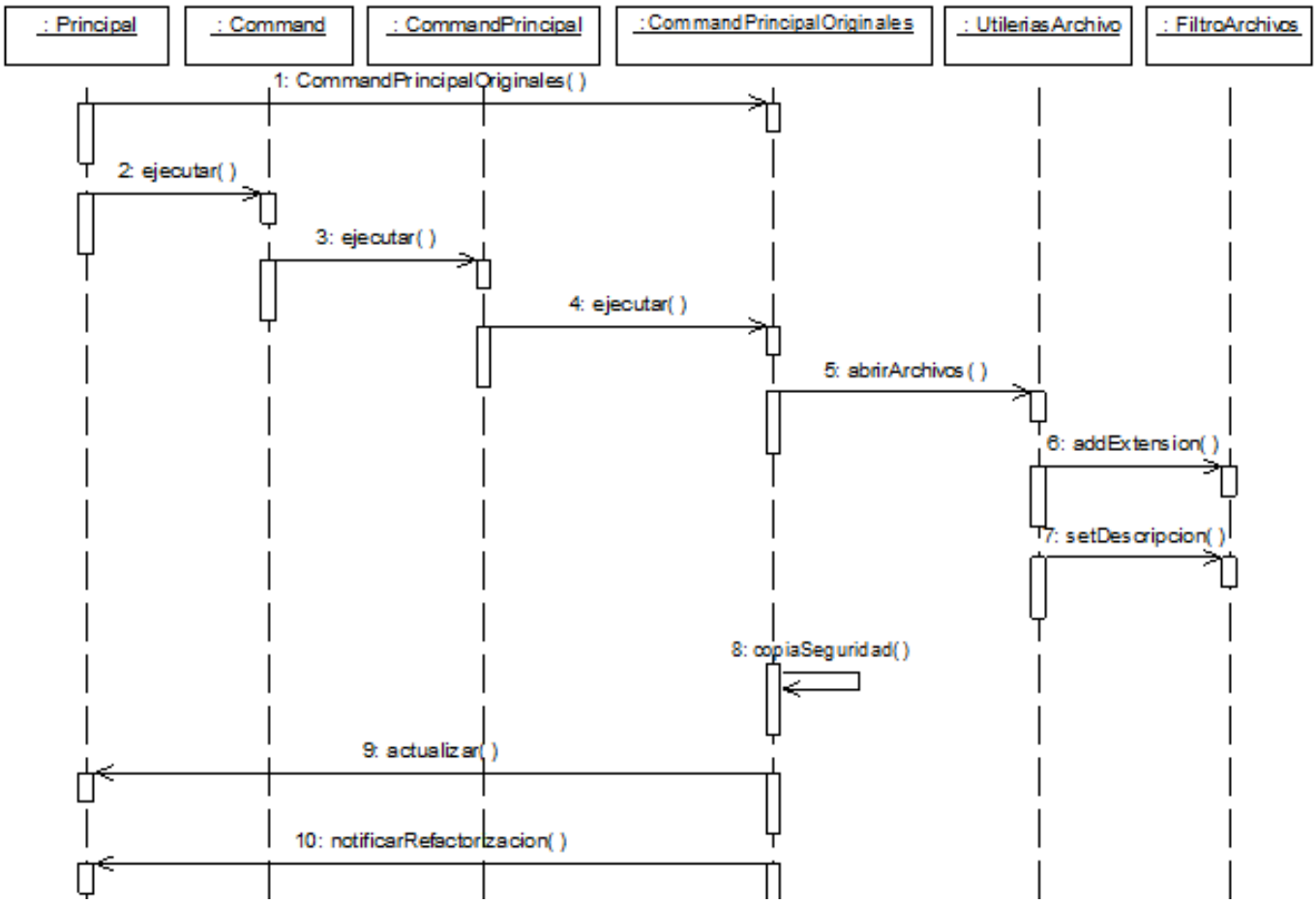


Figura 19. Diagrama de Secuencia para Seleccionar Archivos Originales

En este escenario intervienen las clases que dan soporte a la aplicación, como son las clases '*UtileriasArchivo'*, y '*FiltroArchivos'*. La acción se inicia en la pantalla Principal, cuando el usuario elige el menú de Seleccionar Archivos Originales, lo cual crea un objeto de tipo '*CommandPrincipalOriginales'*. Esta clase está dentro de la jerarquía de comandos, y a través de su clase base se llama al método ejecutar (). La acción de este método consiste en abrir un cuadro de diálogo para seleccionar archivos con código en Java, crear opcionalmente la copia de seguridad del MAOO e inicializar el sistema. En la Figura 20 se muestra el flujo de mensajes cuando se elige calcular la métrica V-DINO.

Esta secuencia de mensajes es iniciada en la pantalla Principal, cuando se elige la opción de Calcular Métrica V-DINO. Se crea una instancia de la pantalla 'FrameMetricaDINO', a través del comando CommandPrincipalMetricaDINO, utilizando el método instanciar (), el cual es parte del patrón '*Singleton*'. Esta pantalla cuenta con las acciones realizar el análisis de código y calcular la métrica para una clase abstracta.

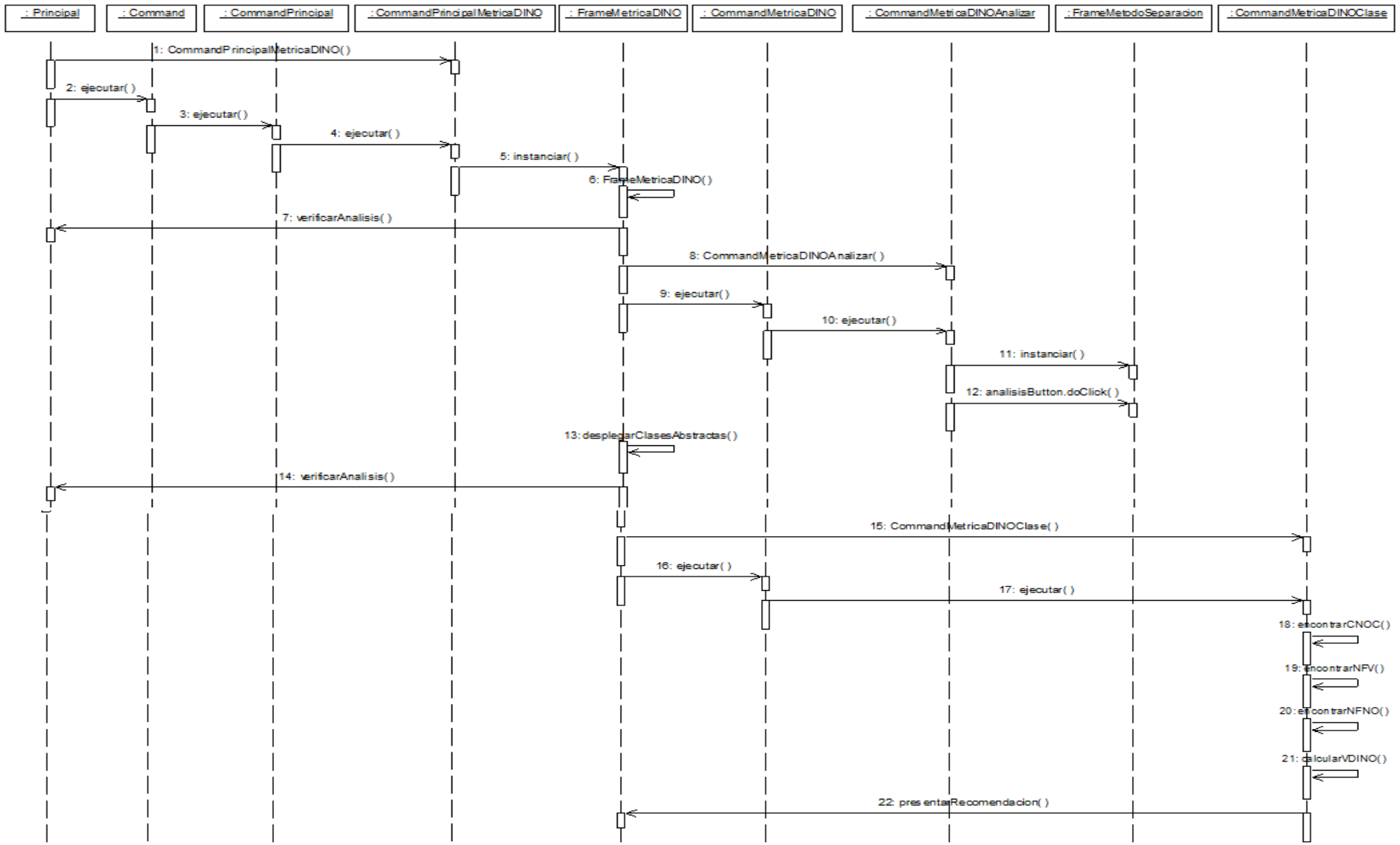


Figura 20. Diagrama de Secuencia para Calcular la Métrica V-DINO

Primeramente, se debe realizar el análisis del código del MAOO seleccionado, utilizando el comando *CommandMetricaDINOAnalizar*. Este comando simplemente llama al análisis del método de Separación de Interfaces y se muestran las clases abstractas obtenidas del análisis. Posteriormente se calcula la métrica con el comando *CommandMetricaDINOClase*, obteniendo los valores que intervienen en la métrica y presentando una recomendación de acuerdo al valor calculado de V-DINO. Para evitar trabajo extra, se lleva un registro de cuando el análisis del código fuente ya ha sido realizado, esto es para no hacerlo de nuevo, o para realizarlo en caso de ser necesario. El análisis de código fuente se realiza cuando se han seleccionado nuevos archivos originales, o cuando ha sido ejecutado un método de refactorización y fueron modificados los archivos.

En la Figura 21 se muestra el flujo de mensajes surgidos a partir de que el usuario elige la opción de *Método de Separación de Interfaces*.

De la misma forma que se creó la pantalla de la métrica se crea la pantalla *FrameMetodoSeparacion*. En caso de que no haya sido realizado antes, se realiza el análisis del código original del MAOO, utilizando para ello al analizador realizado en ANTLR representado por la clase '*CPPParser*'. El análisis arroja la información necesaria y suficiente para realizar la refactorización. El método es ejecutado por cada clase abstracta, verificando primero si es posible Re-factorizar, validando las precondiciones del método.

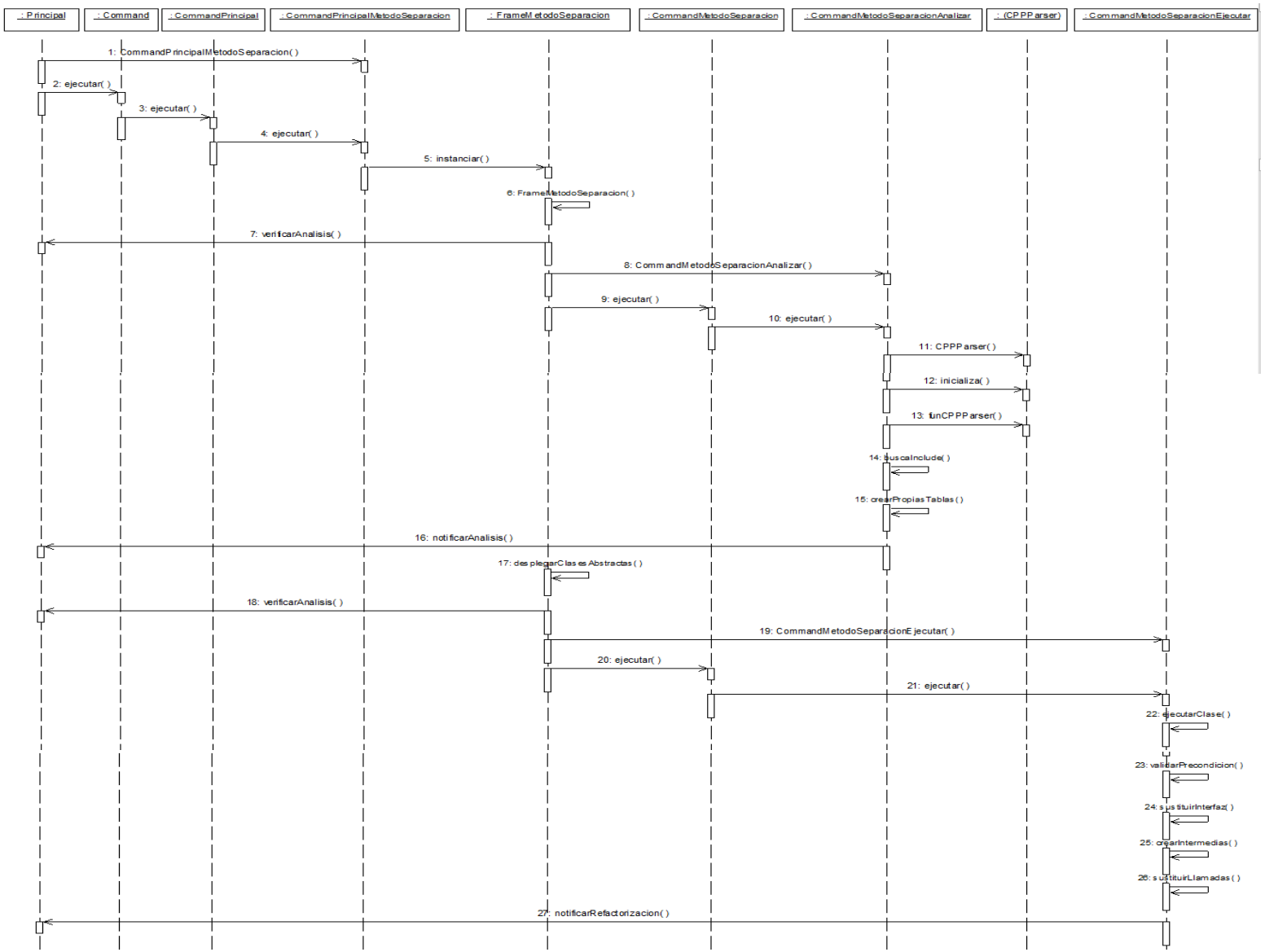


Figura 21. Diagrama de Secuencia para Re-factorizar

Si las precondiciones son satisfechas, se procede a re-factorizar, sustituyendo las interfaces con firmas iguales, creando clases intermedias y sustituyendo las llamadas a interfaces. Al finalizar este proceso, se hace la notificación a los demás métodos y métricas que los archivos originales han cambiado, para que actualicen sus análisis.

Dentro de la herramienta son tres los procesos importantes: el análisis del código fuente de un MAOO, el cálculo de la métrica V-DINO y la aplicación del método de refactorización por Separación de Interfaces. A continuación, se muestran los diagramas de actividades para cada uno de estos procesos.

En la Figura 22 se muestra el diagrama de actividades producto del método *ejecutar ()* de la clase *CommandMetodoSeparacionAnalizar*.

Primeramente, para cada archivo del MAOO original se realiza el proceso de análisis léxico y sintáctico, utilizando el analizador creado en ANTLR, el cual obtiene la información relevante del archivo, en cuanto a clases, métodos, objetos, herencias y llamadas a métodos.

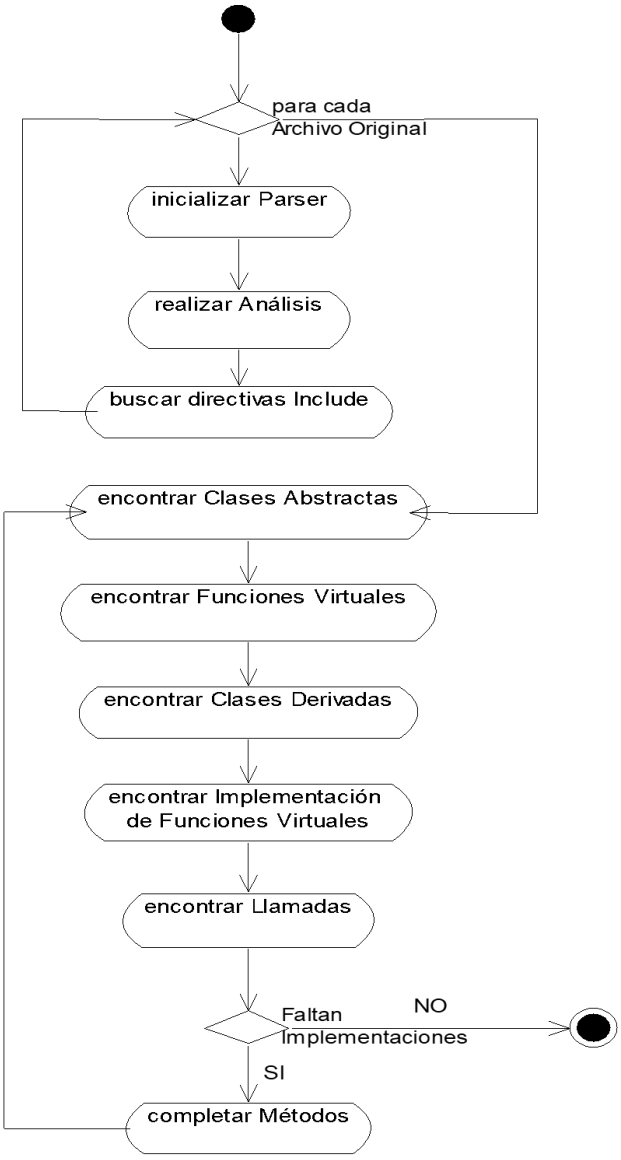


Figura 22. Diagrama de Actividades del Proceso de Análisis

La segunda parte del análisis se realiza cuando ya han sido analizados todos los archivos, y entonces se procede a llenar la estructura de datos encontrando las clases abstractas y funciones abstractas, las clases derivadas de las abstractas, las implementaciones de funciones virtuales en clases derivadas y las llamadas a funciones virtuales. Existe un proceso alterno que puede realizarse si no existen algunas implementaciones de funciones abstractas, y entonces se crean los registros de estas funciones de forma que se pueda complementar la estructura de datos. Este proceso alterno ocurre cuando una clase derivada no implementa alguna función abstracta, y por tanto, esta clase derivada también se considera abstracta.

En la Figura 23 se muestra el diagrama de actividades producto del método *ejecutar ()* de la clase *CommandMetricaDINOClase*.

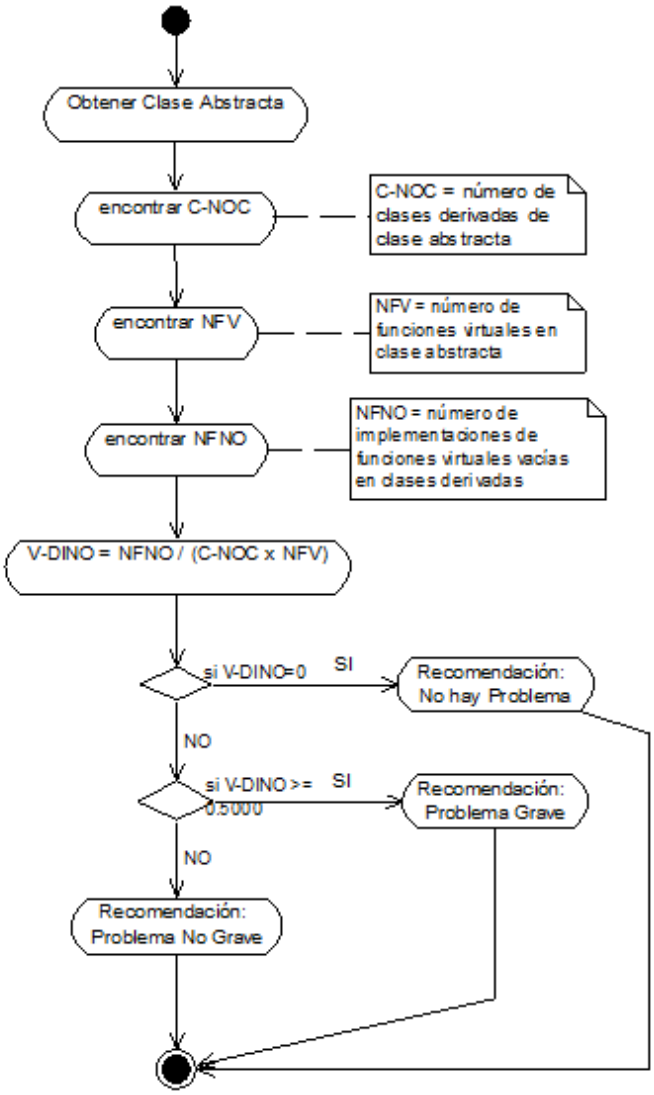


Figura 23. Diagrama de Actividades del Cálculo de la Métrica

En este diagrama se muestra como se calculan los valores de la métrica, y posteriormente se realiza la recomendación de acuerdo al valor calculado. Si el valor es cero, se informa que no hay problema; si el valor es menor a 0.5000, se considera que el problema no es grave porque es muy probable que no se cumplan las precondiciones y cuando es mayor o igual a este valor sí se recomienda al usuario realizar la refactorización.

En la Figura 24 se muestra el diagrama de actividades producto del método ejecutar () de la clase 'CommandMetodoSeparacionEjecutar'.

Para llevar a cabo la refactorización sobre un árbol jerárquico en específico, es decir, para una clase abstracta con sus clases derivadas, primeramente, se encuentran las funciones abstractas para realizar la validación de las precondiciones. Las precondiciones consisten en encontrar funciones con firmas iguales y que sean mutuamente excluyentes, como fue explicado anteriormente.

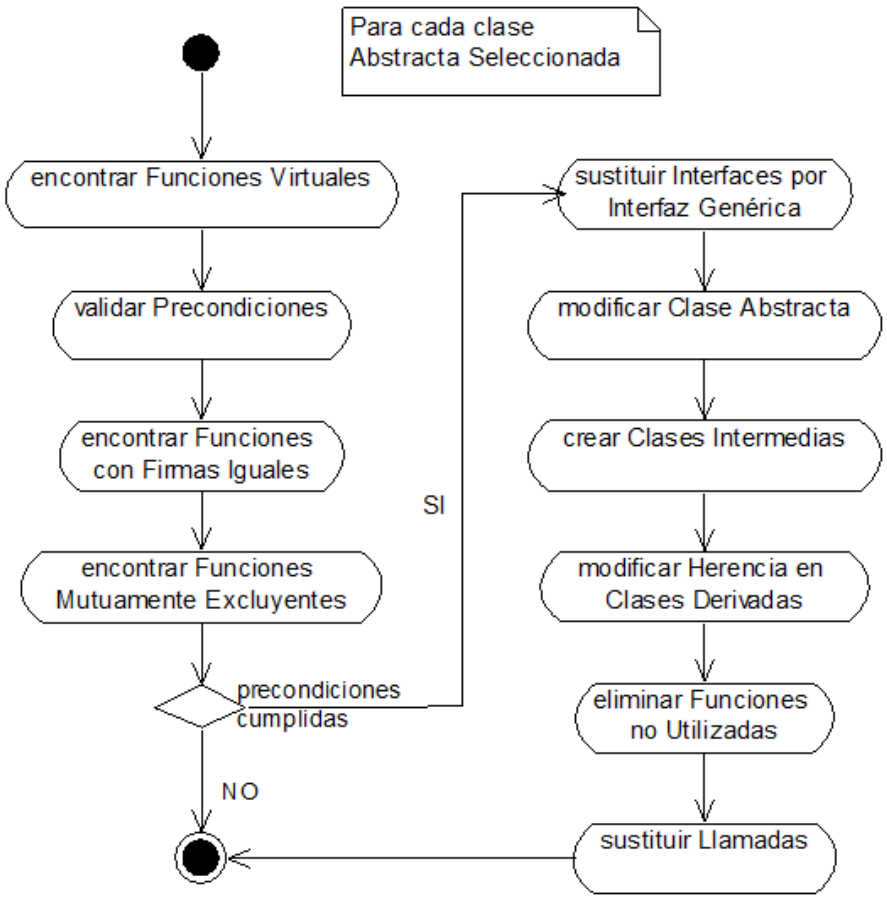


Figura 24. Diagrama de Actividades de la Refactorización

Si las precondiciones son cumplidas, entonces se procede a realizar la refactorización, sustituyendo las funciones con firmas iguales por una interfaz genérica, la cual es derivada de la clase abstracta. Para cada interfaz sustituida se crea una clase intermedia. A las clases derivadas se les cambia la relación de herencia hacia las clases intermedias y se les eliminan las funciones no utilizadas.

5.4. Análisis del Código Fuente

Para realizar el análisis léxico y sintáctico de los archivos de un MAOO, y para implementar las acciones semánticas asociadas al código reconocido se está empleando el generador de analizadores ANTLR, con una gramática para el lenguaje Java.

ANTLR (ANother Tool for Language Recognition) proporciona la capacidad para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes, a partir de las descripciones gramaticales de los mismos (las cuales contienen acciones semánticas).

ANTLR puede reconocer léxicos, analizadores, analizadores de árbol y analizadores combinados de lexer. Los analizadores pueden generar,

automáticamente, árboles de análisis sintáctico o árboles de sintaxis abstractos que pueden procesarse aún más con analizadores de árbol. ANTLR proporciona una única notación coherente para especificar lexers, analizadores y analizadores de árbol.

De forma predeterminada, ANTLR lee una gramática y genera un reconocedor para el lenguaje definido por esa gramática (es decir, un programa que lee una secuencia de entrada y genera un error si la secuencia de entrada no se ajusta a la sintaxis especificada por la gramática). Si no hay errores de sintaxis, la acción predeterminada es simplemente salir sin imprimir ningún mensaje. Para hacer algo útil con el lenguaje, se pueden adjuntar acciones a elementos gramaticales en la gramática. Estas acciones están escritas en el lenguaje de programación en el que se genera el reconocedor. Cuando se genera el reconocedor, las acciones se incrustan en el código fuente del reconocedor en los puntos correspondientes. Las acciones se pueden utilizar para crear y verificar tablas de símbolos y para emitir instrucciones en un idioma de destino, en el caso de un compilador.

Aparte de los reconocedores léxicos y los analizadores, ANTLR se puede utilizar para generar analizadores de árbol. Estos son reconocedores que procesan árboles de sintaxis abstractos que pueden ser generados automáticamente por analizadores

5.5. Herramienta SR2 Refactoring

El sistema que integra la herramienta para resolver el problema de dependencia por interfaces no utilizadas se denomina “SR2 Refactoring” y forma parte del proyecto “SR2: Reingeniería de Software Legado para Reuso” [7]. La pantalla principal de la aplicación se observa en la Figura 25.

El proyecto SR2 consta de otros métodos de refactorización aparte del descrito en este documento de tesis, y por ello fue necesario crear una interfaz que englobara a estos otros métodos de refactorización, los cuales son: Método de Adaptación de Interfaces [37], Método de Reducción de Acoplamiento [15], ésta implementa también la medición de la métrica COF (Coupling Factor) [38] y Método de Re-factorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces que implementa la métrica V-dino[19]. Aparte el trabajo descrito en esta tesis agrega el Método de Métodos de Re-factorización de código java con interfaces y abstracciones incorrectas y la Métrica V-DINO para lenguaje Java.

Como estos métodos de refactorización no serán los únicos en el proyecto, se diseñó la interfaz de tal manera que pueda soportar otras pantallas y métricas, y para ello se utilizaron los patrones de diseño ‘*Singleton’* y ‘*Command’* del catálogo de Gamma [2]. Esta arquitectura del sistema se explicó en la Sección “Diseño del Sistema”.

Otro método que será implementado en la herramienta SR2-Refactoring será el Método de Re-factorización de código para reducir el

acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos. [39]

La pantalla cuenta con seis menús: *Abrir* para seleccionar archivos de origen, cambiar de sesión de usuario y para salir del sistema, *Métricas* para seleccionar la métrica V-DINO o la métrica Coherencia, *Métodos de Refactorización* para seleccionar el método disponible, *Comparación* para comparar los archivos originales con los Re-factorizados, *Usuarios* para manejar los tipos de acceso y usuarios, y *Ayuda* para la información del sistema.

Para autenticar usuarios, ya sea cuando se entra por primera vez al sistema o cuando se cambia de sesión de usuario, se utiliza la pantalla mostrada en la Figura 26.

En esta pantalla se selecciona a uno de los usuarios registrados en el sistema en la lista *Usuario*, escribiendo su nombre de usuario, *Login*, y su clave de acceso, *Password*. Se tienen tres oportunidades para escribir la clave correcta y entrar al sistema. De acuerdo al tipo de acceso que tenga especificado el usuario, se habilitan los menús que tiene derecho a usar. En esta pantalla también se puede modificar la clave de acceso, usando el botón de editar y escribiendo la nueva clave dos veces.

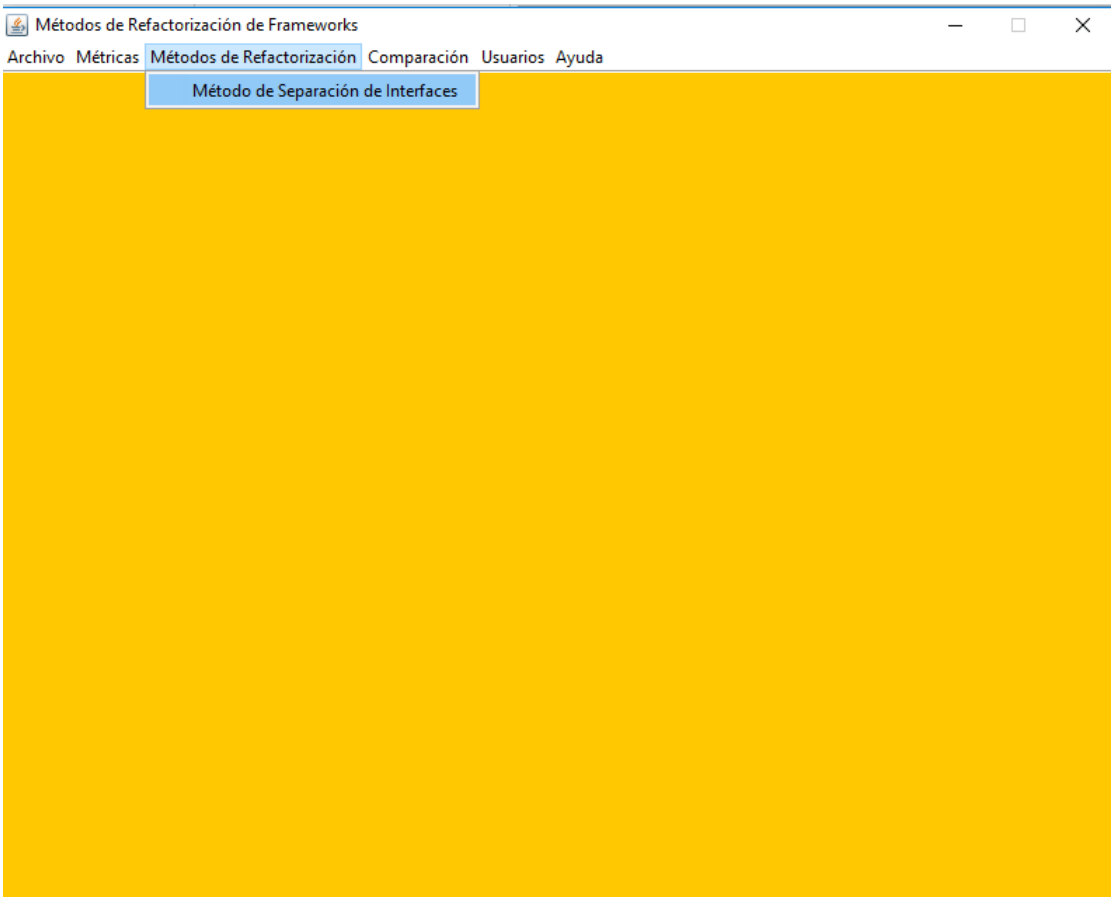


Figura 25: Pantalla principal del SR2 Re-factoring.

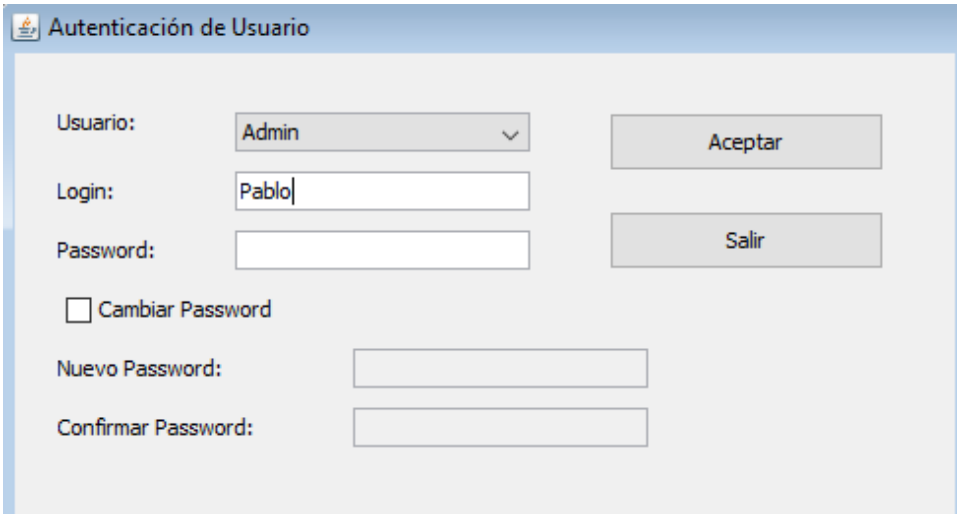


Figura 26: Pantalla de Autenticación de Usuarios

Para seleccionar los archivos originales, se cuenta con el cuadro de diálogo “Seleccionar Archivos Originales”, mostrado en la Figura 27, que permite abrir múltiples archivos fuente de Java con las extensiones *“.java”*. Después de seleccionar los archivos o agregar más a la selección actual, el sistema pregunta si se desea realizar copia de seguridad de estos archivos seleccionados, para conservar la versión original del MAOO ya que los archivos seleccionados pueden ser modificados por los métodos de refactorización.

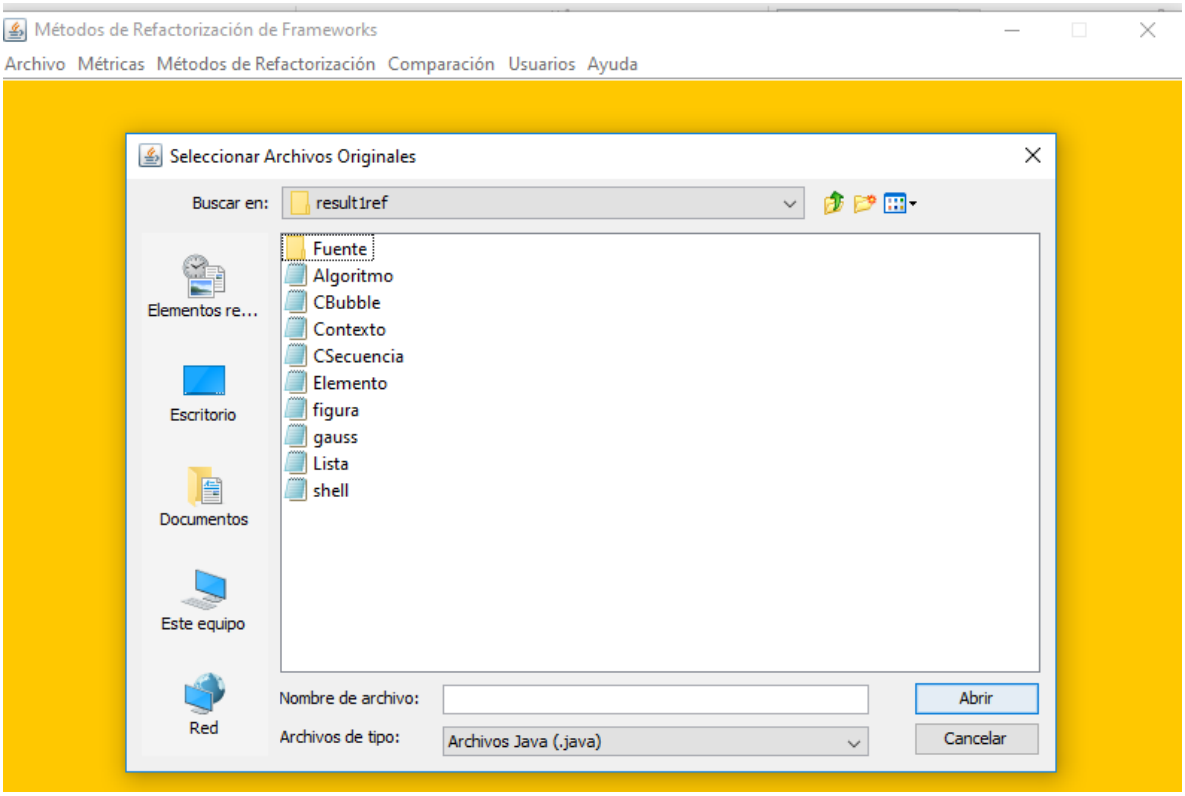


Figura 27: Cuadro de Diálogo para Abrir Múltiples Archivos

Se pueden elegir todos los archivos de un MAOO, y se muestran los archivos con extensiones válidas para Java. Con el botón *Abrir* se elegirán los archivos seleccionados para trabajar con ellos, si se elige *Cancelar* no se modifica la selección y se conserva el estado del sistema.

La Figura 28 muestra la pantalla para el cálculo de la métrica V-DINO, que sirve como referencia al usuario para decidir si realiza o no la refactorización de su código fuente. Esta pantalla tiene el botón que realiza la ejecución del análisis sintáctico del código fuente.

La pantalla cuenta con el botón *Realizar Análisis* para ejecutar el análisis sintáctico del código, después de este proceso se activa la lista desplegable *Clases Abstractas* que muestra a las clases con funciones abtractas. Al seleccionar una clase se muestran los valores de *C-NOC*, *NFV*, *NFNO* y *V-DINO*, aparte se muestra una *Recomendación* dependiendo del valor de la métrica.

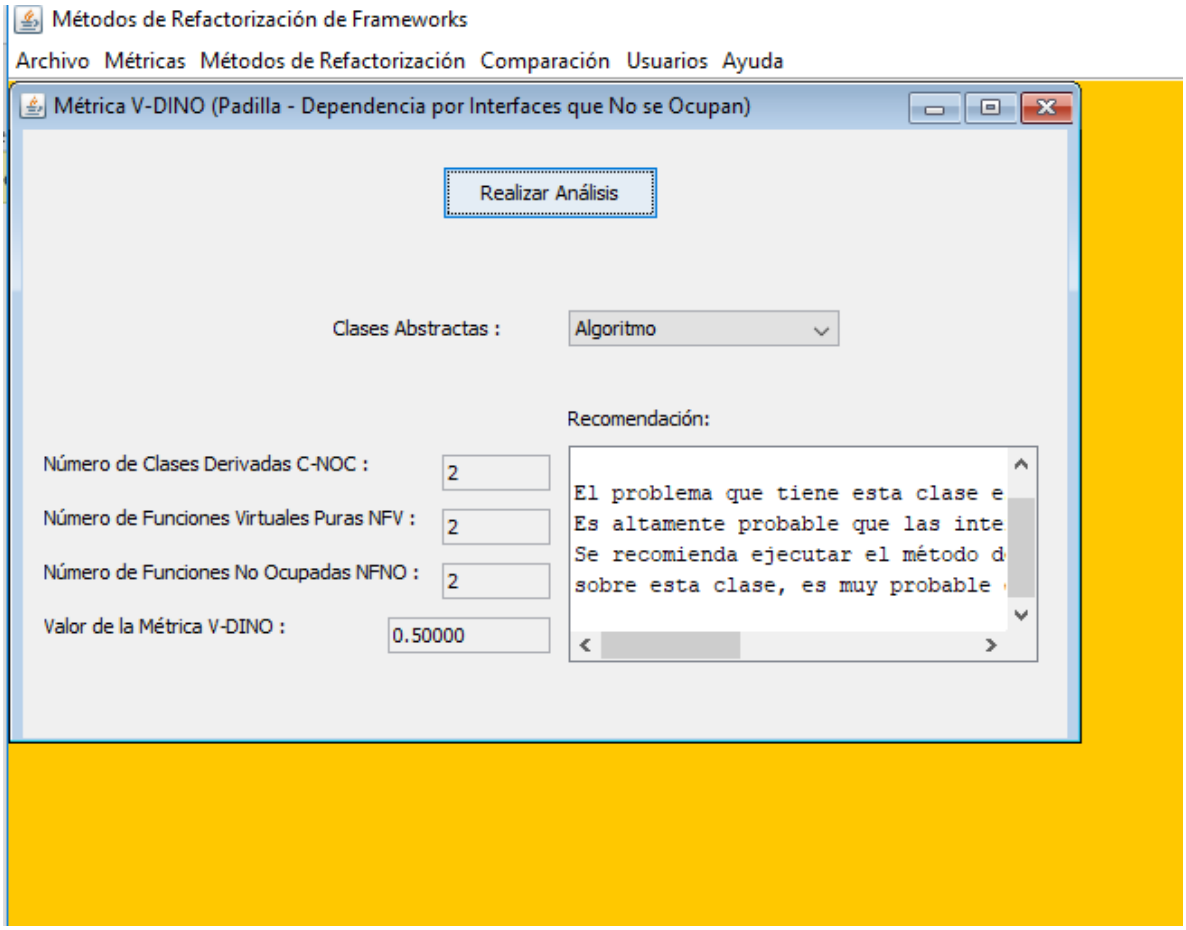


Figura 28: Pantalla para el cálculo de la métrica V-DINO

La Figura 29 muestra la pantalla del método de separación, que sirve para modificar el MAOO de origen del usuario según el método de refactorización. Esta pantalla también cuenta con el botón que Analizar el código del análisis sintáctico, permitiendo poder Re-factorizar sin pasar por la pantalla de la Figura 29, si el proceso ya fue hecho en dicha pantalla, este no se vuelve a realizar.

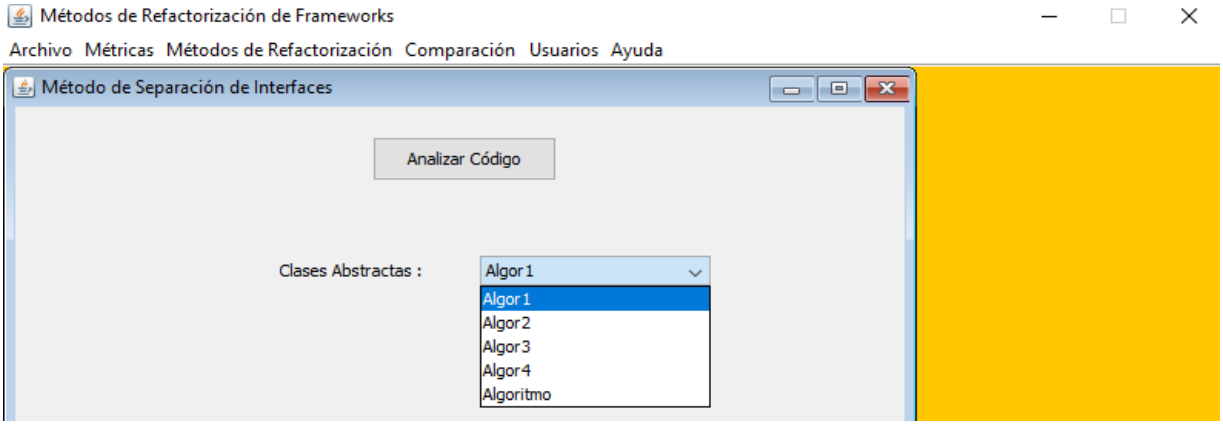


Figura 29: Pantalla del Método de separación de interfaces

Cuenta con el botón *Analizar Código* para ejecutar el análisis sintáctico del código y llenar la base de datos, después de este proceso se activa la lista desplegable *Clases Abstractas* que muestra a las clases para hacer la refactorización sobre todas las clases abstractas. El botón Analizar código será el encargado de realizar la refactorización automática, después de verificar que se cumplan las precondiciones del método y solicitar datos al usuario con respecto a las nuevas clases e interfaces.

Esta interfaz fue enriquecida con una pantalla para comparar archivos, para verificar de manera visual los cambios hechos a los archivos originales, pero esto sólo está disponible si se eligió crear la copia de seguridad de los archivos fuente en la pantalla de la Figura 27. Dicha pantalla se muestra en la Figura 30.

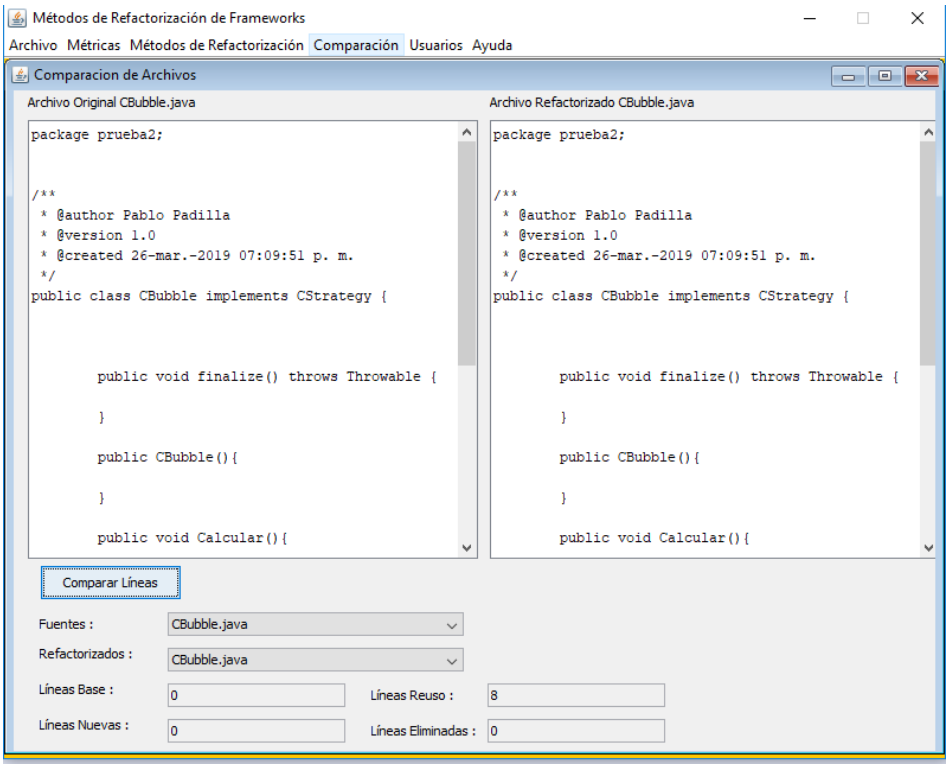


Figura 30. Pantalla de Comparación de Archivos

Capítulo 6) EVALUACIÓN EXPERIMENTAL

6.1. Caso de Prueba 1. MAOO de Listas Doblemente Ligadas

Aquí se analiza un pequeño MAOO de algoritmos de ordenación y búsqueda por medio de listas doblemente ligadas de tipos de datos básicos. Este MAOO se presentó en [Sant04].
El programa cuenta con la funcionalidad adicional de hacer ordenación y búsqueda de datos, utilizando los algoritmos de ordenación por burbuja y búsqueda secuencial, respectivamente.
También se le colocó al MAOO una clase CContexto para funcionar como la interfaz del MAOO. La arquitectura del MAOO se muestra en la Figura 31 con toda la jerarquía de clases, excepto el archivo que funciona como cliente e invoca a la interfaz.
En el diagrama se observa que existe una clase interfaz Algoritmo, la cual cuenta con dos interfaces Busqueda( ) y Ordena( ), que a su vez son implementadas por las clases derivadas CBubble y CSecuenc.
Debido a la naturaleza del MAOO, existen funciones con implementación vacía.

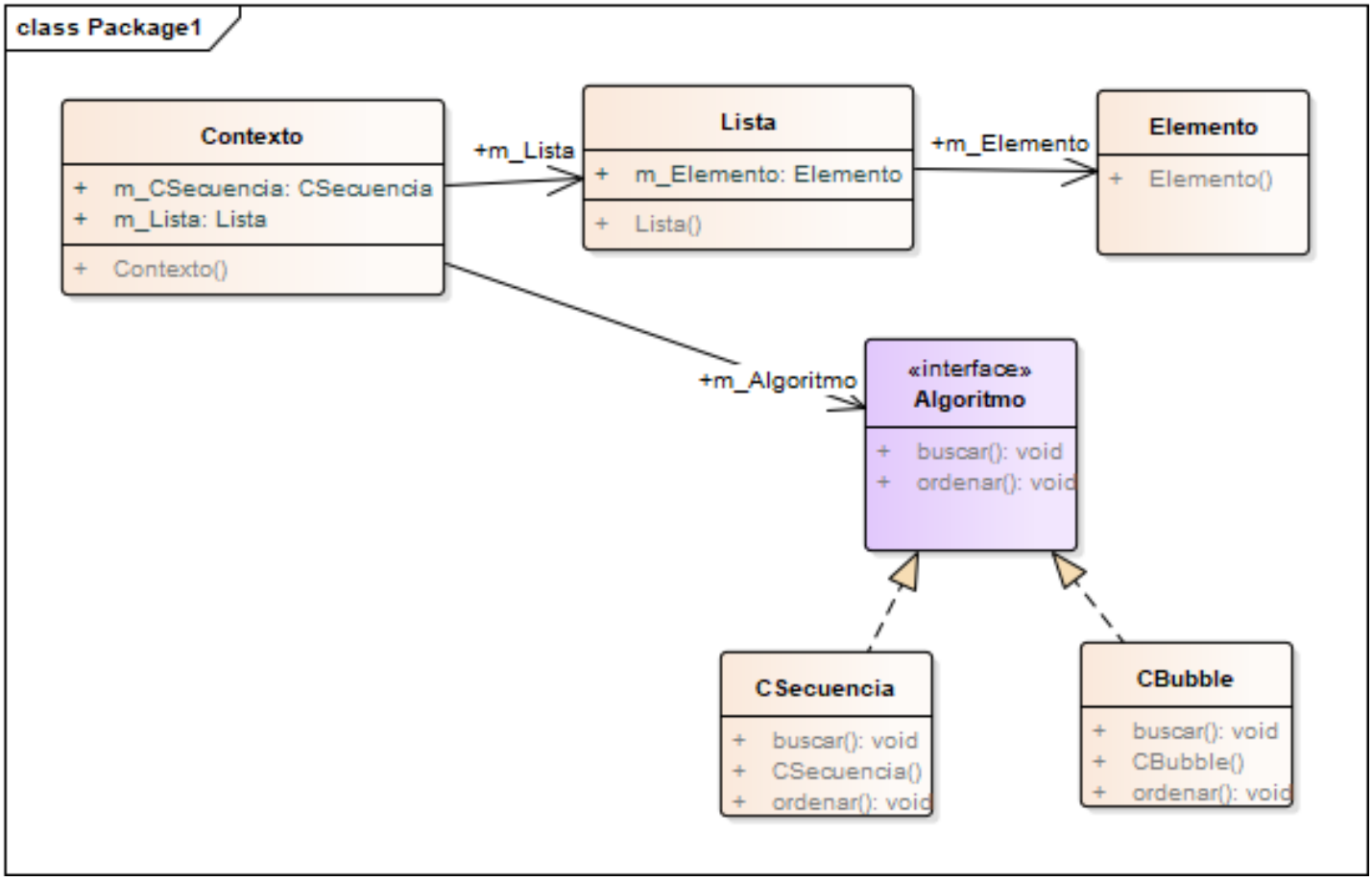


Figura 31: Diagrama de Clases del Caso de Prueba 1. MAOO Original

La clase ***CBubble*** es la encargada del algoritmo de ordenación, por tanto implementa un código a la función abstracta ***Busqueda( )*** ya que no la necesita. Lo mismo pasa con la clase ***CSecuenc***, la encargada del algoritmo de búsqueda, que implementa vacía a la interfaz ***Ordena ()*** porque esta no es necesaria en esta clase.La validación de las precondiciones del método de Separación de Interfaces determina que las dos interfaces tienen la misma firma, debido a que reciben los mismos parámetros y tienen el mismo valor de retorno; y son mutuamente excluyentes, debido a que ninguna clase derivada implementa con código no nulo a ambas interfaces; por tanto es posible fusionar estas interfaces y aplicar la refactorización.

El cálculo de la métrica V-DINO arroja el siguiente resultado, analizando a la clase ***Algoritmo***: C-NOC = 2, NFV = 2, NFNO = 2, y V-DINO = 2 / (2 x 2) = 0.50000. Este caso está justamente en el límite de la métrica V-DINO para considerarlos como grave. El caso presentado es uno de los casos más comunes del problema de dependencia de interfaces, el cual fue mostrado en la Tabla 1.

Debido a que se cumplen las precondiciones y el análisis de V-DINO sugiere Re-factorizar, se procedió a utilizar el método de refactorización por Separación de Interfaces para mejorar la arquitectura del MAOO.

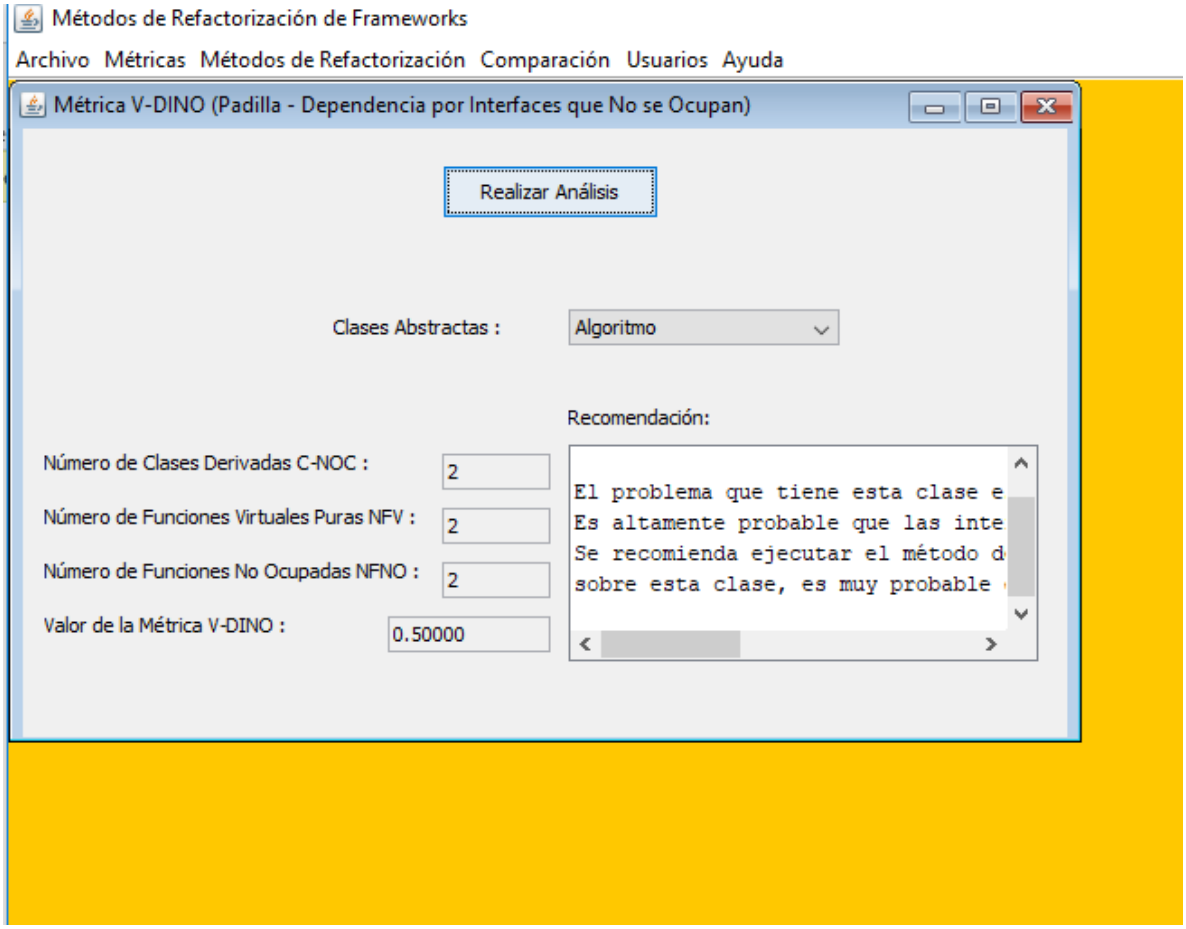


Figura 32: Cálculo de la métrica V-Dino al MAOO original 1

Al aplicar el método de refactorización se obtiene la arquitectura mostrada en la Figura 33.

Primeramente, las dos funciones abstractas de la clase ´Algoritmo´ problemáticas fueron sustituidas por una única función abstracta *AlgoritmoInterfaz()*, que cuenta con la misma firma que las interfaces anteriores. La función abstracta *Buscar ()* es trasladada hacia la recién creada clase intermedia CAlgor1, la cual será ahora la clase base de CSecuenc. La función abstracta Ordenar () es trasladada hacia la también nueva clase CAlgor2, la cual es la clase base de CBubble.

En ambas clases derivadas CBubble y CSecuenc la eliminan las implementaciones con código que no eran necesarias. Por último, en la clase CContexto se sustituyen las llamadas que había hacia las funciones originales por una llamada hacia la nueva función abstracta.
Se procede a corroborar la mejora del MAOO aplicando la métrica V-DINO al MAOO resultante. Ahora existen tres clases abstractas, Algoritmo, CAlgor1 y CAlgor2, por tanto se calcula V-DINO para cada una de ellas. Para la clase Algoritmo: C-NOC = 2, NFV = 1, NFNO = 0 y V-DINO = 0.0000. Para CAlgor1 y CAlgor2: C-NOC = 1, NFV = 1, NFNO = 0 y V-DINO = 0.0000.

Se puede observar, por los valores de la métrica, que el problema de dependencia por interfaces no utilizadas ha quedado totalmente resuelto en este MAOO.

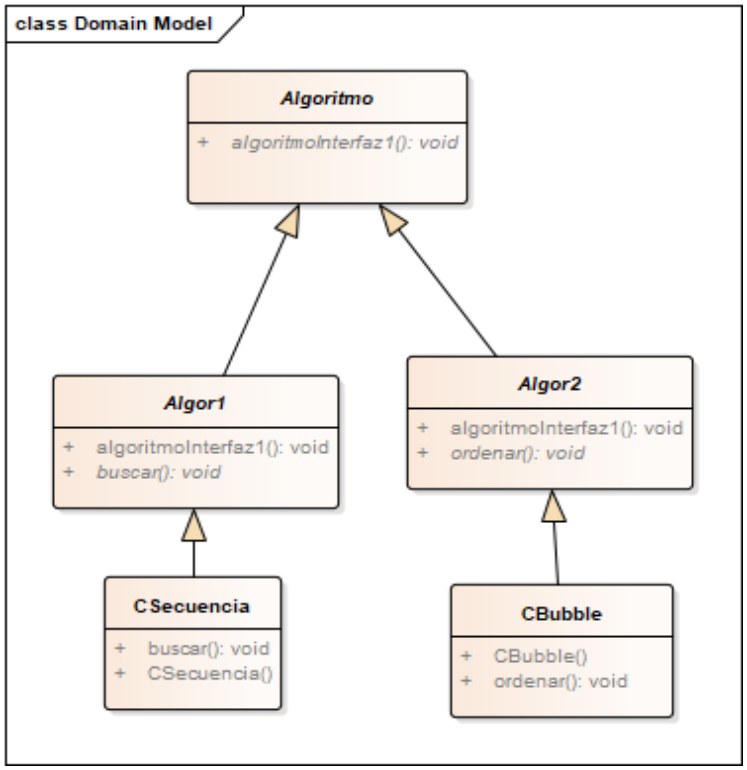


Figura 33: Diagrama de Clases del Caso de Prueba 1. MAOO Re-factorizado

En la siguiente figura 34 se muestra el valor de la métrica V-Dino para la Clase Algoritmo donde presenta que ya no existe el problema de dependencia de interfaces dando un valor de 0.00000.

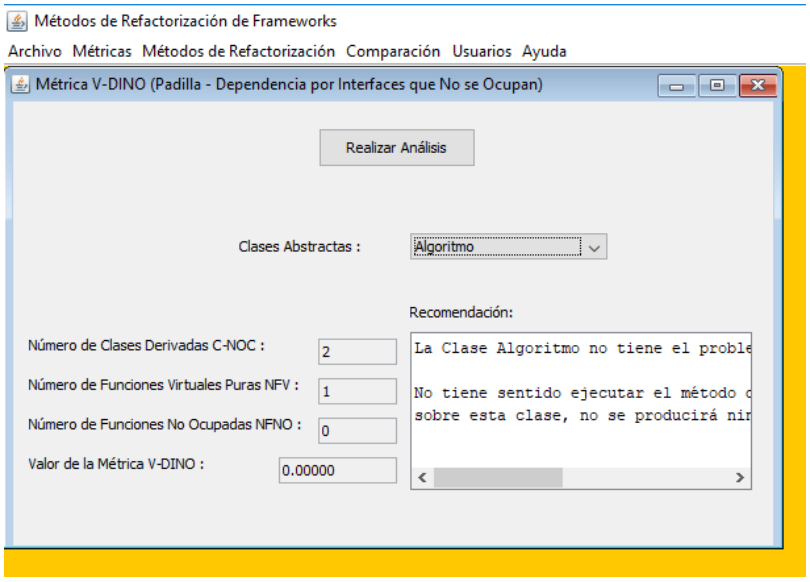


Figura 34. Cálculo de la métrica V-Dino al MAOO Re-factorizado 1

6.2. Caso de Prueba 2. MAOO de Estadística

Aquí se analiza un MAOO del dominio de la estadística, el cual fue presentado en [Sant04b].

Este MAOO proporciona algunas funciones de cálculos estadísticos, utilizando tres listas doblemente ligadas para almacenar la serie de números sobre los que operan las funciones, también se cuenta con dos matrices para manejar operaciones adicionales.

La funcionalidad total que tiene el MAOO completo es el cálculo de medidas de tendencia central, dispersiones, distribuciones, regresiones y correlaciones, y como auxiliares se tienen ordenación de datos y solución de sistemas de ecuaciones lineales. El diagrama de clases completo se muestra en la Figura 35.

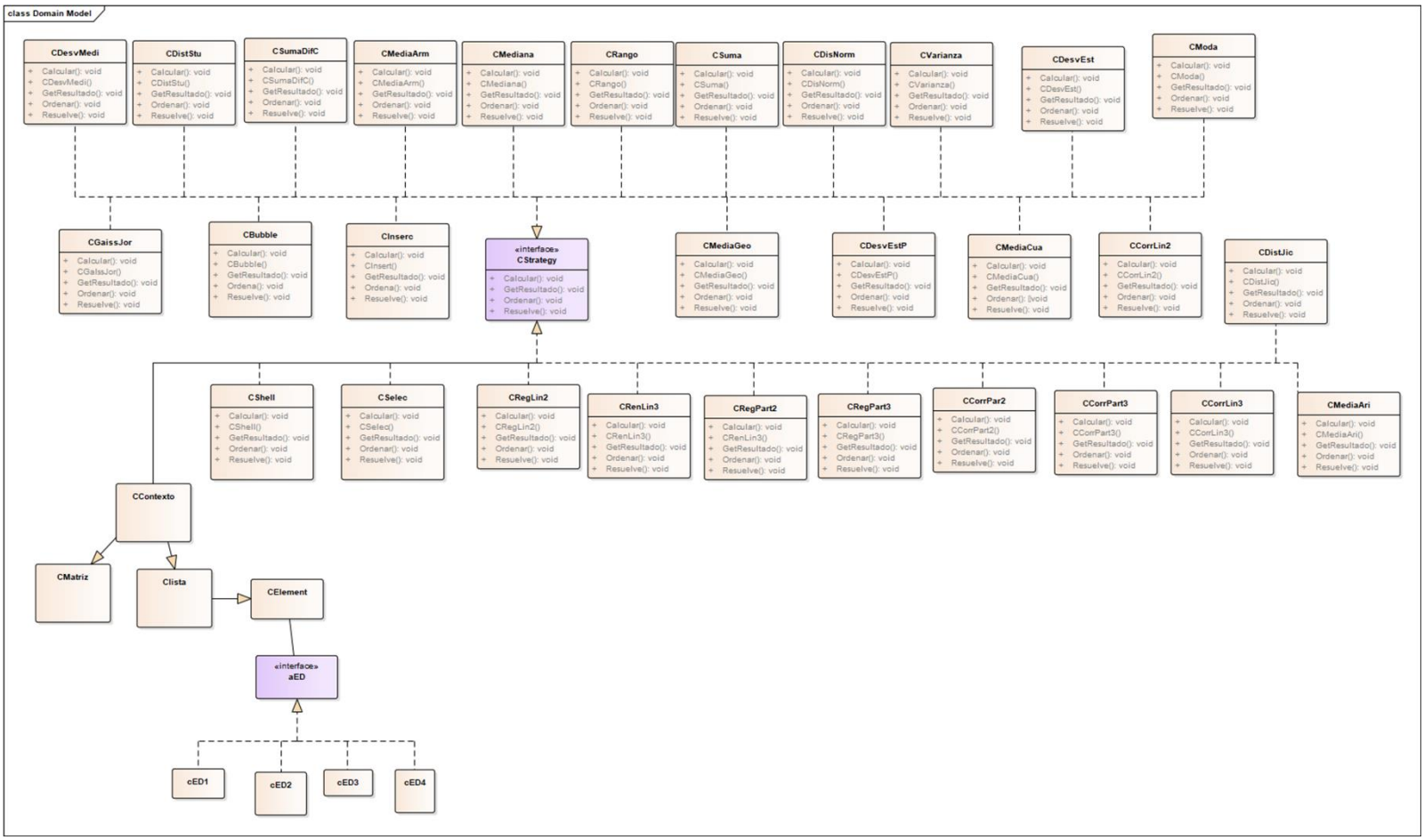


Figura 35: Diagrama de Clases del Caso de Prueba 2. MAOO Original

En este MAOO existen dos clases abstractas, CStrategy y aED. La Clase CStrategy cuenta con 4 funciones abstractas GetResultado (), Resuelve (), Ordena () y Calcula (), las cuales son implementadas por 29 clases derivadas, mostradas en la Figura 35. La clase aED cuenta con 2 funciones GetED() y SetED( ), implementadas con código en las clases cED1, cED2, cED3 y cED4.

La validación de las precondiciones del método de Separación de Interfaces determina que las funciones abstractas *Ordena ()*, *Resuelve ()* y *Calcula ()* tienen la misma firma, y son mutuamente excluyentes, por tanto, es posible fusionar estas interfaces y aplicar la refactorización. Como la función abstracta *GetResultado ()* no cumple con las precondiciones, no será posible resolver el problema para esta interfaz.

El cálculo de la métrica V-DINO arroja el siguiente resultado para la clase *CStrategy*: C-NOC = 29, NFV = 4, NFNO = 87, y V-DINO = 87 / (29 x 4) = 0.75000. Para la clase *aED*: C-NOC = 4, NFV = 0, NFNO = 0, y V-DINO = 0. 0000.Como se muestra en la figura 36.

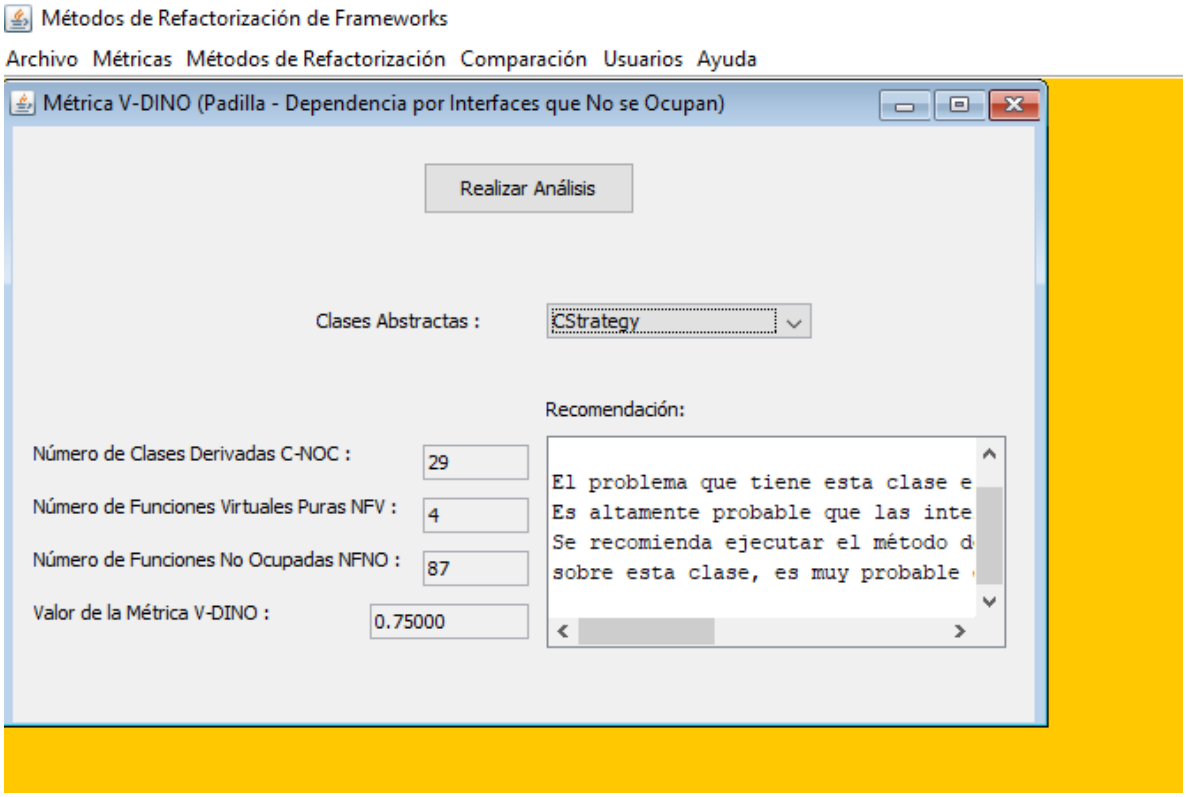


Figura 36: Cálculo de la métrica V-Dino al MAOO original 2

El caso de *CStrategy* está dentro de los límites de la métrica V-DINO para considerarlo como grave. Debido a que se cumplen las precondiciones y el análisis de V-DINO sugiere Re-factorizar. Se procedió a utilizar el método de refactorización por Separación de Interfaces para mejorar la arquitectura del MAOO.

Al aplicar el método de refactorización se obtiene la arquitectura mostrada en la Figura 37.

Primeramente, las tres funciones abstractas problemáticas fueron sustituidas por una única función abstracta *AlgoritmoInterfaz ()*, que cuenta con la misma firma que las interfaces anteriores. La función *Calcula ()* es trasladada hacia la recién creada clase intermedia CAlgor1, la cual será ahora la clase base de las clases CCorrLin2, CCorrLin3, CCorrPart2, CCorrPart3, CDesvEst, CDesvEstP, CDesvMedi, CDisNorm, CDisJic, CDistStu, CMediaAri, CMediaArm, CMediaCua, CMediaGeo,CMediana,CModa, CRango, CRegLin2, CRegLin3, CRegPart2, CRegPart3, CSuma, CSumaDifC y CVarianza. La función Resuelve () es trasladada hacia la también nueva clase CAlgor2, la cual es la clase base de la clase CGaissJor. La función Ordena () es traslada hacia la también nueva clase CAlgor3, la cual es la clase base de las clases CBubble, CInsert, CSelec y CShell.

En ambas clases derivadas se eliminan las implementaciones que no eran necesarias. Por último, en la clase CContexto se sustituyen las llamadas que había hacia las interfaces originales por una llamada hacia la nueva interfaz genérica. En el MAOO sólo existían tres llamadas.

Se procede a corroborar la mejora del MAOO aplicando la métrica V-DINO al MAOO resultante. Ahora existen cuatro clases abstractas, CStrategy, CAlgor1, CAlgor2 y CAlgor3 por tanto se calcula V-DINO para cada una de ellas. Para la clase CStrategy: C-NOC = 3, NFV = 2, NFNO = 0 y V-DINO = 0.00. Para CAlgor1: C-NOC =24, NFV = 1, NFNO = 0 y V-DINO = 0.0000. Para CAlgor2: C-NOC =1, NFV = 1, NFNO = 0 y V-DINO = 0.0000 y Para CAlgor3: C-NOC =4, NFV = 1, NFNO = 0 y V-DINO = 0.0000. Que se muestra en la figura 38.

Se puede observar, por los valores de la métrica, que el problema de dependencia por interfaces no utilizadas ha quedado resuelto en este MAOO.

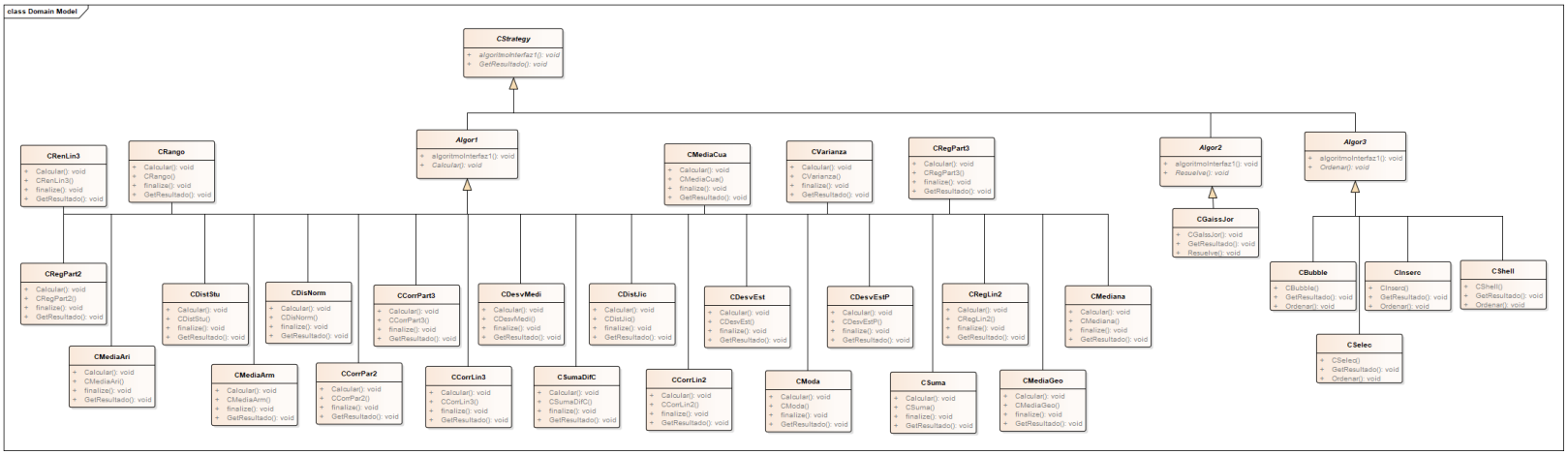


Figura 37: Diagrama de Clases del Caso de Prueba 2. MAOO Re-factorizado

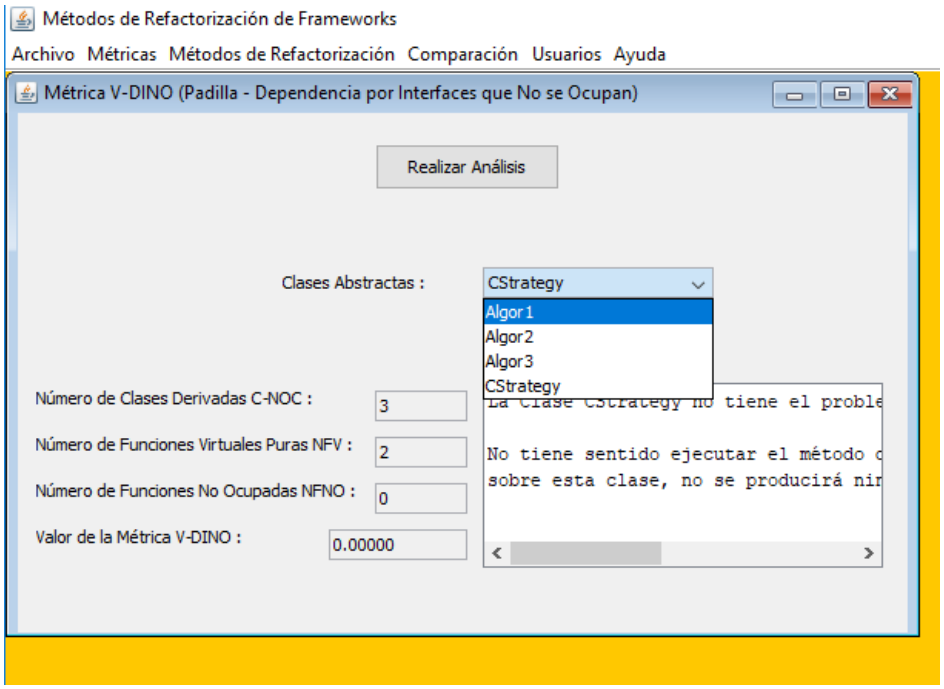


Figura 38. Cálculo de la métrica V-Dino al MAOO Re-factorizado 2

6.3. Caso de Prueba 3. MAOO del sistema de PSP cenidet

Aquí se analizan algunas clases del MAOO del sistema de PSP cenidet, el diagrama de clases completo se muestra en la Figura 39.

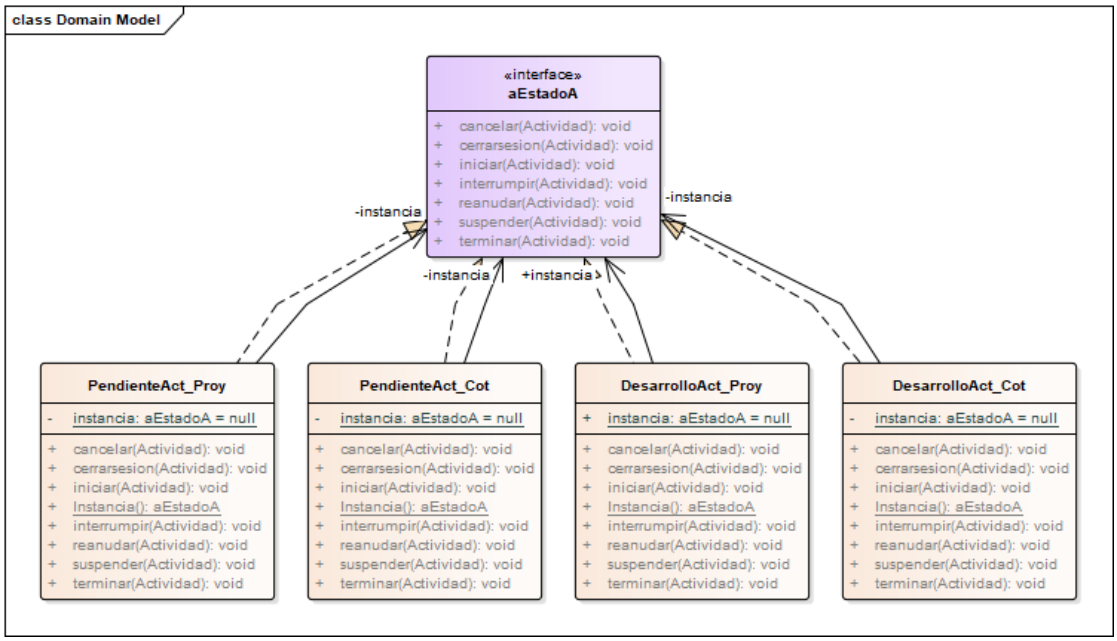


Figura 39: Diagrama de Clases del Caso de Prueba 3. MAOO Original

Este MAOO cuenta con una clase abstracta “aEstadoA”. La Clase aEstadoA cuenta con las funciones cancelar(Actividad), cerrarsesion(Actividad), iniciar (Actividad), interrumpir(Actividad), reanudar(Actividad), suspender(Actividad) y terminar (Actividad), las cuales son implementadas por las 4 clases derivadas que se muestran en la figura 39.

El cálculo de la métrica V-DINO arroja el siguiente resultado para la clase aEstadoA: C-NOC = 4, NFV = 7, NFNO = 11, y V-DINO = 11 / (4 x 7) = 0.39286.

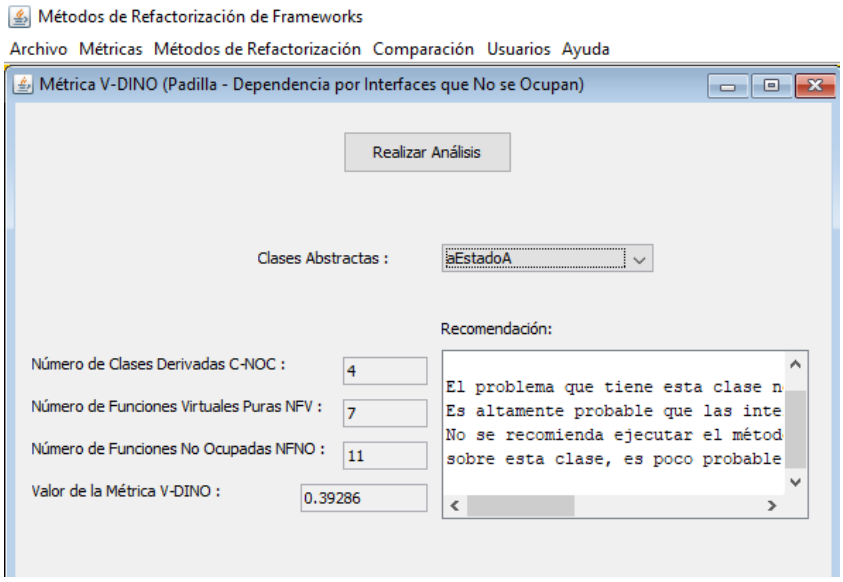


Figura 40: Cálculo de la métrica V-Dino al MAOO original 3

El caso de *aEstadoA* está fuera de los límites de la métrica V-DINO para considerarlo como grave. Se cumplen las precondiciones, pero el análisis de V-DINO no recomienda utilizar el método de refactorización por Separación de Interfaces para mejorar la arquitectura del MAOO.
Por lo tanto, queda al criterio del usuario, la decisión de utilizar o no el método. De todos modos, procedió a utilizar ese método de Re-factorización.

Al aplicar el método de refactorización se obtiene la arquitectura mostrada en la Figura 41.

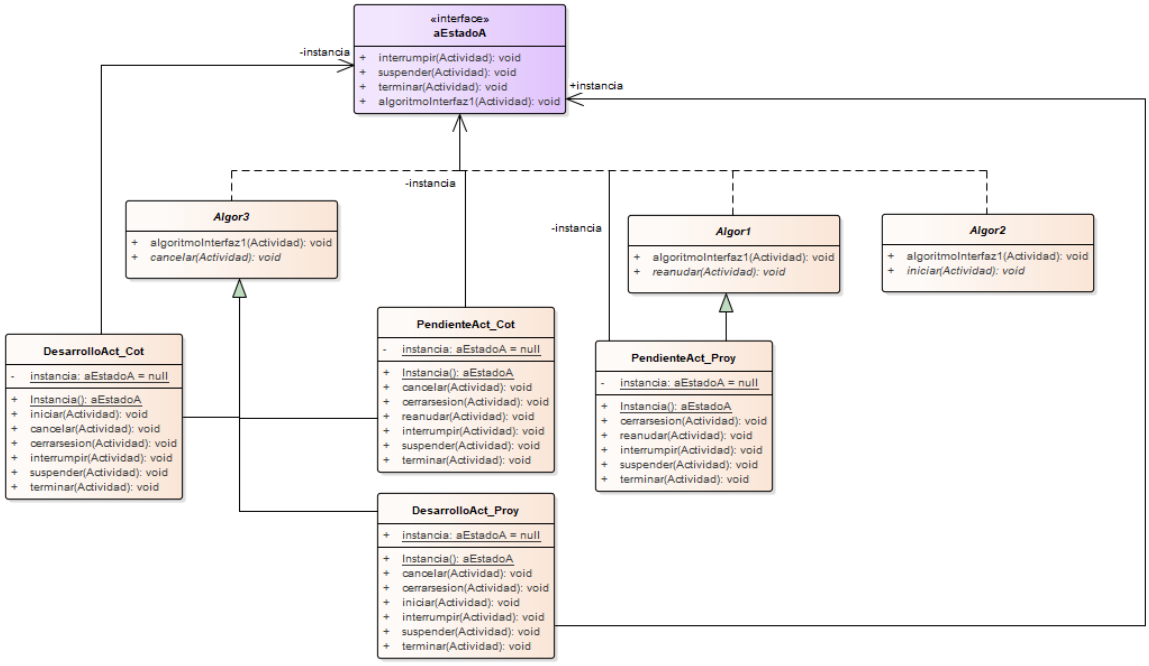


Figura 41: Diagrama de Clases del Caso de Prueba 3. MAOO Re-factorizado

Se procede a corroborar la mejora de este MAOO aplicando la métrica V-DINO. Ahora existen cuatro clases abstractas, aEstadoA, CAlgor1, CAlgor2 y CAlgor3 por tanto se calcula V-DINO para cada una de ellas. Para la clase aEstadoA: C-NOC = 3, NFV = 4, NFNO = 0 y V-DINO = 0.00. Para CAlgor1: C-NOC =1, NFV = 1, NFNO = 0 y V-DINO = 0.0000. Para CAlgor2: C-NOC =0, NFV = 1, NFNO = 0 y V-DINO = NaN y Para CAlgor3: C-NOC =3, NFV = 1, NFNO = 0 y V-DINO = 0.0000. Que se muestra en la figura 42.

Se puede observar, por los valores de la métrica, que el problema de dependencia por interfaces no utilizadas se ha resuelto para este MAOO.

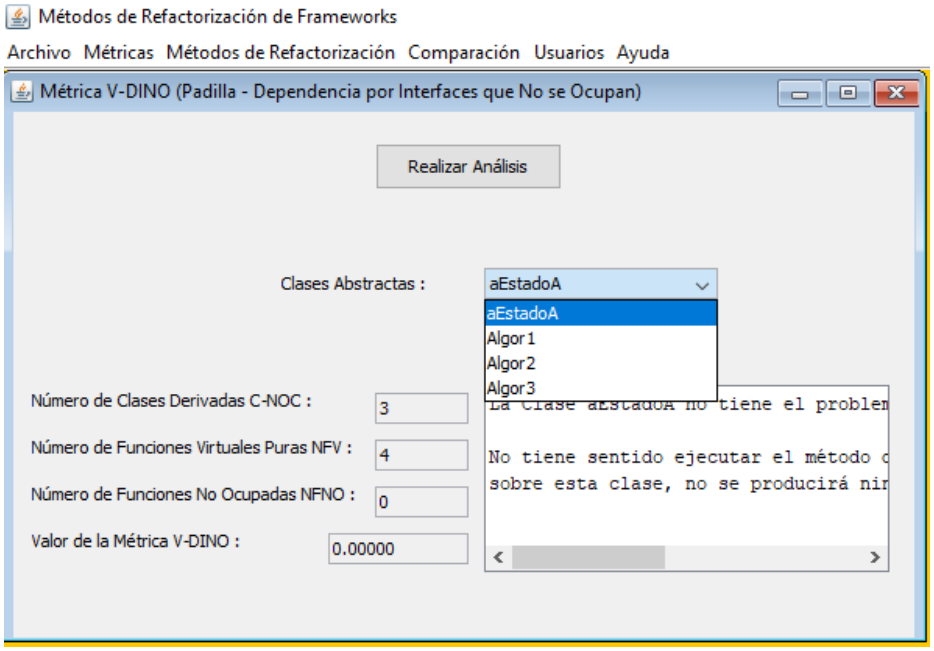


Figura 42. Cálculo de la métrica V-Dino al MAOO Re-factorizado 3

Capítulo 7) CONCLUSIONES Y TRABAJO FUTURO

Conclusiones

El objetivo de todo el proyecto SR2 es el mejor aprovechamiento del software legado existente, para producir nuevas aplicaciones de mejor calidad. Para lograr esto, el SR2 cuenta con varios métodos de refactorización para eliminar el código indeseable y obtener mejores cualidades para reuso y control de la evolución que adquiere el software que es conforme con los principios de diseño O.O.

El proyecto SR2 se encuentra actualmente en la fase de extensión, que consiste en implementar métodos de refactorización que ayudan a mejorar la arquitectura de MAOO's, no con la finalidad de mejorar el funcionamiento actual de los MAOO's, sino para permitir el reuso de los componentes en otras aplicaciones o extender la funcionalidad del MAOO, los cuales son aspectos a considerar para mejorar la calidad de un producto de software.

Al utilizar la herramienta desarrollada en esta tesis, en conjunto con el resto del proyecto SR2, se puede hacer reuso de software legado, obteniendo código con mayor calidad, ya que está basado en múltiples principios y patrones de diseño.

Las aportaciones de esta tesis son: Un método para refactorizar y una métrica V-DINO

1.- Método de refactorización denominado Separación de Interfaces.

- El método tiene la intención de reducir el número de dependencias entre clases debido a la implementación de funciones abstractas no utilizadas. Con esto se busca incrementar las cualidades de reuso y extendibilidad de los MAOO's.

- Mediante la refactorización propuesta, los programas que eran rígidos y frágiles por depender de abstracciones que no ocupan ahora serán flexibles y robustos, y su mantenimiento será más fácil. Son flexibles porque se permite cambiar el comportamiento de objetos en tiempo de ejecución, y es extensible a nuevas funcionalidades; son robustos porque ya no se producen errores en otras partes del MAOO al modificar una interfaz o una clase, debido a que se reducen las dependencias por la separación de las interfaces.

- Una vez que el MAOO se encuentra refactorizado, es más fácil de entender su estructura e intención de cada clase, ya que las clases sólo implementarán las interfaces que necesitan, y quedarán agrupadas por su intención en una misma clase intermedia.

- Se creó una precondición para el método, que exige que las interfaces que pueden sustituirse deben tener los mismos parámetros de entrada y salida, y estas funciones deben ser mutuamente excluyentes. Esto es debido a que se hace el agrupamiento de clases y una clase no puede pertenecer a dos grupos.

- Aparte de esta precondición, al algoritmo se le agregó la precondición básica de no repetir nombres existentes en las nuevas clases que serán añadidas al sistema, con el fin de que el sistema no falle después de la refactorización.

- También se encontró un uso adicional para el método de Separación de Interfaces, el cual es preparar a un MAOO para ser llevado hacia Servicios y microservicios Web. Esto es debido a que un Servicio Web o microservicio debe ofrecer las mejores cualidades de modularidad, entre estos, ofrecer interfaces bien definidas para ser usadas por los usuarios del servicio, y el respeto a los principios de diseño modular y O.O tal como el principio de sustitución, que es visto por ellos como una caja negra. Por tanto, si un MAOO tiene el problema de dependencia de interfaces, no podrá ser llevado a Servicio Web de forma eficiente debido a que el servicio ofrecerá funciones que llevan a código nulo. Esto fue publicado en [Sant04].

2.- Métrica orientada a objetos denominada V-DINO.

- Debido a que no existe documentación en cuanto a cómo medir y evaluar la dependencia de interfaces debido a implementaciones vacías o nulas, se diseñó e implementó la métrica V-DINO (Valdés – Dependencia por Interfaces que No se Ocupan) para poder tomar una decisión acertada en cuanto a realizar la refactorización o saber hasta qué grado un MAOO tiene el problema.

- Esta métrica puede ser utilizada como una medida de la calidad de un software. En el estándar ISO 9126 [ISO91], el cual fue establecido para caracterizar la calidad del software, hay seis atributos que el software debe cumplir para ser considerado como un software con calidad: funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad. La métrica V-DINO puede medir objetivamente el grado de soporte a cambios flexibilidad o extensibilidad a nuevas funcionalidades, el cual es un factor clave en la mantenibilidad de cualquier software.

3.- Casos de estudio que demuestran la efectividad del método y la métrica.

- Mediante las pruebas experimentales se ha demostrado que es correcta la propuesta de que los patrones de diseño ‘Strategy’ y ‘Template Method’ sirven para separar las interfaces.

- Al aplicar este método de refactorización a un MAOO, se logra reducir la dependencia entre clases abstractas y sus clases derivadas. Con esto se reduce la dependencia por interfaces que las clases derivadas no ocupan, de tal manera que las clases derivadas pueden ser reusadas de forma separada.

- Como resultado de la refactorización, se obtendrán MAOO’s que no requerirán de modificaciones cuando se quiera agregar nueva funcionalidad, respetando así el principio de Abierto / Cerrado [Meye98], y las nuevas clases sólo declararán funciones que requieren, respetando así el principio de Separación de Interfaces del diseño orientado a objetos, única responsabilidad, no repetición y el principio de sustitución [Mart96].

- Sin importar que tan alto o bajo sea el valor de la métrica, ha quedado demostrado que un MAOO sí es mejorado si cumple con las precondiciones establecidas en el algoritmo de Separación de Interfaces.

4.- Herramienta de refactorización denominada SR2 Refactoring.

- Se implementó el método de refactorización de Separación de Interfaces para refactorizar código en lenguaje Java, implementando el analizador

léxico y sintáctico en el metalenguaje ANTLR. Lo mismo se implementó la métrica V-DINO. Para que opere sobre código escrito en Java.

- Para preparar la herramienta para tener varios métodos y métricas se hizo su diseño basado en los patrones de diseño 'Singleton' y 'Command'. En un futuro se pueden añadir fácilmente otros métodos de refactorización y métricas de calidad.

Trabajo Futuro

Lo que resta de realizar en el proyecto SR2 es simplemente agregar más métodos de refactorización al sistema SR2 Refactoring. Existen demasiados métodos propuestos en la literatura consultada ([40], [41] y [42]) pero la mayoría no han sido implementados en herramientas automáticas, por tanto, existe todavía una amplia gama de posibilidades en este aspecto.

Inclusive existen otros patrones de diseño de Gamma [2] que pueden ser implementados para resolver problemas de diseño, no necesariamente utilizando la intención original de los patrones, como fue el caso de los patrones 'Strategy' y 'Template Method', usados en esta tesis. Asimismo, existen otros catálogos de patrones que podrían ser empleados.

En lo concerniente a la herramienta realizada en esta tesis, aún se puede disminuir más la dependencia por interfaces no necesitadas, buscando un método alternativo al de Separación de Interfaces, cuando no se cumplen las precondiciones de este método. Una opción para esto sería implementar el algoritmo sugerido en [40].

Primer trabajo futuro: Es completar la gramática de Java de ANTLR para que reconozca la totalidad estándar del lenguaje Java. Esto ayudará a crear mejores analizadores sintácticos, no sólo para este método, sino para futuros métodos.

Segundo trabajo futuro: Que la herramienta funcione cuando las implementaciones vacías se ubican en las clases base y son heredadas vacías a las clases derivadas.

Tercer trabajo futuro: Que la herramienta separe funciones abstractas cuando los paramentos de estas sean incompatibles (distintos).

El proyecto SR2 también tomará el rumbo de la tecnología de Servicios y microservicios Web, para ofrecer de una manera fácil a los desarrolladores de software las herramientas de transformación y refactorización. De ahí que otro trabajo futuro es la separación de herramientas de transformación, de herramientas de refactorización, como la presentada en esta tesis, hacia Servicios Web, utilizando una metodología como la presentada en [19]. Estos servicios pueden entonces ser publicados en la red, utilizando tecnología como la presentada en [43]. De esta manera se tiene visualizado que los administradores de MAOO's tendrán a su disposición el proyecto SR2.

Anexo A) V-DINO COMO MÉTRICA DE COHESIÓN

Cohesión

El concepto de cohesión ha sido usado con referencia a módulos o sistemas modulares. Mide el grado de unión con el cual características relacionadas de un programa están agrupadas en sistemas o módulos. Se asume que mientras más encapsuladas estén ciertas características relacionadas, el sistema será más confiable y con más fácil mantenimiento [44].

De manera general, las características que debe cumplir la cohesión son las siguientes: Se espera que la cohesión no sea negativa y, más importante, que esté normalizada para que la medición sea independiente del tamaño del sistema modular o módulo. Además, si no hay relaciones internas en un módulo o en todos los módulos del sistema, se espera que la cohesión sea nula para ese módulo o sistema, ya que, si no hay relaciones entre los elementos entonces no existirá evidencia para decir que deben ser encapsulados juntos. Relaciones internas adicionales en un módulo no pueden disminuir el valor de la cohesión, ya que estas relaciones darían mayor evidencia de la dependencia entre los elementos y que deben estar encapsulados juntos. Cuando se fusionan dos o más módulos que no tienen relaciones entre sí, la cohesión no puede incrementarse porque elementos no relacionados están agrupados juntos [45].

A continuación, se muestra la definición formal de la cohesión en un módulo con sus cuatro propiedades, basada en la teoría de conjuntos.

Posteriormente se muestra cómo se cumplen estas propiedades para la métrica V-DINO.

Definición de Cohesión de un Módulo [45]
La cohesión de un módulo m = <E_m,R_m> de un sistema modular MS, donde E_m es el conjunto de los elementos del módulo y R_m es el conjunto de las relaciones entre los elementos, es una función Cohesión(m) caracterizada por las siguientes cuatro propiedades de cohesión.

Propiedad 1 de Cohesión: No negatividad y Normalización [45]
La cohesión de un módulo m = <E_m,R_m> de un sistema modular MS = <E,R,M> se encuentra dentro de un intervalo especificado

$$\text{Cohesión}(m) \in [0,Max].$$

La normalización permite comparaciones significativas entre las cohesiones de diferentes módulos, ya que todas pertenecen al mismo intervalo.

Propiedad 2 de Cohesión: Valor Nulo [45]
La cohesión de un módulo m = <E_m,R_m> de un sistema modular MS = <E,R,M> es nula si R_m está vacío, es decir

$$R_m = \varnothing \Rightarrow \text{Cohesión}(m) = 0.$$

Si no hay relaciones entre los elementos de un módulo, entonces la cohesión del módulo es nula.

Propiedad 3 de Cohesión: Monotonicidad [45]
Sean MS' = <E,R',M'> y MS'' = <E,R'',M''> dos sistemas modulares (con el mismo conjunto de elementos E) de tal manera que existan dos módulos m'= <E_m,R_m'> y m'' = <E_m,R_m''> (con el mismo conjunto de elementos E_m) pertenecientes a M' y M'' respectivamente, de tal forma que R' - R_m' = R'' - R_m'', y R_m' $\subseteq$ R_m''. Entonces,

$$\text{Cohesión}(m') \leq \text{Cohesión}(m'').$$

Al añadir más relaciones entre los elementos de un módulo no se disminuye la cohesión modular.

Propiedad 4 de Cohesión: Módulos Cohesivos [45]
Sean MS = <E,R,M> y MS' = <E,R,M'> dos sistemas modulares (con el mismo sistema <E,R>) de tal forma que M' = M – {m_1,m_2} $\cup$ {m'}, con $m_1 \in$ M, $m_2 \in$ M, m' $\notin$ M, y m' = $m_1 \cup m_2$ (Los dos módulos m_1 y m_2 son reemplazados por el módulo m', unión de m_1 y m_2). Si no existen relaciones entre los elementos que pertenecen a m_1 y m_2, entonces

$$\max \{ \text{Cohesión}(m_1), \text{Cohesión}(m_2) \} \geq \text{Cohesión}(m').$$

La cohesión de un módulo obtenido por unir dos módulos no relacionados no es mayor que la máxima cohesión de los dos módulos originales.

Demostración

Recordando, la fórmula de la métrica V-DINO es:

$$V - DINO = \frac{NFNO}{C - NOC \; x \; NFV}$$

donde NFNO es el número de funciones con implementación nula en las clases derivadas, C-NOC es el número de clases derivadas y NFV es el número de funciones virtuales o funciones abstractas.

Para hacer la demostración de estas propiedades para la métrica V-DINO se harán las siguientes consideraciones:

- Se considera como sistema modular MS a cualquier MAOO que tenga clases abstractas.
- Se considerarán como módulos m a diferentes árboles jerárquicos de clases que inicien con una clase abstracta, es decir, que tenga mínimo una función virtual pura o una función abstracta, y que tenga al menos una clase derivada.
- Se considera como conjunto de elementos E al conjunto de funciones virtuales. Esto quiere decir que en E se encuentran las funciones virtuales NFV de la métrica V-DINO y cada implementación, ya sea con código implementado con código nulo o sin él, de estas funciones en las clases derivadas C-NOC de la métrica V-DINO, por tanto |E| = C-NOC x NFV.
- Se considera como conjunto de relaciones R entre un elemento E' y un elemento E'' al conjunto de relaciones de tipo: E' es implementada sin código o con código nulo en E''. Esto quiere decir que se considera como R a cada implementación sin código considerada en el NFNO de la métrica V-DINO, por tanto |R| = NFNO.
- Por último, se considera como función Cohesión(m) a la métrica V-DINO(m).

Propiedad 1 de V-Dino: No Negatividad y Normalización

Primeramente, se demostrará la propiedad de no negatividad. Como se especificó previamente, la métrica aplica para un árbol jerárquico de clases que inician por una clase base (tiene al menos una función virtual o abstracta), que cuenta con al menos una clase derivada, por tanto, C-NOC > 0 y NFV > 0. NFNO es el total de implementaciones vacías, por tanto, NFNO ≥ 0.

Como todos los miembros de la fórmula de V-DINO son positivos, es obvio que V-DINO ≥ 0, por tanto, se demuestra la primera parte de la propiedad 1.

Para demostrar la normalización, se tiene que el máximo número posible de implementaciones vacías es C-NOC x NFV, por tanto, NFNO ≤ (C-NOC x NFV). De ahí se tiene que

$$V\text{-}DINO(m) \in [0,1],$$

es decir, la métrica V-DINO está en el intervalo de 0 a 1 para todos los casos.

Propiedad 2 de V-Dino: Valor nulo

Esta propiedad pide que cuando el conjunto R de relaciones está vacío, el valor de cohesión debe ser nulo. Para la métrica V-DINO se considera que R = ∅ cuando no hay relaciones en donde una función virtual o abstracta con código nulo es implementada vacía en una clase derivada, es decir, NFNO = 0.

Entonces se tiene que V-DINO = 0 / (C-NOC x NFV) = 0, lo cual demuestra que

$$R = \varnothing \Rightarrow V\text{-}DINO(m) = 0.$$

Propiedad 3 de V-Dino: Monotonicidad

Esta propiedad especifica que la cohesión no debe disminuir si se aumenta el número de relaciones a un módulo. Para la métrica V-DINO se dirá entonces que el valor de la métrica no debe disminuir si en un árbol jerárquico de clases se agregan implementaciones vacías de una interfaz ya existente.

Es fácil demostrar esto, supongamos que se tiene un árbol jerárquico m', con un valor de la métrica V-DINO como sigue:

$$V\text{-}DINO(m') = NFNO' / (C\text{-}NOC \times NFV),$$

y que se modifica al árbol m' agregando implementaciones vacías, lo cual genera el árbol m'', con un valor de la métrica V-DINO como sigue:

$$V\text{-}DINO(m'') = NFNO'' / (C\text{-}NOC \times NFV),$$

donde se tiene que NFNO'' > NFNO' porque aumentaron las funciones que no se ocupan.

Comparando ambos valores de la métrica para m' y m'', como los denominadores son iguales, se tiene que:

$$NFNO' < NFNO'' \Rightarrow V\text{-}DINO(m') < V\text{-}DINO(m'')$$

y queda demostrado que V-DINO(m'') no disminuye con respecto a V-DINO(m') cuando se agregan relaciones a m''.

Propiedad 4 de V-Dino: Módulos Cohesivos

La propiedad especifica que la cohesión de un módulo generado por la fusión de dos módulos no debe aumentar con respecto al máximo valor de cohesión de los dos módulos por separado, siempre y cuando estos módulos no tengan relaciones entre ellos.

Traducido para el caso de la métrica V-DINO, se dirá que el valor de la métrica para un árbol jerárquico formado por la unión de dos árboles no debe ser mayor al máximo del valor de la métrica para esos árboles, siempre que no haya implementaciones vacías entre los dos árboles.

Para dejar lo anterior más claro, se explica el concepto de unión para árboles de clases. Al unir dos árboles lo que se tendrá es que una nueva clase

base tendrá las funciones abstractas de ambas clases bases que encabezaban a cada árbol, también se tiene que este nuevo árbol tendrá las clases derivadas de ambos árboles. Cada clase derivada tendrá las implementaciones con o sin código de todas las funciones abstractas.

Por la restricción de la propiedad, que no debe haber relaciones entre los dos árboles, se considera que no habrá implementaciones vacías entre las clases derivadas de un árbol con las funciones abstractas del otro árbol. Esto es muy raro y poco probable que suceda en la realidad, pero la demostración debe hacerse bajo esta condición.

Entonces se considera que existen dos árboles jerárquicos m_1 y m_2 parte del mismo sistema MS, es decir $m_1 \subseteq MS$ y $m_2 \subseteq MS$, donde

$$V\text{-}DINO(m_1) = NFNO_1 / (C\text{-}NOC_1 \times NFV_1),$$
$$V\text{-}DINO(m_2) = NFNO_2 / (C\text{-}NOC_2 \times NFV_2).$$

Ahora se generará un nuevo árbol m' el cual será la unión de los árboles m_1 y m_2, es decir: $m' = m_1 \cup m_2$. El valor de la métrica V-DINO para este árbol es

$$V\text{-}DINO(m') = NFNO' / (C\text{-}NOC' \times NFV'),$$

donde:

$NFNO' = NFNO_1 \cup NFNO_2 = NFNO_1 + NFNO_2 - NFNO_{comunes}$,
$C\text{-}NOC' = C\text{-}NOC_1 \cup C\text{-}NOC_2 = C\text{-}NOC_1 + C\text{-}NOC_2 - C\text{-}NOC_{comunes}$ y
$NFV' = NFV_1 \cup NFV_2 = NFV_1 + NFV_2 - NFV_{comunes}$.

Esto es, si los árboles m_1 y m_2 tenían clases base o clases derivadas comunes, estas no estarán repetidas en m'. Para el peor de los casos en donde los árboles no tengan funciones y clases derivadas comunes, simplemente se tiene que

$NFNO' = NFNO_1 + NFNO_2$,
$C\text{-}NOC' = C\text{-}NOC_1 + C\text{-}NOC_2$ y
$NFV' = NFV_1 + NFV_2$.

Si se sustituyen estos términos en la fórmula de V-DINO para m', y después de hacer las operaciones correspondientes, se tiene

$$V - DINO = \frac{NFNO_1 + NFNO_2}{(C - NOC_1 x NFV_1) + (C - NOC_2 x NFV_1) + (C - NOC_1 x NFV_2) + (C - NOC_2 x NFV_2)}$$

Como se puede observar, el denominador crece con respecto a los denominadores de cada árbol, también lo hace el numerador, pero se sabe por la propiedad 1 de la cohesión que el aumento del numerador no será mayor al aumento del denominador, por eso se puede considerar que

$$\max \{V\text{-}DINO(m_1), V\text{-}DINO(m_2)\} \geq V\text{-}DINO(m').$$

Para cerciorarse que son correctas estas demostraciones, se tomaron los MAOO's de prueba del Capítulo 6, y en todos los casos se pudo verificar que fue correcta la interpretación de V-DINO como métrica de cohesión.

Anexo B) V-DINO COMO ESCALA ORDINAL

Escala Ordinal

La escala ordinal va más allá de la escala nominal, la cual sólo sirve para determinar si un objeto es igual o diferente a otro, colocándoles un identificador.

En este anexo se demuestra como la métrica propuesta V-DINO cumple las condiciones de la escala ordinal, la cual establece un orden entre los objetos medidos.

Definición de Escala Ordinal [35]

Supongamos que (**P**, •≥) es un sistema relacional empírico, donde **P** es un conjunto contable no vacío de diagramas de flujo y donde •≥ es una relación binaria en **P**. Luego existe una función u: **P** -> $\Re$, con

$$P1 \bullet\geq P2 \Leftrightarrow u(P1) \geq u(P2),$$

para todo P1, P2 $\in$ **P**, sí y sólo sí •≥ es un orden débil. Si tal función u existe, entonces es una escala ordinal.

$$((\mathbf{P},\bullet\geq),(\Re, \geq),u)$$

Para el caso de la métrica V-DINO, u, se supondrá que **P** es el conjunto de árboles de clases o jerarquías de herencia que inician con una clase abstracta, $\Re$ es el conjunto de números reales, los cuales sólo estarán en el rango 0 – 1 debido a la normalización de la métrica, •≥ es una relación empírica entre árboles de clases que describe que una jerarquía de clases tiene mayor o igual grado de

dependencia de interfaces debido a interfaces no utilizadas y ≥ es el operador normal “mayor o igual que” para comparar números.

En términos más sencillos, la escala ordinal ((**P**,•≥),(ℜ, ≥),V-DINO) nos indica que cada árbol de clases formado a partir de una clase abstracta tendrá un número real asignado a él, el cual será el valor de la métrica V-DINO, y si un árbol de clases tiene más dependencia de interfaces que otro árbol esto se verá reflejado en los números que representan y fueron asignados a ambos árboles.

Definición de Orden Débil [35]

Para poder describir a la métrica V-DINO como una escala ordinal, se debe comprobar que la relación •≥ es un orden débil, es decir, es una relación binaria que es transitiva y completa:

P1 •≥ P2, y P2 •≥ P3 => P1 •≥ P3	Transitiva
P1 •≥ P2 ó P2 •≥ P1	Completa

para todo P1, P2 y P3 ∈ **P**, donde •≥ es una relación empírica binaria, como igual o más difícil de mantener.

Demostración

La escala ordinal es la base de la medición, todas las otras escalas están basadas en ella. De la escala nominal sólo toma la condición de que los conceptos medidos pueden ser iguales o diferentes.

Si se desea utilizar una métrica en el nivel de escala ordinal, entonces se tienen que cumplir en la realidad las condiciones empíricas de que establezca una relación transitiva y completa. Ambas condiciones son condiciones empíricas, pero son derivadas de las propiedades numéricas de los números reales.

Es fácil observar que si 8 > 5 y 5 > 2 entonces 8 > 2. Esto es una propiedad muy conocida de los números reales. Si se quiere mapear propiedades empíricas hacia números reales entonces se necesita de las propiedades de ser transitiva y completa como condiciones empíricas.

Relación Transitiva

Si •≥ es la relación empírica “igual o con mayor grado de dependencia de interfaces”, entonces una relación transitiva significa que: si P1 tiene mayor grado de dependencia de interfaces que P2; y P2 tiene mayor grado de dependencia de interfaces que P3; implica que P1 tiene mayor grado de dependencia de interfaces que P3. Esto se escribe:

P1 •≥ P2, y P2 •≥ P3 => P1 •≥ P3

Si se combina esta propiedad empírica con la propiedad de ser transitivos de los números, entonces se tendría que:

$$P1 \bullet\geq P2, \text{ y } P2 \bullet\geq P3 \Rightarrow P1 \bullet\geq P3$$
$$\Leftrightarrow$$
$$\text{V-DINO(P1)>V-DINO(P2), V-DINO(P2)>V-DINO(P3) => V-DINO(P1)>V-DINO(P3)}$$

Como se pudo observar en los casos de prueba, esta propiedad se cumplió para los MAOO's analizados. Por ejemplo, el árbol de la clase *CStrategy* de la Figura 35 tiene mayor dependencia de interfaces, debido a la cantidad de interfaces no utilizadas, que el árbol de la clase *CLista* de la Figura 34. Este último árbol a su vez tiene mayor grado de dependencia de interfaces que el árbol de la clase aED de la Figura 35, el cual ni siquiera cuenta con el problema.

Reflejado en números se tiene que:

$$\textit{CStrategy} \bullet\geq \textit{CLista}, \text{ y } \textit{CLista} \bullet\geq \textit{aED} \Rightarrow \textit{CStrategy} \bullet\geq \textit{aED}$$
$$\Leftrightarrow$$
$$0.6120 > 0.5000, \text{ y } 0.5000 > 0.0000 \Rightarrow 0.6120 > 0.0000$$

Relación Completa

Si se tienen P1 y P2, se debe poder decir que P1 tiene mayor o igual grado de dependencia de interfaces que P2, o viceversa. Esto se escribe como:

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

Utilizando la métrica V-DINO y comparando las estructuras de clases con implementaciones vacías de interfaces, en los casos de prueba siempre fue posible determinar cuando existía mayor o menor grado de dependencia de un árbol con otro árbol.

Orden de Clasificación

La tercera condición para una escala ordinal es el homomorfismo entre los sistemas relacionales empírico y numérico por el orden de los objetos. Esto se denota como:

$$P1 \bullet\geq P2 \Leftrightarrow u(P1) \geq u(P2)$$

El orden creado por la métrica V-DINO debe ser el mismo que el orden creado empíricamente mediante experimentación. O viceversa, si se ordenan a los árboles de clases de una forma, la métrica V-DINO debe crear el mismo orden.

Esto se puede observar en los datos contenidos en la Tabla 1, donde se considera que entre haya más clases derivadas con implementaciones vacías, más crece el problema, si se considera al caso más común del problema de dependencia de interfaces. Y este mismo comportamiento se observa en los valores de la métrica V-DINO, en donde también va creciendo el grado de dependencia de interfaces. Esto es:

(6 Clases) •≥ (5 Clases) •≥ (4 Clases) •≥ (3 Clases) •≥ (2 Clases)

⇔

$$0.8333 \geq 0.8000 \geq 0.7500 \geq 0.6666 \geq 0.5000$$

Anexo C) ANÁLISIS DEL SISTEMA

Explicación de Actores

Usuario

Representa una persona o sistema externo que utiliza la herramienta. Tiene básicamente tres acciones que puede ejecutar: Seleccionar archivos para analizarlos, ejecutando el caso de uso *Manejar Archivo*, solicitar el cálculo de la métrica V-DINO, ejecutando el caso de uso *Métrica*, y solicitar la ejecución del método de refactorización denominado “Separación de Interfaces”, ejecutando el caso de uso *Generar Arquitectura*.

Interviene en otras opciones y datos de entrada en los diferentes casos de uso, pero esto es explicado en los flujos donde ocurren estas intervenciones.

Archivo Original

Representa a los archivos .java que el *Usuario* elige abrir en el caso de uso *Manejar Archivo*. Son los archivos donde se encuentra el código que se desea analizar. Estos archivos serán modificados si se elige ejecutar el caso de uso *Generar Arquitectura* y eventualmente se convertirán en los *Archivos Finales*.

Archivo Fuente

Representa a los archivos .java que el *Usuario* elige crear como respaldo en el caso de uso *Manejar Archivo*. Estos archivos son una copia del código original del MAOO a analizar, y no serán modificados. Sólo quedan para respaldo del *Usuario* para regresar a la versión anterior del MAOO.

Archivo Final
Representa a los archivos .java que los casos de uso *Generar Arquitectura* y *Generar Código* producen como salida. Estos archivos tienen el código Re-factorizado después de ejecutar el método de refactorización. Si se aplicara sobre este código otra vez el método, no debería de producirse ningún cambio.

Flujo de Eventos para el Caso de Uso *Manejar Archivo* (Figura 12)

Condiciones Previas
No hay condiciones previas para este caso de uso. Este caso se tiene que ejecutar antes que los demás casos de uso.

Flujo Principal
Este caso de uso empieza cuando el *Usuario* entra al sistema y elige la opción de *Seleccionar Archivos Originales*. El sistema entonces muestra el cuadro de diálogo *Abrir Archivos*. Se presenta al usuario la opción de abrir uno o más archivos, y si esta opción tiene éxito se presenta entonces la opción de crear copias de seguridad. También se puede elegir la opción de cancelar del cuadro de diálogo (E-1).

Si la actividad seleccionada es abrir uno o más archivos, el subflujo S-1: *Seleccionar Archivo* es ejecutado.

Si la actividad seleccionada es crear copias de seguridad, el subflujo S-2: *Crear Copia de Archivo* es ejecutado.

Si la actividad seleccionada es cancelar, termina el caso de uso.

Subflujos
S-1: Seleccionar Archivo.
El sistema despliega el cuadro de diálogo *Abrir Archivos*, que permite seleccionar un directorio dentro de las unidades de la computadora local y dentro de él se pueden seleccionar uno o más archivos .java que identifican a los archivos de Java que el *Usuario* quiere que sean analizados. También se puede escribir él(los) nombre(s) de archivo(s) en el cuadro de edición. Una vez seleccionados o escritos los archivos, se presiona la opción de *Aceptar* (E-1). En dado caso que no haya errores se procede al subflujo S-2, quedando activos en el sistema estos *Archivos Originales* seleccionados.

S-2: Crear Copia de Archivo.
El sistema muestra un cuadro de decisión en donde se pregunta al usuario si desea hacer copias de seguridad de los *Archivos Originales*. Las opciones son SI y NO (E-2). En caso afirmativo, el sistema crea un directorio llamado *Fuente* (E-3, E-4), anidado dentro del directorio que seleccionó el *Usuario* en S-1. En este nuevo directorio se crea una copia de todos los *Archivos Originales* que fueron seleccionados en S-1(E-5, E-6). Estas copias pasan a ser los *Archivos Fuente* activos en el sistema.

Flujos Alternos

E-1: Se introdujo un nombre de archivo inválido, no se puede abrir el archivo o no se eligió ningún archivo. Se informa al *Usuario* de este error y el cuadro de diálogo se cierra. Quedan activos los archivos que hayan sido previamente seleccionados en otra ejecución del caso de uso, si es que había alguno. El error se produce porque los archivos no existen o están dañados. Termina el caso de uso.

E-2: El *Usuario* elige la opción NO. En este caso no quedan *Archivos Fuente* activos en el sistema. Termina el caso de uso.

E-3: Un directorio llamado *Fuente* ya existe. Si esta es la opción, no se crea un nuevo directorio, y se utiliza el ya existente. El caso de uso continúa.

E-4: El sistema no pudo crear el directorio *Fuente*. Se informa de este error al usuario, indicando que no se pudieron crear las copias de seguridad. En este caso no quedan *Archivos Fuente* activos en el sistema. Este error se puede producir si la unidad en donde se quiere crear el directorio es de sólo lectura o está protegida contra escritura. El caso de uso termina.

E-5: Un archivo con el mismo nombre ya existe en el directorio. El archivo anterior es sobrescrito de manera automática (E-6). El caso de uso continúa.

E-6: El sistema no puede copiar o sobrescribir el archivo. Se informa de este error al usuario, indicando que no se pudieron crear las copias de seguridad. En este caso no quedan *Archivos Fuente* activos en el sistema. Este error se puede producir si la unidad en donde se quiere copiar el archivo es de sólo lectura, no tiene espacio libre o está protegida contra escritura. El caso de uso termina.

Flujo de Eventos para el Caso de Uso *Análisis Sintáctico* (Figura 13)

Condiciones Previas

El subflujo *Seleccionar Archivo* del caso de uso *Manejar Archivo* debe de ejecutarse antes de este caso de uso. Esto es porque debe de haber archivos originales activos en el sistema, para que este caso de uso trabaje sobre ellos.

Flujo Principal

Este caso de uso empieza cuando el caso de uso *Métrica* o el caso de uso *Generar Arquitectura* lo solicitan. Se utilizará un analizador sintáctico para recorrer todos los *Archivos Originales*, buscando los datos necesarios para la reestructura del código para almacenarlos en la *Base de Datos*. Las actividades que se realizan son las de encontrar las clases abstractas, encontrar las clases

derivadas y encontrar las funciones con implementación vacía. Estas actividades son llevadas a cabo de forma secuencial.

Si la actividad a realizar es encontrar las clases abstractas, el subflujo S-2: *Encontrar Clase Abstracta* es ejecutado. Para llevar a cabo esto se ejecuta también el subflujo S-1: *Encontrar Función Virtual Pura*.

Si la actividad a realizar es encontrar las clases derivadas, el subflujo S-3: *Determinar Clase Derivada* es ejecutado.

Si la actividad a realizar es encontrar las funciones con implementación vacía, el subflujo S-4: *Encontrar Función Vacía* es ejecutado.

Subflujos

S-1: Encontrar Función Virtual Pura.

El analizador sintáctico, de manera transparente al *Usuario*, recorrerá todos los *Archivos Originales* .java que estén activos. Lo que se busca es la declaración de funciones abtractas, es decir, funciones igualadas a cero (E-1).

S-2: Encontrar Clase Abstracta.

El analizador sintáctico, de manera transparente al *Usuario*, recorrerá todos los *Archivos Originales* .java que estén activos. Para encontrar las clases abstractas primero se ejecuta el subflujo S-1, ya que una clase abstracta es aquella que tiene una o más funciones abtractas.

S-3: Determinar Clase Derivada.

El analizador sintáctico, de manera transparente al *Usuario*, recorrerá todos los *Archivos Originales* .java que estén activos. Para encontrar las clases derivadas de clases abstractas primero se ejecuta el subflujo S-2, ya que una clase derivada es aquella que hereda de una clase abstracta.

S-4: Encontrar Función Vacía.

El analizador sintáctico, de manera transparente al *Usuario*, recorrerá todos los *Archivos Originales* .java que estén activos. Para encontrar las funciones con implementación vacía primero se ejecutan los subflujos anteriores, para tener la información referente a las funciones virtuales, y a las clases abstractas y derivadas. Dentro de la implementación de las clases derivadas se debe de buscar la implementación de cada función virtual que tenga su clase base abstracta, para así determinar si esta implementación está vacía o no. Se considera vacía cuando una función que no tiene valor de retorno sólo tienes las llaves de apertura y cierre, o cuando una función que si tiene valor de retorno regresa un valor nulo (E-6). Termina el caso de uso.

Flujos Alternos

E-1: No se encontraron funciones abtractas. Se buscarán únicamente funciones virtuales que no estén implementadas en la misma clase (E-2), y a estas funciones se les considerará como funciones abtractas. El caso de uso continúa.

E-2: No se encontraron funciones virtuales ni virtuales puras. Cuando tampoco se puede cumplir la condición de E-1, se toma entonces que no existen funciones abtractas, por ende, no hay clases abstractas ni derivadas de éstas. Termina el caso de uso.

E-3: La función virtual pura no tiene parámetros de entrada. El caso de uso continúa.

E-4: El parámetro no tiene valor por defecto. En este caso la *Base de Datos* permite que el campo quede vacío. El caso de uso continúa.

E-5: Una clase tiene más de una función virtual pura. Aquí se debe verificar que antes de crear un nuevo registro, el valor no exista ya en otro registro, y si ya existe no se crea el registro. Con esto se evita tener duplicados del mismo registro. El caso de uso continúa.

E-6: Valores nulos de retorno. Para considerar que una función regresó un valor nulo, sólo se buscará que esa función tenga una sola línea de código “return 0;” para valores de retorno numéricos, “return null;” para valores de retorno de objetos o cadenas, y “return ‘’;” para valores de retorno de caracteres. Si por alguna razón la función tiene código diferente a éste, el analizador sintáctico tomará a la función como necesaria, aunque la función en realidad no sea necesaria. El caso de uso continúa.

E-7: Una clase derivada no tiene implementada a una función virtual pura de su clase abstracta. Este caso sólo se puede dar si antes se entró al flujo alterno E-1. Lo que se hace es hacer un registro de esta función colocándola como vacía, poniendo valores ‘0’ en los campos *línea*, *linea1* y *linea2*. Se necesitan estos registros vacíos para el caso de uso *Métrica*. El caso de uso continúa.

Flujo de Eventos para el Caso de Uso *Métrica* (Figura 14)

Condiciones Previas
El caso de uso *Análisis Sintáctico* debe de ejecutarse antes de este caso de uso, y es llamado automáticamente por *Métrica*.

Flujo Principal
Este caso de uso empieza cuando el *Usuario* elige la opción *Calcular Métrica V-DINO* del menú principal del sistema. Este caso de uso realiza dos actividades en forma secuencial, primero se hace el cálculo de la métrica y posteriormente se regresa al *Usuario* el valor calculado, con la correspondiente interpretación del resultado.

Si la actividad a realizar es el cálculo de la métrica, el subflujo S-1: *Calcular V-DINO* es ejecutado.

Si la actividad a realizar es la interpretación del resultado, el subflujo S-2: *Interpretar Resultado* es ejecutado.

Subflujos

S-1: Calcular V-DINO.

Para calcular la métrica V-DINO de Dependencia de Interfaces que No se Ocupan únicamente se toman datos que obtenemos en la extracción de datos en el Meta-compilador Antlr. El valor de la métrica es el número de funciones vacías entre la multiplicación del número de clases derivadas por el número de funciones abtractas, y este valor se presenta al *Usuario*. El caso de uso continúa.

S-2: Interpretar Resultado.

Cuando se hace el cálculo de V-DINO para una clase abstracta en S-1 se procede a interpretar ese valor. En caso que el valor sea de 0.500 o mayor, se recomendará al *Usuario* que proceda a resolver el problema de dependencia de interfaces, utilizando el método de Separación de Interfaces. En caso que el valor de V-DINO sea menor a 0.500 (E-3), entonces se recomendará al usuario que no hay necesidad de ejecutar el método de refactorización. Dependiendo de la decisión del *Usuario*, se ejecuta el caso de uso *Generar Arquitectura* (E-4). Termina el caso de uso.

Flujos Alternos.

E-1: Este caso se da sólo cuando se ejecuta el flujo alterno E-2 del caso de uso *Análisis Sintáctico*. En este caso no se puede calcular el valor de V-DINO, por tanto, se informa de esto al *Usuario*, y se le dice que no tiene porqué ejecutar el método de refactorización, ya que no hará nada por no existir clases abstractas ni funciones abtractas. Termina el caso de uso.

E-2: Hay registros con campos nulos. Esto sucede sólo cuando se ejecuta el flujo alterno E-1 del caso de uso *Análisis Sintáctico*. Este caso no afecta al cálculo de la métrica ni a su interpretación, pero el sistema debe de tener en cuenta esto si se llega a ejecutar el caso de uso *Generar Arquitectura*. El caso de uso continúa.

E-3: El valor de V-DINO es cero. Este es el único caso en donde no existe el problema, y se debe manejar un mensaje especial para el *Usuario*, indicando que el método de refactorización no tiene razón de ser ejecutado. El caso de uso continúa.

E-4: El *Usuario* no sigue la recomendación generada por medio de V-DINO. Al ejecutar el caso de uso *Generar Arquitectura*, no se toma en cuenta la métrica, pero es altamente probable que no se pueda resolver el problema si V-DINO lo

indicó así. Esto no afecta el funcionamiento del método, sólo se hará trabajo extra y se conoce de antemano el resultado. El caso de uso continúa.

Flujo de Eventos para el Caso de Uso *Generar Arquitectura* (Figura 15)

Condiciones Previas
El caso de uso *Análisis Sintáctico* debe de ejecutarse antes de este caso de uso, y es llamado automáticamente por *Generar Arquitectura* si no ha sido llamado antes por el caso de uso *Métrica*. Esto es porque debe de estar llena la *Base de Datos* antes de realizar la refactorización del MAOO.

Flujo Principal
Este caso de uso empieza cuando el *Usuario* solicita que se intente hacer la reestructura de los *Archivos Originales* utilizando el método de refactorización, ofrecido en la opción *Método de Separación de Interfaces* del menú principal del sistema. Este caso de uso tiene primeramente la función de encontrar si se puede hacer la refactorización, y para ello debe buscar interfaces afines que sean mutuamente excluyentes (E-1), y una vez encontradas éstas se proceden a integrar las interfaces y ofrecer una clase intermedia por interfaz sustituida con métodos de enganche. Estas actividades se realizan en forma secuencial. El proceso se hace completo para cada clase abstracta que se tenga detectada. Después de este caso de uso se ejecuta automáticamente el caso de uso *Generar Código*.

Si la actividad a realizar es buscar interfaces afines mutuamente excluyentes, el subflujo S-1: *Sustituir Interfaces* es ejecutado. Al encontrar un caso de estas interfaces, se procede a integrar interfaces, ejecutando el subflujo S-2: *Crear Interfaz Genérica*.

Si la actividad a realizar es ofrecer clases intermedias por interfaz sustituida, el subflujo S-3: *Crear Clase Intermedia* es ejecutado. Al hacer una clase intermedia, se debe de agregar un método de enganche, ejecutando el subflujo S-4: *Crear 'Template Method'*.

Subflujos
S-1: Sustituir Interfaces.
Para cada clase abstracta, se determina que funciones virtuales implementa, también se determina que clases derivadas tiene, y por último se encuentran todas las implementaciones de las funciones virtuales (E-2. Al tener recopilada esta información se buscan interfaces afines o con la misma firma, es decir, que tengan los mismos parámetros de entrada y salida. Para esto último se utiliza la información recopilada sobre las funciones abtractas y la información sobre sus parámetros de salida, con el campo *retorno*, y sus parámetros de entrada. Si dos o más funciones virtuales tienen iguales estos parámetros de entrada y salida, entonces son candidatas a ser sustituidas por una interfaz genérica. Lo último

que se tiene que determinar es si son mutuamente excluyentes, y para ello se tiene que determinar que una y sólo una función de las candidatas sea implementada por cada clase derivada (E-3). Cuando ya se determinó que son mutuamente excluyentes (E-1), se procede a ejecutar el subflujo S-2. El caso de uso continúa.

S-2: Crear Interfaz Genérica.
Se creará una interfaz genérica que sustituya a las funciones virtuales mutuamente excluyentes encontradas en S-1. Se solicita al *Usuario* que proporcione el nombre que quiere para esta interfaz, ofreciendo el sistema un nombre por defecto (E-4). La función virtual pura es creada entonces en la clase abstracta (E-5), para colocar la información, y la información recopilada para los parámetros de entrada y salida de esta interfaz. El caso de uso continúa.

S-3: Crear Clase Intermedia.
Lo que sigue es agrupar a las clases derivadas por la única función virtual que implementan, del grupo de funciones abtractas mutuamente excluyentes encontradas en S-1. Al tener esta agrupación, se procede a crear una clase genérica para cada grupo que haya surgido. Se solicita al *Usuario* que proporcione el nombre que quiere para cada clase intermedia, y para los nombres de los archivos .java que se crearán para esta clase, ofreciendo el sistema un valor por defecto (E-6). Entonces se crean los nuevos archivos en el mismo directorio donde están los *Archivos Originales* (E-7) y se llenan con la información que ya se tiene recopilada, como es la herencia hacia la clase abstracta y nombre de la clase intermedia, así como el esqueleto de la declaración e implementación (E-8). Se les tiene que crear un método de enganche, para conectarse con las clases derivadas de su grupo, y para ello se ejecuta el subflujo S-4. El caso de uso continúa.

S-4: Crear 'Template Method'.
Un 'Template Method' es un método de enganche que servirá para conectar la nueva función virtual pura creada en la clase abstracta, con la implementación realizada en las clases derivadas. En este proceso sólo se hace la declaración del método en el archivo .h recién creado y el esqueleto de la función en el nuevo archivo .java de la clase intermedia (E-8), es decir sólo se coloca la línea de identificación de la función y las llaves de apertura y cierre. Todos los datos para este método ya se tienen recopilados. Termina el caso de uso.

Flujos Alternos
E-1: No se encontraron funciones abtractas mutuamente excluyentes. En este caso no se puede hacer la separación de interfaces. Se informa de esto al *Usuario* indicando sobre qué clase abstracta no se puede ejecutar el método. El caso de uso termina para esa clase abstracta, se procede con la siguiente.

E-2: Este caso se da sólo cuando se ejecutó el flujo alterno E-1 del caso de uso *Análisis Sintáctico*. Estos registros no deben ser tomados en cuenta para la recopilación de información. El caso de uso continúa.

E-3: De un grupo de funciones candidatas sólo algunas son mutuamente excluyentes. En este caso se tendrá que preguntar al *Usuario* que acción tomar, si sólo sustituir algunas o no modificar nada. Dependiendo de la respuesta del *Usuario* el caso de uso continúa o termina.
E-4: El nombre de la función virtual pura ya está siendo utilizado o no es válido. Es una de las precondiciones del método de refactorización, al encontrar nombres repetidos, se debe solicitar de nuevo al *Usuario* un nombre, indicando que el anterior no puede ser usado. El caso de uso continúa.

E-5: Esta activada la bandera que indica que el flujo alterno E-1 del caso de uso *Análisis Sintáctico* fue ejecutado. En este caso no se debe crear una función virtual pura sino sólo una función virtual sin implementación en la clase abstracta. El caso de uso continúa.

E-6: El nombre de la clase intermedia o de los archivos ya está siendo utilizado o no es válido. Es una de las precondiciones del método de refactorización, al encontrar nombres repetidos, se debe solicitar de nuevo al *Usuario* un nombre, indicando que el anterior no puede ser usado. El caso de uso continúa.

E-7: No se pueden crear los nuevos archivos. Esto se puede dar si el directorio está dentro de una unidad de sólo lectura o si está protegida contra escritura. Se debe dar un mensaje de error al *Usuario* con la opción de reintentar. Si otra vez no se puede realizar la operación se debe cancelar todo el proceso y el caso de uso termina, en caso que, si se efectúe la operación, el caso de uso continúa.

E-8: No se pueden modificar los *Archivos Originales*. Esto se puede dar si el directorio está dentro de una unidad de sólo lectura o si está protegida contra escritura. Se debe dar un mensaje de error al *Usuario* con la opción de reintentar. Si otra vez no se puede realizar la operación se debe cancelar todo el proceso y el caso de uso termina, en caso que, si se efectúe la operación, el caso de uso continúa.

Flujo de Eventos para el Caso de Uso *Generar Código*

Condiciones Previas
El caso de uso *Generar Arquitectura* debe de ejecutarse con éxito antes de este caso de uso. Esto es porque ya debe de haber clases intermedias para separar las interfaces.

Flujo Principal

Este caso de uso empieza cuando el caso de uso *Generar Arquitectura* lo invoca automáticamente. Tiene la función de Re-factorizar todo el código con el nuevo diseño generado por el caso de uso *Generar Arquitectura* para que siga teniendo la misma funcionalidad del MAOO original. Este proceso, al igual que en el caso de uso anterior, se lleva a cabo por cada clase abstracta que tuvo funciones abtractas mutuamente excluyentes. Las actividades de este caso se realizan en forma secuencial. Primeramente, se deben reubicar las funciones abtractas, implementando los 'Template Methods'; después se cambian las relaciones de herencia, modificando las directivas #include; posteriormente se eliminan las funciones con implementación vacía que fueron sustituidas, eliminando con ello sus declaraciones; y por último se modifican las llamadas a las funciones que fueron sustituidas.

Si la actividad a realizar es reubicar las funciones abtractas, el subflujo S-1: *Mover Función Virtual Pura* es ejecutado. Y para llevar a cabo esto se ejecuta también el subflujo S-2: *Implementar 'Template Method'*.

Si la actividad a realizar es cambiar las relaciones de herencia, el subflujo S-3: *Modificar Relación de Herencia* es ejecutado. Y para llevar a cabo esto se ejecuta también el subflujo S-4: *Cambiar Directiva Include*.

Si la actividad a realizar es eliminar las funciones vacías que, si fueron sustituidas, el subflujo S-5: *Eliminar Función Vacía* es ejecutado. Y para llevar a cabo esto se ejecuta también el subflujo S-6: *Eliminar Declaración de Función*.

Si la actividad a realizar es modificar las llamadas a funciones que fueron sustituidas, el subflujo S-7: *Cambiar Llamada a Función* es ejecutado.

Subflujos

S-1: Mover Función Virtual Pura.

Aquí se bajan en la jerarquía de herencia a las funciones abstractas mutuamente excluyentes que fueron sustituidas por una interfaz genérica. Cada una de las funciones virtuales se baja de la clase abstracta hacia una de las clases intermedias que fueron creadas en el caso de uso *Generar Arquitectura*, donde la clase intermedia será la base para el grupo de clases derivadas que implementan esa función virtual pura. Y agregarla en la clase intermedia, utilizando la información generada en el caso de uso previo. Aquí se modifican los *Archivos Originales* .java de la clase abstracta e intermedia (E-1). Después de hacer esto para cada clase intermedia, se debe ejecutar el subflujo S-2. Después de borrar las funciones abstractas innecesarias de la clase abstracta, sus archivos ya no sufrirán más modificaciones (E-2), salvo que los modifique S-7 para cambiar llamadas a funciones. El caso de uso continúa.

S-2: Implementar 'Template Method'.
Los 'Template Methods' ya habían sido creados y declarados en el subflujo S-4 del caso de uso *Generar Arquitectura*. Ahora se modificará el archivo donde se encuentra la implementación del 'Template Method' (E-1). La implementación del método será una sola línea con una llamada hacia la función virtual pura, pasándole sus parámetros de entrada (E-3). Después de realizar esto, los archivos de las clases intermedias ya no ocupan más modificaciones, salvo que los modifique S-7 para cambiar llamadas a funciones. El caso de uso continúa.

S-3: Modificar Relación de Herencia.
Este proceso es para cambiar las relaciones de herencia de las clases derivadas que fueron agrupadas en el subflujo S-3 del caso de uso *Generar Arquitectura*, cambiando la herencia de la clase abstracta hacia la clase intermedia que encabeza el grupo. Colocando ahora el nombre de la clase intermedia correspondiente. Se hace la modificación en el archivo .java de la clase derivada. Después de cambiar esta línea, se ejecuta el subflujo S-4. El caso de uso continúa.

S-4: Cambiar Directiva Include.
Una vez que se cambió la relación de herencia lo que sigue es cambiar la directiva #include. En esa instrucción se pondrá el nombre del archivo .java que tiene la declaración de la clase intermedia correspondiente. El caso de uso continúa.

S-5: Eliminar Función Vacía.
Este proceso ocurre en los archivos .java de las clases derivadas. Lo que se hace es eliminar la implementación de las funciones virtuales que no ocupa esa clase derivada, ya que sólo ocupa una función virtual del grupo de funciones mutuamente excluyentes, determinada en el subflujo S-3 del caso de uso *Generar Arquitectura*. Para borrar estas implementaciones se toma la información de su ubicación. Una vez eliminado este código del *Archivo Original*, se procede a ejecutar el subflujo S-6. El caso de uso continúa.

S-6: Eliminar Declaración de Función.
Este proceso ocurre en los archivos .java de las clases derivadas. Lo que se hace es eliminar la declaración de la función virtual que no ocupa esta clase derivada, cuya implementación ya fue borrada en S-5. Para borrar esta declaración se toma la información sobre su ubicación. Después de esto, los archivos de las clases derivadas ya no ocuparán más modificaciones, salvo que los modifique S-7 para cambiar llamadas a funciones. El caso de uso continúa.

S-7: Cambiar Llamada a Función.
Este proceso puede ocurrir en cualquier *Archivo Original*. Lo que se hace es cambiar los nombres de funciones. Se sustituye el nombre de la función virtual original por el nombre de la interfaz genérica. No hay necesidad de cambiar nada más, ya que ambas funciones tienen los mismos parámetros de entrada y salida. Termina el caso de uso.

Flujos Alternos.
E-1: No se pueden modificar los *Archivos Originales*. Esto se puede dar si el directorio está dentro de una unidad de sólo lectura o si está protegida contra escritura. Se debe dar un mensaje de error al *Usuario* con la opción de reintentar. Si otra vez no se puede realizar la operación se debe cancelar todo el proceso y el caso de uso termina, en caso que, si se efectúe la operación, el caso de uso continúa.

E-2: Las funciones virtuales eliminadas de la clase abstracta no son puras. Esto sucede cuando se ejecutó el flujo alterno E-1 del caso de uso *Análisis Sintáctico*. En este caso también se debe buscar dentro del archivo .java de la clase abstracta la implementación vacía de esta función, para eliminarla. El caso de uso continúa.

E-3: La función virtual pura tiene valor de retorno. Sucede cuando el campo *retorno* del registro de la función tiene un valor diferente a “void”. En este caso se debe cambiar la línea de código de implementación del ‘Template Method’ de una llamada hacia un regreso de valor. Y dentro de este regreso de valor con la instrucción “return” se coloca la llamada a la función. El caso de uso continúa.

E-4: La clase tiene más de una función implementada que no ocupa. Esto no es problema, sino que se tiene que adoptar un algoritmo de borrado, eliminando la implementación o declaración que esté más abajo en el archivo. Termina el caso de uso.

REFERENCIAS

[1] Chidamber, S.R. and Kemerer, C. F. (1991). Towards a Metrics Suite for Object Oriented Design. In *Proc. of the 6th ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)* (pp. 197–211). Phoenix, AZ.

[2] Erich Gamma, Richard Helm, Ralph E. Johnson, y John Vlissides; *Design Patterns: Elements of Reusable Software Architecture*; Addison Wesley; 1995

[3] Alain Abran, James W. Moore, Pierre Bourque, R. D. y L. L. T. (2001). Guide to the Software Engineering Body of Knowledge. Retrieved from http://www.swebok.org

[4] Clemens Sziperski; *Component Software: Beyond Object-Oriented Programming*; Addison Wesley; 1999

[5] Martin Fowler, Kent Beck, John Brant, William F. Opdyke y Donald Roberts; *Refactoring, Improving the Design of Existing Code*; Addison Wesley, 1999

[6] René Santaolaya, Olivia G. Fragoso, Manuel Valdés e Isaac M. Vásquez; "Preparing
Frameworks to Become Web Services"; *Proceedings of the IADIS International Conference Applied Computing 2004*; Lisboa, Portugal; 2004

[7] René Santaolaya; *Modelo de Representación de Patrones de Código para la Construcción de Componentes Reusables*; Tesis de Doctorado; Departamento de Ciencias Computacionales, Centro de Investigación en Computación, IPN; 2002

[8] René Santaolaya, Olivia G. Fragoso, Joaquín Pérez y Lorenzo Zambrano; "Restructuring Conditional Code Structures Using Object Oriented Design Patterns"; *Proceedings of The 2003 International Conference on Computational Science and Its Applications – ICCSA 2003*; Springer-Verlag LNCS 2667; Montreal, Canadá; 2003

[9] " Identificación de Funciones Recurrentes en Software Legado". Anel Sheydi Zamudio López, CENIDET 2001.

[10] "Concepción de un Modelo para el Aseguramiento de Calidad de Componentes Reusables de Software". Javier Santa Olalla Salgado, CIC-IPN

[11] " Sistema de Pruebas de Calidad de Componentes Reusables". Blanca R. Olascoaga Vergara, CIC-IPN

[12] "IPADIC++: Sistema para Identificación de Patrones de Diseño en Código C++". Agustín F. Castro Espinoza, CENIDET

[13] "Factorización de Funciones en Métodos de Plantilla". Laura Alicia Hernández Moreno, CENIDET.

[14] "Reconocimiento de Patrones de Diseño de Gamma a partir de la Forma Canónica definida en el IPADIC++". Patricia Zavaleta Carrillo, CENIDET.

[15] Leonor A. Cárdenas; *Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator*; Tesis de Maestría; Departamento de Ciencias de la Computación, Centro Nacional de Investigación y Desarrollo Tecnológico; 2004

[16] "Reestructuración de Software Escrito por Procedimientos Conducido por Patrones de Diseño Composicionales". Armando Méndez Morales, CENIDET.

[17] "Integración de la Funcionalidad de Frameworks Orientados a Objetos". Juan José Rodríguez Gutiérrez, CENIDET.

[18] "Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos, Utilizando el Patrón de Diseño Adapter". Luis Esteban Santos Castillo, CENIDET.

[19] "Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces", Manuel Alejandro Valdés Marrero 2004, CENIDET.

[20] "Refactorización de Sistemas Legados de Software, para equilibrar la Coherencia, Cohesión y el Factor de Acoplamiento de su estructura interna" Sandro Geovani Vázquez Díaz, CENIDET

[21] William F. Opdyke; *Refactoring Object-Oriented Frameworks*; Tesis de Doctorado; Department of Computer Science, University of Illinois at Urbana Champaign; 1992

[22] Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). The Journal of Systems and Software Identification and application of Extract Class refactorings in object-oriented systems. *The Journal of Systems & Software*, *85*(10), 2241–2260. https://doi.org/10.1016/j.jss.2012.04.013

[23] Murphy-hill, X. G. E. (2014). Manual Refactoring Changes with Automated Refactoring Validation.

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

[25] Dig, D., Marrero, J., & Ernst, M. D. (2009). Refactoring sequential Java code for concurrency via concurrent libraries. *IEEE Computer Society Washington, DC, USA*, 397–407.

[26] Singh, S., & Kaur, S. (2016). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*. https://doi.org/10.1016/j.asej.2017.03.002

[27] Khatchadourian, R. (2016). Automated refactoring of legacy Java software to enumerated types. *Automated Software Engineering*. https://doi.org/10.1007/s10515-016-0208-8

[28] Hamioud, S., & Atil, F. (2015). Model-driven java code refactoring. *Computer Science and Information Systems*, *12*(2), 375–403. https://doi.org/10.2298/CSIS141025015H

[29] Singh, S., & Kaur, S. (2016). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*. https://doi.org/10.1016/j.asej.2017.03.002

[30] Hotta, K., Higo, Y., Igaki, H., & Kusumoto, S. (2012). CRat: A refactoring support tool for Form Template Method. Proceedings of the 20th IEEE International Conference on Program Comprehension, 250–252. https://doi.org/10.1109/ICPC.2012.6240496

[31] Joshua Kerievsky; DRAFT of Refactoring to Patterns, v 0.17; Industrial Logic, Inc; to be Published by Addison Wesley in 2004; 2003

[32] Francisco J. García, José M. Marqués y Jesús M. Maudes; *Análisis y Diseño Orientado al Objeto para Reutilización*; Reporte Técnico, TR-GIRO-01-97V2.1.1; Universidad de Valladolid; 1997

[33] Ivar Jacobson, Magnus Christerson, Patrik Jonsson y Gunnar Övergaard; *Object-Oriented Software Engineering: A Use Case Driven Approach*; Addison Wesley; 1992

[34] Bertrand Meyer; *Object-Oriented Software Construction*; Prentice Hall; 1998

[35] Horst Zuse; *A Framework of Software Measurement*; Walter de Gruytier; 1998

[36] William C. Chu, Chih-Wei Lu, Chih-Peng Shiu y Xudong He; “Pattern-Based Software Reengineering: a Case Study”; *Journal of Software Maintenance: Research and Practice, No 12*; 2000

[37] Luis E. Santos; *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter*; Tesis de Maestría; Departamento de Ciencias Computacionales, Centro Nacional de Investigación y Desarrollo Tecnológico; 2004

[38] Fernando Brito e Abreu, Miguel Goulao y Rita Esteves; “Toward the Design Quality Evaluation of Object-Oriented Software Systems”; *Proceedings of the 5th International Conference on Software Quality*; Austin, Estados Unidos; 1995

[39] Método de Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos. Tema de tesis en curso por el alumno Orlando Ortiz

[40] Joshua Kerievsky; *DRAFT of Refactoring to Patterns, v 0.17*; Industrial Logic, Inc; to be Published by Addison Wesley in 2004; 2003

[41] Mel Ó Cinnéide y Paddy Nixon; “Program Restructuring to Introduce Design

Patterns”; *Proceedings of the IEEE International Conference on Software Maintenance*; Oxford, Inglaterra; 1999

[42] Lance Tokuda; *Design Evolution with Refactorings*; Tesis de Doctorado; Department of Computer Science, University of Texas at Austin; 1999

[43] Sheila L. Delfín; *Sistema para el Descubrimiento y Publicación de Servicios Disponibles a través del Web*; Tesis de Maestría; Departamento de Ciencias de la Computación, Centro Nacional de Investigación y Desarrollo Tecnológico; 2004

[44] Norman E. Fenton y Shari L. Pfleeger; *Software Metrics: A Rigorous and Practical Approach*; PWS Publishing; 1997

[45] Lionel Briand, Sandro Morasca y Victor R. Basili; “Property-Based Software Engineering Measurement”; *IEEE Transactions on Software Engineering, No 1*; 1996

[46] Martin, R. C. (2011). The Clean Coder: A Code of Conduct for Professional Programmerse (Prentice H).

[47] Liskov, Barbara. “Data Abstraction and Hierarchy”. SIGPLAN Notices, Vol. 23(5), 1988.

[48] Meyer, B. (1997). Object-Oriented Software construction (Prentice H).

[49] P. J. Deitel and H. D. M., Como programar en JAVA. Séptima edición, Pearson Ed. México, 2008.

[50] Meyer, B.: Object-Oriented Software Construction. Prentice- Hall, New York,1998.