



Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

AMBIENTE VISUAL E INTERACTIVO DE ORQUESTACIÓN DE MICROSERVICIOS DE REFACTORIZACIÓN

presentada por

Ing. Ninfa Yukary Hernandez Vargas

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México.



Centro Nacional de Investigación y Desarrollo Tecnológico
Departamento de Ciencias Computacionales

Cuernavaca, Mor., **20/AGOSTO/2026**

OFICIO No. DCC/158/2026

Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de **Ninfa Yukary Hernández Vargas** con número de control **M24CE055**, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Ambiente Visual e Interactivo de Orquestación de Microservicios de refactorización"** 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.

ATENTAMENTE

Excelencia en Educación Tecnológica®
"Conocimiento y Tecnología al Servicio de México"

Dr. René Santaolaya Salgado
Director de tesis

Dra. Blanca Dina Valenzuela Robles
Revisor 1

M.C. José Luis Ramírez Alcántara
Revisor 2

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



2026
año de
Margarita
Maza





Centro Nacional de Investigación y Desarrollo Tecnológico
Subdirección Académica

Cuernavaca Mor, 21/agosto/2026

Oficio No. SAC/265/2026

Asunto: Autorización de impresión de tesis

NINFA YUKARY HERNÁNDEZ VARGAS
CANDIDATA AL GRADO DE MAESTRA 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 **"Ambiente Visual e Interactivo de Orquestación de Microservicios de refactorización"**, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

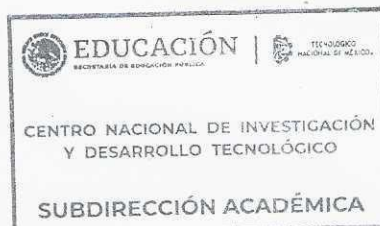
Esperando que el logro del mismo sea acorde con sus aspiraciones profesionales, reciba un cordial saludo.

ATENTAMENTE

Excelencia en Educación Tecnológica®

"Conocimiento y Tecnología al servicio de México"

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO



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

CMAZ/lmz



2026
año de
Margarita Maza

cenidet
Centro Nacional de Investigación
y Desarrollo Tecnológico



Dedicatoria

Esta tesis está dedicada a:

A mis padres, quienes han sido el fundamento de mi formación personal y profesional. Su amor, sacrificio y confianza constante han hecho posible cada uno de mis logros. Este logro es también suyo.

A mi familia, por su apoyo incondicional, comprensión y aliento permanente durante este proceso. Gracias por ser inspiración y motivación para seguir creciendo.

A mi prometido, por su apoyo y acompañamiento a lo largo de este camino académico, y por su comprensión en los momentos de mayor exigencia.

A mis amigos por apoyarme cuando más les necesité y por todo su cariño.

Agradecimientos

Agradezco profundamente a Dios por brindarme la fortaleza, la salud y la determinación necesarias para culminar esta etapa de formación profesional y personal.

Expreso mi más sincero agradecimiento a mis padres, quienes han sido el pilar fundamental de mi vida. Su apoyo incondicional, sus enseñanzas y su confianza constante han sido la base que me ha permitido alcanzar cada una de mis metas. A mi familia, por su comprensión, paciencia y respaldo durante este proceso académico exigente, gracias por acompañarme siempre.

A mi prometido, por su apoyo, comprensión y motivación a lo largo de esta etapa. Su compañía y ánimo fueron fundamentales en los momentos de mayor desafío.

A mi director de tesis, por su guía académica, sus valiosas observaciones y su compromiso con la calidad de esta investigación. Su orientación fue clave para el desarrollo y culminación de este trabajo.

Asimismo, agradezco al Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) y a los docentes del programa de maestría, quienes contribuyeron significativamente a mi formación en el ámbito de la ingeniería de software, fortaleciendo mis competencias profesionales y mi visión investigativa.

Expreso también mi agradecimiento a la Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI), por el apoyo otorgado durante mi formación académica y por contribuir al desarrollo de mis estudios de posgrado.

Finalmente, agradezco a la institución por brindarme el espacio y las herramientas para el desarrollo de esta investigación.

Resumen

Los métodos de refactorización se implementan en herramientas automatizadas capaces de identificar factores de deuda técnica en sistemas de software legado y generar código funcionalmente equivalente. Estas transformaciones tienen como finalidad disminuir defectos de diseño, la deuda técnica y reducir la entropía del software. El presente trabajo analiza el impacto de aplicar refactorizaciones de manera sucesiva dentro de una secuencia, con el propósito de evitar la pérdida de las mejoras alcanzadas en la reducción de la deuda técnica. En particular, se examina cómo el orden de ejecución de los métodos orientados a la protección modular, la carencia de abstracción y la herencia de implementación influye en el resultado final del proceso, considerando dos enfoques de ejecución: uno basado en reglas formales y otro sustentado en el criterio de expertos en ingeniería de software.

Abstract

Refactoring methods are implemented in automated tools capable of identifying technical debt factors in legacy software systems and generating functionally equivalent code. These transformations aim to reduce design defects and technical debt, as well as to decrease software entropy. This work analyzes the impact of applying refactorings successively within a sequence, with the purpose of preventing the loss of improvements achieved in reducing technical debt. In particular, it examines how the execution order of methods focused on modular protection, lack of abstraction, and implementation inheritance influences the final outcome of the process, considering two execution approaches: one based on formal rules and the other supported by the judgment of software engineering experts.

Contenido

Capítulo 1. Introducción	8
Capítulo 2. Antecedentes	9
2.1. Problema	9
2.2. Propuesta de solución	10
2.3. Justificación.....	11
2.3.1. Situación causal	11
2.4. Objetivo(s)	12
2.4.1. Objetivo general	12
2.4.2. Objetivos específicos	12
2.5. Fundamentos	12
2.5.1. Marco conceptual	12
2.5.2. Métodos de refactorización	15
2.5.3. Métricas evaluadas	16
2.6. Estado del arte.....	17
2.6.1. Trabajos relacionados	19
2.6.2. Antecedentes.....	35
Capítulo 3. Metodología.....	37
3.1. Análisis de deuda técnica derivada del orden secuencial de refactorizaciones ..	37
3.2. Evaluación de deuda técnica.....	44
Capítulo 4. Diseño de herramienta	46
4.1. Diagrama de casos de uso.....	46
4.2. Escenarios	47
4.3. Estados de objetos	57
4.4. Diagrama de secuencia: desarrollar la interfaz de usuario	61
4.5. Modelo arquitectural.....	63
Capítulo 5. Pruebas	65
5.1. Plan de pruebas.....	65
5.2. Evaluación de resultados	73
Capítulo 6. Conclusiones y trabajos futuros	77

Bibliografía.....	79
--------------------------	-----------

Anexos.....	85
--------------------	-----------

<i>Anexo 1. Manual de usuario.....</i>	<i>85</i>
--	-----------

<i>Anexo 2. Manual de instalación de microservicios de refactorización de software</i>	<i>92</i>
--	-----------

Listado de tablas

Tabla 1. Métodos de refactorización desarrollados	10
Tabla 2. Métodos de refactorización de la literatura.	15
Tabla 3. Métricas evaluadas durante el proceso de refactorización.	16
Tabla 4. Síntesis de los resultados de la búsqueda bibliográfica.....	18
Tabla 5. Tabla comparativa de trabajos relacionados.....	31
Tabla 6. Resumen de secuencias de refactorización.....	44
Tabla 7. Aplicar refactorización	47
Tabla 8. Autenticación de usuarios	49
Tabla 9. Secuenciar mediante reglas	50
Tabla 10. Secuenciar mediante criterios del usuario	52
Tabla 11. Registro de usuarios	53
Tabla 12. Permisos de acceso	54
Tabla 13. Modificar contraseña	55
Tabla 14. Evaluación de satisfacción del usuario.....	56
Tabla 15. Tabla de estados del sistema visual.....	57
Tabla 16. Tabla de estados del usuario.....	58
Tabla 17. Tabla de estados del SR2-Refactoring	59
Tabla 18. Tabla de estados del base de datos	59
Tabla 19. Tabla de estados del código.....	59
Tabla 20. Tipo de artículo.....	66
Tabla 21. Documentación de pruebas.....	66
Tabla 22. Glosario	66
Tabla 23. Características del caso de prueba ISAVICMR03-01.	69

Tabla 24. Resultados antes y después de la refactorización DBLista.	69
Tabla 25. Características del caso de prueba ISAVICMR03–02.	70
Tabla 26. Comparación de métricas del marco estadístico antes y después de la refactorización.	70
Tabla 27. Características del caso de prueba ISAVICMR03–03.	71
Tabla 28. Comparación de métricas de PSP antes y después de la refactorización	72
Tabla 29. Características del caso de prueba ISAVICMR03–04.	73
Tabla 30. Resultados antes de la refactorización.....	74
Tabla 31. Resultados después de la refactorización enfoque reglas.	75
Tabla 32. Resultados después de la refactorización enfoque criterio.	77

Listado de figuras

Figura 1. Metodología PRISMA.	19
Figura 2. Arquitectura original	37
Figura 3. Refactorización A.....	38
Figura 4. Refactorización B.....	39
Figura 5. Arquitectura original	39
Figura 6. Refactorización C.....	40
Figura 7. Refactorización A.....	41
Figura 8. Refactorización A → C	41
Figura 9. Refactorización B → C	42
Figura 10. Refactorización C → B	43
Figura 11. Refactorización A → B.....	43
Figura 12. Refactorización B → A.....	43
Figura 13. Diagrama de caso de uso.....	47
Figura 14. Diagrama de estados	60
Figura 15. Diagrama de secuencia desarrollar la interfaz de usuario	62
Figura 16. Diagrama de clases.....	64
Figura 17. Identificación única del documento	65
Figura 18. Base de datos.....	85

Figura 19. Ejecución del proyecto.....	86
Figura 20. Inicio de sesión.....	87
Figura 21. Dashboard.	88
Figura 22. Refactorización.	88
Figura 23. Evaluación.....	89
Figura 24. Tablas de datos de evaluación.....	89
Figura 25. Refactorización exitosa.	90
Figura 26. Resultados de refactorización	90
Figura 27. Archivos refactorizados.	91
Figura 28. Administrador.	91
Figura 29. Editar perfil.	92
Figura 30. Seleccionar workspace.....	93
Figura 31. Nuevo proyecto de Java.	93
Figura 32. Importar proyecto de Java.....	94
Figura 33. Uso de la función Maven Build.	95
Figura 34. Ejecución de comandos en la interfaz “Edit Configuration”	95
Figura 35. Archivo ejecutar.bat	96
Figura 36. Ejecución del orquestador	96

Capítulo 1. Introducción

Durante la década de los 70, se dio origen a lo que se conoce como crisis de software. Esta crisis se caracterizó por el hecho de que los sistemas no se desarrollaban dentro de los plazos estipulados, incumplían con los requerimientos dados por el cliente, existía alta labor de mantenimiento, además al modificar el software tendía a ser una tarea difícil y costosa [1].

Aunque la ingeniería de software ha propuesto nuevas tecnologías, modelos, técnicas, métodos, herramientas y plataformas de desarrollo para corregir las causas de esta crisis, aún persisten algunas prácticas equivocadas de desarrollo. Estas prácticas dan como resultado lo que hoy se conoce como “*Code Smell*” (código mal formado)¹, lo que produce deuda técnica que, tarde o temprano, tendrá que ser “pagada”.

El código mal formado consiste en estructuras de código que señalan la necesidad de refactorizar. Estas estructuras de código son generadas desde las primeras etapas del desarrollo e impactan negativamente en el mantenimiento, evolución y realización de pruebas de los productos de software. El código mal formado crea la necesidad de realizar un exceso de manipulación para corregir los defectos que aparecen, lo que incrementa su fragilidad y reduce el tiempo de vida útil de los activos de software de las empresas [2].

Aunque el código mal formado puede cumplir con los requerimientos iniciales, presenta problemas de calidad no deseados, como falta de protección de datos, ausencia de abstracción, exceso de dependencias funcionales y canales de comunicación, código innecesario que es dispensable, dependencias derivadas de la herencia de implementación, y una complejidad excesiva derivada del mal uso de las relaciones entre entidades en las arquitecturas del software.

La deuda técnica se relaciona, principalmente, con la mantenibilidad del software. El principio de la deuda técnica está relacionado con el término “*Code Smell*” y con el tiempo que se usa para resolver el código mal formado, además de un retrabajo continuo para mantener las aplicaciones en operación. Esto implica un aumento en el costo económico debido a las tareas de depuración y corrección del código mal formado [3].

Asimismo, la deuda técnica contribuye al incremento de la entropía del software, entendida como el nivel de desorganización o degradación estructural presente en un sistema. Conforme el software evoluciona sin aplicar mecanismos adecuados de refactorización y control de calidad, aumenta la complejidad del código y disminuyen atributos como la legibilidad, mantenibilidad y modularidad de la arquitectura del software [4].

Con el fin de controlar la deuda técnica y reducir la entropía del software, durante las actividades de mantenimiento se recurre a técnicas como la refactorización. Mientras que el mantenimiento es una actividad amplia que permite realizar mejoras, corrección de errores y defectos, así como eliminar funciones que ya son consideradas obsoletas [5].

La refactorización de software es una técnica para mejorar el diseño interno del software que sirve principalmente para reducir las tareas de mantenimiento, por lo tanto, se vuelve

¹ En este documento, la frase “Code Smell” en idioma inglés, se interpreta como “Código Mal Formado” en idioma español. Se refiere al código incorrecto, defectuoso o aquel que causa deuda técnica o entropía de software.

fundamental en lo referente al mantenimiento y evolución de software, y que permite contar con atributos de buena calidad [6].

A pesar de la existencia de herramientas que apoyan la aplicación de técnicas de refactorización, estas se orientan principalmente a la ejecución de refactorizaciones individuales y ofrecen un soporte limitado para planificar y evaluar secuencias de refactorización [7]. En particular, no proporcionan mecanismos que permitan al ingeniero de software definir el orden de aplicación de los microservicios de refactorización considerando reglas formales de secuenciación, métricas de calidad y el impacto sobre la entropía del software. Esta limitación dificulta la toma de decisiones para reducir la deuda técnica y preservar la calidad arquitectónica del sistema.

Con el propósito de atender esta problemática, la presente investigación propone el desarrollo de un entorno visual e interactivo para la secuenciación de microservicios de refactorización. La propuesta integra reglas formales de secuenciación y criterios definidos por el ingeniero de software para construir y ejecutar secuencias de refactorización.

La propuesta contribuye a minimizar la reintroducción deuda técnica durante la aplicación de múltiples refactorizaciones, al proporcionar al ingeniero de software un entorno visual e interactivo que facilite la selección y ejecución de secuencias de microservicios de refactorización con base en reglas formales de secuenciación y métricas de calidad.

El presente documento se organiza de la siguiente manera. En el Capítulo 1 se presenta la introducción de la investigación, donde se expone el contexto, la problemática, la propuesta y los objetivos del trabajo. El Capítulo 2 describe el estado del arte, así como, los trabajos relacionados. En el Capítulo 3 se presenta la metodología de la investigación, en la que se describen el análisis de la deuda técnica derivada del orden de aplicación de las refactorizaciones y el cálculo del factor de aleatoriedad. El Capítulo 4 describe el diseño de la herramienta propuesta, incluyendo el diagrama de casos de uso, los escenarios y los estados de los objetos. Finalmente, los capítulos posteriores presentan los resultados obtenidos y las conclusiones.

Capítulo 2. Antecedentes

En este capítulo se presentan los trabajos y desarrollos previos relacionados con la refactorización de software legado y la reducción de la deuda técnica. Se describen los métodos de refactorización desarrollados en el laboratorio de Ingeniería de Software del TecNM/CENIDET, así como los aspectos relacionados con su aplicación y ejecución. Estos antecedentes permiten establecer el contexto en el que surge la problemática abordada en esta investigación.

2.1. Problema

En el laboratorio de Ingeniería de Software del TecNM/CENIDET, se han desarrollado varios métodos de refactorización. El proceso de los métodos de refactorización comienza y termina con la medición del grado de deuda técnica generado por algún factor relacionado con código mal formado o defectuoso e incluye la corrección de dicho código. Para efectos de automatización y pruebas, los métodos de refactorización se implementan en herramientas automatizadas en lenguaje de programación Java. Estas herramientas

detectan los factores de deuda técnica en el código de sistemas legados de software y generan código de reemplazo funcionalmente equivalente al original. El código de reemplazo reduce o elimina los defectos de diseño originales, lo que se refleja en una disminución de la deuda técnica o de la entropía de software al final del proceso.

Un entorno que permite la aplicación de refactorizaciones a código legado, sin seguir reglas estrictas de secuenciación, o sin seguir criterios ponderados de sucesión, deja al libre albedrío del usuario el orden de ejecución. En este contexto, surge el *problema de ejecución aleatoria de secuencias de microservicios de refactorización, en la que un microservicio podría reintroducir la deuda técnica reducida o eliminada previamente por otro microservicio de refactorización, afectando la calidad del código previamente mejorado.*

En el laboratorio se ha concluido que los métodos de refactorización desarrollados como se puede observar en la Tabla 1, se adaptan de manera óptima a la arquitectura de microservicios, dado que funcionan como servicios autónomos y aislados. Esto permite beneficiarse de las ventajas propias de esta arquitectura, como la escalabilidad y la flexibilidad, además de facilitar su despliegue en la nube para ofrecer soporte a usuarios que busquen reducir la deuda técnica en sus sistemas de software en operación.

Tabla 1. Métodos de refactorización desarrollados

METODOS DE REFACTORIZACIÓN		
A. Método de refactorización de calificador de alcance	de de	El “Método de refactorización de calificadores de alcance”, analiza cada método de clases de una aplicación Java para asignarle el calificador de acceso adecuado [8].
B. Método de Refactorización de Carencia de Abstracción	de de	El “Método de Refactorización de Carencia de Abstracción” crea arquitecturas adaptables añadiendo abstracción, a partir de la identificación de clases de la cima de la jerarquía que carezcan de abstracción.
C. Reducción de Dependencia en la Herencia de Implementación	de la de	Esta refactorización invierte estas llamadas, convirtiéndolas en “callbacks” del servidor hacia el cliente, usando el patrón “Template Method” [9]

Bajo este escenario, se propuso el desarrollo de un entorno visual e interactivo, con el propósito realizar refactorización a través de dos enfoques de secuenciación orientadas a reducir la deuda técnica y preservar la calidad del software. De esta manera, la solución propuesta no solo contribuye a mejorar la calidad del código, sino que también proporciona una herramienta visual para gestionar de manera efectiva la deuda técnica en entornos de desarrollo de software.

2.2. Propuesta de solución

En este proyecto de tesis se desarrolló un entorno visual e interactivo de orquestación de microservicios de refactorización, en el que el usuario puede seleccionar el orden de ejecución de dichos servicios. Este orden es validado de acuerdo con las reglas formales

de secuenciación definidas en el trabajo de tesis de doctorado “Coreografía de microservicios de refactorización de software legado” en desarrollo por Nélida Barón Pérez. Así mismo, el usuario también tiene la opción de ejecutar los microservicios según su propio criterio como ingeniero de software, fundamentado en las mejores prácticas de diseño, dando prioridad a determinados factores de calidad en detrimento de otros.

2.3. Justificación

La refactorización de software es una actividad crucial dentro del ciclo de vida del software, orientada a mejorar el diseño interno del sistema. Esta práctica es fundamental en las fases de mantenimiento y evolución, pues permite optimizar el código sin alterar su funcionalidad. Además, se realiza considerando los atributos de calidad [6], lo que contribuye a un software más eficiente, fácil de mantener y adaptable a futuros cambios. Sin embargo, uno de los desafíos más importantes que enfrentan los desarrolladores es el alto costo de tiempo, recursos y la introducción de más errores o defectos, que implica un proceso manual de refactorización, especialmente cuando se trata de proyectos medianos o grandes. En estos casos, el tiempo estimado para realizar la refactorización de un proyecto mediano, con alrededor de 30,000 líneas de código, puede ser de aproximadamente seis semanas [10].

Las refactorizaciones automatizadas han adquirido relevancia como mecanismo para la mejora de la calidad del software de manera eficiente y consistente. A diferencia de las refactorizaciones manuales, donde la intervención del desarrollador puede introducir errores o inconsistencias, las técnicas automatizadas reducen significativamente la participación humana durante el proceso, permitiendo aplicar transformaciones estructurales de forma controlada.

Las técnicas de refactorización no tienen un impacto uniforme en todos los atributos de calidad debido a que al mejorar ciertos atributos de calidad se pueden degradar otros [11].

Algunos entornos de desarrollo integrados (IDEs) como Eclipse y IntelliJ incluyen herramientas para la refactorización automática, que ofrecen soporte básico. Dichas herramientas no son utilizadas debido a sus limitaciones, especialmente en términos de personalización [7].

Al carecer de herramientas adecuadas, los desarrolladores a menudo se ven obligados a realizar las modificaciones necesarias de forma manual, lo cual puede llevar a la introducción de defectos y errores que resultan en porciones de código incorrectamente refactorizado [12] [13], lo cual se ve reflejado en la fragilidad y disminución del tiempo de vida útil del software.

Por otro lado, las herramientas de refactorización a pequeña escala suelen centrarse únicamente en eliminar código mal formado específico o para la mejora de calidad en particular del código, sin darle prioridad a las reglas de secuenciación [14].

En este contexto, la creación de herramientas que permitan secuencias automatizadas de refactorizaciones, que den como resultado la reducción del tiempo y la disminución de errores humanos derivados de refactorizaciones manuales, son aspectos clave para mejorar tanto la productividad como la calidad del producto de software.

2.3.1. Situación causal

Debido al aislamiento y la autonomía de los métodos de refactorización existentes,

actualmente no existen reglas de sucesión para su ejecución conjunta, lo que dificulta la reducción eficiente de los factores de deuda técnica. Esto implica que los métodos pueden aplicarse en cualquier orden, sin una coordinación específica. En particular, en el laboratorio de Ingeniería de Software del TecNM/CENIDET, se han desarrollado varios métodos de refactorización que trabajan de manera aislada para eliminar la deuda técnica generada por el código mal formado.

Cada uno de estos métodos ejecuta su propio analizador estático para extraer, desde el código del software en estudio, la información necesaria. Esto incluye tanto los datos necesarios para la evaluación de la métrica asociada al factor de calidad que causa el código mal formado, como la información necesaria para la automatización del método de corrección. Además, cada método de refactorización cuenta con su propia estructura de datos, que alberga la información recabada durante el análisis.

Este escenario sugiere que dichos métodos de refactorización podrían ser transformados en microservicios, y permitir la automatización de su ejecución mediante una orquestación secuencial. Un ambiente gráfico, visual e interactivo basado en reglas facilita esta automatización, advirtiendo sobre posibles ejecuciones incorrectas que podrían dar lugar al problema planteado de reintroducción de deuda técnica

2.4. Objetivo(s)

2.4.1. Objetivo general

El principal objetivo de este proyecto de tesis es “Controlar el orden de ejecución sucesiva de refactorizaciones en aplicaciones de software legado”, mediante dos enfoques alternativos. Un enfoque es utilizando reglas de aserción entre la coreografía de microservicios. El segundo enfoque consiste en permitir la sucesión de refactorizaciones mediante la priorización ponderada de factores de calidad interna, criterios y principios de diseño.

2.4.2. Objetivos específicos

1. Disminuir el factor de aleatoriedad en la aplicación sucesiva de refactorizaciones en aplicaciones legadas de software.
2. Análisis de permutaciones en las secuencias de refactorización en software, para conservar la secuencia óptima entre microservicios de refactorización.

2.5. Fundamentos

2.5.1. Marco conceptual

En esta sección se presentan los principales conceptos relacionados con el desarrollo de la investigación. Se comprende los temas abordados, particularmente aquellos relacionados con la refactorización de software, la deuda técnica, los microservicios y la calidad del software. Estas definiciones permiten establecer una base común para la comprensión de los conceptos utilizados a lo largo de la tesis.

<u>Refactorización</u>	La refactorización es el proceso en el cual se realizan cambios en un sistema de software de manera que no altere la forma en que funciona externamente el código, pero si mejore su estructura interna. Durante la refactorización se puede tomar un mal diseño y convertir en un código con una estructura correcta [15].
<u>Código mal formado</u>	El código mal formado son estructuras de código que impactan negativamente, aumentando la deuda técnica, dificultando el mantenimiento y la evolución del software [2].
<u>Coreografía</u>	Las coreografías son planes de coordinación para sistemas concurrentes, dentro de la programación son utilizadas para tener un acuerdo de las interacciones entre los actores según sus puntos iniciales y finales en la comunicación, que deben realizar para lograr el objetivo que tienen en común. Esto es difícil ya que se debe tener en cuenta cómo interactúan múltiples lenguajes de programación [16].
<u>Aplicaciones Monolíticas</u>	Son las aplicaciones que están conformadas por tres partes las cuales constan de la interfaz de usuario, lógica de negocios y gestión de datos, al estar realizadas en una sola unidad estas limitan las herramientas que se disponen para su realización, dentro de las ventajas que estas tienen es que tienen un menor costo [17].
<u>Microservicios</u>	La arquitectura orientada a servicios (SOA) y el estilo arquitectónico de microservicios se basan en el uso de servicios como el principal componente para implementar funcionalidades de negocios. Los microservicios pueden realizar una comunicación entre ellos, utilizando protocolos estandarizados como es el caso de HTTP y REST, una de sus características más sobresalientes es la descomposición de aplicaciones en componentes de manera que puedan desarrollarse fácilmente [18].
<u>Reutilización</u>	Se refiere a la medida en la que los componentes de un sistema pueden ser asimilados por componentes dentro del mismo u otro sistema [11].
<u>Cohesión</u>	Se realiza calculado con la relación entre los métodos de una clase basándose en una lista de parámetros para los métodos [19]. En el contexto de esta tesis, cohesión es el grado de relación entre las funciones de una clase de objetos por su acceso a atributos compartidos.
<u>Polimorfismo</u>	Se entiende como polimorfismo a la característica de tiempo de ejecución con respecto a un objeto al comprender un mensaje enviado por otro objeto [20]. En el contexto de esta tesis, polimorfismo es la flexibilidad de un objeto de cambiar su comportamiento funcional de conformidad con su tipo asumido en tiempo de ejecución.
<u>Acoplamiento</u>	Se utiliza para contar diferentes números de clases a las que una clase está directamente vinculada. Es una propiedad fundamental para los sistemas de software, entre los componentes de software también influye en el mantenimiento y la evolución del software [21] [19].

<u>Herencia</u>	Es una noción semántica por la cual se considera que las propiedades y restricciones de una subclase incluyen las responsabilidades de una superclase [22].
<u>Jerarquía</u>	En el contexto de esta tesis representa una taxonomía de clases y subclases relacionadas por herencia en una arquitectura orientada a objetos [19].
<u>Complejidad</u>	Se define como el grado en que el diseño o código de un sistema es difícil de entender debido a numerosos componentes [22].
<u>Mensajería</u>	Cuenta el número total de métodos públicos de una clase [19].
<u>Composición</u>	Es el grado de relación entre clases que se conoce mediante el uso de atributos de referencia [19].
<u>JDeodorant</u>	Se trata de un complemento para Eclipse, el cual permite detectar automáticamente los antipatrones de diseño y ofrece posibles refactorizaciones para corregirlos [23].
<u>Feature Envy</u>	Se centra en acceder frecuentemente a datos de otras clases [24].
<u>God Class</u>	Antipatrón donde la Clase de un objeto controla excesivos objetos en el sistema [24].
<u>Long methods</u>	Métodos de clases de objetos que son extensos y que dificultan la legibilidad, reutilización y capacidad de mantenimiento del código [25].
<u>Long parameter lists</u>	Métodos con argumentos excesivos [26].
<u>Refused bequest</u>	Subclases que solo utilizan parcialmente las funciones o propiedades heredadas del padre [27].
<u>Shotgun surgery</u>	Se refiere a situaciones en las que los cambios de un solo requisito afectan al código en varias clases [26].
<u>Cobertura de pruebas</u>	Es utilizada, principalmente, para determinar si los casos de prueba realmente cubren el código del software [28].
<u>Usabilidad</u>	La usabilidad es la medida de facilidad de uso de algo, es un emblema de calidad, hace referencia a estrategias que se utilizan para la mejora continua y contiene cinco componentes los cuales son aprendizaje, eficiencia, memoria, errores y satisfacción [29].
<u>Reglas heurísticas de Jacob Niesel</u>	Jakob Nielsen, a través de sus diez heurísticas de usabilidad proporciona un conjunto de principios que guían el diseño de interfaces intuitivas y eficientes [30].
<u>Aserción</u>	Se remonta a la hipótesis de Williamson (1996), según la cual la posibilidad de afirmar una proposición depende del cumplimiento de una regla sencilla [31].

2.5.2. Métodos de refactorización

Con el propósito de identificar las técnicas de refactorización más utilizadas para corregir problemas de calidad interna del software, se analizaron diversos estudios que relacionan los code smells con los métodos de refactorización, las métricas empleadas para evaluar su impacto y las herramientas que ofrecen soporte para su aplicación. La Tabla 2 resume los principales métodos de refactorización identificados en la literatura, así como los code smells que permiten corregir, las métricas utilizadas para evaluar su efectividad y las herramientas que los implementan.

Tabla 2. Métodos de refactorización de la literatura.

CODE SMELL / CÓDIGO MAL FORMADO	MÉTODO DE REFACTORIZACIÓN	HERRAMIENTAS
Long method, God class, Duplicated code [32], Feature Envy [33]	Extract Method o Método de extracción: Creación de un nuevo método extrayendo una sección de código de un método existente [34]. Métricas: Complejidad ciclomática o cognitiva, Número de líneas de código, Falta de cohesión [35].	Refactoring-Bot [36], JDeodorant, MultiRefactor [37] Eclipse, IntelliJ [7]
Feature Envy [33]	Inline Method o Método en línea (IM): Reemplaza las llamadas y los usos de un método, eliminar su declaración [34]. Métricas: Cohesión, Complejidad, Mensajería. [19].	MultiRefactor, jSparrow [37]
Long Parameter List [33]	Add Parameter o Agregar parámetro (AP): Se proporciona un parámetro para un objeto que pueda transmitir estos datos [11]. Métricas: Tamaño del diseño, la encapsulación y la mensajería [19].	Towards a Live Environment for Code Refactoring [35]
Long Parameter List [33]	Remove Parameter o Eliminar parámetro (RP): Es utilizado cuando el cuerpo del método ya no se utiliza en el parámetro [11]. Métricas: Tamaño del diseño, la encapsulación y la mensajería [19].	MultiRefactor [37]
Data class [38]	Encapsulate Field o Encapsular campo (EF): Al modificar la accesibilidad de los campos públicos, este método se aplica para restringir el acceso a los datos [11]. Métricas: Encapsulación, Complejidad, Mensajería [19].	MultiRefactor [37]
Feature Envy, God class [32], Shotgun Surgery [33]	Move Method o Mover método: Cambia la declaración de un método, de una clase a otra [34]. Métricas: Cohesión y la mensajería [19].	Towards a Live Environment for Code Refactoring [35], Jdeodorant,

		REFBOT, TrueRefactor [33], MultiRefactor [37]
Duplicated Code [33]	Pull-up Method: Mover un método hacia arriba en la cadena de herencia, desde una clase hijo hacia una clase padre [34]. Métricas: Número de declaraciones, tamaño promedio de las declaraciones, la falta de cohesión [39].	REFBOT [10], MultiRefactor [37]
Duplicated Code [33]	Push-down Method: Desplaza hacia abajo un método de la cadena herencia de una clase primaria a una clase secundaria [34]. Métricas: Complejidad, Herencia, tamaño del diseño [40].	REFBOT [10], MultiRefactor [37]
Duplicated Code [33]	Rename Method o Cambiar nombre de método (MM): Cambia el nombre de un identificador de método a uno diferente [34]. Métricas: Complejidad, Legibilidad [19].	MultiRefactor [37]
Data class [38]	Hide Method u Ocultar método (HM): Esta técnica es utilizada para proteger o hacer que el método sea privado. Métricas: Mensajería [11].	Eclipse, IntelliJ [7]
Lazy Class [33]	Inline Class o Clase en línea (IC): Se realiza la copia de todos los métodos y campos de la clase en otra. Métricas: Cohesión [11].	TrueRefactor [33], MultiRefactor [37]
Shotgun surgery [32]	Move Field o Mover campo (MF): Es un método en el que se mueve un campo de una clase principal a una clase apropiada. Métricas: Encapsulación (DAM) [11].	Eclipse, IntelliJ [41]

2.5.3. Métricas evaluadas

Para evaluar el impacto de las secuencias de microservicios de refactorización sobre la calidad del software, se emplea un conjunto de métricas orientadas a medir diferentes atributos relacionados con la modularidad, la abstracción, la herencia, el polimorfismo y la flexibilidad de la arquitectura. Estas métricas permiten cuantificar los cambios producidos por las refactorizaciones y proporcionan información objetiva para analizar la evolución de la calidad del sistema. En la Tabla 3 se presentan las métricas consideradas en esta investigación y su correspondiente descripción.

Tabla 3. Métricas evaluadas durante el proceso de refactorización.

METRICAS EVALUADAS	
AF	Factor de Abstracción.
FFC	Factor de Flexibilidad de Clases.

FHI	Factor de Herencia de Implementación.
FHIAC	Factor de Herencia de Implementación de una Arquitectura de Clases.
FHIJ	Factor de Herencia de Implementación de una Jerarquía de Clases.
FMFAC	Factor Medio de Flexibilidad de Arquitecturas de Clases.
PF	Factor de Polimorfismo.
PM	Protección Modular.
PMFF	Protección Modular de Funciones Friendly.
PMFP	Factor de Protección Modular de Funciones Privadas.
PMFPR	Protección Modular de Funciones Protegidas.
TPM	Total de Protección Modular.

2.6. Estado del arte.

El estado del arte es derivado de la Revisión Sistemática de la Literatura (RSL). Esta revisión sistemática fue desarrollada de acuerdo con las metodologías (Prisma, Kitchenham) como se describe a continuación.

Preguntas de Investigación

- **¿Qué herramientas visuales e interactivas han sido propuestas o desarrolladas para la gestión de refactorización y deuda técnica?**

Dentro de las herramientas propuestas se encontraron RefBot: Intelligent Software Refactoring Bot [10], Towards a Live Environment for Code Refactoring [35] y DART (Detecting and Refactoring Tool) [42].

- **¿Existen herramientas de automatización de refactorización de code smell en la literatura?**

Existen herramientas de automatización; sin embargo, muchas de ellas no permiten modificaciones o, si las permiten, son básicas, como es el caso de Eclipse e IntelliJ [7]. Algunas herramientas que se enfocan exclusivamente en la refactorización son MultiRefactor, JDeodorant, jSparrow y Spartenizer. La más destacable entre ellas es MultiRefactor, ya que es la única que permite la refactorización en múltiples lenguajes [37].

- **¿Existen reglas de ejecución sucesiva de refactorización de código en la literatura?**

En la gran mayoría de los artículos no se hace mención específicamente de reglas de ejecución. Sin embargo, en el artículo "HSHEP: An Optimization-Based Code Smell Refactoring Sequencing Technique" se menciona que las técnicas de secuenciación de refactorización mitigan problemas mejorando la estructura interna de un sistema [43].

Cadena de búsqueda

La cadena de búsqueda se definió con el propósito de identificar trabajos de investigación relacionados recientes que permitan constatar la importancia del tema de estudio. Para su construcción se seleccionaron términos asociados con los principales elementos de la investigación, como la refactorización de software, el software legado, la automatización, la

secuenciación y los microservicios.

("code refactoring") AND ("software quality" OR "quality attributes" OR "code smells") AND ("customization" OR "batch refactoring" OR "refactoring sequencing" OR "refactoring ordering" OR "interactive refactoring" OR "automated refactoring") AND ("tool" OR "framework" OR "approach" OR "model" OR "empirical study" OR "case study")

Búsqueda inicial

La cadena de búsqueda se aplicó en bases de datos académicas ampliamente reconocidas como IEEE Xplore, Google académico (PLOS ONE, ACM Transactions on Software Engineering and Methodology, Springer), las cuales proporcionan acceso a literatura científica de alta calidad en el ámbito de la ingeniería de software.

La selección de los estudios se realizó utilizando las herramientas de búsqueda avanzada disponibles en cada plataforma. La última búsqueda se llevó a cabo en agosto de 2025, considerando publicaciones científicas en revistas y conferencias correspondientes a los últimos seis años.

Como resultado de la aplicación de la cadena de búsqueda, se obtuvo un total de 729 publicaciones como se puede ver en la Tabla 4.

Tabla 4. Síntesis de los resultados de la búsqueda bibliográfica

BASE DE DATOS	RESULTADOS DE LAS BÚSQUEDAS
Google Academico	727
IEEE	2
Total	729

Criterios de calidad y depuración de los estudios

Para evaluar la calidad de los estudios, se realizaron las siguientes preguntas: 1) ¿El estudio menciona alguna herramienta de refactorización? 2) ¿El estudio menciona alguna secuencia de refactorización o menciona su importancia? Estos cuestionamientos, se alinearon con las preguntas de este estudio sistemático para obtener calidad en los estudios seleccionados.

Para el primer refinamiento de los estudios, se aplicaron los siguientes criterios:

Criterios de inclusión: artículos de estudios primarios publicados en revistas científicas, artículos de conferencias, trabajos publicados del 2019 al 2025, artículos clásicos que son base del tema a tratar.

Criterios de Exclusión: Se excluyen los estudios secundarios, libros, patentes, capítulos de libros, tesis y material didáctico, páginas web y presentaciones en Power Point, opiniones personales en blogs de internet, artículos duplicados y artículos no escritos en idioma inglés.

Artículos seleccionados: Identificados: 729, Duplicados eliminados: 129, Registros únicos: 600, Excluidos por título/resumen: 550, Evaluados a texto completo: 50, Excluidos por texto completo: 18

Como resultado, se seleccionaron 32 estudios. Se realizó una revisión exhaustiva del resumen (abstract) de cada estudio para garantizar su relevancia con el tema y su capacidad de responder las preguntas de investigación, como se puede observar en la Figura 1.

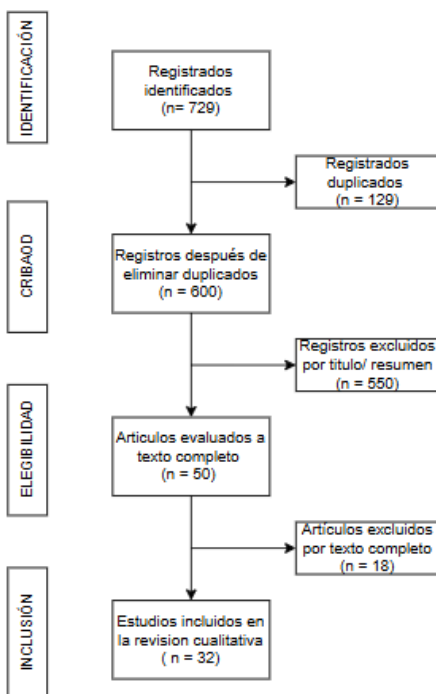


Figura 1. Metodología PRISMA.

2.6.1. Trabajos relacionados

On the Customization of Batch Refactoring [9]

Las herramientas actuales no permiten a los desarrolladores crear refactorizaciones personalizadas. Este artículo propone aplicar la refactorización de forma personalizada, lo que permite al desarrollador realizar refactorizaciones individuales o en lote según la necesidad, además de reutilizar las refactorizaciones en contextos similares.

Aunque entornos de desarrollo como Eclipse o IntelliJ ofrecen herramientas para refactorización automática, estas tienden a quedar en desuso debido a sus limitaciones, especialmente por la falta de opciones de personalización. Por ello, se plantea la necesidad de herramientas flexibles, donde cada refactorización se descomponga en un conjunto de modificaciones menores, denominadas "modificaciones primitivas". Por ejemplo, una transformación como "Extraer Método" podría dividirse en una serie de estas modificaciones primitivas.

En conclusión, es importante contar con herramientas que permitan la personalización en las refactorizaciones. Con este enfoque, se busca diseñar e implementar un complemento para Eclipse que responda a esta necesidad.

RefBot: Intelligent Software Refactoring Bot [10]

Existe la necesidad de crear herramientas que soporten los procesos de integración continua. En este artículo se presenta un bot diseñado para monitorear de forma continua los

repositorios, analizar los archivos modificados durante las solicitudes de extracción y ofrecer recomendaciones de refactorización. El desarrollador tiene la opción de revisar estas sugerencias y decidir si desea aceptarlas o ignorarlas.

La refactorización, aunque es esencial, es un proceso costoso en términos de tiempo y recursos. Por ejemplo, en proyectos de tamaño medio (aproximadamente 30,000 líneas de código), puede tomar hasta seis semanas completar este proceso. RefBot implementa la metodología de evaluación de calidad QMOOD (Quality Model for Object-Oriented Design). Esta metodología evalúa atributos como Reusabilidad, Flexibilidad, Comprensibilidad, Funcionalidad, Extensibilidad y Eficiencia.

RefBot genera o modifica operaciones de refactorización de forma aleatoria durante su inicialización. Además, el número de operaciones seleccionadas se determina aleatoriamente dentro de un rango predefinido. Para garantizar la viabilidad de las soluciones propuestas, se aplican precondiciones y postcondiciones específicas.

Como conclusión podemos decir que RefBot automatiza el análisis y la optimización con reglas predefinidas, no le da tantas opciones al desarrollador de realizar cambios.

The untold story of code refactoring customizations in practicen [13]

En los IDEs populares, como Eclipse y NetBeans, existen herramientas de refactorización que permiten realizar modificaciones estándar en el código, ofrecen un soporte automatizado básico. Sin embargo, estas herramientas no son ampliamente utilizadas debido a la falta de personalización. Por ello, este artículo propone ajustar y modificar las refactorizaciones propuestas según las necesidades y preferencias, incluyendo la capacidad de aceptar, rechazar o modificar cada refactorización. Como resultado, los programadores se ven obligados a realizar los procesos de refactorización manualmente.

El análisis realizado en este artículo se centró en cuatro tipos comunes de refactorización: Extract Method, Inline Method, Pull Up Method y Move Method. Estos tipos fueron seleccionados por dos razones principales: en primer lugar, porque permiten generar diferentes alcances según el contexto; y, en segundo lugar, porque se enfocan en refactorizaciones complejas.

Además, algunos estudios mencionan el concepto de refactorización "Floss", que se refiere a la posibilidad de realizar refactorizaciones junto con otras actividades de desarrollo, como agregar nuevas funciones o corregir errores.

En la conclusión se menciona que es fundamental contar con herramientas que no solo soporten refactorizaciones estándar, sino que también permitan una personalización avanzada. Aunque en este artículo no se han presentado nuevas herramientas, se destaca la importancia de contar con soporte adecuado para personalizar las refactorizaciones, uno de los puntos importantes que no se hace mención es la falta de una secuenciación adecuada en las herramientas aparte de la personalización.

Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems [11]

En este artículo se menciona que la refactorización carece de exploración en profundidad detrás de los impactos constantes de los métodos de refactorización en la mejora de calidad. El modelo de calidad para el diseño orientado a objetos (QMOOD), se emplea como

herramienta de evaluación para determinar la influencia de las técnicas de refactorización en atributos de calidad.

En el texto se menciona que el uso de las herramientas de refactorización actuales puede conducir a porciones de código refactorizadas incorrectamente. Los autores realizaron un estudio que incluyó la selección de diez métodos de refactorización, los cuales son: Agregar parámetro (AP), Encapsular campo (EF), Extraer clase (EC), Extraer superclase (ESP), Ocultar método (HM), Clase en línea (IC), Método en línea (IM), Mover campo (MF), Eliminar parámetro (RP) y Cambiar nombre de método (MM).

Para realizar el estudio fue necesario un modelo de calidad para ello se utilizó QMOOD, con la que se evaluaron seis características, así como once aspectos de diseño externo.

Dentro de los aspectos externos se encuentran Reutilización, Flexibilidad, Eficacia, Extensibilidad, Comprensibilidad y Funcionalidad. En el caso de las características internas se encuentran el tamaño de diseño, abstracción, encapsulación, cohesión, polimorfismo, acoplamiento, herencia, jerarquía, complejidad, mensajería, composición.

En este artículo se menciona que la refactorización puede ser realizada de manera manual o con el uso de herramientas. JDeodorant es la herramienta de refactorización más utilizada. Los resultados mostraron que en Agregar Parámetro se realizó la mejora de la Funcionalidad, Comprensibilidad y la reutilización, mientras que en el Índice de Calidad Total al aumentar la cohesión.

Como conclusión, el artículo resalta el impacto de los métodos de refactorización en las características de calidad del software. Además, la carencia de herramientas adecuadas impone a los desarrolladores realizar refactorizaciones manualmente. Esto deja como evidencia la necesidad de contar con herramientas más avanzadas que permitan una personalización efectiva de los procesos de refactorización.

How Does Incomplete Composite Refactoring Affect Internal Quality Attributes? [44]

Este artículo identifica las formas más comunes de composiciones incompletas y su efecto en atributos de calidad, como el acoplamiento y la cohesión. La eliminación de code smells es de gran importancia para los desarrolladores, se tratan dos tipos de código mal formado los cuales son Feature Envy y God Class, además de cuatro atributos de calidad interna. Se menciona que en las herramientas de refactorización existentes ofrecen poca ayuda para completar compuestos incompletos, en el caso de los atributos de calidad interna se detectan problemas en el código fuente y esto perjudica la comprensibilidad de los programas.

En cuanto a los atributos de calidad se tuvieron en cuenta Cohesión, Acoplamiento, complejidad y Tamaño, se usó una herramienta llamada SciTools Understand para recopilar las métricas de código.

Se espera que los métodos Mover mejoren la cohesión y el acoplamiento al trasladar métodos y disminuir la dependencia entre las clases. Como resultado, se observó que cada métrica puede reducirse o aumentarse después de la refactorización; se descubrió que más del 52% de las métricas de atributos de calidad interna empeoraron y un 22% de composiciones incompletas aumentaron en cuanto a las mediciones de los atributos de calidad.

En resumen, el artículo reafirma la importancia de la eliminación de code smells para la

mejora de la calidad del software mediante el uso de composiciones adecuadas.

A framework to improve quality of a Java system by performing refactoring [32]

En este artículo se propone un marco que permite identificar problemas de diseño y priorizarlos según la experiencia de los desarrolladores, con el objetivo de mejorar la calidad y la refactorización, especialmente en sistemas de software Java. Se llevó a cabo una refactorización sistemática utilizando métricas clave y considerando las opiniones de los desarrolladores.

El marco es capaz de detectar siete tipos clave de olores de código: feature envy, god class, long methods, long parameter lists, refused bequest, shotgun surgery y duplicated code. Se utilizan métricas de calidad como la complejidad ciclomática, la cobertura de pruebas y la mantenibilidad.

En resumen, el artículo revisado propone un marco amplio para identificar y abordar problemas de diseño, basándose en la experiencia y las métricas de calidad.

Refactoring effect on internal quality attributes: What haven't they told you yet? [41]

Este artículo realiza un estudio cuantitativo sobre el efecto de la refactorización en los atributos de calidad interna, evaluando cinco atributos: cohesión, complejidad, acoplamiento, herencia y tamaño. Se esperaba que cada tipo de refactorización afectará de manera diferente a los atributos de calidad interna.

La re-refactorización afecta a los atributos de calidad interna de manera similar a la refactorización simple, empeorando en algunos casos muchos atributos como la cohesión y el acoplamiento. Las tácticas de refactorización, como la refactorización "floss" acompañada de otros cambios aplicados junto con las operaciones de refactorización, también podrían afectar estos atributos.

Se utilizó RefactoringMiner la cual es una herramienta que permite detectar 11 tipos de refactorización. En el artículo se consideran los atributos críticos al aplicar refactorizaciones y re-refactorizaciones, destacando que algunas operaciones, aunque parezcan inofensivas, afectan negativamente la calidad interna del código. Es crucial tener en cuenta la elección de herramientas para la investigación de refactorización.

Según los resultados, tipos de refactorización como Extract Interface, Extract Method, Move Field, Move Method y Push Down Method mejoraron exactamente un atributo, mientras que Inline Method y Push Down Field mejoraron dos. En conclusión, los atributos de calidad son fundamentales a la hora de identificar oportunidades para la refactorización. Los IDEs y herramientas como Eclipse permiten personalizar la refactorización según las necesidades de los desarrolladores.

Towards a Live Environment for Code Refactoring [35]

Este artículo presenta una herramienta visual para la refactorización de código utilizando el método de extracción. Esta herramienta permite hacer recomendaciones y aplicar cambios para optimizar el código. Para llevar a cabo este proceso, se analizaron métricas relevantes para la calidad del software, enfocándose en proyectos desarrollados en Java.

Se seleccionaron códigos que podrían ser candidatos a refactorización, basándose en métricas como la complejidad ciclomática, el número de líneas de código y la falta de

cohesión en las clases. Se priorizaron aquellos códigos con mayor complejidad ciclométrica para determinar cuáles serían refactorizados.

La evaluación se realizó en cuatro grupos: A1, A2, A3 y B. El grupo A1 utilizó la herramienta desarrollada en el estudio, A2 una versión modificada, A3 una versión que listaba candidatos a refactorización en un archivo HTML, y B no utilizó ninguna herramienta de refactorización. Los resultados indicaron que el grupo A1 obtuvo la mayor mejora en la calidad del código. En conclusión, esta herramienta se centra en la implementación de una herramienta específica para optimizar el código, basándose en métricas, no se centra en los atributos de calidad.

A refactoring categorization model for software quality improvement [19]

En este artículo se propone un modelo que permite a los ingenieros de software mejorar los aspectos de calidad interna de los sistemas de software mediante la elección y aplicación de técnicas de refactorización pertinentes en contextos adecuados. El modelo propuesto funciona como una guía, proporcionando a los ingenieros una comprensión mejorada de las correlaciones entre las refactorizaciones y las características de calidad. Se eligió el modelo QMOOD, que incluye 11 características internas de calidad, ofreciendo una visión más amplia de la calidad del software. Los atributos considerados son abstracción, encapsulamiento, tamaño de diseño, herencia, complejidad, jerarquías, polimorfismo, cohesión, acoplamiento, mensajería y composición. Se utilizó el complemento Eclipse Metrics 1.3.8 para la recopilación automática de métricas, como Tamaño del diseño (DSC), Jerarquías (NOH), Abstracción (ANA), Encapsulación (DAM), Acoplamiento (DCC), Cohesión (CAM), Composición (MOA), Herencia (MFA), Polimorfismo (NOP), Mensajería (CIS) y complejidad (NOM). Cada técnica de refactorización se aplicó de forma independiente para determinar su influencia en las características de calidad interna.

La técnica Extract Method (EM) mejoró la mensajería, pero disminuyó la cohesión y aumentó la complejidad. La técnica Inline Method (IM) mejoró la cohesión y redujo la complejidad, mientras que la técnica Move Method (MM) mejoró la cohesión y la mensajería.

Los IDE modernos y otras herramientas de ingeniería de software pueden beneficiarse de esta categorización. Por ejemplo, un IDE podría proporcionar sugerencias de refactorización basadas en los atributos de calidad interna que necesitan mejora en la base de código.

En conclusión, la relación entre los atributos de calidad interna y la categorización de técnicas de refactorización proporciona un enfoque estructurado para mejorar la calidad del software. Se demuestra que las técnicas de refactorización no tienen un impacto uniforme en todos los atributos de calidad; en cambio, algunas mejoran ciertos aspectos mientras que pueden degradar otros.

Reducing redundancy of test cases generation using code smell detection and refactoring [42]

Este artículo se centra en la detección y refactorización de código para reducir el "code smell". Para ello, se emplea una herramienta conocida como DART (Detecting and Refactoring Tool), que se especializa en la eliminación de clases inactivas, métodos pequeños y código duplicado. El enfoque propuesto consiste en detectar la presencia de code smell; si

existe, se realiza la refactorización y se generan los casos propuestos.

Para la detección de código mal formado, es fundamental contar con reglas de detección adecuadas. En este estudio, se utilizaron propiedades de cohesión y dependencia de métodos, así como el análisis de líneas lógicas de código. Los resultados obtenidos mediante DART indican una reducción de hasta el 28% en la redundancia de código.

Se puede concluir que este artículo se centra en la implementación de la herramienta DART, para detectar y refactorizar el código mal formado descrito, logrando una reducción significativa en la redundancia del código.

Next Generation Automated Software Evolution Refactoring at Scale [45]

Este artículo se centra en la necesidad de herramientas que permitan automatizar tareas o actividades complejas, considerando la refactorización automatizada. Herramientas que permiten a los desarrolladores expresar y aplicar cambios de diseño correctamente en la mayoría de los casos. Se categorizan en refactorizaciones “Floss” y “Roo-canal”, donde la mayoría de las herramientas que facilitan estas refactorizaciones se enfocan en la mejora de métricas o la eliminación de code smells.

Definen el término “refactorización” como la reestructuración del software y comentan que a menudo se asocia con lograr cambios funcionales (reutilizando capacidades significativas) o arquitectónicos (migrando a microservicios). Los autores mencionan que actualmente, se está desarrollando un asistente automatizado de refactorización, abordando escenarios comunes en la industria, como la división de monolitos en microservicios, la extracción de características para reutilización y el reemplazo de características.

En conclusión, las herramientas que asisten en la resolución de problemas de refactorización son esenciales, ya que permiten reducir los costos asociados a estas tareas.

A Comparative Study of Automated Refactoring Tools [37]

El objetivo de esta investigación es realizar un estudio comparativo en el que se evaluaron cuatro herramientas de refactorización: MultiRefactor, JDeodorant, jSparrow y Spartenizer. Para llevar a cabo esta evaluación, se utilizaron cinco proyectos de código abierto.

Se identificaron siete categorías para los enfoques de refactorización: aprendizaje automático (ML), optimización, gráficos, métricas de software, reglas, tokens, texto y árboles. En esta investigación, se consideraron cuatro enfoques: basados en búsqueda, agrupamiento, reglas y texto.

Se utilizó iPlasma para identificar 11 tipos de "code smells" los cuales son God Class (GC), Refused Parent Bequest (RPB), Intensive Coupling (IC), Extensive Coupling (EC), Feature Envy (FE), Brain class (BC), Brain Method (BM), Shotgun Surgery (SS), Futile Hierarchy (FH), Tradition Breaker (TB) y Data Class (DC). Ref-Finder para una detección completamente automática.

Los resultados revelan que MultiRefactor destaca por ser la única que permite la refactorización en múltiples lenguajes de programación. Tanto MultiRefactor como Spartenizer son completamente automatizadas, mientras que JDeodorant y jSparrow ofrecen un enfoque semiautomatizado, permitiendo la personalización. jSparrow se destacó por superar a las demás herramientas en características tanto cuantitativas como cualitativas.

Empirical Investigation of the Diverse Refactoring Effects on Software Quality: The Role of Refactoring Tools and Software Size [12]

Este estudio evaluó dos factores clave: las herramientas de refactorización y el tamaño del software. Para ello, se seleccionaron cuatro métodos de refactorización: Remove Parameter, Extract Class, Add Parameter y Extract Superclass.

Más del 80% de los gastos asociados con el desarrollo de software se destinan a esfuerzos de mantenimiento y evolución. Las herramientas de refactorización, integradas en entornos de desarrollo como Eclipse, IntelliJ IDEA y Visual Studio, pueden ahorrar mucho tiempo, pero también son propensas a fallas, lo que puede resultar en porciones de código refactorizado incorrectamente. Aunque existen herramientas como Eclipse Refactor y JDeodorant que soportan algunos métodos, no hay ninguna herramienta que abarque todos los métodos de refactorización.

Como conclusión, este estudio destaca la importancia de los efectos de la refactorización en la calidad del software. Se recomienda que, al realizar refactorizaciones con herramientas, se lleve a cabo una verificación manual posterior, ya que los efectos pueden variar significativamente.

Interactive and Dynamic Multi-Objective Software Refactoring Recommendations [46]

Se han realizado diferentes técnicas y herramientas que permiten realizar la refactorización, estos enfoques se pueden clasificar en tres categorías principales: manual, semiautomática y totalmente automatizada. En la refactorización manual, el desarrollador refactoriza sin ninguna herramienta de apoyo, en semiautomática nos referimos a la situación en la que un desarrollador utiliza las herramientas estándar de refactorización disponibles en los entornos de desarrollo integrados, mientras que en la refactorización totalmente automatizada, se emplea un proceso basado en búsqueda para encontrar una secuencia completa de refactorización que mejore el programa de acuerdo con la función de aptitud empleada.

El artículo presenta una solución que da la opción al desarrollador de aplicar operaciones según lo necesite, permitiéndole aceptar, rechazar o modificar cada refactorización, esta solución se basa en el interés directo para el desarrollador, en lugar de los olores de código o las métricas de calidad del software. Se utiliza la optimización multiobjetivo basada en NSGA-II para encontrar la mejor opción entre las mejoras en la calidad del código y la reducción del número de cambios en el código. Una solución consiste en una secuencia de n operaciones de refactorización que involucran uno o varios elementos del código fuente del sistema a refactorizar. Se utiliza una representación basada en vectores para definir la secuencia de refactorización. Para cada refactorización, se especifican las precondiciones y las postcondiciones para asegurar la viabilidad de la operación.

Como conclusión los resultados de la experimentación confirman que ayuda a los participantes a refactorizar sus sistemas de manera más eficiente, encontrando soluciones de refactorización más relevantes y mejorando la calidad de los diez sistemas evaluados.

On the Alternatives for Composing Batch Refactoring [47]

Este artículo tiene como objetivo reflexionar sobre diversas alternativas para apoyar a los desarrolladores en la composición de lotes, permitiendo la eliminación de code smells. Actualmente, el soporte para la refactorización por lotes es limitado; por ejemplo, JDeodorant

solo recomienda transformaciones de código aisladas.

Se busca anticipar la detección y corrección de code smells con el apoyo de una herramienta semiautomatizada. La primera alternativa propone una herramienta que recomiende transformaciones que compongan un lote. En algunas tareas de revisión de código, como detectar y corregir ciertos code smells, los desarrolladores son más eficientes cuando trabajan de forma colaborativa, algo que no es común en las herramientas de refactorización actuales.

La herramienta estaría integrada en el entorno de desarrollo y recomendaría transformaciones de manera continua, permitiendo detectar y corregir code smells en tiempo real. La segunda alternativa aborda el desafío de las opiniones entre los revisores, permitiendo que cada uno componga su propio lote de transformaciones para eliminar code smells. La tercera alternativa propone la colaboración entre desarrolladores y revisores para componer el lote de refactorización, empleando herramientas semiautomatizadas.

En conclusión, se presentan tres alternativas que podrían mejorar el proceso de composición de lotes de refactorización. Sin embargo, no se desarrolló ninguna herramienta ni se obtuvieron métricas concretas.

Toward a Software Bad Smell Prioritization Model for Software Maintainability [48]

A medida que el software evoluciona, su estructura cambia y la complejidad del software aumenta. Varios estudios han investigado diferentes métodos para detectar y corregir los bad smells. Sin embargo, refactorizar no necesariamente mejora la calidad del código; algunos atributos de calidad pueden mejorar mientras que otros pueden empeorar. Por lo tanto, conocer la mejor secuencia permitirá a los diseñadores conocer el orden de la refactorización para mejorar un atributo de calidad específico, lo que, a su vez, mejorará la calidad general.

El objetivo de este artículo es proponer un modelo de priorización de bad smells de software para la mantenibilidad del software. El modelo de priorización puede ayudar a los profesionales del software a priorizar su esfuerzo en el impacto de los bad smells dentro de la mantenibilidad y puede extenderse para priorizar el impacto de los malos olores en atributos de calidad, la prueba de la refactorización se realizó de manera semiautomática, pero se verificó de manera manual. Las propiedades que se utilizaron para este experimento fueron volumen, duplicación, Tamaño de la unidad, Complejidad de la unidad, Interfaz de la unidad y Acoplamiento

Como conclusión en este artículo se plantea la necesidad de explorar y priorizar el impacto de los bad smells en la calidad del software, no se planteó una herramienta, se prioriza la mantenibilidad, pero dan la posibilidad de extender a otros atributos de calidad.

HSHEP: An Optimization-Based Code Smell Refactoring Sequencing Technique [43]

Las técnicas de secuenciación de refactorización mitigan problemas mejorando la estructura interna de un sistema. Es crucial priorizar los code smells en función de criterios como el riesgo y la importancia. Este artículo introduce un enfoque híbrido utilizando el algoritmo de optimización HSHEP (Hybrid Spotted Hyena and Emperor Penguin), que optimiza la secuencia de errores al incorporar las opiniones y requisitos de los programadores.

El enfoque inicial aplica un algoritmo de optimización novedoso, el HSHEP, que combina las fortalezas de dos enfoques: Spotted Hyena Optimizer (SHO) y Emperor Penguin

Optimizer (EPO). El híbrido HSHEP sirve como base para identificar las secuencias de refactorización efectivas para eliminar los code smells de diseño identificados. Las tareas de refactorización se realizan en base a estas secuencias.

Para mejorar la participación del usuario, el método ofrece tres alternativas para tratar los efectos secundarios, permitiendo a los ingenieros de software tomar decisiones informadas mientras mantienen el control sobre el proceso de refactorización. Los pesos en la función de priorización se pueden ajustar según las preferencias del desarrollador. En términos de mantenibilidad, el enfoque se centra en cinco características clave: modularidad, legibilidad, estabilidad, extensibilidad y reusabilidad.

En conclusión, aunque no se trata de una herramienta en sí, el enfoque se centra en los code smells y emplea la técnica HSHEP para elegir la mejor solución optimizada para la secuenciación de refactorización, mejorando la calidad del software de manera significativa.

Towards an Autonomous Bot for Automatic Source Code Refactoring [36]

Este artículo presenta un bot autónomo diseñado para integrarse al equipo de desarrollo a través de la plataforma de control de versiones existente, específicamente para proyectos en Java. El Refactoring-Bot realiza automáticamente refactorizaciones con el fin de resolver code smells, y permite a los desarrolladores aprobar o rechazar los cambios propuestos.

Una de las características clave de esta herramienta es que no requiere esfuerzo de configuración por parte de cada miembro del equipo, ni implica que los desarrolladores tengan que aprender a usar una nueva herramienta de refactorización. Además, Refactoring-Bot permite aplicar refactorizaciones de manera individual o combinada para eliminar varios tipos de code smells. Entre los code smells que el bot puede corregir se incluyen: el orden incorrecto de los modificadores del lenguaje, la falta de la anotación Override y parámetros de métodos no utilizados.

En particular, el bot se enfoca en la refactorización de long methods mediante la técnica Extract Method, para mejorar la legibilidad y mantenibilidad del código.

En conclusión, Refactoring-Bot facilita la aprobación o rechazo de los cambios propuestos por el bot sin interrumpir el flujo de trabajo del equipo. Sin embargo, es importante destacar que la herramienta aún carece de métricas que sigan una secuencia de refactorización.

Industry's cry for tools that support large-scale refactoring [14]

Este artículo destaca la importancia de disponer de herramientas de refactorización a gran escala. Los desarrolladores necesitan herramientas especializadas y no de propósito general, como los IDEs, especialmente para tareas como la partición de monolitos heredados en unidades pequeñas que sean desplegables, escalables y evolucionen de manera independiente. Las herramientas de refactorización a pequeña escala suelen asumir que el objetivo es eliminar *code smells* específicos o mejorar métricas de calidad del código.

En conclusión, este estudio enfatiza la necesidad de crear herramientas que permitan la refactorización a gran escala, en este se mencionan las refactorizaciones a pequeña escala (floss). Las herramientas son esenciales para abordar los desafíos de la evolución y mantenimiento del software en contextos complejos.

Does migrating a monolithic system to microservices decrease the technical debt? [49]

Este artículo analiza si la deuda técnica se reduce al migrar de sistemas monolíticos a microservicios, utilizando la herramienta SonarQube. El estudio se centra en tres tipos de deuda técnica: tiempo para eliminar problemas que pueden generar fallos, tiempo para eliminar problemas que generan mayor mantenimiento, y tiempo que compromete la seguridad.

SonarQube es una herramienta utilizada para medir la deuda técnica y calcular diferentes métricas, dependiendo del lenguaje de programación, basándose en cinco reglas: robustez, eficiencia de rendimiento, seguridad, transferibilidad y adaptabilidad. Se realizó un estudio en el que, al inicio de la migración, la deuda técnica aumentó, pero luego de estabilizarse, disminuyó debido a la integración y refactorización para mantener un código más limpio. Permanecer en un sistema monolítico impedía actualizaciones y aumentaba los costos, aunque a largo plazo, los beneficios de migrar a microservicios fueron significativos ya que la deuda técnica crecía más rápidamente en un sistema monolítico.

El estudio, realizado en una empresa italiana, mostró que la deuda técnica creció un 90% más lentamente después de implementar microservicios. Las fases del proceso incluyeron la identificación de opciones de descomposición, la implementación del primer microservicio y los microservicios faltantes basándose en el primero.

cASpER: A Plug-in for Automated Code Smell Detection and Refactoring [50]

La refactorización es la práctica en la que los desarrolladores eliminan Code Smells² sin cambiar el comportamiento externo del código fuente. Realizarlo manualmente requiere mucho tiempo y esfuerzo, por lo que, normalmente los desarrolladores prefieren mantener un código de baja calidad en lugar de dedicar ese tiempo para realizar operaciones de refactorización. En este artículo se presenta cASpER, un plugin en IntelliJ IDEA que brinda soporte visual y semiautomático para la detección y refactorización de cuatro tipos diferentes de code smells los cuales son: Feature Envy, Misplaced Class, Blob y Promiscuous Package.

La herramienta ofrece dos tipos de estrategias para la detección de code smells: una basada en el cálculo de métricas (DECOR) y otra basada en texto (TACO). En conclusión, el artículo propone una herramienta que permite la detección y refactorización de los cuatro tipos de code smells mencionados, pero no se enfoca en realizar u obtener la mejor sucesión de refactorizaciones consecutivas.

An empirical study to determine the optimum refactoring sequence of critically affected classes using Software Quality Attributes [51]

El presente estudio contribuye a validar el impacto de la refactorización en los atributos internos y externos del software, utilizando cuatro conjuntos de datos de código abierto. Además, se introduce un nuevo enfoque denominado Puntuación de Atributos de Calidad (QAS, por sus siglas en inglés), el cual se utiliza para determinar la secuencia óptima de refactorización en relación con los atributos de calidad internos y externos de una clase. Este puntaje se calcula después de observar el efecto de la refactorización en los atributos externos

del software.

Primero, se observa el efecto de la refactorización en los atributos internos del software. Para ello, se comparan las métricas del software antes y después de aplicar las técnicas de refactorización. Luego de calcular el QAS para cada técnica de refactorización, se determina la secuencia de aplicación utilizando el orden ascendente del QAS para cada clase crítica respectiva.

En el paradigma de programación orientada a objetos, uno de los métodos en los que los cambios en el software afectan únicamente a los atributos de calidad internos, sin alterar su estructura externa, se conoce como refactorización. Aunque algunos Entornos de Desarrollo Integrado (IDE) modernos, como NetBeans y Eclipse, ofrecen soporte semiautomático para la aplicación de técnicas de refactorización, aún se considera que esta es un área de investigación emergente, por lo que persiste la necesidad de fortalecer esta área de investigación. Es necesario identificar qué técnicas de refactorización deben aplicarse, dónde y cuándo, con el objetivo de obtener un código limpio.

Enhancing Code Quality through Automated Refactoring Techniques [52]

En esta tesis de maestría se centra en desarrollar e implementar una herramienta de refactorización automatizada dentro del entorno de desarrollo integrado IntelliJ IDEA.

Uno de los principales beneficios del código limpio es su impacto positivo en los usuarios finales. Las principales técnicas de refactorización se pueden dividir en 6 categorías, las cuales son Métodos de composición, Moviendo características entre objetos, Organizando datos, Simplificando expresiones condicionales, Simplificando las llamadas a método y Tratando con la generalización.

En nuestra tesis, se aplicaron reglas de secuenciación para estructurar el proceso de refactorización. Por otro lado, la tesis "Enhancing Code Quality through Automated Refactoring Techniques" utilizó reglas sin secuenciación, sin dar énfasis a estas. Además, no incluyó un segundo enfoque ni consideró criterios ponderados.

Un aspecto destacado de dicha tesis fue su enfoque en el desarrollo basado en metodologías ágiles. También identificó problemas en el código mediante el uso de métricas como complejidad ciclomática, acoplamiento y líneas de código. Estas métricas fueron evaluadas a través de umbrales estáticos, que consisten en límites fijos para determinar la calidad del código.

On the Impact of Refactorings on Software Attack Surface [53]

En el artículo analizado, se destaca la importancia de que los desarrolladores consideren cuidadosamente la combinación y secuencia de operaciones de refactorización para mejorar la calidad del software sin comprometer su seguridad. El estudio incluyó análisis empíricos realizados en 30 proyectos de código abierto desarrollados en lenguajes como Python, C y JavaScript. Los resultados evidenciaron que ciertas refactorizaciones, ejecutadas con una conciencia de seguridad, pueden reducir vulnerabilidades y mejorar significativamente la seguridad general del sistema. Para este propósito, se utilizó SonarQube, una herramienta destinada a garantizar la calidad del código mediante la detección y análisis de elementos que forman parte de la superficie de ataque.

Entre los métodos de refactorización estudiados se encuentran: move method, extract

method, rename method, rename class, inline method, add parameter, remove parameter, y pull up method. De estos, el método move method resultó ser uno de los más utilizados por los desarrolladores, debido a su facilidad de ejecución, ya sea de forma manual o mediante las herramientas integradas en los entornos de desarrollo (IDEs).

En el análisis de los proyectos, SonarQube identificó un total de 10 tipos distintos de code smells. Aunque el artículo no profundiza en las métricas de calidad, se centra en ciertas secuencias de métodos de refactorización y su impacto en la seguridad del sistema. No obstante, no se desarrolló ninguna herramienta específica con este objetivo.

Empirical Evaluation of the Impact of Refactoring on Internal Quality Attributes [54]

Esta investigación valida el efecto de la refactorización en los atributos de calidad interna del software, especialmente en el acoplamiento. Aunque la refactorización ha sido promovida como una solución para mejorar la calidad del software, existen pocos estudios empíricos que demuestren sus beneficios en métricas como cohesión, acoplamiento, capacidad de mantenimiento y entropía de cambio.

En la detección de code smells, se destacan dos enfoques principales: Heurístico, basado en reglas y valores umbral, Aprendizaje automático, que emplea conjuntos de datos etiquetados por expertos y modelos supervisados para identificar code smells.

Los atributos internos, como acoplamiento y tamaño, pueden medirse directamente en la fase de codificación usando el código fuente, a diferencia de los atributos externos que requieren interacción con su entorno de ejecución.

En conclusión, se evaluó el impacto de la refactorización en acoplamiento y tamaño mediante IntelliJ JetBrains. La técnica de extracción de métodos demostró reducir el acoplamiento al dividir métodos grandes en otros más pequeños y específicos. Aunque no se implementaron herramientas ni secuencias específicas de refactorización, los resultados evidencian mejoras significativas en estos atributos de calidad interna.

A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework [55]

Este estudio propone un nuevo entorno basado en Modelos de Lenguaje de Agentes Múltiples (LLM) para el diseño y refactorización automatizados de software, con el objetivo de mejorar la eficiencia del desarrollo, reducir la deuda técnica y optimizar la calidad del software. El enfoque busca establecer un marco de reestructuración integral que considere múltiples atributos de calidad de manera simultánea.

A diferencia de las herramientas tradicionales de refactorización, que ofrecen sugerencias aisladas para code smells, este estudio pretende integrar consideraciones más amplias, como seguridad y rendimiento. Para evaluar el enfoque, se utilizaron herramientas tradicionales como analizadores estáticos (SonarQube y ESLint) y métricas como complejidad ciclomática, líneas de código (LOC) y cantidad de code smells detectados.

El estudio evidencia que los LLM, aunque prometedores, presentan desafíos debido a la novedad del enfoque, como la generación de sugerencias semánticamente defectuosas. Aunque no se aborda la secuenciación de refactorizaciones, si se menciona sobre la refactorización los LLM en los atributos de calidad del código.

AI-Driven Code Optimization: Leveraging ML to Refactor Legacy Codebases [56]

Este artículo se enfoca en cómo los modelos de aprendizaje automático pueden analizar estructuras de código, detectar ineficiencias y generar versiones refactorizadas optimizadas sin alterar la funcionalidad original. A medida que las aplicaciones de software evolucionan, los sistemas heredados suelen enfrentar problemas relacionados con estructuras de código obsoletas y acumulación de deuda técnica. Estas aplicaciones, generalmente extensas y complejas, han sido desarrolladas por múltiples programadores a lo largo del tiempo, muchas veces sin una documentación adecuada, lo que dificulta la comprensión de la lógica y las dependencias del sistema.

Las herramientas basadas en inteligencia artificial tienen la capacidad de analizar el comportamiento en tiempo de ejecución, identificar patrones ineficientes y sugerir refactorizaciones conforme a las mejores prácticas. Entre los code smells que pueden detectar se encuentran, por ejemplo, código duplicado y métodos excesivamente largos. No obstante, aún existen desafíos importantes, como la confianza en el código generado por IA y los riesgos asociados a la seguridad. Estas herramientas suelen tener dificultades para interpretar la intención subyacente de un fragmento de código determinado.

Además, muchos desarrolladores prefieren conservar el control manual sobre el proceso de refactorización, lo que genera cierta resistencia a adoptar soluciones automatizadas basadas en IA. Cabe señalar que en el artículo no se menciona la implementación de alguna herramienta específica ni se hace uso de secuencias de refactorización.

Con el propósito de identificar las características, fortalezas y limitaciones de las propuestas existentes en el área de refactorización de software, se realizó un análisis comparativo de los principales trabajos relacionados. Para ello, se consideraron aspectos como el producto desarrollado, el tipo de proceso de refactorización, las métricas empleadas para evaluar la calidad del software, el soporte para la secuenciación de las refactorizaciones y el alcance de cada propuesta. La Tabla 5 se presenta la comparación de los trabajos analizados con base en estos criterios.

Tabla 5. Tabla comparativa de trabajos relacionados.

Trabajo de investigación	Producto resultante	Tipo de proceso	Métricas usadas	Secuenciación	Alcance
Propone un enfoque para que las herramientas sean más flexibles y permitan la personalización					
On the Customization of Batch Refactoring [7]	Análisis	Manual	No Usa Métricas.	---	Caso de estudio
En este artículo se realiza un Bot, el desarrollador puede interactuar con los resultados de refactorización.					
RefBot: Intelligent Software Refactoring Bot [10]	Herramienta	Semiautomática	Métricas basadas en QMOOD	Cuenta con secuenciación, permite control total	Implementado

Trabajo de investigación	Producto resultante	Tipo de proceso	Métricas usadas	Secuenciación	Alcance
Los resultados que se obtuvieron dieron como resultado que los desarrolladores realizaban modificaciones					
The untold story of code refactoring customizations in practice [13]	Análisis	---	---	---	Caso de estudio
En este artículo se propone un marco que permita identificar problemas de diseño.					
A framework to improve quality of a Java system by performing refactoring [32]	Herramienta	semiautomático	Complejidad cognitiva, TurboMQ	---	Implementado
Este artículo realiza un estudio cuantitativo sobre el efecto de la refactorización en los atributos de calidad interna.					
Refactoring effect on internal quality attributes: What haven't they told you yet? [41]	Análisis	Manual	Cohesión, Complejidad, Acoplamiento, Herencia y Tamaño.	---	Caso de Estudio
En este artículo se presenta una herramienta visual para la refactorización.					
Towards a Live Environment for Code Refactoring [35]	Herramienta	Semiautomática	Número de líneas de código, complejidad ciclomática, métricas de Halstead, falta de cohesión	---	Implementado
En este artículo se plantea un modelo que permitirá a los ingenieros de software la capacidad para mejorar los aspectos de calidad interna.					
A refactoring categorization model for software quality improvement [19]	Análisis	Manual	QMOOD	---	Caso de estudio
Se desarrolló una herramienta llamada DART					
Reducing redundancy of test cases generation using code smell detection and refactoring [42]	Herramienta	Semiautomática	Métrica de distancia, propiedad de cohesión.	---	Implementado

Trabajo de investigación	Producto resultante	Tipo de proceso	Métricas usadas	Secuenciación	Alcance
Enfoque que emplea la técnica HSHEP para elegir la mejor solución optimizada para la secuenciación de refactorización.					
HSHEP: An Optimization-Based Code Smell Refactoring Sequencing Technique [43]	Análisis	Manual	CBO LCOM RFC Maintainability Index	Cuenta con secuenciación está fijado por el algoritmo.	Caso de estudio
Este artículo presenta un bot autónomo, específicamente para proyectos en Java.					
Towards an Autonomous Bot for Automatic Source Code Refactoring [36]	Herramienta	Automática	-----	---	Implementado
Este artículo presenta el impacto de la refactorización en los atributos internos y externos del software utilizando cuatro conjuntos de datos.					
An empirical study to determine the optimum refactoring sequence of critically affected classes using Software Quality Attributes [51]	Análisis	Manual	CBO, DIT, LCOM, NOC, RFC, WMC, OCavg, Dcy&Dcy*, jf, jLOC, Jm, LOC, AHF, MHF, PF, AIF	Se aplica secuenciación sin flexibilidad por parte del usuario para modificar estas secuencias.	Caso de estudio
El objetivo de esta investigación es realizar un estudio comparativo en el que se evaluaron cuatro herramientas de refactorización.					
A Comparative Study of Automated Refactoring Tools [37]	Análisis	Manual	-----	---	Caso de Estudio
El artículo presenta una solución que permite al desarrollador aplicar operaciones según lo necesiten					

Interactive and Dynamic Multi-Objective Software Refactoring Recommendations [46]	Análisis	Manual	Tamaño del diseño, Jerarquías, Abstracción, Encapsulación, Acoplamiento, Cohesión, Composición, Herencia, Polimorfismo, Mensajería	Cuenta con secuenciación	Caso de estudio
--	----------	--------	--	--------------------------	-----------------

Trabajo de investigación	Producto resultante	Tipo de proceso	Métricas usadas	Secuenciación	Alcance
Este artículo tiene como propósito reflexionar sobre múltiples alternativas para apoyar a los desarrolladores en la composición de sus lotes.					
On the Alternatives for Composing Batch Refactoring [47]	Análisis	Manual	-----	No cuenta con secuenciación	Caso de estudio
El objetivo de este artículo es proponer un modelo de priorización de bad smells de software para la mantenibilidad.					
Toward a Software Bad Smell Prioritization Model for Software Maintainability [48]	Análisis	Manual	-----	---	Caso de estudio
Este artículo se centra en estudios empíricos realizados en proyectos de código abierto con un enfoque principal en el impacto de la refactorización en la seguridad del software.					
On the Impact of Refactorings on Software Attack Surface [53]	Análisis	Manual	-----	Mención de importancia	Caso de estudio

Trabajo de investigación	Producto resultante	Tipo de proceso	Métricas usadas	Secuenciación	Alcance
Este artículo se centra en el efecto de la refactorización en los atributos de calidad interna del software, especialmente en el acoplamiento.					
Empirical Evaluation of the Impact of Refactoring on Internal Quality Attributes [54]	Análisis	Manual	Métricas acoplamiento: CINT, CDISP	No tiene secuenciación	Caso de estudio
Este artículo se centra en establecer un marco de reestructuración integral que considere múltiples atributos de calidad de manera simultánea.					
A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework [55]	Herramienta	Automática	LOC, complejidad ciclomática	No tiene secuenciación	Implementado
Desarrollar un entorno visual e interactivo para la refactorización					
Ambiente visual de orquestación de microservicios refactorización	Herramienta	Automática	PMP, PMPR, PMF, PM, TPM, FHI, FHIJ, FHIAC, FFC, FMFAC, PF y AF.	Cuenta con reglas de secuenciación	Implementando

2.6.2. Antecedentes

En este apartado se presentan trabajos de investigación relacionados con la refactorización de software y el uso de arquitecturas basadas en microservicios. Los antecedentes seleccionados permiten identificar los avances realizados en la automatización de procesos de refactorización, la migración hacia arquitecturas de microservicios y la secuenciación de servicios de refactorización. Su revisión es relevante para esta investigación, se establece las bases sobre las cuales se desarrolla la propuesta y reconocer los aspectos que aún requieren atención.

Método para migrar servicios web bajo el protocolo SOAP hacia microservicios basados en REST [57]

En este trabajo de tesis se realizó un método automatizado para obtener marcos de microservicios basados en protocolos de comunicación REST, a partir de sistemas de software monolíticos de dominios de aplicaciones, escritas en lenguaje Java. Al refactorizar hacia microservicios estos cuentan con atributos de calidad deseables, tales como: coherencia, cohesión, acoplamiento y tiempo de respuesta: Para la migración de arquitecturas

monolíticas hacia arquitecturas de microservicios se utilizó la herramienta SR2 Transforming realizando una extensión a ésta. La mejora en modularidad de los microservicios permite una mayor reutilización de los objetos. El método que se desarrolló fue definir los límites del servicio, refactorizar el código de la aplicación original y refactorizar los datos.

Coreografía de microservicios de refactorización de software legado

En este proyecto, se desarrolló y desplegó un SaaS (Software como Servicio) en la nube, que los microservicios aplican métodos de refactorización específicos. Estos métodos están diseñados para reducir o eliminar deuda técnica producida por código mal formado (code smell) como la herencia de implementación, la carencia de protección modular y la falta de abstracción.

En este proyecto se determinó una secuenciación de servicios en orden, basados en reglas de precondiciones y postcondiciones de cada microservicio de manera que no se contrapongan entre sí. Este SaaS evita escenarios de ejecución de servicios que se contradicen uno a otro, como cuando el método de falta de abstracción se aplica antes que los otros mencionados, al aplicarse los posteriores, estos disminuyen la parte de la mejora obtenida por el primero. La secuencia correcta para aplicar estos métodos fué MR3-MR2-MR1, ya que el método de falta de abstracción agrega características que consideran las métricas de protección modular y herencia de implementación, pero no modifica las características presentes desde el inicio en la aplicación.

Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente [8]

En este caso se propuso un método de refactorización de calificadores de alcance y cinco métricas que permitan evaluar el grado de protección modular para los niveles de visibilidad: PMFP (Protección Modular de Funciones Privadas), PMFPR (Protección Modular de Funciones Protegidas), PMFF (Protección Modular de Funciones Friendly), PM (Protección Modular) y TPM (Total Protección Modular).

Lo que ocurría es que una gran mayoría de los casos de prueba, sino es que todas, de las funciones se califican en su visibilidad como *public*, no se tomaban en cuenta las reglas de visibilidad de los lenguajes de programación que corresponden a una mayor protección de datos. Ante este escenario se propuso un método de refactorización en el que los métodos son calificados correctamente para lograr una verdadera protección de datos. Se busca que se mejore tanto la modularidad como el encapsulamiento de objetos, lo cual consistió en clasificarlas dependiendo de cómo son invocadas los métodos de clases. Se realizó la integración de la herramienta SR2-Refactoring como método para poder realizar el proceso de refactorización.

Reducción de la fragilidad del software legado originado por el acceso a datos compartidos [58]

El principal objetivo de este trabajo fue reducir la deuda técnica. Se realizó una extensión de la herramienta “SR2-Refactoring”, se utilizaron las métricas propuestas en [8].

En el desarrollo se considera que el uso de modificadores de acceso, así como el ocultamiento de información permite contar con un buen nivel de protección, al tener declarada la información como privada sólo se puede acceder a los detalles internos desde

las entidades locales a la misma entidad o bloque. Al proteger los atributos se evitan modificaciones a los detalles internos, tales como el valor de los datos o atributos. En este desarrollo se extiende la medición de protección modular, evaluando la protección de los atributos y de esta manera se obtiene un mejor encapsulamiento modular de sistemas legados.

Capítulo 3. Metodología

En este capítulo se describe el procedimiento seguido para desarrollar la investigación. Se presentan las actividades realizadas para analizar el efecto del orden de ejecución de las refactorizaciones, establecer la secuencia correspondiente a criterios ponderados de sucesión. La metodología permite alcanzar los objetivos planteados en la investigación.

3.1. Análisis de deuda técnica derivada del orden secuencial de refactorizaciones

Como se describió en la sección de antecedentes, en el laboratorio de Ingeniería de Software del TecNM/CENIDET se han desarrollado de forma independiente tres métodos de refactorización: A) 'Método de refactorización de calificadores de alcance' [8] , B) 'Método de Refactorización de Carencia de Abstracción', y C) 'Reducción de Herencia de Implementación' [9]. La refactorización en secuencia de estos tres métodos puede verse afectada según el orden de ejecución en una secuencia.

Considerando que los tres métodos poseen la misma ponderación, es posible establecer seis secuencias de ejecución en pares ordenados: $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$, $C \rightarrow A$, $B \rightarrow C$ y $C \rightarrow B$.

A. *Método de refactorización de calificadores de alcance*

En el desarrollo de software legado, como se muestra en la Figura 2, es común que los métodos de las clases se declaren como “*public*”, lo que dificulta aplicar correctamente el encapsulamiento y expone datos sensibles, afectando la seguridad del sistema. Mejorar esta práctica fortalece la modularidad y facilita el mantenimiento, al organizar los métodos según cómo son utilizados por entidades externas, encapsulando el comportamiento de un objeto [8].

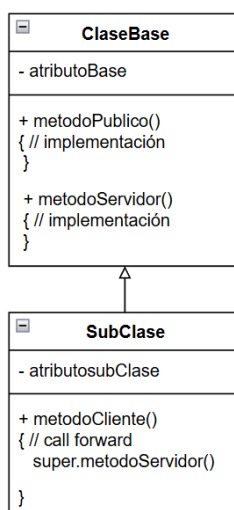


Figura 2. Arquitectura original

El “Método de refactorización de calificador de alcance”, analiza cada método de clases de una aplicación Java para asignarle el calificador de acceso adecuado: “*public*” para métodos constructores o usados por clases externas; “*protected*” para los empleados solo por métodos de subclases; “*friendly*” para los accesibles dentro del mismo paquete, y “*private*” para los de acceso interno de la propia clase. En la Figura 3 se muestra como el resultado de esta estrategia aplicada en Figura 2 contribuye a sistemas de mejor encapsulamiento, seguros y fáciles de mantener [8].

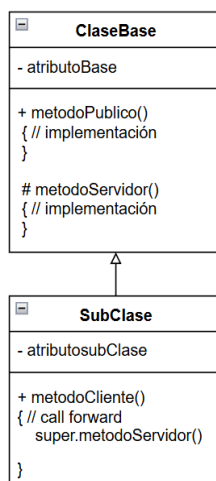


Figura 3. Refactorización A

B. Método de Refactorización de Carencia de Abstracción

La falta de abstracción en software legado orientado a objetos obliga al cliente a interactuar con clases específicas, generando rigidez y dificultad para adaptarse a cambios, como se muestra en la Figura 2 que corresponde a la arquitectura original. La abstracción, mediante clases abstractas e interfaces, mejora la reusabilidad, flexibilidad y polimorfismo, permitiendo modificar comportamientos existentes y agregar nuevas funcionalidades.

El “Método de Refactorización de Carencia de Abstracción” crea arquitecturas adaptables añadiendo abstracción, a partir de la identificación de clases de la cima de la jerarquía que carezcan de abstracción. En caso de haberse creado una clase abstracta, los métodos públicos los hace abstractos y la implementación de estos métodos se mueven a sus correspondientes en las subclases. En caso de haberse creado una interfaz, la totalidad de los métodos son calificados como “*abstract public*” y mueve la implementación a las subclases de manera apropiada, Figura 4.

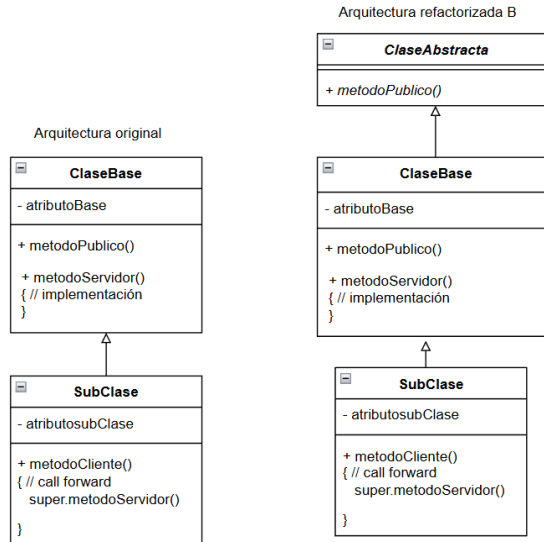


Figura 4. Refactorización B

C. Reducción de Dependencia en la Herencia de Implementación

Este factor se refiere a la presencia de código de implementación ubicado en una clase base dentro de una jerarquía de herencia. Las subclases acceden a dicha implementación mediante invocaciones explícitas, tales como el uso de la sentencia “*super*” en Java o del calificador de alcance “*::*” en C++ o equivalentes en otros lenguajes de programación como se observa en la Figura 5. Estas invocaciones son conocidas como “*call forward*”. El uso inadecuado de la herencia de implementación reduce la flexibilidad ante cambios funcionales, provoca acoplamiento entre clases y la dependencia entre clases limita su movilidad, además de violar los principios de inversión de dependencias y abierto/cerrado. Como consecuencia, las modificaciones en la clase padre pueden propagarse y afectar negativamente a múltiples subclases.

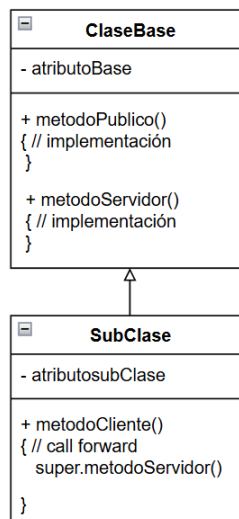


Figura 5. Arquitectura original

Esta refactorización invierte estas llamadas, convirtiéndolas en “*callbacks*” del servidor hacia el cliente, usando el patrón “*Template Method*” [9].

Para ello, primero la clase base se declara “*abstract*” o interfaz de manera apropiada, se cambia el calificador del método servidor a “*abstract protected*” y su implementación se traslada a un nuevo método servidor en la subclase con calificador “*protected*”, para asegurar que este método solo sea accedido desde la clase base. Luego, se crea un método plantilla con calificador “*public*” en la clase base que delega la ejecución del método servidor al correspondiente en la subClase mediante un “*callback*” y se elimina de la subclase el método cliente original. Los métodos invariantes, llamados solo por el método plantilla, se declaran “*private*” [9], Figura 6.

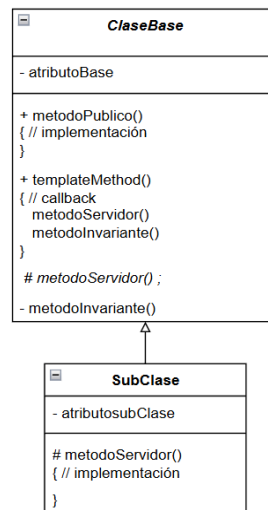


Figura 6. Refactorización C

Secuencia de pasos ordenados $A \rightarrow C$ contra $C \rightarrow A$:

En la secuencia de refactorización $A \rightarrow C$, la refactorización A dejó los métodos accesibles desde el exterior con el calificador de alcance “*public*”. Esto ocurre con el método cliente de la subclase dentro de una jerarquía de herencia. Sin embargo, el método servidor es invocado desde la subclase en la jerarquía de herencia, por lo tanto, este método es calificado como “*protected*” por el método A, como se muestra en la Figura 7.

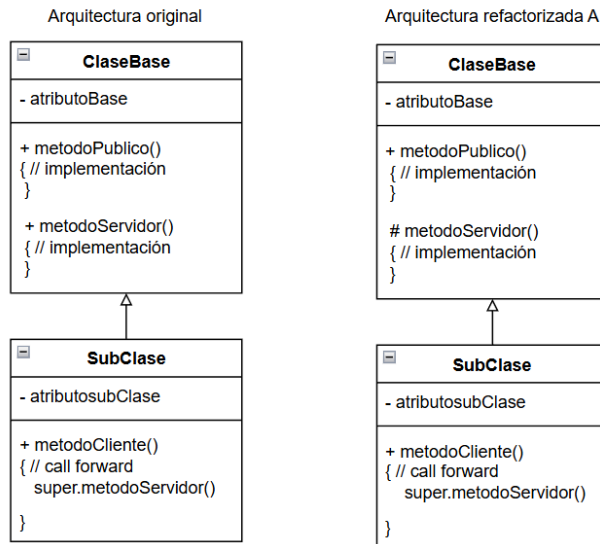


Figura 7. Refactorización A

Sin embargo, al ejecutar posteriormente el método C, el método A es eliminado o se le cambia su calificador a *protected*, desde este punto de vista ejecutar la secuencia $A \rightarrow C$ o $C \rightarrow A$ no tiene impacto, como se observa en la Figura 8. Sin embargo, si ejecutamos la secuencia $A \rightarrow C$, aún no existe la función “*template method*” en la clase base, por lo tanto, su calificador de alcance no podría ser verificado por el método A.

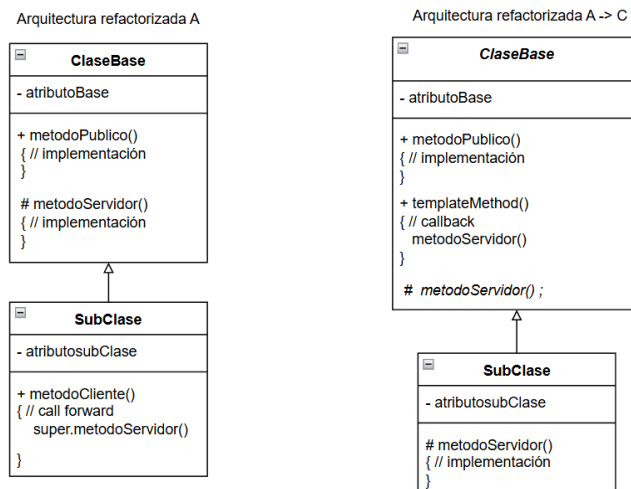


Figura 8. Refactorización $A \rightarrow C$

Conclusión: Es mejor ejecutar la secuencia ordenada C) “Reducción de Herencia de Implementación” $\rightarrow$ A) “Refactorización de calificadores de alcance”.

Secuencia de pasos ordenados $B \rightarrow C$ contra $C \rightarrow B$:

En una arquitectura sin abstracción, dentro de la secuencia de refactorización $B \rightarrow C$, la refactorización B crea una clase abstracta. De manera similar, la refactorización C también genera una clase abstracta que proporciona el patrón “*Template Method*”. Si estas

refactorizaciones se aplican en el orden $B \rightarrow C$, se puede generar una duplicación innecesaria, ya que, como se observa en la Figura 9, tanto *ClaseAbstracta* como *ClaseBase* produce que ambas clases se generen como clases abstractas. Además, la refactorización C obliga a declarar un método plantilla abstracto en la clase que genera, lo cual contraviene los principios de implementación del patrón “*Template Method*” en términos del incumplimiento de la definición de una superclase abstracta [59].

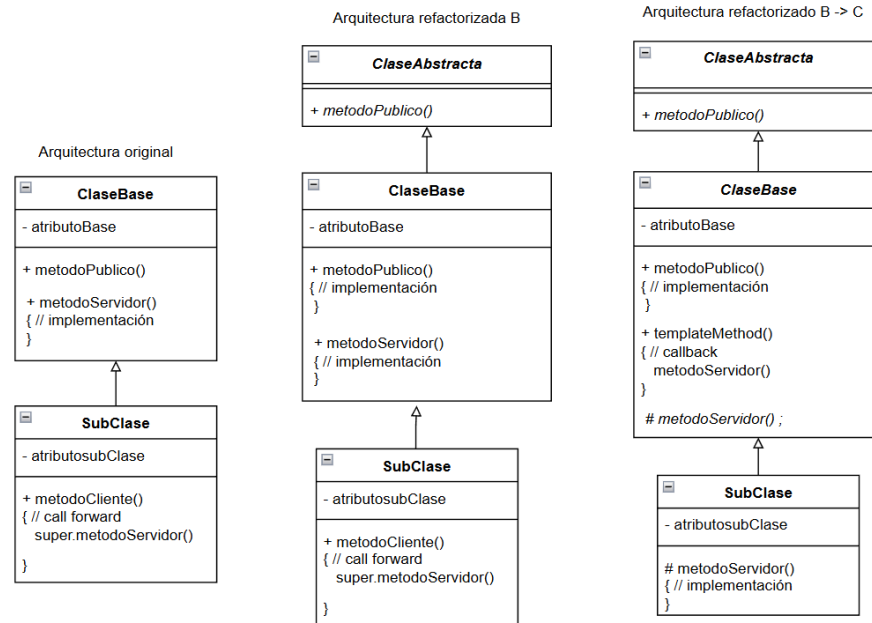


Figura 9. Refactorización $B \rightarrow C$

Conclusión: Carencia de abstracción no genera una clase abstracta si está ya existe, como en el ejemplo de la Figura 10. Por lo tanto, es recomendable aplicar la secuencia en el orden $C \rightarrow B$. Primero, se debe reducir la herencia de implementación y, posteriormente, abordar la carencia de abstracción. Esto evita la duplicación de elementos y asegura un diseño más limpio y eficiente.

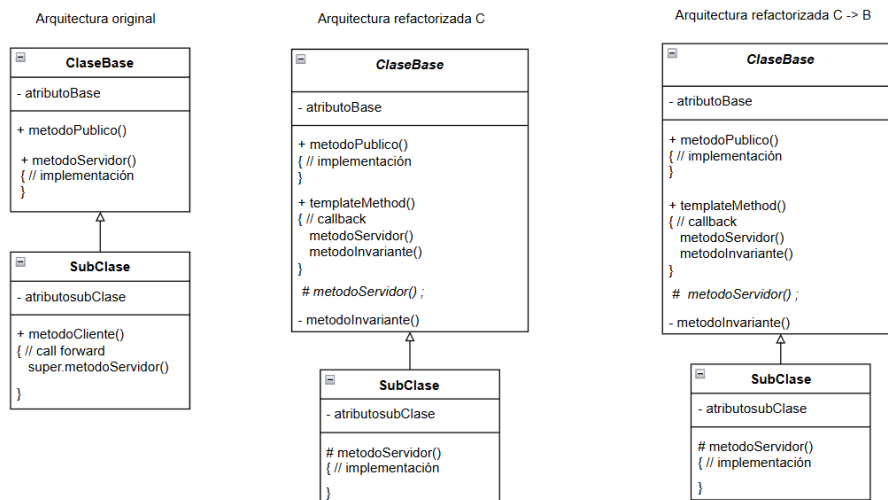


Figura 10. Refactorización $C \rightarrow B$

Secuencia de pasos ordenados $A \rightarrow B$ contra $B \rightarrow A$:

En el orden de ejecución secuencial $A \rightarrow B$, la refactorización A verifica los calificadores de alcance de la arquitectura original. Sin embargo, en esta etapa puede que la clase base abstracta aún no exista. En ese caso, la refactorización B se encarga de generarla. Por lo tanto, la refactorización A no podría verificar los calificadores de alcance de los métodos que residen en la clase abstracta creada posteriormente por la refactorización B, Figura 11.

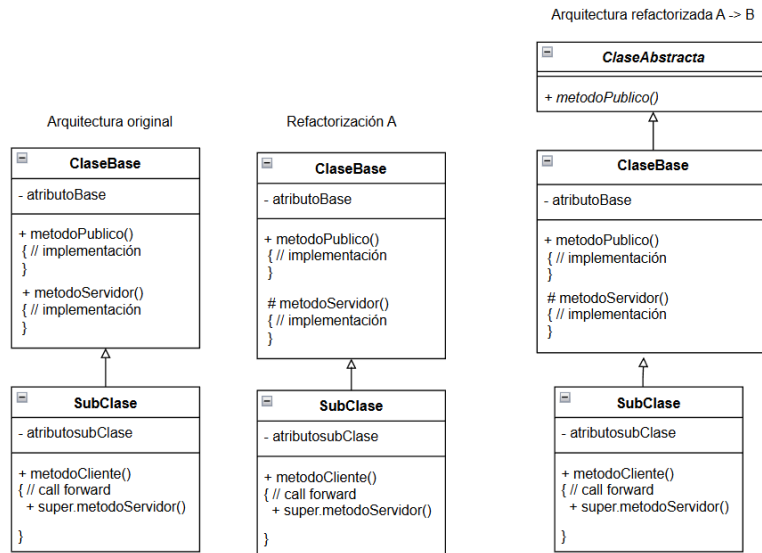


Figura 11. Refactorización $A \rightarrow B$

En el orden de ejecución $B \rightarrow A$, B genera una clase abstracta o interfaz con sus métodos correspondientes, los cuales podrían ser verificados por el método A, como se muestra en la Figura 12.

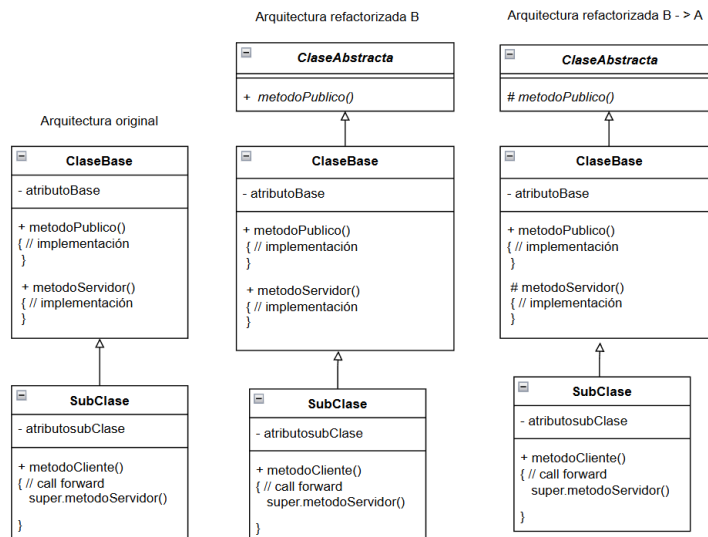


Figura 12. Refactorización $B \rightarrow A$

Conclusión: Es mejor ejecutar la secuencia ordenada B) “Refactorización de Carencia de Abstracción” → A) “Refactorización de calificadores de alcance”. De esta manera, el método A) verifica el calificador de alcance de la arquitectura que ya incluye abstracción.

Las secuencias de ejecución presentadas en la Tabla 6 fueron definidas exclusivamente para los escenarios de evaluación descritos previamente. Por lo tanto, estas secuencias no aplican a todos los casos de prueba desarrollados, sino únicamente a los escenarios seleccionados para ilustrar el proceso de ejecución y evaluación del sistema.

Tabla 6. Resumen de secuencias de refactorización.

Secuencias		
$A \rightarrow B$: La refactorización A no podría verificar los calificadores de alcance de los métodos que residen en la clase abstracta creada posteriormente por la refactorización B.	$A \rightarrow C$: Aún no existe la función “ <i>template method</i> ” en la clase base, por lo tanto, su calificador de alcance no podría ser verificado por el método A.	$B \rightarrow C$: Si estas refactorizaciones se aplican en el orden $B \rightarrow C$, se producirá una duplicación innecesaria de clases abstractas.
$B \rightarrow A$: B genera una clase abstracta o interfaz con sus métodos correspondientes, los cuales podrían ser verificados por el método A	$C \rightarrow A$: Existe el <i>template method</i> y por lo tanto puede ser verificado por A.	$C \rightarrow B$: Es recomendable aplicar la secuencia en el orden $C \rightarrow B$. Primero, se debe reducir la herencia de implementación y, posteriormente, abordar la carencia de abstracción.

3.2. Evaluación de deuda técnica

Entre estos factores se considera la aleatoriedad, utilizada para cuantificar el grado de incertidumbre presente en la ocurrencia de las diferentes alternativas de ejecución.

3.2.1. *Factor de aleatoriedad*

La aleatoriedad enfatiza en la idea de sorpresa e imprevisibilidad (incertidumbre) [60].

La aleatoriedad de ocurrencia de un evento se mide dependiendo de qué aspecto se requiere analizar. En términos formales, se aborda desde la probabilidad, la estadística y la teoría de la información. En sistemas de software (como eventos, fallas, notificaciones, microservicios), la entropía + Poisson es la combinación más común para medir aleatoriedad de ocurrencia.

Se tienen tres eventos distintos A , B , C y se observa el orden de ocurrencia.

El espacio muestral es:

$$\Omega = \{ABC, ACB, BAC, BCA, CAB, CBA\}$$

Número total de resultados posibles:

$$|\Omega| = 3! = 6$$

Fórmula de la Entropía de Shannon

La entropía se define como:

$$H(x) = - \sum_{i=1}^n p(x_i) \log_2 p(x_i) \quad (1)$$

$p(x_i)$ representa la probabilidad de ocurrencia del evento x . Si tienes 6 eventos posibles y todos son igualmente probables, entonces:

$$p(x_i) = \frac{1}{6}$$

Sustituimos:

$$H(x) = -6\left(\frac{1}{6} \log_2 \left(\frac{1}{6}\right)\right)$$

$$H(x) = -\log_2 \frac{1}{6}$$

$$\log_2 \frac{1}{6} = -\log_2 6$$

$$H(x) = -(-\log_2 6)$$

$$H(x) = \log_2 6$$

Si no se tiene logaritmo base 2 se puede calcular mediante el logaritmo base 10.

$$\log_2 6$$

$$\log_2 6 = \frac{\log 6}{\log 2}$$

$$\log 6 \approx 0.7781$$

$$\log 2 \approx 0.3010$$

$$\frac{0.7781}{0.3010} \approx 2.585 \quad (2)$$

Valor numérico de la entropía

$$\log_2 6 \approx 2.585 \text{ bits} \quad (3)$$

Dado un conjunto de tres eventos, existen seis permutaciones posibles. Bajo una distribución uniforme, la entropía alcanza su valor máximo de 2.585 bits como se puede ver en la ecuación **0.77810.3010** $\approx$ **2.585** (2, lo que indica un comportamiento completamente aleatorio en el orden de ocurrencia. Sin embargo, el evento debe ser únicamente: CBA, por lo tanto, el orden está completamente fijado. La variable aleatoria X toma un solo valor con probabilidad 1

Sustituimos:

$$H(x) = -6(1 \log_2 1) \quad (4)$$

Valor numérico de la entropía:

$$\log_2 1 \approx 0 \text{ bits} \quad (5)$$

Aunque existen seis permutaciones posibles para los eventos A, B y C, el sistema impone de manera determinista el orden CBA. En consecuencia, la distribución de probabilidad se

concentra en un único resultado, obteniéndose una entropía de Shannon igual a 0 bits como se puede ver en la ecuación $\log_2 1 \approx 0 \text{ bits}$ (5), lo que indica ausencia total de aleatoriedad en la ocurrencia de los eventos.

Este resultado es consistente con el primer objetivo específico de la investigación, que busca disminuir el factor de aleatoriedad en la aplicación sucesiva de refactorizaciones en aplicaciones legadas de software. Al establecer una secuencia de ejecución determinada mediante alguno de los dos enfoques desarrollados, se elimina la incertidumbre asociada al orden de aplicación de las refactorizaciones, favoreciendo un proceso controlado y predecible.

Capítulo 4. Diseño de herramienta

En este capítulo se presenta el diseño de la herramienta desarrollada para apoyar la ejecución ordenada de los microservicios de refactorización. Se describen el proceso que debe realizar el sistema y la interacción entre los usuarios y la herramienta. El diseño permite establecer de manera estructurada las funcionalidades implementadas en el entorno visual. El diseño de esta interfaz considera importante los siguientes factores de usabilidad: panel de configuración, selector de orden, máximo 5 opciones, advertencias por orden diferente al de las reglas de sucesión, máximo de 4 a 7 elementos gráficos por pantalla.

4.1. Diagrama de casos de uso

La Figura 13 presenta el diagrama de casos de uso del Ambiente Visual, en el que se representan las principales funcionalidades del sistema y la interacción de los actores con ellas. El usuario interactúa con las funcionalidades relacionadas con el “Registro de Usuarios”, la “Modificación de Contraseñas”, “Aplicar la Refactorización” y la selección del enfoque de refactorización y la “Evaluación de Satisfacción”. Por su parte, el Administrador participa en “Registro de Usuarios”, “Modificación de Contraseña” y autenticación de usuarios y permisos de acceso, debido a sus privilegios, puede interactuar con las funcionalidades disponibles en el sistema.

El flujo principal del sistema contempla la selección del enfoque de refactorización, a partir de la cual se obtienen los servicios disponibles y se determina la estrategia de secuenciación. La refactorización puede ser secuenciada mediante reglas o mediante criterios ponderados de sucesión.

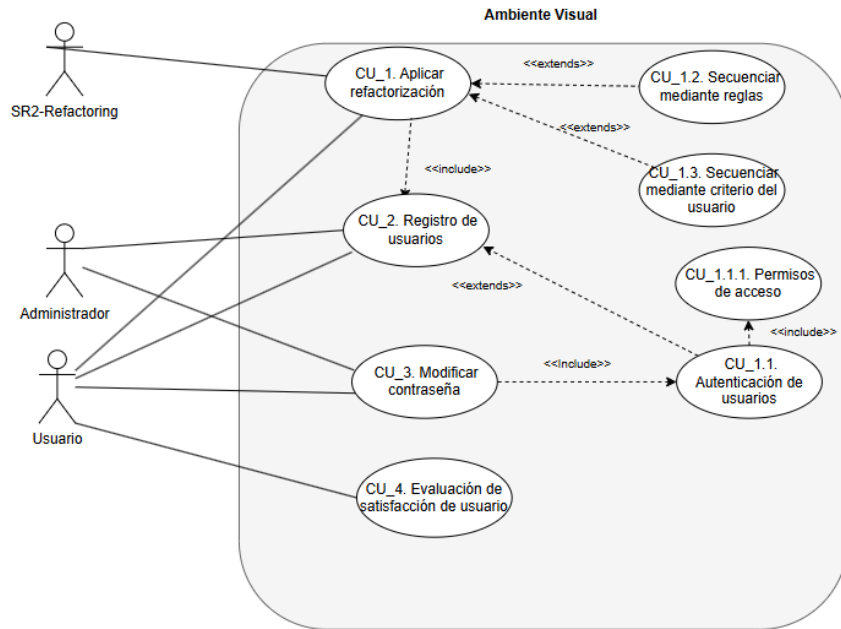


Figura 13. Diagrama de caso de uso.

4.2. Escenarios

En esta sección se describen de manera detallada los escenarios correspondientes a los casos de uso identificados en el diagrama. Cada escenario especifica la información necesaria para comprender cómo se ejecuta una funcionalidad, incluyendo su propósito, actor involucrado, condiciones y secuencia de acciones. Estas descripciones sirven como referencia para definir el comportamiento esperado de la herramienta durante su implementación.

Tabla 7. Aplicar refactorización

1.	ID:	CU_1		
2.	Nombre del Caso de Uso:	Aplicar refactorización		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la Última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring, Administrador		
6.	Descripción:	Esta es la capacidad del sistema para activar una de las secuencias optativas de refactorización de código legado, mediante la aplicación de una secuencia optativa de microservicios de refactorización, elegida por el usuario. Las opciones son por reglas formales (CU_1.2) o basada en el criterio del usuario, como experto en el dominio correspondiente al código legado basándose en		

		criterios y principios de diseño modular y orientado a objetos, así como en métricas internas de calidad en las arquitecturas de software (CU_1.3).
7.	Precondiciones:	Para realizar una secuencia de refactorizaciones, el usuario debe cumplir con los siguientes requisitos: estar registrado en el sistema y estar autorizado, haber seleccionado el código legado que requiere refactorización, y asegurarse de que el sistema de refactorizaciones esté correctamente instalado y listo para ser activado desde esta interfaz de usuario.
8.	Postcondiciones:	Al activar una secuencia de refactorizaciones, el sistema SR2-Refactoring entra en funcionamiento. Como resultado, el sistema de software legado de entrada se refactoriza siguiendo el orden definido en la opción de la secuencia seleccionada por el usuario. Al finalizar el sistema despliega que la refactorización se ejecutó exitosamente y se despliegan los valores de mejora en los factores de deuda técnica tras la ejecución de cada servicio de refactorización incluido en la secuencia.
9.	Escenario Principal de éxito:	El usuario inicia el sistema visual. Y solicita el servicio de refactorización. El sistema visual despliega una caja de diálogo a través de la cual, el usuario introduce sus credenciales (Correo y contraseña). Si el usuario está registrado y cuenta con permisos de acceso, el sistema visual despliega una caja de diálogo a través de la cual, el usuario selecciona el código legado que requiere refactorización. El sistema activa el SR2-Refactoring para evaluar el nivel inicial de de cada factor de deuda técnica que presenta el código legado seleccionado antes de comenzar la refactorización. Desde la interfaz del sistema visual, el usuario selecciona la secuencia de ejecución de los microservicios de refactorización requerida, mediante una de las siguientes opciones: - El sistema visual ejecuta los microservicios en el orden definido por las reglas formales de secuenciación (CU_1.2). - El sistema visual ejecuta los microservicios en el orden ponderado de la secuencia de microservicios de refactorización (CU_1.3). En este caso el sistema despliega el orden de ejecución de los métodos de refactorización en la secuencia ya definida. Una vez seleccionada la opción de la secuencia de refactorizaciones, el sistema inicia el proceso de refactorización invocando al SR2_Refactoring. El sistema confirma el éxito o fracaso del proceso de refactorización. En caso de que el resultado sea exitoso, el sistema despliega los

		valores de mejora en los factores de deuda técnica tras la ejecución de cada servicio de refactorización incluido en la secuencia.
10.	Escenario alternativo (1)	Usuario no registrado se debe ejecutar el caso de CU_2 y continuar el proceso descrito en el escenario principal de éxito.
11.	Escenario alternativo (2)	El usuario está registrado, pero no tiene permisos para ejecutar el servicio de refactorización. El usuario debe solicitar al administrador le conceda el permiso de acceso para refactorizar. Una vez concedido el permiso de acceso para refactorizar, el usuario debe reiniciar el caso de uso CU_1 (escenario principal de éxito).
12.	Escenario alternativo (3)	El usuario no puede seleccionar el código legado porque no cuenta con las carpetas y archivos del código legado que requiere refactorización. Rechazar la transacción, colocar las carpetas y archivos de código legado en el ambiente de ejecución de programa y reiniciar el proceso descrito en el escenario principal de éxito.
13.	Escenario alternativo (4)	El sistema SR2-Refactoring no está disponible para ser ejecutado. Rechazar la transacción. Instalar el sistema SR2-Refactoring en el ambiente de ejecución de programas y reiniciar el proceso descrito en el escenario principal de éxito del CU-1.
14.	Escenario alternativo (5)	No presenta deuda técnica en ninguno de los factores de código mal formado y por lo tanto no hay necesidad de aplicar la secuencia de refactorizaciones. Salir y cerrar el sistema.
23.	Escenario alterno (6)	Durante la ejecución de la secuencia, el proceso es interrumpido por el usuario debido a un imprevisto por parte del usuario, el sistema guarda el progreso, permitiendo reanudar el proceso posteriormente.
24.	Escenario de Fracaso (1):	Ante alguna falla del SR2-Refactoring. El sistema visual muestra un mensaje de error.
25.	Escenario de Fracaso (2):	Factores de deuda técnica que no están programados. El sistema visual muestra un mensaje de error.
26.	Escenario de Fracaso (3):	El usuario intenta ejecutar el proceso de refactorización sin haber colocado el código legado o los archivos incorrectos. El sistema visual muestra un mensaje de error.
27.	Escenario de Fracaso (4):	El usuario no cuenta con los permisos para realizar la refactorización. El sistema visual muestra un mensaje de error, podrá ponerse en contacto con el administrador.

En la Tabla 8 se describe el comportamiento esperado del sistema durante el proceso de autenticación.

Tabla 8. Autenticación de usuarios

1.	ID:	CU_1.1		
2.	Nombre del Caso de Uso:	Autenticación de usuarios		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	04 de enero de 2025
5.	Actores:	Usuario, SR2-Refactoring		
6.	Descripción:	El usuario debe identificarse con correo electrónico y su contraseña.		
7.	Precondiciones:	El usuario debe estar registrado en el sistema. El ambiente visual debe estar disponible y en funcionamiento.		
8.	Postcondiciones:	El usuario obtiene acceso al sistema tras una autenticación exitosa.		
9.	Escenario Principal de éxito:	El usuario inicia la sesión y accede al entorno visual. El usuario ingresa su correo electrónico y su contraseña. El sistema verifica la autenticación del usuario y sus permisos de acceso, si son correctas, el sistema activa al caso de uso CU_1.		
10.	Escenario Alternativo (1):	El usuario olvida su contraseña. El sistema envía un código al correo electrónico del usuario para asegurar su identidad. El usuario escribe el código recibido en una caja de diálogo para tal efecto. En caso de que coincida el código recibido el ambiente visual permite restablecer la contraseña a través del caso de uso CU_3.		
11.	Escenario de Fracaso (1):	El usuario introduce credenciales incorrectas. El sistema muestra un mensaje de error indicando que las credenciales son inválidas.		
12.	Escenario de Fracaso (2):	El sistema no está disponible debido a un error técnico; se notifica al usuario que intente nuevamente más tarde.		

La Tabla 9 presenta el escenario correspondiente al caso de uso “Secuenciar mediante reglas”, se observa el proceso para ejecutar la secuencia de microservicios de refactorización de acuerdo con las reglas formales de secuenciación definidas para SR2-Refactoring.

Tabla 9. Secuenciar mediante reglas

1.	ID:	CU_1.2
2.	Nombre del Caso de Uso:	Secuenciar mediante reglas

3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring		
6.	Descripción:	El sistema determina la secuencia de ejecución de los microservicios de refactorización utilizando las reglas formales de secuenciación definidas para SR2-Refactoring. Posteriormente, ejecuta la secuencia establecida sobre el código legado seleccionado.		
7.	Precondiciones:	El usuario se encuentra autenticado. El usuario seleccionó el código legado que será refactorizado. El usuario seleccionó el enfoque "Secuenciar mediante reglas".		
8.	Postcondiciones:	El sistema visual obtiene la evaluación inicial del código sin refactorizar. El sistema obtiene el código refactorizado y la evaluación final de la deuda técnica.		
9.	Escenario Principal de éxito:	El usuario selecciona el enfoque Secuenciar mediante reglas. El sistema visual consulta las reglas formales de secuenciación. El sistema visual determina automáticamente el orden de ejecución de los microservicios de refactorización. El sistema visual envía la secuencia determinada a SR2-Refactoring. El sistema visual presenta al usuario el código refactorizado y los resultados de la evaluación.		
10.	Escenario Alternativo (1):	El código legado no presenta deuda técnica en ninguno de los factores evaluados. El sistema visual informa al usuario que no es necesario ejecutar la secuencia de refactorización y finaliza el proceso.		
11.	Escenario de Fracaso (1):	SR2-Refactoring no se encuentra disponible para ejecutar la secuencia de refactorización. El sistema visual informa al usuario que el proceso no puede continuar.		
12.	Escenario de Fracaso (2):	Se produce un error durante la ejecución de alguno de los microservicios de refactorización. El sistema visual informa al usuario que la secuencia no pudo completarse.		

La Tabla 10 presenta el escenario correspondiente al caso de uso “Secuenciar mediante

criterios del usuario”, se observa el proceso para ejecutar la secuencia de microservicios de refactorización de acuerdo con los criterios ponderados de sucesión definidos por el usuario.

Tabla 10. Secuenciar mediante criterios del usuario

1.	ID:	CU_1.3		
2.	Nombre del Caso de Uso:	Secuenciar mediante criterios del usuario		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring		
6.	Descripción:	El sistema determina la secuencia de ejecución de los microservicios de refactorización utilizando los criterios ponderados de sucesión de secuenciación definidos por el usuario. Posteriormente, ejecuta la secuencia establecida sobre el código legado seleccionado.		
7.	Precondiciones:	El usuario se encuentra autenticado. El usuario seleccionó el código legado que será refactorizado. El usuario seleccionó el enfoque " Secuenciar mediante criterios del usuario ".		
8.	Postcondiciones:	El sistema visual obtiene la evaluación inicial del código sin refactorizar. El sistema visual obtiene el código refactorizado y la evaluación final de la deuda técnica.		
9.	Escenario Principal de éxito:	El usuario selecciona el enfoque Secuenciar mediante los criterios ponderados de sucesión. El sistema visual consulta la secuencia de criterios ponderados de sucesión. El sistema visual toma el papel de orquestador ante el SR2-Refactoring y envía la secuencia determinada al SR2-Refactoring. El sistema visual presenta al usuario el código refactorizado y los resultados de la evaluación.		
10.	Escenario Alternativo (1):	El código legado no presenta deuda técnica en ninguno de los factores evaluados. El sistema visual informa al usuario que no es necesario ejecutar la secuencia de refactorización y finaliza el proceso.		

11.	Escenario de Fracaso (1):	SR2-Refactoring no se encuentra disponible para ejecutar la secuencia de refactorización. El sistema visual informa al usuario que el proceso no puede continuar.
12.	Escenario de Fracaso (2):	Se produce un error durante la ejecución de alguno de los microservicios de refactorización. El sistema visual informa al usuario que la secuencia no pudo completarse.

En la Tabla 11 se describe el comportamiento del sistema durante el proceso y registro de usuarios.

Tabla 11. Registro de usuarios

1.	ID:	CU_2		
2.	Nombre del Caso de Uso:	Registro de usuarios		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring		
6.	Descripción:	El usuario debe registrarse en el ambiente visual e interactivo de microservicios de refactorización.		
7.	Precondiciones:	El usuario debe tener acceso a internet. El sistema debe estar en funcionamiento y disponible. El sistema debe desplegar una caja de diálogo para el registro de usuarios. El sistema debe desplegar una caja de diálogo para otorgar los permisos de acceso.		
8.	Postcondiciones:	Una vez registrado el sistema debe confirmar la creación exitosa del registro mediante un mensaje y otorgar los permisos de acceso.		
9.	Escenario Principal de éxito:	El usuario inicia la sesión y accede al sistema visual, donde el sistema visual permite crear una nueva cuenta mediante una caja de diálogo para el registro de usuarios. El usuario proporciona la información requerida: nombre, correo electrónico y contraseña. El sistema valida los datos ingresados y, si son correctos, registra al usuario. Se activa una caja de diálogo para otorgar permisos de acceso, finalmente, el sistema asigna por definición el acceso total a la funcionalidad del SR2-Refactoring y muestra un mensaje de		

		confirmación indicando que el registro se ha completado con éxito.
10.	Escenario Alternativo (1):	Si el usuario ya está registrado, el sistema muestra un mensaje indicando que el usuario debe iniciar sesión en lugar de registrarse.
12.	Escenario de Fracaso (1):	El sistema no está disponible debido a un error técnico; se debe notificar mediante un mensaje de error.
13.	Escenario de Fracaso (2):	Faltan datos correspondientes al registro de la cuenta; el sistema debe notificar mediante un mensaje.

En la Tabla 12 presenta los permisos disponibles en el sistema, los cuales pueden ser asignados por el administrador de acuerdo con las funciones que correspondan a cada usuario.

Tabla 12. Permisos de acceso

1.	ID:	CU_1.1.1		
2.	Nombre del Caso de Uso:	Permisos de acceso		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring, Sistema visual		
6.	Descripción:	Este escenario es una descripción del proceso para otorgar los permisos de acceso al SR2-Refactoring, correspondientes al usuario.		
7.	Precondiciones:	El usuario debió haber ingresado previamente al sistema visual para su registro como tal. El sistema SR2-Refactoring debe estar en funcionamiento y disponible.		
8.	Postcondiciones:	Los permisos del usuario se configuran correctamente.		
9.	Escenario Principal de éxito:	El usuario accede al sistema visual. El sistema visual despliega los tipos de usuario en una caja de diálogo (Administrador e usuario). Si el tipo de usuario es administrador el sistema visual otorgará permisos de acceso a la información de cuentas de usuario, para lectura, escritura y modificaciones. Si el tipo de usuario es “usuario” tendrá acceso al		

		sistema SR2-Refactoring para la evaluación de factores de calidad de código mal formado y para refactorizar con cualquiera de las dos políticas de secuenciación de métodos de refactorización. El sistema asigna los permisos de acceso correspondientes, y concede acceso a las funcionalidades específicas, conforme a las políticas de acceso descritas.
12.	Escenario de Fracaso (1):	El usuario intenta registrarse, pero ya existe una cuenta registrada con los mismos datos, el sistema alerta al usuario y lo invita a solicitar al administrador el permiso de acceso.
13.	Escenario de Fracaso (2):	Ocurre un error técnico impide completar el proceso de asignación de permisos. El sistema muestra un mensaje de error al usuario.

La Tabla 13 presenta el escenario correspondiente al caso de uso “Modificar contraseña”, en el cual se describen las condiciones y acciones necesarias para que un usuario registrado pueda actualizar su contraseña dentro del sistema.

Tabla 13. Modificar contraseña

1.	ID:	CU_3		
2.	Nombre del Caso de Uso:	Modificar contraseña		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring, Sistema visual, Administrador		
6.	Descripción:	Este escenario está diseñado para permitir la modificación de la contraseña de un usuario registrado.		
7.	Precondiciones:	El usuario está registrado en el sistema visual. El usuario debe haber iniciado sesión con un perfil válido que permita solicitar modificaciones de contraseña.		
8.	Postcondiciones:	La contraseña se cambia apropiadamente.		
9.	Escenario Principal de éxito:	El usuario accede al sistema visual. El sistema despliega una caja de diálogo para solicitar la contraseña actual. Ejecuta el caso de uso CU_1.1.		

		Si la validación de autenticación es correcta, el sistema solicita al usuario que introduzca una nueva contraseña. El usuario ingresa la nueva contraseña y la confirma. El sistema valida que cumpla con los criterios de seguridad establecidos. Una vez validada, el sistema registra la nueva contraseña en la base de datos. Finalmente, el sistema visual notifica al usuario que su contraseña ha sido modificada exitosamente.
10.	Escenario Alterno (1):	El usuario introduce una contraseña que no cumple con las reglas de seguridad (longitud mínima y caracteres). El sistema muestra un mensaje indicando los criterios que debe cumplir la contraseña. El usuario ingresa una contraseña válida. Continúa con el escenario principal de éxito.
13.	Escenario de Fracaso (1):	Ocurre un error técnico que impide el cambio de contraseña. El sistema muestra un mensaje de error y sugiere intentarlo más tarde.

En la Tabla 14 se puede observar el proceso para evaluar la satisfacción con la herramienta de refactorización a través de una encuesta aplicada después de utilizar el sistema.

Tabla 14. Evaluación de satisfacción del usuario.

1.	ID:	CU_4		
2.	Nombre del Caso de Uso:	Evaluación de satisfacción de usuario		
3.	Autor:	Ninfa Yukary Hernandez Vargas	Última Modificación:	Revisión de Escenarios
4.	Fecha de Creación:	08 de enero de 2025	Fecha de la última Modificación:	12 de agosto de 2026
5.	Actores:	Usuario, SR2-Refactoring, Sistema visual		
6.	Descripción:	Este escenario describe el proceso de evaluación de la satisfacción del usuario con la herramienta de refactorización, a través de una breve encuesta.		
7.	Precondiciones:	El usuario ha iniciado sesión correctamente. El usuario utilizó el SR2-Refactoring para evaluar los factores de calidad de código mal formado y la refactorización correspondiente.		

8.	Postcondiciones:	Los comentarios del usuario se registran en el sistema visual para su registro en un futuro sistema de recomendación.
9.	Escenario Principal de éxito:	El sistema despliega el formato para una encuesta de satisfacción. El usuario completa la encuesta de satisfacción. Los comentarios de la encuesta sirven como retroalimentación para identificar áreas de mejora en la interfaz, usabilidad y eficiencia del proceso.
10.	Escenario Alternativo (1):	El usuario decide no completar la encuesta inmediatamente después de la refactorización, se podrá completar más tarde.
11.	Escenario de Fracaso (1):	Si el usuario no completa correctamente toda la encuesta de satisfacción debe mandar un mensaje de error.
12.	Escenario de Fracaso (2):	El sistema no está disponible debido a un error técnico; se notifica al usuario que intente nuevamente más tarde.

4.3. Estados de objetos

Los objetos principales representan los componentes que participan en la interacción y funcionamiento del sistema. Cada uno mantiene un estado que puede modificarse de acuerdo con las acciones realizadas durante la ejecución de los casos de uso.

Objetos Principales

Sistema Visual: La interfaz gráfica que permite al usuario interactuar con el sistema.

Usuario: El actor principal que interactúa con el sistema para realizar acciones como iniciar sesión, registrarse, evaluar el código legado y refactorizar.

SR2-Refactoring: El responsable de realizar refactorizaciones y evaluaciones en el código legado.

Base de Datos: El componente encargado de almacenar y gestionar los datos necesarios para el funcionamiento del sistema.

Estados y Transiciones

Sistema Visual

Estados:

Tabla 15. Tabla de estados del sistema visual

Estado	Descripción
Iniciando	El sistema visual está iniciando y preparando la interfaz

Desplegando	El sistema visual espera la carga completa de la interfaz, así como desplegando las cajas de diálogo correspondiente.
Esperando	La interfaz está lista y esperando la interacción del usuario.
Mostrando	La interfaz está mostrando la información o procesando la salida según la interacción recibida.

Transiciones:

Iniciando → *Desplegando*: Cuando el sistema visual ha sido inicializado y está listo para usarse.

Desplegando → *Esperando*: Cuando la interfaz se ha cargado completamente y está lista para usarse.

Esperando → *Mostrando*: Cuando el usuario interactúa con la interfaz, y el sistema procesa y muestra la información.

Mostrando → *Esperando*: Cuando la información ha sido mostrada y el sistema vuelve a un estado de espera.

Usuario

Estados:

Tabla 16. Tabla de estados del usuario

Estado:	Descripción
No autenticado	El usuario no ha proporcionado credenciales válidas o no ha iniciado sesión en el sistema.
Autenticado	El usuario ha iniciado sesión correctamente y tiene acceso a las funcionalidades del sistema.
No registrado	El usuario no tiene una cuenta creada en el sistema.
Registrado	El usuario ha completado el proceso de registro y puede acceder al sistema al autenticarse.

Transiciones:

No autenticado → *Autenticado*: Cuando el usuario proporciona credenciales válidas e inicia sesión.

Autenticado → *No autenticado*: Cuando el usuario cierra sesión manualmente o la sesión expira.

No registrado → *Registrado*: Cuando el usuario completa el proceso de registro proporcionando la información requerida.

Registrado → *No registrado*: Cuando se elimina la cuenta del usuario.

SR2-Refactoring

Estados:

Tabla 17. Tabla de estados del SR2-Refactoring

Estado	Descripción
No instalado	SR2-Refactoring no está instalado
Instalado	SR2-Refactoring ha sido instalado y está configurado correctamente.
Inactivo	SR2-Refactoring está instalado, pero no está ejecutando ningún proceso.
Activo	SR2-Refactoring está en ejecución, realizando las operaciones de refactorización o evaluación.

Transiciones:

No instalado → *Instalado*: Cuando SR2-Refactoring se instala y configura en el sistema.

Instalado → *Inactivo*: Cuando SR2-Refactoring está disponible pero no se ha activado aún.

Inactivo → *Activo*: Cuando se activa SR2-Refactoring para ejecutar una refactorización o evaluación.

Activo → *Inactivo*: Cuando SR2-Refactoring termina su operación y vuelve a un estado de espera.

Base de datos

Estados:

Tabla 18. Tabla de estados del base de datos

Estado	Descripción
No instalada	Es necesario que la base de datos sea instalada.
Instalada	La base de datos está instalada con todas las tablas correspondientes

Transiciones:

No instalada → *Instalada*: Cuando se completa el proceso de instalación y configuración de la base de datos.

Código

Estados:

Tabla 19. Tabla de estados del código

Estado	Descripción
--------	-------------

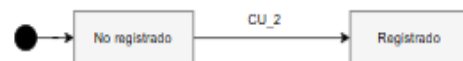
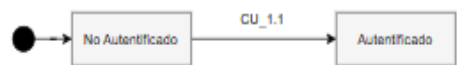
No disponible	El código aún no se ha colocado en el ambiente visual.
Disponible	El código ya puede ser utilizado para poder realizar la refactorización y evaluación.

Transiciones:

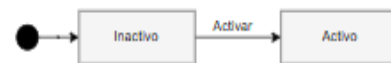
No disponible → *Disponible*: Cuando el usuario coloca al código legado en el lugar correspondiente y puede ser utilizado para refactorización.

Diagrama de estados

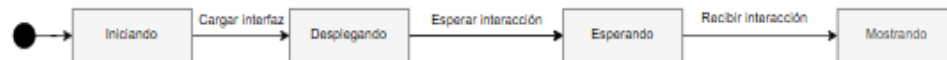
Usuario:



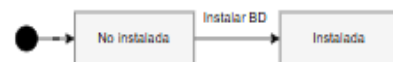
SR2-Refactoring



Sistema visual



Base de datos



Código



Figura 14. Diagrama de estados

Interacción de entidades de software

Entidades Principales

CDialogoRegistrar: Cuadro de diálogo para registrar usuarios en el sistema.

CDialogoPermisos: Cuadro de diálogo relacionado con la gestión de permisos.

CDialogoOpcionR: Cuadro encargado de manejar las opciones relacionadas con la refactorización.

Contexto.oCtx: Representa el contexto del sistema.

aSR2: Entidad que supervisa las acciones de SR2.

aUsuarioPermisos.oEdoPermisos: Gestiona el estado de permisos del usuario.

aUsuarioReg.oEdoReg: Supervisa el estado del registro del usuario.

aCodigoDisponible.oEdoCodigo: Controla el estado del código disponible.

SR21: Responsable de la refactorización con reglas de secuencia.

SR22: Responsable de la refactorización con criterios ponderados.

oCU_2: Caso de uso específico para registro de usuarios.

4.4. Diagrama de secuencia: desarrollar la interfaz de usuario

Objetivos

Mostrar todo el proceso considerado para la refactorización.

Descripción de la Interacción

Inicio de la Interacción

El usuario inicia la interacción enviando una solicitud a través de una interfaz.

Responsabilidad: La entidad CDialogoRegistrar recibe la solicitud inicial.

Procesamiento de la Solicitud

CDialogoPermisos y CDialogoOpcionR colaboran en la gestión de permisos y opciones para el usuario.

Acción: Gestionan los permisos necesarios y seleccionan las opciones referentes con la refactorización.

Actualización de Estados

Las entidades relacionadas con el estado actualizan su información:

- aUsuarioPermisos.oEdoPermisos actualiza el estado de permisos del usuario.
- aUsuarioReg.oEdoReg gestiona el estado del registro del usuario.
- aCodigoDisponible.oEdoCodigo verifica y actualiza el estado del código disponible.

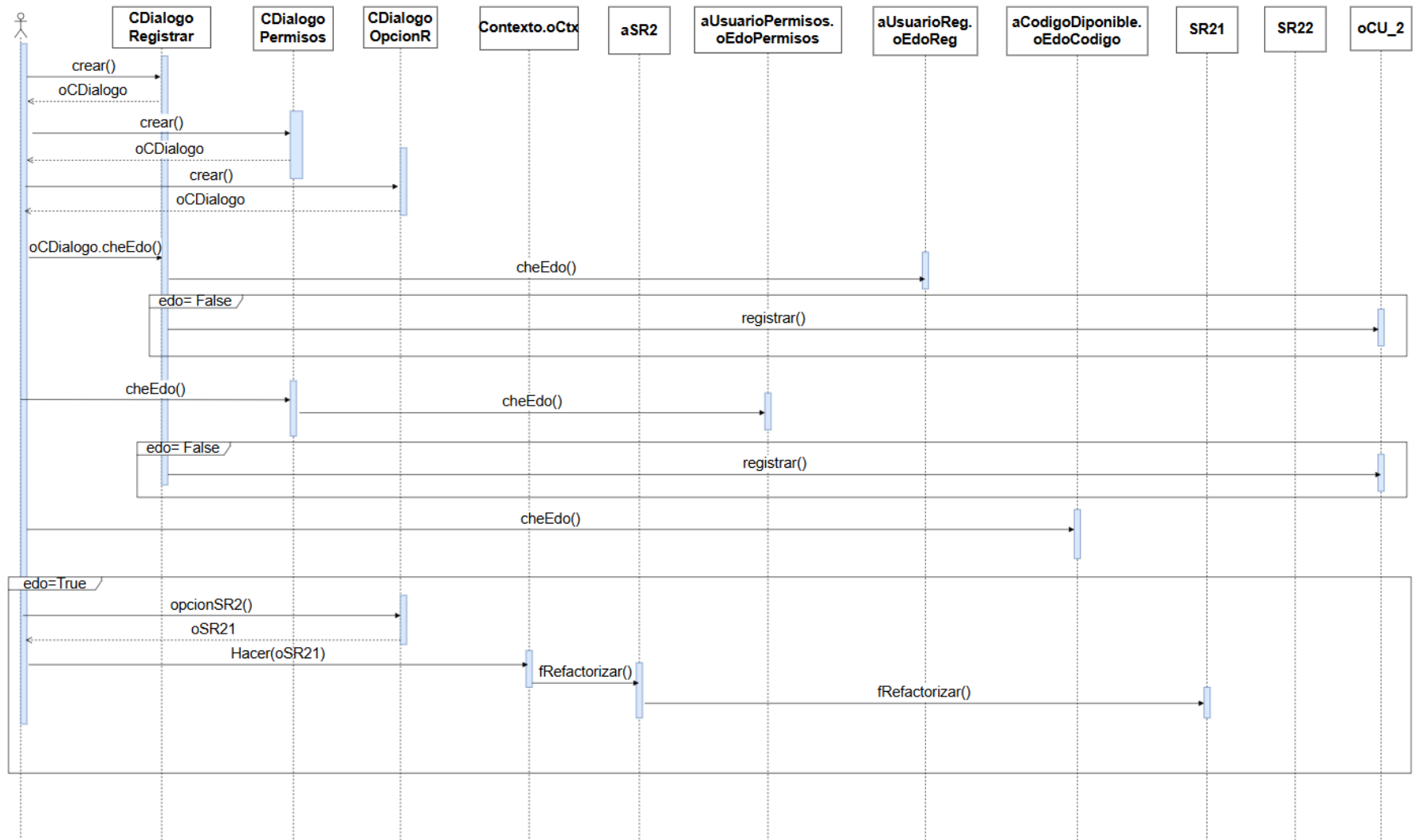


Figura 15. Diagrama de secuencia desarrollar la interfaz de usuario

4.5. Modelo arquitectural

El sistema corresponde a un ambiente visual para la gestión y ejecución de refactorizaciones sobre aplicaciones de software legado mediante una arquitectura basada en microservicios.

El sistema está diseñado para:

- Registrar usuarios y otorgar permisos según su rol.
- Visualizar y ejecutar microservicios de refactorización de forma interactiva.
- Determinar la secuencia de ejecución mediante dos enfoques: reglas de aserción entre microservicios o criterios ponderados basados en factores de calidad interna.

Vista Lógica

El modelo lógico incluye las siguientes clases principales:

- **Cliente**

Responsabilidad: Iniciar la aplicación y coordinar el flujo principal.

- **Contexto**

Responsabilidad: Gestionar el estado del sistema y coordinar las operaciones.

- **aUsuarioPermisos y aUsuarioReg**

Responsabilidad: Gestionar el estado de permisos y registro del usuario.

- **aCodigoDisponible**

Responsabilidad: Validar la disponibilidad del código legado.

- **aSR2 (Refactorización)**

Responsabilidad: Ejecutar la funcionalidad de refactorización según las secuencias definidas.

Vista de Procesos

El sistema sigue los siguientes procesos principales:

- Registro de Usuario:
- Validar datos.
- Crear cuenta.
- Asignar permisos.
- Ejecución de Refactorización:
 - Evaluar código legado antes y después de la refactorización.
 - Refactorización según el enfoque elegido por el usuario y generar resultados.

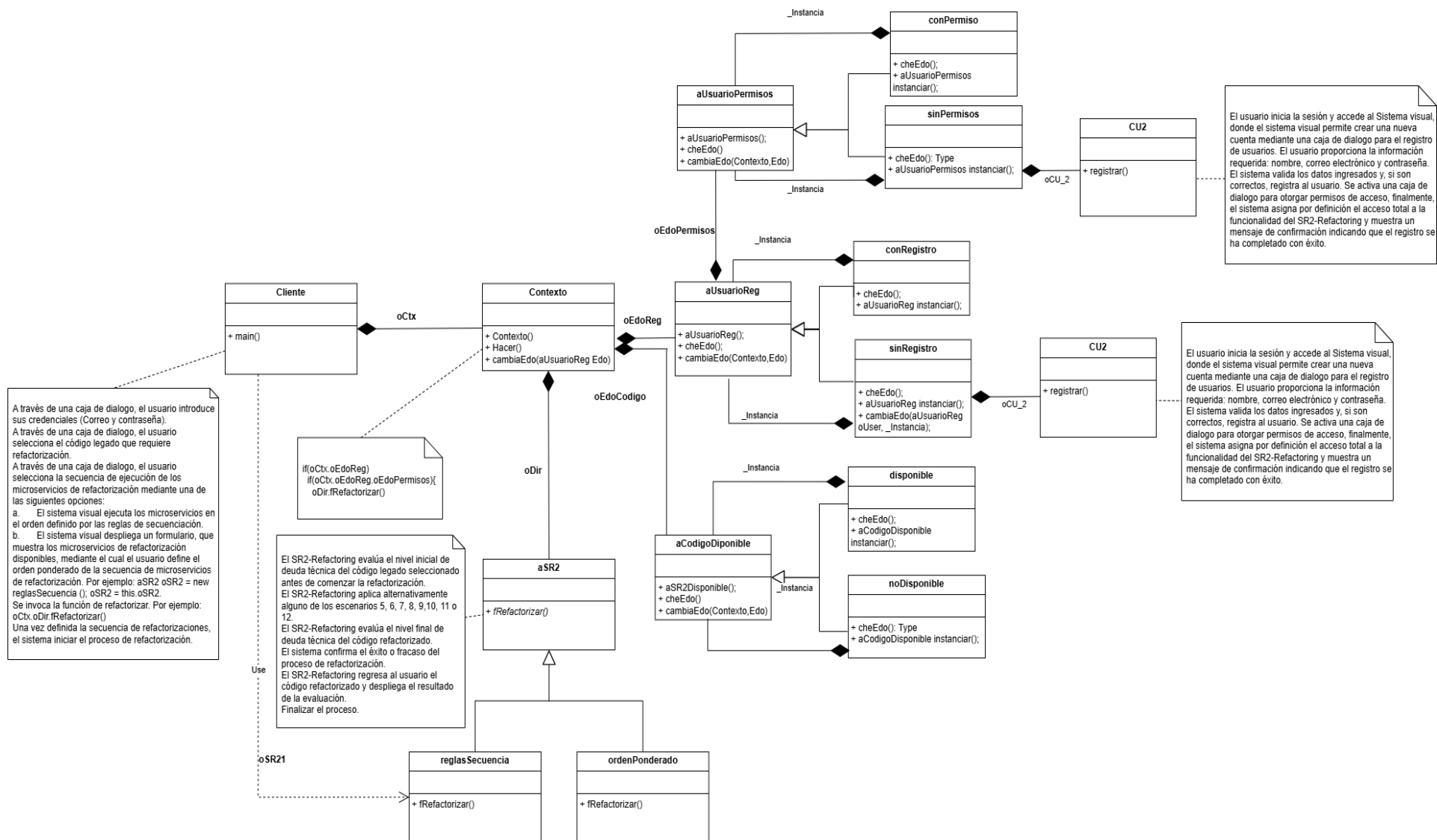


Figura 16. Diagrama de clases

Capítulo 5. Pruebas

Es importante y muy útil la creación de herramientas que permitan reducir la deuda técnica producida por el código mal formado (code smells) por medio de refactorizaciones de manera eficiente, dando como resultado la consideración de los criterios y principios de diseño modular y orientado a objetos, así como los valores correctos de mediciones de calidad interna del código existente en empresas, de negocios, academia, gobierno, e industria, lo que reducirá la fragilidad de los activos de software y la actividad de mantenimiento, así como la ampliación de tiempo de vida útil de los sistemas actuales en operación.

En este proyecto se desarrolló una interfaz al usuario de activación de secuencias ordenadas de métodos de refactorización, bajo dos criterios de secuenciación.

En este documento se presenta el plan de pruebas de esta interfaz al usuario.

5.1. Plan de pruebas

Plan de prueba

ISAVICMR01– 01: Plan de pruebas para la validación de la correcta implementación del ambiente visual e interactivo de orquestación de microservicios de refactorización.

Autoridad de aprobación

Dr. René Santaolaya Salgado

Alcance

El propósito de este Plan de Pruebas es definir las actividades y estrategias necesarias para garantizar que el “ambiente visual e interactivo de orquestación de microservicios de refactorización” cumpla con los requisitos establecidos y opere correctamente.

Objetivos

- Validar que todas las funcionalidades implementadas satisfacen los requisitos funcionales y no funcionales.
- Verificar que el sistema opere de manera correcta de acuerdo a las condiciones previstas de operación.

Identificación única del documento

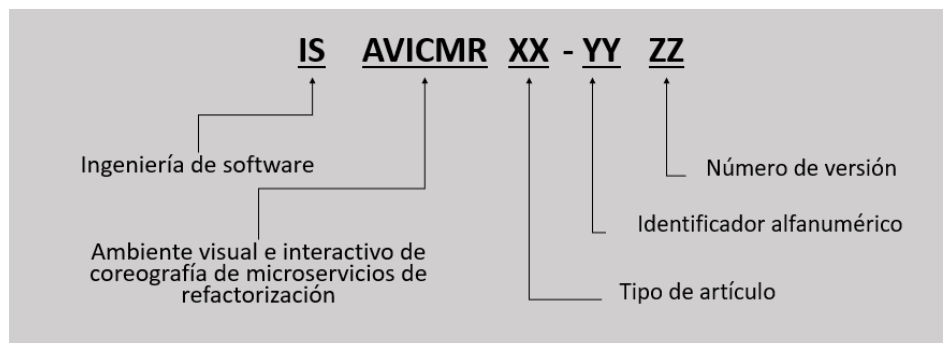


Figura 17. Identificación única del documento

Tabla 20. Tipo de artículo

TIPO DE ARTÍCULO	
01	Plan de pruebas
02	Diseño de pruebas
03	Casos de pruebas

Tabla 21. Documentación de pruebas

DOCUMENTACIÓN DE PRUEBAS	
ISAVICMR01-YYZZ	Plan de pruebas
ISAVICMR02-YYZZ	Diseño de pruebas
ISAVICMR03-YYZZ	Casos de pruebas

Referencias

Las referencias se clasifican en dos categorías para garantizar el cumplimiento de los estándares:

Referencias Externas

Documentos o normativas impuestas por entidades externas:

ISO/IEC/IEEE 29119: Plan de pruebas.

Referencias Internas

Documentación generada por el proyecto:

Análisis de casos de uso del ambiente visual: Documento de especificación de capacidades requeridas para el ambiente visual e interactivo de refactorización, así como el análisis de escenarios de casos de uso.

Estados de objetos: Documento de diagramas de estados de objetos que participan en este proyecto.

Interacción de entidades: Documento de diagramas de secuencia que muestran la vista dinámica de interacción funcional entre objetos de clase participantes en el proyecto.

Modelo arquitectural: Diseño de arquitectura de microservicios que muestra la vista estática de las clases de objetos que participan en el proyecto, sus relaciones y sus responsabilidades.

Glosario

Tabla 22. Glosario

CONCEPTO	DESCRIPCIÓN
Refactorización	La refactorización es el proceso en el cual se realizan cambios en un sistema de software de manera que no altere la forma en que funciona externamente el código, pero si mejore su estructura interna [15].

Código mal formado	El código mal formado son estructuras de código que impactan negativamente, aumentando la deuda técnica, dificultando el mantenimiento y la evolución del software [2].
Coreografía	Las coreografías son planes de coordinación para sistemas concurrentes. Dentro de la programación, la coreografía es utilizada para mantener un acuerdo de las interacciones entre los actores según sus puntos iniciales y finales en la comunicación, que deben realizar para lograr el objetivo que tienen en común. [16].
Microservicios	La arquitectura orientada a servicios (SOA) y el estilo arquitectónico de microservicios se basan en el uso de servicios como el principal componente para implementar funcionalidades de negocios. Los microservicios pueden realizar una comunicación entre ellos, utilizando protocolos estandarizados como es el caso de HTTP y REST [18].
Cobertura de pruebas	Es utilizada principalmente para demostrar, por medio de casos de prueba, que realmente cubren satisfactoriamente los requerimientos especificados a través del código del software [61].

Contexto de la prueba

Proyecto(s) / subproceso(s) de prueba

Esta parte de este documento identifica el alcance y el contexto del plan de pruebas:

Proyecto: "Ambiente visual e interactivo de orquestación de microservicios de refactorización".

Subprocesos de Prueba Cubiertos:

Pruebas de Integración: Interacción entre SR2-Refactoring y el ambiente visual.

Pruebas de Aceptación: Cumplimiento de requisitos especificados para ser cubiertos en este proyecto.

Elementos de Prueba

- Interfaz Visual: Permitir interacción usuario-final con el proceso de secuencias de refactorización.

Alcance de las pruebas

Funcionalidad:

- La correcta aplicación de las dos opciones de secuencias de refactorización.
- Verificar la correcta secuencia de refactorizaciones del método de criterios ponderados.

Limitaciones de las pruebas

- No se prueba que el subsistema SR2-refactoring funcione correctamente, conforme a su especificación y objetivos.
- No se prueba la correcta comunicación entre el subsistema SR2-Refactoring y el subsistema de análisis estático ANTLR.

- No se verifica la correcta secuencia de refactorización del método de reglas de secuenciación.
- No se prueba que las refactorizaciones sean correctas y que eliminen la deuda técnica del código mal formado.
- No se prueba que las métricas de calidad sean calculadas correctamente.

Interfaces:

- Comunicación correcta entre la interfaz de usuario y el sistema SR2-Refactoring
- Consistencia de datos en el ambiente visual e interactivo.

Especificación del diseño de pruebas

Diseño de Prueba: ISAVICMR02–01

Diseño de prueba: Interacción del ambiente visual.

Características:

- En esta prueba se evalúa la correcta autenticación del usuario.
- Se evalúa la ejecución del enfoque mediante el orden definido por las reglas de secuenciación.
- Se evalúa la ejecución del enfoque mediante orden ponderado definido por el usuario.

Resumen

El objetivo es evaluar el correcto funcionamiento de los procesos de interacción del ambiente visual con el subsistema SR2-Refactoring para la refactorización de software legado mediante microservicios. Esto incluye verificar:

- La autenticación adecuada de los usuarios para garantizar un acceso controlado al sistema.
- La capacidad del sistema para adaptar la ejecución de secuencias de refactorización según un orden ponderado establecido por el usuario.

Referencias

- Método para migrar servicios web bajo el protocolo SOAP hacia microservicios basados en REST [57].
- Coreografía de microservicios de refactorización de software legado, proyecto de tesis en desarrollo por parte de Nélida Barón Pérez.
- Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente [8]

Diseño de Prueba: ISAVICMR02–02

Diseño de prueba: Satisfacción del usuario

Características:

- En esta prueba se evaluará la satisfacción del usuario con respecto al entorno visual e interactivo diseñado para la refactorización de software legado.

Resumen

El objetivo principal es evaluar la satisfacción de los procesos de interacción del ambiente visual para la refactorización de software legado. Esto incluye verificar:

- Navegación intuitiva.
- Selección intuitiva de secuencias de microservicios.
- La claridad y accesibilidad de las representaciones visuales.

5.1.1. Casos de Prueba

5.1.1.1. Caso de Prueba: ISAVICMR03–01.

1. Artículo de Prueba: DBLista

Resumen

El DBLista: Sistema desarrollado en lenguaje java que contiene 9 clases, este sistema tiene como objetivo realizar funciones con listas: insertar, eliminar, recorrer, por mencionar algunas.

Objetivo

Verificar que el sistema de refactorización autentica correctamente al usuario, antes de que se puedan realizar el proceso correspondiente. Después de ello se prueba la ejecución de la evaluación y refactorización del código.

Tabla 23. Características del caso de prueba ISAVICMR03–01.

CARACTERÍSTICAS A EJERCITAR	NO. DE ESPECIFICACIÓN DEL DISEÑO DE PRUEBA
Autenticación del usuario	ISAVICMR02–01
Ejecución de la secuencia de refactorizaciones por el método de orden de reglas formales especificadas.	ISAVICMR02–01
Ejecución de la secuenciación de refactorizaciones por el método de orden bajo criterios ponderados.	ISAVICMR02–01

Tabla 24. Resultados antes y después de la refactorización DBLista.

Nº	NOMBRE	PMF	PMMPR	PM	FHIAC	PF	AF	FFC
1	DBLista antes	0.28	0	0.28	0	0	1	0
2	DBLista enfoque reglas	1	0	1	0	0	0	0
3	DBLista enfoque criterios	0.5	0	0.71	0	0	0.5	0

Precondiciones

El usuario debe estar registrado en el sistema. El ambiente visual debe estar disponible y en funcionamiento

Entradas

Para autenticación: Correo electrónico y contraseña del usuario registrados en el sistema.

Para probar la refactorización en ambos enfoques, es importante que el usuario seleccione esta opción y coloque el código correspondiente.

Resultados esperados

Si las credenciales son válidas, el sistema autentica al usuario y permite acceso al ambiente visual interactivo.

Si las credenciales son inválidas, el sistema rechaza el acceso y muestra un mensaje de error de activar CU_2.

Si se tiene el código, se selecciona el enfoque y se refactoriza, el ambiente visual refleja el código refactorizado, así como la evaluación del código legado.

5.1.1.2. Caso de Prueba: ISAVICMR03–02.

1. Artículo de Prueba: Marco estadístico

Resumen

El marco estadístico: Sistema desarrollado en lenguaje java conformado por 46 clases, su objetivo principal es automatizar cálculos estadísticos.

Objetivo

Verificar que el ambiente de refactorización aplica correctamente ambos criterios de refactorización para realizar la refactorización del sistema de Marco Estadístico.

Tabla 25. Características del caso de prueba ISAVICMR03–02.

CARACTERÍSTICAS A EJERCITAR	NO. DE ESPECIFICACIÓN DEL DISEÑO DE PRUEBA
Autenticación del usuario	ISAVICMR02–01
Ejecución de la secuencia de refactorizaciones por el método de orden de reglas formales especificadas.	ISAVICMR02–01
Ejecución de la secuenciación de refactorizaciones por el método de orden bajo criterios ponderados.	ISAVICMR02–01

Tabla 26. Comparación de métricas del marco estadístico antes y después de la refactorización.

N°	NOMBRE	PMF	PMMPR	PM	FHIAC	PF	AF	FFC
1	Marco estadístico antes	0.28	0	0	0	0	0	0
2	Marco estadístico enfoque reglas	1	0	1	0	0	0	0
3	Marco estadístico enfoque criterios	1	0	1	0	0	0.5	0

Precondiciones

El usuario debe estar registrado en el sistema. El ambiente visual debe estar disponible y en funcionamiento

Entradas

Para autenticación: Correo electrónico y contraseña del usuario registrados en el sistema.

Para probar la refactorización en ambos enfoques, es importante que el usuario seleccione la opción y coloque el código correspondiente.

Resultados esperados

Si las credenciales son válidas, el sistema autentica al usuario y permite acceso al ambiente visual interactivo.

Si se tiene el código, se selecciona el enfoque y se refactoriza, el ambiente visual refleja el código refactorizado, así como la evaluación del código legado.

5.1.1.3. Caso de Prueba: ISAVICMR03-03.

1. Artículo de Prueba: PSP.

Resumen

PSP es un sistema permite medir los tiempos que una persona toma para realizar actividades específicas.

Objetivo

Verificar que el ambiente aplica correctamente la refactorización ambos criterios de refactorización para el sistema de PSP.

Tabla 27. Características del caso de prueba ISAVICMR03-03.

CARACTERÍSTICAS A EJERCITAR	NO. DE ESPECIFICACIÓN DEL DISEÑO DE PRUEBA
Autenticación del usuario	ISAVICMR02-01

Ejecución de la secuencia de refactorizaciones por el método de orden de reglas formales especificadas.	ISAVICMR02-01
Ejecución de la secuenciación de refactorizaciones por el método de orden bajo criterios ponderados.	ISAVICMR02-01

Tabla 28. Comparación de métricas de PSP antes y después de la refactorización

Nº	NOMBRE	PMF	PMMPR	PM	FHIAC	PF	AF	FFC
1	PSP antes	0	0	0.2	0	0	1	0
2	PSP enfoque reglas	1	0	1	0	0	0	0
3	PSP enfoque criterios	0.5	0	0.5	0	0	0.5	0

Precondiciones

El usuario debe estar registrado en el sistema. El ambiente visual debe estar disponible y en funcionamiento.

Entradas

Para autenticación: Correo electrónico y contraseña del usuario registrados en el sistema.

Para probar la refactorización en ambos enfoques, es importante que el usuario seleccione la opción y coloque el código correspondiente.

Resultados esperados

Si las credenciales son válidas, el sistema autentica al usuario y permite acceso al ambiente visual interactivo.

Si se tiene el código, se selecciona el enfoque y se refactoriza, el ambiente visual refleja el código refactorizado, así como la evaluación del código legado.

5.1.1.4. Caso de Prueba: ISAVICMR03-04.

1. Artículo de Prueba: PSP Cenidet

Resumen

PSP Cenidet es un sistema permite medir los tiempos que una persona se toma para realizar actividades específicas.

Objetivo

Evaluar la satisfacción del usuario con el sistema PSP Cenidet en relación con la interacción con el ambiente visual.

Tabla 29. Características del caso de prueba ISAVICMR03–04.

CARACTERÍSTICAS A EJERCITAR	NO. DE ESPECIFICACIÓN DEL DISEÑO DE PRUEBA
Satisfacción del usuario	ISAVICMR02–02 Evaluación de experiencia de usuario Por favor, completa la siguiente encuesta para ayudarnos a mejorar tu experiencia. ¡Gracias! Tu respuesta ha sido enviada correctamente. Tu opinión nos ayuda a mejorar Selecciona en cada fila el valor que mejor refleje tu percepción. obstructivo ☐☐☐☐☒☐ Impulsor de apoyo Complicado ☐☐☐☐☒☐ Fácil Ineficiente ☐☐☐☐☐☒ Eficiente

Precondiciones

El usuario debe estar registrado en el sistema. El ambiente visual debe estar disponible y en funcionamiento. El usuario debe haber realizado mínimo una refactorización.

Entradas

Credenciales del usuario para acceder al sistema.

Cuestionario de evaluación de satisfacción.

Resultados esperados

Si el usuario responde el cuestionario para evaluar la experiencia de uso del sistema correctamente, se mostrará un mensaje de éxito.

5.2. Evaluación de resultados

El plan de pruebas tiene como propósito validar la secuencia de refactorización generada a partir del análisis de los métodos: (A) Refactorización de Calificadores de Alcance, (B) Refactorización por Carencia de Abstracción y (C) Reducción de Herencia de Implementación. Dichos métodos tienen como objetivo mejorar la calidad del código legado.

La muestra estuvo conformada por 10 proyectos de software desarrollados en Java 8, seleccionados de repositorios públicos disponibles en internet. La elección de estos proyectos se realizó de manera aleatoria y que presentaran características de código legado.

La selección aleatoria tiene como propósito reducir sesgos en la elección de los casos de estudio, evitando la manipulación intencional de los resultados y favoreciendo la objetividad del

experimento. Al no seleccionar proyectos bajo criterios subjetivos, se asegura que los sistemas analizados representan de manera imparcial distintos escenarios reales de código con problemas estructurales.

El tamaño de la muestra, conformado por 10 proyectos, se determinó bajo un enfoque experimental y no probabilístico, donde el objetivo principal no es la generalización estadística, sino la validación del comportamiento de la secuencia de refactorización propuesta en distintos escenarios de código legado.

Finalmente, los resultados obtenidos muestran un comportamiento consistente en las métricas evaluadas, lo que respalda que el tamaño de la muestra es suficiente para evidenciar el efecto de la secuencia propuesta, sin comprometer la validez interna del estudio.

Si bien cada uno de estos métodos puede ejecutarse de manera independiente, su impacto sobre la arquitectura varía significativamente según el orden en el que se apliquen. Para evitar la reintroducción de deuda técnica, se determinó mediante el análisis de la sección 4.1 realizado, que la secuencia óptima es $C \rightarrow B \rightarrow A$, esto produce un resultado consistente y correcto.

El conjunto de métricas aplicadas permite evaluar diversos aspectos estructurales de los programas analizados antes de la refactorización, particularmente aquellos relacionados con la herencia de implementación, la abstracción y la protección modular. Para ello, se recopilaron distintos proyectos desarrollados en Java 8 que presentan código mal formado y que se encuentran disponibles de manera gratuita en repositorios de internet.

Posteriormente, se realizó la evaluación de cada uno de los proyectos y se obtuvieron los resultados correspondientes a las métricas correspondientes.

En los resultados obtenidos antes de la secuencia de refactorización se observa que la mayoría de los proyectos presentan valores cercanos a 1 en las métricas de herencia. Un valor alto indica que no se logra la reutilización del comportamiento heredado. Esto reduce la capacidad de extender componentes en futuros cambios, disminuyendo la flexibilidad ante modificaciones funcionales.

De forma similar, la métrica AF (Factor de Abstracción) presenta valores de 0 en la mayoría de los sistemas. Los valores bajos significan escasez de clases abstractas e interfaces. Esto dificulta la extensibilidad y la reutilización como se muestra en la Tabla 30.

De igual manera, los proyectos presentan PM (Protección modular) cercano a 0, lo cual indica que los atributos y métodos no están encapsulados correctamente.

Tabla 30. Resultados antes de la refactorización.

N°	NOMBRE	N° CLASES	PMF	PMMPR	PM	FHIAC	PF	AF	FFC
1	Computadora	50	0.08	0	0.11	0	0	1	1.84
2	Control	2	0.5	0	0.5	0	0	1	0
3	lambdasinaccion	2	0.1	0	0.4	0	0	0.5	0

4	Juego-El-ahorcado-en-Java master	3	0	0	0	0	0	0	0
5	Library_Management_System_11 Jan-2022-master	5	0.37	0	0.45	0	0	0	0
6	Movie-Service-Review-Java-main	5	0	0	0	0	0	0	0
7	qurbani-animals-market-main	14	0.019	0	0.038	0	0.083	0.5	0
8	AI-Chat-Bot-master	6	0	0	0	0	0	0	0
9	Llaveros-java-master	4	0	0	0	0	0	0	0
10	FileOperations-master	10	0.04	0	0.08	0	0	1	0

Sea A el enfoque de reglas de aserción entre la coreografía de microservicios, utilizado para determinar el orden de ejecución de las refactorizaciones durante el proceso de evaluación.

Una vez aplicado el enfoque de secuenciación A, los resultados obtenidos son presentados en la Tabla 31. En este caso, la refactorización refleja incrementos considerables en métricas como PMFP y PM, particularmente en proyectos como 3 y 7.

A partir del análisis de la secuencia de refactorización A y de los resultados obtenidos, puede inferirse que probablemente se esté produciendo un fenómeno de duplicación de clases abstractas en este enfoque. A pesar de observarse incrementos en métricas de encapsulación, no se presentan mejoras significativas en el Factor de Abstracción (AF).

Tabla 31. Resultados después de la refactorización enfoque reglas.

Nº	NOMBRE	Nº CLASES	PMFP	PMMPR	PM	FHIAC	PF	AF	FFC
1	Computadora	50	0.12	0	0.20	0.22	0	1	2.34
2	Control	2	0	0	1	0	0	0	0
3	lambdasinaction	2	0.93	0	0.9	0	0	0	0

4	Juego-El-ahorcado-en-Java master	3	0	0	0.34	0	0	0	0
5	Library_Management_System_11 Jan-2022-master	5	0.27	0	0.47	0	0	0	0
6	Movie-Service-Review-Java-main	5	0.61	0	0.75	0	0	0	0
7	qurbani-animals-market-main	14	0.85	0	0.89	0	0	0.18	0
8	AI-Chat-Bot-master	6	0.28	0	0.85	0	0	0	0
9	Llaveros-java-master	4	0.25	0	0.5	0	0	0.29	0
10	FileOperations-master	10	0.03	0	0.66	0	0	0.18	0

Sea B el enfoque que consiste en permitir la sucesión de refactorizaciones mediante la priorización ponderada de factores de calidad interna, criterios y principios de diseño.

Después de aplicar la secuencia de refactorización B, se realizó nuevamente la evaluación de métricas, observándose mejoras importantes en aspectos clave del diseño orientado a objetos. En varios proyectos se aprecia un incremento en la encapsulación y modularidad (PM), así como un uso más adecuado de abstracciones (AF). Asimismo, en el caso del proyecto 1 se observa una mejora en la implementación de herencia, reflejada en un aumento del valor de FHIAC.

En proyectos como 1, 2 y 10, la métrica AF (Factor de Abstracción) alcanza el valor máximo, lo que indica un uso adecuado de clases abstractas e interfaces después de la refactorización. Además, algunos proyectos presentan incrementos en PMFP (Protección de Métodos), como es el caso del proyecto 1.

En términos generales, los proyectos refactorizados con el enfoque B, muestran una estructura organizada y alineada con buenas prácticas de diseño orientado a objetos, mejorando su capacidad para ser mantenidos, ampliados y reutilizados. Estos resultados se presentan en la Tabla 32.

Tabla 32. Resultados después de la refactorización enfoque criterio.

No.	Nombre	No. Clases	PMF P	PMMPPr	PM	FHIAC	PF	AF	FFC
1	Computadora	50	0.11	0	0.16	0.22	0	1	1.90
2	Control	2	0	0	0	0	0	1	0
3	lambdasinaction	2	0.1	0	0.4	0	0	0.5	0
4	Juego-El-ahorcado-en-Java master	3	0.40	0	0.36	0	0	0	0
5	Library_Management_System_11 Jan-2022-master	5	0.27	0	0.55	0	0	0	0
6	Movie-Service-Review-Java-main	5	0	0	0.66	0	0	0	0
7	qurbani-animals-market-main	14	0	0	0.87	0	0.16	0.318	0
8	AI-Chat-Bot-master	6	0.26	0	0.18	0	0	0	0
9	Llaveros-java-master	4	0.21	0	0.22	0	0	0	0
10	FileOperations-master	10	0.03	0	0.22	0	0	1	0

Capítulo 6. Conclusiones y trabajos futuros

6.1. Conclusión

La revisión de los entornos de desarrollo integrados, como Eclipse, IntelliJ IDEA y Visual Studio, permitió identificar que, si bien estas herramientas facilitan la modificación del código y reducen el esfuerzo manual, presentan limitaciones importantes, tales como la generación de refactorizaciones incompletas, soporte restringido a ciertas técnicas y capacidad limitada de adaptación a las necesidades específicas del desarrollador. De forma similar, herramientas especializadas como JDeodorant, MultiRefactor y jSparrow, aunque incorporan mecanismos automatizados o semiautomatizados, no ofrecen un soporte integral que considere simultáneamente la personalización, la evaluación de calidad y la toma de decisiones en el proceso de refactorización. Además, la aplicación sucesiva de refactorizaciones puede generar

contradicciones entre técnicas orientadas a eliminar deuda técnica, provocando efectos no deseados sobre la estructura y calidad del software.

En este contexto, la presente investigación propuso y desarrolló un entorno visual e interactivo para la refactorización mediante microservicios, el cual permite al usuario definir la secuencia de refactorización de manera intuitiva. A diferencia de las herramientas tradicionales, la solución propuesta integra la ejecución de microservicios de refactorización con una representación visual que facilita la comprensión del proceso, así como la selección del orden de aplicación en función de criterios de calidad y reglas de dependencia entre métodos.

Los resultados obtenidos demuestran que el orden de ejecución de las refactorizaciones influye directamente en la calidad del código resultante. En particular, se identificó que la secuencia $C \rightarrow B \rightarrow A$ genera soluciones consistentes, reduciendo redundancias y mejorando la coherencia estructural y semántica del sistema. Asimismo, se comprobó que distintas combinaciones de secuencias producen efectos diferentes en atributos como la abstracción, la flexibilidad y la protección modular, destacando el enfoque de criterios ponderados, ya que se obtienen resultados de manera equilibrada en términos de calidad interna.

Con base en estos resultados, se cumple el objetivo general de la investigación, al desarrollar un entorno visual e interactivo que permite controlar el orden de ejecución sucesiva de refactorizaciones en aplicaciones de software legado mediante dos enfoques complementarios: el primero, basado en reglas de secuenciación entre microservicios de refactorización, y el segundo, fundamentado en la priorización ponderada de factores de calidad interna, criterios y principios de diseño. Ambos enfoques permiten determinar secuencias de ejecución acordes con las necesidades del desarrollador y los objetivos de calidad del software.

Respecto al primer objetivo específico, se logró disminuir el factor de aleatoriedad en la aplicación sucesiva de refactorizaciones al establecer reglas de secuenciación. Este comportamiento fue validado mediante el cálculo de la entropía de Shannon, donde una entropía de 0 bits evidencia que la secuencia queda completamente determinada, eliminando la incertidumbre en el orden de aplicación de las refactorizaciones.

En relación con el segundo objetivo específico, se realizó el análisis de las distintas permutaciones de las secuencias de refactorización, evaluando el impacto de cada uno de los métodos de refactorización. Este análisis permitió identificar la secuencia que conserva mejores características de diseño, proporcionando evidencia para seleccionar una secuencia óptima de ejecución entre los microservicios de refactorización.

De esta manera, el entorno desarrollado no solo permite automatizar la aplicación de refactorizaciones, sino que también apoya activamente la toma de decisiones del desarrollador al proporcionar un medio visual para analizar, comparar y seleccionar secuencias de transformación del código. Esto representa una mejora significativa respecto a las herramientas existentes, al incorporar tanto la dimensión técnica como la cognitiva del proceso de refactorización.

En conclusión, esta investigación demuestra que la integración de microservicios de refactorización en un entorno visual e interactivo contribuye a reducir la complejidad del proceso, mejorar la calidad del software y facilitar la adopción de buenas prácticas de diseño. Asimismo, sienta las bases para futuras investigaciones orientadas a fortalecer la automatización inteligente y la evaluación integral de la calidad en procesos de modernización de software.

6.2. Trabajos futuros

A partir de los resultados obtenidos en esta investigación, se identifican diversas líneas de trabajo que pueden fortalecer y ampliar las capacidades del entorno visual e interactivo propuesto para la refactorización mediante microservicios.

En primer lugar, se plantea la incorporación de mecanismos automatizados de recomendación de secuencias de refactorización, mediante el uso de técnicas de aprendizaje automático o heurísticas basadas en métricas de calidad. Esto permitiría que el sistema no solo facilite la selección manual del orden de ejecución, sino que también sugiera alternativas óptimas en función de las características del código y objetivos del desarrollador.

De igual forma, se plantea la validación del entorno mediante estudios empíricos con desarrolladores, a fin de evaluar su usabilidad, utilidad y efectividad en comparación con herramientas tradicionales. Este tipo de evaluación permitiría identificar áreas de mejora y sustentar con evidencia el impacto de la propuesta.

Bibliografía

- [1] «Modelos de Desarrollo de Software,» *Revista Cubana de Ciencias Informáticas*, p. 15, 2020.
- [2] A. Kovačević, J. Slivka, D. Vidaković y K.-G. Grujić, «Automatic detection of Long Method and God Class code smells through,» *ELSEVIER*, p. 18, 2022.
- [3] M. A. Al Mamun, A. Martini, M. Staron y C. Berger, «Evolution of technical debt: An exploratory study,» *DiVA*, vol. 2476, pp. p. 87-102, 2019.
- [4] W. B. D. Stanislaw Jerzy Niepostyn, «Entropy as a Measure of Consistency in Software Architecture,» *MDPI*, vol. 25, nº 2, p. 328, 2023.
- [5] C. S. N. & K. N. Singh, «Analysis of software maintenance cost affecting factors and estimation models,» *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, vol. 8, nº 09, pp. 276-281, 2019.
- [6] A. Almogahed, «A Refactoring Classification Framework for Efficient Software Maintenance,» *IEEE*, vol. 11, pp. 78904-78917, 2023.
- [7] A. C. B. a. A. G. D. Oliveira, «On the Customization of Batch Refactoring,» *IEEE/ACM 3rd International Workshop on Refactoring (IWorR)*, pp. 13-16, 2019.
- [8] N. B. Pérez, Método de refactorización para mejorar la Protección Modular de Arquitecturas Orientadas a Objetos de Sistemas de Software Existente, Cuernavaca: Centro Nacional de Investigación , 2020.
- [9] O. O. Gutierrez, Re-Factorización De Código Para Reducir El Acoplamiento Entre Clases

Relacionadas Por Herencia De Implementación En Arquitecturas Orientadas A Objetos, Cuernavaca: Centro Nacional de Investigación y Desarrollo Tecnológico, 2020.

- [10] M. A. O. M. K. a. M. V. Alizadeh, «RefBot: Intelligent Software Refactoring Bot,» *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 823-834, 2019.
- [11] H. M. M. O. N. H. Z. G. M. a. Z. A. A. Almogahed, «Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems,» *EEE Access*, vol. 11, pp. 99143-99158, 2023.
- [12] H. M. M. O. N. H. Z. A. A. a. S. O. B. A. Almogahed, «Empirical Investigation of the Diverse Refactoring Effects on Software Quality: The Role of Refactoring Tools and Software Size,» *3rd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1-6, 2023.
- [13] D. O. e. al., «The untold story of code refactoring customizations in practice,» *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 108-120, 2023.
- [14] J. a. N. R. L. a. O. I. a. S. C. a. T. C. S. a. K. M. Ivers, «Industry's cry for tools that support large-scale refactoring,» *Association for Computing Machinery*, p. 163–164, 2022.
- [15] K. B. Martin Fowler, J. Brant, W. Opdyke y D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.
- [16] S. a. M. F. a. P. M. Giallorenzo, «Choral: Object-oriented Choreographic Programming,» *Association for Computing Machinery*, vol. 46, n° 1, p. 59, 2024.
- [17] L. M. .. P. R. V. A. .. S. G. S. A. .. & V. R. J. A. Alamilla Hernández, «Arquitectura REST para el desarrollo de aplicaciones web empresariales,» *Revista Electrónica Sobre Tecnología, Educación Y Sociedad*, vol. 8, n° 15, 2021.
- [18] A. O. a. A. P. G. Blinowski, «Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,» *IEEE*, vol. 10, pp. 20357-20374, 2022.
- [19] A. M. H. O. M. Z. N. H. G. Y. H. A.-M. M. A. & S. Y. Almogahed, «A refactoring categorization model for software quality improvement,» *PLOS ONE*, vol. 18, n° 11, 2023.
- [20] F. B. ,.-T. G. OmarPizarro-Vasquez, «Encapsulation Component and Its Incidence into Scientific Software Performance,» *Communication, Smart Technologies and Innovation for Society . Smart Innovation*, vol. 252, p. 709–719, 2021.
- [21] «An aggregated coupling measure for the analysis of object-oriented software systems,» *Istvan Gergely Czibula, Gabriela Czibula, Diana-Lucia Miholca, Zsuzsanna Onet-Marian*, vol. 148, pp. 1-20, 2019.

- [22] H. W. a. T. A. H. Hourani, «A Code Complexity Model of Object Oriented Programming (OOP),» *EEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, pp. 560-564, 2019.
- [23] A. a. A. E. A. a. M. M. W. a. O. A. Bogart, «Increasing the Trust In Refactoring Through Visualization,» *Association for Computing Machinery*, p. 334–341, 2020.
- [24] S. A. Y. C. M. E. M. a. J. A. T. G. K. Alkharabsheh, «Analysing Agreement Among Different Evaluators in God Class and Feature Envy Detection,» *IEEE Access*, vol. 9, pp. 145191-145211, 2021.
- [25] M. W. M. a. A. O. E. A. AlOmar, «Behind the Intent of Extract Method Refactoring: A Systematic Literature Review,» *IEEE Transactions on Software Engineering*, vol. 50, n° 4, pp. 668-694, 2024.
- [26] E. A. A. M. W. M. A. O. Alex Bogart, «Licensed Under Creative Commons Attribution CC BY Code Smells in Software Development: A Review of Common Issues and Refactoring Approaches,» *International Journal of Science and Research (IJSR)*, vol. 10, p. 334–341, 2021.
- [27] H. a. R. Z. a. Q. L. a. Z. Z. a. G. G. a. M. Y. a. J. H. Chen, «AdaBoost-based Refused Bequest Code Smell Detection with Synthetic Instances,» *International Conference on Dependable Systems and Their Applications (DSA)*, pp. 78-89, 2020.
- [28] I. A. a. M. S. Al-Amri, «Test coverage and impact analysis for detecting refactoring faults: a study on the inline method,» *International Journal of Business Information Systems*, vol. 32, n° 2, pp. 161-169, 2019.
- [29] F. R. R. Ulloa, HEURÍSTICAS INTERCULTURALES PARA EVALUAR LA USABILIDAD DE SITIOS WEB TRANSACCIONALES, Peru: PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ ESCUELA DE POSGRADO, 2021.
- [30] I. H. O.-I. C. C.-M. Edú Ortega Ibarra, «Usabilidad en el Gobierno Electrónico: Heurísticas de Jakob Nielsen y Principios Prácticos de Steve Krug,» *Edähi Boletín Científico de Ciencias Sociales y Humanidades del ICSHu*, pp. 74-81, 2025.
- [31] N. Marsili, «The definition of assertion,» *SSRN Electronic Journal*, 2020.
- [32] A. B. a. A. K. R. Singh, «A framework to improve quality of a Java system by performing refactoring,» *International Journal of System of Systems Engineering*, vol. 10, n° 4, p. 324–324, 2020.
- [33] F. P. M. P. a. Y. G. G. G. Lacerda, «Code smells and refactoring: A tertiary systematic review of challenges and observations,» *Journal of Systems and Software*, vol. 167, p.

110610, 2020.

- [34] E. L. J. A. K. AlOmar, «On the Documentation of Refactoring Types,» *Autom Softw Eng*, vol. 29, n° 9, 2021.
- [35] S. Fernandes, «Towards a Live Environment for Code Refactoring,» *ACM Other conferences*, n° 126, pp. 1-5, 2023.
- [36] M. W. a. J. Bogner, «Towards an Autonomous Bot for Automatic Source Code Refactoring,» *IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pp. 24-28, 2019.
- [37] M. A. a. M. Alshayeb, «A Comparative Study of Automated Refactoring Tools,» *IEEE Access*, vol. 12, pp. 18764-18781, 2024.
- [38] M. A. a. A. Chug, «A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques,» *Journal of Information Processing Systems*, vol. 19, n° 4, pp. 915-934, 2020.
- [39] B. A. a. G. M. Iyad Alazzam, «Impact of Refactoring on OO Metrics: A Study on the Extract Class, Extract Superclass, Encapsulate Field and Pull up Method,» *International Journal of Machine Learning and Computing*, vol. 10, n° 1, 2020.
- [40] A. O. ., E. A. A. ., M. . O. C. e. ., M. W. M. Oumayma Hamdi, «An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications,» *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pp. 28-29, 2021.
- [41] A. C. A. G. I. F. D. C. L. S. W. O. Eduardo Fernandes, «Refactoring effect on internal quality attributes: What haven't they told you yet?,» *Information and Software Technology*, vol. 126, p. 106347–106347, 2020.
- [42] M. A. R. N. S. J. Rosziati Ibrahim, «Reducing redundancy of test cases generation using code smell detection and refactoring,» *Journal of King Saud University - Computer and Information Sciences*, vol. 32, n° 3, pp. 367-374, 2020.
- [43] R. K. N. & K. A. Maini, «HSHEP: An Optimization-Based Code Smell Refactoring Sequencing Technique,» *Journal of Computational and Cognitive Engineering.*, vol. 00, p. 1–17, 2024.
- [44] A. B. e. a. ., «How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?,» *IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pp. 149-159, 2020.
- [45] J. a. O. I. a. N. R. L. a. S. C. Ivers, «Next generation automated software evolution

- refactoring at scale,» *Association for Computing Machinery*, p. 1521–1524, 2020.
- [46] V. K. M. M. W. O. M. O. A. & C. Alizadeh, « Interactive and dynamic multi-objective software refactoring recommendations,» *Proceedings of the 33rd ACM/IEEE International Conference on Automated*, pp. 1-30, 2019.
- [47] A. U. A. C. B. A. G. Eduardo Fernandes, «On the Alternatives for Composing Batch Refactoring,» *IEEE/ACM 3rd International Workshop on Refactoring (IWor)*, pp. 9-12, 2019.
- [48] T. A. M. Alshammari, «Toward a Software Bad Smell Prioritization Model for Software Maintainability,» *Arabian Journal for Science and Engineering*, vol. 46, p. 9157–9177, 2021.
- [49] F. L. N. S. D. T. Valentina Lenarduzzi, «Does migrating a monolithic system to microservices decrease the technical debt?,» *Journal of Systems and Software*, vol. 169, n° 110710, p. 2020, 2020.
- [50] M. a. G. M. S. a. P. F. a. P. F. a. D. L. A. De Stefano, «cASpER: A Plug-in for Automated Code Smell Detection and Refactoring,» *Association for Computing Machinery*, n° 79, pp. 1-3, 2020.
- [51] A. C. Sandhya Tarwani, «An empirical study to determine the optimum refactoring sequence of critically affected classes using Software Quality Attributes,» *Research Article*, 2024.
- [52] K. Tsybulka, *Enhancing Code Quality through Automated Refactoring Techniques*, Madrid: Universidad Politécnica de Madrid, 2024.
- [53] A. S. N. N. E. ESTOMII EDWARD, «On the Impact of Refactorings on Software Attack Surface,» *IEEE Access*, vol. 12, pp. 128570-128584, 2024.
- [54] M. a. M. P. Riansyah, «Empirical Evaluation of the Impact of Refactoring on Internal Quality Attributes,» *2020 International Conference on Advanced Computer Science and Information Systems (ICACISIS)*, pp. 463-470, 2020.
- [55] V. a. B. D. a. P. S. C. a. C. M. a. C. V. Rajendran, «A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework,» *SoutheastCon 2025*, pp. 488-493, 2025.
- [56] S. Podduturi, «AI-Driven Code Optimization: Leveraging ML to Refactor Legacy Codebases,» *North American Journal of Engineering Research*, vol. 6, n° 1, 2025.
- [57] O. H. López, *Método para migrar servicios web bajo el protocolo SOAP hacia, Cuernavaca: Centro Nacional de Investigación*, 2018.

- [58] M. R. Meza, REDUCCION DE LA FRAGILIDAD DEL SOFTWARE LEGADO ORIGINADO POR EL ACCESO A DATOS COMPARTIDOS, Cuernavaca: Centro Nacional de Investigación y Desarrollo Tecnológico, 2022.
- [59] H. R. R. J. V. J. Gamma Erich, Design Patterns: Elements of Reusable Object-Oriented Software, ADDISON-WESLEY, 1994.
- [60] R. G. G. J. d. J. C. N. Jaime Aguilar García, «Vista de La aleatoriedad como parte de lo cotidiano,» *milenaria*, nº 24, pp. 63-65, 2024.
- [61] I. A. a. M. S. Al-Amri, «Test coverage and impact analysis for detecting refactoring faults: a study on the inline method,» *International Journal of Business Information Systems*, vol. 32, nº 2, pp. 161-169, 2019.
- [62] N. B. Pérez, COREOGRAFÍA DE MICROSERVICIOS DE REFACTORIZACIÓN, Cuernavaca: Centro Nacional de Investigación y Desarrollo Tecnológico, 2023.

Anexos

Anexo 1. Manual de usuario

Desarrollo

Base de Datos

La base de datos está organizada en tres tablas principales: usuario, encuesta y resultados. Estas tablas se relacionan para permitir que los usuarios respondan encuestas como se muestra en la Figura 18.

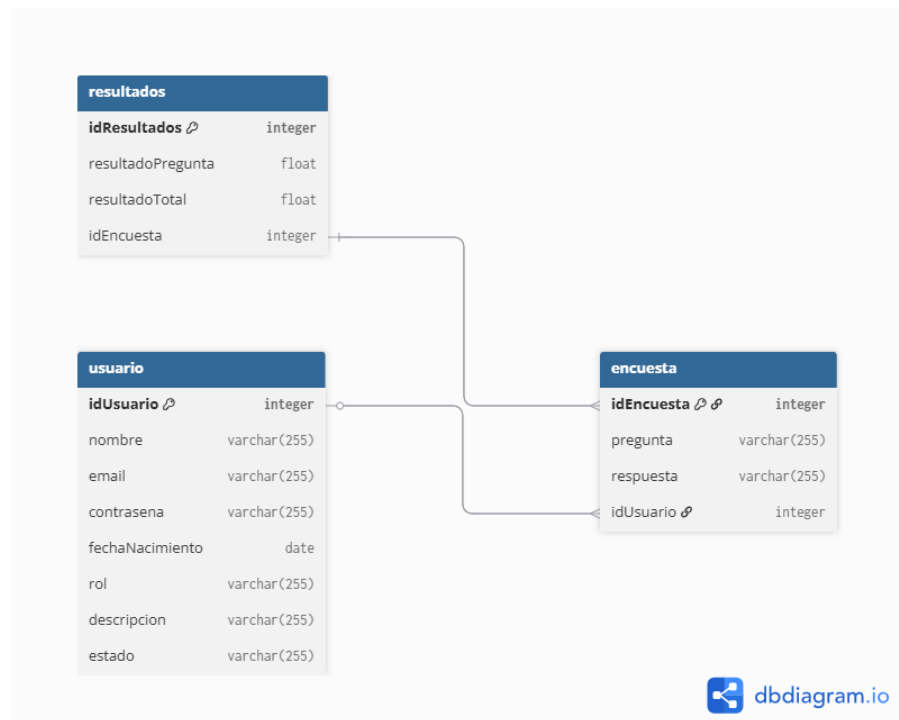


Figura 18. Base de datos.

Manual de usuario

Proyecto: AMBIENTEVISUAL

Contenido: Manual de usuario

Fecha de generación: 2025-10-07

Proyecto React + TypeScript

Este manual describe cómo instalar, ejecutar y mantener un proyecto basado en React, TypeScript y Vite. Incluye instrucciones de uso, estructura del proyecto y configuración recomendada.

1. Requisitos Previos

Antes de comenzar, asegúrese de contar con lo siguiente, instalado en su sistema:

- **Node.js** versión 18 o superior.
- **npm** o **yarn** como gestor de paquetes.
- Un editor de código recomendado: **VS Code**.

2. Instalación del Proyecto

Clona el repositorio o genera el proyecto usando Vite:

```
npm create vite@latest nombre-del-proyecto --template react-ts
```

Ingresa al directorio del proyecto:

```
cd nombre-del-proyecto
```

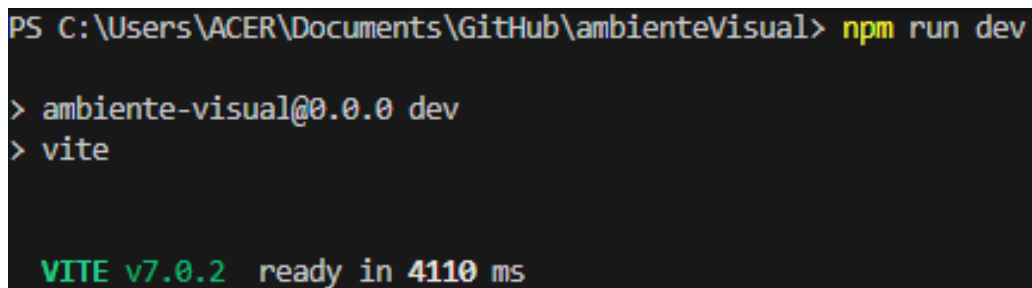
Instala las dependencias:

```
npm install
```

3. Ejecución del Proyecto

Para iniciar el entorno de desarrollo como se puede observar en la Figura 19:

```
npm run dev
```



```
PS C:\Users\ACER\Documents\GitHub\ambienteVisual> npm run dev  
  
> ambiente-visual@0.0.0 dev  
> vite  
  
VITE v7.0.2 ready in 4110 ms
```

Figura 19. Ejecución del proyecto.

El proyecto estará disponible en el navegador en:

<http://localhost:5173>

Para generar la versión de producción:

```
npm run build
```

4. Estructura del Proyecto

```
|— src/  
| |— assets/      # Recursos estáticos  
| |— components/  # Componentes reutilizables  
| |— App.tsx      # Componente principal  
| |— main.tsx     # Punto de entrada  
|— index.html     # Documento base HTML  
|— tsconfig.json  # Configuración de TypeScript  
|— vite.config.ts # Configuración de Vite
```

5. Scripts Disponibles

- **npm run dev** → Modo desarrollo.
- **npm run build** → Construcción para producción.
- **npm run preview** → Vista previa del build.

6. Buenas Prácticas

- Mantener componentes pequeños y reutilizables.
- Asegurar tipado estricto con TypeScript.
- Usar Prettier para un estilo de código consistente.
- Dividir lógica en hooks personalizados cuando sea necesario.

7. Páginas

Inicio de sesión

Al entrar en la aplicación, se nos pedirá iniciar sesión en la aplicación. Para ello se introducirá el nombre de usuario y la contraseña en los campos proporcionados y se pulsará el botón “iniciar sesión” como se ve en la Figura 20.

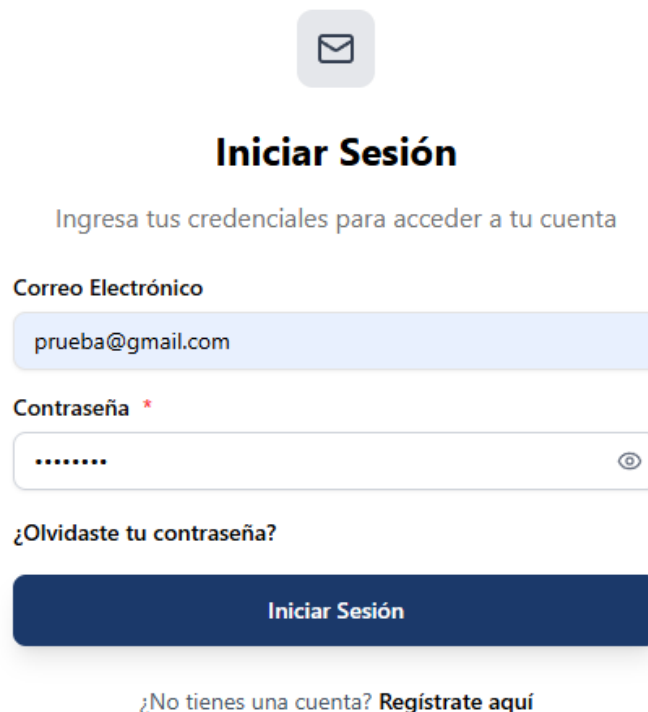


Figura 20. Inicio de sesión

Dashboard

Una vez iniciada la sesión, como se muestra en Figura 21, se despliegan varias opciones: la sección de Refactorización, que permite al ingeniero evaluar y mejorar el código; la sección de Evaluación del Código, destinada únicamente a analizar la deuda técnica; y finalmente la

Encuesta de Evaluación, que nos ayuda a identificar las áreas de oportunidad de la herramienta.



Figura 21. Dashboard.

Refactorización / Evaluación de código

Para el caso de la refactorización es necesario colocar un documento o una carpeta completa, seleccionar el enfoque que desea realizar ya sea por reglas de secuenciación o por criterio de usuario y dar inicio a refactorización como se ve en la Figura 22.



Figura 22. Refactorización.

Para ambos casos se permite realizar la evaluación del código y para ello es necesario seleccionar las métricas que se desean evaluar como se ve en la Figura 23.

- ProcesadorB.java
- Radio.java
- Television.java

0 archivo(s) listo(s) para evaluar

Elige las métricas que deseas usar

☐ Todas (?)	☒ af (?)	☒ ffc (?)	☐ fhi (?)
☐ fhi4 (?)	☐ fhiac (?)	☐ fhij (?)	☐ fmfac (?)
☐ pf (?)	☒ pm (?)	☒ pmff (?)	☒ pmfp (?)
☒ pmfpr (?)	☐ prueba10 (?)	☐ prueba11 (?)	☐ respaldoffc (?)
☐ tpm (?)			

Evaluar código

Figura 23. Evaluación.

Se realiza la evaluación del código dependiendo de las métricas que se le hayan colocado y se puede iniciar la refactorización con el botón que dice iniciar refactorización como se ve en la Figura 24, en dado caso de que sea solo la evaluación este botón no aparece.

Análisis Inicial del Código

Estado actual del código antes de la refactorización

← Volver

▶ Iniciar Refactorización

AF (Factor De Abstracción) Actual:0.3333333333333333	FHI (Factor De Herencia De Implementación) Actual: 1	PF (Factor De Polimorfismo) Actual: 0.6	PM (Protección Modular) Actual: 0
PMFF (Protección Modular De Funciones Friendly) Actual: 0	PMFP (Factor De Protección Modular De Funciones Privadas) Actual: 0	PMFPR (Protección Modular De Funciones Protegidas) Actual: 0	

Figura 24. Tablas de datos de evaluación.

Una vez realizada la refactorización te permitirá realizar la descarga del código como se ve en la Figura 25.

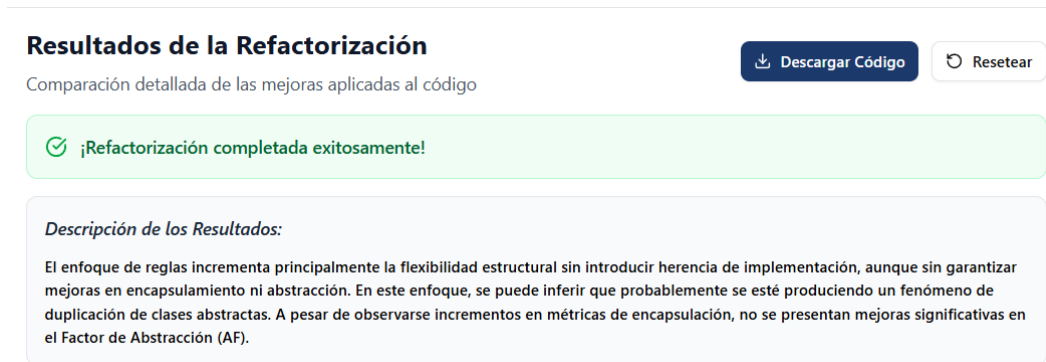


Figura 25. Refactorización exitosa.

En la Figura 26 se pueden observar los resultados de las métricas una vez que se realiza la refactorización junto con la mejora realizada.



Figura 26. Resultados de refactorización

Como se muestra en la Figura 27 se puede observar cada uno de los archivos que se han refactorizado en la carpeta indicada.

« Documentos » ST » reglas » step_3_proteccion_modular

Buscar en ste...

Nombre	Fecha de modificación	Tipo
AbstractProcesador	17/02/2026 08:49 a. m.	Archivo de origen ...
AbstractProcesadorA	17/02/2026 08:49 a. m.	Archivo de origen ...
AbstractProcesadorB	17/02/2026 08:49 a. m.	Archivo de origen ...
Dispositivo	17/02/2026 08:49 a. m.	Archivo de origen ...
Procesador	17/02/2026 08:49 a. m.	Archivo de origen ...
ProcesadorA	17/02/2026 08:49 a. m.	Archivo de origen ...
ProcesadorB	17/02/2026 08:49 a. m.	Archivo de origen ...
Radio	17/02/2026 08:49 a. m.	Archivo de origen ...
Television	17/02/2026 08:49 a. m.	Archivo de origen ...

Figura 27. Archivos refactorizados.

Administrador /página información

Para el caso del administrador tiene una página para realizar las modificaciones de los diferentes usuarios, además de eliminar usuarios como se puede ver en la Figura 28.

Ambiente Visual de Refactorización

Información de Usuarios

Q Filtro por nombre...

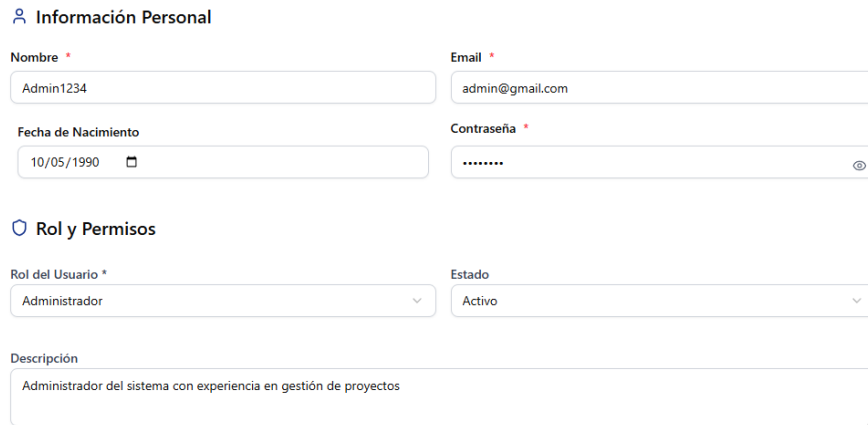
Columnas

Nombre	Email	Rol	Estado	
Admin1234	admin@gmail.com	Administrador	Activo	...
Usuario Prueba122	prueba@gmail.com	Ingeniero	Activo	...
Carlos López	carloslopez@example.com	Administrador	En Revisión	...
Ana Martínez	ana.martinez@example.com	Ingeniero	Activo	...
Luis Rodríguez	luis.rodriguez@example.com	Ingeniero	Activo	...
Sofia Hernández	sofia.hernandez@example.com	Ingeniero	Activo	...

Figura 28. Administrador.

Perfil/editar usuario

Tanto en la edición de usuario como en la edición del propio perfil, cada usuario puede modificar su nombre, correo electrónico, fecha de nacimiento y descripción. En el caso del usuario administrador, además de estos campos, también cuenta con la opción de actualizar el estado y el rol del usuario, tal como se muestra en la Figura 29.



El formulario se divide en dos secciones principales: 'Información Personal' y 'Rol y Permisos'. La sección 'Información Personal' contiene campos para 'Nombre' (con valor 'Admin1234'), 'Email' (con valor 'admin@gmail.com'), 'Fecha de Nacimiento' (con valor '10/05/1990' y un icono de calendario) y 'Contraseña' (con caracteres ocultos por puntos y un icono de ojo). La sección 'Rol y Permisos' contiene un campo para 'Rol del Usuario' (con valor 'Administrador' y una flecha hacia abajo) y un campo para 'Estado' (con valor 'Activo' y una flecha hacia abajo). Debajo de estas secciones hay un campo de texto para la 'Descripción' con el valor 'Administrador del sistema con experiencia en gestión de proyectos'.

Figura 29. Editar perfil.

Anexo 2. Manual de instalación de microservicios de refactorización de software

Introducción

Este manual describe el proceso de instalación y configuración de un entorno de desarrollo de microservicios en Eclipse IDE, enfocado en la implementación de microservicios de refactorización.

Requisitos del sistema

Para la correcta ejecución del entorno, se requiere:

- Java JDK versión 11 o superior
- Eclipse IDE
- Apache Maven
- Spring Boot
- Postman (para pruebas)

Importación de los microservicios en Eclipse

Dado que los microservicios ya están desarrollados, se deben importar como proyectos existentes:

1. Abrir Eclipse IDE
2. Se selecciona el workspace, como se muestra en la Figura 30.

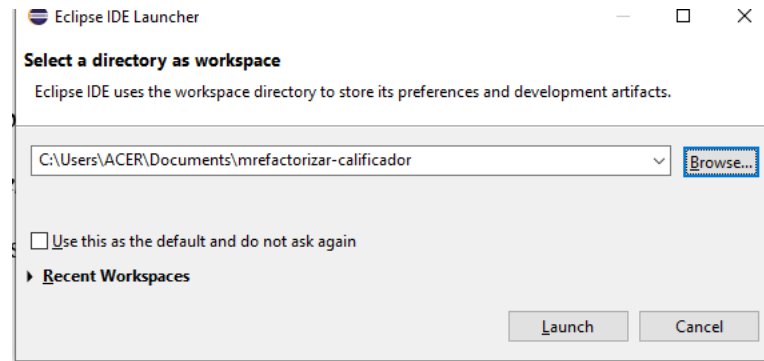


Figura 30. Seleccionar workspace.

3. Crear un nuevo proyecto de Java como se ve en la Figura 31.

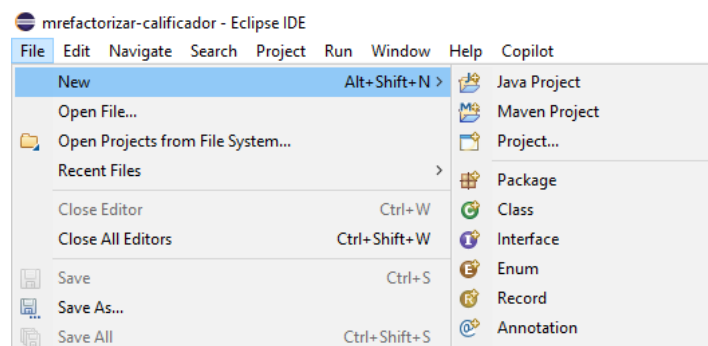


Figura 31. Nuevo proyecto de Java.

4. Ir a:

File → Import → Existing projects from File system or Archive

3. Seleccionar la carpeta del proyecto, como se muestra en la Figura 32.

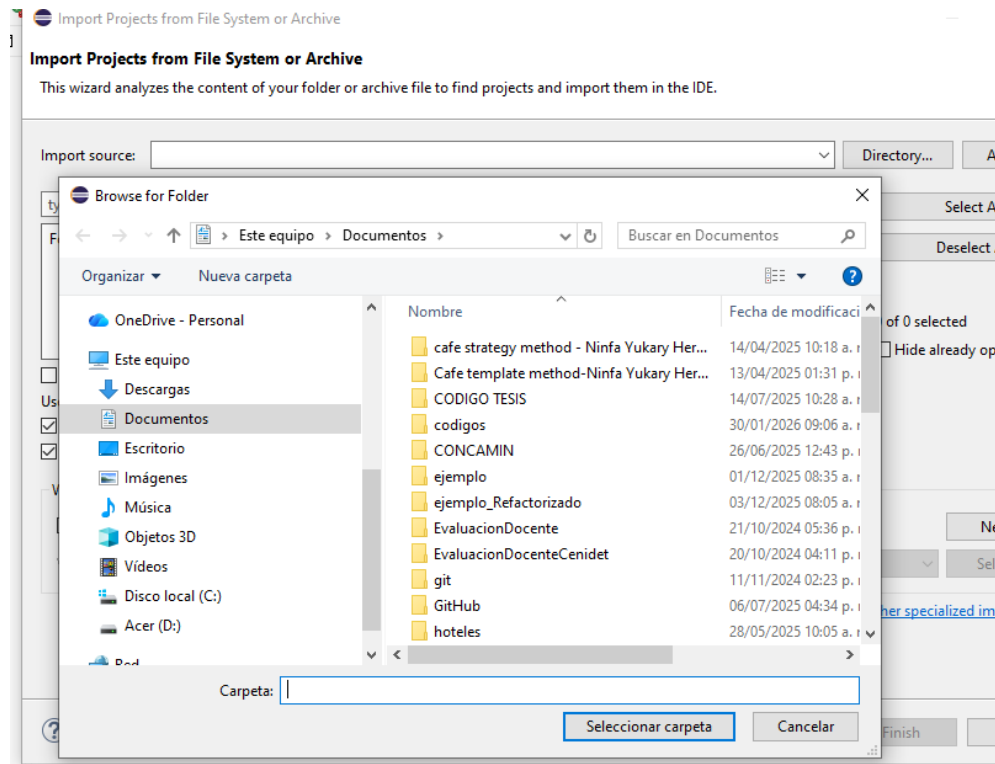


Figura 32. Importar proyecto de Java.

4. Finalizar la importación

Verificación de dependencias

1. Click derecho en cada proyecto
2. Seleccionar:

Java → Update Project

3. Esperar a que se descarguen las dependencias
4. Dar clic derecho para ejecutar cada microservicio desde Maven build como se ve en la Figura 33.

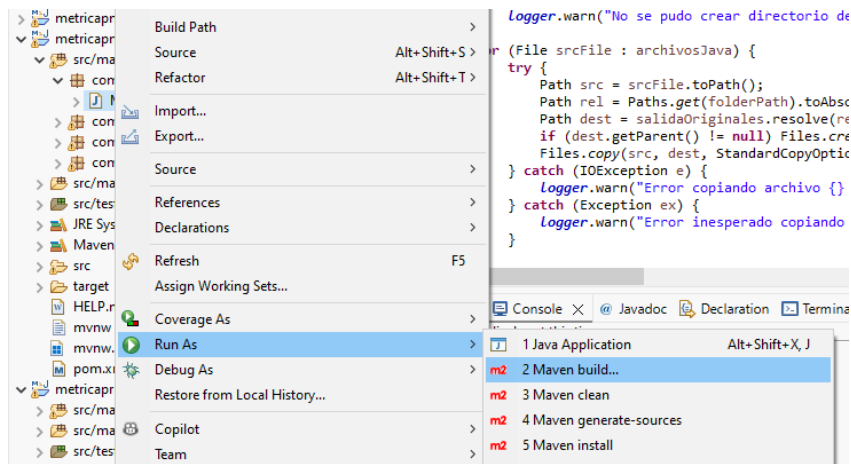


Figura 33. Uso de la función Maven Build.

5. En el campo Goals, ingresar el comando clean package, tal como se muestra en la configuración de la Figura 34, y posteriormente hacer clic en Run.

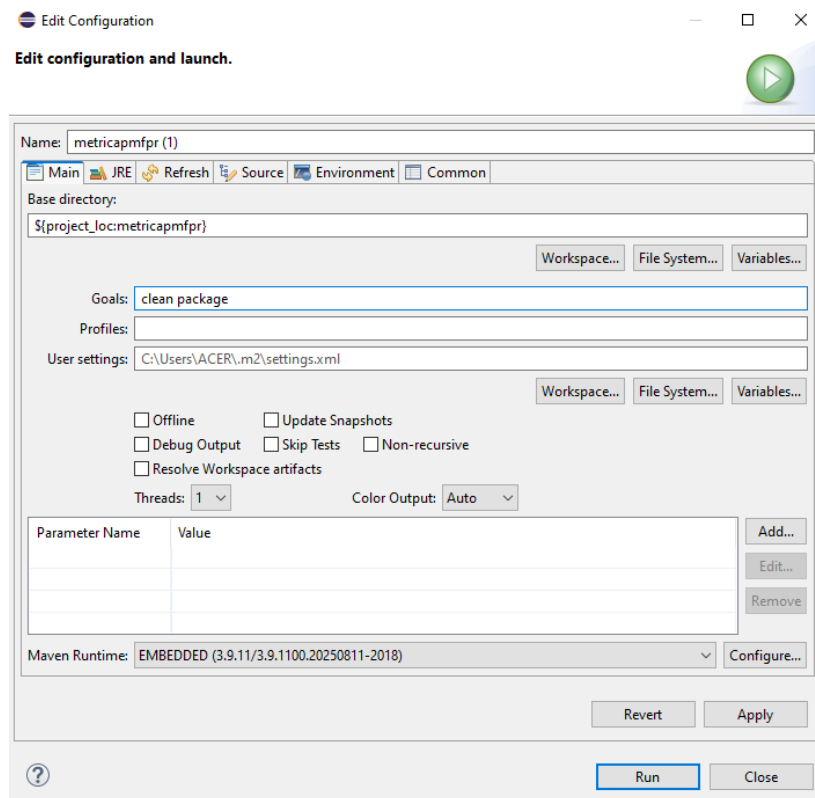


Figura 34. Ejecución de comandos en la interfaz “Edit Configuration”

6. Para la ejecución de todas las métricas, es necesario utilizar el archivo ejecutar.bat, ubicado en la carpeta serviciosNeli. Este archivo contiene la configuración necesaria para ejecutar todas las métricas, como se muestra en la Figura 35. Es importante ajustar previamente las rutas de refactorización de acuerdo con la ubicación de los microservicios en el equipo.

```

REM ===== MÉTRICAS (carpeta serviciosNeli) =====
start cmd /k "cd metricaaf && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafac && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafhi && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafhi4 && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafhiac && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafhij && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricafmfac && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricapf && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricapm && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricapmff && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricapmfp && for %%f in (target\*.jar) do java -jar %%f"
start cmd /k "cd metricapmfpr && for %%f in (target\*.jar) do java -jar %%f"

REM ===== REFACTORIZACIÓN (rutas externas bien formadas) =====

start "" cmd /k cd /d "C:\Users\ACER\Documents\mrefactorizar-calificador\mrefactoriz
|
start "" cmd /k cd /d "C:\Users\ACER\Documents\marco-metricas-service\marcometricas"

start "" cmd /k cd /d "C:\Users\ACER\Downloads\SR2 Refactoring Java\proyectoorando"

```

Figura 35. Archivo ejecutar.bat

Una vez ejecutado este archivo, se debe abrir y ejecutar el microservicio correspondiente al orquestador, dar clic al documento proyectoorquestador.java en el botón izquierdo y ejecutar como java application como se ve en la Figura 36.

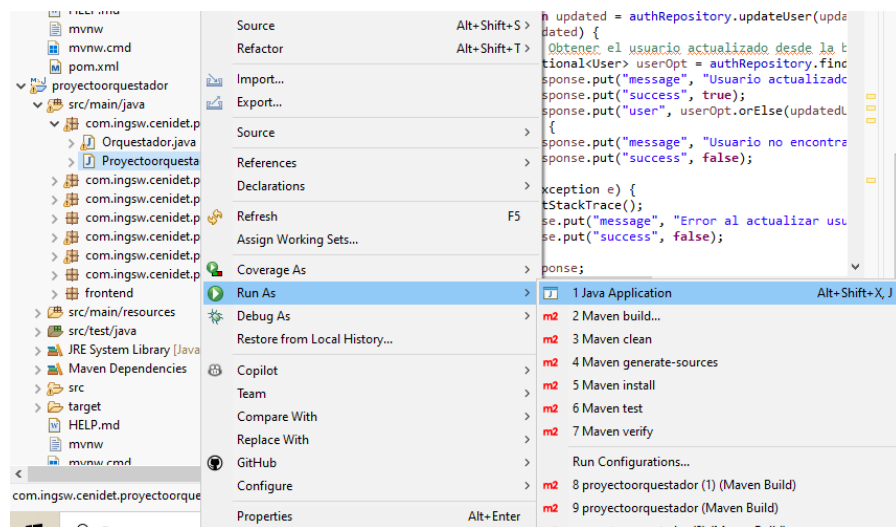


Figura 36. Ejecución del orquestador