EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Re-factorización de Módulos Colaborativos con Carencia de Abstracción.

presentada por

Ing. Fernando Sánchez Rogel

como requisito para la obtención del grado de

Maestro en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Octubre de 2023.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

SEP TecNM CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
RECIBIDO
27 SEP 2023
SUBDIRECCIÓN ACADÉMICA

Cuernavaca, Mor., 18/septiembre/2023

OFICIO No. DCC/ 176 /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 FERNANDO SÁNCHEZ ROGEL con número de control M20CE088, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"RE-FACTORIZACIÓN DE MÓDULOS COLABORATIVOS CON CARENCIA DE ABSTRACCIÓN"** 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
Revisor 1

HUMBERTO HERNÁNDEZ GARCÍA
Revisor 2

EDUCACIÓN
27 SEP 2023
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SERVICIOS ESCOLARES
RECIBIDO

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

Cuernavaca, Mor., 27/septiembre/2023
No. De Oficio: SAC/157/2023
Asunto: Autorización de impresión de tesis

FERNANDO SÁNCHEZ ROGEL
CANDIDATO AL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"RE-FACTORIZACIÓN DE MÓDULOS COLABORATIVOS CON CARENCIA DE ABSTRACCIÓN"**, 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"

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

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

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

CMAZ/LMZ

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO
28 SEP 2023
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SERVICIOS ESCOLARES
RECIBIDO

DEDICATORIAS

Esta tesis está dedicada a:

A mis padres que cada día me alentaron a seguir adelante, porque siempre me llevaron en sus oraciones y jamás me dejaron solo, gracias doy a Dios por sus vidas.

A mis hermanos, por ser mi ejemplo a seguir y por siempre apoyarme en todo, a Leonel mi cuñado por todos sus consejos y palabras de aliento, a mis sobrinos por ser siempre un soporte en mi vida.

A Valeria y su familia por todo el apoyo brindado durante este proceso y por sus palabras de ánimo que me dieron mucha fuerza.

A mis amigos por apoyarme cuando más les necesité, por extenderme su mano para aquellos retos difíciles que se presentaron en este camino y por todo su cariño.

AGRADECIMIENTOS

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

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.

A mi director de tesis, Dr. René Santaolaya Salgado por su excelencia al orientarme sabiamente en este camino de la investigación, con toda su experiencia científica para que este trabajo de tesis llegará a su culminación.

A mis revisores, Dra. Blanca Dina Valenzuela Robles y M.C. Humberto Hernández García por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

A Dios gracias por nunca dejarme solo y siempre estar a mi lado dándome la sabiduría y entendimiento para poder comprender cada uno de los retos en el proceso de mis estudios de la maestría.

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

A mis padres y hermanos que son pilares de mi vida, por siempre estar a mi lado en cada momento enseñándome a ser mejor cada día.

A Valeria y su familia por darme el apoyo durante este proceso de estudio de maestría, por sus consejos y oraciones.

A mis compañeros y amigos que estuvieron conmigo en todo el recorrido de mi carrera, por la convivencia y confianza que tuvimos, Fernando, Rebecca, Elías, Enrique, Merino, Andros, Diego, Juan Antonio, Nélida y Valentín.

RESUMEN

El paradigma de programación orientada a objetos es un modelo o estilo de programación que ofrece pautas sobre cómo debemos ~~de~~ trabajar con base en propiedades como son la herencia, abstracción y polimorfismo, para estructurar programas de software en piezas reusables simples, con la finalidad de crear software de calidad. Dicho paradigma, se apoya en el uso de los principios de diseño de software, como son los principios de "*Abierto-Cerrado*" e "*Inversión de Dependencias*".

Un sistema de software que no hace uso de elementos abstractos como es la clase abstracta o la interfaz dentro de su funcionamiento indica que carece de abstracción. Esta carencia inhabilita el polimorfismo, en consecuencia, la arquitectura presenta rigidez a cambios de comportamiento funcional, lo que indica que se encuentra atada a comportamientos específicos. Por otra parte, al no haber polimorfismo no es posible que una arquitectura de clases de objetos se extienda a nuevas funcionalidades o comportamientos. Tan importante es la abstracción que a través de esta propiedad es como aplicamos los principios fundamentales del diseño orientado a objetos, como lo son: el principio de abierto-cerrado y el principio de inversión de dependencias en arquitecturas de software. A través de esta propiedad se habilita el polimorfismo, logrando que las arquitecturas de clases sean flexibles a cambios de comportamiento en tiempo de ejecución y es posible extender a nuevas funcionalidades y comportamientos. La abstracción es una manera para cambiar diseños arquitecturales frágiles, rígidos e inamovibles en diseños robustos, flexibles y reusables, de reducir la complejidad y permite diseños e implementación más eficiente en sistemas de software complejos.

Con la finalidad de obtener diseños robustos, flexibles y reusables de arquitecturas de software carentes de la propiedad de abstracción, en este trabajo se planteó el "Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción". El método fue implementado en lenguaje Java para automatizar el proceso de refactorización. Se aplican las métricas de Abstracción y Factor de Polimorfismo para evaluar el grado de abstracción y de las situaciones polimórficas que contiene un proyecto o sistema de software existente, en uso. En la etapa de análisis del problema se identificaron diferentes escenarios para los cuáles opera el método de refactorización en este trabajo de investigación. Este método de refactorización tiene como objetivo principal identificar la carencia de abstracción en el diseño de arquitecturas modulares de software mediante un análisis estático de abajo hacia arriba (del código escrito en lenguaje Java hacia el diseño), con el fin de cambiar clases base y clases abstractas por *interfaces. Este* método de

refactorización agrega capacidad, complementa y amplia al *“Método de refactorización de arquitecturas de software con carencia de abstracciones”* .

ABSTRACT

The object-oriented programming paradigm is a programming model or style that offers guidelines on how we should work based on properties such as inheritance, abstraction and polymorphism, to structure software programs in simple reusable pieces, in order to create quality software. This paradigm is supported by the use of software design principles, such as "Open-Closed" and "Dependency Inversion".

A software system that does not make use of abstract elements such as the abstract class or interface in its operation indicates that it lacks abstraction. This lack disables polymorphism, consequently, the architecture presents rigidity to changes in functional behavior, which indicates that it is tied to specific behaviors. On the other hand, in the absence of polymorphism, it is not possible for an object class architecture to extend to new functionalities or behaviors. Abstraction is so important that it is through this property that we apply the fundamental principles of object-oriented design, such as: the open-closed principle and the principle of dependency inversion in software architectures. Through this property, polymorphism is enabled, making class architectures flexible to behavior changes at runtime and it is possible to extend to new functionalities and behaviors. Abstraction is a way to change fragile, rigid and immovable architectural designs into robust, flexible and reusable designs, reducing complexity and allowing more efficient design and implementation in complex software systems.

In order to obtain robust, flexible and reusable designs of software architectures lacking the abstraction property, in this work the "Method of Refactoring Collaborative Modules with Lack of Abstraction" was proposed. The method was implemented in Java language to automate the refactoring process. Abstraction and Polymorphism Factor metrics are applied to evaluate the degree of abstraction and polymorphic situations contained in an existing software project or system in use. In the problem analysis stage, different scenarios were identified for which the refactoring method operates in this research work. The main objective of this refactoring method is to identify the lack of abstraction in the design of modular software architectures through a bottom-up static analysis (from the code written in Java language to the design), in order to change base classes and abstract classes by interfaces. This refactoring method adds capability, complements and extends the "Method of refactoring software architectures with lack of abstractions".

Contenido

Índice de Figuras.

Índice de Tablas.

1.- Introducción

La refactorización no solamente es la intención de mejorar las cosas desde un punto de vista estético o perfeccionista, más bien es una estrategia bien definida, con un conjunto de técnicas enfocadas en aspectos donde el código fuente de proyectos de software pueden mejorar. Hacer uso de estas técnicas de refactorización puede generar código mejor diseñado, mejor modularizado, flexible, fácil de entender, y por lo tanto con menor necesidad de mantenimiento, lo que previene la fragilidad del software y se gana en movilidad.

El desarrollo de software que es orientado a objetos demanda en el ser humano una gran capacidad de imaginación, abstracción y creatividad, con la finalidad de analizar y plantear buenas soluciones a problemas informáticos que sustentados en el desarrollo de sistemas de software. Estas capacidades en algunas ocasiones para el ser humano son difíciles de alcanzar.

Cuando un desarrollador de software carece de experiencia y habilidades en el desarrollo de sistemas de software, suelen producirse sistemas o unidades de programa que técnicamente quedan a deber, lo cual se conoce como “deuda técnica”.

La deuda técnica acumulada que se produce debido a la falta de habilidades, experiencias, presiones de tiempo y costo en el desarrollo de sistemas de software y unidades de programa, conducirá a malas prácticas y decisiones de diseño incorrectas que, finalmente, derivan en la producción de código desagradable (Smell Code), el cual tiene efectos colaterales en la modularidad, fragilidad, flexibilidad, extensibilidad del software, lo cual impide su reuso, dificulta su mantenimiento y encarece la producción de software y el control de su evolución en el tiempo.

Una situación común que presentan los desarrolladores es la generación de deuda técnica y código desagradable en el software legado causado por la “carencia de abstracción”. El código desagradable por esta carencia se presenta cuando clases cliente solicitan servicios (a través de mensajes dirigidos a clases de objetos) directamente a clases servidoras concretas o específicas, en donde, en estas arquitecturas no hay una relación de herencia de la clase concreta hacia una clase abstracta o interfaz, por lo tanto, no puede haber asociación dinámica de tipos ni polimorfismo. Arquitecturas que carecen de la propiedad de abstracción son “rígidas”, esto es que no pueden cambiar dinámicamente su comportamiento funcional en tiempo de ejecución; así mismo, no son “extensibles” a nuevos comportamientos por nuevas funcionalidades requeridas. La única manera de cambiar o extender el comportamiento de una clase servidora específica, es adaptar

esta clase a nuevos usos, mediante la edición del código original, rompiendo así al principio fundamental de diseño enunciado como “principio de abierto - cerrado”, lo cual puede causar problemas de “fragilidad” funcional de los sistemas de software.

Una forma viable de corregir estas arquitecturas es aplicar el principio de diseño llamado "principio de inversión de dependencia".

La inversión de dependencias de módulos de alto nivel con respecto a los módulos de bajo nivel, en conjunto con la propiedad de polimorfismo promueven la “flexibilidad” a múltiples comportamientos funcionales, pudiendo cambiar el comportamiento actual o bien agregar nuevos comportamientos, sin necesidad de violentar el “principio de abierto-cerrado”. Así se corrigen los problemas de “rigidez” y “no-extensibilidad”.

El proyecto desarrollado en esta tesis aporta una extensión al “Método de refactorización de arquitecturas de software con carencia de abstracciones” desarrollado dentro del Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET). El método aplica el “principio de inversión de dependencias”, para eliminar código desagradable que impide la flexibilidad de múltiples comportamientos funcionales y la extensibilidad de nuevos comportamientos funcionales, por la interacción funcional en arquitecturas de software compuestas por módulos colaborativos.

La implementación del método de refactorización que se propone en este trabajo de tesis aporta mayor capacidad al “Método de refactorización de arquitecturas de software con carencia de abstracciones”, para limpiar de código desagradable de sistemas existentes de software en uso que potencialmente podrían presentar múltiples cambios y adecuaciones frecuentes e incrementar el riesgo de problemas en el software. Con este método se mejoran las propiedades genéricas para reuso que repercute en una disminución en el costo de producción de nuevos sistemas de software al incorporar al software legado una estructura que mejora su reuso y simplifica el protocolo interactivo entre objetos.

2.- Antecedentes

2.1.- Planteamiento del problema

El problema que trata este trabajo de tesis radica en que el diseño de unidades de programa del software legado que no son conformes con los principios de “abierto - cerrado” e “inversión de dependencias”, por lo que exhiben problemas de rigidez, fragilidad y no extensibilidad.

Para una mejor comprensión de este problema, en la Figura 1 se muestra el ejemplo en donde se visualiza una arquitectura carente de abstracción que está atada a comportamientos específicos, impide el polimorfismo, y por lo tanto rígida a cambios de comportamiento, y se impide se extensibilidad a nuevos comportamientos funcionales.

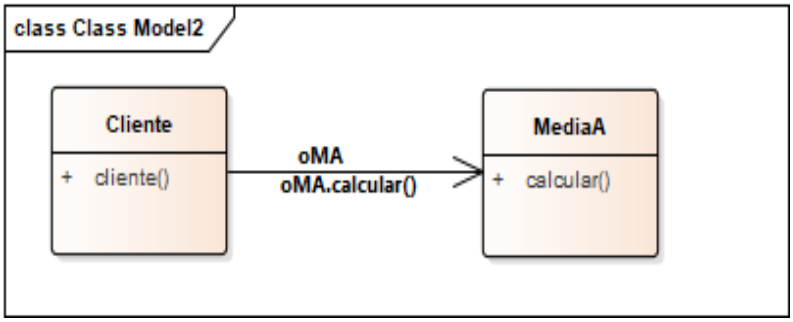


Figura 1 Arquitectura de software con Carencia de Abstracción.

Está carencia de abstracción, previene cumplir con las mejores especificaciones de calidad para la satisfacción con completitud y suficiencia a las demandas funcionales requeridas por nuevas aplicaciones desarrolladas a partir de éstas, por mecanismos de integración, ensamble y/o composición; así como las demandas de calidad para su mantenimiento y el control de su evolución.

2.2.- Solución Propuesta

Las metas u objetivos de unidades reusables de programa serán el uso flexible de las funciones entre clases, el nivel de abstracción y polimorfismo de módulos de programa, y la satisfacción del principio abierto-cerrado y de inversión de dependencias.

A partir de un sistema de análisis estático de abajo hacia arriba (del código escrito en lenguaje Java hacia el diseño), que estime el problema de carencia de abstracción y polimorfismo de clases de objetos mediante el uso de métricas para tal efecto; y un método de refactorización de software legado o marcos de aplicaciones orientados a objetos escritos en lenguaje Java, que agregue correcta y lógicamente la abstracción funcional entre las unidades o módulos de software de su arquitectura, conducido por el principio de inversión de dependencias, para conseguir mejores condiciones de flexibilidad y extensibilidad de arquitecturas orientadas a objetos sin menoscabo de conservar la funcionalidad original.

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. El servicio deberá medir el grado de abstracción y

de Polimorfismo de arquitecturas de clases de programas legados o marcos orientados a objetos y tendrá la capacidad de sugerir su arquitectura mejorada. Finalmente, después de la refactorización deberá medirse nuevamente el grado de estas métricas de las arquitecturas de clases de programa mejoradas y alertar al usuario sobre el grado de mejora en estas dimensiones para aceptar la refactorización automática de (los) problema(s) de diseño que se presente(n).

2.3.- Justificación

El uso de comportamientos funcionales directamente de las clases concretas, donde su comportamiento es fijo, impacta en su nivel de suficiencia y completitud para alcanzar las metas u objetivos de los clientes. Para el desarrollo de este proyecto de tesis, se requiere específicamente, mejorar la generalidad, modularidad, flexibilidad, extensibilidad y autonomía de arquitecturas de clases de objetos que exhiben en su estructura un incorrecto uso de la propiedad de abstracción, asociación dinámica de tipos y de polimorfismo, mediante su refactorización arquitectural.

Esto significa, incorporar las propiedades de abstracción y de polimorfismo de clases de objetos para invertir las dependencias funcionales. Haciendo flexible el comportamiento funcional de objetos, y la propiedad de extensibilidad para la incorporación de nuevas funcionalidades que se ajustan al principio abierto-cerrado y el principio de inversión de dependencias.

2.4.- Objetivo

2.4.1.- Objetivo General

Complementar la mejora del Diseño de Arquitecturas de Componentes de Software Existentes escritos en lenguaje Java, de las arquitecturas de software que no soportan las propiedades de asociación dinámica de tipos y de polimorfismo que brinda el modelo de programación orientada a objetos.

2.4.2.- Objetivos Específicos

- Disminuir la Deuda Técnica en sus dimensiones de Flexibilidad y Extensibilidad.
- Mejorar la modularidad en las dimensiones de asociación dinámica de tipos y de polimorfismo.

2.5.- Estado del Arte

El trabajo de tesis desarrollado es parte de un proyecto denominado SR2: Reingeniería de Software Legado para Reuso, proyecto propuesto por el área de ingeniería de software del CENIDET.

En el CENIDET se han planteado e implementado varios proyectos de reingeniería por medio de proponer, analizar, diseñar, e implementar métodos de refactorización, para mejorar arquitecturas de software existentes escritos en lenguajes de programación C++ y Java, donde se cuenta con el desarrollo de herramientas para:

- Adaptar interfaces lógicas que no empatan a las necesidades de los clientes.
- Medir el problema de interfaces no utilizadas en marcos de aplicaciones orientados a objetos y refactorizar esas interfaces no utilizadas.
- Medir el valor de acoplamiento que produce dependencias, refactorizando este factor al mínimo indispensable.
- Refactorización de marcos orientados a objetos con abstracciones incorrectas, que equilibran las métricas Depth of Inheritance (DIT) y Number of Children (NOC).
- Refactorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos.
- Refactorización de arquitecturas de marcos orientados a objetos con funciones atómicas globalmente visibles, lo cual viola el criterio de protección modular.
- Refactorizar para equilibrar la coherencia, cohesión y el factor de acoplamiento.
- Método de refactorización de arquitecturas de software con carencia de abstracciones.

Para el cumplimiento del proyecto SR2, dentro del CENIDET se han realizado y se continúan realizando un conjunto de tesis relacionadas al tema de la refactorización de software legado.

A continuación, se mencionan las tesis que se han desarrollado en el CENIDET para la refactorización de software legado.

- Manuel Alejandro Valdés Marrero, “Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 2 de diciembre del 2004.

- Leonor Adriana Cárdenas Robledo, “Refactorización de Marcos Orientados a Objetos para Reducir el Acoplamiento Aplicando el Patrón de Diseño Mediator”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 19 de agosto del 2004.
- Luis Esteban Santos Castillo, “Adaptación de interfaces de Marcos de Aplicaciones Orientados a Objetos por Medio del Patrón de Diseño Adapter”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 27 de enero del 2005.
- Pablo Padilla Salgado, “Métodos de refactorización de código Java con interfaces y abstracciones incorrectas”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 29 de julio del 2019.
- Orlando Ortíz Gutiérrez, “Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 15 de junio del 2020.
- Nélida Barón Pérez, “Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 11 de diciembre del 2020.
- Ricardo Tello Diaz, “Método de re-factorización de arquitecturas de software con carencia de abstracciones”, tesis de maestría aún en desarrollo en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico.
- Marisol Ramírez Cruz, “Métodos de re-factorización de código java para mejorar su modularidad y reducir las dependencias entre clases de objetos”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 16 de diciembre del 2022.
- Juan Antonio Diaz Diaz, “Disminución de Deuda Técnica producida por arquitecturas de clases con más de Una Responsabilidad”, tesis de maestría desarrollada en el TecNM/Centro Nacional de Investigación y Desarrollo Tecnológico, con fecha de terminación 17 de marzo del 2022.

El sistema SR2 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.

Cabe señalar que el desarrollo de esta tesis es una extensión del "*Método de refactorización de arquitecturas de software con carencia de abstracciones*" para implementar la funcionalidad de refactorización a los módulos colaborativos que cuenten con carencia de abstracción.

2.6.- Trabajos Relacionados

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

La finalidad de esta publicación es presentar un enfoque de refactorización el cual es totalmente automatizado y basado en restricciones, que ayuda a los desarrolladores a aprovechar las interfaces mejoradas que se introducen en Java 8. Este enfoque presenta un amplio conjunto de reglas que cubren varios casos en los que no se pueden utilizar métodos predeterminados.

Para demostrar la aplicabilidad, se implementó como un complemento de Eclipse y fue utilizado para comprobar su correcto funcionamiento en 19 proyectos de Java del mundo real, así como solicitudes de extracción enviadas a los repositorios populares de GitHub. Los resultados de la evaluación demostraron que es útil para migrar los métodos de implementación "Squeletal" a las interfaces como métodos default, arroja luz sobre el uso del patrón "Squeletal" y proporciona información a los diseñadores de lenguajes sobre cómo este nuevo constructo se aplica al software existente.

Arcan: a tool for architectural smells detection. (Fontana et al., 2017)

Arcan es una herramienta de código abierto la cual ha sido desarrollada para la detección de olores en el código de una arquitectura de software, los cuales se detectan a través de una evaluación de varios problemas de dependencia en la arquitectura, las técnicas de detección utilizadas en esta herramienta explotan la tecnología empleada a través de su base de datos gráfica, lo cual permite una gran escalabilidad en la detección de código mal oliente y una gran gestión de las dependencias de múltiples tipos.

La herramienta funciona brindando la carpeta de archivos "class" para su lectura y detección de clases que son específicos del proyecto y no parte de componentes

externos. Arcan es diseñada en su infraestructura mediante cuatro componentes principales: La interfaz definida en Java FX, la unidad de procesamiento desarrollada en Java 8, la implementación del framework Tinkerpop como interfaz a la base de datos y Neo4j como base de datos de grafos. El uso de Tinkerpop facilita la construcción y acceso al grafo de dependencias que representa al proyecto analizado por Arcan, además de permitir la explotación de diferentes servicios de base de datos de grafos. Arcan aplica algoritmos al grafo construido del proyecto analizado para extrapolar información detallada de la estructura del proyecto. Los algoritmos aplicados agregan nuevos nodos y hojas como “smells nodes” los cuales indican la presencia de arquitecturas con mal olor en el proyecto.

Architectural Refactoring: A Task-Centric View on Software Evolution. (Zimmermann, 2015)

El principal tema que aborda este trabajo trata sobre la refactorización arquitectónica (AR), ya que si bien la refactorización ha tenido gran auge en los últimos años, es sorprendente que para la AR no despegue aún, por lo cuál se muestra cómo la AR puede servir como una técnica de evolución que revisa las decisiones arquitectónicas e identifica las tareas relacionadas con el diseño, la implementación y la documentación.

AR se enfoca en la documentación y la manifestación de la arquitectura en el código en tiempo de ejecución; aborda los olores arquitectónicos, los cuales son sospechas o indicaciones de que algo en la arquitectura ya no es adecuado según los requisitos y restricciones. Por lo tanto, una AR, es un conjunto de actividades coordinadas deliberadamente que eliminan un olor arquitectónico particular y mejoran al menos un atributo de calidad sin cambiar el alcance y funcionalidad del sistema.

JDeodorant Eclipse Plugin. (Tsantalis Nikolaos et al., 2018)

JDeodorant es una herramienta de refactorización que engloba una serie de técnicas que sugieren y aplican automáticamente oportunidades de refactorización en código fuente Java, de tal forma que por parte del desarrollador se requiere un esfuerzo limitado. A diferencia de otros enfoques de refactorización que se basan en estrategias genéricas que pueden adaptarse a varios olores de código, este plugin adopta estrategias ad-hoc para cada olor teniendo en cuenta las características particulares del diseño subyacente o el problema que se presenta en el código; ha contribuido con técnicas de recomendación de refactorización para una variedad de olores de código, incluyendo Feature Envy, State Checking and Type Checking, Longh Method, God Class, Duplicated Code, and Refused Bequest.

Actualmente JDeodorant cumple con las características de convertir la experiencia de uso en procesos automatizados, así como también en realizar una evaluación previa de la efectividad sobre cada solución propuesta para mejorar el código a analizar, guiando a los usuarios a comprender los problemas de diseño que presentan en su código, con la finalidad de mejorar la calidad de su diseño.

SonarQube. (Lenarduzzi et al., 2020)

SonarQube es una herramienta de revisión de código automática y autoadministrada que ayuda sistemáticamente a entregar un código limpio, realiza un análisis estático el cual corresponde al proceso de evaluar un software sin ejecutarlo, esto con el objetivo de advertir al cliente sobre diferentes puntos a mejorar dentro del código y obtener métricas para su mejora, propone un conjunto de reglas de codificación, que representan algo erróneo en el código que pronto se reflejará en un fallo o aumentará el esfuerzo de mantenimiento.

Este sistema, a través de un conjunto de reglas, analiza la conformidad del código. Si el código infringe una regla, SonarQube añade el tiempo necesario para refactorizar la regla infringida como parte de la deuda técnica. También identifica un conjunto de reglas como “errores”, en donde alega que “representan algo incorrecto en el código y pronto se reflejarán en un fallo”; además, también afirman que se esperan cero falsos positivos de los “errores”.

Una de las principales ventajas que tiene una herramienta de análisis de código estático como lo es SonarQube, se destaca la capacidad de identificación de aspectos tales como: código duplicado, código muerto, estándares de codificación, complejidad ciclomática, comentarios, test unitarios y test de integración.

Automated Refactoring to the Strategy Design Pattern. (Christopoulou et al., 2012)
El método es un complemento de Jdeodorant que se enfoca al patrón de diseño “State”. Este trabajo se centra en la identificación automática de oportunidades de refactorización para el patrón de diseño “Strategy”, mediante la introducción de polimorfismo para eliminar los olores de código asociados con el uso extensivo de declaraciones condicionales complejas.

Durante la investigación, se implementó un algoritmo para identificar oportunidades de refactorización para adoptar el patrón de diseño 'Strategy'. El trabajo específica un proceso de refactorización 'estratégico' para sentencias condicionales definidas.

Maneja casos especiales en los que la refactorización elimina por completo las declaraciones condicionales mediante el uso de la llamada al método de instancia de patrón de política adecuado. El método busca oportunidades de refactorización entre sentencias condicionales declaradas en una clase contexto.

Cabe señalar que los Trabajos de tesis de la M.C Laura Alicia Hernández Moreno y el M.C César Bustamante Laos, que aplican los patrones de diseño “Strategy” y “State” en el mismo sentido que la publicación en (Christopoulou et al., 2012). Los resultados de estos trabajos de tesis fueron presentados en el año 2003 en la International Conference, Montreal, Canada y publicado en Computational Science and its Applications (ICCSA 2003), con título “Restructuring conditional code structures using object oriented design patterns”. Autores René Santaolaya S., Oliva G. Fragoso D., Joaquín Pérez O., y Lorenzo Zambrano S.

Improving the Quality of Software by Refactoring. (Kaur & Singh, 2017)

La meta de este trabajo es reducir el costo de mantenimiento de sistemas de software al usar un algoritmo que proponen los autores para la mejora del proceso de refactorización.

Los autores realizaron pruebas en diferentes versiones del proyecto Junit identificando estructuras de código que han sido refactorizadas anteriormente, calcularon las métricas de código en cada versión usando la herramienta Halstead, usaron la herramienta Ref-finder para extraer diferencias entre refactorizaciones de código en versiones del proyecto Junit, posteriormente analizaron las métricas obtenidas de las versiones del proyecto. Aplicaron el algoritmo de refactorización mejorado que proponen, nuevamente aplican las métricas de código con Halstead y comparan estos resultados con las técnicas existentes. Los resultados que brinda Halstead son para las siguientes métricas:

- Nested Block Depth (NBD): Ayuda a identificar si una clase o un método está sirviendo para más de un propósito y que puede seguir agregando líneas de código volviéndolo inmanejable.
- Number Of Parameters (NOP): Indica que con el aumento de funciones deseadas en un método el número de parámetros también aumenta, y mientras más parámetros sean añadidos, mayor será la complejidad del método. Por lo tanto, un menor número de parámetros ayuda a la mantenibilidad del código.
- Number Of Classes (NOC): Define la separación de funciones entre clases. El número de clases incrementa conforme se refactoriza.

Support for Architectural Smell Refactoring. (Rizzi et al., 2018)
Este trabajo busca ampliar la atención de los desarrolladores para la identificación de arquitecturas no sustentables que contienen el problema de dependencias cíclicas en aplicaciones desarrolladas en el lenguaje de programación Java.

La herramienta describe el camino que un desarrollador debe tomar para remover la dependencia cíclica. El método aplica la función de prioridad que es una modificación de función de ganancia desarrollada por Caracciolo y varios. La modificación diferencia en una mejor forma el impacto de cada elemento del ciclo, comprender la complejidad de las dependencias requiere de la aplicación de métricas que tomó como instanciabilidad, que es calculado con el número de métodos llamados desde/hacia una clase y la métrica de abstracción que es calculada con el número de dependencias con clases abstractas. La función de prioridad sugiere qué elemento tiene la prioridad de ser removido.

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

El trabajo presenta los argumentos para la investigación de hacer un algoritmo de refactorización y mejora a las implementaciones del patrón de diseño “Skeletal” en código legado Java, utilizando las nuevas características implementadas en Java 8. En esta investigación el enfoque fue desarrollar una herramienta que realice la refactorización de forma automática a las implementaciones del patrón de diseño “Skeletal” para el uso de las nuevas características que aporta Java 8. Esto es mediante la identificación de cuerpos de métodos que pueden ser usados como el cuerpo default para los métodos que se definen en las interfaces, esta capacidad permite que las clases concretas necesiten únicamente las interfaces en vez de utilizar clases abstractas que implementan la interfaz y que son extendidas por las clases concretas.

3.- Marco Teórico

3.1.- Paradigma de Programación Orientado a Objetos

Un paradigma es una metodología que intenta unificar y simplificar la manera en que se resuelve un cierto grupo de problemas. En el contexto de la programación, un paradigma es un conjunto de principios y métodos que sirven para resolver problemas a los que los desarrolladores de software se enfrentan al momento de construir sistemas de software grandes y complejos. (Cervantes O et al., 2016)

En la programación, existen diferentes paradigmas, para este desarrollo de tesis se trabajó sobre uno de los más importantes que es el *Paradigma Orientado a Objetos,* en el cual, para crear aplicaciones, los programas trabajan con base en unidades llamadas objetos. Se basa en los siguientes pilares fundamentales que son: *herencia, polimorfismo, encapsulamiento y abstracción*, de los cuales, el polimorfismo y la abstracción son los que se trabajan en el desarrollo de este método de refactorización.

3.1.1.- Herencia

La herencia permite definir nuevas clases denominadas “derivadas, hijas o subclases”, a partir de clases ya existentes, llamadas “base, padre o superclase”. De esta manera los objetos pueden construirse en base a otros objetos ya creados.

La “herencia” o “derivación de clases” es el mecanismo para compartir automáticamente los métodos (comportamiento) y datos (estado implícito) de la “clase base”, agregando otros nuevos a las “clases derivadas”. Es decir, la herencia impone una organización jerárquica entre clases, permitiendo que los datos y métodos de una clase sean heredados por sus descendientes. (Mazón Olivo et al., 2015)

3.1.2.- Polimorfismo

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder de diferentes formas a un mismo mensaje o evento en función del tipo asociado dinámicamente del objeto particular. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

El polimorfismo consiste en conseguir que un objeto de una clase base se comporte como un objeto de cualquiera de sus subclases. Una forma de conseguir objetos polimórficos es mediante el uso de punteros a la superclase (University, 2016).

3.1.3.- Abstracción.

La abstracción es la habilidad de un problema de ignorar los detalles de las partes para enfocar la atención en un nivel más alto de generalidad, es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. La abstracción posee diversos grados o niveles, los

cuales ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. (Angélica Nakayama C Ing Jorge A Solano Gálvez & Alejandro Velázquez Mena, 2017)

En programación, la abstracción es un concepto importante y es utilizada para reducir la complejidad de un programa, a través del uso de clases e interfaces, que permite el reuso de código y la simplificación de éste a través de la herencia.

Con la abstracción habilitamos la asociación dinámica de tipos y el polimorfismo, con la presencia de esta propiedad se tiene una arquitectura flexible a cambios de comportamiento en tiempo de ejecución y esto hace posible que se pueda extender a nuevas funcionalidades o comportamientos requeridos en los sistemas de software.

3.1.4.- Interfaz

El concepto de interfaces lleva un paso más adelante la idea de las clases abstractas. Una Interfaz es una clase abstracta pura, esto quiere decir que todos sus métodos son abstractos, es un contrato sobre qué puede hacer la clase, sin decir cómo lo va a hacer, y debido a que es una interfaz abstracta, no se es posible instanciarla. Todos los métodos declarados en una interfaz siempre son públicos y abstractos, de igual forma puede contener atributos, los cuales siempre son públicos, estáticos y finales. (ARKAITZ GARRO, 2014)

En Java, una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementación, similarmente a un método abstracto) que especifican un protocolo de comportamiento para una o varias clases.

Una interfaz es como una clase donde todos sus métodos son abstractos y públicos, es decir los métodos sólo se declaran y no contienen implementación de su código. Se utilizan interfaces para asociar métodos que son comunes a un grupo de objetos indistintamente de si son o no de una misma clase. Una clase puede implementar una o más interfaces (múltiples interfaces) (Mazón Olivo et al., 2015).

Una interfaz puede parecer similar a una clase abstracta, pero existen una serie de diferencias entre una interfaz y una clase abstracta:

- Implícitamente, todos los métodos de una interfaz son declarados como abstractos y públicos.

- En una clase abstracta no se pueden implementar los métodos declarados como abstractos, pero al menos uno de sus métodos tiene cuerpo de implementación. En una interfaz no es posible implementar algún método (ya que todos son abstractos).
- Tanto en las interfaces como en las clases abstractas, no es posible declarar variables de instancia.
- Una subclase puede implementar varias interfaces, pero sólo puede tener una clase base ascendiente directa.

3.2.- Principios de Diseño Orientados a Objetos.

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

Generalmente, en un mal diseño, modificar una funcionalidad durante el ciclo de vida, suele conllevar una cadena de cambios en módulos dependientes, y este efecto puede propagarse en cascada si el código está muy acoplado. La forma de realizar diseños de software, para que permanezcan estables ante los cambios, es el tema que aborda el principio abierto – cerrado enunciado por Bertrand Meyer, el cual dice lo siguiente: "*Las entidades de Software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerradas para su modificación*" (Meyer, 1997).

- Se dice que un módulo es abierto a extensiones si éste puede ampliar su comportamiento. Por ejemplo, debe ser posible ampliar su conjunto de operaciones o agregar campos a sus estructuras de datos.

- Se dice que un módulo es cerrado a modificaciones si éste puede ser utilizado por otros módulos. Esto supone que el módulo ha recibido una descripción estable y bien definida (su interfaz en el sentido de la ocultación de información). A nivel de implementación, el cierre de un módulo también implica que se puede compilar, quizá almacenar en una biblioteca, y ponerlo a disposición de otros (sus clientes) para que lo utilicen. En el caso del diseño de un módulo o de una especificación, cerrar un módulo significa simplemente que la dirección lo aprueba, que se agrega al repositorio oficial del proyecto de elementos de software aceptados (a menudo denominado línea de base del proyecto) y que se puede publicar su interfaz en beneficio de otros autores de módulos.

La idea principal de este principio expresa que, ante la necesidad de cambios en los requisitos, el diseño de las entidades existentes permanezca inalterado, lo cual

significa que, ante peticiones de cambio en el sistema, se debe ser capaz de agregar funcionalidades sin modificar (sin editar) las funcionalidades ya existentes (José et al., n.d.).

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

El principio de inversión de dependencias (por sus siglas en inglés DIP) es una técnica fundamental y una de las que más absorbemos a diario si queremos que nuestro código sea comprobable y mantenible. Gracias al principio de inversión de dependencias, el código central de una aplicación puede ser independiente de los detalles de implementación.

En su libro "*Agile Software Development*", Robert C. Martin menciona que el término "*inversión*" se debe a que los métodos tradicionales de desarrollo de software, como el análisis y diseño procedural (conocido como estructurado), tienden a crear estructuras de software en las que los módulos de alto nivel dependen de los módulos de bajo nivel, y en las que las políticas importantes de un negocio dependen de los detalles. La estructura de dependencia de un programa orientado a objetos bien diseñado está "invertida" con respecto a la estructura de dependencia que normalmente resulta de los métodos procedurales tradicionales. (R. C. Martin, 2014)

La definición que se suele dar a este principio es:

a) *Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.*

b) *Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.*

3.3.- Refactorización

La programación de software conlleva un proceso largo que suele involucrar varios desarrolladores, por lo que durante el desarrollo el código fuente a menudo se extiende más y esto provoca que se tengan modificaciones en su funcionamiento. Además, se toma en cuenta que con la implementación de metodologías ágiles se trabaja contrarreloj, como consecuencia observamos como resultado una acumulación de elementos defectuosos en el código fuente, los llamados *"code smells"*. Estos puntos débiles, que van aumentando a medida que avanza el proceso, ponen en peligro la funcionalidad y la compatibilidad del software en

cuestión. Para evitar esta erosión continua del programa, se utiliza la refactorización o refactoring.

Martin Fowler en su libro *"Refactoring, improving the design of existing code"* define a la refactorización como: "*así como en el tiempo el software se deteriora poco a poco, la refactorización debería mejorar el código poco a poco*" (Fowler et al., 2019). La refactorización tiene el sencillo y claro propósito de mejorar el código. Con un código de más calidad, puede facilitarse la integración de nuevos elementos sin incurrir en errores nuevos. Además, cuanto más fácil resulte a los programadores leer el código, más rápido se familiarizarán con él y podrán identificar y evitar los defectos de forma más eficiente. Otro objetivo de la refactorización es mejorar el análisis de defectos y minimizar la necesidad de mantenimiento del software.

3.4.- ANTLR

Para el desarrollo del método de refactorización y el cálculo de métricas del proyecto de este trabajo de tesis, se hace uso de la herramienta ANTLR (Another Tool for Language Recognition) (Parr, 2013). ANTLR es un generador de analizadores, que proporciona un reconocedor de lenguajes como Java, C++ y C#. La mayoría de la gente llama a este tipo de herramientas como "*compiladores de compiladores*" dado que ayudar a implementar compiladores es su uso más popular. ANTLR es un programa escrito en lenguaje Java, por lo que se necesita de alguna máquina virtual de Java para poder ejecutarlo. Es software libre, lo que significa que cuando se descarga el código desde su sitio oficial se obtiene el código fuente compilado en forma de archivos *.class y archivos *.java.

3.5.- JUnit

JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java, es una herramienta de software libre, por lo cual se puede extender. JUnit provee al usuario un conjunto de herramientas, clases y métodos que facilitan la tarea de realizar pruebas en sus sistemas y así asegurar su consistencia y funcionalidad.

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que

regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente. (Vertiz Alexis, 2017)

3.6.- Teoría de la Medición

La necesidad de medir es evidente en la mayoría de las actividades técnicas o científicas. Para ésto debemos recordar la definición de medición como “el proceso por el cual se asignan números o símbolos a atributos de entidades del mundo real de tal forma que los describa de acuerdo con reglas claramente definidas” (Fenton & Pfleeger, 1997).–El hecho de que los numerales puedan ser asignados bajo diferentes reglas nos lleva a diferentes tipos de escalas y, por ende, diferentes tipos de medición. (Stevens, 1946)

La validez de la medición en cualquier disciplina técnica o científica se basa en el respeto a los principios de la teoría general de la medición (en concreto, nos apoyaremos en la llamada teoría representacional de la medición). Esta idea es análoga a lo que se hace en matemáticas (por ejemplo, en geometría) donde se definen una serie de axiomas básicos y, a partir de ellos, se van estableciendo nuevas conclusiones.

El fundamento de la teoría representacional consiste en que toda medición debe asegurar una adecuada representación del atributo real medido mediante los símbolos o números asignados. Una representación por medición de un atributo de una entidad es adecuada si es coherente con la idea conceptual que sobre dicho atributo es comúnmente aceptada por los expertos. (Teoría de Medición, 1998.)

3.5.1.- Escalas o niveles de Medición.

De acuerdo con la definición clásica de Stevens, medir significa “asignar números, símbolos o valores a las propiedades de objetos o eventos de acuerdo con reglas”. Las mediciones se pueden realizar por medio de cuatro escalas de medición. Dos de las escalas miden variables categóricas y las otras dos miden variables numéricas. Las escalas categóricas se usan comúnmente para variables cualitativas, mientras que las numéricas son adecuadas para la medición de variables cuantitativas.

Los datos se pueden clasificar en cuatro niveles de medición:

- **Nivel de escala nominales**
- **Nivel de escala ordinal**
- **Nivel de escala de intervalo**

- **Nivel de escala de razón**

Las dos primeras (nominal y ordinal) se conocen como escalas categóricas, y las dos últimas (intervalo y razón) como escalas numéricas.

Se utilizan para ayudar a clasificar variables, desarrollar preguntas para medir variables e incluso indicar el tipo de análisis estadístico apropiado para el procesamiento de datos. La propiedad básica de la medición es que depende de la probabilidad de cambio. La validez y confiabilidad de la medición de una variable depende de las decisiones que se tomen para manipularla y comprender cabalmente los conceptos, evitando imprecisiones y ambigüedades; de lo contrario, la variable tiene un riesgo inherente de volverse inválida porque no proporciona información confiable.

3.5.1.1.- Nivel de Escala Nominal.

Este nivel de escala es utilizado para las variables cualitativas y determina igualdad o pertenencia a una categoría, es decir, nos permite clasificarlas. Es la escala más elemental y la forma más rudimentaría de hacer una medición. En una escala como ésta se clasifica a las unidades de estudio (objetos, personas, etc.) en categorías, basándose en una o más características, atributos o propiedades distintivas y observadas, dándole a cada categoría un nombre (de ahí lo de “nominal”). (Coronado Padilla, 2007.)

Con las escalas nominales, los números asignados definen cada grupo distinto y sirven meramente como etiquetas o identificadores.

3.5.1.2.- Nivel de Escala Ordinal.

El nivel de escala ordinal otorga una clasificación y orden de datos sin que realmente se establezca el grado de variación entre ellos. Los datos ordinales son básicamente datos estadísticos que tienen la misma naturalidad, pero existe una diferencia entre ellos que es desconocida. Los valores dentro de esta escala se pueden clasificar en categorías y se pueden ordenar en jerarquías con respecto a la característica que se evalúa.

Una escala de medición ordinal se logra cuando las observaciones pueden colocarse en un orden relativo con respecto a la característica que se evalúa. Es decir, las categorías de datos están clasificadas y ordenadas de acuerdo con la característica especial que poseen. Aquí, las etiquetas o símbolos de las categorías indican jerarquía. Si utilizamos números, la magnitud de esos no es arbitraria, sino que representa el orden del rango del atributo observado. Se supone un continuo subyacente en los números, de modo que las relaciones típicas son, en este caso,

<<más alto que>> o <<preferible a>>. Sólo las relaciones <<mayor que>>, <<menor que>>, e <<igual a>> tienen significado en una escala de medición ordinal.

Para clasificar una métrica como escala ordinal, debe cumplir los requisitos del axioma de orden débil, los cuales son: que ·> es una relación binaria completa y transitiva. Las propiedades de transitividad y completitud son las siguientes (Zuse, 1992):

- Transitividad: P ·> P', P' ·> P'' => P ·> P''
- Completitud: P ·> P' o P' ·> P

Para todo P', P'' Ɛ P, donde P es un conjunto y •≥ es una relación empírica binaria como "igual o más compleja que". Supóngase que (P, •≥) es un sistema relacional empírico, donde P es un conjunto contable no vacío y •≥ es una relación binaria en P. Luego existe una función μ: P → ℜ, con: P' •≥ P'' ↔ μ(P') ≥ μ(P'') para todo P', P'' Ɛ P, sí y sólo sí, •≥ es de orden débil. Si tal homomorfismo existe, entonces, ((P, •≥), (ℜ, ≥), μ) es de escala ordinal. La medida μ en una escala es un homomorfismo.

3.5.1.3.- Nivel de Escala de Intervalo

La escala de intervalo comprende datos continuos o datos discretos que contienen un elevado número de posibles valores. Las distancias entre cualquier par de números de la escala tienen una dimensión conocida y constante por lo que es posible conocer con certeza la magnitud de los intervalos.

Se caracteriza por tener una unidad común de medida que asigna un número real a todos los pares de objetos en el conjunto ordenado. Aún cuando el punto cero y la unidad de medida son arbitrarios, la razón entre dos intervalos es independiente de esa unidad y de ese punto.

Este tipo de escala es más refinada puesto que además del orden o jerarquía entre categorías, las etiquetas o números consecutivos establecen intervalos iguales en la medición. La medición en una escala de intervalos se basa en suponer que puede conocerse exactamente la diferencia entre los objetos medidos según esta escala. (Coronado Padilla, 2007)

3.5.1.4.- Nivel de Escala de Razón.

Este tipo de escala es similar a la de intervalo, con la única diferencia que el cero en esta escala indica la ausencia de atributo, es cero absoluto, es una escala de medida o modo de medición que permite hacer la comparación del tipo cuántas

veces es mayor un elemento que otro con relación a una variable que se quiere medir. El rasgo distintivo de una escala de razón es la existencia de un cero absoluto, de modo que el cero indica que el elemento no tiene nada de la propiedad a estudiar.

4.- Materiales y Método de Solución.

4.1.- Método de Refactorización.

Para el método de refactorización desarrollado en esta tesis, se requiere específicamente mejorar la modularidad en las dimensiones de *flexibilidad* ya que con ella se tiene la capacidad de hacer cambios a los comportamientos existentes de un sistema, y *extensibilidad*, donde se tiene la capacidad de agregar nuevas funcionalidades al sistema sobre las ya existentes, respetando así los principios de *abierto-cerrado* e *inversión de dependencias* del software legado de aplicaciones de software.

Aplicando cada una de estas dimensiones, se incorporan las propiedades de abstracción y polimorfismo en unidades o módulos de programa colaborativos para invertir las dependencias funcionales, flexibilizando el comportamiento funcional de objetos y la propiedad de extensibilidad para la incorporación de nuevas funcionalidades

La función principal del método de refactorización de módulos colaborativos con carencia de abstracción es identificar clases abstractas en paquetes de programa que puedan ser refactorizados a una Interfaz, cumpliendo con los siguientes criterios para refactorizar:

- Una clase abstracta con solo métodos públicos abstractos se deberá convertir a una interfaz.
- Si una clase abstracta cuenta por lo menos con un método con cuerpo entonces no se aplica la refactorización.
- En clases abstractas con modificadores de acceso “*private*” o “*protected*” no se realizará la refactorización.
- Métodos virtuales definidos en clases abstractas cuyos cuerpos sean nulos (que no tienen código) deben convertirse a interfaces sin cuerpo de código.
- Toda clase abstracta que define métodos virtuales cuyo cuerpo sea nulo se deberá convertir a una interfaz.

- Una clase abstracta que define un método virtual sin cuerpo, pero implementa más métodos en la clase con cuerpo no se deberá convertir a interfaz.
- Clases abstractas que contengan atributos de clase no son refactorizadas a interfaces, aun cuando cuente con al menos un método abstracto.

Como se mencionó anteriormente, este trabajo de investigación es una extensión del proyecto de tesis de maestría "*Método de Refactorización de Arquitecturas de Software con Carencia de Abstracciones*". Se hizo uso del modelo de objetos de la Figura 2, generado en ese trabajo con la finalidad de poder realizar la refactorización que se pretende abarcar en esta tesis, no obstante, para tener una buena refactorización de clases abstractas a interfaces, se hicieron algunos ajustes a este modelo.

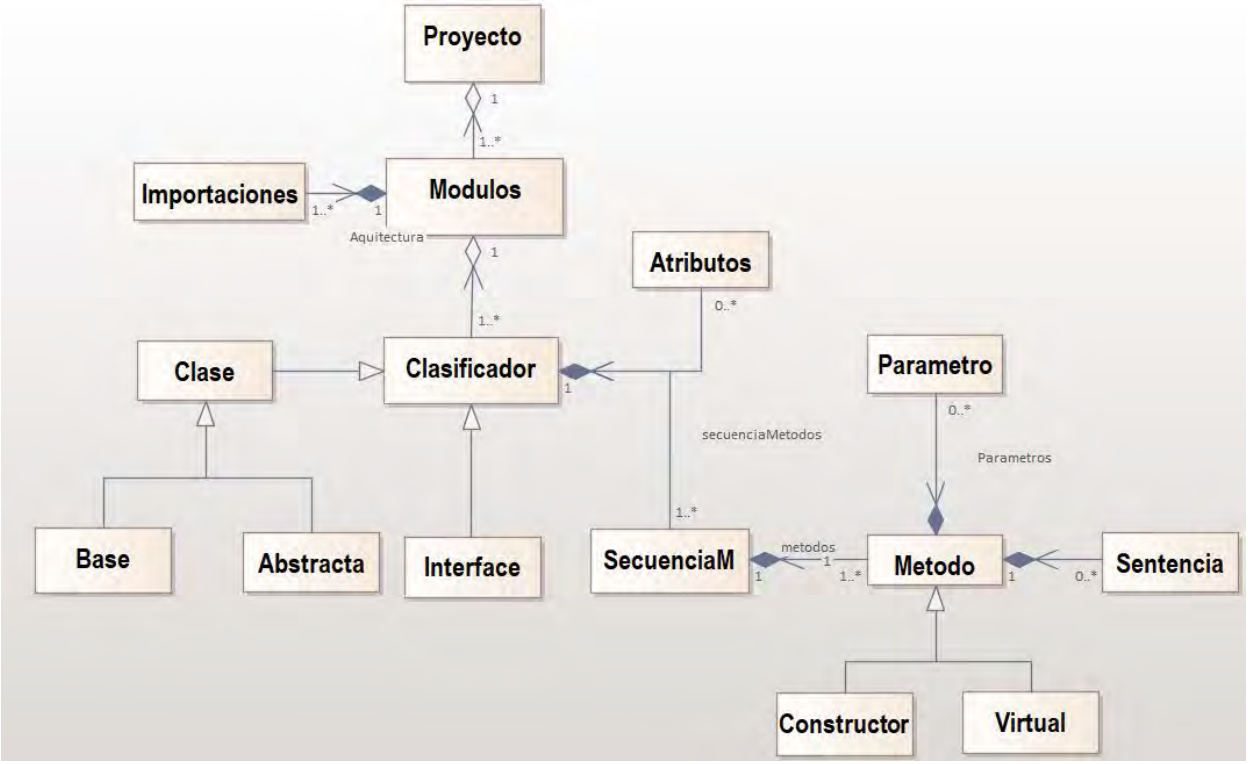


Figura 2 Diagrama de clases del Metamodelo.

4.2.- Métricas.

Para la medición del grado de abstracción y polimorfismo de arquitecturas de clases de programas legados o marcos orientados a objetos, se hizo uso de las métricas Factor de Abstracción (FA) y Factor de Polimorfismo (FP).

A fin de comprobar que la refactorización ha sido o es exitosa, el servicio deberá medir el grado de abstracción y polimorfismo antes y después de aplicar el método de refactorización.

Una Interfaz puede parecerse a una clase abstracta, pero existen una serie de diferencias entre estas dos:

- Todos los métodos de una interfaz se declaran implícitamente como abstractos y públicos.
- Una clase puede implementar varias interfaces, pero sólo puede tener una clase ascendente directa.

4.2.1.- Métrica Factor de Abstracción.

La métrica de factor abstracción en este proyecto de tesis se utiliza para medir la propiedad de abstracción en un paquete de un sistema de software. La abstracción permite satisfacer el principio de diseño "*Abierto-Cerrado*" en donde la idea principal de este principio es que las clases están abiertas a su extensión, pero cerradas a la modificación, lo cual significa que ante peticiones de cambiar o modificar algo en nuestro sistema, se debe ser capaz de añadir funcionalidad sin modificar la ya existente.

Acorde con Robert Martin, si un paquete del sistema fuera mínimamente estable, el sistema sería completamente inmutable, por lo que si deseamos que un sistema sea extensible a cambios de comportamientos este debe ser abstracto para que pueda ampliar su funcionamiento fácilmente (R. Martin, 1994).

Una categoría abstracta debe tener dependientes, ya que debe haber clases, fuera de la categoría abstracta, que hereden de ella e implementen las interfaces puras que faltan. Sin embargo, no queremos fomentar las dependencias de categorías inestables.

La métrica de abstracción es una métrica lineal la cual mide lo abstracto que es un paquete, en el sentido de que calculará la relación entre el número de clases abstractas o interfaces y el número total de clases dentro de un paquete. La fórmula para la métrica de abstracción es la siguiente:

$$FA = \frac{Na}{Nc}$$

Donde:

- Na es el número de clases abstractas e interfaces en el paquete.

- Nc es el número total de clases concretas en el paquete.

Esta métrica dará valores en el rango de [0, 1]. FA = 0 significa que todas las clases de este paquete no son abstractas, FA = 1 indica un paquete que consta únicamente de clases e interfaces abstractas. Los paquetes más abstractos son mejores porque son más fáciles de mantener y reutilizar.

4.2.2.- Métrica Factor de Polimorfismo.

Está métrica es parte del conjunto de métricas MOOD (Metrics for Object Oriented Design) definidas por (Brito Abreu & Melo, 1996), estás métricas operan a nivel de sistema, y se refieren a mecanismos estructurados los cuales están bajo el paradigma de la orientación a objetos, tales como encapsulación, herencia, polimorfismo y paso de mensajes, así también la calidad de software y la productividad en el desarrollo. Este tipo de métricas pueden ser utilizadas en las fases de diseño y han sido definidas para ser aplicadas a un nivel de diagrama de clases.

Para este proyecto de tesis, se hace uso de la métrica de Factor Polimorfismo (FP), la cual se define como "*la proporción entre el número real de posibles diferentes situaciones polimórficas para una clase Ci entre el máximo número posible de situaciones polimórficas en Ci*" (Rodríguez & Harrison, n.d.). El propósito de FP es una medida del polimorfismo y una medida indirecta de la asociación dinámica en un sistema.

Esta métrica de FP se formula de la siguiente manera:

$$FP = \frac{\sum_{i=1}^{TC} M_o\,(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) \times DC(C_i)]}$$

Donde:

$M_d(C_i) = M_n(C_i) + M_o(C_i)$

$DC(C_i)$ es el número de descendientes de Ci.
$M_n(C_i)$ es el número de métodos nuevos.
$M_o(C_i)$ es el número de métodos redefinidos.
TC es el número total de clases.

4.3.- Diagrama BPMN

Un diagrama BPMN (Business Process Modeling Notation), es un lenguaje de modelado ampliamente utilizado para crear flujos de trabajo de procesos. “En BPMN, los “Procesos de Negocio” involucran la captura de una secuencia ordenada de las actividades e información que utiliza el proceso” (Hitpass, 2012).

El propósito de utilizar la notación BPMN en este trabajo de investigación, es identificar específicamente la secuencia de actividades del proceso de refactorización. La Figura 6 muestra el proceso del método de refactorización para la detección, automatización y corrección del software. El proceso de refactorización consiste en los siguientes subsistemas:

- **Cliente:** usuario encargado de seleccionar el proyecto o sistema de software, escrito en lenguaje Java, para comenzar con el análisis de los módulos que conforman dicho proyecto o sistema.
- **Analizador sintáctico:** derivado de ANTLR (Parr, 2013) analiza el código fuente a través de tokens con la finalidad de capturar la información relacionada con las clases abstractas que contiene el proyecto para generar una lista de clases abstractas candidatas a refactorizar de acuerdo al diagrama de la Figura 2.
- **Cálculo de Métricas:** tarea encargada de la ejecución de las mediciones de abstracción y polimorfismo que contenga el proyecto o sistema a refactorizar, esto a través del uso de las métricas de *Abstracción* y *Factor de Polimorfismo*.
- **Refactorización:** recibe como entrada la lista de clases abstractas candidatas a refactorizar, el método de refactorización al recibir esta lista de clases abstractas, a través del candidateo, realiza la conversión de estas clases abstractas a interfaces.
- **Generación de Código:** última fase del proceso de refactorización, en donde se toman las interfaces creadas en la fase de refactorización y se produce como resultado un código equivalente al que se contiene en el proyecto inicial.

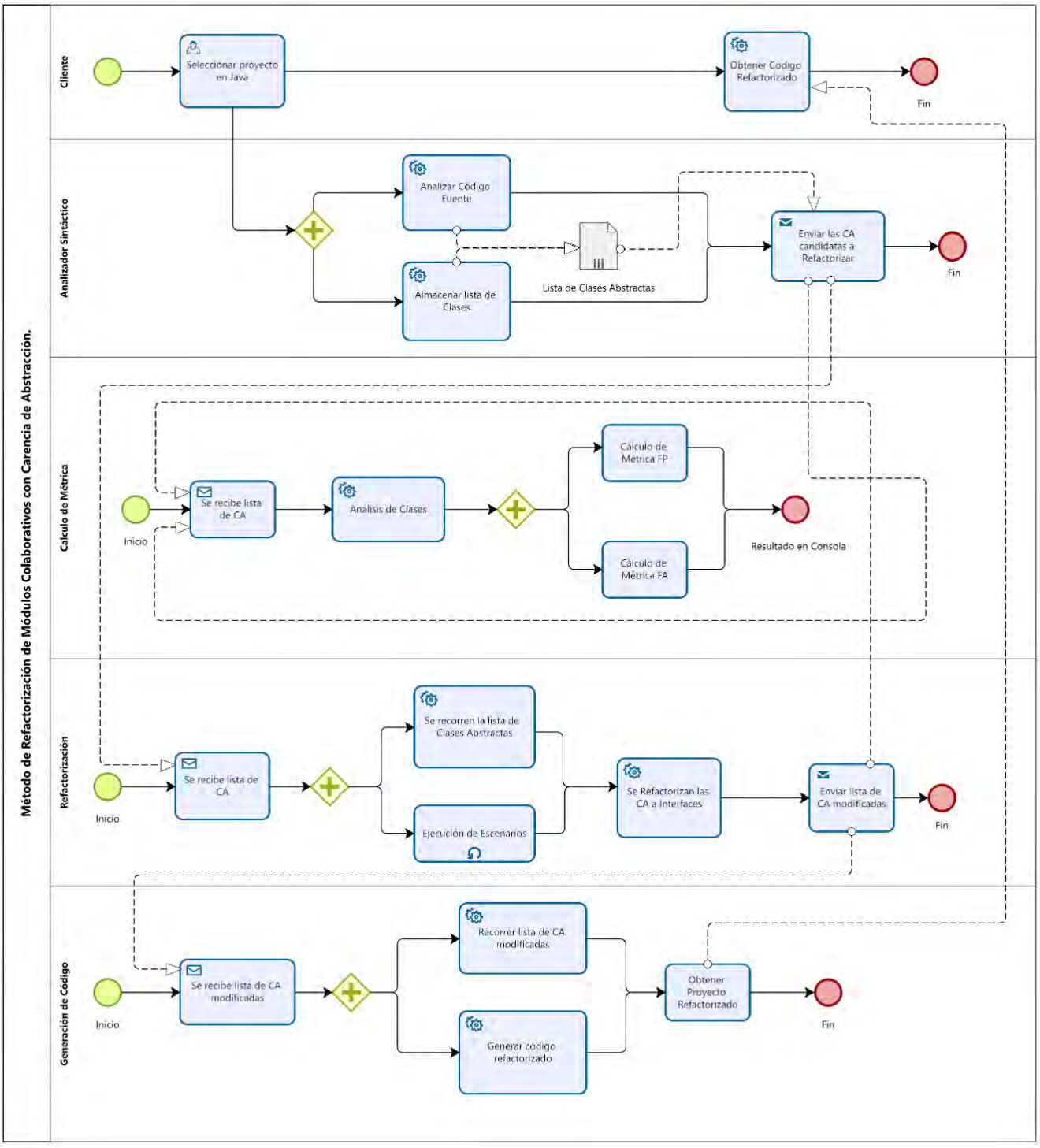


Figura 3 Diagrama BPMN del Método de Re-factorización de Módulos Colaborativos con Carencia de Abstracción.

5.- Diseño de la herramienta

El software desarrollado para realizar la refactorización de manera automática ha sido diseñado bajo el estándar UML. Se utilizan diagramas de casos de uso, diagrama de secuencias y diagramas de clases.

5.1.- Diagrama de Caso de Uso General del Sistema de Refactorización.

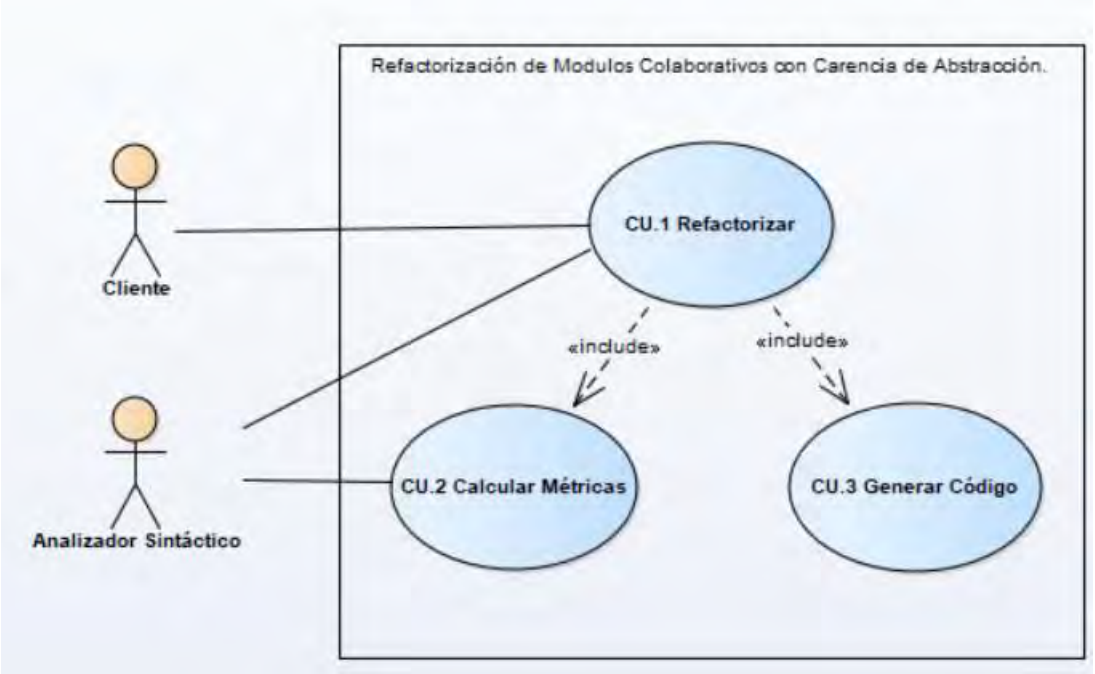


Figura 4 Caso de uso general del método de refactorización de clases base y clases abstractas.

	CU.1 – Refactorizar.
Descripción:	En este caso de uso la refactorización se encarga de transformar el código de las clases base y clases abstractas a interfaces, mejorando la flexibilidad de la arquitectura del código legado.
Actor primario:	Cliente.
Actor secundario:	Analizador Sintáctico.
Precondiciones:	El cliente debe contar con un proyecto candidato a refactorizar escrito en lenguaje Java. El proyecto a refactorizar debe estar libre de errores y fallas.
Escenario de éxito principal:	1. El cliente selecciona el proyecto escrito en lenguaje Java al cual se le desea aplicar la refactorización.

	2. El analizador sintáctico inicia el proceso de análisis del código legado para almacenar la información requerida para la refactorización. 3. El sistema realiza el cálculo de las métricas para evaluar si la arquitectura requiere ser refactorizada o no, en este escenario se presenta el mejor grado de abstracción por lo que no se requiere la refactorización. 4. Termina el caso de uso.
Escenario Alternativo 1:	1. El cliente escoge el proyecto escrito en lenguaje Java al cual se desea aplicar la refactorización. 2. El analizador sintáctico inicia el proceso de análisis del código legado para almacenar la información requerida para la refactorización y cálculo de métricas. 3. El sistema realiza el cálculo de las métricas para evaluar si la arquitectura requiere ser refactorizada o no, en este escenario se presenta la oportunidad de refactorización dado los valores obtenidos. 4. El sistema continúa con el proceso de transformación de clases base y/o clases abstractas hacia interfaces, según los diferentes escenarios analizados. 5. El sistema hace un recálculo de métricas para estimar la mejora en la arquitectura de clases del sistema original, se eje. 6. Termina el caso de uso.
Postcondiciones:	El código generado es equivalente en funcionalidad al código original.
Escenario de fracaso 1:	1. El cliente escoge el proyecto escrito en lenguaje Java al cual se desea aplicar la refactorización. 2. El analizador sintáctico inicia el proceso de análisis del código legado para

	almacenar la información requerida para la refactorización y cálculo de métricas. 3. El sistema hace el cálculo de métricas y se presenta la oportunidad de refactorización dado a los valores obtenidos. 4. La arquitectura presenta un escenario no considerado y el sistema no lo reconoce, por lo que procede a continuar buscando escenarios considerados. 5. Termina caso de uso.

	CU.2 – Calcular Métricas.
Descripción:	En este caso de uso se utiliza el modelo de datos obtenido del analizador sintáctico para realizar el cálculo de las métricas correspondientes a la identificación de la carencia de abstracción y de la proporción del factor de polimorfismo presente en la arquitectura.
Actor primario:	Analizador Sintáctico.
Precondiciones:	Contar con el modelo de objetos poblado de la información requerida para la refactorización, generado a partir del análisis sintáctico del código original
Escenario de éxito principal:	1. Identificar los criterios requeridos para el cálculo de las métricas FA y FP. 2. Calcular métricas de FA y FP. 3. Termina el caso de uso.
Postcondiciones:	El resultado de los cálculos automatizados obtenidos es correspondiente a los cálculos manuales de las métricas FA y FP.
Escenario de fracaso 1:	1. La arquitectura no presenta herencia, por lo tanto, no cuenta con abstracción, ni polimorfismo en su estructura. 2. Termina caso de uso.

	CU.3 – Generar Código.
Descripción:	En este caso de uso se utiliza el modelo de datos refactorizado del CU.1 para generar el

	código refactorizado a través del uso de la plantilla “StringTemplate”.
Actor primario:	Cliente.
Actor secundario:	Analizador sintáctico.
Precondiciones:	Contar con el modelo de datos refactorizado derivado del CU.1.
Escenario de éxito principal:	1. Se recorre el modelo de datos y se llena la información de la plantilla “StringTemplate”. 2. Se genera el código refactorizado en la carpeta destino del proyecto. 3. Termina caso de uso.
Postcondiciones:	El código generado es equivalente a la funcionalidad del código original.

5.2.- Diagrama de Caso de Uso del Analizador Sintáctico.

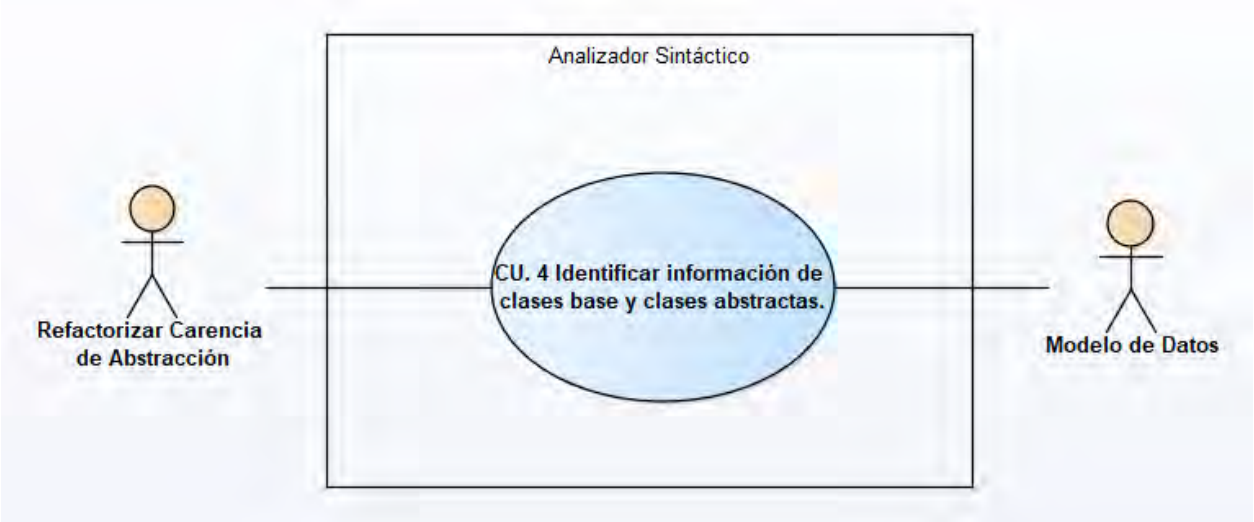


Figura 5 Diagrama de Caso de Uso CU.4 – Identificar información de clases base y clases abstractas.

	CU.4 – Identificar información de clases base y clases abstractas.
Descripción:	El analizador sintáctico se encarga de analizar localizar en el código las clases base y las clases abstractas que contiene el proyecto con código legado en lenguaje Java proveniente del CU.1 con la finalidad de almacenar información de las clases candidatas en el modelo de datos para aplicar el método de refactorización, dentro de éste. También se analizan todos los métodos

	públicos abstractos los cuales son parte importante en las clases abstractas para refactorizarlas en interfaces.
Actor primario:	Refactorizar Carencia de Abstracción.
Precondiciones:	La sintaxis del lenguaje Java debe estar en su versión 8.
Escenario de éxito principal:	1. El sistema de refactorización solicita al analizador el análisis sintáctico y semántico sobre el proyecto elegido por el cliente. 2. El analizador sintáctico crea una lista de clases base y clases abstractas mediante el candidateo semántico a ser refactorizadas por interfaces haciendo uso del modelo de datos. 3. Se almacena en el modelo de datos la información de la lista de clases para realizar la refactorización y el cálculo de métricas. 4. Termina caso de uso.
Postcondiciones:	El análisis crea una lista de clases abstractas candidatas a refactorizar.
Escenario de fracaso 1:	1. El sistema de refactorización solicita al analizador el análisis sintáctico y semántico sobre el proyecto elegido por el cliente. 2. Durante el análisis se detecta que el código fuente escogido por el cliente presenta errores de sintaxis. 3. El sistema termina su ejecución. 4. Termina caso de uso.

5.3.- Diagrama de Secuencia del Sistema de Refactorización

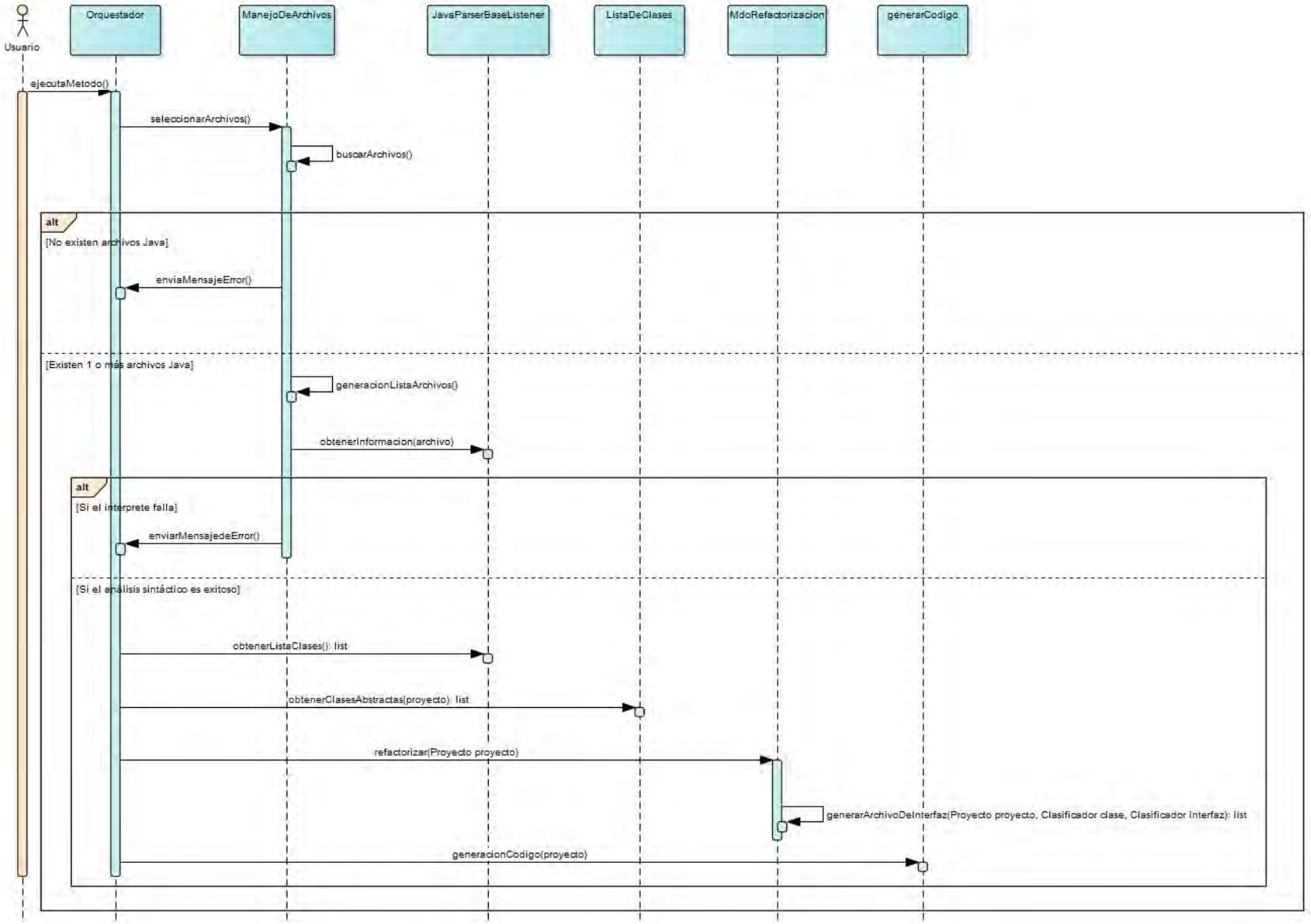


Figura 6 Diagrama de Secuencia del Sistema de Refactorización.

El diagrama de secuencia que se muestra en la Figura 6 muestra las secuencias del sistema de refactorización, y se describe de la siguiente manera:

1. El cliente ejecuta el método de refactorización.

2. El cliente se encarga de seleccionar los archivos Java a ser refactorizados.

3. Si el cliente selecciona una carpeta, el sistema se encarga de buscar todos los archivos con el código escrito en lenguaje Java.

4. Si el sistema no encuentra ningún archivo con código escrito en lenguaje Java el sistema envía un mensaje de error.

5. Si el sistema encuentra uno o más archivos con código escrito en lenguaje Java el sistema obtiene una lista de clases.

6. El sistema obtiene la información necesaria de cada una de las clases abstractas y de las clases base para el proceso de refactorización.

7. Se realiza el cálculo de métricas para evaluar la procedencia de la refactorización.

8. Se validan todos los criterios para refactorizar clases abstractas o clases base a interfaz.

9. Se aplica el método de refactorización sobre las clases abstractas o clases base para convertirlas a interfaces.

10. El sistema actualiza la lista de clientes que contenga el proyecto y crea correctamente las dependencias del proyecto refactorizado.

11. Se genera el archivo de la interfaz creada a partir de la refactorización, agregándola al mapa de clasificadores del archivo.

12. Teniendo toda esta información recabada se genera el código refactorizado equivalente al código original.

13. Se realiza un recálculo de métricas para evaluar la mejora en la arquitectura de clases después de haber sido refactorizada.

14. El sistema almacena el proyecto refactorizado en la carpeta denominada **<NombreDelProyecto_Refactorizado>** con los archivos refactorizados.

5.4.- Diagrama de Secuencia del Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción.

Lo anteriormente expuesto, de la Figura 6, muestra el funcionamiento general del método de refactorización, desde su ejecución hasta la generación del código del proyecto refactorizado. Para comprender de una mejor forma el procedimiento del método de refactorización de módulos colaborativos con carencia de abstracciones, se diseñó específicamente un diagrama de secuencias en donde se muestra la interacción de las clases que comprenden el método de refactorización. En la Figura 7 se presenta dicho diagrama de secuencia del método desarrollado.

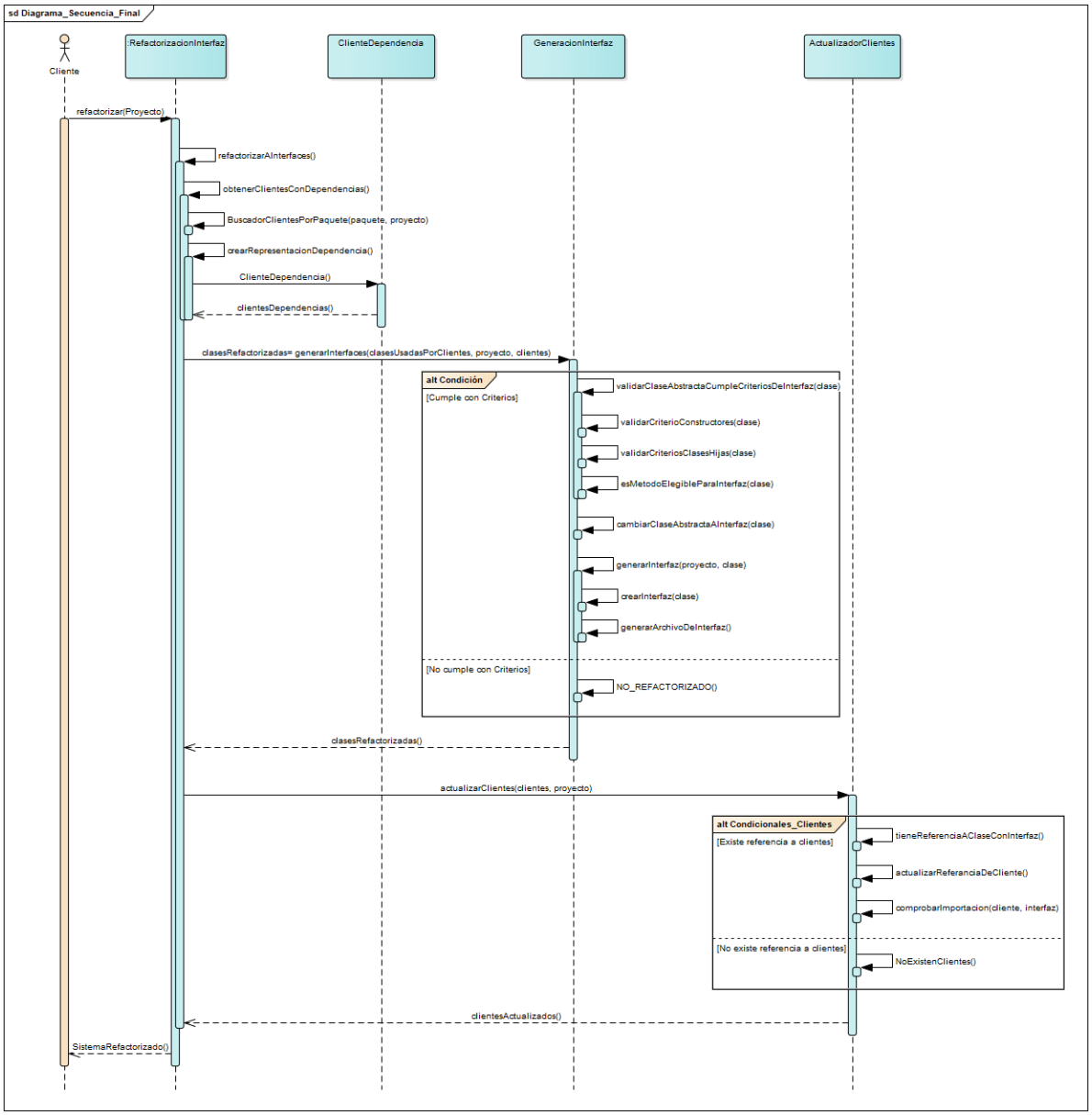


Figura 7 Diagrama de Secuencia del Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción.

El diagrama de secuencia se describe de la siguiente manera, como primera parte se muestran las primeras secuencias del método:

1. El cliente ejecuta el método refactorizar(Proyecto), este método llama internamente al método refactorizarAInterfaces().

2. Se muestra el mensaje refactorizarAInterfaces(proyecto: Proyecto), que es el método principal para realizar la refactorización a interfaces del proyecto.

3. El método obtenerClientesConDependencias() interactúa con el objeto BuscadorClientesPorPaquete, el cual se encarga de obtener a los clientes que contenga el paquete.

4. Después de obtener la lista de clientes con dependencias, se muestra el mensaje crearRepresentacionDependencia() que crea una instancia de la clase interna ClienteDependencia y establece las dependencias del cliente.

5. El método refactorizarAInterfaces() crea una instancia de la clase GeneracionInterfaz y se muestra el mensaje generarInterfaces() enviado a GeneracionInterfaz. Este método se encarga de generar las interfaces para las clases usadas por los clientes y devuelve un conjunto de clases refactorizadas.

6. Dentro de la clase Generación Interfaz, se realizan las validaciones necesarias para candidatear a aquellas clases abstractas o clases base que deben ser refactorizadas a una interfaz, para ello se utilizan los métodos validarClaseAbstractaCumpleCriteriosDeInterfaz, validarCriteriosClasesHijas, validarCriteriosConstructoresHijos, validarCriterioConstructores y esMetodoElegibleParaInterfaz.

7. Si las clases a refactorizar cumplen con los criterios de refactorización se ejecuta el método cambiarClaseAbstractaAInterfaz, en donde se realiza lo siguiente:

- Se establece el tipo de la clase como TipoClasificador.INTERFAZ mediante la instrucción clase.setTipo(TipoClasificador.INTERFAZ).
- Se reemplaza la lista de hijos de la clase con una nueva lista vacía mediante clase.setHijos(new ArrayList<Clasificador>()). Esto implica eliminar todos los hijos de la clase.
- Se itera sobre cada uno de los hijos en la lista hijos utilizando un bucle for-each.
- Para cada hijo, se realiza lo siguiente:
- Se establece el padre del hijo como null mediante hijo.padre = null, lo que significa que ya no tiene un padre.
- Se agrega la clase como interfaz al hijo mediante hijo.agregarInterfaz(clase). Esto establece una relación de implementación entre la clase y el hijo.

- Se agrega el hijo como implementador de la clase mediante clase.agregarImplementador(hijo). Esto establece que el hijo implementa la interfaz representada por la clase.
- Finalmente, se agrega la clase al conjunto clasesRefactorizadas mediante clasesRefactorizadas.add(clase).

8. Si ninguna de las condiciones anteriores se cumple, se establece el estado del objeto 'metodo' a 'NO_REFACTORIZADO'.

9. El método 'generarInterfaz' recibe un objeto 'proyecto' y un clasificador 'clase', dentro de este método se crea la interfaz llamando al método 'crearInterfaz' y genera el archivo de la interfaz haciendo uso del método 'generarArchivoDeInterfaz', además, agrega la interfaz al clasificador actual y al mismo tiempo al conjunto de 'clasesRefactorizadas'.

10. El método 'generarArchivoDeInterfaz' recibe un objeto 'Proyecto', un clasificador 'clase' y un clasificador de interfaz 'interfaz', y se encarga de generar un nuevo archivo de interfaz, en donde establece su nombre, ruta y paquetes, luego, se copian las importaciones del archivo original de la clase.

11. Una vez generados los archivos de interfaz se devuelve un conjunto de clases refactorizadas.

12. A continuación, se muestra el mensaje 'actualizarClientes' enviado a 'ActualizadorClientes', el cual es encargado de actualizar a los clientes de las clases refactorizadas.

13. Dentro del método 'tieneReferenciaAClaseConInterfaz', verifica si algún elemento en el conjunto de 'clasesRefactorizadas' tiene el mismo nombre que la referencia dada, si el valor devuelto es true, indica que hay una referencia a una clase con interfaz dentro del conjunto de las 'clasesRefactorizadas', de lo contrario devuelve un false.

14. Para colocar las referencias actualizadas de los clientes en las interfaces creadas, el método 'actualizarReferenciaDeCliente', crea la instancia de 'ReferenciaDeclaracion' llamada 'refD' utilizando el parámetro 'nuevaReferencia', estableciendo diferentes atributos tomando los valores correspondientes de la referencia actual, y finalmente devuelve la instancia 'refD' actualizada.

15. Por último, para comprobar las importaciones de los clientes, a través del método 'comprobarImportación' se verifica que el cliente ha importado la interfaz, utilizando el método 'esImportado' del objeto 'Archivo'. En caso contrario de no haber importado la interfaz, se crea una instancia de 'Importacion' llamada 'importaciónInterfaz', donde se establecen los diferentes atributos de importacionInterfaz, como clasificador, paquetes y

cadenaClasificador, utilizando valores del interfaz. Finalmente, agrega importacionInterfaz a las importaciones del archivo del cliente.

16. Agregadas las importaciones de los clientes del proyecto, se devuelven al proyecto refactorizado los clientes actualizados con sus importaciones correctas.

5.5.- Diagrama de Clases del Método de Refactorización.

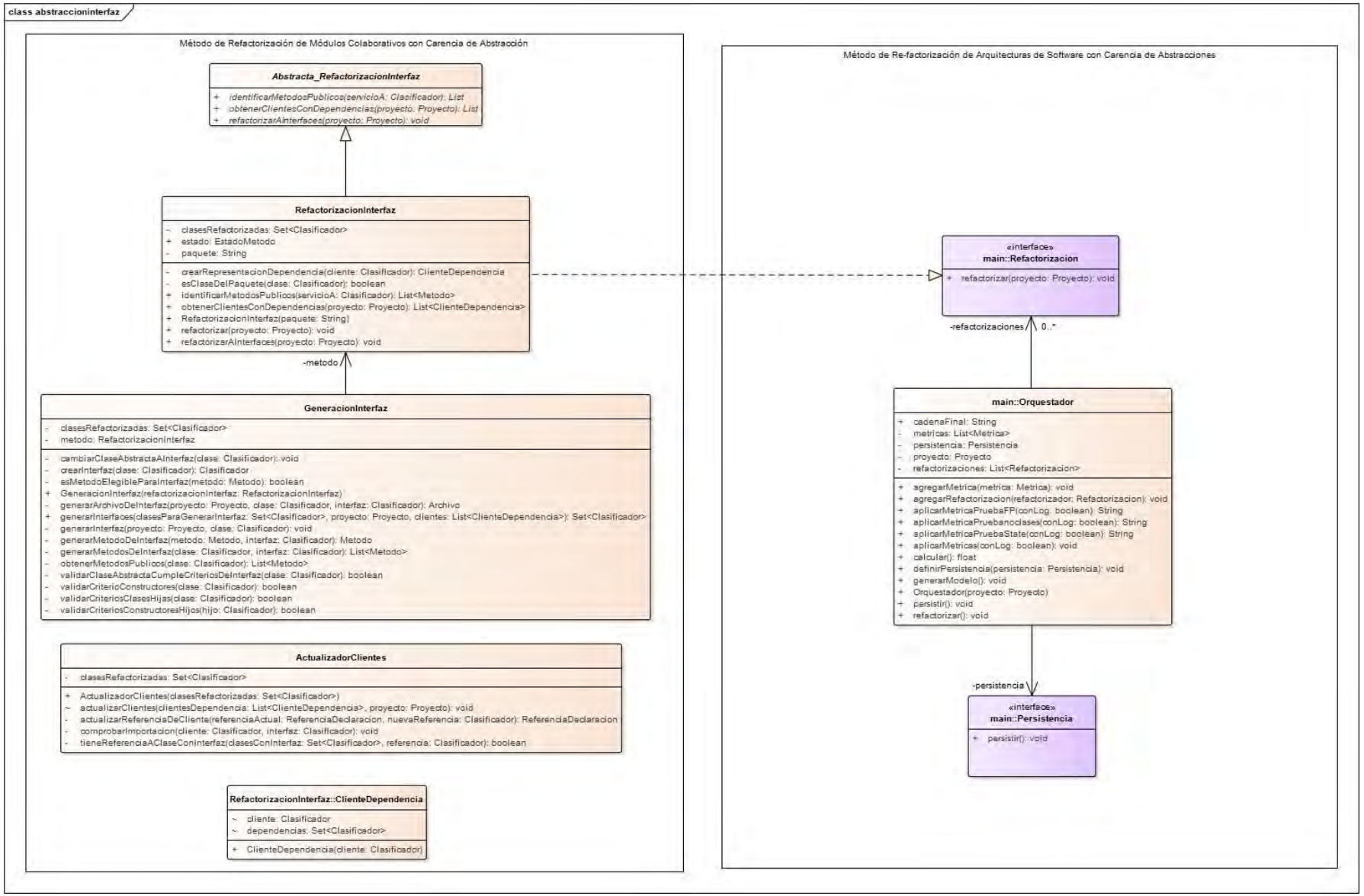


Figura 8 Diagrama de Clases del paquete abstracciónInterfaz

En la Figura 8, del lado izquierdo se muestra la arquitectura de clases del método de refactorización de módulos colaborativos con carencia de abstracción. Cabe mencionar que inicialmente está arquitectura no contaba con abstracción, pero aplicando los métodos de refactorización desarrollados en CENIDET se pudo agregar abstracción a esta arquitectura demostrando de esta manera que los métodos funcionan de forma correcta.

La clase 'RefactorizacionInterfaz' tanto hereda de su clase abstracta del mismo paquete como también implementa la interfaz 'Refactorizacion' para utilizar su método 'refactorizar', la cual, junto con la clase Orquestador son los encargados de aplicar las métricas, generar el modelo y refactorizar las clases.

6.- Pruebas

6.1.- Análisis de Escenarios.

Los escenarios son diferentes situaciones que pueden suceder en la vida real, en éstos se especifica cómo el sistema debe ser usado. Estos resultan ser contenedores de la mayoría de los requisitos del sistema de software. Muchas de las buenas prácticas o prácticas recomendadas en el proceso de desarrollo de software se basan en la existencia de un documento de requisitos a partir de escenarios ya construidos.

Para comprender de mejor forma los escenarios con los cuales se están trabajando para este método de refactorización, se realizaron diagramas de clases en UML, los cuales son un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando cada una de sus clases, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.

6.1.1.- Escenario 1.

Arquitectura Inicial.

En este primer escenario cómo se observa en la Figura 9 una arquitectura de software puede carecer de abstracción haciendo que sea rígida a cambios de comportamiento, y no se pueda extender a nuevas funcionalidades o comportamientos, violando de esta manera los principios de "*Abierto - Cerrado*" e "*Inversión de Dependencias*".

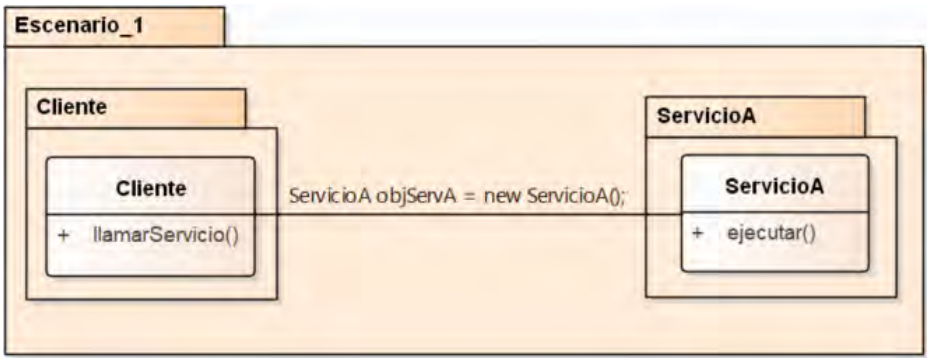


Figura 9 Arquitectura de software rígida sin abstracción.

Código Legado – Escenario 1

Antes de Refactorizar.

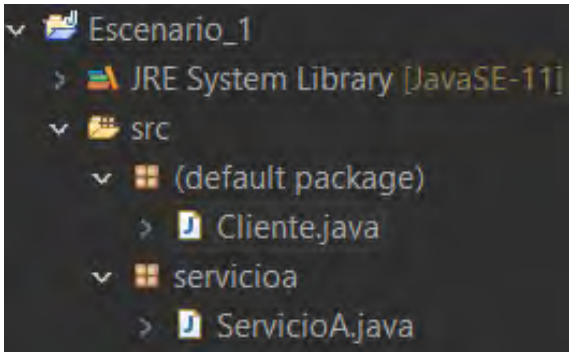


Figura 10 Proyecto Escenario 1, estructura original antes de refactorizar.

La Figura 10 nos muestra la arquitectura original del proyecto antes de aplicar el método de refactorización. A continuación, en la Figura 11 se muestra el código de la clase Cliente.

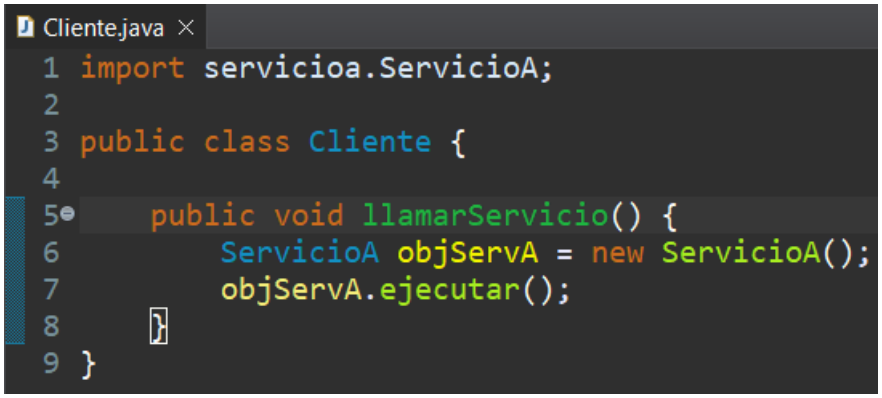


Figura 11 Código original de Clase Cliente instanciando una variable de referencia sobre la clase ServicioA

Dentro de clase Cliente que es una clase concreta, podemos observar que hay una instancia de referencia (objServA) de tipo ServicioA. El tipo estático de este objeto

es “ServicioA” y no hace transformación de tipos dinámicamente. Es decir, conserva al objeto con el mismo tipo, conforme a su declaración.

El método “llamarServicio” de esta clase invoca directamente al método “ejecutar” del objeto “objServA” Figura 12. Como consecuencia de ello podemos observar que este tipo de comportamiento no es correcto ya que la arquitectura de clases que otorga el servicio es rígida a cambios de comportamiento.

ServicioA.java

```
1 package servicioa;
2
3 public class ServicioA {
4     public void ejecutar() {
5         System.out.println("Hola mundo!");
6     }
7 }
```

Figura 12 Clase ServicioA implementadora del método ejecutar().

Arquitectura Esperada.

La solución a este escenario es crear una nueva interfaz como se muestra en la Figura 13, con el propósito de hacer que la arquitectura sea flexible a cambios de comportamiento en tiempo de ejecución y de que sea posible extender su funcionamiento con nuevas funcionalidades o comportamientos. Respetando los principios de abierto – cerrado y de inversión de dependencias.

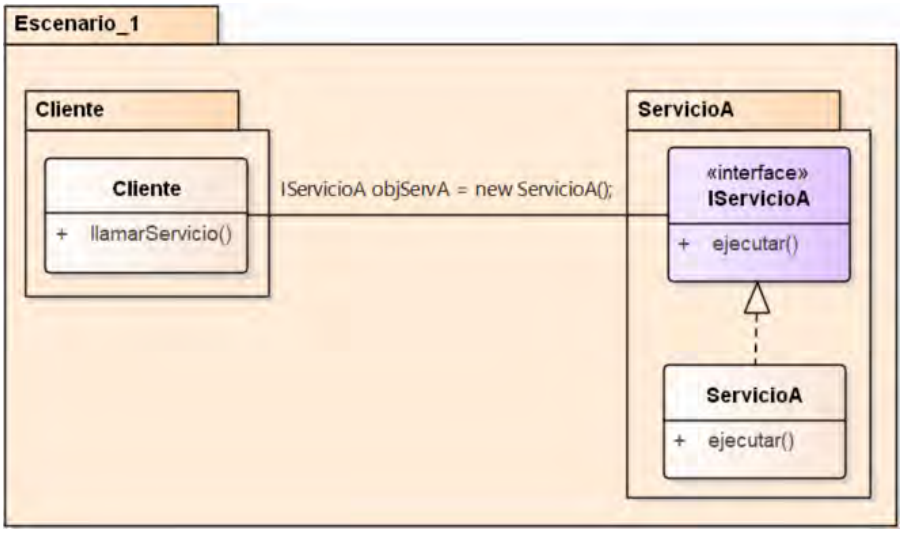


Figura 13 Refactorización a Arquitectura rígida implementando una Interfaz

Código Refactorizado – Escenario 1

Después de Refactorizar.

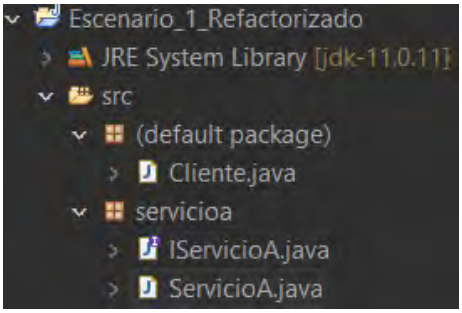


Figura 14 Proyecto de Escenario 1 Refactorizado.

Como se puede observar, una vez aplicada la refactorización se crea un nuevo proyecto al cual se le agrega la etiqueta “Refactorizado” para poder identificarlo dentro de nuestra carpeta de proyectos. De igual forma, podemos notar la refactorización dentro del paquete de clases del servicio en donde podemos encontrar que la clase ServicioA ha sido refactorizada por una interfaz, el comportamiento no cambia, pero al hacer está refactorización estamos respetando el principio de abierto – cerrado, ya que como lo menciona Bertrand Meyer(Meyer, 1997.) “Las entidades de Software deben estar abiertas para su extensión, pero cerradas para su modificación”. El código después de haber hecho la refactorización se muestra de la siguiente forma:

Cliente.java

```
1
2  import servicioa.ServicioA;
3  import servicioa.IServicioA;
4
5  public class Cliente {
6      public void llamarServicio() {
7          IServicioA objServA = new ServicioA();
8          objServA.ejecutar();
9      }
10 }
```

Figura 15 Clase Cliente después de haber sido refactorizada.

Dentro de la clase Cliente.java se puede observar que el nombre del tipo estático del objeto “objServA” es “IservicioA”, pero cambia su tipo dinámicamente, al tipo “ServicioA”, esto es aplicar la asociación dinámica de tipos y por lo tanto se habilita el polimorfismo. Se crea la interfaz "IservicioA” y la clase original “ServicioA” pasa a ser una subclase de esta interfaz, la cual implementa al método “ejecutar”.

IServicioA.java

```java
package servicioa;

public interface IServicioA {
    void ejecutar();
}
```

Figura 16 Interfaz IServicioA en donde se define el método ejecutar.

ServicioA.java

```java
package servicioa;

public class ServicioA implements IServicioA {
    @Override
    public void ejecutar() {
        System.out.println("Hola mundo!");
    }
}
```

Figura 17 Clase ServicioA en donde el método ejecutar pasa a ser sobrescrito.

Aplicando las métricas de abstracción y polimorfismo sobre este proyecto se obtuvieron los siguientes resultados.

Tabla 1 Resultado del cálculo de métricas Escenario 1

	Escenario 1	
	Antes de Refactorizar	Después de Refactorizar
FA	0.0	0.33333334
FP	0.0	1.0

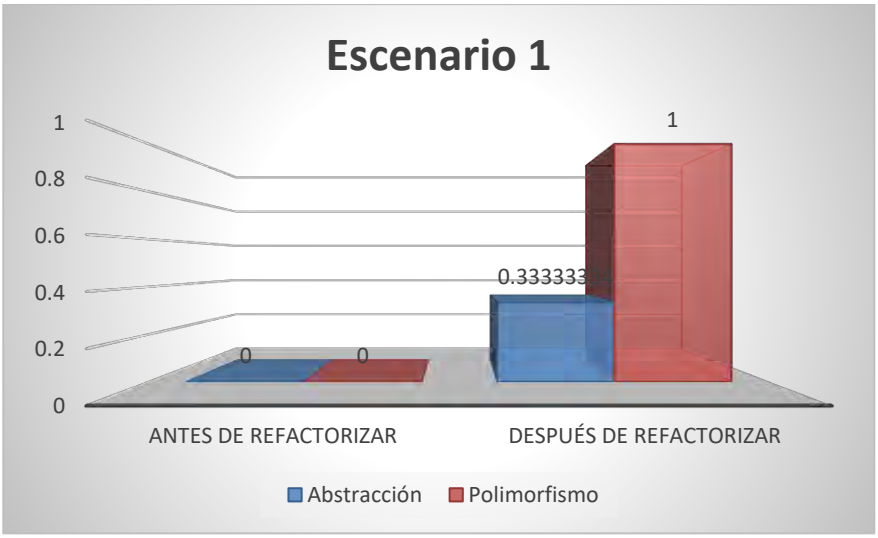


Figura 18 Resultados de las métricas antes y después de refactorizar.

Como se puede observar, para este primer escenario las propiedades de abstracción y polimorfismo aumentaron su valor inicial, lo que significa que la refactorización fue aplicada correctamente sobre el código legado, y ahora obtenemos un código flexible y extensible a cambios de comportamiento en el sistema.

Arquitectura Obtenida

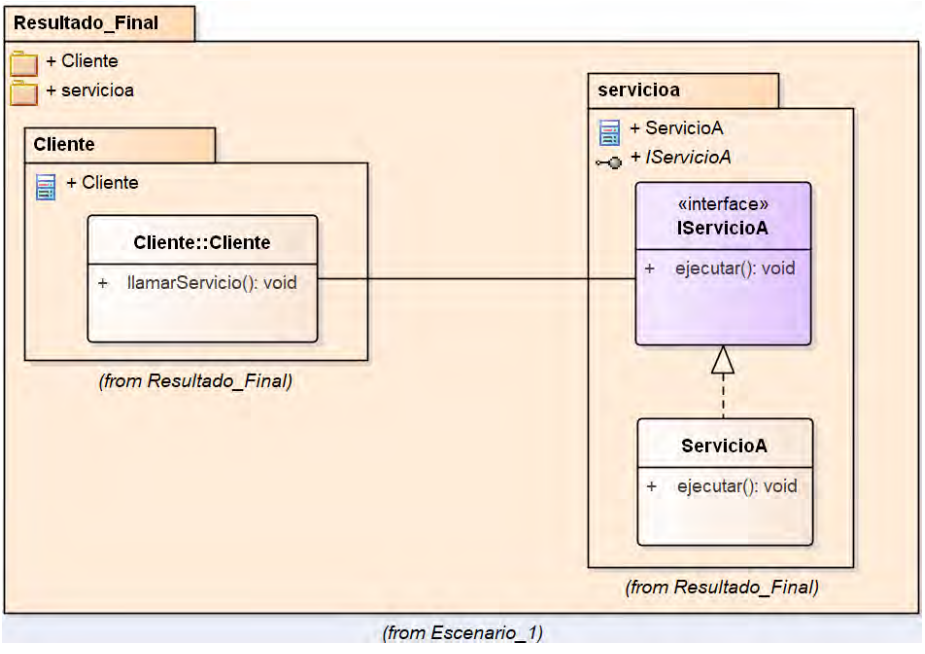


Figura 19 Arquitectura de clases obtenida después de aplicar la refactorización

6.1.2.- Escenario 2.

Arquitectura Inicial.

Como segundo escenario se cuenta con una arquitectura en la cual se tiene implementada una clase abstracta que sólo define métodos abstractos públicos como se ve en la Figura 20. Para el método de refactorización ésta es una clase que puede ser refactorizada a una interfaz ya que como se define en los criterios, una clase abstracta que solo defina métodos públicos abstractos deberá ser refactorizada a una interfaz, esto permanece sin alterar el comportamiento de la clase que ha sido refactorizada.

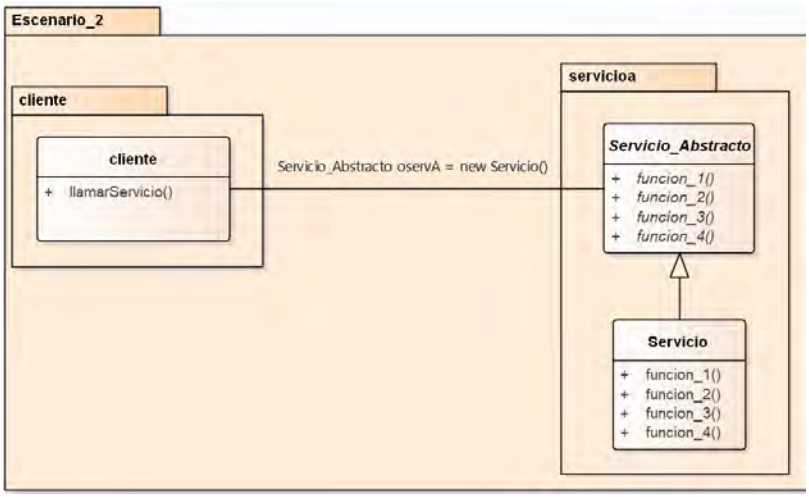


Figura 20 Arquitectura con una clase abstracta que puede ser refactorizada a Interfaz

Código Legado – Escenario 2
Antes de Refactorizar.

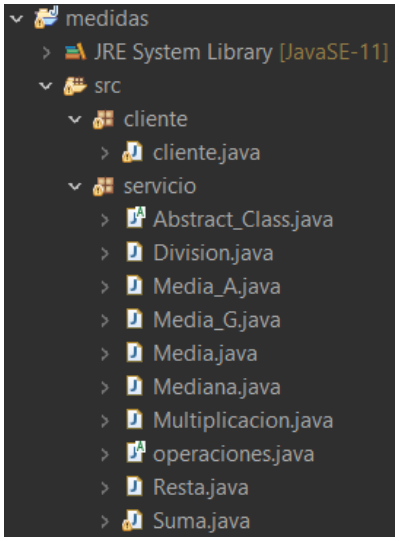


Figura 21 Proyecto Escenario 2 Antes de Refactorizar.

Para está refactorización las clases abstractas “Abstract_Class” y “operaciones” a ser refactorizadas deben implementar métodos públicos abstractos que estén definidos en ella como se muestra en la Figura 22, de igual forma sus clases derivadas son aquellas que heredan (extends) los métodos de la clase abstracta, los cuales se sobrescriben en estas mismas clases derivadas.

Abstract_Class.java

```
1  package servicio;
2
3  public abstract class Abstract_Class {
4
5      public abstract void MTC();
6      public abstract void calcular();
7
8  }
9
```

operaciones.java

```
1  package servicio;
2
3  public abstract class operaciones {
4
5      public abstract void suma();
6      public abstract void resta();
7      public abstract void multiplicacion();
8      public abstract void division();
9
10 }
```

Figura 22 Clases abstractaa Abstract_Class y operaciones con métodos definidos como públicos abstractos.

Arquitectura Esperada.

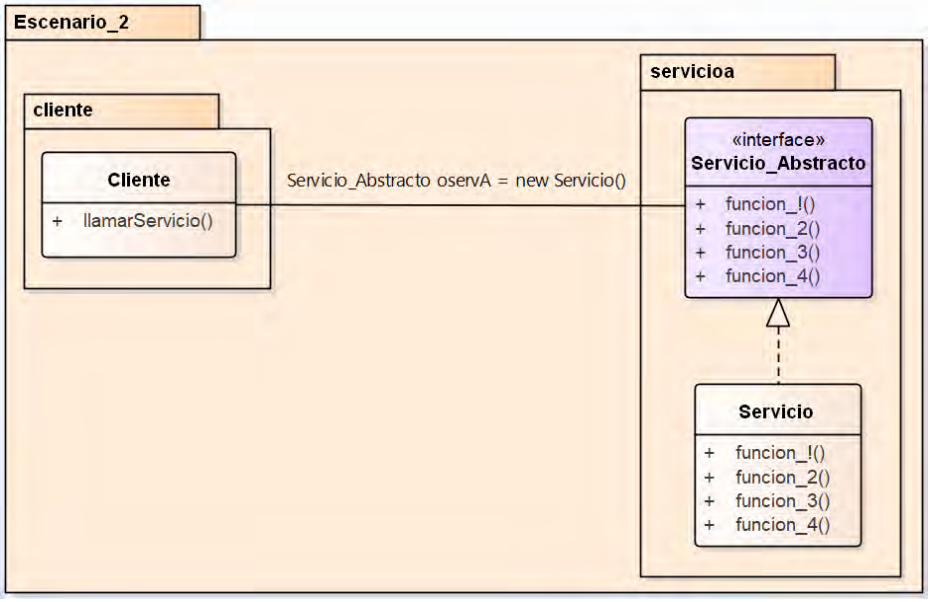


Figura 23 Arquitectura esperada una vez hecha la refactorización.

Código Refactorizado – Escenario 2

Una vez que se aplica la refactorización a las clases abstractas “Abstracts_Class” y “operaciones” estás cambiarán a ser una interfaz como se muestra en la Figura 16, en donde los métodos definidos en ella seguirán teniendo el mismo comportamiento, pero con diferente forma de implementación ya que, aunque aún sigan siendo métodos abstractos estos cambian a ser de tipo void.

La estructura del proyecto cambia a la forma que se muestra en la Figura 24.

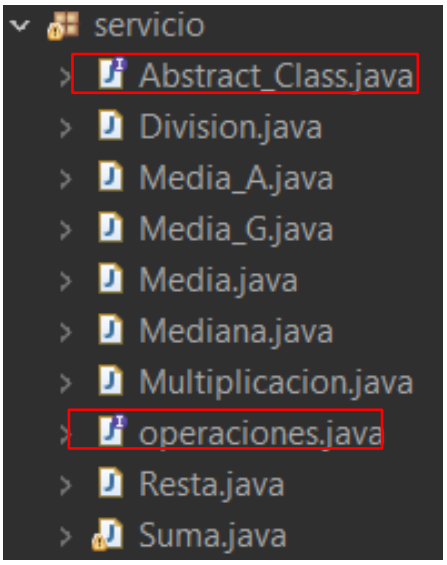


Figura 24 Proyecto Escenario_2_Refactorizado

cliente.java

```
1  package cliente;
2
3  import servicio.Abstract_Class;
8  public class cliente {
9      public static void llamarServicioMedia() {
10         Abstract_Class oServicio = new Media();
11         oServicio.calcular();
12     }
13
14     public static void suma() {
15         operaciones oServicio = new Suma();
16         oServicio.suma();
17     }
18
19     public static void main(String[] args) {
20         llamarServicioMedia();
21         suma();
22     }
23 }
```

Figura 25 Clase Cliente refactorizada, la instancia sigue tomando el nombre de la clase abstracta que ha sido refactorizada a una interfaz.

Abstract_Class.java

```
1  package servicio;
2
3  public interface Abstract_Class {
4      void MTC();
5
6      void calcular();
7  }
```

operaciones.java

```
1  package servicio;
2
3  public interface operaciones {
4      void suma();
5
6      void resta();
7
8      void multiplicacion();
9
10     void division();
11 }
```

Figura 26 Interfaces creadas a partir de haber aplicado la refactorización.

Aplicando las métricas de abstracción y polimorfismo sobre este proyecto se obtuvieron los siguientes resultados.

Tabla 2 Resultado del cálculo de métricas Escenario 2

	Escenario 2	
	Antes de Refactorizar	Después de Refactorizar
FA	0.18181819	0.18181819
FP	1.0	1.0

En este segundo escenario, los valores antes y después de la refactorización son idénticos, esto a causa de que una interfaz al igual que una clase abstracta implementa todos sus métodos declarados como abstractos y públicos, es por este motivo que los valores no cambian por esta gran similitud que hay en ellas.

Arquitectura Obtenida

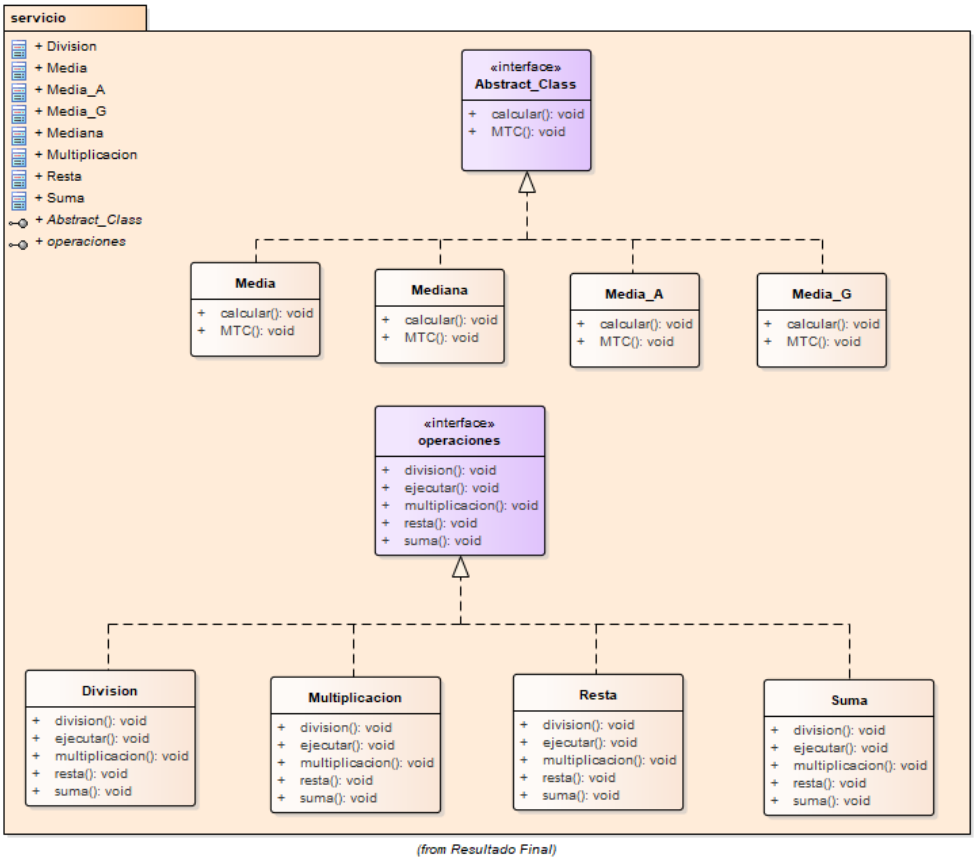


Figura 27 Arquitectura de clases obtenida después de aplicar la refactorización.

Lo anteriormente expuesto, es la arquitectura obtenida una vez que se ha aplicado la refactorización, en este diagrama de clases se puede observar que se sigue

respetando el principio de inversión de dependencias el cual se ha propuesto en la tesis de la cual se extiende este proyecto donde los módulos de alto nivel no deberían depender de los de bajo nivel, ambos deberían depender de abstracciones. Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones.

6.1.3.- Escenario 3.

Para este escenario la refactorización no podría efectuarse, ya que como se plantea dentro de los criterios de este proyecto, una clase abstracta en la cual se definan métodos con calificadores de alcance no públicos y que éstos a la vez contengan cuerpo, no se realizará la refactorización. La clase abstracta seguirá teniendo la misma estructura Figura 19.

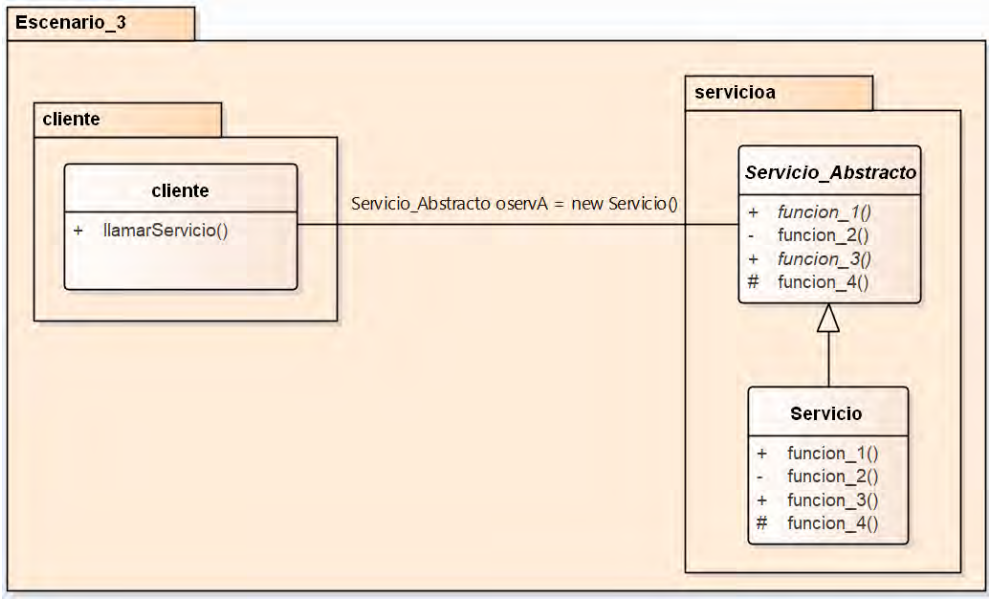


Figura 28 Clase abstracta a no refactorizar

Código – Escenario 3.

El código para este escenario no se refactoriza ya que como se había explicado anteriormente, cuando la clase abstracta contiene métodos que no son públicos e implementan código no se debe de refactorizar.

```java
package cliente;

import servicioa.Servicio;
public class Cliente {
    public void llamarServicio() {
        Servicio_Abstracto oservA = new Servicio();
        oservA.funcion_1();
    }
}
```

Figura 29 Clase Cliente, no contiene cambios ya que no se aplica la refactorización.

```java
package servicioa;

public abstract class Servicio_Abstracto {

    public abstract void funcion_1();

    private void funcion_2() {
        System.out.println("Bloque de c?digo");
    }

    public abstract void funcion_3();

    protected void funcion_4() {
        this.funcion_2();
    }

}
```

Figura 30 Clase abstracta no refactorizada ya que no cumple con los criterios de refactorización.

6.1.4.- Escenario 4 Funciones Virtuales.

Una función o método virtual, es una función cuyo comportamiento, al ser “virtual”, es determinado por la definición de una función con la misma cabecera en ~~alguna de~~ sus clases derivadas. El concepto de función virtual es una parte muy importante sobre el factor polimorfismo en la programación orientada a objetos; en lenguaje Java, todos los métodos no privados y no finales son métodos virtuales.

Dentro de este escenario se muestra la arquitectura con una clase que implementa una función virtual Figura 31, por lo tanto, se tomaron en cuenta dos criterios para poder refactorizar este tipo de funciones, los cuales son:

- Si las funciones virtuales definidas dentro de una clase abstracta están sin cuerpo, pero se implementan con cuerpo las clases derivadas, se deberá refactorizar la clase abstracta a una interfaz.

- Si las funciones virtuales definidas dentro de una clase abstracta cuentan con cuerpo implementado, está no deberá de ser refactorizada con la finalidad de respetar los criterios de la refactorización y respetar los diferentes comportamientos de esas funciones en las clases derivadas.

Arquitectura Original

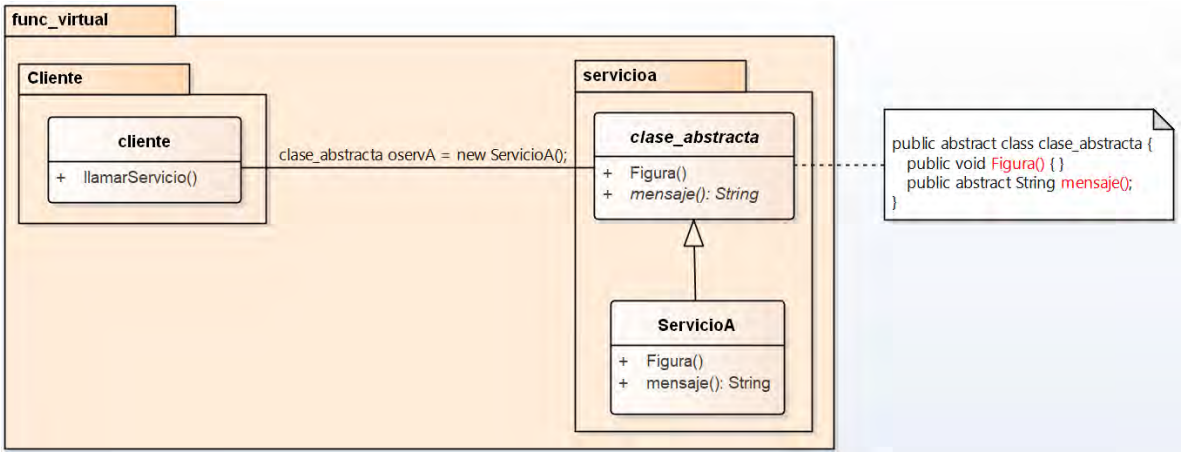


Figura 31 Arquitectura con función virtual antes de ser refactorizada, en la nota se aprecia que esta función virtual no contiene cuerpo lo cual es candidata para refactorizar.

Cómo ya fue mencionado al aplicar el método de refactorización sobre este escenario, la clase abstracta que implementa está función virtual es refactorizada por una interfaz, dado que las funciones o métodos de una interfaz en Java son virtuales por defecto ya que son públicos.

Código Legado – Escenario 4 Funciones Virtuales.
Antes de Refactorizar.

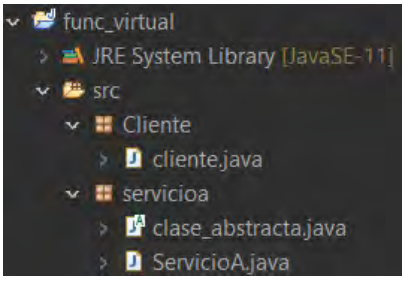


Figura 32 Proyecto Funciones Virtuales antes de ser refactorizado

Antes de aplicar la refactorización se observa que en la clase “clase_abstracta” Figura 33 se hace la definición de dos métodos, el método público abstracto ”mensaje” y el método público virtual “Figura”, el método virtual se hereda en la clase derivada “ServicioA” Figura 34, dentro de esta clase se observa que este método virtual contiene un cuerpo. La solución para resolver este escenario en específico es hacer la refactorización de la clase abstracta que contiene el método virtual por una interfaz, respetando la implementación de los métodos públicos abstractos sin causar alguna alteración en la funcionalidad del proyecto refactorizado.

clase_abstracta.java

```
1  package servicioa;
2
3  public abstract class clase_abstracta {
4
5      public void Figura() {
6
7      }
8
9      public abstract String mensaje();
10
11 }
```

Figura 33 Clase abstracta a refactorizar con una función virtual.

ServicioA.java

```
1  package servicioa;
2
3  public class ServicioA extends clase_abstracta{
4
5      public void Figura() {
6          System.out.println("HOLA MUNDO");
7      }
8
9      @Override
10     public String mensaje() {
11         // TODO Auto-generated method stub
12         return "HOLA";
13     }
14
15 }
```

Figura 34 Clase ServicioA que hereda los métodos de la clase abstracta.

Arquitectura Esperada.

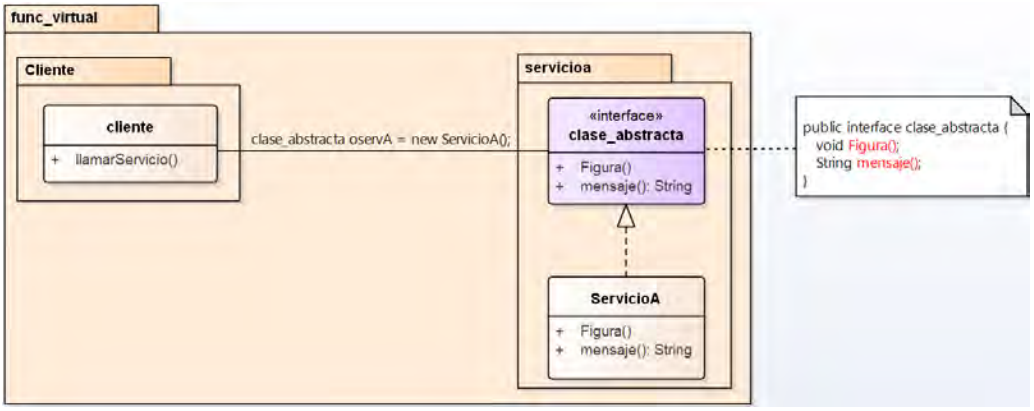


Figura 35 Refactorización de una clase abstracta con una función virtual a una Interfaz.

Código Refactorizado.

Al aplicar el método de refactorización sobre el proyecto creado para este escenario, se puede constatar en el código que la clase abstracta es refactorizada por una interfaz como se muestra en la Figura 36, removiendo el cuerpo vacío del método virtual, dejando solamente su implementación en la clase derivada “ServicioA”.

clase_abstracta.java

```
1 package servicioa;
2
3 public interface clase_abstracta {
4     void Figura();
5
6     String mensaje();
7 }
```

Figura 36 Interfaz creada al refactorizar la clase abstracta.

La clase, “ServicioA” queda definida de la misma manera, no se altera el comportamiento de los métodos que implementa, solamente podemos apreciar que la palabra reservada “*extends*”, la cual es utilizada para referir a la herencia que se hace de una clase abstracta, se modifica por la palabra reservada “*implements*” ya que ahora esta clase derivada implementa los métodos de la interfaz creada a partir de la refactorización.

ServicioA.java

```
1 package servicioa;
2
3 public class ServicioA implements clase_abstracta {
4     @Override
5     public void Figura() {
6         System.out.println("HOLA MUNDO");
7     }
8     @Override
9     public String mensaje() {
10        return "HOLA";
11    }
12 }
```

Figura 37 Clase ServicioA que ahora implementa una Interfaz.

Al ejecutar el método de refactorización, los valores obtenidos a partir de las métricas de Abstracción y Factor Polimorfismo son los siguientes:

Tabla 3 Resultado del cálculo de métricas Escenario 3

	Escenario 3	
	Antes de Refactorizar	Después de Refactorizar
FA	0.33333334	0.33333334
FP	1.0	1.0

Cómo se había mencionado anteriormente, los valores que se obtienen al aplicar el método de refactorización son iguales, ya que la métrica de Abstracción incluye dentro de ella el número total de clases abstractas como de interfaces, esto es debido a que tanto clases abstractas como interfaces son clases totalmente abstractas.

Arquitectura Obtenida.

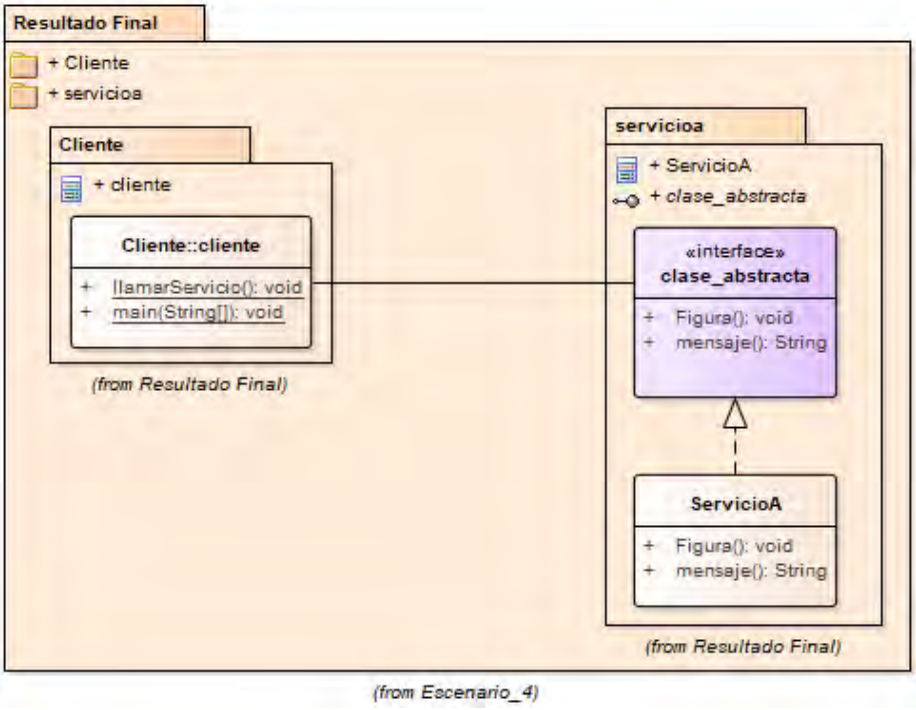


Figura 38 Arquitectura obtenida al aplicar la refactorización sobre un método virtual definido ahora en una interfaz

La arquitectura que se muestra en la Figura 38, nos indica que el método virtual sin cuerpo ahora pasa a ser un método abstracto respetando su implementación dentro de la clase derivada que lo implementa.

6.1.5.- Escenario 5 Constructores

En el transcurso del análisis de escenarios, se planteó este escenario enfocado a constructores generados por default, que en ocasiones no son implementados en Java por desarrolladores que comienzan a programar, este tipo de constructores no contienen parámetros y por ende no realizan alguna acción.

Para instanciar diferentes objetos, toda clase necesita de un constructor, es por ello que en Java los constructores por default son creados automáticamente dentro de una clase cuando el desarrollador no lo proporciona dentro de ella. Cuando se crea un objeto de esta clase, se invoca automáticamente el constructor por default y se inicializan los campos de la clase con valores predeterminados. Por ejemplo, los valores numéricos se inicializan en 0, los booleanos a false y las referencias de objetos a null.

Para entender el proceso de aplicación del método de refactorización sobre este tipo de constructores, en la Figura 39 se observa la arquitectura inicial de un proyecto, donde se ven implementados tres métodos públicos abstractos con un constructor el cual no contiene parámetros. La refactorización dentro de este escenario consiste en que si se encuentra un constructor sin implementar parámetros y código en su cuerpo se debe retirar de la clase con la finalidad de solo utilizar el constructor por default que maneja Java.

Arquitectura Original

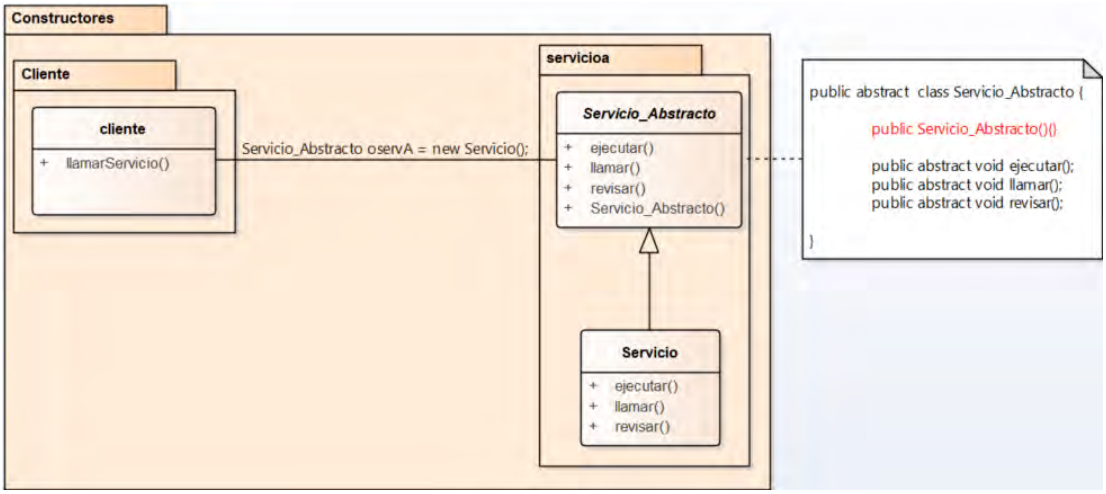


Figura 39 Implementación de un constructor sin parámetros dentro de una clase abstracta.

Al ejecutar el método de refactorización sobre este escenario, la clase “Servicio_Abstracto” se refactoriza a una interfaz, removiendo el constructor implementado por el desarrollador, ya que es un constructor el cual no tiene alguna función dentro del código, lo que indica que no tiene cuerpo que implemente alguna función, como resultado en este tipo de refactorización, no se afecta al funcionamiento del sistema y se siguen respetando los principios de diseño, sobre los cuales se trabajaron en este proyecto de tesis.

Código Legado – Escenario 5 Constructores
Antes de Refactorizar.

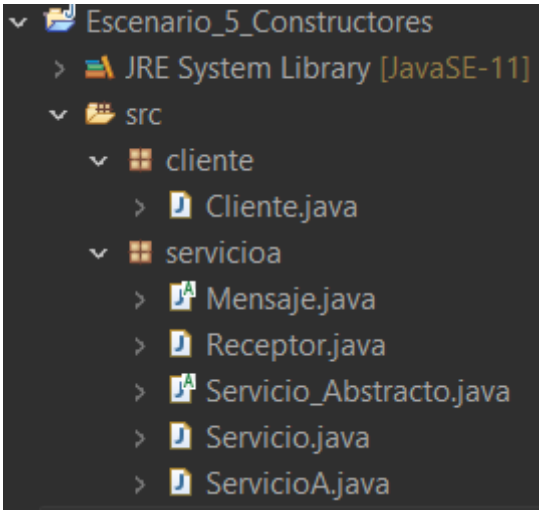


Figura 40 Arquitectura original antes de refactorizar.

En la Figura 40, se muestra la arquitectura original del proyecto sobre el cual se aplica este escenario. En este proyecto dentro de la clase “Servicio_Abstracto” es donde se encuentra el método constructor “*Servicio_Abstracto*”, el cual como se muestra en la Figura 41, a nivel de código, el método constructor “Servicio_Abstracto” no implementa ninguna función

Servicio_Abstracto.java

```
1  package servicioa;
2
3  public abstract  class Servicio_Abstracto {
4
5      public Servicio_Abstracto(){}
6
7      public abstract void ejecutar();
8      public abstract void llamar();
9      public abstract void revisar();
10
11 }
12
```

Figura 41 Clase abstracta con un constructor por defecto.

Arquitectura Esperada

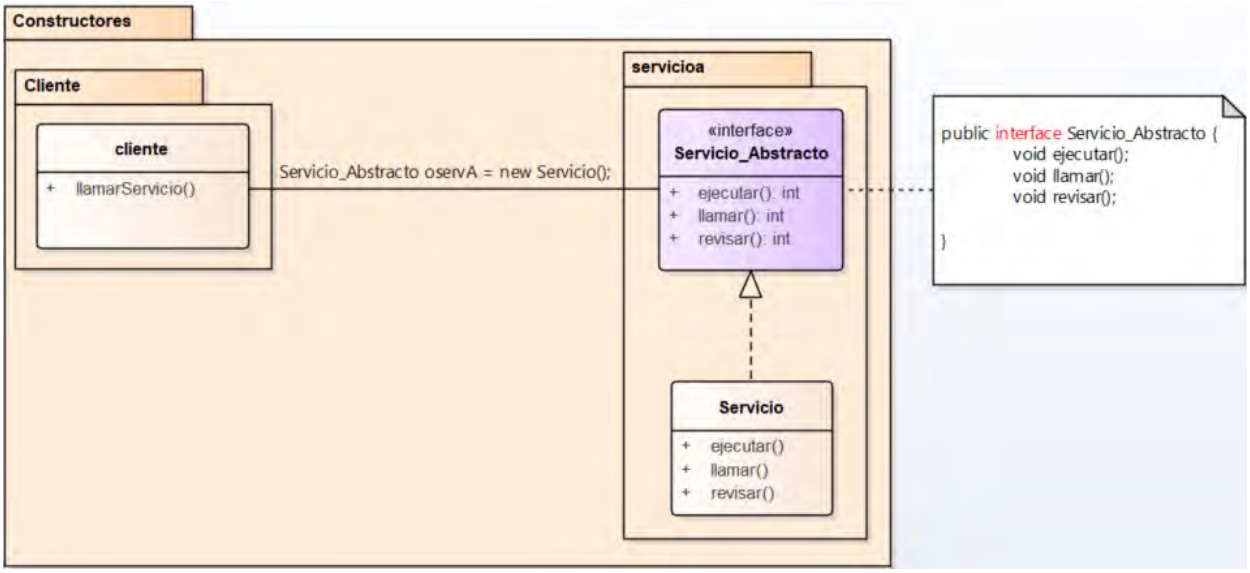


Figura 42 Refactorización de una clase abstracta con un constructor default el cual es retirado ya que no tiene una función dentro del código.

Código Refactorizado

Al hacer la aplicación del método de refactorización, se observa en la arquitectura obtenida que el constructor por default implementado es removido dejando implementados únicamente los métodos como públicos, en la Figura 43 se muestra a nivel de código el resultado de la refactorización.

Servicio_Abstracto.java

```
1 package servicioa;
2
3 public interface Servicio_Abstracto {
4     void ejecutar();
5
6     void llamar();
7
8     void revisar();
9 }
```

Figura 43 Interfaz creada en donde se retira el constructor vacío (defecto).

Tabla 4 Resultado del cálculo de métricas Escenario 5

	Escenario 5	
	Antes de Refactorizar	Después de Refactorizar
FA	0.33333334	0.42857143
FP	1.0	1.0

Al aplicar las métricas de refactorización, podemos notar un aumento después de hacer la refactorización esto debido a que en este proyecto se creó una interfaz más por una de las clases que se ha refactorizado, cabe mencionar que esto no afecta en el funcionamiento del programa ya que lo que se busca en este proyecto es poder extender las funcionalidades de los módulos de programa que conformen a cada uno de los sistemas que se ha de refactorizar.

6.2.- Ejecución del método en los casos de pruebas.

6.2.1.- Caso de Prueba ISRMCCA0501

Nombre del Caso de prueba: Sistema de banco.

En la Figura 44 se muestra un segmento del diagrama de clases de la arquitectura original del proyecto Banco-master, dentro de este proyecto se cuenta con el paquete “dao” en el que se almacenan las clases “Main”, “ConnectionFactory”, “ContaDAO” y “ClienteDAO”, dicho sistema está escrito en lenguaje Java, ha sido compilado y revisado para constatar que su funcionamiento es correcto.

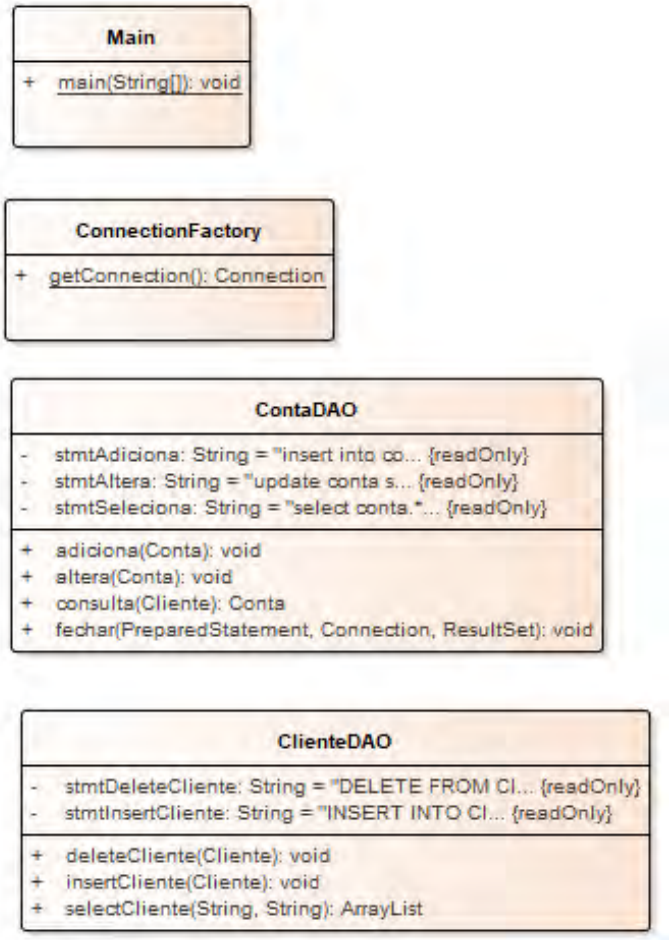
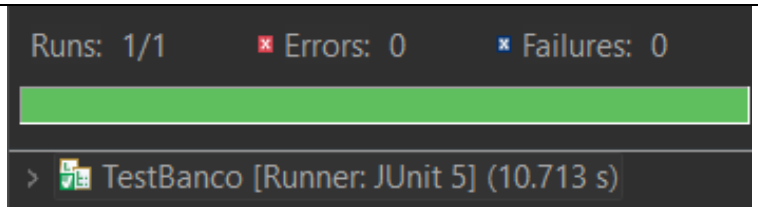


Figura 44 Arquitectura de Clases del proyecto Sistema de Banco.

Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes:

Características a probar	**Pruebas Unitarias**

Cálculo de la métrica *"Factor de abstracción" y "Factor de polimorfismo"*.	
	Test-Banco
	`@Test`
	`void pruebaProyectoBanco() {`
	`JFileChooser jfc = new JFileChooser();`
	`jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);`
	`jfc.setAcceptAllFileFilterUsed(false);`
	`jfc.showOpenDialog(null);`
	`String jfcRuta = jfc.getSelectedFile().getAbsolutePath();`
	`String ruta = "C:\\Users\\ferdi\\OneDrive\\Documentos\\Proyecto_Tesis_Fernando Sanchez Rogel\\Pruebas_Proyecto\\Banco-master\\src";`
	`assertEquals(ruta, jfcRuta);`
	`LectorArchivos lector = new LectorArchivos(ruta);`
	`Proyecto proyecto = new Proyecto();`
	`proyecto.definirArchivos(lector.construirArchivos());`
	`System.out.println("========== Proyecto Sistema de Banco ==========");`
	`orquestador = new Orquestador(proyecto);`
	`orquestador.generarModelo();`
	`orquestador.agregarMetrica(new NumeroClases());`
	`orquestador.agregarMetrica(new ContadorLineasCodigo());`
	`orquestador.agregarMetrica(new FactorAbstraccion());`
	`orquestador.agregarMetrica(new FactorPolimorfismo(false));`
	`orquestador.aplicarMetricas(true);`
	`RefactorizacionInterfaz metodo = new RefactorizacionInterfaz("bancoswing");`
	`RefactorizacionInterfaz metodo2 = new RefactorizacionInterfaz("Classes");`
	`RefactorizacionInterfaz metodo3 = new RefactorizacionInterfaz("dao");`
	`System.out.println("\n------------ Refactorizado ------------");`
	`metodo.refactorizarAInterfaces(proyecto);`
	`metodo2.refactorizarAInterfaces(proyecto);`
	`metodo3.refactorizarAInterfaces(proyecto);`
	`orquestador.aplicarMetricas(true);`
	`orquestador.definirPersistencia(new EscritorDisco(ruta, proyecto));`
	`orquestador.persistir();`
	`assertEquals(orquestador.cadenaFinal, "Numero de clases = 15.0\n"`
	`+ "LOC = 658.0\n"`
	`+ "Factor Abstraccion = 0.13333334\n"`
	`+ "Factor Polimorfismo = 0.7\n"`
	`+ "Numero de clases = 17.0\n"`
	`+ "LOC = 686.0\n"`
	`+ "Factor Abstraccion = 0.23529412\n"`
	`+ "Factor Polimorfismo = 0.7777778\n");`
	`}`
	`}`

Las pruebas unitarias fueron aprobadas al comparar la igualdad de los resultados esperados con los reales a través del método assertEquals. Por lo anterior las pruebas del cálculo de métricas son aprobadas y aceptadas.

6.2.2.- Caso de Prueba ISMRTM0502

Nombre del Caso de prueba: Sistema de banco.
Se procede a ejecutar la prueba unitaria observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en el aumento de la propiedad de abstracción mediante el cálculo de las métricas. En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

Características por probar	Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.

El método de refactorización de código legado con carencia de abstracción y la generación de código son probados utilizando los diagramas de clases como referencia.
Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación, para corroborar la correcta refactorización se genera el diagrama de clases de la Figura 45 y se compara la carencia de abstracción con la arquitectura de la Figura 44.
A continuación, se muestra el diagrama de clases del sistema refactorizado donde se generaron las interfaces satisfactoriamente tomando en cuenta los distintos escenarios previstos:

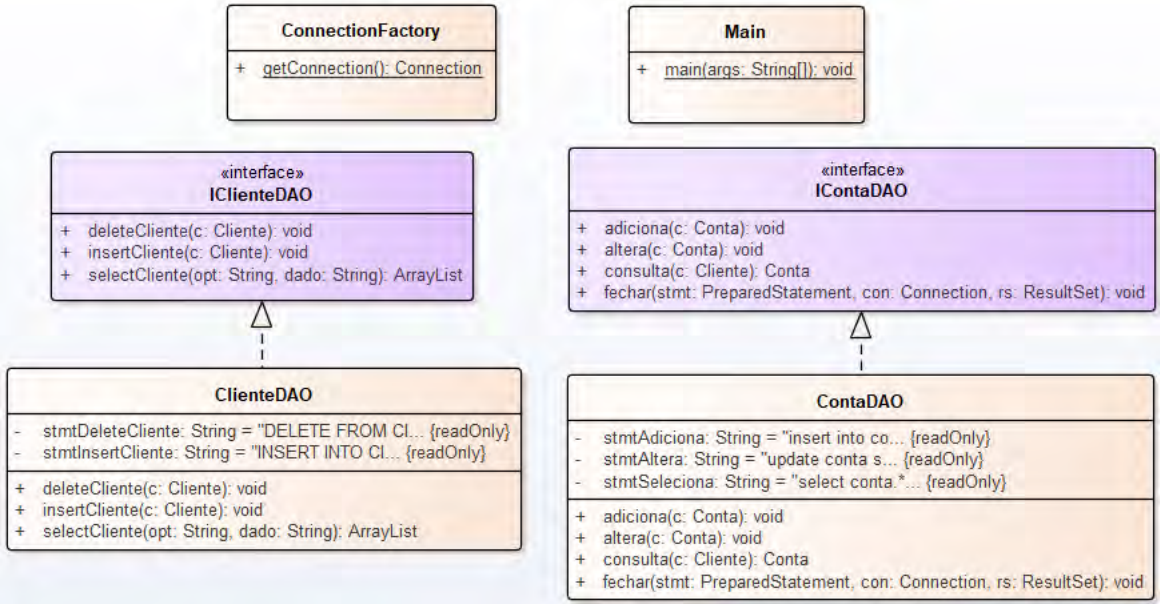


Figura 45 Arquitectura refactorizada del proyecto Sistema de Banco.

En las clases de la Figura 44 podemos visualizar que no hay abstracción, ya que están cerradas a extender más comportamientos dentro de ellas, es por ello por lo que a las clases “ClienteDAO” y “ContaDao” son candidatas para convertir a

interfaces, con la finalidad de contar con abstracción en el código, habilitando de esta manera la flexibilidad, extensibilidad y modularidad del programa. Al aplicar el método de refactorización a esta prueba aplicando las métricas de FA y FP, los resultados obtenidos fueron los siguientes:

```
========== Proyecto Sistema de Banco ==========
Numero de clases = 15.0
LOC = 658.0
Factor Abstraccion = 0.13333334
Factor Polimorfismo = 0.7

------------ Refactorizado -------------
Numero de clases = 17.0
LOC = 686.0
Factor Abstraccion = 0.23529412
Factor Polimorfismo = 0.7777778
```

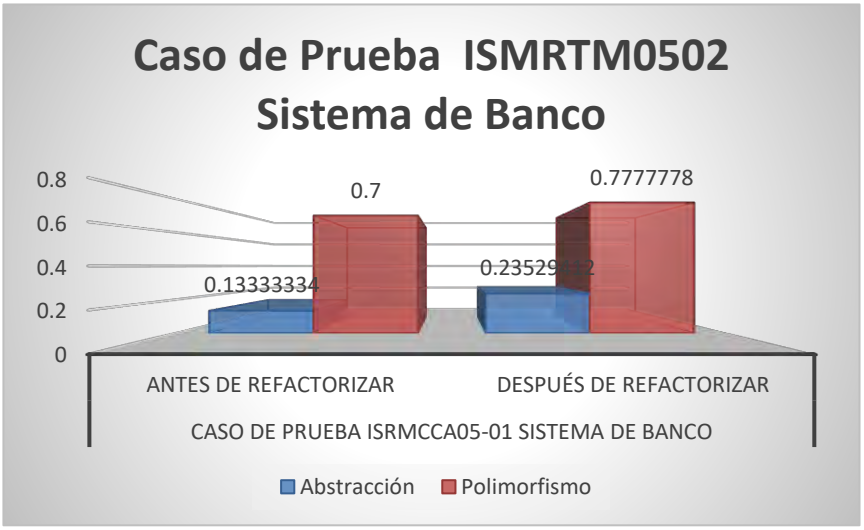


Figura 46 Resultados al aplicar las métricas de Abstracción y Polimorfismo al sistema de Banco.

Se puede apreciar un aumento de abstracción y polimorfismo después de haber aplicado el método de refactorización, esto quiere decir que se han aumentado el número de clases, en este caso implementando las interfaces de las clases antes ya mencionadas, a partir de esto se puede decir que el código es ahora más extensible a nuevos requerimientos, y que por lo tanto aplica un grado de abstracción más alto, lo que habilita la propiedad de polimorfismo respetando los principios de abierto – cerrado e inversión de dependencias.

Características por probar	Verificación del funcionamiento del código refactorizado.
El sistema funciona correctamente antes y después de la refactorización	

Las pruebas son aprobadas al comparar la igualdad de los resultados esperados con los reales, el método de refactorización y generación de código refactorizado es aprobado al generar las clases de tipo interfaz, finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización,

por lo que se concluye que la refactorización no altera el comportamiento del sistema.

6.2.3.- Caso de Prueba ISRMCCA0503

Nombre del Caso de prueba: Sistema de Punto de Venta.
En la siguiente figura se muestra un segmento del diagrama de clases de la arquitectura original del sistema donde se encuentran las clases del paquete “accesoDatos”, el sistema está escrito en lenguaje Java y ha sido compilado y revisado su funcionamiento.

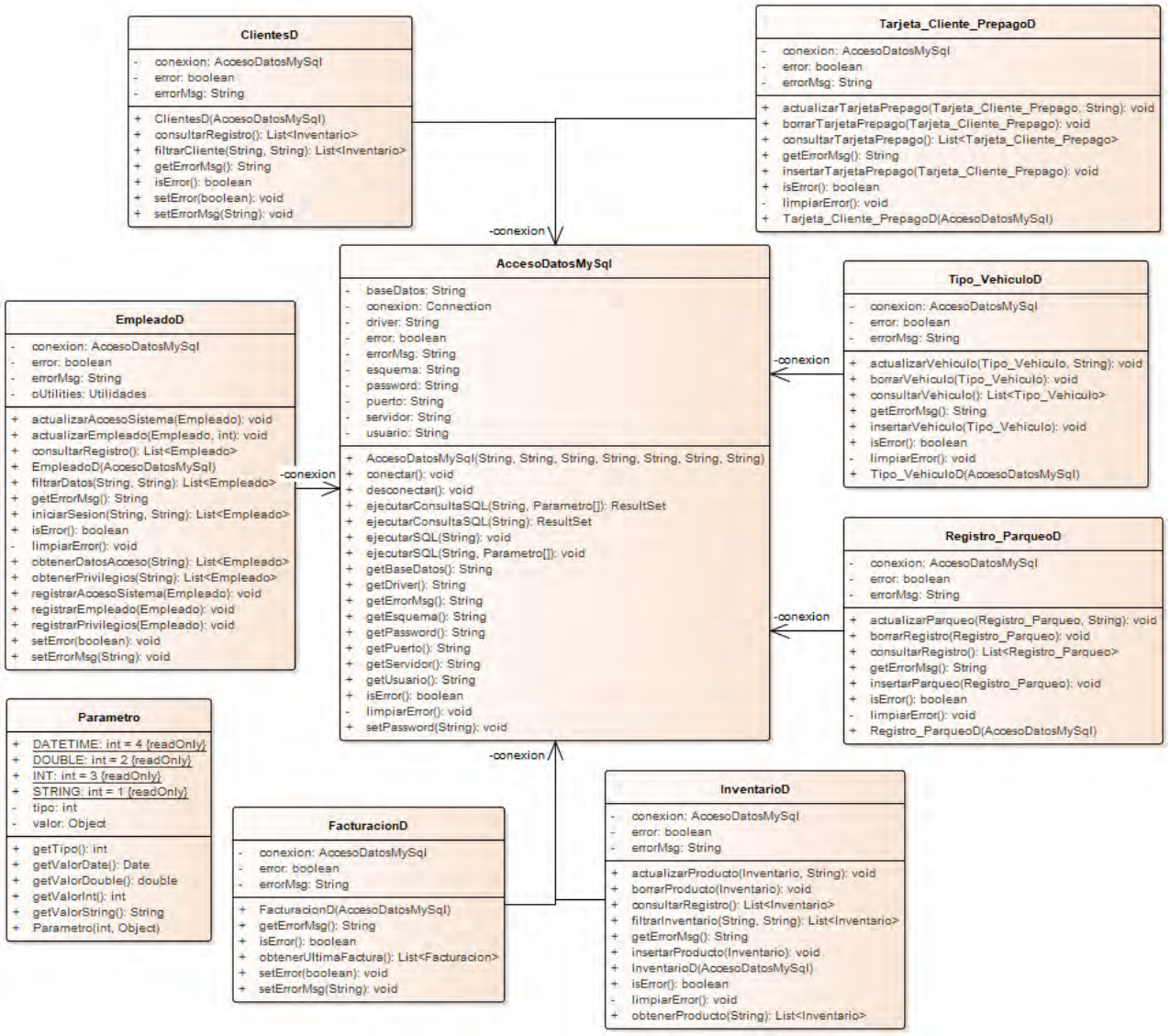
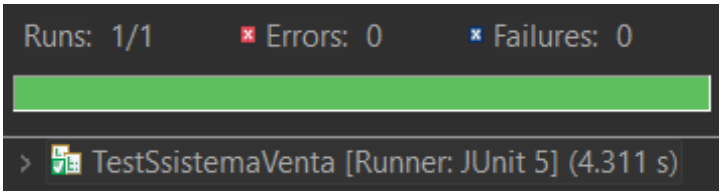


Figura 47 Arquitectura Original del proyecto Sistema Punto de Venta.

Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes:

Características a probar	Pruebas Unitarias
Cálculo de la métrica *“Factor de abstracción” y “Factor de polimorfismo”*.	 

Las pruebas unitarias fueron aprobadas al comparar la igualdad de los resultados esperados con los reales. Por lo anterior las pruebas del cálculo de métricas son aprobadas y aceptadas.

6.2.4.- Caso de Prueba ISMRTM0504

Nombre del Caso de prueba: Sistema de Punto de Venta.
Se procede a ejecutar la prueba unitaria observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en el aumento de la propiedad de abstracción mediante el cálculo de las métricas y finalmente se prueba que el comportamiento del sistema siga siendo el mismo.

En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

Características a probar	Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.
El método de refactorización de código legado con carencia de abstracción y la generación de código son probados utilizando los diagramas de clases como referencia. Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación, para corroborar la correcta refactorización se genera el diagrama de clases de la Figura 48 y se compara la carencia de abstracción con la arquitectura de la Figura 47. A continuación, se muestra el diagrama de clases del sistema refactorizado donde se generaron las interfaces satisfactoriamente tomando en cuenta los distintos escenarios previstos:	

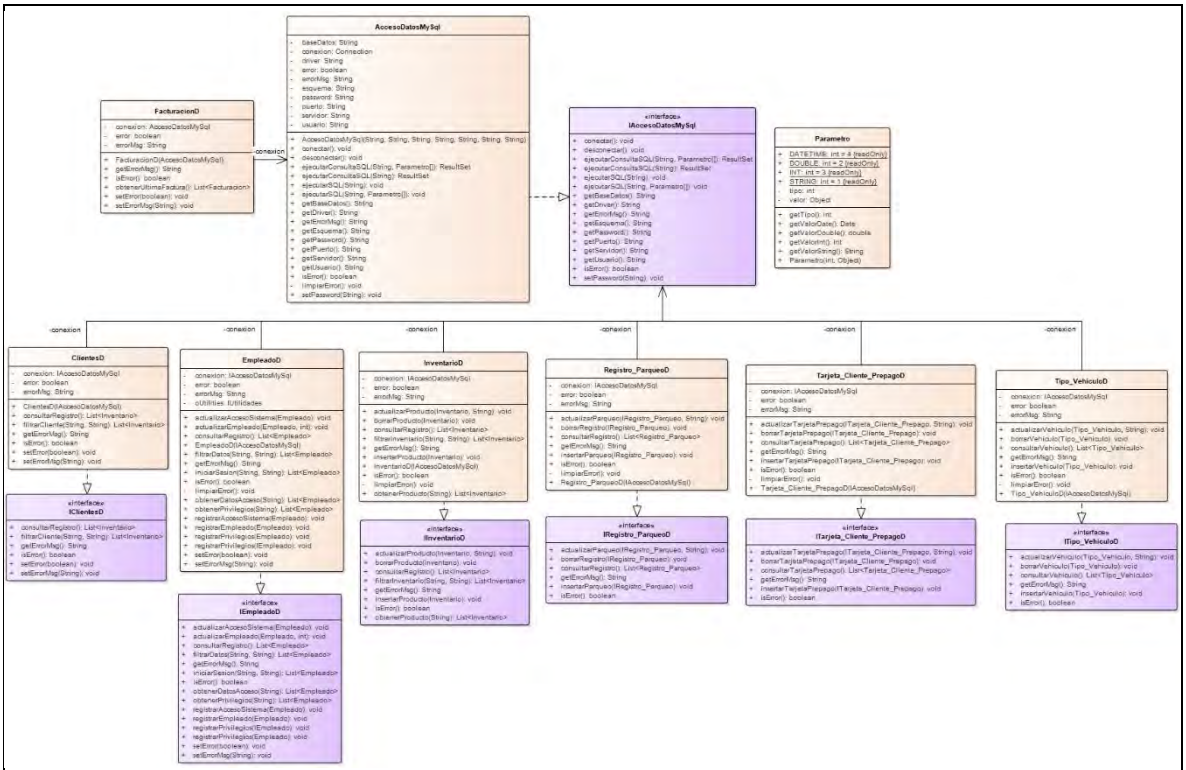


Figura 48 Arquitectura refactorizada del proyecto Sistema de Venta.

En las clases de la Figura 47 podemos visualizar que no hay abstracción, ya que están cerradas a extender más comportamientos dentro de ellas, es por ello que a las clases “AccesoDatosMySql”, “ClientesD”, “EmpleadoD”, “InventarioD”, “Registro_ParqueoD”, “Tarjeta_Cliente_PrepagoD” y “Tipo_VehiculoD” son candidatas para convertir a una interfaz esto con la finalidad de tener abstracción en el código habilitando de esta manera la flexibilidad, extensibilidad y modularidad del programa. Al aplicar el método de refactorización a esta prueba aplicando las métricas de FA y FP, los resultados obtenidos fueron los siguientes:

```
========== Proyecto Sistema de Punto de Venta ==========
Numero de clases = 61.0
LOC = 6240.0
Factor Abstraccion = 0.0
Factor Polimorfismo = 0.0
------------ Refactorizado -------------
Numero de clases = 75.0
LOC = 6554.0
Factor Abstraccion = 0.18666667
Factor Polimorfismo = 1.0511364
```

En el anexo B se presenta gráficamente los valores obtenidos de la refactorización aplicada a este caso de prueba. Se puede apreciar el aumento de abstracción y polimorfismo después de haber aplicado el método de refactorización, esto quiere

decir que se han aumentado el número de clases implementado en este caso las interfaces de las clases antes ya mencionadas, a partir de esto se puede decir que el código ahora ya es más extensible a nuevos requerimientos, y que por lo tanto aplica un grado de abstracción más alto, lo que habilita la propiedad de polimorfismo respetando los principios de abierto – cerrado e inversión de dependencias.	
Características a probar	Verificación del funcionamiento del código refactorizado.
El sistema funciona correctamente antes y después de la refactorización	

Las pruebas son aprobadas al comparar la igualdad de los resultados esperados con los reales, el método de refactorización y generación de código refactorizado es aprobado al comparar al generar las clases de tipo interfaz, finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización, por lo que se concluye que la refactorización no altera el comportamiento del sistema.

6.2.5.- Caso de Prueba ISRMCCA0505

Nombre del Caso de prueba: PSPCenidet.

En la siguiente figura se muestra un segmento del diagrama de clases de la arquitectura original del sistema donde se encuentran las clases del paquete “pckcommand”, el sistema está escrito en lenguaje Java y ha sido compilado y revisado su funcionamiento.

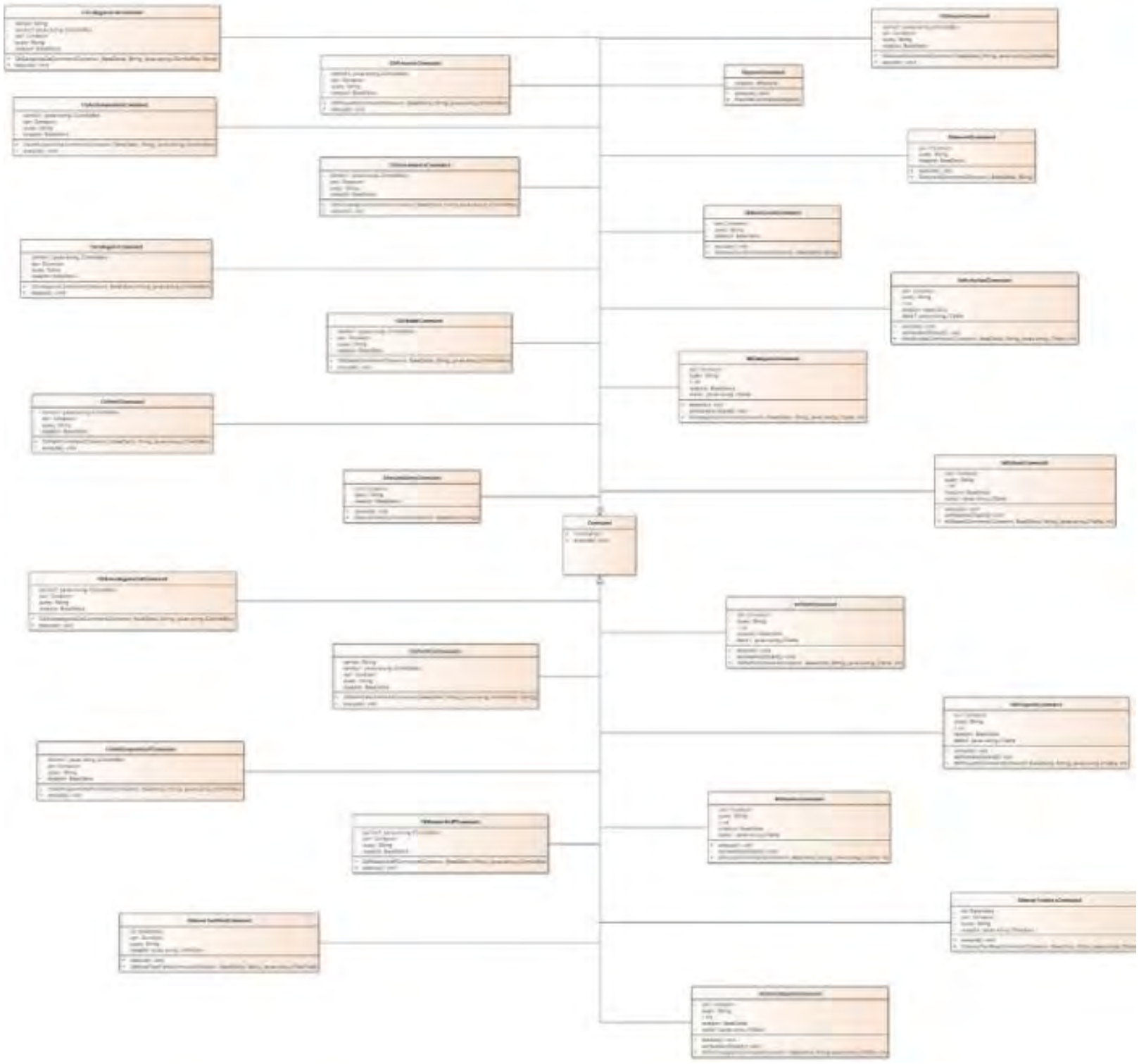
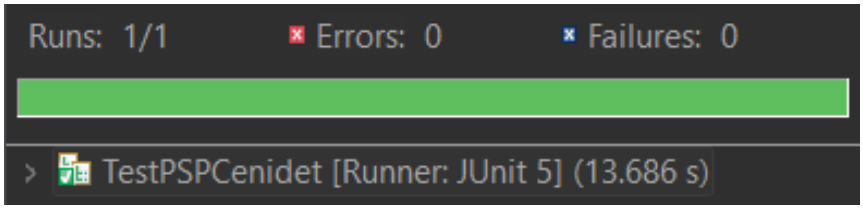


Figura 49 Arquitectura original del proyecto PSPCenidet

Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes:

Características a probar	Pruebas Unitarias

Cálculo de la métrica *“Factor de abstracción” y “Factor de polimorfismo”*.	
	Test-PSPCenidet
	@Test
	void pruebaPSPCenidet() {
	JFileChooser jfc = new JFileChooser();
	jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
	jfc.setAcceptAllFileFilterUsed(false);
	jfc.showOpenDialog(null);
	String jfcRuta = jfc.getSelectedFile().getAbsolutePath();
	String ruta = "C:\\Users\\ferdi\\OneDrive\\Documentos\\Proyecto_Tesis_Fernando Sanchez Rogel\\Pruebas_Proyecto\\PSPCenidet\\src";
	assertEquals(ruta, jfcRuta);
	LectorArchivos lector = new LectorArchivos(ruta);
	Proyecto proyecto = new Proyecto();
	proyecto.definirArchivos(lector.construirArchivos());
	System.out.println("========== Proyecto PSP Cenidet ==========");
	orquestador = new Orquestador(proyecto);
	orquestador.generarModelo();
	orquestador.agregarMetrica(new NumeroClases());
	orquestador.agregarMetrica(new ContadorLineasCodigo());
	orquestador.agregarMetrica(new FactorAbstraccion());
	orquestador.agregarMetrica(new FactorPolimorfismo(false));
	orquestador.aplicarMetricas(true);
	RefactorizacionInterfaz metodo = new RefactorizacionInterfaz("pckcommand");
	RefactorizacionInterfaz metodo2 = new RefactorizacionInterfaz("pckmodelo");
	RefactorizacionInterfaz metodo3 = new RefactorizacionInterfaz("pckcenidet");
	System.out.println("------------ Refactorizado -------------");
	metodo.refactorizarAInterfaces(proyecto);
	metodo2.refactorizarAInterfaces(proyecto);
	metodo3.refactorizarAInterfaces(proyecto);
	orquestador.aplicarMetricas(true);
	orquestador.definirPersistencia(new EscritorDisco(ruta, proyecto));
	orquestador.persistir();
	assertEquals(orquestador.cadenaFinal, "Numero de clases = 81.0\n"
	+ "LOC = 3597.0\n"
	+ "Factor Abstraccion = 0.09876543\n"
	+ "Factor Polimorfismo = 0.8484849\n"
	+ "Numero de clases = 81.0\n"
	+ "LOC = 3597.0\n"
	+ "Factor Abstraccion = 0.09876543\n"
	+ "Factor Polimorfismo = 0.8484849\n");
	}
	}

Las pruebas unitarias fueron aprobadas al comparar la igualdad de los resultados esperados con los reales. Por lo anterior las pruebas del cálculo de métricas son aprobadas y aceptadas.

6.2.6.- Caso de Prueba ISMRTM0506

Nombre del Caso de prueba: PSPCenidet.

Se procede a ejecutar la prueba unitaria observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en el aumento de la propiedad de abstracción mediante el cálculo de las métricas y finalmente se prueba que el comportamiento del sistema siga siendo el mismo. En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

Características a probar	Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.
El método de refactorización de código legado con carencia de abstracción y la generación de código son probados utilizando los diagramas de clases como referencia. Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación, para corroborar la correcta refactorización se genera el diagrama de clases de la Figura 50 y se compara la carencia de abstracción con la arquitectura de la Figura 49. A continuación, se muestra el diagrama de clases del sistema refactorizado donde se generaron las interfaces satisfactoriamente tomando en cuenta los distintos escenarios previstos:	

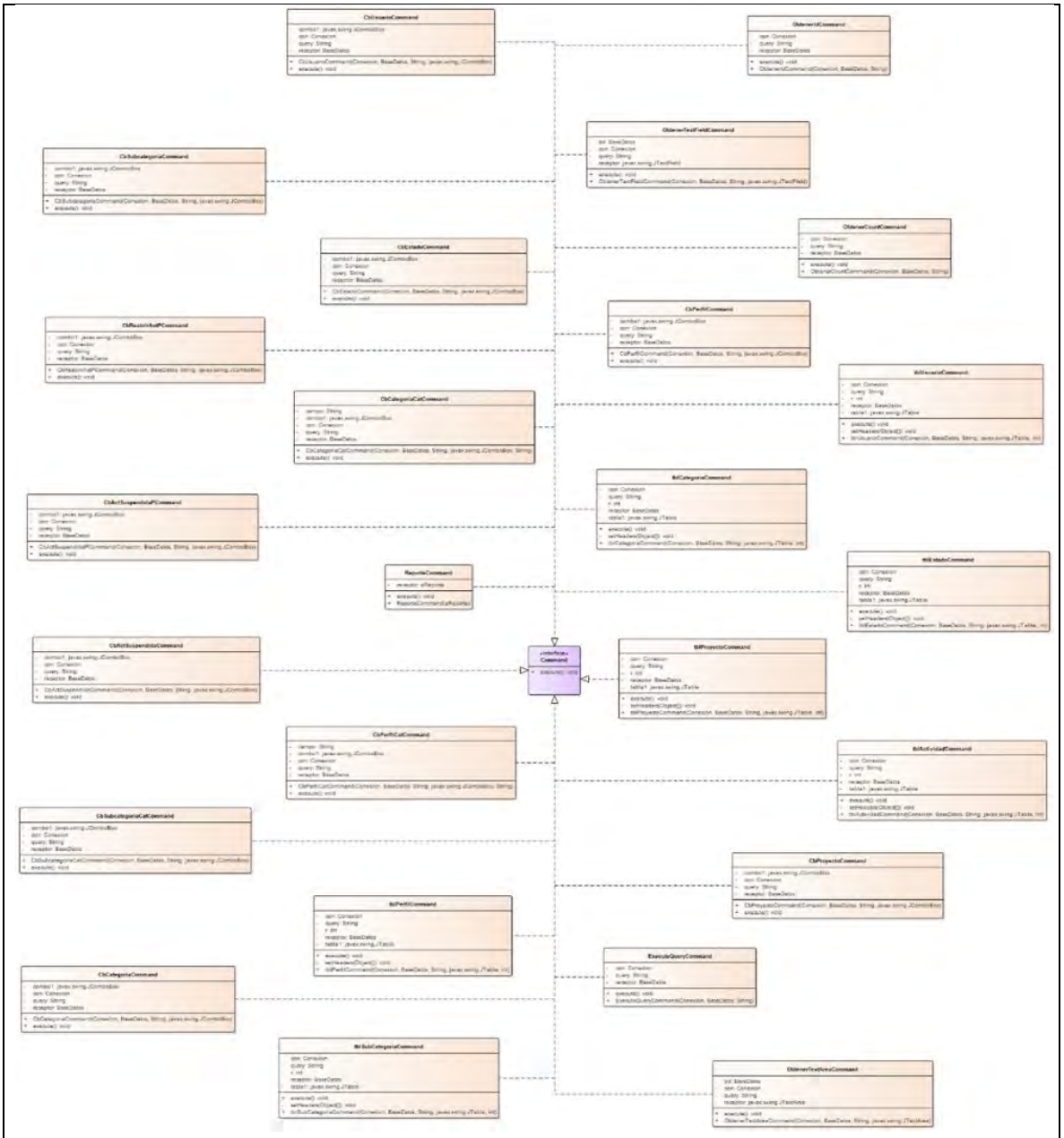


Figura 50 Arquitectura refactorizada el proyecto PSPCenidet.

En las clases de la Figura 49 podemos visualizar que no hay abstracción, ya que están cerradas a extender más comportamientos dentro de ellas, es por ello que a la clase “Command” es candidata para convertir a una interfaz esto con la finalidad de tener abstracción en el código habilitando de esta manera la flexibilidad, extensibilidad y modularidad del programa.

Al aplicar el método de refactorización a esta prueba aplicando las métricas de FA y FP, los resultados obtenidos fueron los siguientes:

```
========== Proyecto PSP Cenidet ==========
Numero de clases = 81.0
LOC = 3597.0
Factor Abstraccion = 0.09876543
Factor Polimorfismo = 0.8484849
------------ Refactorizado -------------
Numero de clases = 81.0
LOC = 3597.0
Factor Abstraccion = 0.09876543
Factor Polimorfismo = 0.8484849
```

Se puede apreciar que la abstracción y polimorfismo después de haber aplicado el método de refactorización se mantiene igual, no se altera, esto debido a que durante su ejecución se detectó una clase abstracta con un método público abstracto y un constructor por defecto, es por esto por lo que los valores se mantienen idénticos porque como ya se había mencionado una interfaz es aquella que también implementa métodos abstractos públicos, lo que hace que sea abstracta.

Características a probar	Verificación del funcionamiento del código refactorizado.
El sistema funciona correctamente antes y después de la refactorización	

Las pruebas son aprobadas al comparar la igualdad de los resultados esperados con los reales, el método de refactorización y generación de código refactorizado es aprobado al comparar al generar las clases de tipo interfaz, finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización, por lo que se concluye que la refactorización no altera el comportamiento del sistema.

Una vez que ya se ha aplicado la refactorización en estos casos de prueba se prosiguió a comprobar que los sistemas refactorizados funcionaran correctamente. En la siguiente tabla se describe el estado de los sistemas que fueron refactorizados.

	Antes de Refactorizar	Después de Refactorizar
ISRMCCA05-01 Sistema de Banco	**Funcional**	**Funcional**
ISRMCCA05-02 Sistema de Punto de Venta	**Funcional**	**Funcional**
ISRMCCA05-02 PSPCenidet	**Funcional**	**Funcional**

Los resultados obtenidos durante la fase de pruebas fueron satisfactorios ya que el método de refactorización cumple con los objetivos planteados para este proyecto de tesis.

La refactorización realizada a estos casos de prueba respeta los principios de "Inversión de Dependencias" y "Abierto - Cerrado", al igual que habilita el polimorfismo gracias a la abstracción que se introduce a los sistemas, y como ya se había mencionado anteriormente la arquitectura a través de estos cambios es más flexible a cambios de comportamiento en el tiempo de ejecución y se es posible extender a nuevas funcionalidades o comportamientos.

7.- Conclusiones y Trabajos Futuros

La abstracción dentro de sistemas de software beneficia en gran manera el diseño arquitectural y su implementación en código, haciendo que el software sea flexible a cambios de comportamiento. Por otra parte, si un sistema de software hace uso de abstracciones, se habilita el polimorfismo lo que indica que los objetos implementados en una clase tienen la capacidad de ofrecer un comportamiento diferente en su invocación, según el tipo del objeto al que se le envía el mensaje.

El desarrollo de esta tesis de investigación cumple con el objetivo de dar soporte a las propiedades de asociación dinámica y polimorfismo en las arquitecturas de software, esto se logró disminuyendo la deuda técnica en sus dimensiones de flexibilidad y extensibilidad y mejorando su modularidad.

El principal aporte de esta tesis de investigación es un método y técnica de refactorización extensible para el "*Método de refactorización de arquitecturas de software con carencia de abstracciones"* desarrollado en *CENIDET* , este aporte se logra a través de la creación automatizada de interfaces en los módulos de clases base y clases abstractas las cuales son definidas correctamente. Este método de refactorización al implementar interfaces dentro de sistemas con carencia de abstracción proporciona una abstracción completa, donde solo proporciona prototipos de métodos, más no su implementación. Uno de los beneficios que se logra a través de la técnica de refactorización automatizada presentada en esta investigación es crear propiedades evolutivas para los sistemas legados a partir de la generación de interfaces, las cuales mejoran el factor polimórfico y abstracto del sistema y nos permiten simular la herencia múltiple en Java, prolongando el tiempo de vida del sistema legado, ya que de manera nativa este lenguaje de programación no permite la herencia múltiple.

Algo muy importante por mencionar es que, al aplicar las métricas de FA y FP, se pueden obtener resultados idénticos al refactorizar dado que técnicamente una Interfaz es de un tipo abstracto utilizado para especificar el comportamiento de una clase.

El impacto y beneficio de utilizar este método de refactorización se puede ver aplicado a cualquier sector de la actividad económica que haga uso de sistemas orientados a objetos y la técnica de refactorización será aplicada a diversos sectores que hagan uso de sistemas escritos en lenguaje de programación Java, mejorando la calidad y prolongando el tiempo de vida del software.

Con la refactorización aplicada de este método a los casos de prueba se obtiene como el resultado el cumplimiento de los principios de “Inversión de Dependencias” y “Abierto - Cerrado”, al igual que habilita el polimorfismo gracias al aumento de abstracción que se introduce en los sistemas de caso de prueba, a partir de estos cambios, estas arquitecturas se volvieron más flexibles a cambios de comportamiento en el tiempo de ejecución y extensibles a nuevas funcionalidades.

En el transcurso del desarrollo de este método de refactorización, se encontraron diferentes puntos importantes a tratar, los cuales no se pudieron implementar en este trabajo de tesis ya que se necesitaría un escudriñamiento más profundo en ellos, estos puntos importantes se mencionan a continuación:

1). En casos de clases con funciones abstractas, en las que en sus clases derivadas algunas de éstas no tuvieran implementación, antes de aplicar este método es necesario separar estas abstracciones, con el propósito de respetar el principio de Segregación de interfaces, el cual ayudaría a los clientes de un sistema a no ser forzados utilizar abstracciones que no se usan, ya que esto resulta en un acoplamiento innecesario entre ellos mismos. Una vez segregadas las abstracciones, el método desarrollado en esta tesis cambia las funciones abstractas en interfaces.

2). Si durante la ejecución del método de refactorización se encuentra una clase base con alguna(s) función(es) virtual(es) y esta función tiene cuerpo pero no implementado, entonces: si en las clases derivadas no se invalida el cuerpo de esta función y reemplazada por una implementación concreta, se estaría violando el principio de sustitución puesto que los clientes de está función esperaría un resultado como postcondición cuando en realidad no recibe nada, además habría herencia de implementación aun cuando ésta fuera nula; solo en casos en que las funciones correspondientes implementen su comportamiento no se violaría el

principio de sustitución puesto que habría un resultado esperado por el cliente como postcondición.

3). Al aplicar la refactorización sobre clases base y clases abstractas por interfaces podemos encontrarnos con el escenario de que habrá interfaces que se repitan, esto quiere decir que la firma de sus métodos sea idéntica, esto provoca que se viole el principio DRY (en inglés Don’t Repeat Yourself), el cual nos dice que no debemos repetir el mismo código en el sistema, con el fin de tener un código con mayor mantenibilidad, más legible y fácil de entender, y que sea reusable

Anexo A.- Plan de Pruebas.

1.- Introducción

1.1.- Identificador Del Documento

ISRMCCA0301.

La convención de la nomenclatura de los identificadores es la siguiente:

- IS = Ingeniería de Software.
- RMCCA = Refactorización de módulos colaborativos con carencia de abstracción.

Tipo de artículo:

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

Identificador:

- XX = Identificador Numérico

1.2.- Alcance

Este plan de pruebas abarca una evaluación completa del funcionamiento del método de refactorización que tiene como objetivo implementar la refactorización sobre clases abstractas que puedan ser cambiadas por una interfaz en una aplicación en lenguaje Java, así como la evaluación del correcto funcionamiento del marco me métricas.

1.3.- Referencias.

Los siguientes documentos fueron usados como fuente de información para este plan de pruebas:

- Documento de análisis y diseño.

1.4.- Puntos De Prueba

Los siguientes módulos que serán probados se muestran a continuación:

Sistema	Función	Identificador
Marco de métricas de Módulos Colaborativos con Carencia de Abstracción	Subsistema de estimación de carencia de abstracción y	ISRMCCA0101

	polimorfismo de clases de objetos	
Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción	Sistema de evaluación y refactorización de clases abstractas a interfaces.	ISRMCCA0102

1.5.- Procedimientos De Control De Tareas

Los procedimientos de control para las tareas del método de refactorización y para las tareas de la evaluación de las métricas.

Sistema	Función	Identificador
Programa de aplicación.	Cálculo de la métrica *"Factor de abstracción" y "Factor de polimorfismo"*.	ISRMCCA0201.
Programa de aplicación.	Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.	ISRMCCA0202
Programa de aplicación.	Verificación del funcionamiento del código refactorizado.	ISRMCCA0203

2.- Características Para Ser Probadas

Las características que deben ser probadas son:

Diseño de prueba	Identificador
Cálculo de métricas de carencia de abstracción y polimorfismo de clases de objetos	ISRMCCA0401
Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción	ISRMCCA0402

3.- Características Para No Probar

A continuación, las características que no serán probadas:

- Los casos para probar no incluirán todas las combinaciones sintácticas del lenguaje java.
- El método de refactorización no señala si el código legado está libre de errores.
- La interfaz no es probada.

4.- Enfoque

El maestrando del CENIDET Fernando Sánchez Rogel realizará las pruebas, de esta manera se pude verificar que las pruebas son de acorde al desarrollo del sistema.

4.1.- Pruebas De Calidad

Las pruebas de calidad son dadas por la aplicación de las métricas de Factor Abstracción y Factor Polimorfismo en los casos de prueba y comparando resultados esperados con los obtenidos por el subsistema del marco de métricas.

4.2.- Pruebas De Refactorización

La prueba de refactorización consiste en generar interfaces a partir de clases abstractas si es que así lo requiere la arquitectura de software para mejorar su diseño. La validación será dada mediante la aplicación de las métricas de calidad para corroborar el aumento de del factor de abstracción y polimorfismo, así como en la ejecución del código legado y del código refactorizado, en donde se comprobará el mismo funcionamiento y comportamiento con las mismas entradas y esperando los mismos resultados.

5.- Criterios Aprobado/Desaprobado

5.1.- Aprobación/Desaprobación Pruebas De Calidad

Las pruebas de calidad serán aprobadas o desaprobadas mediante la comparación de los resultados esperados del cálculo de las métricas con los resultados obtenidos a partir de la automatización del cálculo de las métricas.

5.2.- Aprobación/Desaprobación Pruebas De Refactorización

El criterio de aprobación o desaprobación para las pruebas de refactorización será comparando el comportamiento de la arquitectura de software antes y después de la generación de clases de tipo interfaz.

6.- Criterios De Suspensión Y Reanudación

Las pruebas no se suspenderán definitivamente, cada vez que se presente una prueba desaprobada se procederá a evaluar y corregir el error.

7.- Liberación De Pruebas

La liberación y aceptación de las pruebas será dada mediante la entrada y salida de datos de las pruebas.

8.- Diseño de pruebas

8.1.- Diseño De Prueba ISRMCCA0401

1. Diseño de prueba:

   ISRMCCA0401 - Cálculo de métricas de carencia de abstracción y polimorfismo de clases de objetos
2. Características para probar:
   - Se evaluará la correcta aplicación de las métricas para calcular el factor de polimorfismo y el factor de abstracción mediante pruebas unitarias.
3. Refinamiento del enfoque:

   El objetivo es evaluar la correcta aplicación de las métricas para calcular el factor de polimorfismo y el factor de abstracción en el software legado.

   El código legado deberá no presentar fallas por lo que será compilado y ejecutado en un compilador para el lenguaje de programación java, posteriormente el marco orientado a objetos para el cálculo de estas métricas realizará un reconocimiento sintáctico del código legado para generar una estructura de datos con la información necesaria para el cálculo de las métricas.
4. Aprobación/desaprobación de la evaluación de las características:

La aprobación de los casos de prueba del cálculo de las métricas será dada por la comparativa de los resultados obtenidos con los resultados esperados, para que esta comparación sea aprobatoria deben ser resultados iguales en cada una de las métricas de abstracción y polimorfismo.

8.2.- Diseño De Prueba ISRMCCA0402

1. Diseño de prueba:
   ISRMCCA0402 - Método de Refactorización de Módulos Colaborativos con Carencia de Abstracción.
2. Características a ser probadas:
   - Se evaluará el funcionamiento correcto del método de refactorización para módulos colaborativos con carencia de abstracción.
3. Refinamiento del enfoque:
   El objetivo es evaluar la generación correcta de interfaces que requiera el código legado.
   El código legado que presenta deuda técnica deberá no presentar fallas por lo que será compilado y ejecutado en un compilador para el lenguaje de programación java, así como verificar el comportamiento del software legado, posteriormente el sistema de refactorización de código tomará el archivo que implementa el código legado para realizar un reconocimiento sintáctico con el objetivo de generar estructuras de datos con la información necesaria para la refactorización.
4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del método de refactorización será dada por la comparativa de los resultados obtenidos de la aplicación de las métricas automáticamente antes y después de la refactorización, para que esta comparación sea aprobatoria debe existir mejora en el resultado de la arquitectura refactorizada contra los resultados obtenidos del cálculo de las métricas del código legado, así como también debe ser igual el comportamiento externo del software legado.

9.- Especificación de Casos de Prueba

9.1.- Caso de Prueba ISRMCCA0501

Nombre del Caso de prueba: Sistema de banco.

La función de este sistema de banco es el poder dar de alta a clientes, en donde, se puede administrar y se puede generar el tipo de cuenta que manejará para sus cuentas, Cuenta con un total de 15 clases y 658 líneas de código.

Características a probar:

Características a probar	Diseño de la prueba
Cálculo de la métrica *"Factor de abstracción"*.	ISRMCCA0401
Cálculo de la métrica *"Factor de polimorfismo"*.	ISRMCCA0401

Las clases pertenecientes al "Sistema de banco" serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan los resultados calculados de cada una de las métricas.

9.2.- Caso de Prueba ISRMCCA0502

Nombre del Caso de prueba: Sistema de banco.

La función de este sistema de banco es el poder dar de alta a clientes, en donde, se puede administrar y se puede generar el tipo de cuenta que manejará para sus cuentas, Cuenta con un total de 15 clases y 658 líneas de código.

Características a probar

:

Características a probar	Diseño de la prueba
Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.	ISRMCCA0402
Verificación del funcionamiento del código refactorizado.	ISRMCCA0402

Las clases pertenecientes al sistema "Sistema de banco" serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan las clases que pertenecen al sistema "Sistema de banco" refactorizadas libre de la deuda técnica originada por la carencia de abstracción en módulos colaborativos.

9.3.- Caso de Prueba ISRMCCA0503

Nombre del Caso de prueba: Sistema de Punto de Venta.
Este sistema de venta se encarga de llevar un control de productos de una tienda, en donde se puede dar de alta a proveedores, clientes, se puede llevar un inventario, etc. Cuenta con un total de 61 clases con 6,240 LOC.
Características a probar:

Características a probar	Diseño de la prueba
Cálculo de la métrica *"Factor de abstracción" y "Factor de polimorfismo"*.	ISRMCCA0401

Las clases pertenecientes al sistema "Sistema de Punto de Venta" serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente. Como salida se esperan los resultados calculados de cada una de las métricas.

9.4.- Caso de Prueba ISRMCCA0504

Nombre del Caso de prueba: Sistema de Punto de Venta.
Este sistema de venta se encarga de llevar un control de productos de una tienda, en donde se puede dar de alta a proveedores, clientes, se puede llevar un inventario, etc. Cuenta con un total de 61 clases con 6,240 LOC.
Características a probar:

Características a probar	Diseño de la prueba
Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.	ISRMCCA0402
Verificación del funcionamiento del código refactorizado.	ISRMCCA0402

Las clases pertenecientes al sistema "Sistema de Punto de Venta" serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente. Como salida se esperan las clases que pertenecen al sistema

“Sistema de Punto de Venta” refactorizadas libre de la deuda técnica originada por la carencia de abstracción en módulos colaborativos.

9.5.- Caso de Prueba ISRMCCA0505

Nombre del Caso de prueba: PSP CENIDET.

El sistema “PSPCenidet” es un sistema que mide los tiempos en que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software. PSPCenidet cuenta con un total de 81 clases y con un LOC de 3597.

Características a probar:

Características a probar	Diseño de la prueba
Cálculo de la métrica *“Factor de abstracción”*.	ISRMCCA0401
Cálculo de la métrica *“Factor de polimorfismo”*.	ISRMCCA0401

Las clases pertenecientes al sistema “PSPCenidet” serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan los resultados calculados de cada una de las métricas

9.6.- Caso de Prueba ISRMCCA0506

Nombre del Caso de prueba: PSP CENIDET.

El sistema “PSPCenidet” es un sistema que mide los tiempos en que los usuarios utilizan para realizar tareas cotidianas y/o tareas específicas al desarrollo de software. PSPCenidet cuenta con un total de 81 clases y con un LOC de 3597.

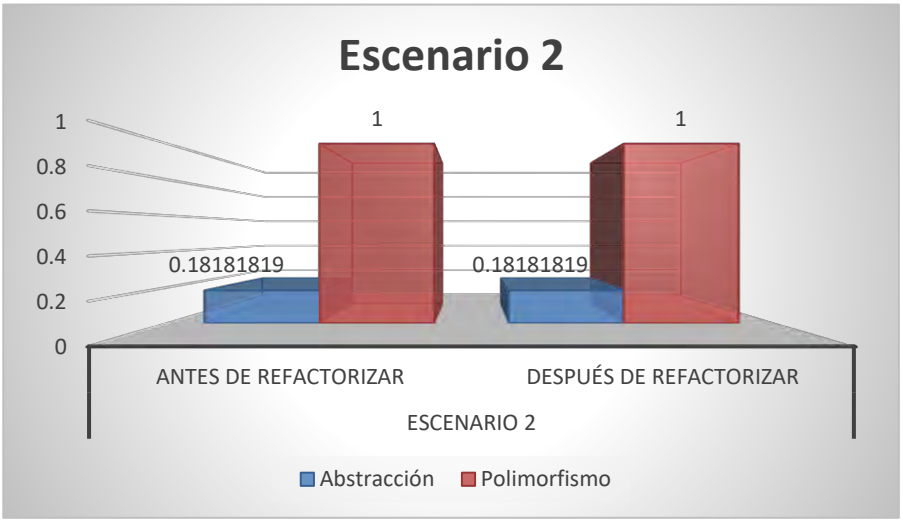
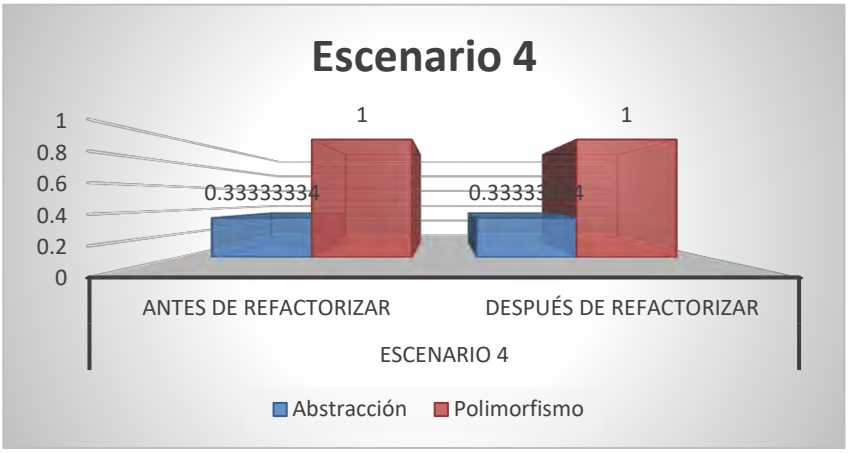
Características a probar:

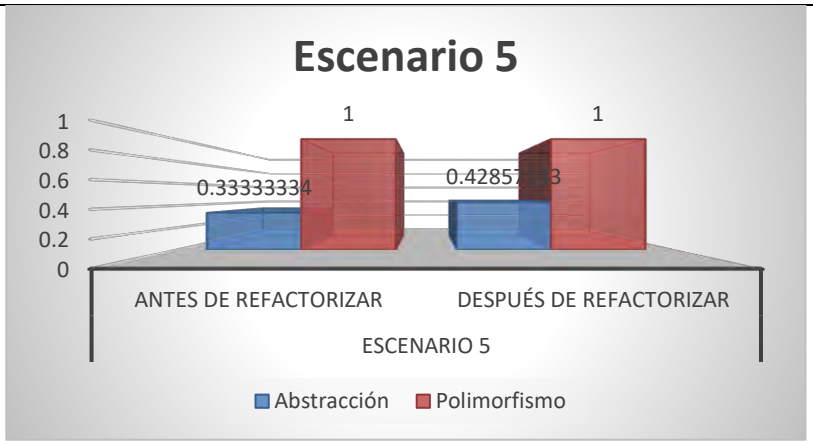
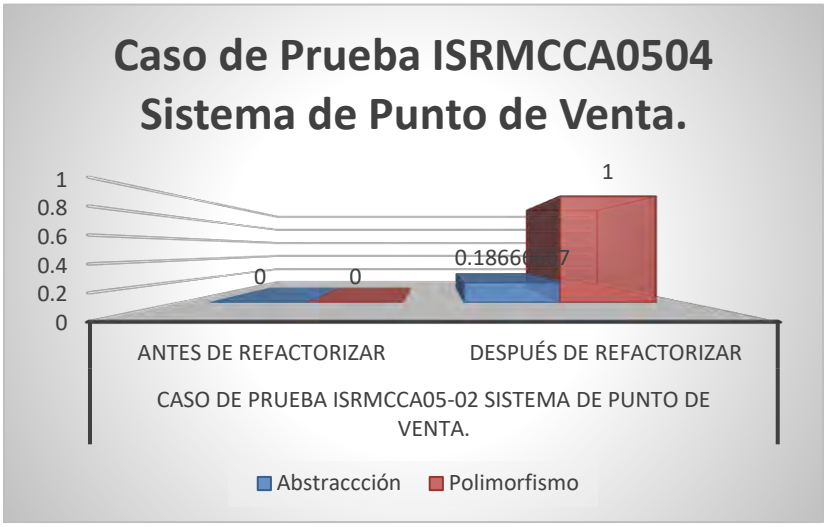
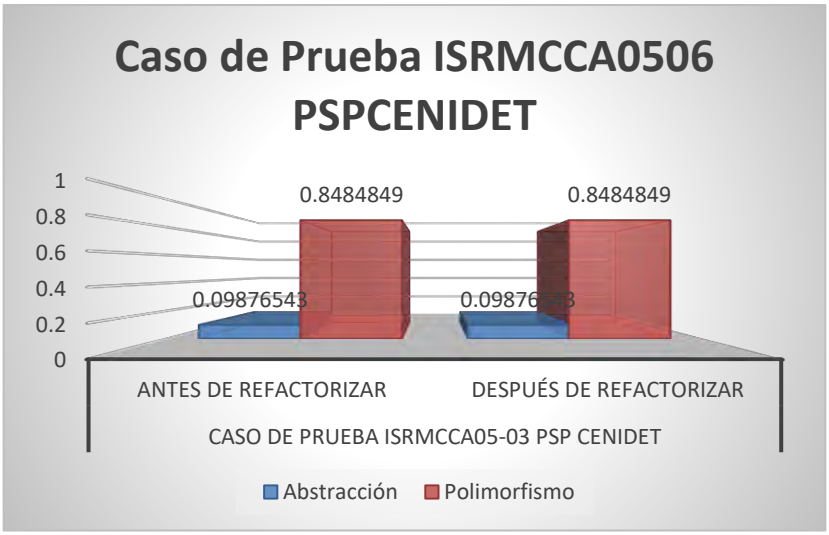
Características a probar	Diseño de la prueba
Método de refactorización de clases abstractas a interfaces y generación de código refactorizado.	ISRMCCA0402
Verificación del funcionamiento del código refactorizado.	ISRMCCA0402

Las clases pertenecientes al sistema “PSPCenidet” serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan las clases que pertenecen al sistema “PSPCenidet” refactorizadas libre de la deuda técnica originada por la carencia de abstracción en módulos colaborativos.

Anexo B.- Resultados de la medición de las métricas Abstracción y Factor Polimorfismo en los escenarios y casos de prueba.

Escenario 2. Arquitectura de software con una clase abstracta implementando métodos públicos abstractos, candidata a refactorizar.	
Escenario 4. Este escenario se enfoca en Funciones Virtuales implementadas en una clase las cuales no cuentan con alguna función y es por ello que las clases abstractas que las implementan son candidatas a refactorizar.	
Escenario 5. Constructores por default, estos se pueden encontrar dentro de sistemas y no tienen ninguna función en ellos, es por ello que al aplicar la refactorización de clases abstractas estos son	

removidos y la clase pasa a ser una interfaz.	
Caso de Prueba ISMRTM0504 Sistema de Punto de Venta.	
Caso de Prueba ISMRTM0506 PSPCenidet.	

Referencias

Angélica Nakayama C Ing Jorge A Solano Gálvez, M. M., & Alejandro Velázquez Mena, M. (2017). *Guía práctica de estudio 05: Abstracción y Encapsulamiento Elaborado por: Autorizado por*.

ARKAITZ GARRO. (2014, April 15). *CAPITULO 18 INTERFACES*. Https://Www.Arkaitzgarro.Com/Java/.

Brito Abreu, F., & Melo, W. (1996). *Evaluating the Impact of Object-Oriented Design on Software Quality*.

Cervantes O, J., Gómez F, M. del C., González P, P. P., & García N, A. (2016). *Introducción a la programación orientada a objetos*.

Christopoulou, A., Giakoumakis, E. A., Zafeiris, V. E., & Soukara, V. (2012). *Automated refactoring to the Strategy design pattern*. https://doi.org/10.1016/j.infsof.2012.05.004

Coronado Padilla, I. J. (2007). *Escalas de Medición*. *2*(2), 104–125.

Fenton, N. E., & Pfleeger, S. L. (1997). Software metrics: a rigorous and practical approach. 1997. *Brooks/Cole Pub Co*.

Fontana, F. A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., & Nitto, E. Di. (2017). *Arcan: a Tool for Architectural Smells Detection*. https://doi.org/10.1109/ICSAW.2017.16

Fowler, M., Beck Boston, K., New, C. •, San, Y. •, Amsterdam, F. •, Cape, •, Dubai, T., London, •, Madrid, •, Munich, M. •, Paris, •, Montreal, •, Toronto, •, Delhi, •, Mexico, •, São, C., Sydney, P. •, Kong, H., Seoul, •, … Tokyo, •. (2019). *Refactoring Improving the Design of Existing Code Second Edition*. www.EBooksWorld.ir

Hitpass, B. (2012). *Business process management (BPM) : fundamentos y conceptos de implementación*. BHH.

José, F., Peñalvo, G., & Pardo Aguilar, C. (n.d.). *El Principio Abierto/Cerrado*.

Kaur, G., & Singh, B. (2017). Improving the Quality of Software by Refactoring. *International Conference on Intelligent Computing and Control Systems*.

Khatchadourian, R., Moore, O., & Masuhara, H. (2016). Towards improving interface modularity in legacy Java software through automated refactoring. *MODULARITY Companion 2016 - Companion Proceedings of the 15th International Conference on Modularity*. https://doi.org/10.1145/2892664.2892681

Khatchadourian, R. T., Khatchadourian, R., & Masuhara, H. (2017). Automated Refactoring of Legacy Java Software to Default Methods. *IEEE/ACM 39th International Conference on Software Engineering,*. https://academicworks.cuny.edu/hc_pubs/287Discoveradditionalworksat:https://academicworks.cuny.edu

Lenarduzzi, V., Lomio, F., Huttunen, H., & Taibi, D. (2020). Are SonarQube Rules Inducing Bugs? *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*. https://doi.org/10.1109/SANER48275.2020.9054821

Martin, R. (1994). *OO Design Quality Metrics An Analysis of Dependencies*.

Martin, R. C. (2014). *Agile Software Development,Principles, Patterns, and Practices 1/E*. www.EBooksWorld.ir

Mazón Olivo, B., Cartuche Calva, J., Chimarro Chipantiza, V., & Rivas Asanza, W. (2015). Fundamentos de Programación Orientada a objetos en java. In *Universidad Técnica de Machala*.

Meyer, B. (n.d.). *Object-Oriented Software Construction SECOND EDITION*. http://www.tools.com

Meyer, B. (1997). *Object-Oriented Software Construction SECOND EDITION*. http://www.tools.com

Parr, T. (2013). *The Definitive ANTLR 4 Reference*.

Ricardo Tello Díaz. (2020). *Método de Refactorización de Arquitecturas de Software con Carencia de Abstracciones*. CENIDET.

Rizzi, L., Fontana, F. A., & Roveda, R. (2018). Support for architectural smell refactoring. *IWoR 2018 - Proceedings of the 2nd International Workshop on Refactoring, Co-Located with ASE 2018*, 7–10. https://doi.org/10.1145/3242163.3242165

Rodríguez, D., & Harrison, R. (n.d.). *CAPÍTULO 4 MEDICIÓN EN LA ORIENTACIÓN A OBJETOS*.

Stevens, S. S. (1946). Sobre la Teoría de las Escalas de Medición. *SCIENCE*, *103*.

Teoría de medición. (n.d.). Retrieved January 22, 2023, from http://www.sc.ehu.es/jiwdocoj/remis/docs/teoriamedicion.html

Tsantalis Nikolaos, Chaikalis Theodoros, & Chatzigeorgiou Alexander. (2018). *Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells*. IEEE.

University, A. I. (2016). Polimorfismo. In *Lenguajes de Programación Orientados a Objetos*.

Vertiz Alexis, D. (n.d.). *JUnit*. Retrieved February 1, 2021, from JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

Zimmermann, O. (2015). Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, *32*(2). https://doi.org/10.1109/MS.2015.37

Zuse, H. (1992). Properties of software measures. *Software Quality Journal*, *1*(4). https://doi.org/10.1007/BF01885772