EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Disminución de deuda técnica producida por arquitectura de clases con más de una responsabilidad

presentada por

Ing. Juan Antonio Díaz Díaz

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. Blanca Dina Valenzuela Robles

Cuernavaca, Morelos, México. Marzo de 2023.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

Cuernavaca, Mor., 24/febrero/2023
OFICIO No. DCC/048/2023
Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes del Comité Tutorial de JUAN ANTONIO DÍAZ DÍAZ, con número de control M20CE031, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"DISMINUCIÓN DE DEUDA TÉCNICA PRODUCIDA POR ARQUITECTURAS DE CLASES CON MÁS DE UNA RESPONSABILIDAD"** y hemos encontrado que se han atendido todas las observaciones que se le indicaron, por lo que hemos acordado aceptar el documento de tesis y le solicitamos la autorización de impresión definitiva.

RENÉ SANTAOLAYA SALGADO
Director de tesis

BLANCA DINA VALENZUELA ROBLES
Codirectora de tesis

JUAN CARLOS ROJAS PÉREZ
Revisor 1

OLIVIA GRACIELA FRAGOSO DÍAZ
Revisor 2

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

MYHP/ibm

EDUCACIÓN
29 FEB 2023
RECIBIDO

Cuernavaca, Mor., 01/marzo/2023
No. De Oficio: SAC/049/2023
Asunto: Autorización de impresión de tesis

JUAN ANTONIO DIAZ DIAZ
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 **"DISMINUCIÓN DE DEUDA TÉCNICA PRODUCIDA POR ARQUITECTURAS DE CLASES CON MAS DE UNA RESPONSABILIDAD"**, 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"

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

EDUCACIÓN | CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SUBDIRECCIÓN ACADÉMICA

EDUCACIÓN
01 MAR 2023
RE 170

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

CMAZ/RMA

DEDICATORIAS

Esta tesis está dedicada a:

A mis padres (Tomasa y Nahúm), por su gran amor y apoyo, por sembrar en mí los conocimientos que hoy me ponen en la situación de obtener el grado de Maestría.

A Laura (chaparrita) por todo el amor incondicional que me has dado, por tu apoyo, por tener en ti una persona que siempre me impulsa para seguir y conseguir mis metas.

A mi amada hija Valentina, que se convirtió en mi fortaleza en mis momentos de flaqueza, por ella seguiré preparándome y cumpliendo todas mis metas.

A mis hermanos (Yasmin, Anahí y Nahúm) que siempre vieron en mí una persona dedicada y capaz de realizar las cosas que me proponía, por sus consejos y el apoyo brindado.

AGRADECIMIENTOS

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

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

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

A mi director de tesis, Dr. René Santaolaya Salgado, por su generosidad al darme la oportunidad de trabajar en este tema de tesis, por su paciencia, confianza y consejos para culminar esta investigación.

A mi codirectora de tesis, Dra. Blanca Dina Valenzuela Robles, por el tiempo, observaciones y comentarios puntuales en el desarrollo de este trabajo de investigación.

A mis revisores, Dra. Olivia Graciela Fragoso Diaz y Dr. Juan Carlos Rojas Pérez, por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

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

A mis compañeros y amigos que estuvieron conmigo durante el recorrido de mi carrera, por la convivencia que tuvimos: Edson, Leslie, Alejandra, Iván, Yasmin, Orlando, Javier, Leticia, Marisol, Nélida y Ricardo.

A todos aquellos compañeros que se quedaron en el camino por alguna circunstancia, espero que algún día puedan retomar nuevamente el andar y culminar con aquello que alguna vez se propusieron.

RESUMEN

Para desarrollar sistemas de software de calidad, se requiere por parte de los desarrolladores de software conocimiento sobre los principios de diseño, técnicas adecuadas de programación y decisiones de diseño que sean adaptables a las necesidades. Si se carecen de habilidades como; imaginación, creación y abstracción se pueden llegar a desarrollar sistemas que no llegan a cumplir con estándares de calidad, desarrollando sistemas que contengan código desagradable, los cuales a su vez originan deuda técnica, lo que causa que arquitecturas de software no sean reutilizables, y se eleve el costo de mantenimiento.

Uno de los principales factores que originan la deuda técnica es la existencia de arquitecturas modulares con más de una responsabilidad, lo que ocasiona un aumento en su fragilidad por presentar más de un motivo para cambios, lo que violenta el Principio de Única Responsabilidad.

Con el objetivo de resolver esta problemática, en este trabajo de investigación, se propone un método de refactorización denominado "Método de Refactorización de Separación de Responsabilidades", cuyo objetivo es separar las responsabilidades existentes en arquitecturas modulares de programas, escritos bajo el lenguaje Java, dejando cada uno de ellos en la mejor manera posible con una única responsabilidad.

Durante la investigación de este trabajo de investigación, no se encontró en la literatura alguna métrica que mide las responsabilidades en arquitecturas modulares. Por este motivo se propone una métrica de calidad, la cual consiste en medir el número de responsabilidades: Número de Responsabilidades (NR).

Se presentan tres casos de estudio a los cuales se les realiza el cálculo de la métrica de calidad definido antes y después de la refactorización. Así como la refactorización de forma automática.

ABSTRACT

To develop quality software systems, software developers need knowledge of design principles, appropriate programming techniques, and design decisions that are adaptable to needs. If you lack skills like; Imagination, creation and abstraction can lead to developing systems that do not meet quality standards, developing systems that contain Code Smell, which in turn cause technical debt, which causes software architectures not to be reusable, and increases the cost of maintenance.

One of the main factors that originate technical debt is the existence of modular architectures with more than one responsibility, which causes an increase in their fragility by presenting more than one reason for changes, which violates the Single Responsibility Principle.

To solve this problem, a refactoring method called "Refactoring Method of Separation of Responsibilities" is proposed, whose objective is to separate the existing responsibilities in modular architectures of programs, written under the Java language, leaving each of them in the greatest possible way with a single responsibility.

During the investigation of this research work, no metric that measures the responsibilities in modular architectures was found in the literature. For this reason, a quality metric is proposed, which consists of measuring the number of responsibilities: Number of Responsibilities (NR).

Three case studies are presented to which the calculation of the defined quality metric is performed and the method of “Separation of Responsibilities” is applied.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO DE TÉRMINOS

Acoplamiento	El criterio de acoplamiento es una medida para evaluar cómo un sistema ha sido modularizado. Se refiere al grado de interdependencia entre módulos. El grado de acoplamiento se puede utilizar para evaluar la calidad de un diseño de sistema (Gamma et al., 1994). Es preciso minimizar el acoplamiento entre módulos, es decir, minimizar su interdependencia.
Clase	Colección de objetos que posee características, operaciones y relaciones comunes necesarias para crear objetos (Morelli & Walde, 2016). Elemento de software que describe tipos de datos abstractos y su implementación parcial o total.
Código desagradable (Code Smell)	Se refiere al código fuente de programas de computadora que, aunque no de la mejor manera, cumple con su meta de valor u objetivo, y que posiblemente pueda presentar algún problema de fragilidad, rigidez, movilidad, etc. (Fowler et al., 1999) El código desagradable engloba malas opciones de diseño o implementación, opuestas a los modismos y, en cierta medida, a los principios de diseño, en el sentido de que pertenecen a la implementación, mientras que los patrones de diseño pertenecen al diseño. Son soluciones "pobres" a problemas de implementación recurrentes. Técnicamente, el código indeseable no es incorrecto, no son defectos ni impide el correcto funcionamiento del programa. Más bien indica cierta debilidad en el diseño que pueda degradar su desarrollo e incrementar el riesgo de fallas.
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”.
Cohesión	La fortaleza interna de un módulo, esto es, lo fuertemente (estrictamente) relacionadas que están entre sí las partes de un módulo (Yourdon & Constantine, 1978).
Composición	Es la capacidad con la que cuenta el software para favorecer la producción de elementos separados del mismo, los cuáles pueden ser combinados libremente con otros para producir nuevos sistemas (Meyer, 1997).
Continuidad Modular	Un método de diseño satisface la continuidad modular, cuando los cambios no afectan la arquitectura del sistema. Es decir, la relación existente entre los módulos no cambia (Meyer, 1997).
Deuda técnica	La Deuda Técnica (DT) es un fenómeno habitual del proceso de construcción de software, la cual es reconocida como un conjunto de prácticas y decisiones incorrectas en la fase de construcción de software y aceptada generalmente como efecto colateral en proyectos de software, donde existe una presión del cronograma y se deben realizar entregas continuas de valor al cliente a corto plazo (Holvitie et al., 2018). La metáfora de la Deuda Técnica o (del inglés: “Technical Debt”), describe un fenómeno en el cual los problemas de calidad técnica en un sistema de software pueden conducir a problemas futuros si no se resuelven inmediatamente.

Descomposición	Se requiere del uso de métodos de diseño que ayuden a la descomposición de un gran problema en otros subproblemas más pequeños, cuya solución pueda ser obtenida separadamente (Meyer, 1997).
Extensibilidad	Es la facilidad que presenta el software para poder ampliar sus funcionalidades sin que estos cambios requieran de un gran esfuerzo y sin modificar la arquitectura original, respetando el criterio de continuidad modular (Shatnawi & Li, 2011).
Flexibilidad	Es la facilidad con la que un sistema o componente puede cambiar su comportamiento funcional o de ejecución para su uso en aplicaciones o entornos distintos para los que fue diseñado originalmente (Shatnawi & Li, 2011).
Fragilidad	Es una debilidad que tiene un sistema de software para adaptarse a algún cambio. Implica generar fallas cuando en módulos que aparentemente trabajan bien cuando hay cambios en alguna parte del sistema. Ocurre cuando un cambio en un módulo se propaga a otros módulos, aunque no haya una relación directa (C. Martin & Martin, 2006).
Genericidad	Es la propiedad que permite construir abstracciones modelo (clases genéricas) de clases. El modelo puede parametrizarse con otras clases, objetos y/o operaciones. A nivel de métodos, es la propiedad que permite definir métodos que tienen como parámetros elementos de cualquier tipo (Sarnath & Brahma, 2015).
Inmovilidad	Es la resistencia por parte del software o sus componentes para ser utilizado en otros proyectos o parte de otros proyectos (C. Martin & Martin, 2006).
Legibilidad	Es la capacidad de un método para producir módulos o unidades de programa que puedan ser entendidos separadamente por un lector humano.
Meta	La meta de valor representa un objetivo de negocio a alcanzar a través de una *unidad de software*.
Modularidad	La modularidad implica romper una solución en partes que puedan ser autónomas, encapsuladas y reusables, lo cual implica que se puede cambiar su comportamiento concreto sin afectar el resto de la solución (Pressman, 2010).
Movilidad	Es la facilidad que tiene el software para ser reusado en otros proyectos o parte de otros proyectos.
Principio Abierto/Cerrado	El principio de *"abierto cerrado"* establece que las entidades de software debieran estar cerradas a modificaciones (originados por cambios a requerimientos o por defectos insertados en el software) y abiertas a las extensiones (Meyer, 1997). La cerradura es estratégica ante cambios, es decir que sólo se cierra a cambios por las dimensiones que tienen impacto en usos futuros como elementos reusables.

Principio de Única Responsabilidad	Establece que un método, clase o componente solo debe de tener un motivo para cambiar (C. Martin, 2018).
Protección modular	Es la capacidad de un método de diseño, cuando éste resulta en arquitecturas en las que los efectos anormales que ocurren en tiempo de ejecución de un módulo, estén confinados a ese único módulo, o cuando mucho se propague a algunos pocos módulos vecinos (Meyer, 1997).
Refactorización	Un cambio realizado en la estructura interna del software para facilitar su comprensión y más barato de modificar sin cambiar su comportamiento observable. 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 (Fowler et al., 1999).
Responsabilidad	Responsabilidad se define como una razón de cambio y cada una obedece a una *meta de valor*. Cada responsabilidad es un eje o una dimensión de cambio. Cuando un requerimiento cambia, se manifestará a través de un cambio de responsabilidad entre las clases. Si una clase asume más de una responsabilidad, esa clase tendrá más de una razón de cambio.
Rigidez	Es una debilidad del software que dificulta los cambios en tiempo de ejecución, incluso en las cosas más sencillas. Entre más módulos se afecten por los cambios, más rígido es el sistema (C. Martin & Martin, 2006).

CAPÍTULO 1.- INTRODUCCIÓN

En la actualidad, la producción de 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 los desarrolladores de software, dichas capacidades son difíciles de ejercer y aún más difíciles de usarlas conjuntamente.

Cuando los desarrolladores de software carecen de experiencia y habilidad en el desarrollo, se producen unidades de programas que técnicamente quedan a deber, lo cual se conoce como "Deuda Técnica" (DT), concepto introducido por Ward Cunningham en 1992 (Cunningham, 1992).

La deuda técnica hace referencia a que un programa cumple con los objetivos, sin embargo, éste no lo hace de la mejor manera y por lo tanto puede reducir su tiempo de vida útil. Esto se debe a que el software cuenta con deficiencias que deben ser atendidas, lo cual aumenta su fragilidad y dificulta su mantenimiento, de no hacerse de manera oportuna, como en toda deuda, los intereses a pagar serán mayores.

Si bien se reconoce este fenómeno de forma general, lo que sigue sin reconocerse, es cómo la DT se manifiesta y afecta específicamente los procesos de software, y cómo las técnicas de desarrollo de software empleadas acomodan o mitigan la presencia de esta deuda (Holvitie et al., 2018).

La carencia de habilidades, experiencia, presiones de tiempo y costo en el desarrollo de sistemas de software y unidades de programa, conduce a la práctica y decisiones incorrectas que derivan en la producción de código desagradable (del inglés: "Code Smell") (Fowler et al., 1999), y finalmente dan paso a la acumulación de deuda técnica. Código desagradable se refiere a ciertas estructuras en el código que violan principios fundamentales de diseño e impactan negativamente en su calidad.

Técnicamente, el código desagradable no es incorrecto, no son defectos ni impiden el correcto funcionamiento del programa, más bien indica cierta debilidad en el diseño que puede degradar su desarrollo e incrementar el riesgo de fallas. Estos cumplen con su meta de valor u objetivo, aunque no de la mejor manera, y que, posiblemente, pueden presentar algún problema de *modularidad*, *fragilidad*, *rigidez*, *movilidad*, *autonomía*, etc.

Algunos ejemplos de incorrectas decisiones de diseño que originan código desagradable son:

- Arquitecturas de clases (módulos o paquetes de programa): Arquitecturas con carencia de abstracciones, abstracciones incorrectas, más de una responsabilidad por arquitectura, uso indiscriminado de la herencia, excesiva herencia de implementación, profundas jerarquías de herencia de clases, demasiadas clases hermanas derivadas de una misma clase, altos factores de acoplamiento entre clases, etc.
- Clases: clases demasiado grandes, clases muy pequeñas, clases con excesiva dependencia de implementación de otras clases, violación a la regla de sustitución de Liskov, incorrecto encapsulamiento (sin restricciones de visibilidad), clases complejas con muchos datos y/o métodos, clases con más de una responsabilidad (clases con sobrados métodos y datos para cumplir con una meta de valor u objetivo),

clases insuficientes e incompletas (clases que dependen del servicio de otras para alcanzar su meta de valor), clases con métodos nulos, clases con variables huérfanas, clases con excesiva complejidad ciclomática, etc.

Estos factores impiden el reuso, y posiblemente afectan la capacidad de mantenimiento (Kruchten et al., 2012) y encarecen la producción del software y el control de su evolución en el tiempo.

Una causa que origina deuda técnica en el software existente es debido al código desagradable en clases de objetos que exhiben más de un motivo para cambios porque no es conforme con el principio de diseño de *"Única Responsabilidad"*.

El concepto de única responsabilidad es un término relativo, que depende de la estructura interna de un componente o entidad de software, porque para atender un servicio, éste puede ser a nivel de función, de requerimiento, de un subsistema o un sistema completo; y depende de la capacidad funcional requerida en esos niveles de abstracción.

Por ejemplo, en su mínima expresión, una función cuya responsabilidad es alcanzar la meta con una única operación alcanzable, o bien puede ser, en su máxima expresión, un sistema completo cuya responsabilidad sea alcanzar la meta de gestión de un proceso de negocios. En estos dos extremos existen niveles intermedios tales como: una clase de objetos con responsabilidad cuya meta de valor es la representación del comportamiento *de una* entidad del mundo real; o bien comunidades de clases en colaboración, cuya meta es la responsabilidad de satisfacer requerimientos específicos.

Si se puede pensar en más de un motivo de cambio en una arquitectura modular, entonces esa arquitectura puede que tenga más de una responsabilidad, lo cual es difícil de ver. El efecto indeseable que se tiene en arquitecturas de software con esta característica radica en que arquitecturas modulares que tienen más de una responsabilidad llegarán a estar altamente acopladas, con bajos niveles de cohesión y de coherencia.

A continuación, se presenta el diseño e implementación de un método que tiene como objetivo reducir el número de responsabilidades en arquitecturas modulares. También se realizó el diseño e implementación de una métrica que mide el número de responsabilidades en arquitecturas modulares orientadas a objetos.

Organización de este documento de tesis

El presente trabajo de investigación está compuesto por siete capítulos, una sección de referencias y una sección de anexos. A continuación, se describe de forma general los capítulos que conforman esta tesis.

Capítulo 2. Antecedentes.

Se describe el problema planteado en la investigación, los objetivos de la investigación, así como también los trabajos antecedentes realizado en el CENIDET y los trabajos relacionados.

Capítulo 3. Marco teórico.

Este capítulo está integrado por los conceptos utilizados durante la investigación.

Capítulo 4. Materiales y métodos de solución.

Se describe la métrica, la cual fue definida, diseñada y desarrollada en esta investigación. Esta métrica es NR (Número de Responsabilidades).

Capítulo 5. Desarrollo del sistema.

Se describe el proceso para el desarrollo de la solución propuesta, la cual consiste del método de refactorización de separación de responsabilidades. Este proceso está conformado por el modelado del sistema en diagramas de casos de uso, diagramas de secuencia, diagramas de clases, descripción de escenarios y análisis de escenarios e implementación.

Capítulo 6. Evaluación de pruebas.

Se muestra el plan de pruebas, donde se definen los objetivos de las pruebas y el diseño de las pruebas para comprobar el correcto funcionamiento tanto del método de refactorización como de la métrica.

Capítulo 7. Conclusiones y trabajos futuros.

Se presentan las conclusiones, aportaciones de esta tesis, trabajos a futuros y hallazgos.

CAPÍTULO 2.- ANTECEDENTES

En este capítulo se describe el problema planteado en la investigación, los objetivos de la investigación, así como también los trabajos antecedentes realizado en el CENIDET y los trabajos relacionados.

2.1 Planteamiento del problema

Una causa que origina deuda técnica es cuando existen arquitecturas modulares, clases o unidades de programa con más de una responsabilidad, esto genera que se impida la aplicación del principio de única responsabilidad exhibiendo problemas de rigidez, fragilidad, extensibilidad e inmovilidad, lo que previene su reúso y dificulta su mantenimiento.

2.2 Solución Propuesta

La propuesta de solución consiste en desarrollar un método que automatice la detección, valoración y corrección de código desagradable presente en los activos de software existentes; por el quebranto del principio de única responsabilidad; para separar las secuencias de métodos interactivos que denotan responsabilidades en clases diferentes, y relacionarlas por composición-delegación de clase para reunir la implementación de servicios. Este método partirá de un analizador léxico-sintáctico con las acciones semánticas que identifique la situación de duplicidad de responsabilidades y que recabe la información suficiente que permita cuantificar el nivel del factor de deuda técnica generada por este código desagradable, para cumplir con los criterios y principios de diseño; y en consecuencia mejorar la genericidad, autonomía, flexibilidad, y extensibilidad de entidades de software.

Las metas u objetivos de unidades reusables de programa son el uso correcto de las relaciones entre clases, el nivel autosuficiente de granulación, así como la satisfacción de los criterios de modularización de programas, tales como: Descomposición; Composición; Legibilidad; Continuidad; y Protección modular; así como de los principios de diseño tanto modular como orientados a objetos, tales como: El uso de Unidades Lingüísticas; Pocas Interfaces (Alta Cohesión); Pequeñas Interfaces (Acoplamiento débil); Interfaces Explícitas; Ocultamiento de Datos; y los principios orientados a objetos el principio; de Abierto-Cerrado; Inversión de Dependencias; Sustitución; Única Responsabilidad; No-Repetición; y Separación de Interfaces.

El usuario deberá solicitar el servicio por medio de un menú de opciones y deberá seleccionar la (las) unidad (es) de programa escrita(s) en lenguaje Java que requieren de la refactorización. Al final de la generación de código refactorizado, el servicio deberá cuantificar el nivel del factor de deuda técnica prevaleciente, para medir el nivel de mejora, así como evaluar la conservación de la funcionalidad del código de entrada. Se debe considerar que el código original a refactorizar posee cierto comportamiento funcional el cual debe ser preservado totalmente después del proceso de refactorización.

2.3 Justificación

Actualmente, no existe en la literatura especializada referencia a métodos de refactorización que mejoren la autonomía de las arquitecturas modulares de objetos que exhiben más de una responsabilidad, por lo que no se satisface el principio de abierto/cerrado. Más de una responsabilidad implica más de un objetivo o meta de valor por lo que existe más de una

razón de cambio, aumentando la fragilidad de las entidades de software. La fragilidad origina que partes inesperadas del sistema dejen de funcionar ante cambios a los requerimientos originales o por extensión de nuevos requerimientos o por defectos encontrados. Al aparecer nuevos problemas en áreas que aparentemente no tienen relación conceptual con el área que sufrió un cambio, se decrementa la credibilidad del personal de diseño y mantenimiento.

Más de una responsabilidad implica también que es difícil de reusar cada una de estas responsabilidades en otras aplicaciones, porque estas no pueden ser separadas de la entidad de software actual. Esta característica es conocida como inmovilidad, y se requiere de mucho re-trabajo para separar la porción deseable del diseño de las porciones no deseables. Se estima que la mayoría de veces el costo de esta separación es más alto que el costo del desarrollo del diseño.

2.4 Objetivos

2.4.1 Objetivo General

El objetivo general de este trabajo de tesis es reducir la deuda técnica de sistemas de software existentes que es producida por el código desagradable derivado de arquitecturas modulares que contienen más de una responsabilidad.

2.4.2 Objetivos Específicos

- Reducción de código desagradable motivado por cambios a requerimientos o por defectos encontrados.
- Aumentar los Factores coherencia y cohesión de clases en Arquitecturas Orientadas a Objetos.
- Extender la funcionalidad del “SR2-Refactoring”, para dar soporte a sus métodos de refactorización de alto impacto en código escrito en lenguaje Java.

2.5 Antecedentes

En apoyo a este planteamiento, en el laboratorio de Ingeniería de Software del TecNM/CENIDET, se han planteado varios proyectos de reingeniería y refactorización para extender el tiempo de vida útil y para el reúso del software existente, a través de métodos de refactorización de alta y baja escala; así como de métodos de transformación, que operan para código escrito en lenguajes de programación C++ y Java.

Entre estos proyectos se cuenta con el desarrollo de herramientas para A) Adaptar interfaces lógicas que no empatan a las necesidades de los clientes; B) medir el problema de interfaces no utilizadas en marcos de aplicaciones orientados a objetos y refactorizar para separar esas interfaces no utilizadas; C) medir el factor de acoplamiento que produce dependencias y refactorizar para reducir este factor al mínimo indispensable; D) refactorización de marcos orientados a objetos con abstracciones incorrectas, que equilibran las métricas DIT y NOC; E) refactorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos; F) refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles, lo cual viola el criterio de protección modular; G) refactorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos; y H) refactorización de arquitecturas de software con carencia de abstracciones. Algunos de estos métodos de refactorización operan para código fuente escrito en lenguaje C++, mientras que otros operan en código fuente escrito en lenguaje Java.

Actualmente se está trabajando para que todos los métodos operen para ambos lenguajes de programación.

2.5.1 SR2-Refactoring

La herramienta principal de antecedente es la SR2-Refactoring (Sistema de Reingeniería para Reúso). Este sistema implementa varios métodos de refactorización, y cuatro más que actualmente se encuentran en desarrollo, que en conjunto permiten mejorar la arquitectura de marcos de aplicaciones orientados a objetos de dominios, tres de los cuales atienden código escrito en lenguaje C++ y 5 de ellos atienden código escrito en lenguaje Java.

Actualmente el sistema implementa varias métricas para detectar problemas específicos de diseño en los marcos de aplicaciones orientados a objetos. El sistema SR2-Refactoring cuenta con un menú, para realizar las acciones de refactorización y cálculo de métricas, así como acciones adicionales a los métodos de refactorización, como son la selección y comparación de archivos y el manejo de usuarios.

La mayoría de las opciones de los menús llevan a pantallas o cuadros de diálogo, y cada uno de ellos cuenta con elementos gráficos, como botones y cuadros de texto. Actualmente se tiene un proyecto desarrollado en un SaaS desplegado en la nube que ofrezca los diferentes servicios de refactorización tanto de alta escala como de baja escala.

El sistema SR2-Refactoring fue implementado en lenguaje Java, utilizando el ambiente Eclipse, y como soporte utiliza el manejador de Base de Datos MySQL. Algunos métodos de refactorización que se encuentran actualmente en desarrollo y que serán aporte para el sistema SR2-Refactoring son los siguientes:

1. “Método de refactorización de arquitecturas de software con carencia de abstracciones”, Ricardo Tello Díaz.
2. “Refactorización de módulos colaborativos con carencia de abstracciones”, Fernando Sánchez Rogel.
3. “Plug-In para integrar los métodos de refactorización del SR2-Refactoring al IDE-Eclipse”, Leslie Alejandra Ruíz Bustos.
4. “Reducción de la deuda técnica para la fragilidad modular de arquitecturas de software legado”, Miguel Romero Meza.
5. “Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado”, Elías A. Ramírez García

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

El objetivo general de esta tesis es instrumentar un mecanismo basado en patrones de diseño, que permita refactorizar la arquitectura de clases de MAOO’s escritos en lenguaje de programación C++ que presentan el problema de dependencia de interfaces.

Se propone un método de refactorización denominado Separación de Interfaces, cuyo algoritmo fue implementado satisfactoriamente en una herramienta que hace refactorización de manera automática.

2.5.3 Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter (Santos Castillo, 2005).

En esta teses se presenta un método de adaptación de interfaces, usando el patrón de diseño Adapter. Este método se implementó en un sistema computacional, que adapta la interfaz entre un cliente y un servicio del MAOO, bajo tres circunstancias diferentes. Con el método desarrollado se logra un proceso de adaptación transparente para el cliente y para el MAOO, ya que no es necesario modificarlos para que el cliente haga uso de un servicio del MAOO, respetando los principios de diseño orientado a objetos.

2.5.4 Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator (Robledo Cárdenas, 2014).

El objetivo de esta tesis es presentar un método de refactorización para reducir el acoplamiento entre clases de marcos de aplicaciones orientados a objetos, incorporando a su arquitectura una estructura dirigida por el patrón de diseño 'Mediator'.

La idea es que en lugar de que las clases estén acopladas porque se comunican entre sí, se comuniquen a través de un módulo intermediario, cuya finalidad sea la de realizar dicha comunicación de manera indirecta.

Este algoritmo fue implementado en una herramienta que realiza refactorización de manera automática, la cual incluye la implementación de la métrica que permite calcular los niveles de acoplamiento (Métrica del Factor de Acoplamiento COF) entre las clases antes y después de haber aplicado el proceso de refactorización.

2.5.5 Método de Refactorización de código java con interfaces y abstracciones incorrectas (Padilla Salgado, 2019).

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 reúso. Mediante la mejora de la generalidad y la flexibilidad; así como una mejor autosuficiencia y autonomía de entidades de software.

En esta tesis se implementa una métrica orientada a objetos que mide el grado de dependencia debido a interfaces que no se ocupan.

2.5.6 Refactorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos (Ortiz Gutiérrez, 2020).

El objetivo de esta tesis es mejorar la flexibilidad y la autonomía del software legado orientado a objetos escrito en lenguaje Java, está encaminado a lograr su mayor calidad para facilitar su reúso y su mantenimiento. Se diseñó e implemento un método de refactorización que disminuye el acoplamiento por herencia de implementación en arquitecturas de clases orientadas a objetos de sistemas de software existente.

El método tiene como propósito reducir la interdependencia entre objetos y clases por herencia de implementación, las métricas están enfocadas a medir el factor de herencia de implementación y el factor de flexibilidad de aplicaciones existentes orientadas a objetos.

2.5.7 Método de refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles (Barón Pérez, 2019).

El objetivo de esta tesis es aplicar el criterio de protección modular. Es decir, que sólo las funciones que son iniciadores de Casos de Uso o funciones abstractas en clases abstractas o

funciones de interfaces; todas de acceso externo en el código de MAOO's estén certificadas con el alcance "Public".

El proyecto incluye el cálculo de la protección modular de los servicios de software reusables obtenidos desde el software legado de MAOO's, así como del número de servicios obtenidos antes y después de la refactorización. Actualmente, no existe una métrica para medir el grado de protección modular; sin embargo, en este proyecto de tesis se propone una métrica sustentada en la teoría de la medición.

2.6 Trabajos Relacionados

Identification and Application of Extract Class Refactoring's in object-oriented systems (Fokaefs et al., 2012).

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 Plug-in de Eclipse, el método incluye un cálculo de 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 mejora la calidad del diseño del sistema subyacente.

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

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

Al evaluar métricas como Fan-in, Fan-out (métricas estructurales), WMC, NOC, DIT, CBO, RFC y LCOM (Métricas orientadas a objetos) se abogó por la posibilidad de mejorar la modularidad de un sistema orientado a objetos mediante la combinación de las métricas CBO, LCOM, Fan-in y Fan-out. Dentro de los experimentos se proporcionan valores de varias características de la estructura (clases, métodos y atributos) y las métricas calculadas para varios sistemas de software. Estos valores son comparados con los tiempos del uso de

una sola CPU y una GPU Tesla. La ganancia de tiempo obtenida por el GPU que se obtuvo es buena en comparación con la CPU.

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

JDeodorant: Identification and Application of Extract Class Refactoring's (Fokaefs et al., 2011).

Los cambios generalmente incluyen la corrección de errores, la incorporación de nuevas funciones, la modificación y extensión de la base de código. El ajuste del diseño, la evolución del equipo de desarrollo y sus prácticas. Esto resulta en módulos complejos, difíciles de manejar, poco elegantes, difíciles de entender y mantener, que en el caso del software Orientado a Objetos se manifiestan como "God Classes"

El problema de las "God Classes" puede abordarse mediante varias refactorizaciones, las cuales tienen como objetivo extraer algunos de sus datos y funciones en otras clases. Los elementos extraídos se pueden redistribuir a los colaboradores de la "God Classes" (a través de las refactorizaciones de "Move Method" y "Move Attribute") o re-empaquetarlos como nuevas clases separadas (a través de "Extract Class"). En principio, la última solución es preferible, ya que es menos probable que reduzca la cohesión de las clases receptoras.

El artículo presenta una extensión de Eclipse: JDeodorant, esta herramienta emplea una técnica de agrupamiento para identificar oportunidades de refactorización de Extract Class, en todas las clases del sistema para sugerir refactorizaciones que mejoren el diseño general del sistema.

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

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

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

La evaluación del trabajo se divide en tres fases: La primera es obtener tanto conocimiento como sea posible sobre el problema, en la segunda fase, se implementan las propuestas de investigación de acuerdo a la información, y en la tercera fase, los resultados se validarán con un estudio de caso. Se concluye que la estrategia seguida por el método MORe puede utilizarse por métodos similares que apunten a automatizar la refactorización, con respecto a diferentes propiedades de calidad.

AutoRefactoring: A platform to build refactoring agents (Fonseca et al., 2015).

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

En este trabajo se propone una plataforma basada en agentes, llamada plataforma AutoRefactoring, que permite el desarrollo de un agente para realizar de manera autónoma la refactorización. Para aplicar de manera segura una refactorización, se deben considerar varios problemas: (1) identificar las partes del código que deben mejorarse; (2) determinar los cambios que deben aplicarse al código para mejorarlo; (3) evaluar los impactos de las correcciones en la calidad del código; y (4) verificar que el comportamiento observable del software se conservará después de aplicar las correcciones.

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

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

Los métodos predeterminados de Java 8, que permiten que las interfaces contengan implementaciones de métodos (de instancia) que son útiles para el patrón de diseño *"Skeletal implementation"*. El objetivo de la investigación es mejorar la estructura del código de java heredado.

Se utiliza un enfoque de refactorización para la migración a los métodos predeterminados de la interfaz mejorada de Java 8. Se abren las clases a la herencia permitiendo que las clases hereden múltiples definiciones de interfaz.

En la experimentación se implementó el enfoque como un plug-in de Eclipse y se aplicó a 19 proyectos Java del mundo real, así como a solicitudes de extracción enviadas a los populares repositorios de GitHub. Los resultados muestran que la herramienta puede escalar y refactorizar, a pesar de sus limitaciones de lenguaje y conservatividad.

En este trabajo se concluye que, de acuerdo al estudio, se destaca el uso de patrones de código y brinda información a los diseñadores de "idioms" sobre la aplicabilidad al software existente. Como trabajo complementario se puede explorar variaciones de patrones, por ejemplo, permitiendo métodos de entrada de clases concretas, lo que requiere analizar instancias y determinar métodos predeterminados adecuados de clases concretas.

Análisis de Dependencias entre Refactorings para Solucionar Code Smells (Marcos et al., 2018).

Para solucionar código desagradable (Smell) se deben aplicar un conjunto de refactorizaciones que permitan reestructurar el sistema. Sin embargo, al aplicar un conjunto

de refactorizaciones en un orden determinado, pueden surgir problemas que impidan que éstos se apliquen. Por estos motivos, para aplicar un conjunto de refactorizaciones, se deben analizar las dependencias que existen entre estos para poder establecer el orden de aplicación.

Esta investigación presenta una herramienta que identifica y soluciona los conflictos originados por dependencias entre refactorizaciones para luego aplicar automáticamente los mismo. La herramienta que se presenta se llama RAtool (Refactorings Analysis Tool), la cual realiza un análisis de dependencias entre las refactorizaciones recomendadas para solucionar los problemas identificados por código desagradable.

En este trabajo, se propone un enfoque llamado Refactoring Analysis (RA) que permite analizar las dependencias entre refactorizaciones para solucionar los párrafos de código *desagradable* de un sistema. El objetivo del enfoque es encontrar un orden correcto de aplicación de las Refactorizaciones y resolver los conflictos que existen entre ellos. El enfoque recibe como entrada el código del sistema, junto con los *códigos desagradables* y las Refactorizaciones asociadas para solucionarlos. Como salida, se obtiene el código del sistema refactorizado.

El enfoque se divide en tres fases: fase de análisis, fase de resolución de conflictos y fase de generación de código. La fase de análisis, recibe la lista de Refactorizaciones e identifica los conflictos que se generan por la aplicación en cascada de los mismos. Una vez concluida la fase de análisis, se obtiene una lista ordenada de Refactorizaciones y los conflictos que existen entre ellos. Los conflictos, se toman como parámetro de entrada de la segunda fase, denominada fase de resolución de conflictos. Durante esta fase, se utiliza un conjunto de reglas para corregir los conflictos identificados en la fase anterior. Esta fase genera como salida una lista ordenada de Refactorizaciones sin conflictos. En la última fase, la de generación de código, se aplican las Refactorizaciones sobre el código, generando como salida el código refactorizado. El enfoque ha sido materializado en la herramienta RAtool la cual es un Plug-in para Eclipse.

JSPIRIT: a flexible tool for the analysis of code smells (Vidal et al., 2015).

En este artículo se presenta JSpIRIT, es una herramienta que permite identificar código desagradable (Code Smell) en un sistema Java. El enfoque permite priorizar los códigos desagradables encontrados en base a diferentes criterios, tales como, historia de la aplicación, impacto de los códigos desagradables en escenarios de modificabilidad o la relevancia de los códigos desagradables. Sin embargo, la tarea de refactorizar los códigos desagradables encontrados es delegada al desarrollador.

Es una herramienta flexible para priorizar la deuda técnica presentada en código desagradable (Code Smell). JSpIRIT permite al usuario centrarse en el código desagradable que es crítico para el sistema, lo que hace que su análisis sea rentable. El diseño de JSpIRIT admite un esquema de priorización basado en múltiples criterios, que los desarrolladores pueden proporcionar y ampliar fácilmente.

JSpIRIT es ligero porque puede funcionar con un pequeño conjunto de criterios y proporcionar una clasificación útil a los usuarios. La flexibilidad de la arquitectura de la herramienta para soportar nuevos criterios se evaluó instanciando JSpIRIT con tres criterios. JSpIRIT se puede extender con un esfuerzo razonable para priorizar los códigos desagradables con un amplio tipo de criterios. Además, JSpIRIT permite la combinación de

diferentes estrategias para identificar aglomeraciones de código desagradable, así como la creación de estrategias de priorización flexibles para clasificar los códigos desagradables.

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

Resultado o Producto: El cual consiste en una herramienta (H), un análisis (A), un plug-in para eclipse (P) o una extensión (EXT).

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

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

Tipo de Proceso: Dentro de los procesos utilizados en los trabajos relacionados se clasifican dependiendo de la intervención del usuario, los cuales pueden ser: Automático (A), sin ninguna intervención del usuario, Semi-automático (S), con una mínima intervención del usuario y Manual (M), se refiere a una máxima intervención del usuario en el proceso.

Métodos de Refactorización: Consiste en los diferentes métodos de refactorización utilizados en los diferentes trabajos relacionados.

- Move Method.
- Extract Class.
- Move Attribute.
- Encapsulate Fields.
- Rename.

Métricas Usadas:

- Bajo Acoplamiento (CBO).
- Cohesión (LCOM).
- Coherencia de Caso de Uso (CrCU).
- Entity Placement (EP) que consiste en Alta Cohesión y Bajo Acoplamiento.
- Métodos Ponderados para la Clase (WMC).
- Cohesión de Clase Estrecha (TCC).
- Acceso a Datos Externos (ATFD).

La Tabla 1 muestra los resultados de la comparación de los diferentes trabajos de investigación

Tabla 1 Comparativa entre trabajos relacionados

Trabajo de investigación	Objetivo	Code Smell Identificados	Método de Refactorización	Aportación	Producto Resultante	Alcance	Tipo de Proceso	Métrica Usada	Resultados
(Fokaefs et al., 2012)	Reconocer oportunidades para simplificar clases grandes, complejas, difíciles de manejar y menos cohesivas utilizando la técnica “Extract Class".	God Class	Extract Class	M	H	IM	A	EP	Herramienta automática que Refactoriza extractos de clases, mediante la aplicación de un algoritmo de agrupamiento jerárquico
(Napoli et al., 2013)	Mejorar la modularidad de los sistemas de software a nivel de clases y métodos y reducir el tiempo de cálculo de métricas en grandes sistemas de software.	N/A	Solo cálculos de métricas	MD	A	ID	A	CBO LCOM	Algoritmo para reducir el tiempo de ejecución de métrica.
(Fokaefs et al., 2011)	Tiene como objetivo extraer algunos de sus datos y funciones en otras clases.	God Class	Move Method Move Attribute Extract Class	M	EXT	IM	A	N/A	Herramienta que emplea una técnica de agrupamiento para identificar oportunidades de refactorización de Extract Class.
(Bryton & Brito e Abreu, 2008)	Detectar código desagradable en los sistemas de Software	N/A	MORe Method	M	H	IM	A	N/A	Herramienta MORe Method para la refactorización automatizada.
(Fonseca et al., 2015)	Mejora la calidad del software en cuanto a reusabilidad, flexibilidad, efectividad y extensibilidad.	Public Fields Data Clumps Encapsulate Fields	Extract Class Move Method	M	EXT	IM	A	CBO LCOM	Se implementó un agente para la refactorización automática.
(Khatchadourian & Masuhara, 2017)	Mejorar la estructura del código java.	NA	Pull Up Method	M	P	IM	A	N/A	Se implementó un Plug-in de código abierto para el IDE de Eclipse.

(Marcos et al., 2018)	Encontrar un correcto orden de aplicación para las refactorizaciones y resolver los conflictos que existen entre ellos.	Feature Envy God Class Data Class Brain Class Shotgun Surgery	Extract Method Extract Class Move Method Rename	M	P	IM	A	CBO	Obtención de código refactorizado a través de un proceso establecido.
(Vidal et al., 2015)	Permitir la combinación de diferentes estrategias para identificar aglomeraciones de Code Smell, así como la creación de estrategias de priorización flexibles para clasificarlos.	Brain Class Brain Method Data Class God Class Shotgun Surgery	Move Method Move Field	M	H	IM	S	WMC TCC	Se puede extender con un esfuerzo razonable para priorizar los Code Smells con un amplio tipo de criterios.
Tesis	Disminución de deuda técnica producida por arquitecturas de clases con más de una responsabilidad.	Violación al principio de única responsabilidad.	Separación de responsabilidades.	M, A, MT	EXT – SR2 Refactoring	IM	A	NR CrCU LCOM	Método de refactorización para el sistema SR2 Refactoring. Métrica de Número de Responsabilidades.

En la Tabla 1 se puede observar que el sistema que se desarrolló en esta tesis tiene como objetivo separar las responsabilidades en arquitecturas modulares. Algunas diferencias de esta tesis con los trabajos relacionados son las siguientes:

1. El proceso es de “abajo hacia arriba” el cual realiza un análisis estático de código fuente, con lo cual se genera un Parser y un árbol de sintaxis abstracto (AST) a partir de la gramática de lenguaje Java.
2. Mediante el Parser se reconocen las secuencias de métodos, se extrae la información relativa para la separación de las responsabilidades.
3. Esta extensión mide el Número de Responsabilidades (NR) tanto del código fuente original, como del código refactorizado, de tal manera que se mide su nivel de mejora.
4. El diseño e implementación de la métrica NR, fue sustentada por la teoría de la medición.

CAPÍTULO 3.- MARCO TEÓRICO

Este capítulo está integrado por los conceptos utilizados durante la investigación, con la finalidad de otorgar al lector una buena comprensión sobre el tema.

3.1 Paradigma de Programación Orientada a Objetos

El Paradigma de la Programación Orientación a Objetos (POO) se utiliza mucho en la ingeniería de software moderna, permite manejar ampliamente las nuevas plataformas que garanticen desarrollar aplicaciones robustas, portables y reutilizables que puedan ofrecer una solución a largo plazo en un mundo donde los cambios se dan a cada momento.

La Programación Orientada a Objetos es una metodología para el desarrollo lógico de programas que se remonta a la década de 1960, dicha programación no es la solución definitiva, pero permite construir programas complejos a partir de entidades de software más simples llamadas *objetos*, que son instancias reales o muestras de clases. Esto mejora la productividad del programador y facilita la extensión y la reutilización de clases en otras aplicaciones (Velarde de Barraza et al., 2006).

Uno de los objetivos de la Programación Orientada a Objetos es la reusabilidad. Cuando se desarrolla un sistema aplicando el Paradigma Orientado a Objetos, la clase proporciona los mecanismos de encapsulamiento, abstracción y ocultación de la información, además de ser un componente elemental en la reutilización (García Peñalvo et al., 1997).

3.1.1 Clase

Una clase es una plantilla para un objeto. Esta encapsula los datos (*variables o campos*) y de funciones (*métodos*) que operan sobre esos datos (Morelli & Walde, 2016).

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones cuya ejecución conoce; dichas acciones se conocen como servicios o métodos. Una clase también incluye todos los datos necesarios para describir los objetos creados a partir de la clase. Los cuales se conocen como atributos, variables o variables instancia; el primer término se utiliza en análisis y diseño orientado a objetos, los otros, en programación orientada a objetos (Joyanes Aguilar & Zahonero Martínez, 2007).

La declaración de una clase consta de palabras reservadas e identificadores: una secuencia opcional de modificadores, la palabra reservada *class*, el nombre de la clase, un nombre opcional de la clase padre, una secuencia opcional de interfaces y el cuerpo de la clase con sus miembros, en la Figura 1 se muestra un ejemplo de declaración de una clase.

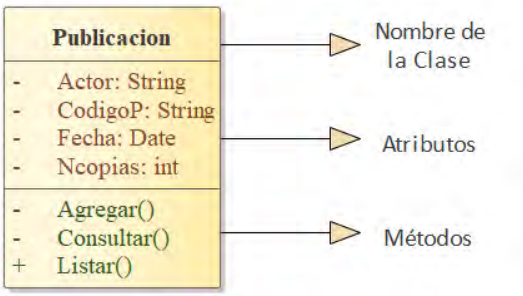

Figura 1 Modelo conceptual de una clase

3.1.2 Objeto

Los objetos son sujetos o cosas tangibles de nuestro mundo real que tienen un estado y un comportamiento. En POO, un objeto está compuesto por datos y métodos. Estos objetos tienen la estructura y el comportamiento definido por la clase, y muestra el comportamiento reflejado por sus operaciones (métodos), que actúan sobre sus datos. Los métodos son el único medio para acceder a los datos del objeto; no es posible hacerlo directamente. Los datos están ocultos y eso asegura que no puedan modificarse por accidente por métodos externos al objeto (Velarde de Barraza et al., 2006).

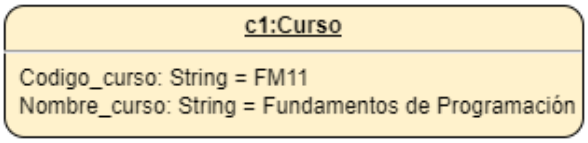

Figura 2 Ejemplo de Objeto c1 de la clase Curso

3.1.3 Atributos de Clases

Los atributos son las características individuales que diferencian a un objeto de otro; el conjunto de ellos constituye su estado; cada objeto almacena información acerca de su estado actual y en un momento dado éste corresponde a una selección determinada de valores posibles de los diversos atributos.

El estado de un objeto puede cambiar con el tiempo como consecuencia de llamadas a métodos. Es una clase, los atributos se definen mediante variables, las cuales son sitios donde se almacena la información de un programa de computadoras.

3.1.4 Métodos de Clases

Un método (*función)* es un conjunto de instrucciones o sentencias que realizan una determinada tarea (Velarde de Barraza et al., 2006), los cuales se encuentran definidos dentro de una clase; estos se identifican con un nombre y puede o no devolver un valor. En los lenguajes orientados a objetos, los métodos son operaciones o servicios que describen un comportamiento asociado a un objeto.

El comportamiento de un objeto es el conjunto de capacidades y aptitudes que describen sus operaciones, funciones y reacciones; además responde a lo que se puede hacer con dicho objeto o a los métodos que se le pueden aplicar.

Cuando se llama a un método, el flujo de control del programa se transfiere al método y se ejecutan una a una las instrucciones que lo integran; cuando se ejecutan todas, el control regresa al punto desde donde se hizo la llamada y posteriormente el programa continúa con la siguiente sentencia o instrucción. En la Figura 3 se muestra un ejemplo de flujo de control de llamada a métodos.

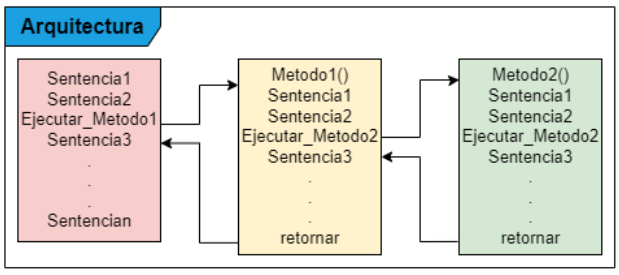

Figura 3 Flujo de control de llamadas a métodos

3.1.5 Calificadores de Alcance

Un principio fundamental en programación orientada a objetos es la ocultación de la información; esto significa que no se puede acceder por métodos externos a la clase a determinados datos internos.

Para controlar el acceso a los miembros de la clase se utilizan diferentes calificadores de acceso: *default, privados*, *públicos* y *protegidos,* y se pueden aplicar a los atributos (datos) y métodos de una clase. A continuación, se especifica con más detalle cada uno de los modos de acceso (Velarde de Barraza et al., 2006):

- *Miembros default (friendly ~).* Es el valor por defecto si no se indica alguno, solo las clases en el mismo paquete pueden acceder a la propiedad o método.
- *Miembros privados (-).* Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos.
- *Miembros públicos (+).* Son aquellos a los que puede accederse desde dentro de la clase y también por parte del objeto.
- *Miembros protegidos (#).* Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos. Es importante aclarar que este tipo de acceso se asocia directamente con la herencia.

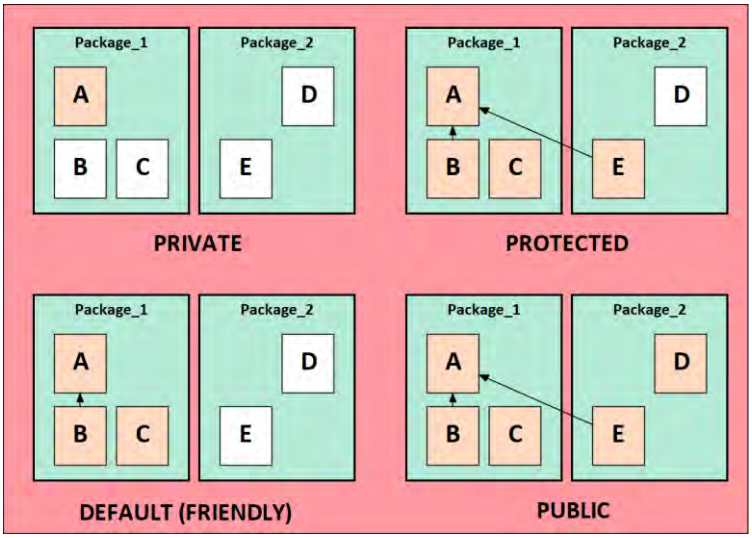

Figura 4 Representación del acceso de los calificadores de alcance

3.2 Principios de Diseño Orientado a Objetos

Como parte de una estrategia general para la programación, se propone una serie de principios de diseño orientado a objetos para el diseño y programación de sistemas informáticos, para que estos sean fáciles de mantener y extender en el tiempo. Estos principios son pautas para que los programadores las apliquen en el desarrollo de software para eliminar el código desagradable.

3.2.1 Principio de Diseño de "Única Responsabilidad"

Durante el trabajo de esta investigación se trata específicamente una causa que origina deuda técnica en el software existente, debido al código desagradable en arquitecturas modulares que exhiben más de un motivo para cambios porque no son conformes con el principio de diseño de "Única Responsabilidad".

El principio de diseño de única responsabilidad conocido como *"Single Responsibility Principle (SRP)"* establece que una clase o módulo debe tener una, y sólo una razón para cambiar (C. Martin, 2018).

La Figura 5 muestra una arquitectura con una única responsabilidad, dicha responsabilidad está dada por la siguiente secuencia de métodos:

Cliente → Metodo_1 → Metodo_2 → Metodo_3 → Metodo_5 → Metodo_6 → Metodo_7

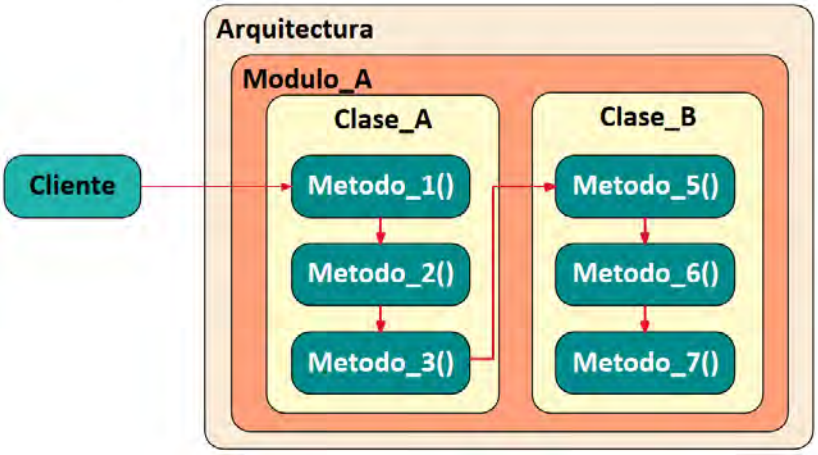

Figura 5 Arquitectura con una única responsabilidad

En la Figura 6 se puede observar una arquitectura con dos responsabilidades, las responsabilidades están dadas por las siguientes secuencias de métodos.

Responsabilidad 1: Cliente → Metodo_1 → Metodo_2 → Metodo_3 → Metodo_5 → Metodo_6 → Metodo_7

Responsabilidad 2: Cliente → Metodo_1 → Metodo_2 → Metodo_3 → Metodo_4

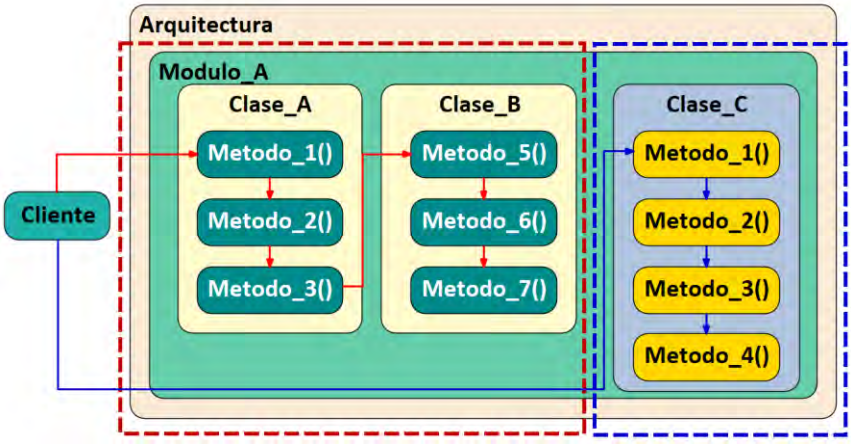

Figura 6 Arquitectura con dos responsabilidades

3.2.1.1 Módulo

En términos de software, consideramos que cada módulo proporciona una serie de subrutinas o funciones que pueden provocar cambios de estado y otras funciones o procedimientos que pueden dar a un programa de usuario los valores de las variables que componen ese estado (Lorge Parnas, 1972).

En este trabajo de investigación, se considera a un módulo de programa o paquete, a una secuencia funcional completa para atender a una responsabilidad o requerimiento.

3.2.1.2 Responsabilidad

Una responsabilidad está dada por una secuencia de métodos en arquitecturas modulares, dicha secuencia inicia con un *método público*, dentro de una *clase pública*. La secuencia está conformada por las invocaciones de un método a otro método. Una secuencia termina cuando un método ya no invoca a otro método y regresa el control hacía el método llamador y así sucesivamente.

3.2.2 Principio de Diseño "Abierto-Cerrado"

En (Meyer, 1997) acuñó el término del principio Abierto-Cerrado, que dice:

"Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertos para la extensión, pero cerrados para las modificaciones"

Si existe un solo cambio en un programa que da como resultado una cascada de cambios en los módulos dependientes, el diseño se vuelve rígido. Este principio, nos aconseja refactorizar el sistema para que más cambios de ese tipo no provoquen las modificaciones.

3.3 ANTLR

ANTLR es un poderoso generador de analizadores, que puede ser usado para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. Es ampliamente utilizado dentro de la industria y la academia para construir todo tipo de lenguajes, herramientas y frameworks. Es un analizador léxico y sintáctico para archivos, utiliza un archivo con la definición de la gramática a utilizar para el análisis (John Parr, 2013).

3.3.1 String Template

String Template (ST) es un motor de plantillas de Java para la generación de código, páginas web, correos electrónicos o cualquier otro formato de texto. String Template utiliza archivos con formato *".st"* para diseñar la plantilla de la información que se desea usar, utilizando métodos get para obtener la información que se requiere.

3.4 Refactorización

Dependiendo el contexto (Fowler, 2018) define el término Refactorización:

Como nombre o sustantivo: refactorización es "un cambio realizado en la estructura interna del software para hacerlo más fácil de entender y más económico de modificar sin cambiar su comportamiento observable".

Como verbo: refactorización es la "reestructura del software aplicando una serie de refactorizaciones sin cambiar su comportamiento observable".

Fowler combina ambas definiciones en una sola, quedando el término de refactorización como: "El proceso de cambiar un sistema de software, de tal forma que no se altere su comportamiento externo, aunque mejore su estructura interna".

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.

3.5 Teoría de la Medición

La medición sirve como una forma útil de planificar y controlar de manera sólida el desarrollo y la ejecución de proyectos de software durante el ciclo de vida del mismo. Sin embargo, al realizar mediciones, es importante conocer los niveles de escalas del proceso de medición.

La medición puede definirse como la asignación de números a objetos y eventos de acuerdo con ciertas reglas; la manera como se asignan esos números determina el tipo de escala de medición (Stevens, 1946).

Los números se asignan a los individuos de acuerdo con un procedimiento repetible cuidadosamente prescrito. La teoría de la medición es una rama de la estadística aplicada que intenta describir, categorizar y evaluar la calidad de las mediciones, mejorar la utilidad, la precisión y el significado (J. Allen & M. Yen, 1979).

Esto conduce a la existencia de diferentes tipos de escalas, por lo que el problema se transforma en explicar a) las reglas para asignar números, b) las propiedades matemáticas de las escalas resultantes, y c) las operaciones estadísticas aplicables a las medidas hechas con cada tipo de escala.

3.5.1 Conceptos básicos de la Teoría de la medición

Para las mediciones se pueden considerar dos sistemas relacionales: el *empírico* y el *sistema relacional formal.* Como primer punto se introducen algunas definiciones básicas; notación empírica, sistema relacional binario, medida y escala, definidas en (Zuse & Bollmann-Sdorra, 1992) y (Briand et al., 1996).

$$A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$$

Sea un sistema relacional empírico, donde A es un conjunto de objetos no vacío, R_i son relaciones ki-arias en A donde $i = 1, \dots, n$ y $o_j, j = 1, \dots, m$ son operaciones binarias cerradas en A. Por ejemplo:

Sistema Relacional Empírico:

$A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$

- A Es un conjunto no vacío de objetos empíricos que se van a medir (en nuestro caso, arquitecturas modulares).
- R_i Son relaciones ki-arias en *A* con *i=1, …, n,* por ejemplo, la relación empírica *"menor o igual que"*
- o_j Son operaciones binarias en objetos empíricos de *A* que van a ser medidos (por ejemplo, secuencia de métodos) con *j=1, …, m.*

$$B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$$

Sea un sistema relacional formal, donde B es un conjunto no vacío de objetos formales, por ejemplo; números o vectores, $S_1, i = 1, \dots, n$ son relaciones ki-arias en B y $\bullet_j, j = 1, \dots, m$ son operaciones binarias cerradas en B. Por ejemplo:

Sistema Relacional Formal:

$B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$

- B Es un conjunto no vacío de objetos formales (por ejemplo, números o vectores).
- S_i Son relaciones ki-arias en B con $i=1, \dots, n$, (por ejemplo, *"mayor que", "menor o igual que"*
- $\bullet_j$ Son operaciones binarias cerradas en B que van a ser medidos (por ejemplo, suma o multiplicación).

Una medida μ (la letra griega "mu") se define como un mapeo $\mu: A \to B$, que produce para cada objeto empírico $a \epsilon A$ un objeto formal (valor de medición) $\mu(a) \epsilon B$

Sea $A = (A, R_1, \dots, R_n, o_1, \dots, o_m)$ un sistema relacional empírico y $B = (B, S_1, \dots, S_n, \bullet_1, \dots, \bullet_n)$ un sistema relacional formal y μ una medida. La tripleta (A, B, μ) es una escala si y solo si para todo i, j y para todo $a_1, \dots, a_k, a, b \epsilon A$ sostiene que:

$$Ri(a_1, \dots, a_k) \Leftrightarrow Si\big(\mu(a_1), \dots, \mu(a_k)\big)$$

$$and$$

$$\mu\big(a\ o_j\ b\big) = \mu(a)\ \bullet_j \mu(b)$$

Si $B = R$ es el conjunto de los números reales, la Tripleta (A, B, μ) es una escala real.

3.5.2 Escalas o niveles de Medición

La medición puede ocurrir en diferentes niveles y la relación entre los valores asignados determina el nivel de medición.

Hasta la fecha, la mayoría de los psicólogos, científicos sociales e ingenieros suscriben los cuatro niveles jerárquicos de medición identificador por (Stevens, 1946): nominal, ordinal, de intervalo y de razón. Cada nivel de medición específica cómo los números que se asignan a los individuos se relacionan con la propiedad que se mide. Las primeras dos escalas (nominal y ordinal) son conocidas como escalas categóricas, y las dos últimas (intervalo y razón) como escalas numéricas.

3.5.2.1 Escala nominal

Representa la asignación de numerales menos restringida, la cual está compuesta por conjuntos de categorías en las que se clasifican los objetos. Dichos numerales son utilizados solo como etiquetas, además de utilizar palabras y letras. Una escala nominal es simplemente una colocación de datos en categorías, sin ningún orden o estructura.

Dentro de la Ingeniería de Software, un lenguaje de programación utilizado para un programa es una medida nominal, ya que permite que los programas se clasifiquen en diferentes categorías y no se disponga de ningún orden entre ellos.

Las escalas nominales son las menos informativas, pero pueden proporcionar información importante, por ejemplo, el lenguaje de programación utilizado es fundamental para interpretar modelos construidos sobre la cantidad de líneas de código.

3.5.2.2 Escala ordinal

Surge de la operación de ordenación de rangos. Se asignan números a los objetos en un orden particular, pero cualquier número que mantenga ese orden es igualmente bueno. Cualquier función creciente *t* es una transformación admisible.

De acuerdo con (Zuse & Bollmann-Sdorra, 1992) para poder clasificar una métrica como una escala ordinal, se debe de cumplir con el axioma de orden débil, el cual consiste en una relación binaria que debe de ser transitiva y completa:

Transitividad: $\mathrm{P} \leq\bullet\ \mathrm{P}', \mathrm{P}' \leq\bullet\ \mathrm{P}'' \Rightarrow \mathrm{P} \leq\bullet\ \mathrm{P}''$

Completes: $\mathrm{P} \leq\bullet\ \mathrm{P}'$ o $\mathrm{P}' \leq\bullet\ \mathrm{P}$

Para toda $\mathrm{P}', \mathrm{P}'' \epsilon P$, donde P es un conjunto y $\leq\bullet$ es una relación empírica binaria como *"igual o menos compleja que".*

Supóngase que $(\boldsymbol{P}, \leq\bullet)$ es un sistema relacional empírico, donde $\boldsymbol{P}$ es un conjunto contable no vacío y $\leq\bullet$ es una relación binaria en $\boldsymbol{P}$. Tenemos una función $\mu: P \rightarrow \Re$, con:

$$\mathrm{P}' \leq\bullet\ \mathrm{P}'' \ \Leftrightarrow\ \mu(\mathrm{P}') \leq\ \mu(\mathrm{P}'')$$

Para toda $\mathrm{P}', \mathrm{P}'' \epsilon P$, donde P, sí o solo si, $\leq\bullet$ es de orden débil. Si tal homomorfismo existe, entonces; $((P, \leq\bullet), (\Re, \leq), \mu)$ es una escala ordinal.

3.5.2.3 Escala de Intervalo

Las escalas de intervalo o cardinales son más refinadas, puesto que, además del orden o jerarquía entre categorías, las etiquetas o números consecutivos establecen intervalos iguales en la medición. Esta escala asigna números a los objetos de tal manera que el intervalo entre dos valores de medida es significativo en todo el rango de valores. Solo las funciones lineales positivas $t(x) = ax + b$ son transformaciones admisibles.

3.5.2.4 Escalas de razón o proporción

Las escalas de razón son aquellas ampliamente encontradas en física y sólo son posibles cuando existen operaciones para determinar todas las cuatro relaciones: *igualdad, ordenación de rangos, igualdad de intervalos e igualdad de razones*. Una vez que la escala es erigida, sus valores numéricos puedes ser transformados. Por ejemplo, se puede decir que un segmento de programa es el doble de largo que otro segmento de programa.

3.5.2.5 Escalas Absolutas

Este tipo de escalas son las más informativas, pero son usadas con muy poca frecuencia en la práctica. Por ejemplo, si se considera el atributo "número de líneas de código" de un segmento de programa (dicho atributo no es lo mismo que el tamaño, ya que es posible que se desee medir el tamaño a través de diferentes medidas).

CAPÍTULO 4.- MATERIALES Y MÉTODO DE SOLUCIÓN

Durante la investigación de este trabajo de tesis se realizó una búsqueda de métricas enfocadas a medir las responsabilidades en arquitecturas modulares. En dicha búsqueda revisada no se encontraron métricas que cumplieran con dicho propósito. En este trabajo de tesis se propone una métrica de calidad, que sirve para medir el número de responsabilidades existentes dentro de arquitecturas modulares: NR (Número de Responsabilidades). A continuación, se muestra la descripción de la métrica de calidad NR.

4.1 Métrica NR (Número de Responsabilidades)

Se definió la métrica NR (Número de Responsabilidades) para medir el número de responsabilidades (secuencias) existentes dentro de arquitecturas modulares, para así cumplir con el *"Principio de Única Responsabilidad"*. Esta métrica consiste en dividir el número de responsabilidades ideal, en este caso 1 entre la suma del número de secuencias existentes dentro de cada arquitectura modular (paquete/módulo).

La expresión matemática para la métrica NR es la siguiente:

$$NR = \frac{1}{\sum_{i=1}^{i=n} secuencia_i} \quad (1)$$

En donde:

NR = Número de responsabilidades

n = Número total de métodos públicos. Donde cada método público inicia una secuencia de métodos y cada secuencia es una responsabilidad.

Secuencia = Secuencia de método en una o más clases dentro de una arquitectura modular. Dichas secuencias inician con un método público. Las secuencias se forman por las invocaciones de un método a otro método. Una secuencia termina cuando un método ya no invoca a otro método y regresa el control hacia el método llamador y así sucesivamente.

La métrica fue normalizada con el objetivo de que los valores obtenidos estén dentro del rango del 0 al 1, el mejor valor es 1, lo cual significa que la arquitectura modular cuenta con una única responsabilidad. El valor menos deseado es el 0, lo cual significa que las responsabilidades tienden al infinito.

NR → A medida que el valor tienda a 0 indica que el número de responsabilidades tiende hacía el infinito.

NR → Una medida que tienda a 1 nos indica que el número de responsabilidades disminuye.

NR → Cuando el valor sea igual a 1, esto nos indicara que la arquitectura modular solo cuenta con una única responsabilidad

La métrica fue sustentada con base a la teoría de la medición como escala ordinal (Zuse & Bollmann-Sdorra, 1992), para dicho sustento de la métrica, se tomaron tres arquitecturas modulares diferentes, demostrando que cada una de ellas cumple con los

requerimientos del axioma de orden débil y que además se cumpla la propiedad de homomorfismo. Dicho sustento se puede encontrar a detalle en el *"Anexo A.- Sustento de la métrica NR como escala ordinal"*.

4.2 Método de refactorización de separación de responsabilidades.

En la Figura 7 se muestra el modelo BPMN (Business Process Model and Notation) del proceso general del método de refactorización de separación de responsabilidades, el cual está conformado por los subprocesos de a) análisis de código fuente, b) cálculo de métrica NR, c) refactorización y d) generación de código.

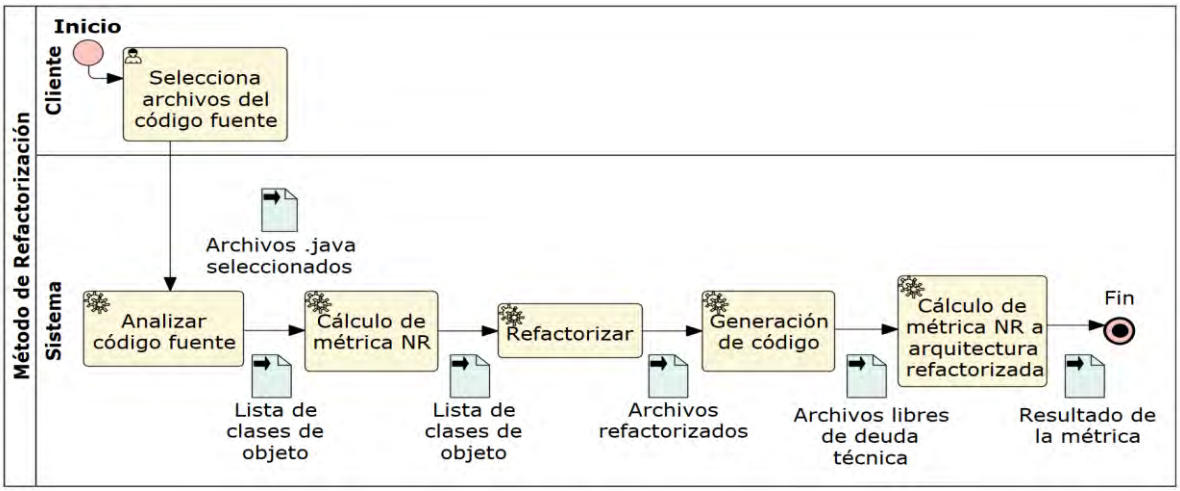

Figura 7 Proceso general del proceso de refactorización

El subproceso de análisis de código fuente recibe como entrada los archivos del código fuente original de la arquitectura modular que se requiere refactorizar, esto se realiza mediante un análisis sintáctico utilizando la herramienta ANTLR con la gramática del lenguaje Java en su versión 8.0. Se extrae la información necesaria para el subproceso de cálculo de métrica NR, así como también para la refactorización de separación de responsabilidades y generación de código refactorizado.

El subproceso de cálculo de métrica tiene como finalidad calcular el número de responsabilidades existentes dentro de arquitecturas modulares, antes y después de la refactorización.

El subproceso de refactorización recibe como entrada la información contenida en la lista de clases de objetos que resultada del proceso de análisis. Con dicha información en caso de existir más de una responsabilidad el sistema lleva a cabo la refactorización.

En el subproceso de generación de código se recibe como entrada el código refactorizado contenido en una lista de clases de objetos resultante del subproceso de refactorización. Esta información utiliza la plantilla *"String template"* para generar el código refactorizado equivalente en comportamiento a la arquitectura original.

4.2.1 Proceso de refactorización del método de separación de responsabilidades

En la Figura 8 se ilustra el método de refactorización de separación de responsabilidades. A continuación, se describen los pasos que conforman el algoritmo que implementa este método de refactorización.

1. Analizar el proyecto de entrada.
    1.1. Obtener todas las clases del proyecto.
    1.2. Analizar las clases que pertenecen a los módulos.
    1.3. Buscar los clientes del módulo, es decir, la dependencia hacía el módulo que se está analizando, para encontrar las clases que están siendo llamadas desde fuera del módulo.
2. Analizar cada uno de los clientes, para obtener las clases y los métodos que pertenecen a la(s) responsabilidad(es).
3. Almacenar cada uno de los métodos públicos que están siendo utilizados desde fuera del módulo.
4. Almacenar la clase y el método del módulo que está siendo llamado desde fuera del mismo.
5. Por cada uno de los métodos, se maneja un contador para saber cuántas veces está siendo utilizado fuera del módulo.
    5.1. Si el método iniciador de la responsabilidad es utilizado por dos clientes diferentes, dicha responsabilidad solo es contabilizada una vez, para más información sobre este tipo de escenarios ver Capítulo 5 Escenario 3 – Clientes.
6. Buscar las dependencias de las clases que tienen estos métodos de entrada.
7. Si las responsabilidades son mayores a 1, se lleva a cabo la refactorización.
    7.1. En caso de que exista una única responsabilidad (mejor de los casos), la arquitectura no es refactorizada.
8. Dividir la arquitectura dependiendo las responsabilidades encontradas.
    8.1. Cada clase que está siendo usada como entrada es llenada con la información correspondiente (atributos, métodos).

Precondiciones del Método de Refactorización de Separación de Responsabilidades

- El código de la arquitectura a analizar no debe contar con errores léxicos ni sintácticos.
- Los calificadores de alcance de la arquitectura a analizar deben de ser correctos, para ello tendrá que ejecutarse previamente el *"Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente"*.
- Debido a que este método de refactorización contabiliza las responsabilidades en arquitecturas modulares, también llega a contabilizar las responsabilidades a nivel de clase, esto implica que deba ejecutarse previamente el *"Método de Refactorización de Código Java para reducir las Dependencias entre Clases de Objetos"*.

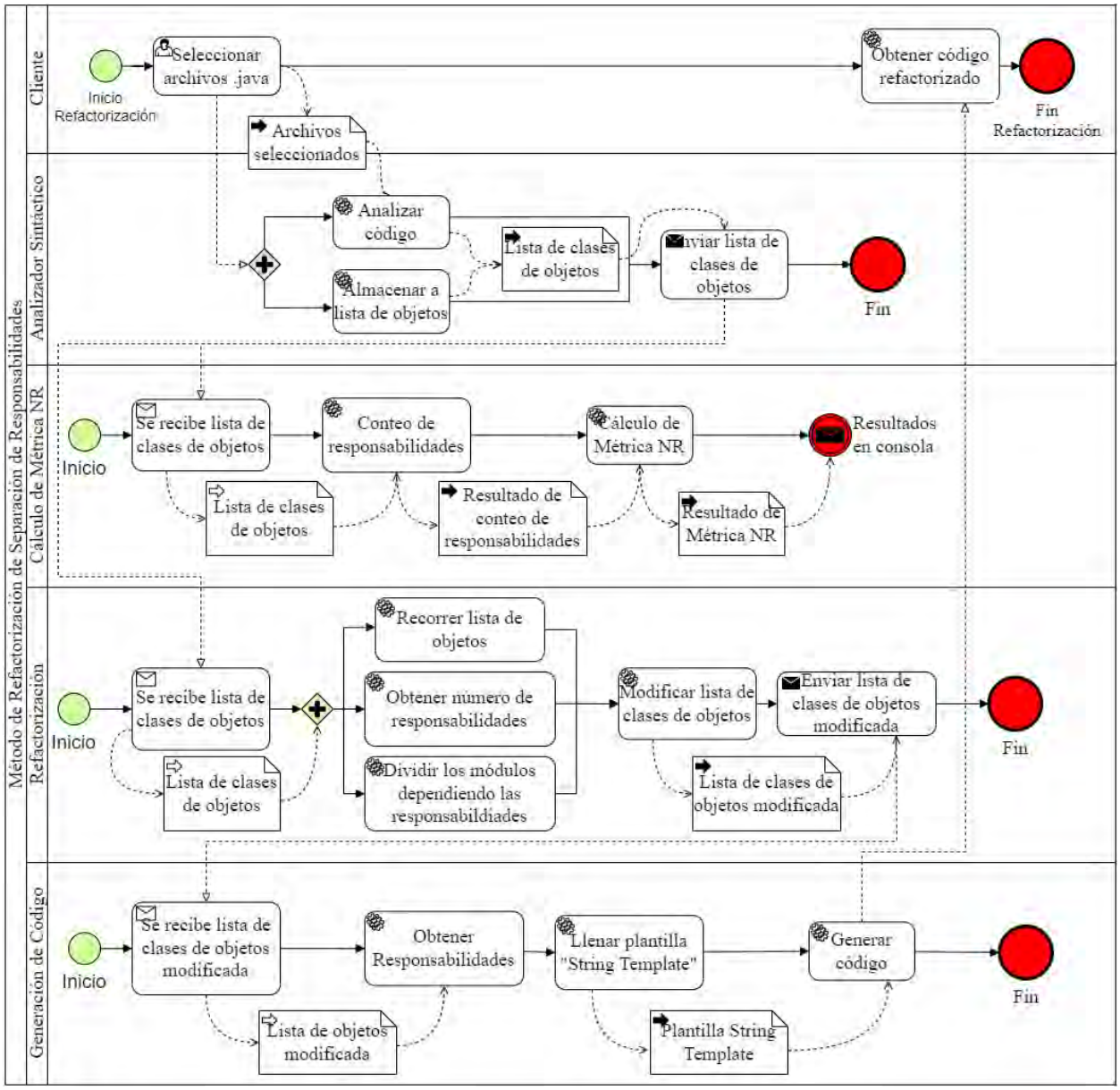

Figura 8 Diagrama BPMN del Método de Refactorización de Separación de Responsabilidades

CAPÍTULO 5.- DESARROLLO DEL SISTEMA

En este capítulo se describe el análisis realizado para desarrollar el método de separación de responsabilidades. Se utilizó la técnica de modelado UML, tales como: casos de uso, de clases, así como también la descripción de la herramienta utilizada para llevar a cabo el análisis de código fuente.

5.1 Análisis de Casos de Uso

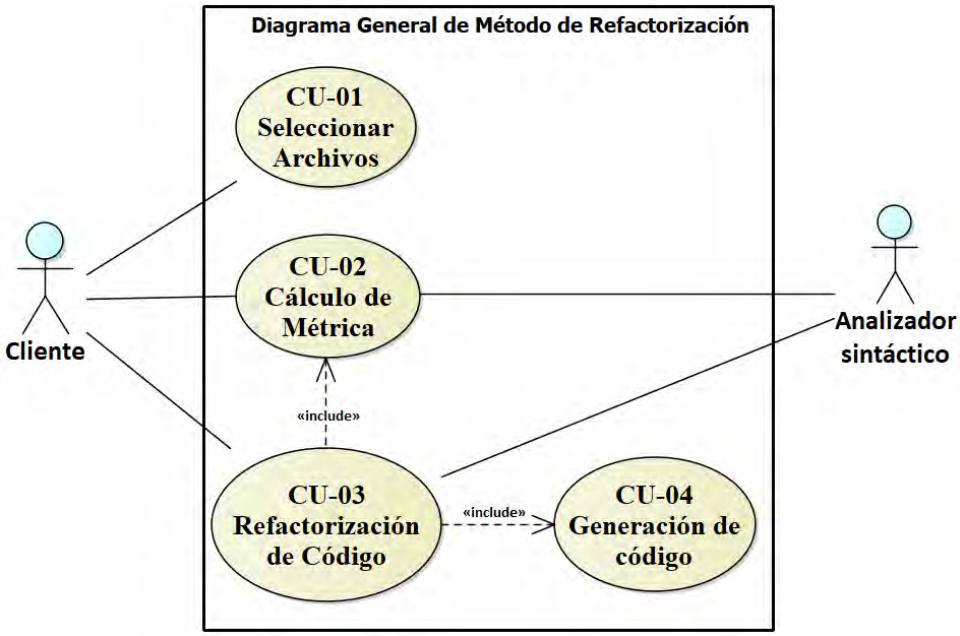

Figura 9 Diagrama de caso de uso del método de refactorización de separación de responsabilidades.

A continuación, se describe el análisis de cada uno de los casos de uso. La Figura 9 muestra el Diagrama General del Método de Refactorización, mostrando los principales casos de uso, los cuales son: *Seleccionar archivos, Cálculo de Métrica, Refactorización de Código y Generación de código.*

5.1.1 CU-01 Seleccionar archivos

El primer paso a llevar a cabo por parte del cliente/usuario será seleccionar los archivos del Marco Orientado a Objetos que se desea analizar y refactorizar. En este punto se aplica el caso de uso *"Seleccionar Archivos"*, el cual se muestra en la Figura 10, cabe mencionar que éste debe ser invocado antes de que el usuario quiera hacer cualquier otra operación.

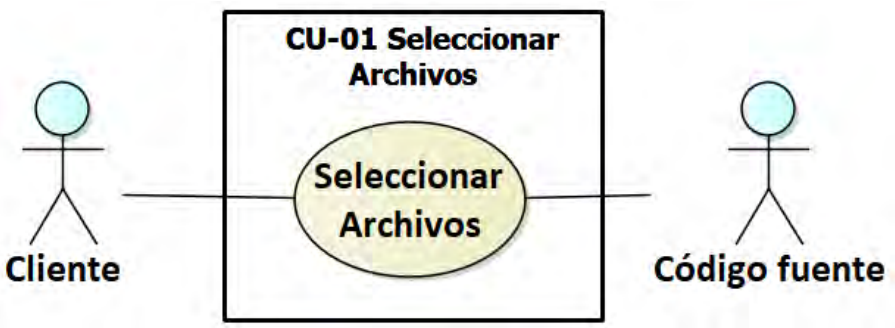

Figura 10 CU-01 Seleccionar archivos

5.1.2 CU-02 Cálculo de métrica

Después de seleccionar los archivos, se procede a realizar el análisis sintáctico, este tendrá como resultado la estructura de datos con la información recabada para el cálculo de la métrica de número de responsabilidades. La Figura 11 muestra el diagrama de caso de uso "Cálculo de métrica".

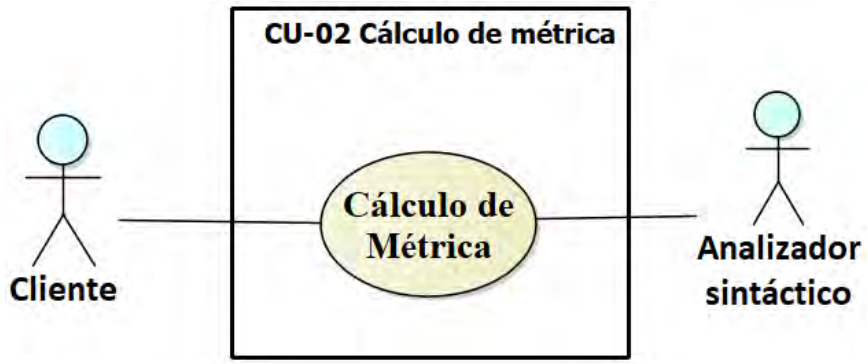

Figura 11 CU-02 Cálculo de métrica

5.1.3 CU-03 Refactorización de código

La Figura 12 muestra el diagrama del caso de uso *"Refactorización de Código"*, el cual es invocado cuando se desea realizar la refactorización. Este caso consiste en restructurar el código, separando las arquitecturas modulares que contengan más de una responsabilidad.

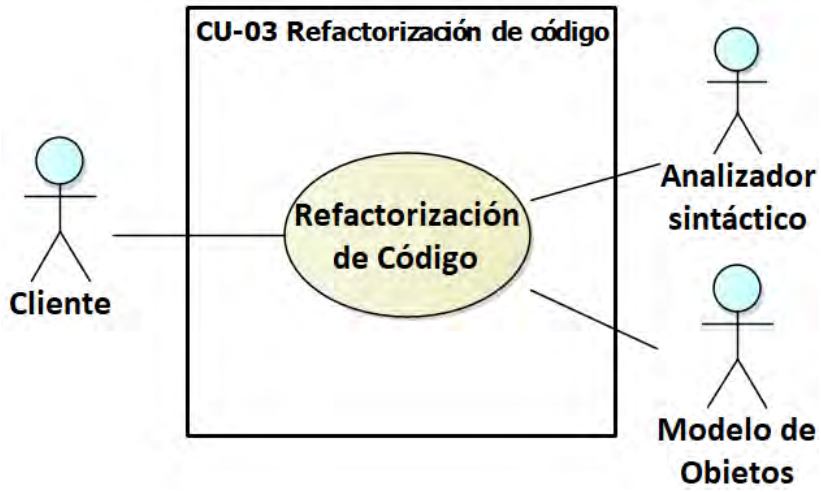

Figura 12 CU-03 Refactorización de código

5.2 Explicación de actores

Los actores son los siguientes:

Cliente: Representa una entidad física (persona) o una entidad lógica (sistema externo) que usará este método de refactorización. Existen 3 funciones que puede ejecutar el usuario, las cuales son:

- Seleccionar los archivos que se van a analizar.
- Solicitar el cálculo de la métrica de Número de Responsabilidades, ejecutando el caso de uso *Cálculo de Métrica.*
- Solicitar la ejecución del método de refactorización, ejecutando el caso de uso de *Refactorización de Código.*

Código Fuente: Será(n) el(los) archivo(s) Java original(es) que el Cliente seleccionará, en el caso de uso *"Seleccionar Archivos"*. Serán los archivos donde se encuentra el código que se desea analizar ya sea para aplicar el método de refactorización o el cálculo de la métrica de número de responsabilidades.

Modelo de Objetos: Este modelo representa toda la información proveniente del código fuente original que se necesita tanto para el cálculo de la métrica de "número de responsabilidades" (NR), como para el proceso del método de refactorización que remueve el código desagradable por arquitecturas modulares con más de una responsabilidad, en un mapa de clases de objetos.

Analizador sintáctico: Representa el subsistema encargado de realizar el análisis sintáctico. Recibe como entrada un proyecto de software escrito en lenguaje Java y regresa como salida un modelo de objetos con la información sobre los archivos que contienen el código fuente, ya sea para el cálculo de la métrica o para la refactorización.

5.3 Descripción de escenarios

A continuación, se describe el análisis de cada uno de los casos de uso del Diagrama General de la Figura 9. En esta descripción se muestra el nombre, actores, descripción, precondiciones, postcondiciones, escenario principal de éxito, escenarios alternos y los posibles escenarios de fracaso.

Tabla 2 Descripción de CU-01 Seleccionar Archivos

Nombre:	CU-01 Seleccionar Archivos.
Actores:	Usuario, Código Fuente.
Descripción:	El usuario deberá de seleccionar los archivos del sistema legado de software, del cual se requiere la refactorización y/o el cálculo de métricas, para capturar los metadatos del código original y realizar una copia de seguridad.
Precondiciones:	• Los archivos a analizar deberán estar en un mismo directorio. • El código original deberá estar libre de errores de sintaxis. • El código original deberá compilar correctamente.

Postcondiciones:	Como resultado se obtiene una lista de clases de objeto de los archivos con extensión Java seleccionados, los cuales se someterán al análisis sintáctico.
Escenario principal de éxito:	1. Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona el directorio o el conjunto de archivos Java (.java) de un sistema de software legado que requiere la refactorización y/o el cálculo de métricas. 3. El sistema verifica que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema extrae la información necesaria de los archivos, tanto para realizar el cálculo de las métricas como para la aplicación del método de refactorización. Esto es, la información sobre la interacción entre métodos de clases u objeto, tales como "el nombre del paquete" o "espacio de nombres", "nombres de clases", "nombre de atributos", "nombre de métodos", "parámetros de métodos", etc. 5. El sistema almacena la información extraída en una estructura de datos no persistente. Se usará un modelo de objetos. 6. El sistema conserva la información para los procesos de refactorización y cálculo de métricas. 7. El sistema realiza una copia de seguridad del código fuente original en directorios replicados.
Escenario alternativo (1)	Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona de diferentes directorios los archivos Java (.java) a extraer de un sistema de software legado que se requiere la refactorización o el cálculo de métricas. 3. Se ejecutan los puntos 3-7 del escenario principal de éxito.
Escenario de fracaso (1)	1. El usuario no selecciona ningún proyecto durante 5 minutos, entonces el proceso será abortado y tendrá que reiniciarse las veces que sean necesarias.
Escenario de fracaso (2)	1. Alguno(s) de lo(s) archivo(s) con el código fuente están disponibles sólo para su lectura. En este caso el sistema debe enviar un mensaje para que el usuario modifique los derechos de acceso y reiniciar el caso de uso.
Escenario de fracaso (3)	1. Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona un directorio. 3. El sistema verifica que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema no encuentra ningún archivo Java (.java) en el directorio seleccionado. 5. El sistema alertará al usuario que no se encontraron archivos Java (.java).

	6. En este caso el sistema debe enviar un mensaje al usuario indicándolo que no encontró archivos Java (.java). 7. Se reinicia el caso de uso.
Escenario de fracaso (4)	1. Se muestra el cuadro de dialogo de selección de archivos para seleccionar los archivos Java. 2. El usuario selecciona un directorio. 3. El sistema verifica que la totalidad de archivos originales estén considerados en el proceso. 4. El sistema no encuentra ningún archivo Java (.java) en el directorio seleccionado. 5. El sistema realizará la refactorización con los archivos encontrados, pero la funcionalidad original no será conservada. 6. Localizar la funcionalidad de los archivos faltantes. 7. Incorporar los archivos faltantes. 8. Reiniciar la refactorización.

Tabla 3 Descripción de CU-02 Cálculo de métrica

Nombre:	CU-02 Cálculo de métrica
Actores:	Sistema, Analizador Sintáctico, Código Fuente.
Descripción:	El sistema realiza el cálculo de la métrica que el usuario eligió para el análisis del proyecto. De acuerdo al valor obtenido se decide si se realiza o no la refactorización para reducir las responsabilidades existentes en arquitecturas modulares.
Precondiciones:	• El cálculo de la métrica se realiza a nivel de arquitecturas modulares. • Cada módulo de programa representado en una arquitectura obedece a un requerimiento o caso de uso específico del sistema de software original. • El código legado existente a refactorizar debe estar libre de defectos y/o errores, es decir, debe compilar y funcionar correctamente. • Contar con la estructura de datos con toda la información necesaria derivada de (CU-01).
Postcondiciones:	Como resultado se obtiene el valor de la métrica NR (Número de responsabilidades).
Escenario principal de éxito:	1. El usuario selecciona la métrica NR del marco de métricas. 2. El sistema realiza el cálculo de la métrica. 3. El sistema almacena la información de los resultados obtenidos. 4. El sistema muestra los resultados de la métrica al usuario. 5. El usuario con base al valor obtenido del cálculo de la métrica decide si aplicar la refactorización al código original (CU-3).

Escenario de fracaso (1)	1. El usuario no selecciona ningún proyecto durante 5 minutos, entonces el proceso será abortado y tendrá que reiniciarse las veces que sean necesarias.
Escenario de fracaso (2)	1. El sistema concluye su funcionamiento sin calcular la métrica.

Tabla 4 Descripción de CU-03 Refactorización de código

Nombre:	CU-03 Refactorización de código
Actores:	Usuario, Sistema, Código Fuente, Analizador Sintáctico, Modelo de Objetos
Descripción:	El sistema aplica el método de refactorización, con el fin de separar las responsabilidades.
Precondiciones:	1. El código legado existente a refactorizar debe estar libre de defectos y/o errores, es decir, debe compilar y funcionar correctamente. 2. Contar con el modelo de objetos con toda la información necesaria. Dicha información es resultado del proceso del subsistema Analizador Sintáctico. 3. El cálculo de métrica debe de ejecutarse antes del proceso de refactorización.
Postcondiciones:	• El código refactorizado se encuentra libre de código desagradable. • La arquitectura de software del código original es mejorada al remover el código desagradable por más de una responsabilidad en la arquitectura. • Se ve disminuida la deuda técnica. • La Funcionalidad del código refactorizado no cambia con respecto al código original. • Se obtendrán los archivos con el código refactorizado en directorios replicados de salida de manera correspondiente.
Escenario principal de éxito:	1. El Usuario selecciona la ruta del proyecto de software de entrada a refactorizar. 2. El sistema ejecuta el subsistema “Analizador Sintáctico” para realizar el análisis de código fuente. 3. El sistema recibe el modelo de objetos obtenida por el subsistema “Analizador Sintáctico”, con tenida en una estructura contenedora de datos. 4. El sistema analiza la información contenida en la estructura de datos para identificar las responsabilidades existentes en la arquitectura modular. 5. El sistema realiza la división de responsabilidades. 6. El sistema crea módulos de acuerdo a las responsabilidades encontradas. 7. El sistema actualiza las dependencias entre los módulos creados, así como las llamadas a métodos, tipos de datos, importaciones.

	8. El sistema genera código refactorizado (CU-04).
Escenario alternativo (1)	1. Se ejecutan los puntos 1-4 del escenario principal de éxito. 2. El sistema realiza la división de responsabilidades parcialmente en la arquitectura modular analizada, debido a que necesite la ejecución de algún otro método de refactorización. 3. El sistema crea módulos de acuerdo a las responsabilidades encontradas. 4. El sistema actualiza las dependencias entre los módulos creados, así como las llamadas a métodos, tipos de datos, importaciones. 9. El sistema genera código refactorizado (CU-04).
Escenario de fracaso (1)	1. El sistema concluye su funcionamiento sin refactorizar *(ver descripción de escenarios de código de entrada).*

Tabla 5 Descripción de CU-04 Generación de código

Nombre:	CU-04 Generación de código
Actores:	Sistema, Modelo de Objetos
Descripción:	El sistema alimenta a la plantilla *"String Template"* con la información de cada uno de los objetos presente en la estructura de datos, generada en el CU-03 y genera el código refactorizado en cada arquitectura modular.
Precondiciones:	• Contar con el modelo de objetos necesario para llenar la plantilla ST.
Postcondiciones:	• Código refactorizado almacenado con el mismo nombre del sistema analizado más la extensión *"_refactorizado"*. • Código refactorizado equivalente en funcionalidad al código original de entrada.
Escenario principal de éxito:	1. Se recorre la estructura de datos plasmando la información en la plantilla *String template.* 2. Se genera el código refactorizado desde la plantilla *String template* equivalente en funcionalidad al código original de entrada. 3. El sistema almacena el código refactorizado con el mismo nombre del sistema analizado, agregando al final del mismo la extensión *"_refactorizado"*.
Escenario de fracaso (1)	1. Se recorre la estructura de datos plasmando la información en la plantilla *String template.* 2. El sistema determina un escenario no previsto y detiene su funcionamiento, por lo que no termina el proceso de generación de código. 3. Termina caso de uso.

5.4 Análisis del código fuente

Para llevar a cabo el análisis léxico y sintáctico de las arquitecturas modulares, se utilizó el generador de analizadores ANTLR, junto con la gramática para el lenguaje Java en su versión 8.0.

ANTLR (Another Tool for Language Recognition) es una herramienta para reconocimiento de lenguajes que está escrito en lenguaje Java, proporciona la capacidad para construir reconocedores, intérpretes, compiladores y traductores de lenguajes (John Parr, 2013).

5.5 Análisis e implementación de escenarios

Dada la diversidad de estilos de programación que existen, se realizó un análisis lo más completo posible de estos estilos, con la finalidad de comprender cómo puede estar integrado el código de entrada a refactorizar. El análisis de escenarios se efectuó bajo lineamientos del Lenguaje Unificado de Modelado (UML). Los escenarios describen las posibles formas y estilos de programación en los que puede llegar a estar el sistema de software que se sujeta a estudio para su refactorización.

Cada responsabilidad está dada por la ejecución en una secuencia de métodos, dicha secuencia inicia con un *método público*. La secuencia está integrada por las invocaciones de un método a otro en secuencia lógica, hasta alcanzar el objetivo o meta de valor de cada entidad de software modular. Una secuencia termina cuando un método ya no invoca a otro método y regresa el control hacía el método llamador y así sucesivamente. Esto nos permitirá identificar las responsabilidades existentes dentro de las arquitecturas modulares.

Cada uno de los escenarios estarán conformados de la siguiente forma:

1. Punto de entrada *(Clientes)* hacía las arquitecturas modulares.
2. Se cuenta con el *Paquete/Módulo,* contenedor de la(s) *Clase(s)*.
3. Ejemplo(s) de una(s) *Clase(s)* con sus respectivos atributos y métodos.

A continuación, se presentan diagramas de clases que ejemplifican las representaciones de los diferentes escenarios encontrados. A dichas representaciones se les aplica el cálculo de la métrica definida en la ecuación 1, así como el método de refactorización de separación de responsabilidades en arquitecturas modulares.

5.5.1 Escenario 1 – Única responsabilidad

El mejor de los escenarios posibles, es cuando un módulo sólo cuenta con una única responsabilidad, por lo tanto, el sistema al detectar este tipo de escenarios, deja el módulo sin cambios, tal como se encuentra.

En este caso, podemos llegar a tener la secuencia de métodos como se muestra en la Figura 13:

$$\boldsymbol{Cliente} \rightarrow ejecutarA() \rightarrow metodo1() \rightarrow metodo2() \rightarrow metodo3()$$

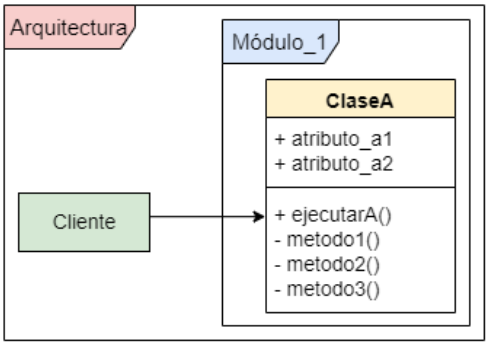

Figura 13 Escenario 1. Única Responsabilidad

Prueba escenario 1

Como prueba para el escenario 1 se muestra la arquitectura en la Figura 14.

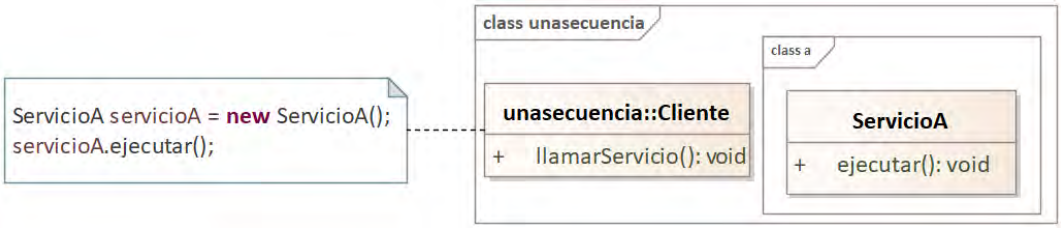

Figura 14 Proyecto Escenario 1, arquitectura original antes de refactorizar

En la Figura 15 se puede observar el fragmento de código de la clase Cliente. Se puede observar que el método *"llamarServicio"* hace una llamada al método "*ejecutar"* del objeto *"servicioA"*.

```
1  package suite.escenarios.responsabilidad.sencillo.unasecuencia.paquete.diferente;
2
3  import suite.escenarios.responsabilidad.sencillo.unasecuencia.paquete.diferente.a.ServicioA;
4
5  public class Cliente {
6
7      public void llamarServicio() {
8          ServicioA servicioA = new ServicioA();
9          servicioA.ejecutar();
10     }
11 }
```

Figura 15 Código de Clase Cliente

Al aplicar la métrica de Número Responsabilidad, nos da como resultado 1, como se muestra en la Figura 16; esto indica que solo existe una secuencia de métodos y por ende una única responsabilidad.

```
====================================
Responsabilidad de paquete : a
Número de entradas: 1
|Clase                  Entradas|
|ServicioA.ejecutar           1 |
Responsabilidad: 1.0
```

Figura 16 Escenario 1. Resultado de la métrica

Lo anterior representa el mejor de los escenarios, por lo que el método de refactorización deja la arquitectura original sin cambios.

5.5.2 Escenario 2 – Más de una responsabilidad

En este escenario (Figura 17) se puede encontrar un módulo que puede llegar a tener más de una responsabilidad, se tiene un cliente que accede a dos puntos de entrada hacía clases diferentes, generando más de una secuencia modular, en este escenario son dos secuencias.

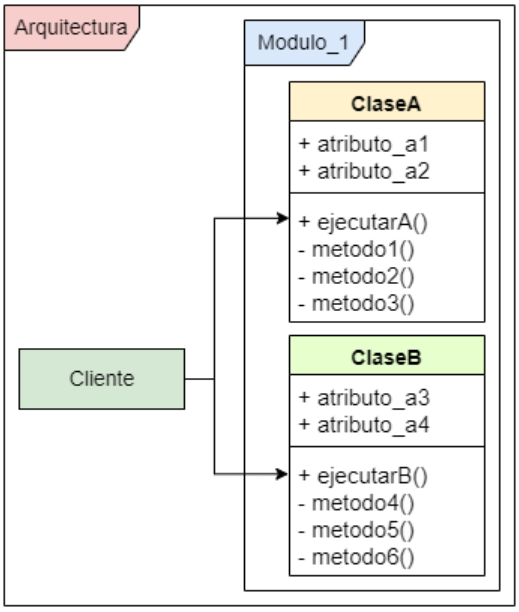

Figura 17 Escenario 2. Más de una Responsabilidad

Como ejemplo, en la Tabla 6 se puede observar cómo puede llegar a tener dichas secuencias de métodos:

Tabla 6 Escenario 2. Ejemplo de Secuencias de Métodos

Módulo	Clases	Secuencias	Responsabilidades (Secuencias de Métodos)
Modulo_1	ClaseA	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()* → *metodo3()*	1
	ClaseB	***Cliente*** → *ejecutarB()* → *metodo4()* → *metodo5()* → *metodo6()*	1
	Número total de Responsabilidades en Módulo_1		2

Dicha arquitectura al presentar más de una responsabilidad, tiene que ser evaluada y refactorizada. Después de realizar la refactorización, como salida se tiene la arquitectura que se muestra en la Figura 18, donde cada módulo estará formado por una sola secuencia de métodos, es decir una única responsabilidad modular.

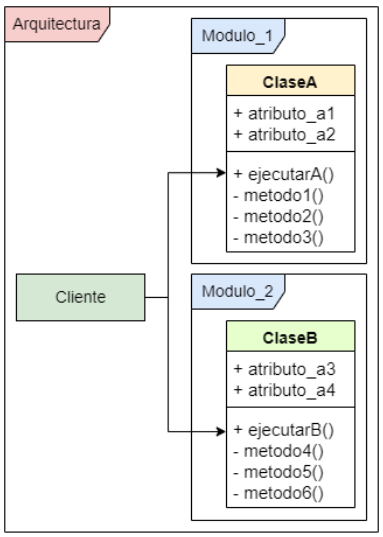

Figura 18 Escenario 2. Refactorizado

Secuencia Modulo_1: Cliente → ejecutarA → metodo1() → metodo2() → metodo3().

Secuencia Modulo_2: Cliente → ejecutarB → metodo4() → metodo5() → metodo6().

Prueba escenario 2

Como prueba para el escenario 2, en la Figura 19 se muestra la arquitectura y la carpeta de archivos que corresponden a dicho escenario.

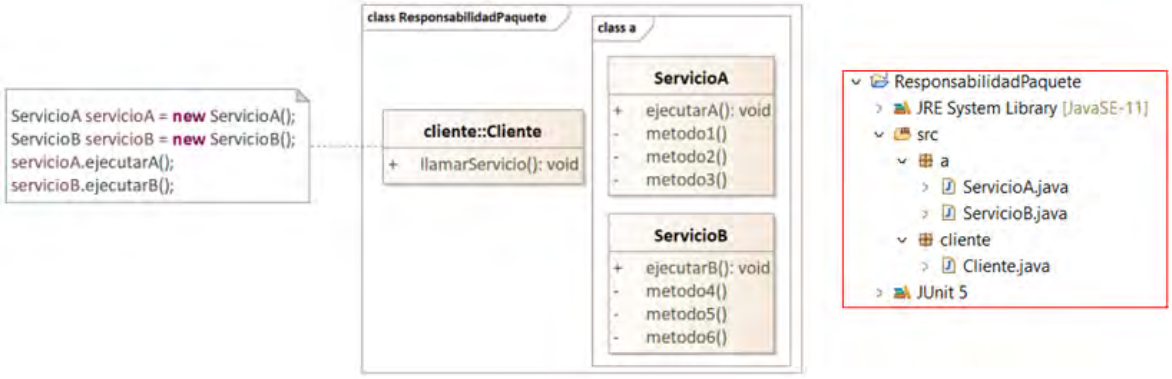

Figura 19 Proyecto Escenario 2, arquitectura original antes de refactorizar

Dentro de la clase Cliente (Figura 20) se puede observar que se cuenta con el siguiente código: el método *"llamarServicio"* hace dos llamadas a diferentes métodos "*ejecutar"*, uno del objeto *"servicioA"* y otro del objeto *"servicioB"*.

```
1  package cliente;
2  import a.ServicioA;
3  import a.ServicioB;
4
5  public class Cliente {
6
7      public void llamarServicio() {
8          ServicioA servicioA = new ServicioA();
9          ServicioB servicioB = new ServicioB();
10         servicioA.ejecutar();
11         servicioB.ejecutar();
12     }
13 }
```

Figura 20 Fragmento de código de Clase Cliente

Al aplicar el cálculo de la métrica de Número de Responsabilidad, el resultado es 0.5; esto nos indica que dicha arquitectura cuenta con más de una secuencia de métodos y por ende con más de una única responsabilidad, como se muestra en la Figura 21.

```
====================================
Responsabilidad de paquete : a
Número de entradas: 2
|Clase                  Entradas|
|ServicioA.ejecutar            1 |
|ServicioB.ejecutar            1 |
Responsabilidad: 0.5
```

Figura 21 Escenario 2. Resultado de la métrica

Al ser ésta una arquitectura que violenta el principio de única responsabilidad, se aplica el método de refactorización para mejorar las responsabilidades, la Figura 22 muestra la arquitectura obtenida, así como su carpeta de archivos.

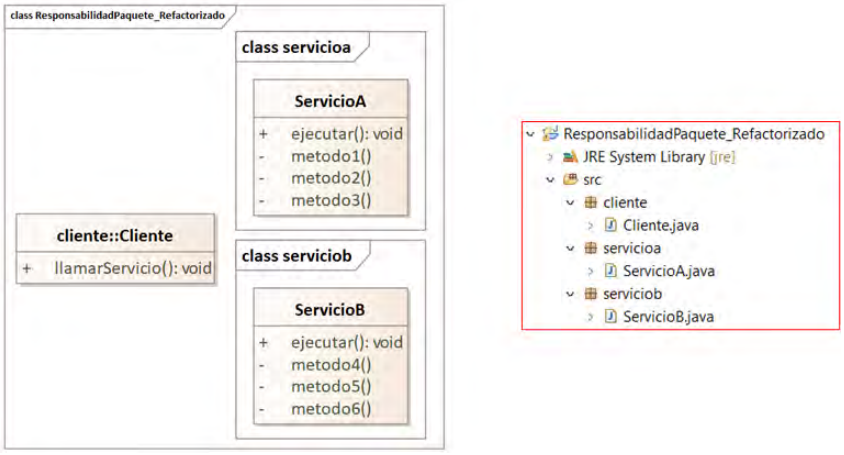

Figura 22 Proyecto Escenario 2, arquitectura después de refactorizar

Para comprobar que la refactorización mejora las responsabilidades en arquitecturas modulares, ejecutamos nuevamente la métrica, pero ahora sobre la arquitectura ya refactorizada, los valores obtenidos se muestran en la Figura 23. Estos valores indican que cada módulo perteneciente a la arquitectura cuenta con una única responsabilidad.

```
========================================
Responsabilidad de paquete : servicioa
Número de entradas: 1
|Clase                  Entradas|
|ServicioA.ejecutar            1 |
Responsabilidad: 1.0
========================================
Responsabilidad de paquete : serviciob
Número de entradas: 1
|Clase                  Entradas|
|ServicioB.ejecutar            1 |
Responsabilidad: 1.0
```

Figura 23 Escenario 2. Resultado de la métrica después de la refactorización

5.5.3 Escenario 3 – Clientes

En caso de existir arquitecturas con más de un cliente que inicializan la secuencia de métodos en un mismo módulo, se interpretará como si fueran dos entradas independientes. Esto ocasionará que cada cliente tenga su propia secuencia de métodos, aun cuando su ejecución empiece por el mismo método, por lo tanto, será una única responsabilidad. Un ejemplo de este tipo de escenarios se muestra en la Figura 24.

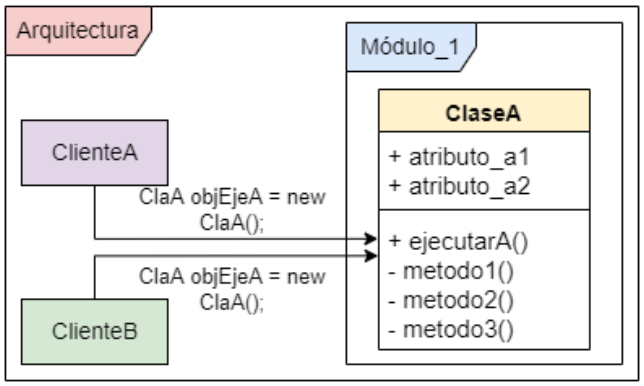

Figura 24 Escenario 3. Más de un cliente

Como ejemplo, en la Tabla 7 se puede observar cómo podemos llegar a tener dichas secuencias de métodos:

Tabla 7 Escenario 3. Ejemplo de Secuencias de Métodos

Módulo	Clases	Secuencias	Responsabilidades (Secuencias de Métodos)
Modulo_1	ClaseA	***ClienteA*** → *ejecutarA()* → *metodo1()* → *metodo2()* → *metodo3()*	1
Modulo_1	ClaseA	***ClienteB*** → *ejecutarA()* → *metodo1()* → *metodo2()* → *metodo3()*	1
	Número total de Responsabilidades Módulo_1 con ClienteA		1
	Número total de Responsabilidades Módulo_1 con ClienteB		1

A pesar de que se puede interpretar que dicho módulo cuenta con dos responsabilidades, debemos de tomar en cuenta que cada uno de los clientes es independiente, es decir, cada uno accede al mismo punto de entrada a la secuencia del mismo módulo para llevar a cabo la secuencia de métodos. Dichas arquitecturas no requieren que sean refactorizadas, ya que, al aplicar la métrica de única responsabilidad, esta dará 1 como resultado.

Prueba escenario 3

Como prueba para el escenario 3 se cuenta con la arquitectura y carpeta de archivos mostrada en la Figura 25.

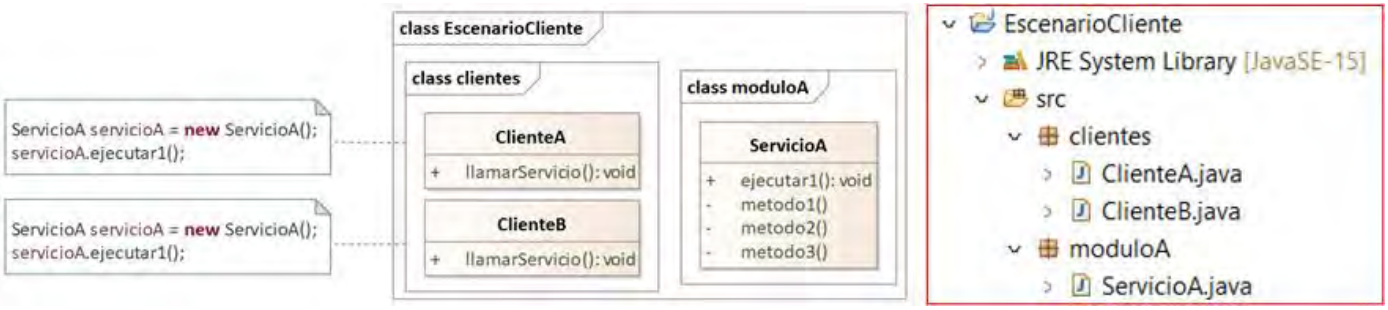

Figura 25 Proyecto Escenario 3, arquitectura original antes de refactorizar

Dentro de la Clases ClienteA y ClienteB (Figura 26 y 27 respectivamente) podemos observar que se encuentra el siguiente código:

```
1  package clientes;
2
3  import moduloA.ServicioA;
4
5  public class ClienteA {
6      public void llamarServicio() {
7          ServicioA servicioA = new ServicioA();
8          servicioA.ejecutar1();
9      }
10 }
```

Figura 26 Código de Clase ClienteA

```
1  package clientes;
2
3  import moduloA.ServicioA;
4
5  public class ClienteB {
6      public void llamarServicio() {
7          ServicioA servicioB = new ServicioA();
8          servicioB.ejecutar1();
9      }
10 }
```

Figura 27 Código de Clase ClienteB

Se puede observar que tanto el ClienteA como el ClienteB llaman al mismo método de entrada ejecutar1() de la Clase ServicioA. Como se mencionó anteriormente, cada uno de estos clientes es tomado de forma independiente, por lo cual, al ejecutar la métrica, da como resultado Responsabilidad = 1, como se muestra en la Figura 28.

```
=======================================
Responsabilidad de paquete : moduloA
Número de entradas: 1
|Clase                  Entradas|
|ServicioA.ejecutar1           2 |
Responsabilidad: 1.0
```

Figura 28 Escenario 3. Resultado de la métrica

5.5.4 Escenario 4 – Dependencia entre módulos

Otro escenario considerado es cuando las clases de más de un módulo están relacionadas. Esto permite que una clase conozca los atributos, operaciones y relaciones de otras clases, como se muestra en la Figura 29.

Una dependencia entre clases significa que una clase utiliza, o tiene conocimiento de otra clase, o dicho de otro modo "lo que una clase necesita conocer de otra clase para utilizar objetos de esa clase" (Hamilton & Miles, 2006).

Un criterio clave en la división de módulo es la independencia, esto es, el menor acoplamiento entre módulos; otro criterio es que cada módulo deba ejecutar una sola secuencia. Los criterios fundamentales son acoplamiento y cohesión entre módulos.

El acoplamiento se refiere al grado de interdependencia entre módulos, por lo cual es preciso minimizar dicho acoplamiento entre éstos. La cohesión es la fortaleza interna de un módulo, esto es, lo fuertemente relacionadas que están entre sí las entidades locales de dicho módulo.

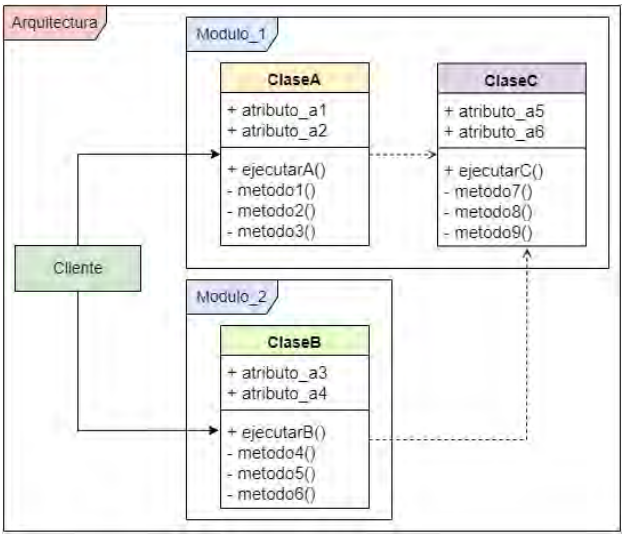

Figura 29 Escenario 4. Dependencia entre Clases

A continuación, en la Tabla 8 se muestran ejemplos de cómo pueden estar dadas las secuencias en cada uno de los módulos.

Tabla 8 Escenario 4. Ejemplo de Secuencias de Métodos

Módulo	Clases	Secuencias	Responsabilidades (Secuencias de Métodos)
Modulo_1	ClaseA ClaseC	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()* → *metodo3()* → *ejecutarC()* → *metodo7()* → *metodo8()* → *metodo9()*	1
	ClaseB	***Cliente*** → *ejecutarC()* → *metodo7()* → *metodo8()* → *metodo9()*	1
Modulo_2	ClaseB ClaseC	***Cliente*** → *ejecutarB()* → *metodo4()* → *metodo5()* → *metodo6()* → *ejecutarC()* → *metodo7()* → *metodo8()* → *metodo9()*	1
	Número total de Responsabilidades Modulo_1		2
	Número total de Responsabilidades Modulo_2		1

Dicha arquitectura al presentar más de una responsabilidad, tiene que ser evaluada y refactorizada. Como salida después de la refactorización, se tiene la arquitectura que se muestra en la Figura 30, donde cada módulo estará formado por una sola secuencia de métodos, es decir una única responsabilidad.

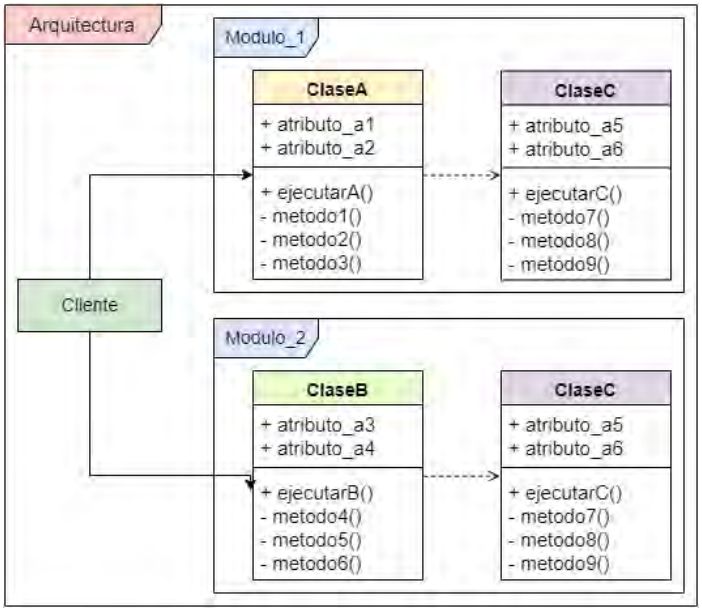

Figura 30 Escenario 4. Refactorizado

Prueba escenario 4

Como prueba para el escenario 4 se cuenta con la arquitectura y carpeta de archivos mostrada en la Figura 31.

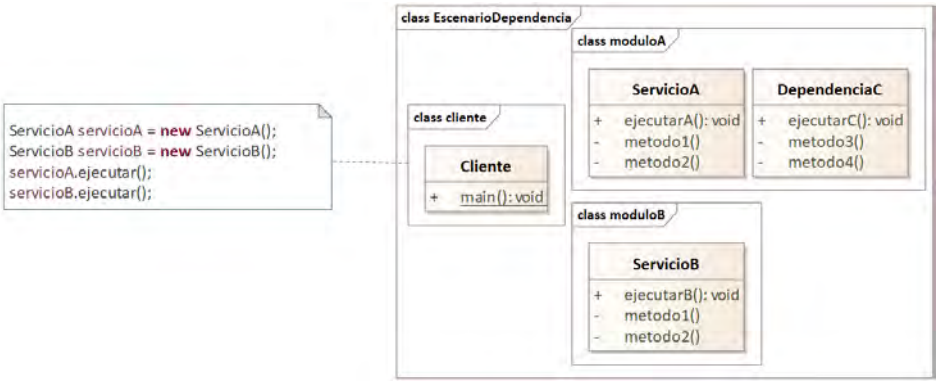

Figura 31 Proyecto Escenario 4, arquitectura original antes de refactorizar

Dentro de la Clase Cliente (Figura 32) podemos observar que se encuentra el siguiente código:

```
1  package cliente;
2
3  import moduloA.ServicioA;
4  import moduloB.ServicioB;
5
6  public class Cliente {
7
8      public static void main(String[] args) {
9          ServicioA servicioA = new ServicioA();
10         ServicioB servicioB = new ServicioB();
11         servicioA.ejecutar();
12         servicioB.ejecutar();
13     }
14 }
```

Figura 32 Código de Clase Cliente

Se puede observar que el método *"Main"* hace dos llamadas a diferentes métodos "*ejecutar"*, uno del objeto *"servicioA" y* otro del objeto *"servicioB".* Al aplicar la métrica de Número de Responsabilidades, da como resultado 0.5 en el móduloA y 1 en el móduloB, tal y como se explicó previamente, estos valores los podemos observar en la Figura 33; esto indica que la arquitectura modular "móduloA" cuenta con más de una secuencia de métodos y por ende con más de una responsabilidad.

```
[        NR: Módulo a Refactorizar        ]
=============================================
Responsabilidad de paquete : moduloA
Número de entradas: 2
|Clase                  Entradas|
|DependenciaC.ejecutar          |
|ServicioA.ejecutar             |
Responsabilidad: 0.5
=============================================
Responsabilidad de paquete : moduloB
Número de entradas: 1
|Clase                  Entradas|
|ServicioB.ejecutar             |
Responsabilidad: 1.0
```

Figura 33 Escenario 4. Resultado de la métrica

Al ser una arquitectura que violenta el principio de única responsabilidad, se aplica el método de refactorización para mejorar las responsabilidades, obteniendo la arquitectura que se muestra en la Figura 34, así como su carpeta de archivos.

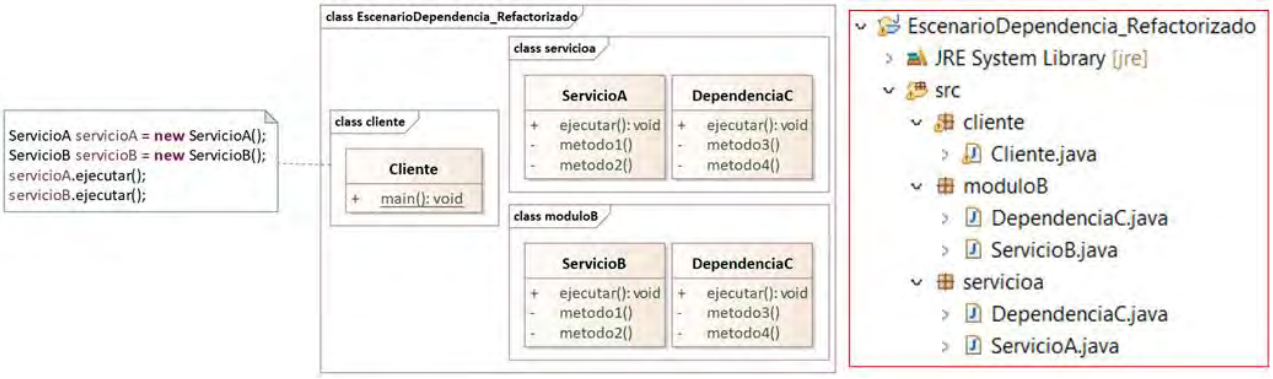

Figura 34 Proyecto Escenario 4, arquitectura después de refactorizar

Para comprobar que la refactorización mejora las responsabilidades en arquitecturas modulares, ejecutamos nuevamente la métrica, pero ahora sobre la arquitectura ya refactorizada, los valores obtenidos se muestran en la Figura 35. Estos valores indicarán que cada módulo perteneciente a la arquitectura cuenta con una única responsabilidad.

```
----- Valores después de la Refactorización -----
=================================================
Responsabilidad de paquete : servicioa
Número de entradas: 1
|Clase                  Entradas|
|ServicioA.ejecutar             |
Responsabilidad: 1.0
=================================================
Responsabilidad de paquete : moduloB
Número de entradas: 1
|Clase                  Entradas|
|ServicioB.ejecutar             |
Responsabilidad: 1.0
```

Figura 35 Escenario 4. Resultado de la métrica después de la refactorización

5.5.5 Escenario 5 – Coreografía

Mediante la coreografía se delega la responsabilidad de la toma de decisiones y la secuenciación. Es decir, se comunican principalmente mediante un intercambio de llamadas.

Durante este tipo de escenarios se tienen las interacciones de las clases participantes, es decir, se define la secuencia de interacciones entre dos o más clases participantes, así como también los módulos.

En la Figura 36 se puede apreciar un ejemplo de arquitectura modular coreografiada, dicha arquitectura cuenta con dos clientes diferentes, donde el ClienteA llama al método *"ejecutarA"*, el cual a su vez llama al método *"ejecutarB"* de la clase B, continuando la secuencia con la ejecución del método *"ejecutarC"* y por último la llamada al método *"ejecutarD"*, también se puede apreciar que existe otro Cliente llamado ClienteB, este cliente llama al método *"ejecutarB"* de la clase B, continuando la secuencia con la ejecución del método *"ejecutarC"* y para finalizar llama al método *"ejecutarD" de la ClaseD.*

Al existir dos puntos de entrada diferente, es necesario llevar a cabo la refactorización de la arquitectura.

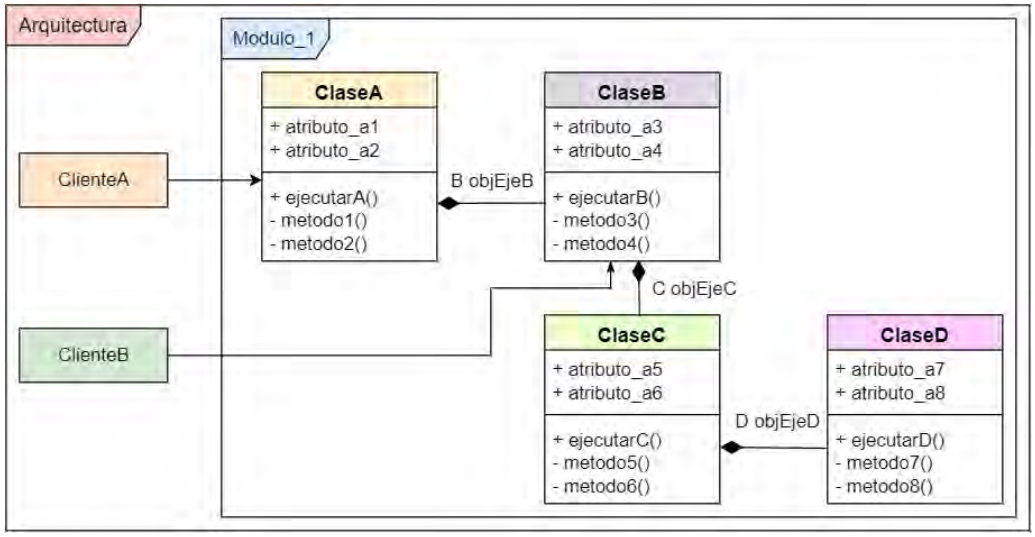

Figura 36 Escenario 5. Coreografía

Como ejemplo, en la Tabla 9 se puede observar cómo puede llegar a tener dichas secuencias de métodos:

Tabla 9 Escenario 5. Ejemplo de Secuencias de Métodos

Módulo	**Clases**	**Secuencias**	**Responsabilidades (Secuencias de Métodos)**
Modulo_1	ClaseA ClaseB ClaseC ClaseD	***ClienteA*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarB()*→ *metodo3()* → *metodo4()* → *ejecutarC()*→ *metodo5()* → *metodo6()* → *ejecutarD()*→ *metodo7()* → *metodo8()*	1
Modulo_1	ClaseB ClaseC ClaseD	***ClienteB*** → *ejecutarB()*→ *metodo3()* → *metodo4()* → *ejecutarC()*→ *metodo5()* → *metodo6()* → *ejecutarD()*→ *metodo7()* → *metodo8()*	1
	Número total de Responsabilidades Modulo_1.		2

Dado que la arquitectura presenta más de dos responsabilidades, tiene que ser evaluada y refactorizada. Al llevar a cabo la refactorización, como salida se tiene la arquitectura que se muestra en la Figura 37, donde cada módulo estará formado por una sola secuencia de métodos, es decir una única responsabilidad.

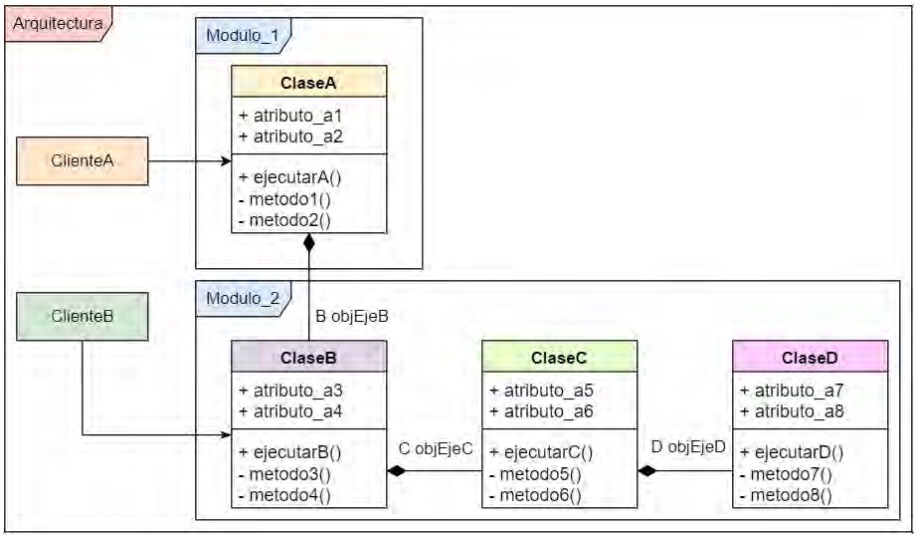

Figura 37 Escenario 5. Refactorizado

Prueba escenario 5

En la Figura 38 se puede observar la arquitectura de prueba para el escenario 5, así como la carpeta contenedora de archivos.

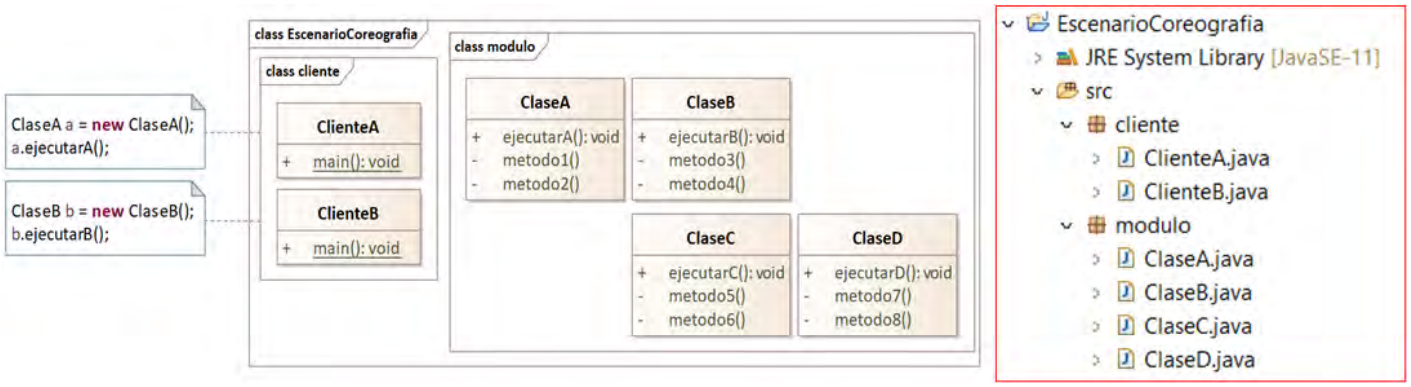

Figura 38 Proyecto Escenario 5, arquitectura original antes de refactorizar

En la Figura 39 se puede observar que la clase ClienteA sirve como punto de entrada para llevar a cabo las llamadas coreografiadas dentro de la arquitectura.

```
1  package cliente;
2
3  import modulo.ClaseA;
4
5  public class ClienteA {
6
7      public static void main(String[] args) {
8          ClaseA a = new ClaseA();
9          a.ejecutarA();
10     }
11 }
```

Figura 39 Código de Clase ClienteA

En la Figura 40 se muestre el código correspondiente al ClienteB, el cual también es un iniciador de secuencias de métodos.

```
1 package cliente;
2
3 import claseb.ClaseB;
4 public class ClienteB {
5     public static void main(String[] args) {
6         ClaseB b = new ClaseB();
7         b.ejecutarB();
8     }
9 }
```

Figura 40 Código de Clase ClienteB

Para verificar si la arquitectura incumple con el principio de única responsabilidad se ejecuta la métrica de Única Responsabilidad, obteniendo como resultado módulo = 0.5, como se observa en la Figura 41.

```
[        NR: Módulo a Refactorizar        ]
=============================================
Responsabilidad de paquete : modulo
Número de entradas: 2
|Clase                  Entradas|
|ClaseA.ejecutarA               |
|ClaseB.ejecutarB               |
Responsabilidad: 0.5

--------Valores antes de la Refactorización
|Paquete         Responsabilidad|
|cliente                0.0     |
|modulo                 0.5     |
```

Figura 41 Escenario 5. Resultado de la métrica

Al ser una arquitectura que violenta el principio de única responsabilidad, se aplica el método de refactorización para mejorar las responsabilidades, En la Figura 42 se muestra la arquitectura obtenida, así como la carpeta de archivos.

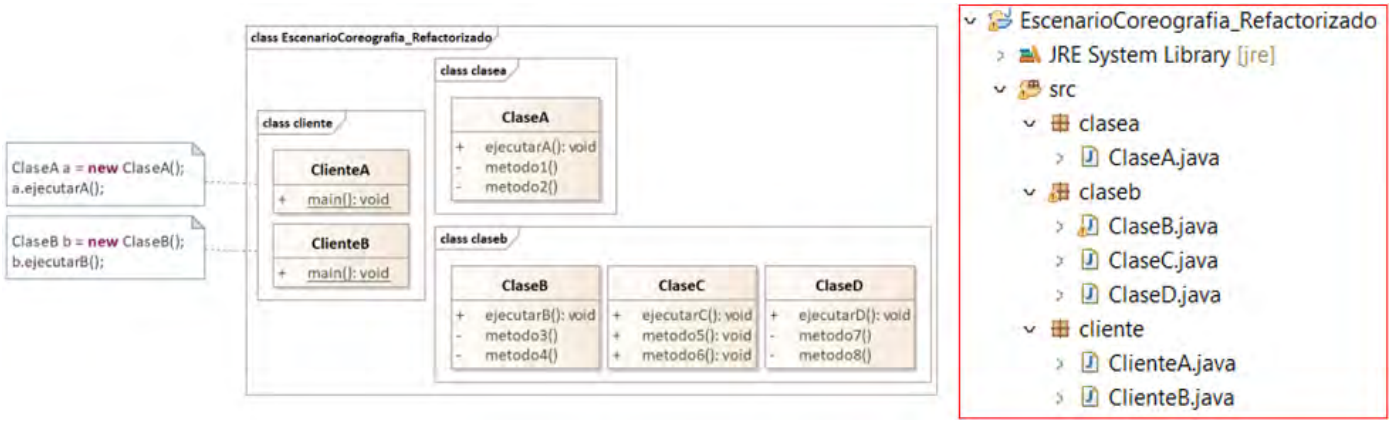

Figura 42 Proyecto Escenario 5, arquitectura después de refactorizar

Para comprobar que la refactorización mejora las responsabilidades en arquitecturas modulares, ejecutamos nuevamente la métrica, pero ahora sobre la arquitectura ya refactorizada, los valores obtenidos se muestran en la Figura 43. Estos valores nos indicaran que cada módulo perteneciente a la arquitectura cuenta con una única responsabilidad.

```
----- Valores después de la Refactorización -----
================================================
Responsabilidad de paquete : claseb
Número de entradas: 1
|Clase                  Entradas|
|ClaseB.ejecutarB               |
Responsabilidad: 1.0
================================================
Responsabilidad de paquete : clasea
Número de entradas: 1
|Clase                  Entradas|
|ClaseA.ejecutarA               |
Responsabilidad: 1.0
```

Figura 43 Escenario 5. Resultado de la métrica después de la refactorización

5.5.6 Escenario 6 – Orquestación

La orquestación es usada comúnmente para indicar el proceso o el flujo del funcionamiento de una arquitectura. En este caso un componente es el encargado de coordinar las llamadas a los métodos que se necesitan de forma secuencial.

Una secuencia puede alinear grupos de actividades relacionadas en una lista determinada por un orden de ejecución secuencial. Las secuencias son especialmente útiles cuando una parte de la lógica de la aplicación depende del resultado de otra.

En la Figura 44 se puede apreciar un ejemplo de arquitectura modular orquestada, donde el Cliente crea un objeto de la ClaseA y llama al método *"ejecutarA"*, después la ClaseA crea un objeto de la ClaseB y llama al método *"ejecutarB"* una vez que dicha secuencia termina, regresa el control a la ClaseA, la cual ahora crea un objeto de la ClaseC y llama al método *"ejecutarC,* cuando esta termina de realizar las acciones pasa nuevamente el control a la ClaseA, la cual crea objetos de las ClaseD y ClaseE respectivamente, las cuales realizan las acciones y regresan el control a la ClaseA nuevamente.

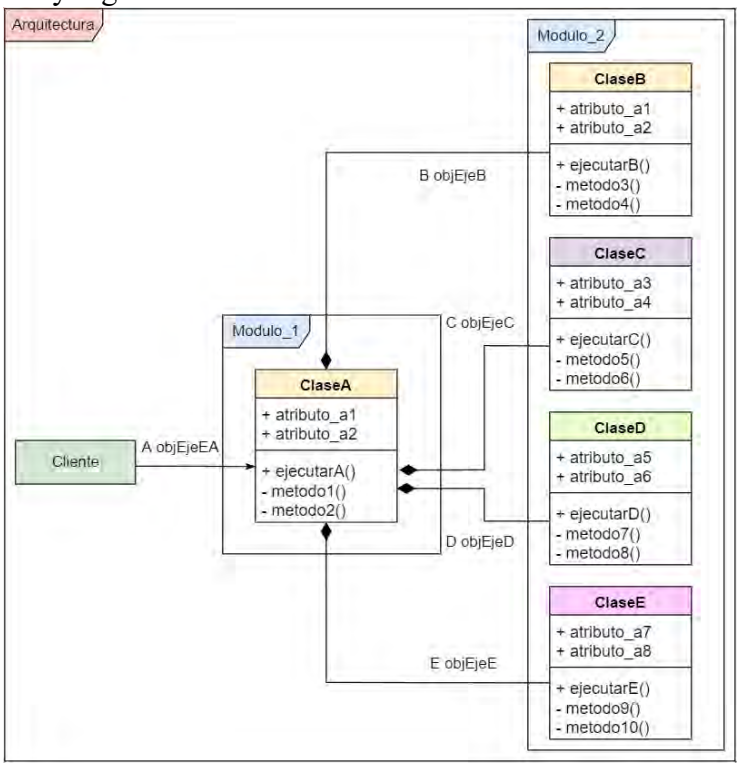

Figura 44 Escenario 6. Orquestación

En la Tabla 10 se observa cómo pueden llegar a estar formadas las secuencias de métodos para la arquitectura anterior:

Tabla 10 Escenario 6. Ejemplo de Secuencias de Métodos

Módulo	**Clases**	**Secuencias**	**Responsabilidades (Secuencias de Métodos)**
Modulo_1	ClaseA	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*	**1**
Modulo_1 Modulo_2	ClaseA ClaseB	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarB()*→ *metodo3()* → *metodo4()*	1
	ClaseA ClaseC	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarC()*→ *metodo5()* → *metodo6()*	1
	ClaseA ClaseD	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarD()*→ *metodo7()* → *metodo8()*	1
	ClaseA ClaseE	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarE()*→ *metodo9()* → *metodo10()*	1
	Número total de Responsabilidades Modulo_1		1
	Número total de Responsabilidades Modulo_2		4

La arquitectura esperada después de la refactorización es la que se muestra en la Figura 45.

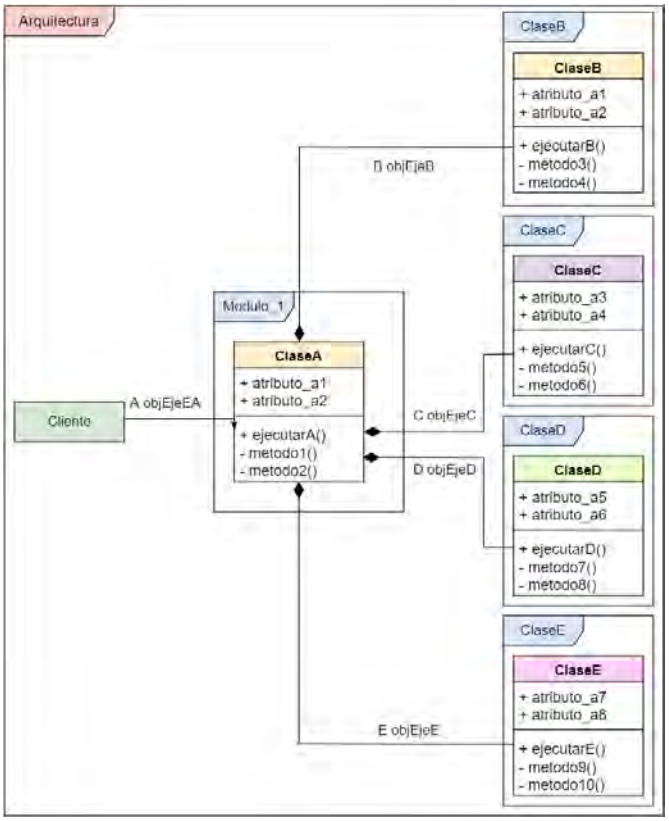

Figura 45 Escenario 6 Refactorizado

Prueba escenario 6

Para la prueba del escenario 6 se cuenta con la arquitectura y carpeta de archivos mostradas en la Figura 46.

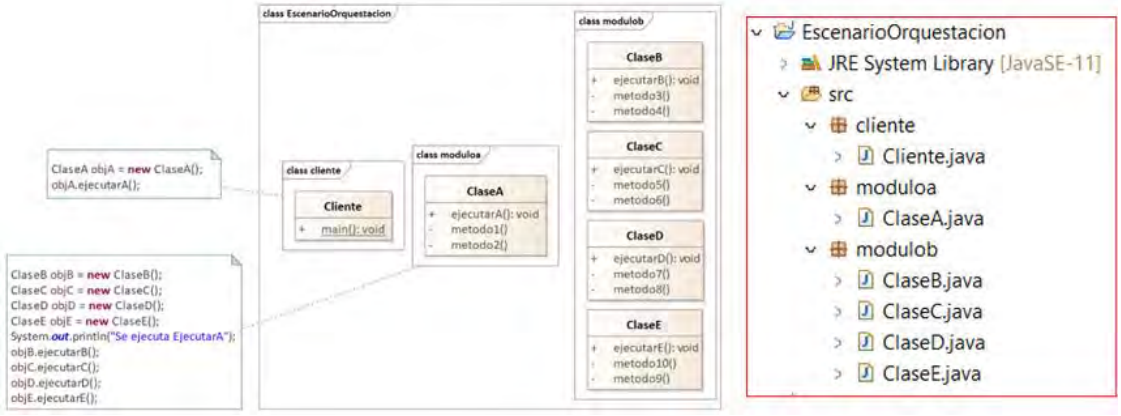

Figura 46 Proyecto Escenario 6, arquitectura original antes de refactorizar

Como código de ejemplo para la Clase Orquestadora se tiene el código mostrado en la Figura 47.

```
1  package moduloa;
2
3  import modulob.ClaseB;
4  import modulob.ClaseC;
5  import modulob.ClaseD;
6  import modulob.ClaseE;
7
8  public class ClaseA {
9
10     public void ejecutarA() {
11         ClaseB objB = new ClaseB();
12         ClaseC objC = new ClaseC();
13         ClaseD objD = new ClaseD();
14         ClaseE objE = new ClaseE();
15         System.out.println("Se ejecuta EjecutarA");
16         objB.ejecutarB();
17         objC.ejecutarC();
18         objD.ejecutarD();
19         objE.ejecutarE();
20     }
21 }
```

Figura 47 Código de ClaseA

Se puede observar que ClaseA sirve como punto de entrada, es decir es la clase Orquestadora. Para comprobar si la arquitectura incumple con el principio de única responsabilidad ejecutamos la métrica, obteniendo como resultado lo que se observa en la Figura 48.

```
[     Valores antes de la Refactorización     ]
================================================
Responsabilidad de paquete : moduloa
Número de entradas: 1
|Clase                  Entradas|
|ClaseA.ejecutarA               |
Responsabilidad: 1.0
================================================
Responsabilidad de paquete : modulob
Número de entradas: 4
|Clase                  Entradas|
|ClaseB.ejecutarB               |
|ClaseD.ejecutarD               |
|ClaseC.ejecutarC               |
|ClaseE.ejecutarE               |
Responsabilidad: 0.25
```

Figura 48 Escenario 6. Resultado de la métrica

Al ser una arquitectura que violenta el principio de única responsabilidad, se aplica el método de refactorización para mejorar las responsabilidades, obteniendo la arquitectura que se muestra en la Figura 49.

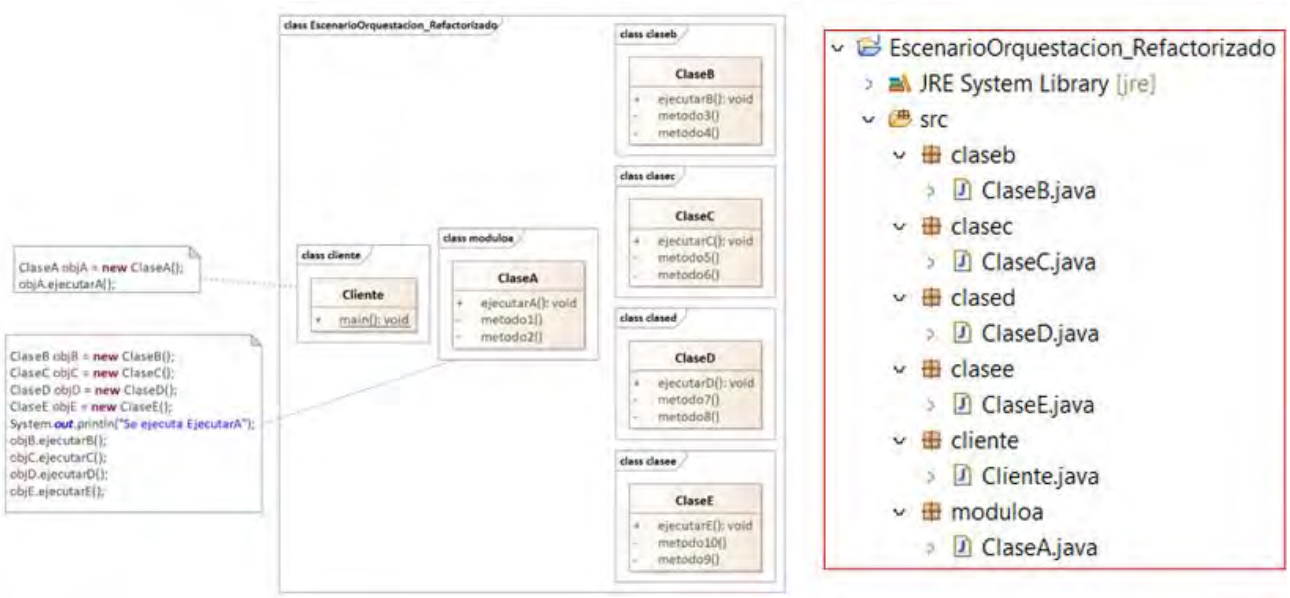

Figura 49 Proyecto Escenario 6, arquitectura después de refactorizar

Para comprobar que la refactorización disminuye las responsabilidades en arquitecturas modulares, ejecutamos nuevamente la métrica, pero ahora sobre la arquitectura refactorizada, los valores obtenidos se muestran en la Figura 50. Estos valores nos indicaran que cada módulo perteneciente a la arquitectura mejoró en sus responsabilidades.

```
-----Valores después de la Refactorización-----
|Paquete            Responsabilidad|
|cliente                   0.0     |
|moduloa                   1.0     |
|clasee                    1.0     |
|clased                    1.0     |
|clasec                    1.0     |
|claseb                    1.0     |
```

Figura 50 Escenario 6. Resultado de la métrica después de la refactorización

5.5.7 Escenario 7 – Orquestación – Coreografía

Otro tipo de escenario contemplado es cuando existen arquitecturas modulares con estilos de combinados; tal es el caso de la Orquestación-Coreografía o viceversa.

En la Figura 51 se puede apreciar un ejemplo de arquitectura modular Orquestada-Coreografiada.

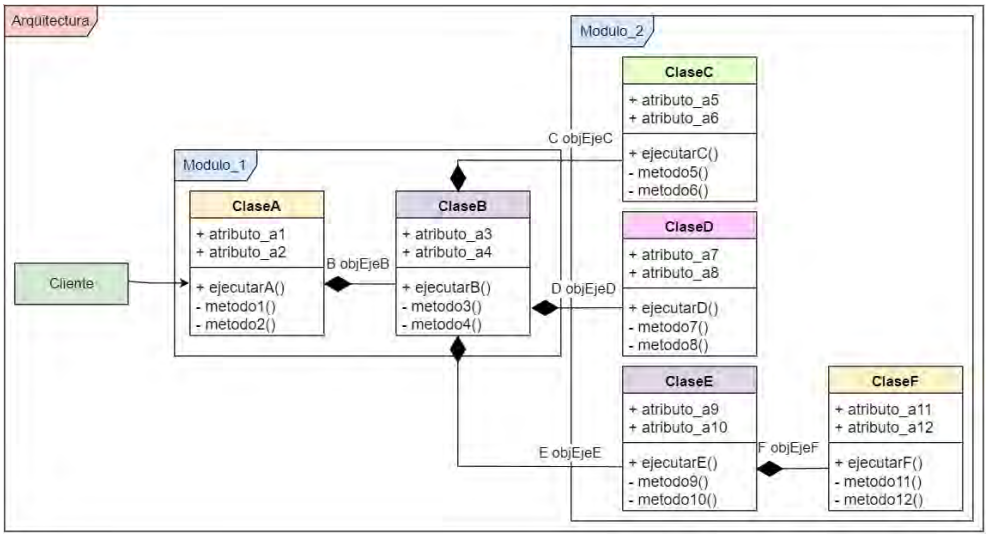

Figura 51 Escenario 7. Orquestación-Coreografía

Como ejemplo, en la Tabla 11 se puede observar cómo puede llegar a tener secuencias de métodos la arquitectura anterior:

Tabla 11 Escenario 7. Ejemplo de Secuencias de Métodos

Módulo	**Clases**	**Secuencias**	**Responsabilidades (Secuencias de Métodos)**
Módulo_1	ClaseA ClaseB	***Cliente*** → *ejecutarA()* → *metodo1()* → *metodo2()*→ *ejecutarB()*→ *metodo3()* → *metodo4()*	1
Modulo_2	ClaseB ClaseC	***Cliente*** → *ejecutarB()* → *metodo3()* → *metodo4()*→ *ejecutarC()*→ *metodo5()* → *metodo6()*	1
	ClaseB ClaseD	***Cliente*** → *ejecutarB()* → *metodo3()* → *metodo4()*→ *ejecutarD()*→ *metodo7()* → *metodo8()*	1
	ClaseB ClaseE ClaseF	***Cliente*** → *ejecutarB()* → *metodo3()* → *metodo4()*→ *ejecutarE()*→ *metodo9()* → *metodo10()*→ *ejecutarF()*→ *metodo11()* → *metodo12()*	1
	Número total de Responsabilidades Modulo_1		1
	Número total de Responsabilidades Modulo_2		3

Dado que la arquitectura presenta más de una responsabilidad en el Módulo_2, tiene que ser evaluada y refactorizada. Al llevar a cabo la refactorización, somo salida se tiene la arquitectura que se muestra en la Figura 52, donde cada módulo estará formado por una sola secuencia de métodos, es decir una única responsabilidad.

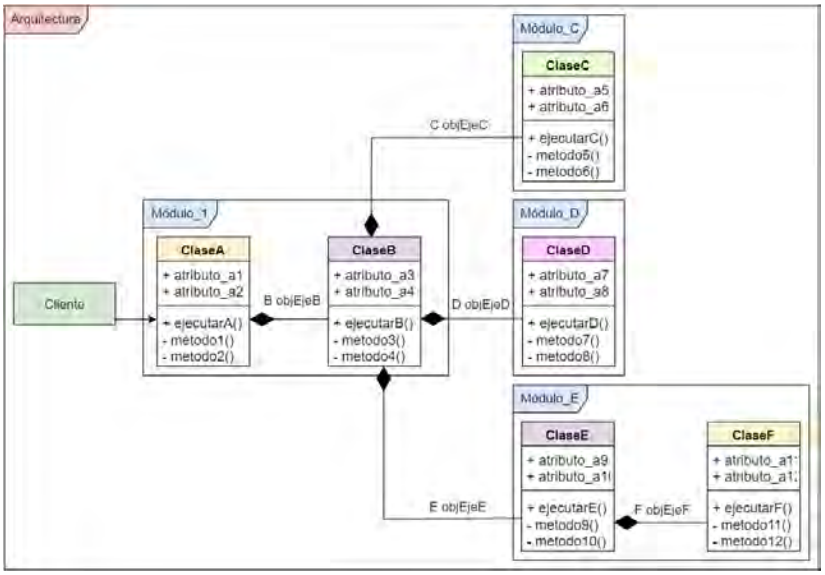

Figura 52 Escenario 7. Refactorizado

Prueba escenario 7

En la Figura 53 se muestra la arquitectura de prueba para el escenario 7, así como la carpeta contenedora de archivos.

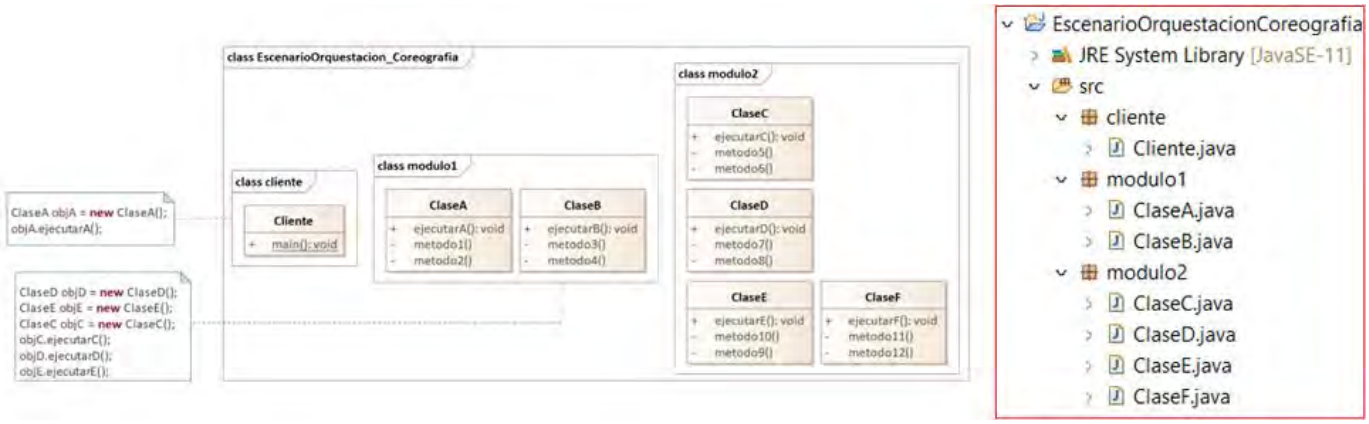

Figura 53 Proyecto Escenario 7, arquitectura original antes de refactorizar

En la Figura 54 se muestra un código de ejemplo de la Clase cliente.

```
1  package cliente;
2
3  public class Main {
4
5      static Cliente cliente = new Cliente();
6
7      public static void main(String[] args) {
8          f1();
9      }
10
11     private static void f1() {
12         System.out.println("Se ejecuta a f1");
13         cliente.f2();
14     }
15
16 }
```

Figura 54 Código de ejemplo

Se puede observar que la Clase Cliente sirve como punto de entrada, es decir es la clase que inicia con la Coreografía, posteriormente la ClaseB, se convierte en la Clase Orquestadora. Para comprobar que la Arquitectura violenta el principio de única responsabilidad, ejecutamos la métrica, obteniendo los resultados mostrados en la Figura 55.

```
[   Valores antes de la Refactorización   ]
==========================================
Responsabilidad de paquete : modulo1
Número de entradas: 1
|Clase                  Entradas|
|ClaseA.ejecutarA               |
Responsabilidad: 1.0
==========================================
Responsabilidad de paquete : modulo2
Número de entradas: 3
|Clase                  Entradas|
|ClaseD.ejecutarD               |
|ClaseE.ejecutarE               |
|ClaseC.ejecutarC               |
Responsabilidad: 0.33333334
```

Figura 55 Escenario 7. Resultado de la métrica

Al ser una arquitectura que violenta el principio de única responsabilidad, se aplica el método de refactorización para mejorar las responsabilidades, obteniendo la arquitectura que se muestra en la Figura 56.

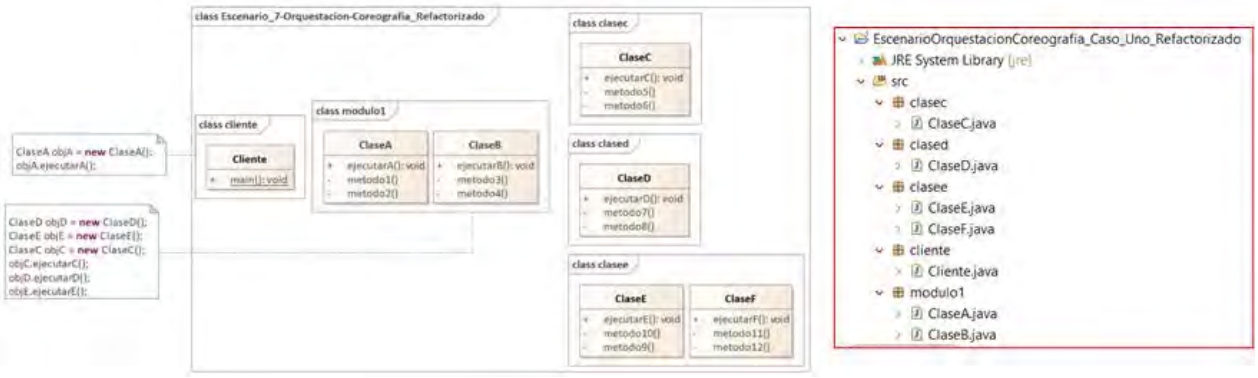

Figura 56 Proyecto Escenario 7, arquitectura después de refactorizar

Para comprobar que la refactorización mejora las responsabilidades en arquitecturas modulares, ejecutamos nuevamente la métrica, pero ahora sobre la arquitectura ya refactorizada, los valores obtenidos se muestran en la Figura 57. Estos valores indicaran que cada módulo perteneciente a la arquitectura mejoró en sus responsabilidades.

```
-----Valores después de la Refactorización--
|Paquete          Responsabilidad|
|cliente                  0.0    |
|modulo1                  1.0    |
|clasee                   1.0    |
|clased                   1.0    |
|clasec                   1.0    |
```

Figura 57 Escenario 7. Resultado de la métrica después de la refactorización

CAPÍTULO 6.- EVALUACIÓN DE PRUEBAS

En este capítulo se describe como se realizaron y planearon las pruebas para comprobar el correcto funcionamiento del método de refactorización de separación de responsabilidades, así como el funcionamiento del cálculo de la métrica de número de responsabilidades NR propuesta en este documento de tesis.

6.1 Convención de nombres

En la Figura 58 se muestra la convención de nombres que se utilizaron durante la evaluación del método de refactorización de responsabilidad única.

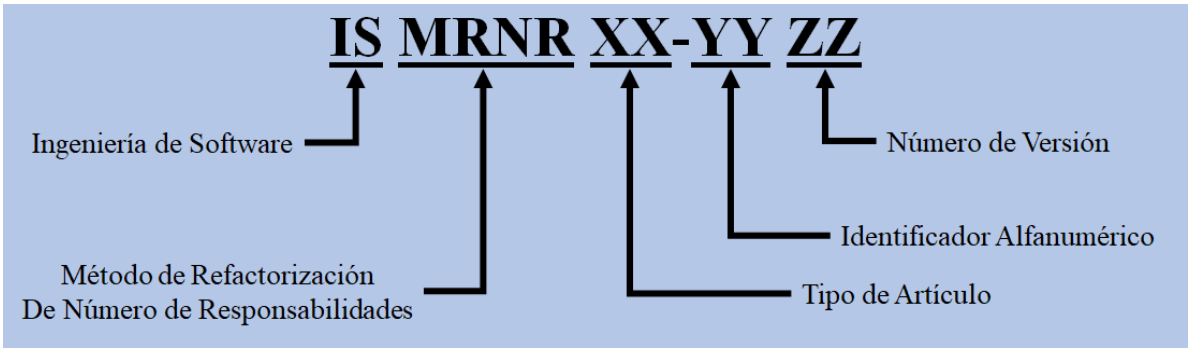

Figura 58 Convención de nombres

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

Programas de Control	
ISMRNR01-YYZZ	Módulo de programa.

Módulo de Programa	
ISMRNR02-YYZZ	Programas de control, utilerías, ordenadores, entre otros.

Documentación de Pruebas	
ISMRNR03-YYZZ	Plan de pruebas.
ISMRNR04-YYZZ	Especificación de diseño de pruebas.
ISMRNR05-YYZZ	Especificación de casos de prueba.

6.2 Plan de Pruebas

1.- Plan de Prueba: ISMRNR03-01: Plan de pruebas para la validación del correcto funcionamiento del método de refactorización de única responsabilidad.

2.- Introducción

El método de refactorización de arquitecturas modulares orientados a objetos con más de una responsabilidad, tiene como finalidad analizar la secuencia de métodos iniciadas por un

método con calificador de alcance *"public"* y analizar las relaciones existentes entre los métodos y atributos dentro de un módulo, para así cumplir con el principio de diseño de única responsabilidad.

3.- Artículos de prueba. En la Tabla 12 se muestran los módulos a ser probados.

Tabla 12 Módulos de programa.

Sistema	Función	No.
Método de Refactorización de Única Responsabilidad	Subsistema de evaluación de secuencia de métodos.	ISMRNR01- 01
Marco de Métricas de Calidad de Arquitecturas Orientadas a Objetos	Cálculo de métrica de número de responsabilidad.	ISMRNR01- 02

4.- Procedimiento de control de tareas. Se muestra en la Tabla 13.

Tabla 13 Control de tareas.

Sistema	Función	No.
Programa de aplicación	Localización del paquete al que pertenece la clase.	ISMRNR02- 01
Programa de aplicación	Localización de las importaciones de la clase.	ISMRNR02- 02
Programa de aplicación	Localización del nombre de la clase.	ISMRNR02- 03
Programa de aplicación	Localización de la clase padre de la clase (si existiera).	ISMRNR02- 04
Programa de aplicación	Localización de los atributos de la clase.	ISMRNR02- 05
Programa de aplicación	Localización de los métodos de la clase.	ISMRNR02- 06
Programa de aplicación	Localización de la información de los métodos de clase, tales como; calificador de alcance, si es abstracto, si es estático, nombre, parámetros, sentencias del cuerpo del método.	ISMRNR02- 07
Programa de aplicación	Conteo de las secuencias de métodos iniciadas por un método con calificador de alcance *"public"* existente en las clases.	ISMRNR02- 08
Programa de aplicación	Conteo del número de relaciones entre los métodos y atributos existentes en las clases.	ISMRNR02- 09
Programa de aplicación	Cálculo de la métrica NR.	ISMRNR02- 10
Programa de aplicación	Método de Refactorización de Única Responsabilidad.	ISMRNR02- 11
Programa de aplicación	Llenado de la plantilla ST con código Refactorizado.	ISMRNR02- 12
Programa de aplicación	Compilación del código Refactorizado.	ISMRNR02- 13
Programa de aplicación	Verificación del funcionamiento del código Refactorizado.	ISMRNR02- 14

5.- Características a ser probadas.

A continuación, se listan las características que deben de ser probadas.

Tabla 14 Características a probar.

Diseño de Prueba No. de especificación	Descripción
ISMRNR04-01	Cálculo de métrica de calidad.
ISMRNR04-02	Método de Refactorización de Única Responsabilidad.

6.- Características a NO ser probadas.

6.1.- Los casos de prueba no incluyen todas las posibles construcciones sintácticas y combinaciones de éstas para código escrito en lenguaje *"Java"*.

6.2.- El método no indica si el código de entrada se encuentra libre de errores.

6.3.- No se pretende comprobar que la totalidad del proceso de reingeniería para reuso es automático, sino que se reconoce que es necesario cierto nivel de intervención del experto en el dominio o el experto en programación.

6.4.- Así mismo, no se comprueba la interfaz del sistema.

7.- Enfoque

La realización de los casos de prueba y la actividad de pruebas estará asistida por el alumno de maestría del Cenidet Juan Antonio Díaz Díaz. Esto ayudará para asegurar que las pruebas representan efectivamente el desarrollo y uso del sistema SR2 completo.

7.1.- Pruebas del proceso de análisis del código fuente. La comprobación del proceso de análisis de código fuente, se realiza mediante la obtención de información de cada una de las clases que se reciban como entrada, como son: paquete, librerías, nombre, funciones y atributos de cada una de ellas, comprobándose además la generación de la lista de clases de objetos requeridas por el método de refactorización.

7.2.- Pruebas de Refactorización. La validación del método de Refactorización de arquitecturas de marcos orientados a objetos con más de una responsabilidad, que consiste en dividir las secuencias de métodos en arquitecturas modulares, se realiza mediante la ejecución del código original contra la ejecución del código refactorizado. Comprobando, que, bajo las mismas entradas, ambos sistemas deben comportarse de la misma manera y ofrecer los mismos resultados.

7.3.- Pruebas de Calidad. Las pruebas de calidad incluyen la aplicación de la métrica de número de responsabilidades, para la medición comparativa del código original contra el código orientada a objetos obtenido durante el proceso de refactorización.

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

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

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

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

9.- Criterio de suspensión y requisitos de reanudación.

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

10.- Librería de pruebas.

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

6.3 Especificación del diseño de pruebas

6.3.1 Diseño de Prueba: ISMRNR04-01

1.- Diseño de Prueba: ISMRNR04-01. Método de refactorización de Única Responsabilidad.

2.- Características a ser probadas.

2.1. Durante esta prueba se evaluará el correcto funcionamiento del método de refactorización de única responsabilidad.

3.- Refinamiento del enfoque.

El objetivo es evaluar el correcto funcionamiento del método de refactorización de única responsabilidad, al dividir las secuencias de métodos de arquitecturas modulares de acuerdo al número de responsabilidades.

Antes de realizar cada caso de prueba, el sistema legado deberá ser compilado y ejecutado previamente en un compilador para el lenguaje Java, con la finalidad de corroborar que su construcción está correctamente escrita. Posteriormente, el subsistema de refactorización de código tomará este archivo y realizará un reconocimiento léxico y sintáctico, generando las estructuras de datos en memoria con la información necesaria para efectos de la refactorización.

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

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

6.3.2 Diseño de Prueba: ISMRNR04-02

1.- Diseño de Prueba: ISMRNR04-02. Cálculo de métrica de calidad.

2.- Características a ser probadas.

2.1. En esta prueba se evaluará el correcto cálculo de la métrica de número de responsabilidades (NR).

3.- Refinamiento del enfoque.

El objetivo es evaluar el correcto funcionamiento de la métrica de única responsabilidad al evaluar un sistema.

Antes de realizar cada caso de prueba, éste será compilado y ejecutado previamente en un compilador para el lenguaje Java, con el objetivo de corroborar que su construcción está correctamente escrita. Posteriormente, el Marco Orientado a Objetos para el cálculo de métricas tomará este archivo y realizará un reconocimiento léxico y sintáctico, generando las estructuras de datos en memoria con la información requerida para dicho cálculo.

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

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

6.4 Especificación de Casos de Prueba

6.4.1 Caso de Prueba ISMRNR05-01

1. Artículo de Prueba: PSPCenidet

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

2. Las Características a probar del proceso del cálculo de la métrica de calidad se muestra en la Tabla 15.

Tabla 15 Características a probar del proceso del cálculo de la métrica de calidad

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-02	Cálculo de la métrica (NR) Número de Responsabilidades.

3. Especificación de entrada.

Las entradas al proceso de medición de la métrica de calidad son:

- El código fuente del PSPCenidet compilado y probado previamente.

4. Especificación de salida.

En la salida del proceso de medición de la métrica se deberá tener:

- El valor obtenido de la métrica.

6.4.2 Caso de Prueba ISMRNR05-02

1. Artículo de Prueba: PSP Cenidet

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

2. Características a probar del proceso del método de refactorización de única responsabilidad se muestran en la Tabla 16.

Tabla 16 Características a probar del proceso de refactorización.

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-01	Búsqueda del Cliente.
ISMRNR04-01	Análisis de secuencia de métodos.
ISMRNR04-01	Llenado de plantilla ST.
ISMRNR04-01	Generación de código refactorizado en Java.
ISMRNR04-01	Compilación del código refactorizado.

3. Especificación de entrada.

Las entradas al proceso de reestructura son:

- El código fuente del PSPCenidet compilado y probado previamente.

4. Especificación de salida.

En la salida del proceso de refactorización se tiene:

- El código fuente original refactorizado libre de deuda técnica producida por el código desagradable que se definió como arquitecturas modulares con más de una responsabilidad.
- Funcionalidad de la arquitectura PSPCenidet una vez refactorizada.

6.4.3 Caso de Prueba ISMRNR05-03

1. Artículo de Prueba: Marco Estadístico

El Marco Estadístico es un sistema desarrollado en lenguaje Java que tiene como finalidad realizar automáticamente cálculos estadísticos.

2. Las Características a probar del proceso del cálculo de la métrica de calidad se muestra en la Tabla 17.

Tabla 17 Características a probar del proceso del cálculo de la métrica de calidad.

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-02	Cálculo de la métrica (NR) Número de Responsabilidades.

3. Especificación de entrada.

Las entradas al proceso de medición de la métrica de calidad son:

- El código fuente del Marco Estadístico compilado y probado previamente.

4. Especificación de salida.

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

- El valor obtenido de la métrica.

6.4.4 Caso de Prueba ISMRNR05-04

1. Artículo de Prueba: Marco Estadístico

El Marco Estadístico es un sistema desarrollado en lenguaje Java que tiene como finalidad realizar automáticamente cálculos estadísticos.

2. Características a probar del proceso del método de refactorización de única responsabilidad se muestran en la Tabla 18.

Tabla 18 Características a probar del proceso de refactorización.

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-01	Búsqueda del Cliente.
ISMRNR04-01	Análisis de secuencia de métodos.
ISMRNR04-01	Llenado de plantilla ST.
ISMRNR04-01	Generación de código refactorizado en Java.
ISMRNR04-01	Compilación del código refactorizado.

3. Especificación de entrada.

Las entradas al proceso de reestructura son:

- El código fuente del Marco Estadístico compilado y probado previamente.

4. Especificación de salida.

En la salida del proceso de refactorización se tiene:

- El código fuente original refactorizado libre de deuda técnica producida por el código desagradable que se definió como arquitecturas modulares con más de una responsabilidad.

6.4.5 Caso de Prueba ISMRNR05-05

1. Artículo de Prueba: ProyectoAFND Autómata

El Proyecto Autómata tiene como finalidad la creación de un AFN (Autómata Finito no Determinista) por medio del algoritmo de Thomson. Así como también la conversión del AFN a un AFD (Autómata Finito Determinista) por medio de la agrupación de subconjuntos.

Además, dicho proyecto crea directamente un AFD por medio de un árbol sintáctico y agrupación, y por último también puede llevar a cabo la minimización de un AFD de forma directa, así como también su minimización por medio de subconjuntos.

2. Las Características a probar del proceso del cálculo de la métrica de calidad se muestran en la Tabla 19.

Tabla 19 Características a probar del proceso del cálculo de la métrica de calidad

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-02	Cálculo de la métrica (NR) Número de Responsabilidades.

3. Especificación de entrada.

Las entradas al proceso de medición de la métrica de calidad son:

- El código fuente del ProyectoAFND compilado y probado previamente.

4. Especificación de salida.

En la salida del proceso de medición de las métricas se tiene:

- El valor obtenido de la métrica.

6.4.6 Caso de Prueba ISMRNR05-06

1. Artículo de Prueba: ProyectoAFND Autómata

El Proyecto Autómata tiene como finalidad la creación de un AFN (Autómata Finito no Determinista) por medio del algoritmo de Thomson. Así como también la conversión del AFN a un AFD (Autómata Finito Determinista) por medio de la agrupación de subconjuntos.

Además, dicho proyecto crea directamente un AFD por medio de un árbol sintáctico y agrupación, y por último también puede llevar a cabo la minimización de un AFD de forma directa, así como también su minimización por medio de subconjuntos.

2. Características a probar del proceso del método de refactorización de única responsabilidad se muestran en la Tabla 20.

Tabla 20 Características a probar del proceso de refactorización.

No. de especificación del diseño de prueba	Característica a Ejecutar
ISMRNR04-01	Búsqueda del Cliente.
ISMRNR04-01	Análisis de secuencia de métodos.
ISMRNR04-01	Llenado de plantilla ST.
ISMRNR04-01	Generación de código refactorizado en Java.
ISMRNR04-01	Compilación del código refactorizado.

3. Especificación de entrada.

Las entradas al proceso de reestructura son:

- El código fuente de la arquitectura ProyectoAFND1 compilado y probado previamente.

4. Especificación de salida.

En la salida del proceso de refactorización se tiene:

- El código fuente original refactorizado libre de deuda técnica producida por el código desagradable que se definió como arquitecturas modulares con más de una responsabilidad.
- Funcionalidad de la arquitectura ProyectoAFND una vez refactorizada.

6.5 Ejecución de Pruebas

6.5.1 Caso de Prueba ISMRNR05-01

Artículos de Prueba: PSPCenidet

En la Tabla 21 se enlistan las características a probar del proceso del cálculo de la métrica de calidad de número de responsabilidades en la aplicación PSPCenidet.

Tabla 21 Características a probar del proceso de cálculo de la Métrica de Calidad,

Características a Probar
Conteo de puntos de entrada.
Cálculo de la métrica (NR) Número de Responsabilidades.

En la Figura 59 se muestra la carpeta que contiene los archivos que conforman el proyecto PSPCenidet y en la Figura 60 se puede observar la arquitectura modular de dicho proyecto,

el cual tiene como finalidad medir los tiempos que los usuarios utilizan para realizar tareas cotidianas, así como también tareas relacionadas con el desarrollo de software.

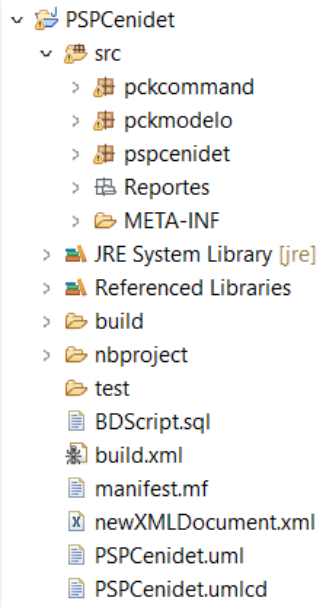

Figura 59 Carpeta del Proyecto PSPCenidet

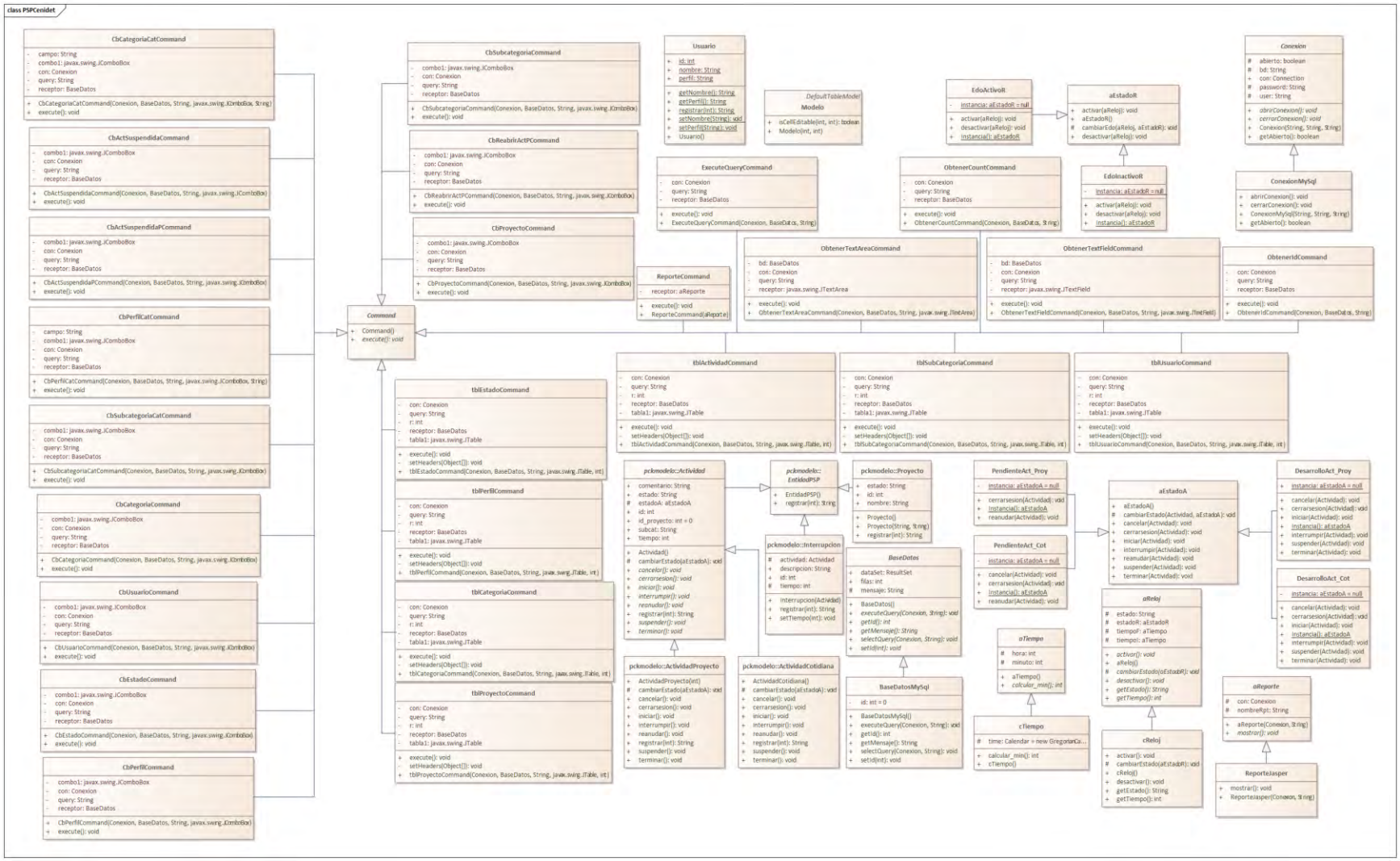

Figura 60 Arquitectura Proyecto PSPCenidet

Conteo de puntos de Entrada.

Para verificar que el conteo de puntos de entrada se realiza de manera correcta, primero se realiza el conteo de forma manual en el módulo *"pckcommand"* de la arquitectura modular de la aplicación PSPCenidet, posteriormente se realiza un conteo de forma automática para la comprobación de ambos resultados.

En la Tabla 22 se puede observar que en el conteo manual y el conteo automático se obtuvieron los mismos valores, por lo que se comprueba que el conteo de puntos de entrada se realizó de manera correcta.

Tabla 22 Conteo manual y automática de los puntos de entrada

Conteo de forma manual el número de puntos de entrada	**Conteo de forma automática el número de puntos de entrada**
cbreabriractcommand	[NR: Módulo a Refactorizar] ========== Responsabilidad de paquete : pckcommand Número de entradas: 24 |Clase |CbPerfilCatCommand | |CbReabrirActPCommand | |CbCategoriaCatCommand | |tblEstadoCommand | |tblPerfilCommand | |ObtenerTextFieldCommand | |CbCategoriaCommand | |ObtenerIdCommand | |CbUsuarioCommand | |tblSubCategoriaCommand | |CbSubcategoriaCommand | |tblActividadCommand | |ObtenerTextAreaCommand | |CbActSuspendidaCommand | |CbProyectoCommand | |ObtenerCountCommand | |Command | |tblUsuarioCommand | |CbActSuspendidaPCommand | |CbSubcategoriaCatCommand | |CbPerfilCommand | |ReporteCommand | |ExecuteQueryCommand | |tblCategoriaCommand |
obteneridcommand	
cbperfilcatcommand	
cbactsuspendidapcommand	
cbsubcategoriacatcommand	
cbsubcategoriacommand	
executequerycommand	
obtenercountcommand	
tblaactividadcommand	
cbcategoriacommand	
obtenertextfieldcommand	
cbproyectocommand	
tblcategoriacommand	
tblperfilcommand	
command	
tblusuariocommand	
obtenertextareacommand	
reportecommand	
tblesubcategoriacommand	
cbcategoriacatcommand	
tblestadocommand	
cbactsuspendidacommand	
cbusuariocommand	
cbperfilcommand	
Número de entradas: 24	**Número de entradas:** 24

Cálculo correcto de métrica de Responsabilidades (NR)

Una vez hecho el conteo de los puntos de entrada, se procede a calcular la métrica de número de responsabilidades y para comprobar que se hace de manera correcta, primero se lleva a cabo de manera manual y posteriormente de manera automática, con la finalidad de comprobar que ambos resultados son iguales.

El cálculo de la métrica se hace al proyecto general, para los módulos: *"pckmodelo" y "pckcommand".*

Cálculo de la métrica de Número de Responsabilidades (NR)

Expresión matemática:

$$NR = \frac{1}{\sum_{Pi=1}^{Pi=n} secuencia_i}$$

Donde:

NR = Número de responsabilidades

Pi = *"i-esimo"* punto de entrada.

Secuencia = Número total de puntos de entrada

Los valores del cálculo manual y automático de la métrica NR, correspondientes a la arquitectura de la Figura 60 se pueden observar en la Tabla 22. Además, se puede observar que el valor de la métrica NR obtenido de forma manual y automática son iguales, por lo que se comprueba que el cálculo se lleva a cabo de forma correcta.

Tabla 23 Calculo manual y automático de la Métrica NR de la aplicación PSPCenidet

Módulo	Calculo manual	Calculo automático
pckmodelo	$NR = \frac{1}{29} = 0.034483$	--------Valores antes de la Refactorización-------- |Paquete Responsabilidad| |pckmodelo 0.03448276 | |pspcenidet 0.0 | |pckcommand 0.041666668 |
pckcommand	$NR = \frac{1}{24} = 0.0416667$	

Al ser una arquitectura que violenta el principio de única responsabilidad se procede a llevar a cabo la refactorización.

6.5.2 Caso de Prueba ISMRNR05-02

Artículos de Prueba: PSPCenidet

En la Tabla 24 se enlistan las Características a probar del proceso de refactorización para la aplicación PSPCenidet.

Tabla 24 Características a probar del proceso de refactorización para la aplicación PSPCenidet

Características a Probar
Separación de responsabilidades.
Comprobar que el comportamiento de la aplicación se mantiene después de la Refactorización.
Disminuir el número de responsabilidades en arquitecturas modulares.

Separación de responsabilidades en la Aplicación PSPCenidet

Para comprobar que la separación de responsabilidades se realiza de manera correcta, primero se analizan las clases que pueden servir como punto de entrada hacía secuencia de métodos.

En la Figura 61 se muestra la arquitectura modular de la aplicación PSPCenidet refactorizada, en donde se puede observar que cada responsabilidad es separa en un módulo diferente.

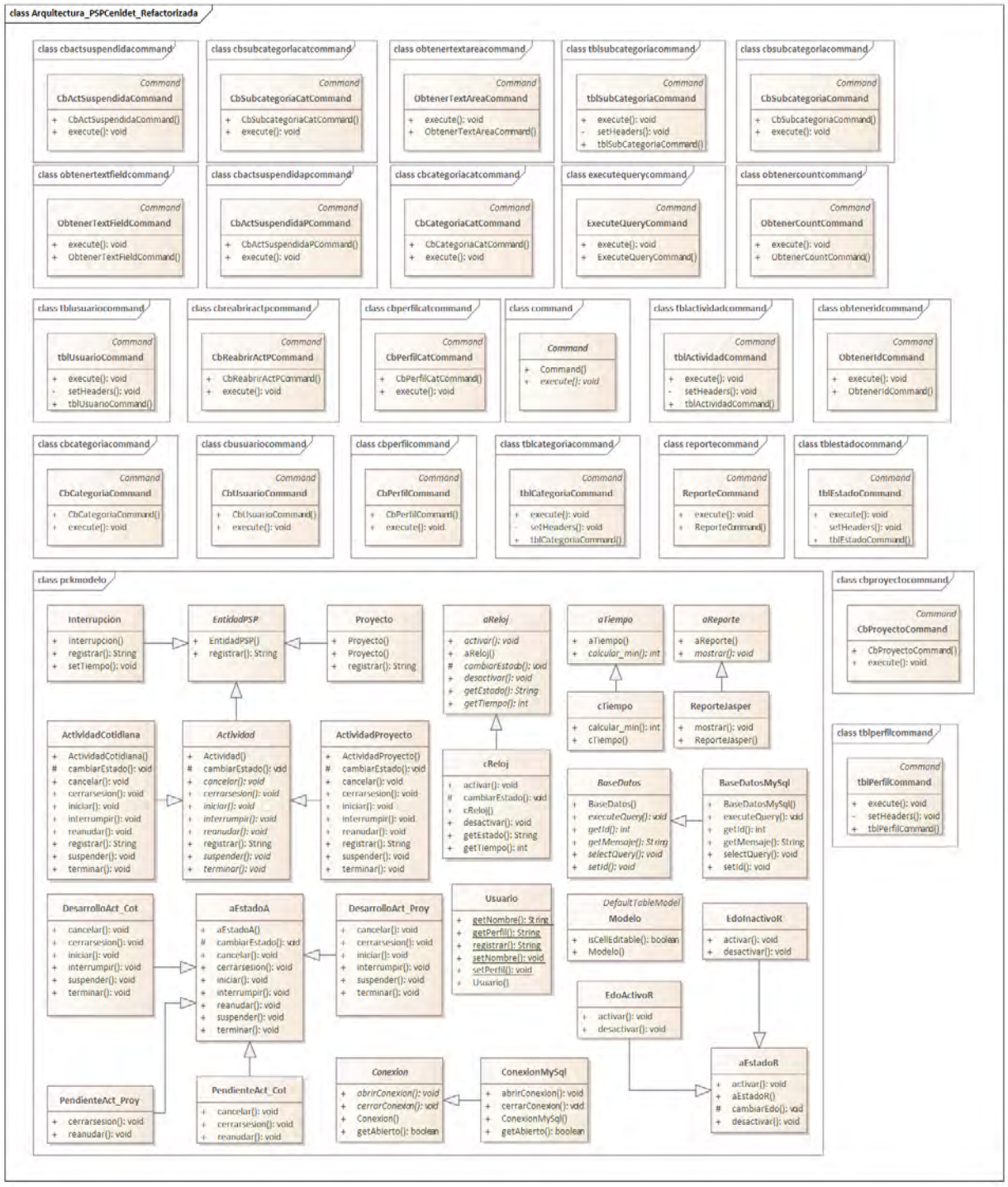

Figura 61 Arquitectura Proyecto PSPCenidet Refactorizada

Comprobar que el comportamiento de la aplicación PSPCenidet se mantiene después de la Refactorización.

Para comprobar que la aplicación PSPCenidet mantiene el mismo comportamiento después de ser sometido al método de refactorización, se procede a agregar un usuario, esto con la finalidad de verificar el funcionamiento de la aplicación.

La Figura 62 muestra a los usuarios registrados en la aplicación PSPCenidet antes de agregar al nuevo usuario, se puede observar que solo se cuenta con tres usuarios registrados.

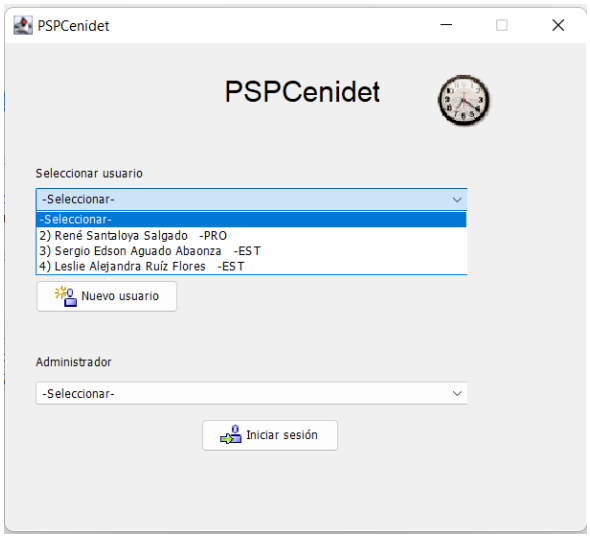

Figura 62 Usuarios registrados en la aplicación PSPCenidet antes de la Refactorización

En la Figura 63 se muestra el registro de un nuevo usuario ejecutando la aplicación PSPCenidet Refactorizada.

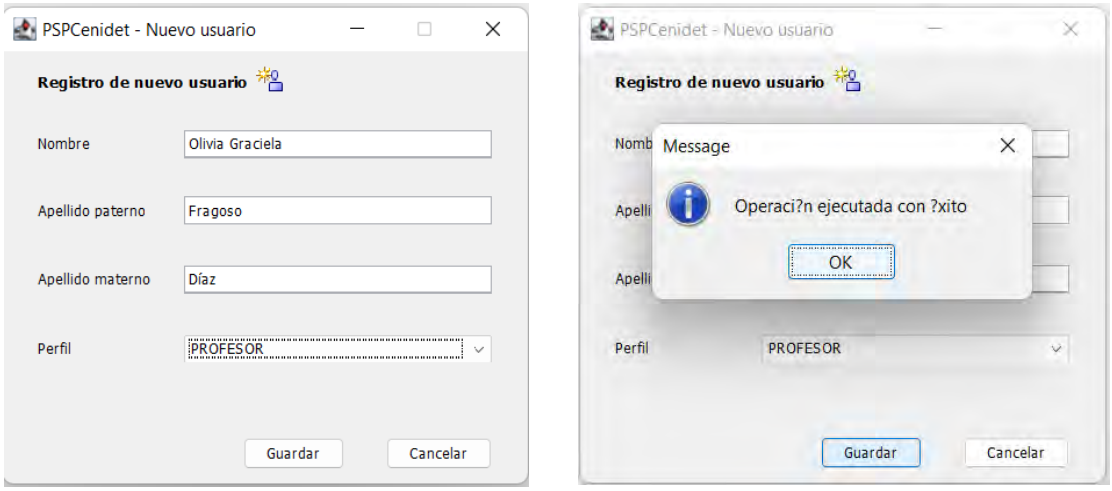

Figura 63 Registro de nuevo usuario en Aplicación PSPCenidet_Refactorizado

Se puede comprobar que el funcionamiento de la aplicación PSPCenidet_Refactorizada sigue siendo el mismo después de ser sometida al método de refactorización, por lo que se concluye que el dicho método no altera el comportamiento de la aplicación.

Comprobación en la disminución del número de Responsabilidades.

Para comprobar que el número de responsabilidades en la aplicación PSPCenidet disminuyó, es sometida al cálculo de la métrica NR antes y después de la refactorización.

Para ello al descomponer la arquitectura modular a sus respectivas responsabilidades, esto provoca que se creen nuevos módulos, cada uno de ellos con una única responsabilidad. Con base a esto, para llevar a cabo una interpretación adecuada, aplicamos la ecuación mostrada en la Tabla 25, que consiste en sumar los valores (responsabilidades) obtenidos para cada módulo que fue creado a partir de la arquitectura a refactorizar y mejorada, esta suma se dividide entre el número total de nuevos módulos creados.

En la Tabla 25 se muestra la comparación de los valores de la métrica NR.

Tabla 25 Comparación de valores de la Métrica NR antes y después de la Refactorización de la Aplicación PSPCenidet

Módulo	**Valor antes de la Refactorización**	**Módulo**	**Valor después de la Refactorización**
pckcommand	0.03846154	cbreabriractcommand	1.0
		obteneridcommand	1.0
		cbperfilcatcommand	1.0
		cbactsuspendidapcommand	1.0
		cbsubcategoriacatcommand	1.0
		cbsubcategoriacommand	1.0
		executequerycommand	1.0
		obtenercountcommand	1.0
		tblaactividadcommand	1.0
		cbcategoriacommand	1.0
		obtenertextfieldcommand	1.0
		cbproyectocommand	1.0
		tblcategoriacommand	1.0
		tblperfilcommand	1.0
		command	1.0
		tblusuariocommand	1.0
		obtenertextareacommand	1.0
		reportecommand	1.0
		tblesubcategoriacommand	1.0
		cbcategoriacatcommand	1.0
		tblestadocommand	1.0
		cbactsuspendidacommand	1.0
		cbusuariocommand	1.0
		cbperfilcommand	1.0
TOTAL	0.03846154	**TOTAL** $NR = \frac{\sum_{Pi=1}^{Pi=N} NR}{N}$	$NR = \frac{24}{24} = 1$

En el Gráfico 1 se muestran los valores obtenidos para la métrica NR antes y después de la refactorización para la aplicación PSPCenidet, se observa que para la arquitectura

refactorizada se llegó al mejor de los resultados esperados, que consiste en mantener dicha arquitectura con una única responsabilidad.

El número de responsabilidades para la arquitectura "pckcommand" mejoró de un 3.85% a un 100%. Con esto se comprueba que el método de refactorización desarrollado en este trabajo de tesis es capaz de disminuir el número de responsabilidades en arquitecturas modulares escritas en lenguaje Java.

Gráfico 1 Responsabilidades PSPCenidet antes y después de la refactorización

Comprobación de mejoría de Coherencia y Cohesión

Para llevar a cabo el cálculo de la métrica de carencia de Cohesión y Coherencia de casos de uso, se utilizaron las métricas desarrolladas en el tema de tesis "Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos".

Estas métricas son las siguientes:

LCOM* (Carencia de Cohesión): mide el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase.

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

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

CrCU (Coherencia de Caso de Uso): esta métrica mide la cantidad de método que interactúan para satisfacer una responsabilidad en relación al número total de métodos del módulo que los contiene.

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

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

En la Tabla 26 se muestran los valores obtenidos para el proyecto PSPCenidet antes y después de la refactorización, se puede observar que el valor obtenido de la métrica LCOM* mejoro, debido a que la mayoría de los módulos de la aplicación PSPCenidet después de la refactorización cuentan con una correcta división entre sus atributos y métodos.

Tabla 26 Valores obtenidos de la Métrica LCOM antes y después de la Refactorización.*

Cohesión - Proyecto PSPCenidet			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
pckcommand	0.1999	CbActSuspendidaCommand	0.0
		CbActSuspendidaPCommand	0.0
		CbCategoriaCatCommand	0.0
		CbCategoriaCommand	0.0
		CbPerfilCatCommand	0.0
		CbPerfilCommand	0.0
		CbProyectoCommand	0.0
		CbReabrirActPCommand	0.0
		CbSubcategoriaCatCommand	0.0
		CbSubcategoriaCommand	0.0
		CbUsuarioCommand	0.0
		Command	0.0
		ExecuteQueryCommand	0.0
		ObtenerCountCommand	0.0
		ObtenerIdCommand	0.0
		ObtenerTextAreaCommand	0.0
		ObtenerTextFieldCommand	0.0
		ReporteCommand	0.0
		tblActividadCommand	0.13
		tblCategoriaCommand	0.13
		tblEstadoCommand	0.13
		tblPerfilCommand	0.13
		tblSubCategoriaCommand	0.13
		tblUsuarioCommand	0.13
		Sumatoria	0.78
Valor	**0.1999**	**Valor**	**0.0325**

En la Tabla 27 se muestran los valores obtenidos de la métrica CrCU antes y después de la refactorización, dado que cada módulo después de la refactorización está conformado por una única responsabilidad, se concluye que cada uno de éstos atiende a una responsabilidad, por lo cual se obtiene el mejor resultado para cada uno de ellos.

Tabla 27 Valores obtenidos de la Métrica CrCU antes y después de la Refactorización.

Coherencia - Proyecto PSPCenidet			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
pckcommand	0.828	CbActSuspendidaCommand	1.0
		CbActSuspendidaPCommand	1.0
		CbCategoriaCatCommand	1.0
		CbCategoriaCommand	1.0
		CbPerfilCatCommand	1.0
		CbPerfilCommand	1.0
		CbProyectoCommand	1.0
		CbReabrirActPCommand	1.0
		CbSubcategoriaCatCommand	1.0
		CbSubcategoriaCommand	1.0
		CbUsuarioCommand	1.0
		Command	1.0
		ExecuteQueryCommand	1.0
		ObtenerCountCommand	1.0
		ObtenerIdCommand	1.0
		ObtenerTextAreaCommand	1.0
		ObtenerTextFieldCommand	1.0
		ReporteCommand	1.0
		tblActividadCommand	1.0
		tblCategoriaCommand	1.0
		tblEstadoCommand	1.0
		tblPerfilCommand	1.0
		tblSubCategoriaCommand	1.0
		tblUsuarioCommand	1.0
		Sumatoria	24.0
Valor	**0.828**	**Valor**	**1.0**

6.5.3 Caso de Prueba ISMRNR05-03

Artículos de Prueba: Marco estadístico.

El marco estadístico es un sistema desarrollado en lenguaje Java que tiene como finalidad realizar automáticamente cálculos estadísticos.

En la Tabla 28 se enlistan las características a probar del proceso del cálculo de métrica de calidad de Número de Responsabilidades en la aplicación Marco Estadístico.

Tabla 28 Características a probar del proceso de cálculo de Métrica de Número de Responsabilidades

Características a Probar
Conteo de puntos de entrada.
Cálculo de la métrica (NR) Número de Responsabilidades.

En la Figura 64 se muestra la carpeta que contiene los archivos que conforman el proyecto Marco Estadístico y en la Figura 65 se puede observar la arquitectura modular, dicho proyecto tiene como finalidad realizar cálculos estadísticos automáticamente.

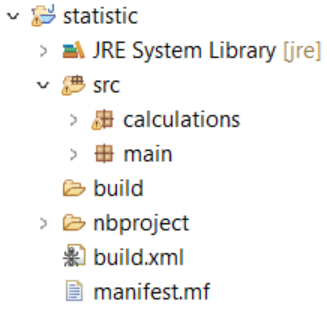

Figura 64 Carpeta de Proyecto Marco Estadístico

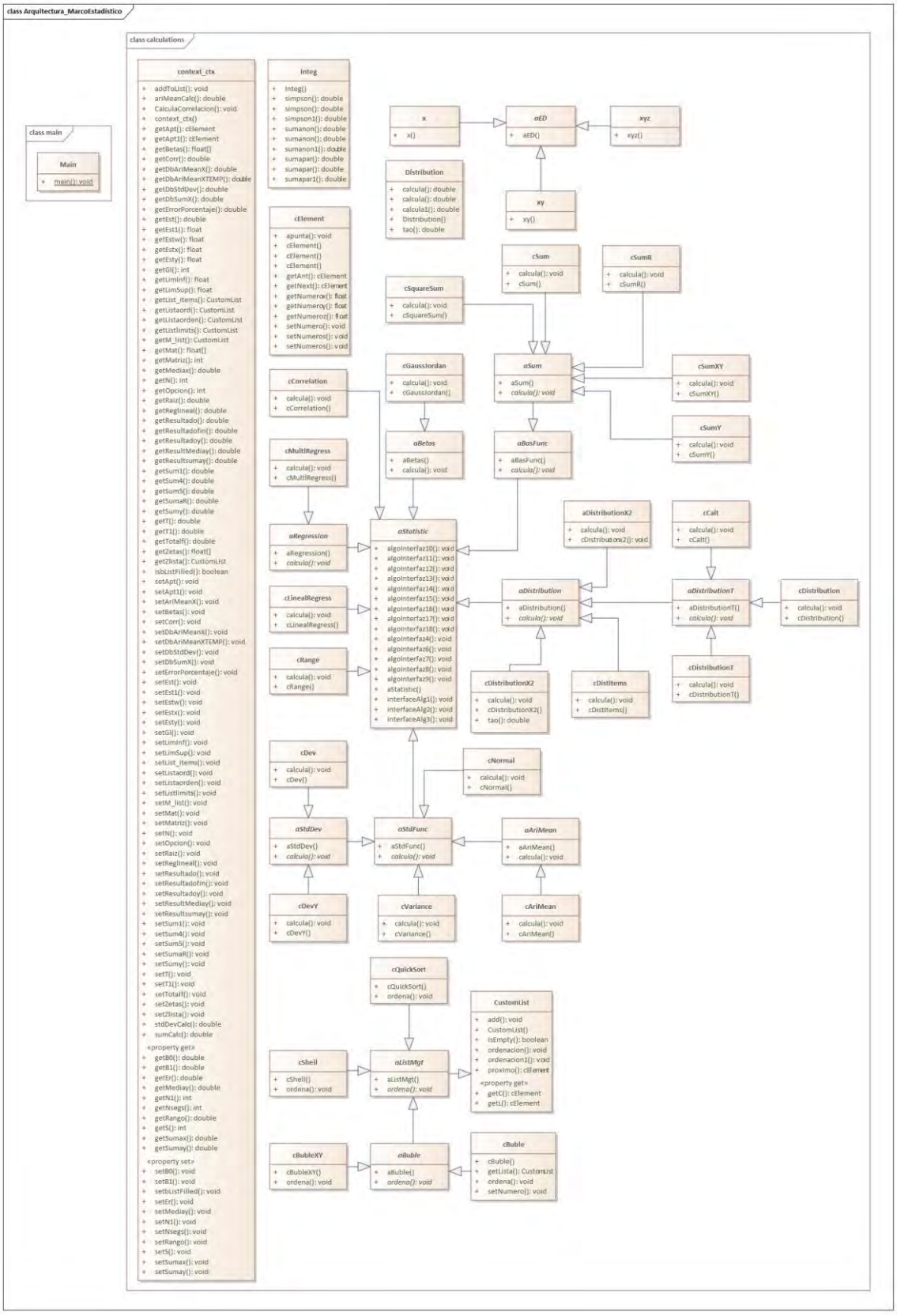

Figura 65 Arquitectura Modular Proyecto Marco Estadístico

Conteo de puntos de Entrada.

Para verificar que el conteo de puntos de entrada se realiza de manera correcta, primero se realiza el conteo de forma manual en el módulo *"calculations"* de la arquitectura modular de la aplicación Marco Estadístico, posteriormente se realiza un conteo de forma automática para la comprobación de ambos resultados.

En la Tabla 29 se puede observar que en el conteo manual y el conteo automático se obtuvieron los mismos valores, por lo que se comprueba que el conteo de puntos de entrada se realizó de manera correcta.

Tabla 29 Conteo manual y automática de los puntos de entrada

Conteo de forma manual el número de puntos de entrada	**Conteo de forma automática el número de puntos de entrada**
cSum	[NR: Módulo a Refactorizar] ========== Responsabilidad de paquete : calculations Número de entradas: 7 |Clase |cSum | |cSumXY | |aSum | |cSumY | |cSumR | |cSquareSum | |context_ctx |
cSumXY	
aSum	
cSumY	
cSumR	
cSquareSum	
context_ctx	
Número de entradas: 7	**Número de entradas:** 7

Cálculo correcto de métrica de Responsabilidades (NR)

Una vez hecho el conteo de los puntos de entrada, se procede a calcular la métrica de número de responsabilidades y para comprobar que se hace de manera correcta, primero se lleva a cabo de manera manual y posteriormente de manera automática, con la finalidad de comprobar que ambos resultados son iguales.

El cálculo de la métrica se realiza para el módulo: *"calculations"*.

Cálculo de la métrica de Número de Responsabilidades (NR)

Expresión matemática:

$$NR = \frac{1}{\sum_{Pi=1}^{Pi=n} secuencia_i}$$

Donde:

NR = Número de responsabilidades

$\boldsymbol{P_i}$ = *"i-esimo"* punto de entrada.

Secuencia = Número total de puntos de entrada

Los valores del cálculo manual y automático de la métrica NR, correspondientes a la arquitectura de la Figura 65 se pueden observar en la Tabla 30.

Tabla 30 Calculo manual y automático de la Métrica NR de la aplicación Marco Estadístico

Módulo	Calculo manual	Calculo automático
calculations	$NR = \frac{1}{7} = 0.14286$	-----Valores antes de la Refactorización----- |Paquete Responsabilidad| |main 0.0 | |calculations 0.14285715 |

Al ser un módulo que violenta el principio de única responsabilidad se procede a llevar a cabo la refactorización.

6.5.4 Caso de Prueba ISMRNR05-04

Artículos de Prueba: Marco Estadístico

En la Tabla 31 se enlistan las características a probar del proceso de refactorización en la aplicación Marco Estadístico.

Tabla 31 Características a probar del proceso de refactorización en la aplicación Marco Estadístico

Características a Probar
Separación de responsabilidades.
Comprobar que el comportamiento de la aplicación se mantiene después de la Refactorización.
Disminuir el número de responsabilidades en arquitecturas modulares.

Separación de responsabilidades en la Aplicación Marco Estadístico

Para comprobar que la separación de responsabilidades se realiza de manera correcta se analizaron las clases que pueden servir como punto de entrada hacía secuencia de métodos.

En la Figura 66 se muestra la arquitectura modular de la aplicación Marco Estadístico refactorizada, en donde se puede observar que cada responsabilidad es separada en un módulo diferente.

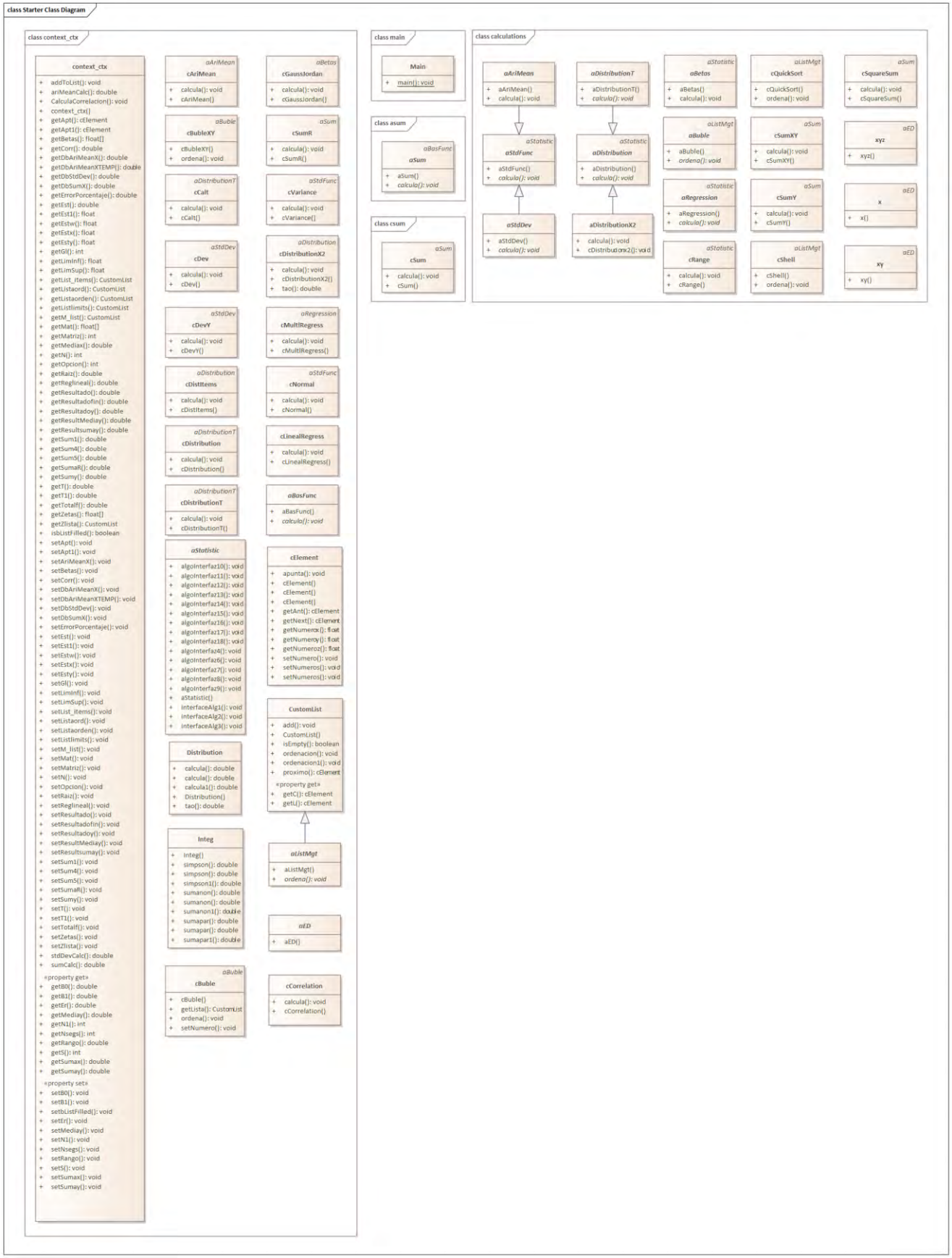

Figura 66 Arquitectura Proyecto Marco Estadístico Refactorizada

Comprobar que el comportamiento de la aplicación Marco Estadístico se mantiene después de la Refactorización.

Para comprobar que la aplicación Marco Estadístico mantiene el mismo comportamiento después de ser sometido al método de refactorización, se procede a agregar un usuario antes y después de la refactorización, esto con la finalidad de verificar el funcionamiento de la aplicación.

La Figura 67 muestra a los usuarios registrados en la aplicación Marco Estadístico antes de ser refactorizada, se puede observar que solo se cuenta con dos usuarios registrados.

Lista 1	Lista 2
10	4
56	22
26	8
37	15
44	78
82	45
36	29
65	74
71	19
6	10

Figura 67 Listas a ser evaluados por la aplicación Marco Estadístico después de la refactorización

En la Figura 68 se muestra el resultado de la suma al ejecutarse la aplicación Marco_Estadístico_Refactorizado.

```
Suma de la Lista 1: 433.0
Suma de la lista 2: 304.0
```

Figura 68 Resultado de la suma al ejecutarse la aplicación Marco_Estadístico _Refactorizado

Se puede comprobar que el funcionamiento de la aplicación Marco Estadístico Refactorizado sigue siendo el mismo después de ser sometida al método de refactorización, por lo que se concluye que dicho método no altera el comportamiento de la aplicación.

Comprobación en la disminución del número de Responsabilidades.

Para comprobar que el número de responsabilidades en la aplicación Marco Estadístico disminuyó, es sometida al cálculo de la métrica NR antes y después de la refactorización.

En la Tabla 32 se muestra la comparación de los valores de la métrica NR. Para llevar a cabo una interpretación adecuada, aplicamos la ecuación que se muestra en la Tabla 25, que consiste en sumar los valores (responsabilidades) obtenidos para cada módulo que fue

creado a partir de la arquitectura a refactorizar y mejorar y dividirlo entre el número total de estos.

Tabla 32 Comparación de valores de la Métrica NR antes y después de la Refactorización de la Aplicación Marco Estadístico

Módulo	Valor antes de la Refactorización	Módulo		Valor después de la Refactorización
calculations	0.03846154	context_ctx		0.083333336
		csum		1.0
		asum		1.0
TOTAL	0.03846154	**TOTAL**	$NR = \frac{\sum_{Pi=1}^{Pi=N} NR}{N}$	$NR = \frac{2.083333336}{3} = 0.69444$

En el Gráfico 2 se muestran los valores obtenidos para la métrica NR antes y después de la refactorización para la aplicación Marco Estadístico, se observa que para la arquitectura refactorizada hubo una disminución en el número de responsabilidades.

El número de responsabilidades para la arquitectura modular "calculations" mejoró de un 3.85% a 69.44%. Con lo cual se comprueba que el método de refactorización desarrollado en este trabajo de tesis disminuyó el número de responsabilidades en arquitecturas modulares escritas en lenguaje Java.

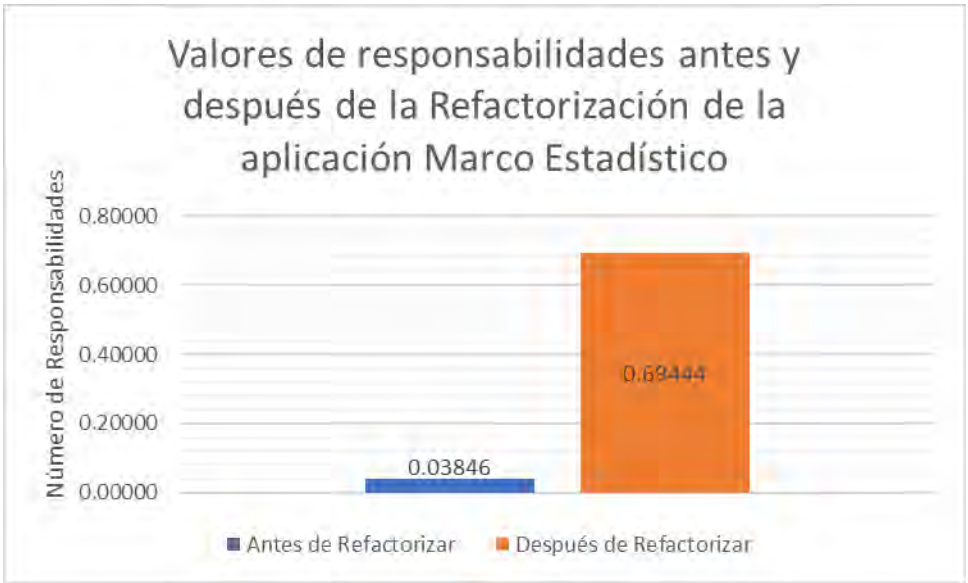

Gráfico 2 Responsabilidades Proyecto Marco_Estadístico antes y después de la refactorización

Comprobación de mejoría de Coherencia y Cohesión

En la Tabla 33 se muestran los valores obtenidos para el proyecto Marco_Estadístico antes y después de la refactorización, se puede observar que el valor obtenido de la métrica LCOM* (2) mejoró, debido a que la mayoría de los módulos de la aplicación

Marco_Estadístico después de la refactorización cuentan con una correcta división entre sus atributos y métodos.

Tabla 33 Valores obtenidos de la Métrica LCOM antes y después de la Refactorización.*

Cohesión - Proyecto Marco Estadístico			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
calculations	0.1316	context_ctx	0.2329
		csum	0.0
		asum	0.0
		Sumatoria	0.2329
Valor	**0.1316**	**Valor**	**0.0582**

En la Tabla 34 se muestran los valores obtenidos de la métrica CrCU (3) antes y después de la refactorización, dado que cada módulo después de la refactorización está conformado por una única responsabilidad, se concluye que cada uno de éstos atiende a una responsabilidad, por lo cual se obtiene el mejor resultado para cada uno de ellos.

En el caso del módulo context_ctx, la mejora es baja ya que dicho módulo presenta varias responsabilidades, pero estas son a nivel de clase.

Tabla 34 Valores obtenidos de la Métrica CrCU antes y después de la Refactorización.

Coherencia - Proyecto Marco Estadístico			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
calculations	0.152	context_ctx	0.2600
		csum	1.0
		asum	1.0
		Sumatoria	2.2600
Valor	**0.152**	**Valor**	**0.5650**

6.5.5 Caso de Prueba ISMRNR05-05

Artículos de Prueba: ProyectoAFND Autómata

En la Tabla 35 se enlistan las Características a probar del proceso del cálculo de métricas de calidad de Número de Responsabilidades en la aplicación ProyectoAFND.

Tabla 35 Características a probar del proceso de cálculo de Métrica de Número de Responsabilidades

Características a Probar
Conteo de puntos de entrada.
Cálculo de la métrica (NR) Número de Responsabilidades.

En la Figura 69 se muestra la carpeta que contiene los archivos que conforman el proyecto ProyectoAFND y en la Figura 70 se puede observar la arquitectura modular.

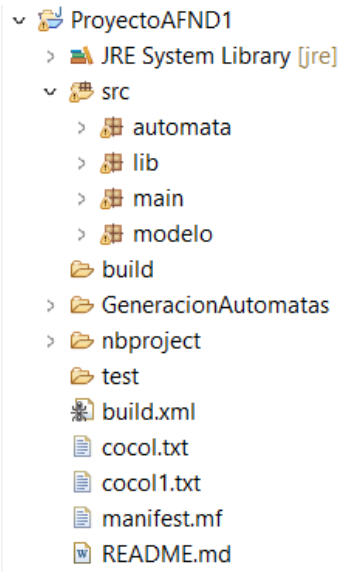

Figura 69 Carpeta Proyecto ProyectoAFND

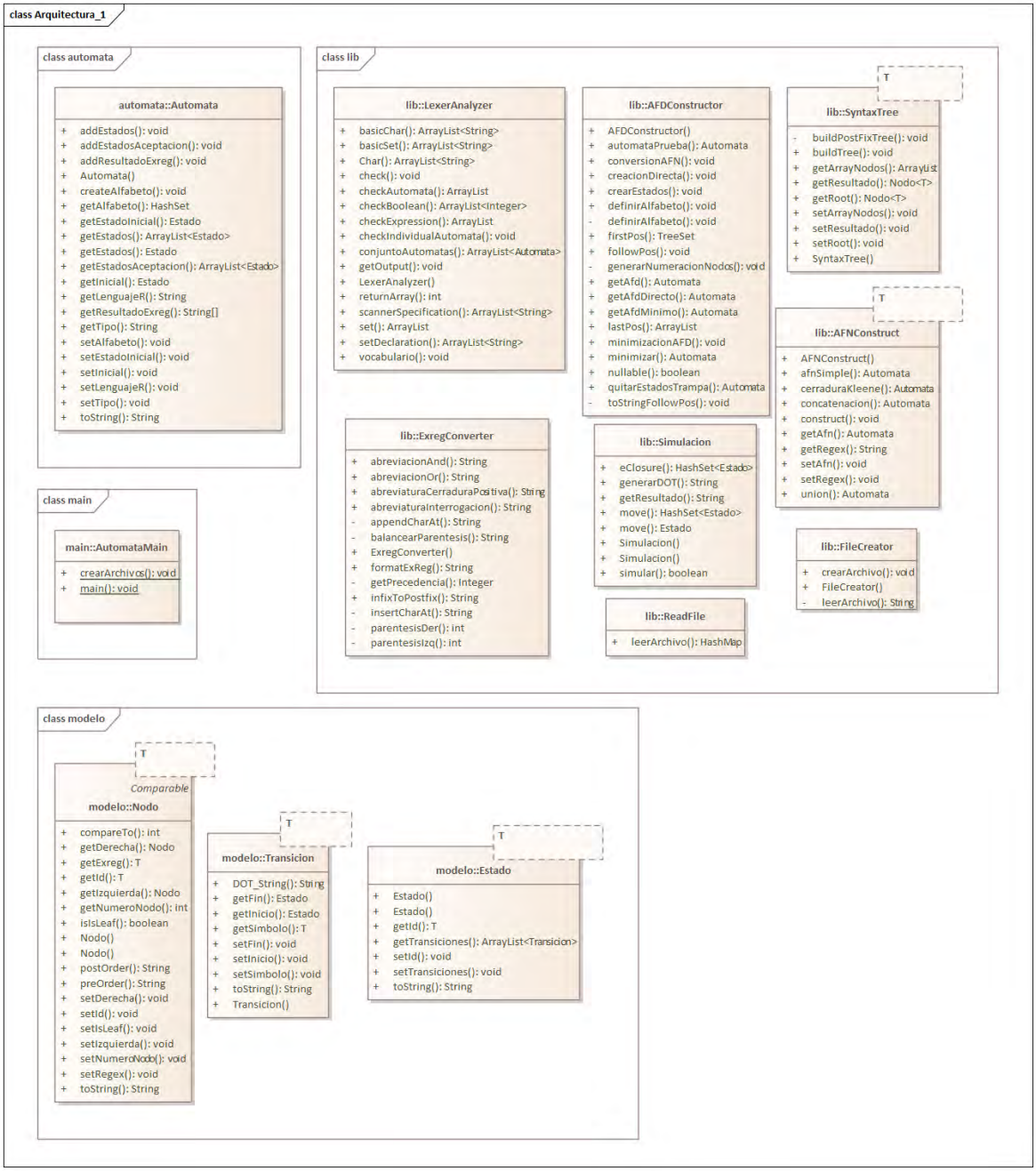

Figura 70 Arquitectura modular de ProyectoAFND

Conteo de puntos de Entrada.

Para verificar que el conteo de puntos de entrada se realiza de manera correcta, primero se realiza el conteo de forma manual en el módulo *"lib"* de la arquitectura modular de la aplicación ProyectoAFND, posteriormente se realiza un conteo de forma automática para la comprobación de ambos resultados.

En la Tabla 36 se puede observar que en el conteo manual y el conteo automático se obtuvieron los mismos valores, por lo que se comprueba que el conteo de puntos de entrada se realizó de manera correcta.

Tabla 36 Conteo manual y automática de los puntos de entrada

Conteo de forma manual el número de puntos de entrada	**Conteo de forma automática el número de puntos de entrada**
AFDConstructor.minimizar	[NR: Módulo a Refactorizar] ========== Responsabilidad de paquete : lib Número de entradas: 15 |Clase Entradas| |LexerAnalyzer.vocabulario 1 | |Simulacion.simular 1 | |Simulacion.generarDOT 1 | |FileCreator.creacion_Archivo_Prueba 1 | |AFNConstruct.construct 1 | |LexerAnalyzer.check 1 | |AFDConstructor.conversionAFN 1 | |AFDConstructor.getAfdDirecto 1 | |AFDConstructor.creacionDirecta 1 | |AFDConstructor.quitarEstadosTrampa 1 | |ReadFile.leerArchivo 1 | |AFNConstruct.getAfn 1 | |SyntaxTree.buildTree 1 | |AFDConstructor.getAfd 1 | |AFDConstructor.minimizar 1 | Responsabilidad: 0.06666667
AFDConstructor.getAfd	
AFDConstructor.quitarEstadosTrampa	
AFDConstructor.creacionDirecta	
AFDConstructor.conversionAFN	
AFDConstructor.getAfdDirecto	
SyntaxTree.buildTree	
AFNConstruct.getAfn	
AFNConstruct.construct	
Simulacion.generarDOT	
Simulacion.simular	
FileCreator.crearArchivo	
LexerAnalyzer.vocabulario	
LexerAnalyzer.check	
ReadFile.leerArchivo	
Número de entradas: 15	**Número de entradas:** 15

Cálculo correcto de métrica de Responsabilidades (NR)

Una vez hecho el conteo de los puntos de entrada, se procede a calcular la métrica de número de responsabilidades y para comprobar que se hace de manera correcta, primero se lleva a cabo de manera manual y posteriormente de manera automática, con la finalidad de comprobar que ambos resultados son iguales.

El cálculo de la métrica se realiza para el módulo: *"lib"*.

Cálculo de la métrica de Número de Responsabilidades (NR)

Expresión matemática:

$$NR = \frac{1}{\sum_{Pi=1}^{Pi=n} secuencia_i}$$

Donde:

NR = Número de responsabilidades

$\boldsymbol{P_i}$ = *"i-esimo"* punto de entrada.

Secuencia = Número total de puntos de entrada

En la Tabla 37 se puede observar que el valor de la métrica NR obtenido de forma manual y automática son iguales, por lo que se comprueba que el cálculo se lleva a cabo de forma correcta.

Tabla 37 Calculo manual y automático de la Métrica NR de la aplicación ProyectoAFND

Módulo	Calculo manual	Calculo automático
lib	$NR = \frac{1}{15} = 0.066667$	-------- Valores antes de la Refactorización -------- |Paquete Responsabilidad| |lib 0.06666667| |main 0.0 | |automata 0.1 | |modelo 0.5 |

Al ser un módulo que violenta el principio de única responsabilidad se procede a llevar a cabo la refactorización.

6.5.6 Caso de Prueba ISMRNR05-06

Artículos de Prueba: ProyectoAFND Autómata

En la Tabla 38 se enlistan las características a probar del proceso de refactorización en la aplicación ProyectoAFND Autómata.

Tabla 38 Características a probar del proceso de refactorización en la aplicación ProyectoAFND

Características a Probar
Separación de responsabilidades.
Comprobar que el comportamiento de la aplicación se mantiene después de la Refactorización.
Disminuir el número de responsabilidades en arquitecturas modulares.

Separación de responsabilidades en la Aplicación ProyectoAFND

Para comprobar que la separación de responsabilidades se realiza de manera correcta se analizaron las clases que pueden servir como punto de entrada hacia secuencia de métodos.

En la Figura 71 se muestra la arquitectura modular de la aplicación ProyectoAFND refactorizada, en donde se puede observar que cada responsabilidad es separa en un módulo diferente.

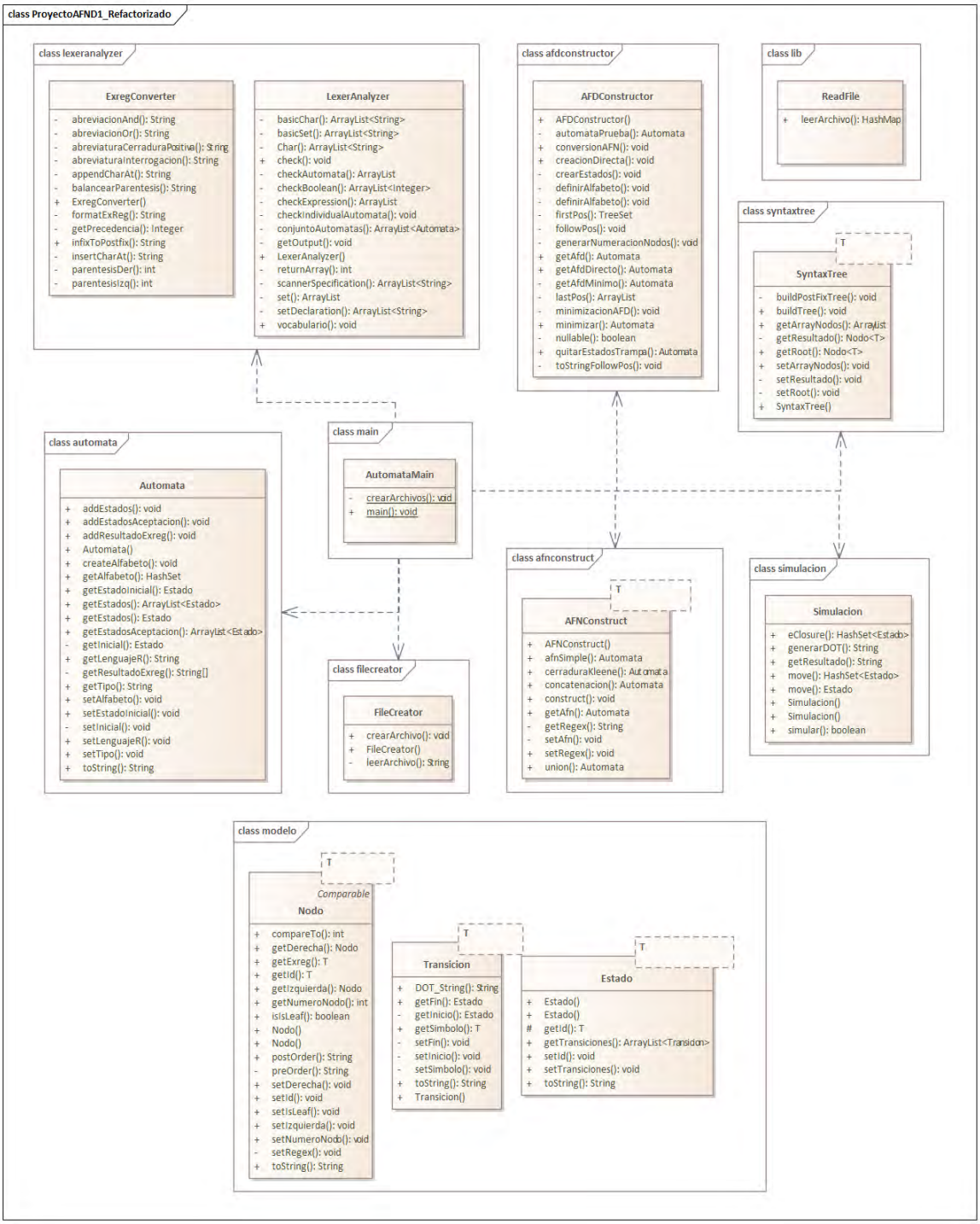

Figura 71 Arquitectura modular de ProyectoAFND Refactorizada

Comprobar que el comportamiento de la aplicación ProyectoAFND se mantiene después de la Refactorización.

Para comprobar que la aplicación ProyectoAFND mantiene el mismo comportamiento después de ser sometido al método de refactorización, se procede a realizar la tabla de transiciones de una expresión regular después de la refactorización, esto con la finalidad de verificar el funcionamiento de la aplicación.

La Figura 72 se muestra de lado izquierdo la expresión regular y de lado derecho la cadena a simular con el ProyectoAFND después de ser refactorizado.

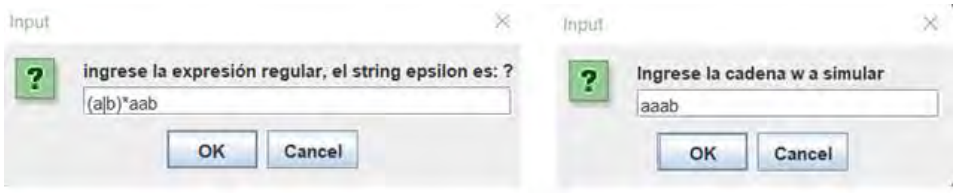

Figura 72 Expresión regular y cadena a simular con ProyectoAFND

En la Figura 73 se muestra el resultado de la expresión con su tabla de transiciones, así como su conjunto de estados y, conjunto de estados de aceptación al ejecutarse la aplicación ProyectoAFND_Refactorizado.

```
_____________________________________
Alfabeto: [a, b]
Estado inicial: 0
Conjutos de estados de aceptacion: [4]
Conjunto de Estados: [0, 1, 2, 3, 4]

Tabla de transiciones:
[(0-a-1), (0-b-2)]
[(1-a-3), (1-b-2)]
[(2-a-1), (2-b-2)]
[(3-a-3), (3-b-4)]
[(4-a-1), (4-b-2)]
_____________________________________
```

Figura 73Resultado de expresión regular al ejecutarse ProyectoAFND_Refactorizado

Se puede comprobar que el funcionamiento de la aplicación ProyectoAFND Refactorizado sigue siendo el mismo después de ser sometida al método de refactorización, por lo que se concluye que dicho método no altera el comportamiento de la aplicación.

Comprobación en la disminución del número de Responsabilidades.

Para comprobar que el número de responsabilidades en la aplicación ProyectoAFND disminuyó, es sometida al cálculo de la métrica NR antes y después de la refactorización.

En la Tabla 39 se muestra la comparación de los valores de la métrica NR. Para llevar a cabo una interpretación adecuada, aplicamos la ecuación mostrada en la Tabla 34, que consiste en sumar los valores (responsabilidades) obtenidos para cada módulo que fue creado a partir de la arquitectura a refactorizar y mejorada y dividirlo entre el número total de estos.

Tabla 39 Comparación de valores de la Métrica NR antes y después de la Refactorización de la Aplicación ProyectoAFND

Módulo	Valor antes de la Refactorización	Módulo		Valor después de la Refactorización
lib	0.066667	readfile		1.0
		syntaxtree		0.33333334
		simulacion		0.33333334
		filecreator		1.0
		lexeranalyzer		0.5
		afnconstruct		0.16666667
		afdconstructor		0.16666667
TOTAL	0.066667	**TOTAL**	$NR = \frac{\sum_{Pi=1}^{Pi=n} NR}{N}$	$NR = \frac{3.50000002}{7} = 0.50$

En el Gráfico 3 se muestran los valores obtenidos para la métrica NR antes y después de la refactorización para la aplicación ProyectoAFND, se observa que para la arquitectura refactorizada hubo una disminución en el número de responsabilidades.

El número de responsabilidades para la arquitectura modular "lib" mejoró de un 6.67% a 50%. Con lo cual se comprueba que el método de refactorización desarrollado en este trabajo de tesis disminuyó el número de responsabilidades en arquitecturas modulares escritas en lenguaje Java.

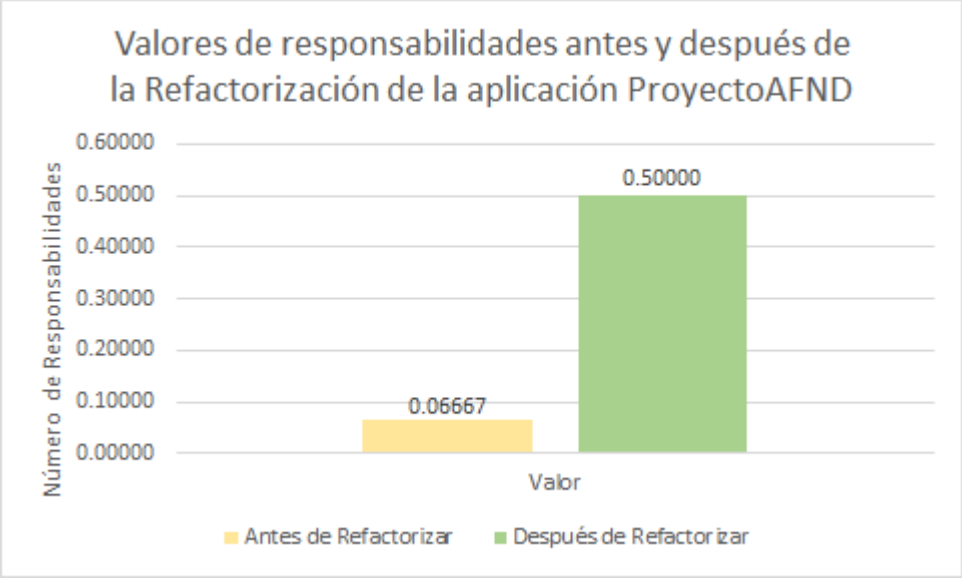

Gráfico 3 Responsabilidades ProyectoAFND antes y después de la refactorización

Comprobación de mejoría de Coherencia y Cohesión

En la Tabla 40 se muestran los valores obtenidos para el proyecto ProyectoAFND antes y después de la refactorización, se puede observar que el valor obtenido de la métrica LCOM* (2) mejoró un poco, debido a que no existe buena división entre atributos y métodos.

Tabla 40 Valores obtenidos de la Métrica LCOM antes y después de la Refactorización.*

Cohesión - ProyectoAFND1			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
lib	0.5963	afdconstructor	0.8353
		afnconstruct	0.5625
		filecreator	0.0
		lexeranalyzer	0.9472
		readfile	0.0
		simulacion	0.8000
		syntaxtree	0.6786
		Sumatoria	3.8236
Valor	**0.5963**	**Valor**	**0.5462**

En la Tabla 41 se muestran los valores obtenidos de la métrica CrCU (3) antes y después de la refactorización. En este caso solo 2 módulos lograron satisfacer una responsabilidad, esta arquitectura presenta varios módulos que sus clases cuentan con más de una responsabilidad, por lo cual algunos valores son bajos.

Tabla 41 Valores obtenidos de la Métrica CrCU antes y después de la Refactorización.

Coherencia - ProyectoAFND1			
Valores antes de la Refactorización		**Valores después de la Refactorización**	
Módulo	**Valor**	**Módulo**	**Valor**
lib	0.225	afdconstructor	0.379
		afnconstruct	0.666
		filecreator	1.0
		lexeranalyzer	0.600
		readfile	1.0
		simulacion	0.778
		syntaxtree	0.417
		Sumatoria	4.840
Valor	**0.225**	**Valor**	**0.6914**

CAPÍTULO 7.- CONCLUSIONES Y TRABAJOS FUTUROS

7.1 Conclusiones

Al contar con arquitecturas monolíticas, donde las aplicaciones poseen una única base de código, dichas aplicaciones pueden ofrecer una variedad de servicios. Una de las desventajas que presenta este enfoque es que, si se requiere realizar una modificación sobre alguna de las funcionalidades, tendrá un impacto sobre las demás funcionalidades del sistema, de modo que aumenta el costo de realizar dicha tarea.

Además, el software monolítico no es fácil de entender para un ingeniero de software, debido a su alto número de trayectorias de control, alcance de referencias, número de variables y por su complejidad general. Es por ello que se pretende la división del software en componentes más pequeños, llamados módulos, que se integran para satisfacer los requerimientos del problema.

Durante la realización de las pruebas de refactorización se pudo observar que en algunos módulos no se disminuyeron las responsabilidades, esto se explica porque el método de refactorización de esta tesis no reduce las responsabilidades a nivel de clase, sin embargo, las métricas si cuentan las responsabilidades en este nivel.

Un ejemplo de esto, se explica a continuación. En la Figura 74, se muestran los resultados obtenidos después de haber realizado la refactorización al ProyectoAFND, se observa que existen módulos que aún cuentan con más de una responsabilidad. Por ejemplo, el módulo "*lexeranalyzer"* contiene dos responsabilidades, las cuales son a nivel de clase.

Módulo	Valor antes de la Refactorización	Módulo		Valor después de la Refactorización
lib	0.066667	readfile		1.0
		syntaxtree		0.33333334
		simulacion		0.33333334
		filecreator		1.0
		lexeranalyzer		0.5
		afnconstruct		0.16666667
		afdconstructor		0.16666667
TOTAL	0.066667	**TOTAL**	$NR = \frac{\sum_{Pi=1}^{Pi=n} NR}{N}$	$NR = \frac{3.50000002}{7} = 0.50$

Figura 74 Responsabilidades a nivel de clase

Para mejorar este tipo de escenarios es importante aplicar en un orden secuencial los métodos de refactorización, para ello tendría que ejecutarse primero el *"Método para Reducir las Dependencias entre Clases"* (Ramírez Cruz, 2022), una vez que las responsabilidades de clase sean divididas en diferentes clases, aplicar el método de *"Separación de Responsabilidades"* a nivel de modulo, de lo que trata este trabajo.

Los métodos del SR2-Refactoring deben de ser aplicados en un orden secuencial de manera razonable, como se muestra en la Figura 75.

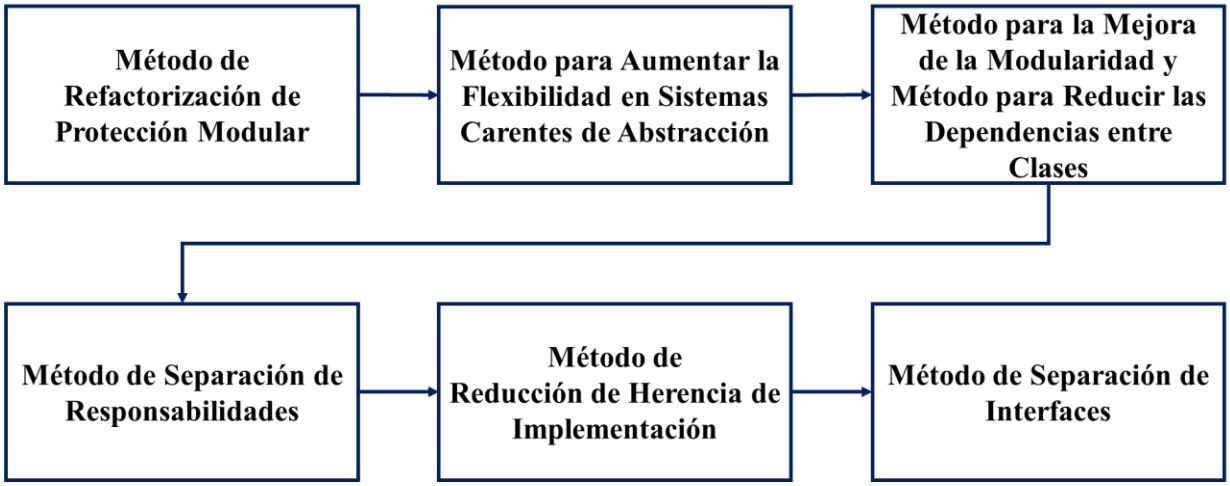

Figura 75 Secuencia de aplicación de los Métodos de Refactorización del SR2-Refactoring

El método de refactorización desarrollado extiende la funcionalidad del sistema SR2-Refactoring, con lo cual, se cumple con los objetivos específicos de esta tesis, los cuales son:

a) Reducir el código desagradable en arquitecturas modulares.
b) Aumentar los niveles de Coherencia.
c) Aumentar los niveles de Cohesión.

7.2 Aportaciones de la tesis

7.2.1 Método de refactorización de número de responsabilidades

- El método es capaz de disminuir las responsabilidades en arquitecturas modulares escritos en lenguaje Java, mediante la búsqueda de entradas a métodos públicos que son utilizados y que inicializan secuencias de métodos en dichas arquitecturas.
- Una vez que la arquitectura modular se encuentra refactorizada, se respetará el principio de "Abierto-Cerrado", así como también con el principio de "Única Responsabilidad", haciendo que su mantenimiento sea más fácil y sea reusable.

7.2.2 Métrica para medir el número de responsabilidades en arquitecturas modulares

- Hasta el termino de este trabajo de investigación no existe documentación en cuanto a cómo medir y evaluar las responsabilidades en arquitecturas modulares, se definió la siguiente métrica:
    1 NR (Número de Responsabilidades).

7.3 Trabajos a Futuro

7.3.1 Evaluar arquitecturas que cuenten con código repetido después de realizar la refactorización a escenarios que cuenten con dependencia entre módulos.

Actualmente el método de refactorización no cuenta con un método que compruebe si existe código repetido después de la refactorización en arquitecturas modulares que violenten el principio de *"No-Repetición"*, en el capítulo de Hallazgos se describe a detalle este tipo de trabajo y como poder llegar a solucionarlo.

7.4 Observaciones

7.4.1 Principio DRY (Don't Repeat Yourself)

Al llevar a cabo el análisis de arquitecturas como la mostrada en la *"Figura 29 Escenario 3. Dependencia entre Módulos",* nos encontramos con problemas que relacionan el principio "DRY (Don't Repeat Yourself)" el cual nos dice que; una pieza de código debe existir exactamente en un lugar. Pero de cierto modo se estará cumpliendo con la finalidad de este tema de investigación, el cual consiste en mantener módulos con una única responsabilidad.

Una alternativa para eliminar el problema existente con el principio de "No-Repetición" sería mediante la creación de un módulo que contenga la(s) clase(s) que son usadas por otras clases pertenecientes a módulos diferentes, tal y como se muestra en la Figura 76.

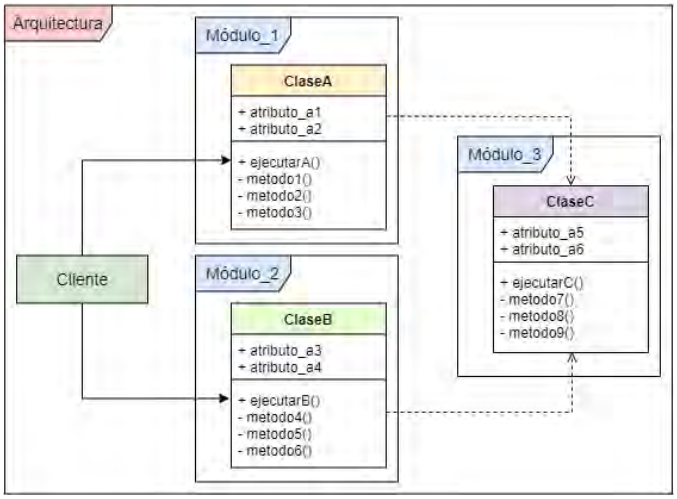

Figura 76 Alternativa Escenario 3. Dependencia entre Clases

Dada la lingüística de programación de cada desarrollador, cada uno puede elegir que ganar o que perder en el sentido de que, en cuál de los dos principios quiere ganar. Para efectos de este trabajo, se decide ganar en el principio de única responsabilidad, que es el problema que se está abordando.

7.4.2 Herencia

Otros escenarios analizados y que de momento son complicados de llevar a cabo por el método de refactorización, es donde existe herencia. La Figura 77 sirve para representar una arquitectura donde existe Herencia, así como también para exponer los puntos importantes para este tipo de escenarios.

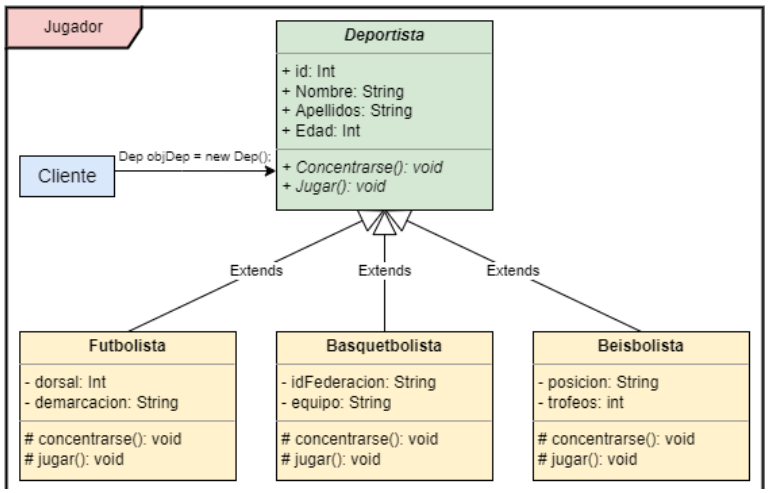

Figura 77 Escenario Herencia

1. Java no permite que la Clase Padre *"Deportista"* defina métodos abstractos *"públicos"* y que las clases hijas reduzcan su visibilidad a "*protected"*. Se puede apreciar en la Figura 78, que el compilador marca un error de nivel de visibilidad en los modificadores de alcance.

```java
package jugador;

public abstract class Deportista {

    public int id;
    public String nombre;
    public String apellidos;
    public int edad;

    public abstract void jugar();
    public abstract void concentrarse();
}
```

```java
package jugador;

public class Futbolista extends Deportista {

    private int dorsal;
    private String demarcacion;

    @Override
    protected void concentrarse() {
        System.out.println("Futbolista se concentra");
    }

    @Override
    protected void jugar() {
        System.out.println("Futbolista juega");
    }
}
```

Figura 78 Clase Padre "Deportista" y *Clase Hija "Futbolista"*

2. Como alternativa posibles es que la Clase Padre *"Deportista"* defina los métodos abstractos como *"protected"* y las Clases Hijas aumenten la visibilidad a *"public"*, como se muestar en la Figura 79.

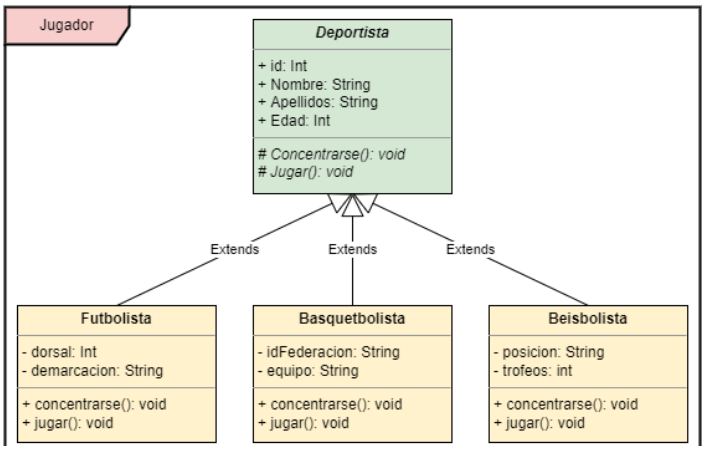

Figura 79 Arquitectura con cambio en los modificadores de acceso

El problema de contar con una arquitectura de esta manera implica que los clientes puedan acceder desde las Clases Hijas, en lugar de acceder por la Clase Padre, ya que serán las clases que contendrán los métodos públicos, tal como se muestra en la Figura 80.

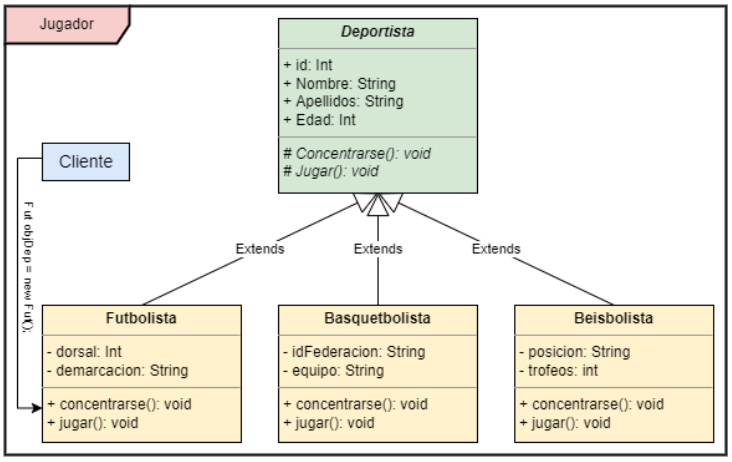

Figura 80 Alternativa posible

El problema de este tipo de Arquitecturas es que deshabilita la abstracción y por lo tanto el código se vuelve rígido.

ANEXO A.- SUSTENTO DE LA MÉTRICA NR COMO ESCALA ORDINAL

Número de Responsabilidades (NR) como escala ordinal

Se tiene el siguiente sistema relacional empírico:

1. P es el conjunto de módulos de programa.
2. $\bullet\geq$ es una relación empírica entre módulos, la cual describe si tiene mayor o igual número de responsabilidades.
3. $\Re$ denota el conjunto de los números reales.
4. $\geq$ *"mayor o igual que"* es una relación binaria entre números.

Entonces $((P,\bullet\geq),(\Re,\geq),NR)$ es una escala ordinal si, y solo si, se cumplen las siguientes condiciones:

1. La relación binaria $\bullet\geq$, es de *orden débil*
2. P1 $\bullet\geq$ P2 $\Leftrightarrow$ NR(Módulo1) $\geq$ NR(Módulo2).

Comprobación

Como parte del proceso de comprobación de las condiciones dadas anteriormente, se utiliza la ecuación (1), para realizar el cálculo de NR de las arquitecturas modulares 1, 2 y 3, mostradas en las Figuras 81, 82 y 83 respectivamente. Donde el valor deseable es 1 y en el peor de los casos cercano a 0.

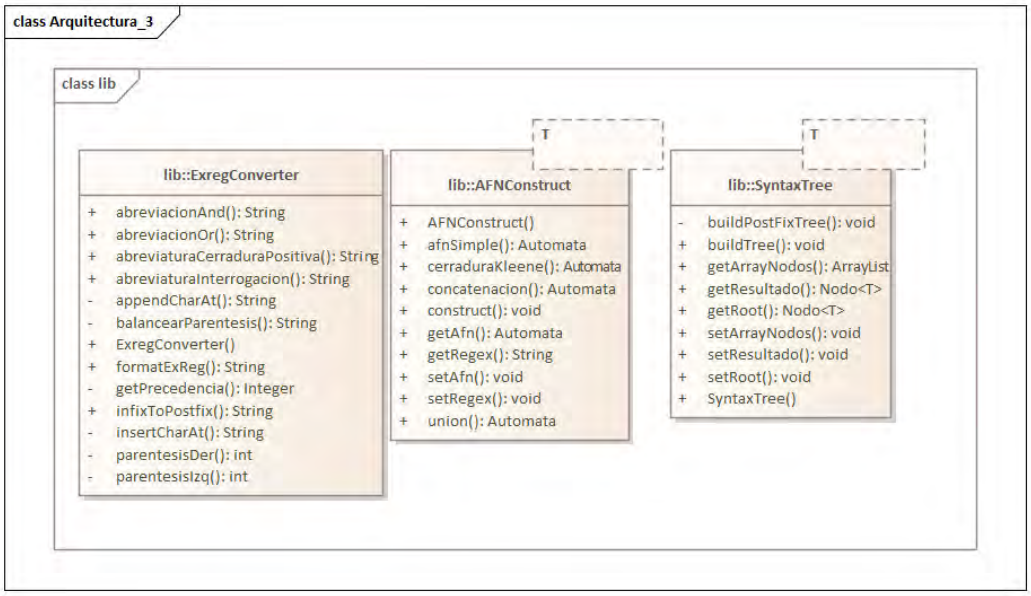

Figura 81 Arquitectura modular 1.

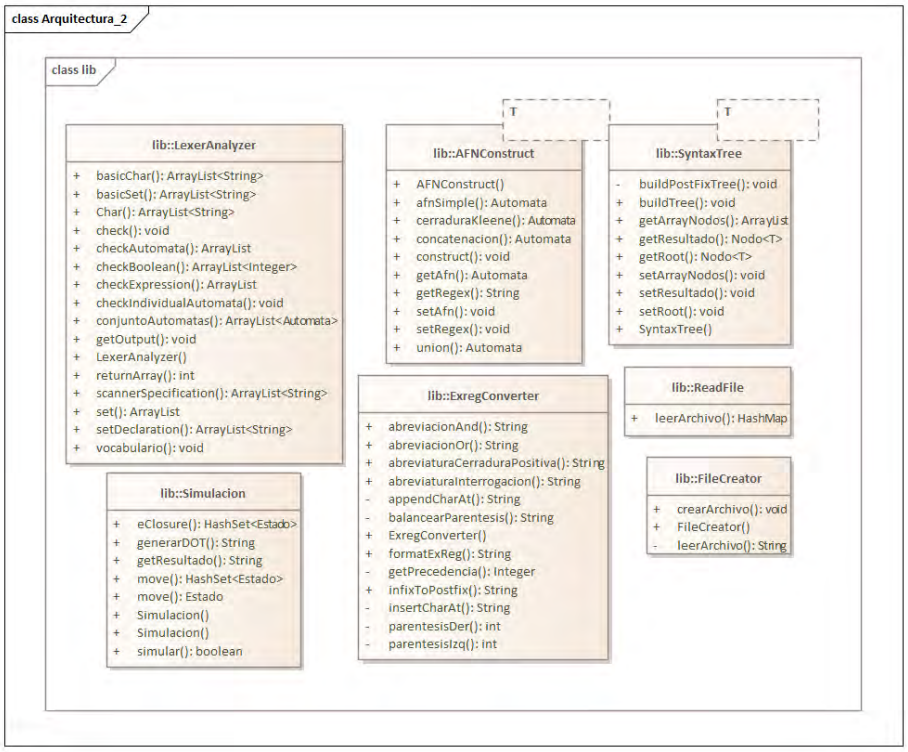

Figura 82 Arquitectura modular 2.

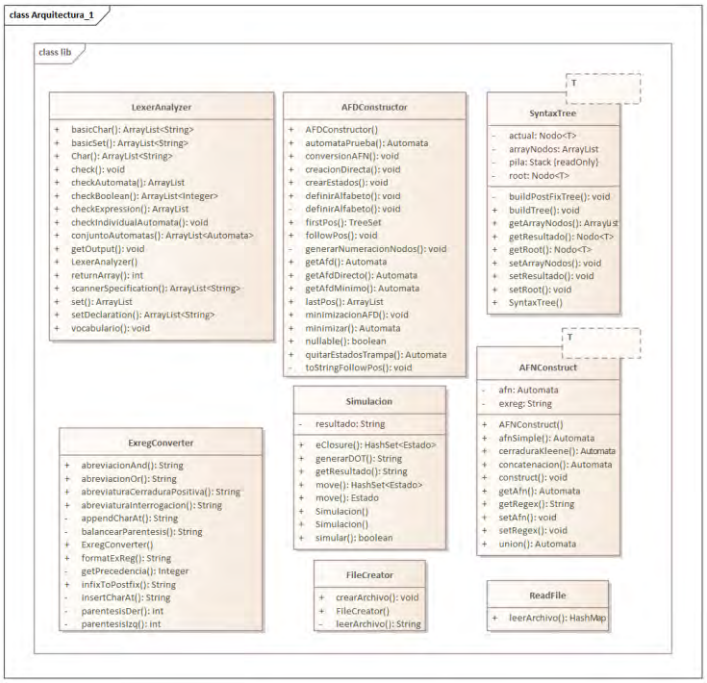

Figura 83 Arquitectura modular 3.

Sustituyendo los valores en la métrica se obtienen los resultados mostrados en la Tabla 42.

Tabla 42 Resultados manuales y automáticos de la métrica NR de las arquitecturas 1, 2 y 3.

	Resultados Manuales	Resultados obtenidos implementando la métrica NR
Arquitectura 1	$NR = \frac{1}{3} = 0.3333$	========== Responsabilidad de paquete : lib Número de entradas: 3 Responsabilidad: 0.33333334
Arquitectura 2	$NR = \frac{1}{10} = 0.1$	========== Responsabilidad de paquete : lib Número de entradas: 10 Responsabilidad: 0.1
Arquitectura 3	$NR = \frac{1}{16} = 0.0625$	========== Responsabilidad de paquete : lib Número de entradas: 16 Responsabilidad: 0.0625

Transitividad:

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

$$\Leftrightarrow$$

$$equivalente\ a:$$

$$NR(Arquitectura\ 1) \bullet\geq \mathrm{NR}(Arquitectura\ 2)\ \&\ NR(Arquitectura\ 2) \bullet\geq \mathrm{NR}(Arquitectura3) \rightarrow NR(Arquitectura\ 1) \bullet\geq \mathrm{NR}(Arquitectura\ 3)$$

Comprobación formal es una relación binaria transitiva

Reflejado en números, se tiene que:

$$0.3333 \geq 0.1\ \&\ 0.1 \geq 0.0625 \rightarrow 0.3333\ \geq 0.0625$$

Se puede observar, que la arquitectura 1 mejoró el número de responsabilidades en comparación a la arquitectura 2. Mientras que esta, a su vez presenta una mejoría en el número de responsabilidades que la arquitectura 3. Se puede concluir que la métrica NR es una relación binaria y cumple con la propiedad de transitividad.

Relación Total:

Si se tiene la arquitectura 1 y 2, se debe poder decir que la arquitectura *"tiene mayor o igual mejoría de número de responsabilidades que"* la arquitectura 2, o viceversa. Es Decir:

$$\text{Arquitectura 1} \bullet\geq \text{Arquitectura 2}\ o\ \text{Arquitectura 2} \bullet\geq \text{Arquitectura 1}$$

Comprobación formal es una relación binaria completa

Sustituyendo los valores obtenidos aplicando la métrica NR a las arquitecturas 1 y 2, obtenemos el siguiente resultado: $0.3333 \geq 0.1$.

Dado que la arquitectura 1 tiene un valor de NR = 0.3333 y la arquitectura 2 un valor de NR = 0.1, se tiene que NR (Arquitectura 1) ≥ NR (Arquitectura 2), por lo tanto, se concluye que la métrica NR es una relación binaria completa.

Homomorfismo:

Comprobando la segunda condición quedaría de la siguiente manera:

$$\text{Arquitectura 1} \bullet\geq \text{Arquitectura 2} \Leftrightarrow \text{NR(Arquitectura 1)} \geq \text{NR(Arquitectura 2)}$$

Comprobación formal es un homomorfismo

La Arquitectura 1 $\bullet\geq$ la Arquitectura 2

$\Leftrightarrow$

$equivalente\ a$:

$$0.3333 \geq 0.1$$

Del resultado se puede observar que al aplicar los valores de la métrica NR a los módulos de programa, el sistema empírico tiene equivalencia al sistema formal.

La métrica esta normalizada para que a medida que el valor de esta tienda a 0 indicaría un mayor número de responsabilidades, y cuando el valor de ésta sea 1 indicaría una única responsabilidad en el módulo.

Conclusión

La relación binaria *"tiene mayor o igual mejoría de número de responsabilidades que"* de la métrica NR, cumple con las condiciones de orden débil y además es un homomorfismo. Por lo cual se concluye que la métrica es de escala ordinal.

REFERENCIAS

Barón Pérez, N. (2019). *Método de refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Briand, L., Emam, K. E., & Morasca, S. (1996). On the Application of Measurement Theory in Software Engineering. *Empirical Software Engineering*, *1*(1), 61–88.

Bryton, S., & Brito e Abreu, F. (2008). *Modularity-Oriented Refactoring*. 294–297.

C. Martin, R. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson.

C. Martin, R., & Martin, M. (2006). *Agile Principles, Patterns, and Practices in C#* (1st Edition). Prentice Hall.

Cunningham, W. (1992). *The WyCash Portfolio Management System*. 29–30.

Fokaefs, M.-E., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2011). *JDeodorant: Identification and application of extract class refactorings*. 1037–1039.

Fokaefs, M.-E., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, *85*(10), 2241–2260.

Fonseca, B., Ribeiro, M., Silva, V., Braga, C., Lucena, C., & Costa, E. (2015). AutoRefactoring: A platform to build refactoring agents. *Expert Systems with Applications*, *42*(3), 1652–1664.

Fowler, M., Beck, K., Brant, J., Opdyke's, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code* (1st Edition). Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

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

Hamilton, K., & Miles, R. (2006). *Learning UML 2.0: A Pragmatic Introduction to UML* (1st Edition). O´Reilly Media, Inc.

Holvitie, J., A. Licorish, S., O. Spínola, R., Hyrynsalmi, S., G. MacDonell, S., S. Mendes, T., Buchan, J., & Leppänen, V. (2018). Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey. *Information and Software Technology*, *96*, 141–160.

J. Allen, M., & M. Yen, W. (1979). *Introduction to Measurement Theory* (1st Edition). Brooks/Cole Publishing Company.

John Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2nd Edition). The Pragmatic Bookshelf.

Joyanes Aguilar, L., & Zahonero Martínez, I. (2007). *Estructura de Datos en Java* (Primera Edición). McGraw Hill.

Khatchadourian, R., & Masuhara, H. (2017). *Automated Refactoring of Legacy Java Software to Default Methods*. 83–93.

Kruchten, P., Ozkaya, I., & L. Nord, R. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Computer Society Press*, *29*(6), 18–21.

Lorge Parnas, D. (1972). A Technique for Software Module Specification with Examples. *Communications of the ACM*, *15*(5), 330–336.

Marcos, C., Vidal, S., & Díaz-Pace, J. A. (2018). Análisis de dependencias entre Refactorings para solucionar Code Smells. *Revista Tecnología Y Ciencia*, *33*, 1–22.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd Edition). Prentice Hall.

Morelli, R., & Walde, R. (2016). *Java, Java, Java: Object-Oriented Problem Solving* (Third Edition). Prentice Hall.

Napoli, C., Pappalardo, G., & Tramontana, E. (2013). *Using Modularity Metrics to assist Move Method Refactoring of Large Systems*. 529–534.

Ortiz Gutiérrez, O. (2020). *Refactorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientas a objetos.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Padilla Salgado, P. (2019). *Método de Refactorización de código java con interfaces y abstracciones incorrectas.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Pressman, R. S. (2010). *Ingeniería de Software: Un enfoque práctico* (Séptima Edición). McGraw Hill.

Ramírez Cruz, M. (2022). *Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos*. CENIDET.

Robledo Cárdenas, L. A. (2014). *Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Santos Castillo, L. E. (2005). *Adaptación de Interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Sarnath, R., & Brahma, D. (2015). *Object-Oriented Analysis, Design and Implementation: An Integrated Approach* (Second Edition). Springer.

Shatnawi, R., & Li, W. (2011). An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model. *International Journal of Software Engineering and Its Applications*, *5*(4), 24.

Stevens, S. S. (1946). On the Theory of Scales of Measurement. *American Association for the Advancement of Science*, *103*(2684), 677–680.

Valdés Marrero, M. A. (2004). *Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces.* [Tesis de Maestría en Ciencias Computacionales]. CENIDET.

Velarde de Barraza, O., Murillo de Velásquez, M., Gómez de Meléndez, L., & Castillo de Krol, F. (2006). *Introducción a la Programación Orientada a Objetos* (Primera Edición). Prentice Hall.

Vidal, S., Vazquez, H., Díaz-Pace, J. A., Marcos, C., Garcia, A., & Oizumi, W. (2015). *JSpIRIT: a flexible tool for the analysis of code smells*. 1–6.

Yourdon, E., & Constantine, L. L. (1978). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Second Edition). Yourdon Press.

Zuse, H., & Bollmann-Sdorra, P. (1992). Measurement Theory and Software Measures. En T. Denvir, R. Herman, & R. W. Whitty (Eds.), *Formal Aspects of Measurement* (pp. 219–259). Springer London.