SERVICIOS Y SUMINISTROS EN INFORMÁTICA, S.A. DE C.V.

“IMPACTO DE LA REFACTORIZACIÓN EN LA COMPLEJIDAD CICLOMATICA DE UNA APLICACIÓN DE GESTIÓN COMERCIAL”

TESIS PARA TITULACIÓN INTEGRAL

INGENIERÍA EN SISTEMAS COMPUTACIONALES

EGRESADO:
JONATHAN DAVID UITZ EUAN

DIRECTOR Y CODIRECTOR:
DR. JOSE LUIS LIRA TURRIZA
DR. JOSE MANUEL LIRA TURRIZA

Calkiní, Campeche, México 2025

Educación
Secretaría de Educación Pública

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico Superior de Calkiní
Subdirección de Planeación y Desarrollo

Instituto Tecnológico Superior de Calkiní
Departamento de Administración y
Apoyo al Estudiante

Calkiní, Campeche 07/03/2025
P/03/25/622
INGENIERÍA EN SISTEMAS COMPUTACIONALES

CONSTANCIA DE NO INCONVENIENCIA

MODALIDAD: **TESIS**

C. JONATHAN DAVID UITZ EUAN 7255
PASANTE DE LA CARRERA DE
INGENIERÍA EN SISTEMAS COMPUTACIONALES.
PRESENTE

Con relación a lo establecido en el Reglamento de Estudios de Licenciatura, capítulo VII, fracción I, vigente en este Instituto, y una vez emitido el fallo favorable por el director DR. JOSE LUIS LIRA TURRIZA, y el co-director DR. JOSE MANUEL LIRA TURRIZA, respectivamente, este Instituto consta de no tener inconveniencia para la realización del acto protocolario de titulación integral de su trabajo profesional titulado: "**IMPACTO DE LA REFACTORIZACIÓN EN LA COMPLEJIDAD CICLOMATICA DE UNA APLICACIÓN DE GESTIÓN COMERCIAL**".

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
Sabiduría, Fortaleza de Nuestra Cultura®

ITESCAM
INSTITUTO TECNOLÓGICO SUPERIOR DE CALKINÍ
"DEPARTAMENTO DE ADMINISTRACIÓN ESCOLAR Y APOYO A ESTUDIANTES"
CLAVE: 04MSU0013P
CALKINÍ, CAMPECHE, MEXICO.

Dr. Jorge Alfonso Hau Puc.
Departamento de Administración y Apoyo al Estudiante.

C.c.p. Archivo

2025 Año de La Mujer Indígena

Av. Ah Canul S/N por Carretera Federal, Calkiní, Campeche,
C.P. 24900 Tel. 996-8134870,
e-mail: se_dcalkini@tecnm.mx | itescam.edu.mx

ITESCAM

ESPACIO 100% LIBRE DE PLÁSTICO DE UN SOLO USO

Dedicatoria

A Dios, por ser mi guía y fortaleza en cada paso de este camino, iluminándome con fe y esperanza para alcanzar este logro. A mis padres, por su amor incondicional, su apoyo constante y por enseñarme, con su ejemplo, el valor del esfuerzo y la perseverancia. A mis maestros, por compartir su conocimiento y dedicación, dejando una huella imborrable en mi formación profesional y personal.

A todos ellos, mi más profundo agradecimiento.

Agradecimientos

Quiero expresar mi más profundo agradecimiento a Dios, quien ha sido mi guía, fortaleza y fuente de inspiración en todo momento, sin Él este logro no habría sido posible.

A mis padres, especialmente a mi madre, por su amor, apoyo incondicional y sacrificios constantes, y a la memoria de mi padre, quien sigue siendo una luz que guía mi camino y una inspiración en mi vida. Agradezco también a mi familia en general por su cariño y respaldo en cada etapa de este proceso.

A mi pareja, una persona muy especial que ha sido parte fundamental en esta etapa de mi vida, brindándome apoyo, motivación y compañía incondicional. Su presencia ha significado mucho para mí, y estoy profundamente agradecido por todo lo que compartimos durante este tiempo.

A mis compañeros de generación, quienes con su compañerismo y colaboración hicieron de este camino una experiencia más enriquecedora y llevadera.

Agradezco especialmente a los maestros que dejaron una huella en mi formación profesional, en particular al Dr. José Luis Lira Turriza, a la Dra. Yaqueline Pech Huh, y al Dr. José Manuel Lira Turriza, por compartir su conocimiento y orientación, y por ser un pilar en mi desarrollo académico.

Mi más sincero agradecimiento a Plenumsoft, cuya guía y retroalimentación fueron fundamentales bajo la modalidad dual. Agradezco la oportunidad de haber formado parte de la empresa, lo que me permitió desarrollarme profesionalmente y participar en proyectos que contribuyeron a mi crecimiento en el área.

Finalmente, agradezco de corazón a la Fundación José Ortiz Ávila, cuyo apoyo invaluable ha sido fundamental en mi formación y en la realización de este proyecto, por creer en mí y respaldar mis sueños.

Índice

Índice Figuras

Índice de tablas

Resumen

El desarrollo de aplicaciones de gestión comercial enfrenta desafíos relacionados con la mantenibilidad y escalabilidad del código. Uno de los principales indicadores de estas problemáticas es la Complejidad Ciclomática (CC), una métrica que mide la cantidad de caminos lógicos dentro del código y que puede afectar la facilidad de modificación y prueba del software. En esta tesis, se evaluó el impacto de la refactorización en la CC dentro de un módulo de una aplicación de gestión comercial desarrollada en Flutter.

Para ello, se realizó un análisis preexperimental en el que se midió la CC en todos los módulos de la aplicación, identificando aquel con el mayor nivel de complejidad para ser optimizado. Se aplicó la técnica Extract Method, con el objetivo de modularizar el código y reducir su complejidad. Posteriormente, se compararon las métricas antes y después de la refactorización.

Los resultados demostraron que la refactorización permitió disminuir la CC y mejorar la organización del código, facilitando su comprensión y futuras modificaciones. Sin embargo, algunos métodos mantuvieron valores elevados de CC, lo que sugiere que la refactorización es un proceso iterativo que puede requerir optimizaciones adicionales.

Estos hallazgos pueden servir como referencia para desarrolladores y empresas que deseen mejorar la calidad estructural de sus aplicaciones mediante estrategias de refactorización, optimizando la escalabilidad y mantenibilidad del código en sistemas de gestión comercial.

Abstract

The development of commercial management applications faces challenges related to code maintainability and scalability. One of the main indicators of these issues is Cyclomatic Complexity (CC), a metric that measures the number of logical paths within the code and can affect the ease of software modification and testing. In this thesis, the impact of refactoring on CC was evaluated within a module of a commercial management application developed in Flutter.

A pre-experimental analysis was carried out, measuring CC across all application modules to identify the one with the highest complexity for optimization. The Extract Method technique was applied to modularize the code and reduce its complexity. Subsequently, metrics were compared before and after refactoring.

The results demonstrated that refactoring reduced CC and improved code organization, making it easier to understand and modify. However, some methods retained high CC values, suggesting that refactoring is an iterative process that may require further optimizations.

These findings can serve as a reference for developers and companies seeking to improve the structural quality of their applications through refactoring strategies, enhancing code scalability and maintainability in commercial management systems.

Introducción

El desarrollo de aplicaciones de gestión comercial enfrenta desafíos constantes en términos de mantenibilidad y escalabilidad del código. A medida que las aplicaciones crecen y se expanden, la complejidad de su estructura puede aumentar significativamente, dificultando su comprensión, modificación y evolución. Para abordar estos problemas, es fundamental contar con estrategias que permitan optimizar la calidad del código y garantizar su sostenibilidad a largo plazo.

Uno de los principales indicadores de la dificultad de un sistema de software es la Complejidad Ciclomática (CC), una métrica que mide el número de caminos lógicos dentro de un código. Valores elevados de CC pueden hacer que un sistema sea más difícil de probar, depurar y modificar, lo que impacta directamente en su mantenibilidad. En entornos de desarrollo ágil y frameworks modernos como Flutter, la acumulación de código complejo puede convertirse en un obstáculo para la evolución del software, afectando tanto su eficiencia como su capacidad de adaptación a nuevos requerimientos.

Para reducir la CC y mejorar la organización del código, se emplea la refactorización, una técnica que permite optimizar la estructura interna del software sin alterar su funcionalidad. Sin embargo, aunque se ha demostrado que la refactorización puede contribuir a mejorar la calidad del código, su impacto específico en la CC no siempre es predecible y puede depender de factores como la naturaleza del código y las estrategias aplicadas.

Ante este panorama, el propósito de esta tesis es evaluar el impacto de la refactorización en la Complejidad Ciclomática dentro de un módulo de una aplicación de gestión comercial desarrollada en Flutter. Para ello, se realizó un análisis en el que se midió la CC de los módulos de la aplicación y se identificó aquel con el mayor nivel de complejidad para ser optimizado mediante la técnica Extract Method, con el objetivo de mejorar la organización del código y facilitar su mantenibilidad.

Esta tesis es relevante porque contribuye a la comprensión del papel de la refactorización en la reducción de la CC y en la optimización del código fuente en sistemas de gestión comercial. Además, los hallazgos obtenidos pueden servir como referencia para desarrolladores y equipos de software interesados en aplicar buenas prácticas de desarrollo para mejorar la calidad estructural de sus aplicaciones.

El documento se encuentra estructurado de la siguiente manera: en el Capítulo 1, se presenta el planteamiento del problema, los objetivos y la importancia del estudio; en el Capítulo 2, se aborda el marco teórico, incluyendo los conceptos fundamentales sobre Complejidad Ciclomática y refactorización, así como la hipótesis de la tesis; en el Capítulo 3, se describe la metodología utilizada para la evaluación de la CC antes y después de la refactorización; en el Capítulo 4, se exponen los resultados obtenidos y su análisis; y finalmente, en el Capítulo 5, se presentan las conclusiones y recomendaciones de la tesis.

A través de este análisis, se espera aportar conocimiento sobre cómo la refactorización puede influir en la calidad del código y proporcionar estrategias efectivas para mejorar su mantenibilidad y escalabilidad en aplicaciones comerciales.

Capítulo 1

Problema de investigación

Planteamiento del problema

El desarrollo de aplicaciones de gestión comercial enfrenta múltiples desafíos en términos de calidad y mantenimiento del código. A medida que las aplicaciones crecen y evolucionan, la complejidad del software puede aumentar significativamente, dificultando su comprensión, modificación y escalabilidad. Uno de los principales indicadores de esta problemática es la Complejidad Ciclomática (CC), una métrica que cuantifica la cantidad de caminos lógicos en el código y permite estimar su dificultad de prueba y mantenimiento.

En aplicaciones móviles desarrolladas con frameworks modernos como Flutter, la estructuración del código es crucial para garantizar la estabilidad y eficiencia del sistema. Sin embargo, en el desarrollo de aplicaciones de gestión de ventas, que suelen integrar múltiples módulos interdependientes, la acumulación de código sin una adecuada planificación y optimización puede resultar en estructuras altamente complejas y difíciles de mantener. Este problema no solo impacta en la capacidad de escalabilidad del software, sino que también puede incrementar los costos y tiempos de desarrollo debido a la dificultad para implementar nuevas funcionalidades o corregir errores.

La refactorización es una técnica ampliamente utilizada para mejorar la calidad del código sin alterar su comportamiento funcional. Mediante la reorganización y optimización del código fuente, se busca reducir la complejidad innecesaria y mejorar la mantenibilidad del software. No obstante, investigaciones previas han señalado que el

impacto de la refactorización en la Complejidad Ciclomática no siempre es predecible, ya que puede reducirse en algunas áreas del código mientras que en otras podría desplazarse o incluso aumentar. Esto resalta la importancia de evaluar empíricamente sus efectos en contextos específicos, especialmente en aplicaciones de gestión comercial, donde la estabilidad y escalabilidad son factores clave.

En este estudio, se analizará cómo la refactorización afecta la Complejidad Ciclomática en uno de los módulos de una aplicación móvil de gestión comercial desarrollada en Flutter. Para ello, se medirá esta métrica antes y después de aplicar mejoras en el código, lo que permitirá cuantificar el impacto de la refactorización en términos de reducción de la complejidad lógica. A través de este análisis, se busca proporcionar evidencia empírica sobre la importancia de una arquitectura optimizada para mejorar la sostenibilidad del software a largo plazo. Además, los resultados de este estudio pueden servir como referencia para otros desarrolladores y equipos que trabajen en aplicaciones con características similares, ayudando a tomar decisiones informadas sobre estrategias de optimización del código.

Objetivos generales y específicos

Objetivo General

Evaluar el impacto de la refactorización en la Complejidad Ciclomática (CC) de un módulo en una aplicación de gestión comercial, mediante la comparación de métricas antes y después de la aplicación de mejoras en el código.

Objetivos específicos

- Medir la Complejidad Ciclomática de todos los módulos de la aplicación móvil para obtener una visión general de la complejidad del sistema.
- Identificar el módulo con el mayor promedio de Complejidad Ciclomática en comparación con el resto del sistema.
- Analizar los métodos dentro del módulo seleccionado y determinar cuáles presentan los valores más elevados de Complejidad Ciclomática.
- Aplicar técnicas de refactorización en los métodos identificados con el fin de optimizar su estructura y reducir la complejidad innecesaria.
- Comparar los valores de Complejidad Ciclomática antes y después de la refactorización para evaluar el impacto de los cambios realizados.
- Documentar los resultados obtenidos y generar conclusiones sobre la efectividad de la refactorización como estrategia para mejorar la mantenibilidad del código.

Impacto potencial

El presente estudio tiene el potencial de generar conocimiento aplicable en el desarrollo y mantenimiento de aplicaciones de gestión comercial, proporcionando estrategias concretas para mejorar la calidad del código mediante la refactorización. La importancia de esta investigación radica en los siguientes aspectos:

- Mejora de la mantenibilidad del software: Reducir la Complejidad Ciclomática facilita la comprensión del código y agiliza su modificación, permitiendo una evolución más eficiente del sistema.

- Optimización del tiempo de desarrollo: Un código más claro y estructurado disminuye la carga cognitiva de los desarrolladores, reduciendo el tiempo necesario para implementar cambios y corregir errores.
- Reducción de costos a largo plazo: La refactorización preventiva evita la acumulación de deuda técnica, lo que contribuye a minimizar los costos de mantenimiento y soporte del software.
- Base para estudios futuros: Al enfocarse en aplicaciones de gestión comercial, este estudio proporciona un marco de referencia para futuras investigaciones sobre calidad del software en este tipo de sistemas.

A través de este análisis, se busca no solo demostrar el impacto positivo de la refactorización en la Complejidad Ciclomática, sino también proporcionar una guía práctica que permita a otros desarrolladores y equipos de software aplicar estas estrategias en sus propios proyectos.

.

Capítulo 2

Fundamentación teórica

Antecedentes

La gestión de inventarios es un componente crítico en la operación de empresas comerciales, ya que influye directamente en la eficiencia de las ventas, la reducción de costos y la optimización de recursos. En la actualidad, el uso de aplicaciones móviles ha facilitado significativamente este proceso, permitiendo un control más preciso del stock y una mayor accesibilidad a la información en tiempo real.

Según (Quisaguano Collaguazo et al., 2022), el desarrollo de aplicaciones móviles ha crecido de manera exponencial en la última década, impulsado por la necesidad de mejorar la eficiencia en diversas áreas comerciales. Los avances tecnológicos han llevado a la adopción de frameworks modernos para el desarrollo de aplicaciones híbridas, lo que ha permitido reducir los tiempos de codificación y mejorar la portabilidad de los sistemas en diferentes plataformas. Este tipo de soluciones resulta especialmente beneficioso para la gestión de inventarios, donde la disponibilidad y actualización constante de los datos son esenciales para la toma de decisiones empresariales.

Diferentes estudios han demostrado que una gestión eficiente del inventario tiene un impacto directo en el desempeño de las empresas. (Fuentes Romero y Tovar Giraldo, 2019) realizaron un análisis detallado de los problemas asociados con la administración de inventarios, identificando factores como el aumento de costos, el desorden en el almacenamiento y la falta de stock. A través de la implementación de estrategias como el uso de modelos de clasificación (ABC), metodologías de organización (5S) y mejoras

en la política de compras, lograron reducir el exceso de stock en un 69.7% y aumentar las ventas en un 54.7%, lo que demuestra la importancia de optimizar estos procesos mediante soluciones tecnológicas.

El desarrollo de software para la gestión de inventarios ha sido abordado en diversas investigaciones, con el objetivo de mejorar la precisión y eficiencia del control de existencias. En el estudio titulado “Propuesta de diseño de un sistema de gestión de inventarios para Motovalle S.A.S” de (Espinosa Sánchez et al., 2022) destacaron en su estudio la importancia de implementar un sistema de gestión de inventarios basado en un Warehouse Management System (WMS). Su investigación resalta que el uso de estos sistemas permite minimizar errores, mejorar la administración del almacenamiento y garantizar la disponibilidad de productos en tiempo real, aspectos que son fundamentales en entornos comerciales con alta rotación de inventario.

Por su parte, la tesis de (Flores Saca y Condori Champi, 2022) realizaron un estudio centrado en evaluar el impacto de la digitalización en la gestión de inventarios y ventas dentro de una farmacia. Su análisis reveló que la implementación de un sistema web permitió optimizar la actualización del stock y mejorar la eficiencia en los procesos de venta, lo que se tradujo en una mayor precisión en la gestión operativa y una reducción en los tiempos de procesamiento de transacciones.

A pesar de los beneficios que estos sistemas pueden ofrecer, su eficiencia y sostenibilidad dependen en gran medida de la calidad del software en el que están basados. Aplicaciones con código fuente desorganizado, altamente acoplado o con una complejidad excesiva pueden volverse difíciles de mantener y escalar con el tiempo, lo que impacta negativamente en la operación del negocio.

En este contexto, la refactorización del código se convierte en una estrategia clave para garantizar la mantenibilidad de los sistemas de gestión de inventarios. Según (Opdyke, 1992), la refactorización es "el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código, pero mejore su estructura interna". Esta técnica permite mejorar la legibilidad y modularidad del código, facilitando su comprensión y adaptación a nuevas necesidades empresariales sin comprometer su estabilidad.

(Almogahed et al., Categorization Refactoring Techniques based on their Effect on Software Quality Attributes, 2019) realizaron una revisión sistemática de la literatura con el objetivo de analizar cómo las técnicas de refactorización afectan los atributos de calidad del software. De un total de 71 estudios elegibles, identificaron 14 investigaciones que categorizaron técnicas de refactorización según su impacto en la calidad del software. Sin embargo, señalaron que estas categorizaciones presentan limitaciones, ya que abarcan un conjunto reducido de técnicas y atributos de calidad, dificultando la generalización de los resultados. Uno de los principales hallazgos de su estudio fue la existencia de opiniones contradictorias sobre la influencia de la refactorización en la calidad del software, debido a que diferentes técnicas pueden tener efectos positivos o negativos en distintos atributos. Ante esta situación, los autores recomiendan el desarrollo de modelos de referencia y predicción que ayuden a los desarrolladores a seleccionar estrategias de refactorización adecuadas para cada objetivo de calidad.

Por otro lado, (Kaur y Singh, 2019) realizaron un análisis comparativo de estudios en entornos académicos e industriales para evaluar el impacto de la refactorización en la calidad del software. Sus resultados indicaron que, en entornos académicos, la

refactorización tuvo un impacto más positivo que en el ámbito industrial. Además, encontraron que no todos los atributos de calidad mejoran con la refactorización; en algunos casos, métricas como cohesión, complejidad, herencia, propensión a fallos y consumo de energía se vieron afectadas negativamente. También identificaron problemas abiertos en la investigación sobre refactorización, como la falta de validación en contextos industriales y la cobertura limitada de herramientas automatizadas para medir el impacto de estas técnicas.

En una revisión de 20 estudios enfocados en el impacto de la refactorización en entornos industriales, (Almogahed et al., Impact of Software Refactoring on Software Quality in the Industrial Environment: A Review of Empirical Studies, 2018) encontraron que la refactorización tiene efectos positivos en atributos de calidad internos y externos del software. Sin embargo, señalaron que existe una escasez de estudios empíricos que analicen en profundidad las técnicas específicas utilizadas y su relación con la calidad del software en la industria. Su investigación reveló que, si bien la refactorización puede mejorar la calidad del software, la falta de consenso sobre su impacto en distintos atributos y la ausencia de estándares claros dificultan su aplicación en entornos productivos.

Uno de los aspectos clave en la evaluación de la calidad del software es la Complejidad Ciclomática (CC), una métrica introducida por McCabe (1976) que mide el número de caminos lógicos independientes dentro de un código. Esta métrica es utilizada para estimar la dificultad de prueba, mantenimiento y modificación del software. Un código con alta complejidad ciclomática puede volverse difícil de comprender y propenso a errores, lo que impacta negativamente en su mantenibilidad y escalabilidad

En el contexto de la refactorización, estudios han demostrado que la CC puede aumentar o disminuir dependiendo de cómo se apliquen las modificaciones al código. (Cardacci, 2016) analizó el comportamiento de la CC en procesos de refactorización de código estructurado hacia código orientado a objetos, encontrando que la refactorización no garantiza una reducción directa de la complejidad ciclomática, sino que en muchos casos puede generar un desplazamiento de la complejidad hacia otras capas del sistema, como la interfaz de usuario (IU) o la capa de acceso a datos (DAL). Este fenómeno indica que una reducción de la CC en un área específica del código no siempre implica una mejora en la mantenibilidad general del software.

Aunque los estudios previos han analizado diversos atributos de calidad en relación con la refactorización, existe una brecha en la investigación sobre su impacto específico en la Complejidad Ciclomática dentro de aplicaciones de gestión comercial. En este contexto, el presente estudio busca contribuir a la literatura existente mediante una evaluación cuantitativa del impacto de la refactorización en la CC, proporcionando evidencia empírica sobre su efectividad como estrategia para mejorar la estructura del código en aplicaciones móviles.

Bases teóricas

Aplicación móvil

Una aplicación móvil es un sistema adaptado en móviles, una aplicación puede ser muy sencilla o compleja y satisfacen a una demanda, estas aplicaciones se encuentran disponibles en diferentes plataformas de distribución para diferentes sistemas operativos.

Según (Palma Muñoz et al., 2021) menciona que “las aplicaciones móviles en la actualidad son muy aplicadas en la vida cotidiana, por lo que hace la subsistencia más fácil, proporcionan el desarrollo de la tecnología, las oportunidades en diferentes sectores empresariales.”

Control de inventarios

El control de inventarios es una parte crucial en la administración de cualquier negocio, especialmente en aquellos que manejan productos, como es el caso de este estudio, una aplicación de gestión comercial.

Según (Calzado-Mesa, 2022) la gestión de inventarios se ha convertido en uno de los procesos esenciales de las empresas, pues no solo permite tener un mayor control de las mercancías, sino también lograr mejores resultados económicos y mayor rentabilidad. La logística son aquellas actividades que aseguran la correcta planificación y gestión de todas las operaciones que están directamente relacionadas con el flujo de materias primas, productos semielaborados y productos terminados, desde su origen hasta el consumidor final.

Proceso de ventas

Algo que también es importante en una aplicación de gestión comercial como es este el caso es el proceso de ventas. Según (Acosta Véliz et al., 2018) el proceso de ventas, tal como lo define la American Marketing Association, se basa en la acción de asistir o convencer a un posible cliente para que adquiera un producto o servicio, o bien actúe de manera favorable hacia una propuesta comercial. Este proceso, que puede ser tanto personal como impersonal, es esencial en los negocios modernos, ya que requiere la

coordinación de diversas personas y procedimientos para lograr el objetivo común de la venta.

AWS Lambda

En este proyecto, se utilizó AWS Lambda para la ejecución de funciones relacionadas al manejo de los datos del sistema. “AWS Lambda es un servicio informático de AWS que usa un modelo de ejecución sin servidor o Serverless basado en eventos. AWS Lambda es un FaaS (Function as a Service) el cual permite la ejecución de código sin administrar o aprovisionar servidores.” (AWS, Aws lambda, s.f.).

Amazon API Gateway

De igual manera, Amazon API Gateway fue utilizado en el desarrollo del proyecto para el manejo de APIs los cuales ejecutaban las funciones hechas en Lambda. “Amazon API Gateway es un servicio completamente administrado que facilita a los desarrolladores la creación, la publicación, el mantenimiento, el monitoreo y la protección de API a cualquier escala. Las API actúan como la "puerta de entrada" para que las aplicaciones accedan a los datos, la lógica empresarial o la funcionalidad de sus servicios de backend. (AWS, Aws api gateway, s.f.)

EL uso de AWS API Gateway juega un papel fundamental en la optimización y automatización del flujo del sistema, facilita la integración y comunicación entre la aplicación móvil y los servicios backend, en cuanto a AWS Lambda permite ejecutar funciones específicas como la actualización de inventarios o la gestión de pedidos en tiempo real, utilizando estos servicios se crea una infraestructura escalable y de alta disponibilidad.

API

Una API es una especie de puente que funciona como intermediario entre aplicaciones diferentes comunicándose entre sí, la comunicación que estos pueden tener puede ser envió o recuperación de datos. Por ello Amazon en su página de AWS explica la funcionalidad de una API de la siguiente manera “La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor.” (Amazon, s.f.). Este proyecto se basa en el consumo de APIs para un buen funcionamiento entre cliente servidor

Endpoint

Un endpoint en una API es una URL especifica que permite a una aplicación o sistema acceder a datos de una API Web, estos elementos son importantes porque permiten que el servidor y el cliente y servidor interactúen entre sí.

IBM interpreta el funcionamiento de un endpoint de la siguiente manera “Los endpoints funcionan de manera similar a los números de teléfono: al igual que un usuario marca un número de teléfono para comunicarse con una determinada persona o empresa, los clientes de API (el software que realiza una llamada de API) proporcionan una URL de endpoint para comunicarse con un recurso específico. Una URL de endpoint proporciona la ubicación de un recurso en un servidor de API y ayuda a conectar el cliente de API con el recurso que está solicitando” (IBM, 2024)

Editor de código

Un editor de código es una herramienta de software la cual permite a los programadores escribir, editar y guardar código fuente de programas, estos editores permiten crear

aplicaciones de escritorio, móvil o web, por lo que el manejo de uno en este proyecto es de mucha importancia.

Según Aurora en IDbootcamps menciona que “A diferencia de los editores de texto básicos, los editores de código están optimizados para la programación y ofrecen características específicas que ayudan a los desarrolladores a escribir código de manera más eficiente y con menos errores.” (Aurora, 2024).

Lenguaje de programación

La UNAM define un lenguaje de programación de la siguiente manera “En términos generales, un lenguaje de programación es una herramienta que permite desarrollar software o programas para computadora. Los lenguajes de programación son empleados para diseñar e implementar programas encargados de definir y administrar el comportamiento de los dispositivos físicos y lógicos de una computadora. Lo anterior se logra mediante la creación e implementación de algoritmos de precisión que se utilizan como una forma de comunicación humana con la computadora.” (UNAM, s.f.)

Dart

Dart es un lenguaje de programación de código abierto que se utiliza para desarrollar aplicaciones móviles, web, de escritorio y de servidor, es muy conocido por su eficiencia y su velocidad ya que se diseñó con el objetivo de hacer el proceso de desarrollo lo más cómodo y rápido posible para los desarrolladores, este incluye herramientas integradas como su propio gestor de paquetes, varios compiladores, un analizador y un formateador.

Framework

Anteriormente se definió que es un lenguaje programación, sin embargo, en este proyecto se utilizó un framework el cual está basado en un lenguaje de programación. Un framework es un conjunto de herramientas y estructuras que se utilizan para desarrollar y organizar software más fácilmente y de manera eficiente. Sus principales ventajas de su uso es que reducen el tiempo para desarrollar una aplicación al igual que debido a los componentes y herramientas este permite reutilizar código.

Flutter

Para el desarrollo de este proyecto se utilizaron tecnologías como Flutter para el manejo de la parte móvil ya que se requiere el uso de tecnologías avanzadas.

“Flutter, un kit de desarrollo de software de código abierto creado por Google ha ganado popularidad por su capacidad para crear aplicaciones nativas de alta calidad en múltiples plataformas desde un solo código base.” (Developers, 2021). Por lo que el uso de esta tecnología facilita el manejo de un solo código en el cual se encuentra tanto la creación de la vista, así como también el consumo del API el cual obtienen los datos los cuales son mostrados al usuario.

Widgets en flutter

En Flutter, un widget es un componente de la interfaz de usuario de una aplicación. AWS explica que “están diseñados para que los desarrolladores puedan personalizarlos fácilmente. Flutter logra esto a través de un enfoque de composición. Esto significa que la mayoría de los widgets están formados por otros más pequeños y que los más básicos tienen propósitos específicos. Esto permite a los desarrolladores combinar o editar

widgets para crear otros nuevos.” (Amazon, s.f.) lo cual proporciona una abstracción de alto nivel sobre la interfaz de usuario.

En el proceso de este proyecto se manejaron diversos widgets para la creación de componentes o funcionalidades.

Modulo

Un módulo en la programación es una parte de un programa que contiene código relacionado y que realiza una o más tareas, Gavin Wright define que “En software de computadoras, un módulo es una extensión de un programa principal dedicado a una función específica. En programación, un módulo es una sección de código que se agrega como un todo o está diseñada para una fácil reutilización.” (Wright, 2022).

Programación modular

La programación modular o estructurada es una técnica de programación que divide un programa en módulos independientes por lo que la aplicación de esta técnica a una aplicación de gestión comercial facilita el desarrollo. Tom Nolle explica que “La programación estructurada fomenta la división de un programa de aplicación en una jerarquía de módulos o elementos autónomos que, a su vez, pueden contener otros elementos similares. Dentro de cada elemento, el código puede estructurarse aún más utilizando bloques de lógica relacionada diseñados para mejorar la legibilidad y la facilidad de mantenimiento.” (Nolle, 2023).

Código limpio

Clean code o código limpio se le conoce al hecho de escribir código de programación de manera clara y fácil de entender, esto hace que el código sea más sostenible a lo largo del tiempo, y mejora su calidad. El código limpio hace que para los desarrolladores sea más fácil de comprender y proporcionar una base clara y sólida para hacer futuras modificaciones y mejoras. Este término se debe gracias al autor e ingeniero de software (Martin, 2009) en donde explica y fomenta los principios y prácticas de Clean Code.

Entre sus principales principios y fundamentos se encuentran abordados por (QindelGroup, 2023):

- Nombres con sentido: Facilitan la comprensión del código.
- Comentarios útiles: Ayudan a documentar decisiones importantes.
- Funciones bien definidas: Deben ser cortas y realizar una única tarea específica.
- Formato estructurado: Un buen formato mejora la legibilidad.
- Clases con responsabilidad única: Cada clase debe tener un propósito claro.
- Principio DRY (Don't Repeat Yourself): Evita duplicaciones innecesarias.
- Encapsulación adecuada: Los objetos deben ocultar su implementación interna.
- Pruebas limpias (Clean Tests): Los tests deben ser claros y centrados en casos específicos.

La importancia del código limpio radica en su impacto a largo plazo en la mantenibilidad del software. Como lo menciona (Latte et al., 2019), la calidad del código debe ser preservada a lo largo del ciclo de vida del software para evitar su degradación. La adopción de código limpio no solo es un reto técnico, sino también un aspecto cultural

dentro de los equipos de desarrollo, requiriendo un compromiso con estándares de calidad y prácticas ágiles.

Métricas de calidad de código

Las métricas de calidad de código son mediciones que evalúan las características de un sistema de software. Estas métricas pueden estar enfocadas en la estructura, el tamaño, rendimiento y seguridad.

(Microsoft, 2023) define las métricas de código como “un conjunto de medidas de software que proporcionan a los programadores una mejor visión del código que están desarrollando. Al aprovechar las métricas de código, los desarrolladores pueden comprender qué tipos o métodos se deben volver a poner en marcha o probar más exhaustivamente.” Por lo que los equipos de desarrollo pueden identificar posibles riesgos, comprender el estado actual de un proyecto y realizar un seguimiento de progreso durante el desarrollo de software.

Por lo que un código de alta calidad es confiable, fácil de probar, de leer, comprender y trabajar con él a lo largo del tiempo,

Complejidad ciclomática

En este estudio es importante definir la complejidad ciclomática.

La complejidad ciclomática es una métrica que mide la complejidad de un programa o sistema. Este concepto viene de (McCabe, 1976), esta métrica es particularmente útil para identificar áreas propensas a errores y difíciles de probar.

Si la complejidad es alta, por ejemplo, sugiere una base de código con numerosos puntos de decisión, bucles y declaraciones condicionales, lo que indica una mayor complejidad. Esto está directamente relacionado con la mantenibilidad del código; A medida que aumenta el CC, aumenta la dificultad de comprender, modificar y mantener el código. (EN-COM, 2024).

Análisis de código estático

El análisis estático de código es una metodología que examina el código fuente de un programa informático sin necesidad de ejecutarlo, se utiliza para identificar errores, vulnerabilidades y áreas de mejora.

El proceso permite comprender la estructura del código y puede ayudar a garantizar que el código cumpla con los estándares de la industria. El análisis estático se utiliza en ingeniería de software por parte de los equipos de desarrollo de software y control de calidad. (Gills, 2020)

Herramientas de análisis estático

Estas herramientas son programas que ayudan a los desarrolladores a identificar problemas en el código. (EN-COM, 2024) menciona que las herramientas proporcionan una medida de complejidad cuantitativa, lo que permite a los equipos evaluar y priorizar áreas de mejora. Los desarrolladores pueden identificar regiones de alta complejidad, refactorizar el código para mejorar la mantenibilidad y reducir el riesgo de defectos. Además, comprender la complejidad ciclomática ayuda a optimizar las rutas de ejecución, lo que contribuye a un software más eficiente y robusto.

Dart Code Metrics

Para el estudio de este proyecto, manejaré una herramienta de análisis estático desarrollada especialmente para dart y flutter. Esta herramienta aumenta la productividad porque sus principales características son:

- Conjunto de reglas integral.
- Métricas y análisis de código.
- Integración IDE.

DCM ayuda a detectar problemas de forma temprana, reducir el tiempo dedicado a corregir errores, acelerar la revisión de código y reducir la deuda técnica como lo menciona en su documentación (MDC, 2024).

Refactorización

Este estudio tiene como principal tema la refactorización, este es el proceso de modificar un código fuente de un programa sin cambiar su funcionalidad, mejorando la estructura del código haciéndolo más legible, fácil de entender y más simple.

Según (AWS, s.f.) la refactorización reconstruye el mismo código para que sea de mayor calidad o posea más rendimiento. El código debe probarse de forma exhaustiva antes y después de la refactorización con el fin de asegurar que no se introduzcan errores en el proceso de desarrollo.

La refactorización siempre tiene el sencillo y claro propósito de mejorar el código. Con un código más efectivo, puede facilitarse la integración de nuevos elementos sin incurrir en errores nuevos. Además, cuanto más fácil les resulte a los programadores leer el código, más rápido se familiarizarán con él y podrán identificar y evitar los bugs de forma más eficiente. Otro objetivo de la refactorización es mejorar el análisis de errores y la

necesidad de mantenimiento del software. Poner a prueba el código ahorra esfuerzo a los programadores (IONOS, 2020).

Métodos de refactorización

Como se menciona anteriormente, la refactorización es un proceso de modificación de código con el objetivo de optimizar la estructura del código y hacerlo más fácil de entender, sin embargo, también es un proceso multidimensional que se puede aplicar mediante 5 técnicas diferentes (Slingerland, 2021):

- Refactorización rojo-verde
- Método de extracción
- Simplificando métodos
- Método de composición
- Abstracción

(Refacturing Guru, s.f.) agrupa diversas técnicas de refactorización en seis categorías principales. En primer lugar, las técnicas de composición de métodos buscan reducir la longitud de los métodos, eliminar código duplicado y facilitar futuras mejoras; entre ellas se encuentran Extract Method, Inline Method y Extract Variable. En segundo lugar, las técnicas para mover funciones entre objetos permiten reorganizar la funcionalidad entre clases, ocultar detalles de implementación y mejorar la cohesión del código, con técnicas como Move Method, Extract Class y Hide Delegate.

Por otro lado, las técnicas de organización de datos facilitan el manejo de estructuras de datos complejas y mejoran la reutilización de clases, como Change Value to Reference y Self Encapsulate Field. Asimismo, las técnicas de simplificación de expresiones

condicionales ayudan a reducir la complejidad en la toma de decisiones dentro del código mediante estrategias como Replace Conditional with Polymorphism o Introduce Null Object.

Además, las técnicas de simplificación de llamadas a métodos optimizan la interacción entre clases y mejoran la legibilidad, por ejemplo, con Rename Method e Introduce Parameter Object. Finalmente, las técnicas de generalización buscan mejorar la abstracción del código mediante la manipulación de jerarquías de herencia, incluyendo Pull Up Method, Extract Superclass y Extract Interface (Refacturing Guru, s.f.).

Metodología

Una metodología de desarrollo de software es un enfoque en la gestión y desarrollo de proyectos que se entra en la entrega continua y colaborativa de valor. Según (Abuchar Porras, 2023) las metodologías tradicionales para el desarrollo de software surgen aproximadamente en los años sesenta, junto con el modelo del ciclo de vida básico que estaba vigente para ese entonces: análisis, diseño, codificación, implantación y mantenimiento.

Metodología SCRUM

Según el estudio de (Estrada Velasco et al., 2021) la metodología SCRUM promueve un entorno de desarrollo controlado, donde se realizan actividades organizadas y validadas que se adaptan a los requerimientos del proyecto. En caso de que surjan eventualidades, se resuelven en equipos de trabajo que enfrentan problemas de manera efectiva, facilitando avances exitosos en la implementación. Sin embargo, algunos autores como (Sangama Oñate, 2020) en su trabajo de investigación donde hace una comparación

entre metodologías agiles menciona que la metodología SCRUM requiere de mucha definición de tareas y plazos, así como también un equipo que tenga una alta calificación o formación por la experiencia que aportan a los proyectos, aunque al final en sus conclusiones reconoce que SCRUM es una de las metodologías que mejores resultados brindan debido a sus principios y objetivos para obtener software funcional de calidad en el menor tiempo posible.

En este proyecto se manejó la metodología SCRUM y los elementos que conformaron esta metodología son los siguientes a considerar:

Sprint: Son ciclos de trabajo que dividen al proyecto, con duraciones de dos a tres semanas, cada sprint tiene un objetivo y busca generar avances como nuevas características o mejoras, al final de cada sprint se entrega un incremento de producto funcional lo cual facilita la adaptación de cambios en el desarrollo del proyecto

Planificación del Sprint: En esta fase inicial, se establecieron los objetivos del proyecto y se definieron las historias de usuario que guiarían el desarrollo. Se priorizaron las tareas más relevantes para el cliente y se asignaron al equipo de desarrollo.

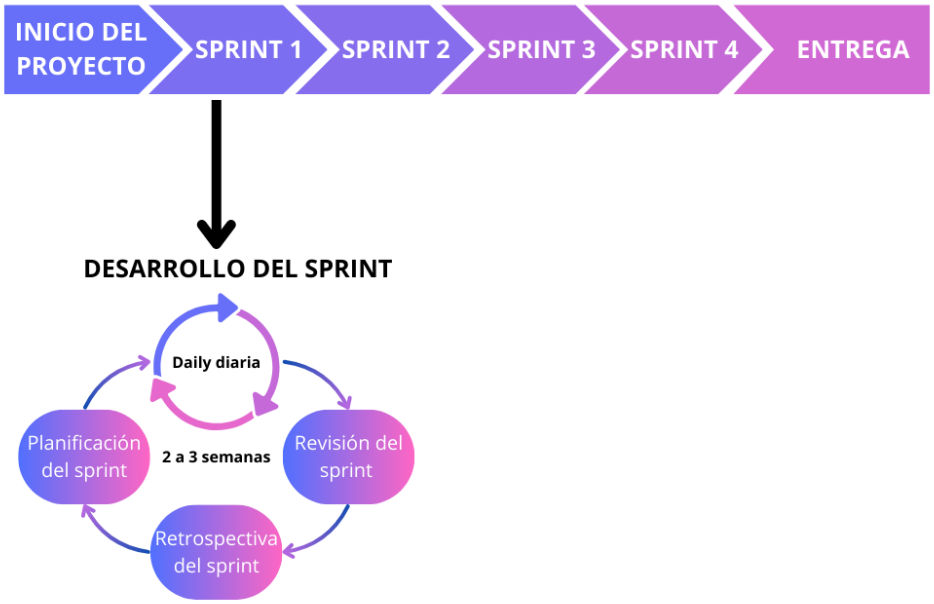
Daily Diaria: El equipo se reúne para revisar los pendientes y sincronizar las asignaciones y ajustar el plan según sea necesario para seguir avanzando.

Revisión del Sprint: Al final de cada Sprint, se lleva a cabo una reunión de revisión donde se presentan los resultados obtenidos de las asignaciones asignadas y se decide si hay ajustes o mejoras que se deban realizar.

Retrospectiva del Sprint: Después de la revisión el equipo reflexiona sobre cómo ha funcionado el sprint para identificar oportunidades de mejora en el proceso y en la colaboración del equipo.

El desarrollo del proyecto se estructuró en etapas organizadas en sprints, como se muestra en la Figura 1.

Figura 1. *Diagrama de etapas SCRUM.*



Nota: Fuente de elaboración propia

Hipótesis

La refactorización de un módulo con alta Complejidad Ciclomática en una aplicación de gestión comercial reduce significativamente la CC, facilitando la comprensión, modificación y escalabilidad del sistema.

Capítulo 3.

Metodología

Diseño de investigación

Este estudio sigue un enfoque cuantitativo, ya que se basa en la recopilación y análisis de métricas de calidad del código antes y después de aplicar estrategias de refactorización. Se adopta un diseño experimental de tipo preexperimental, en el cual se analiza la estructura del código de una aplicación móvil para evaluar cómo la reducción de la Complejidad Ciclomática (CC) impacta en su mantenimiento, escalabilidad y facilidad de modificación.

El experimento consiste en medir la CC en todos los módulos de la aplicación, identificar el módulo con mayor promedio de CC y, dentro de ese módulo, seleccionar los métodos con mayor complejidad. Posteriormente, se aplicarán técnicas de refactorización enfocadas en reducir la complejidad lógica del código y mejorar su organización. Finalmente, se realizará una nueva medición para evaluar si las estrategias aplicadas facilitaron la estructuración del código y su comprensión.

Población

La población de estudio está constituida por el código fuente de una aplicación móvil de gestión comercial desarrollada en Flutter. Esta aplicación cuenta con cuatro módulos principales:

- Inventario: Gestión de productos disponibles, stock y reposición.

- Ventas: Manejo del proceso de ventas, selección de productos y cálculos de costos.
- Órdenes: Administración de pedidos y solicitudes de abastecimiento.
- Configuraciones del negocio: Modificación de datos y parámetros generales del sistema.

Dado que cada módulo contiene múltiples métodos con diferentes niveles de complejidad, la investigación se enfoca en evaluar y refactorizar el módulo con el mayor promedio de Complejidad Ciclomática.

Muestra

Para este estudio, se seleccionará un solo módulo de la aplicación, basándose en los siguientes criterios:

- Alto nivel de Complejidad Ciclomática (CC): Se priorizará el módulo que presente el mayor promedio de CC entre todos los módulos analizados.
- Relevancia dentro del sistema: Se elegirá un módulo que tenga un impacto significativo en la funcionalidad de la aplicación.
- Presencia de métodos altamente complejos: Se identificarán los métodos dentro del módulo seleccionado que presenten los valores más elevados de CC para ser refactorizados.

Este módulo será evaluado antes y después de la refactorización para medir la reducción en la Complejidad Ciclomática.

Instrumentos

Para la recopilación y análisis de datos, se utilizarán las siguientes herramientas:

- Dart Code Metrics: Herramienta de análisis estático en Flutter que permite calcular la Complejidad Ciclomática de los métodos. Para su correcta configuración, se requiere definir reglas específicas en el archivo analysis_options.yaml. En el Anexo 1 se muestra la configuración utilizada en este estudio.
- Editor de Código (VS Code): Para visualizar, modificar y refactorizar el código fuente.
- Hojas de cálculo: Para organizar y comparar los valores de las métricas antes y después de la refactorización.

Las métricas analizadas en este estudio son:

- Complejidad Ciclomática (CC): Mide el número de caminos lógicos en un método, permitiendo evaluar su dificultad de prueba y mantenimiento.

Procedimiento

El estudio se compone de cinco etapas diseñadas para evaluar cómo la reducción de la Complejidad Ciclomática (CC) impacta en la facilidad de modificación y escalabilidad del código.

Actividades

Etapa 1: Análisis inicial del código

1.1 Ejecución de análisis estático

- Se utilizará Dart Code Metrics para obtener automáticamente las métricas de Complejidad Ciclomática (CC) en todos los módulos del sistema.
- Se registrarán los valores iniciales en una tabla para su comparación posterior.

1.2 Identificación del módulo con mayor CC

- Se calculará el promedio de CC de cada módulo con base en los valores obtenidos.
- Se seleccionará el módulo con el mayor promedio de CC para su análisis.

Etapa 2: Análisis del módulo y detección de problemas

2.1 Análisis cualitativo del código.

- Se revisará manualmente el código del módulo seleccionado.
- Se identificarán patrones problemáticos, como código repetitivo o difícil de leer.

2.2 Selección de métodos para refactorización.

- Se analizarán los métodos con mayor CC para determinar cuáles requieren intervención.
- Se evaluarán estrategias de refactorización aplicables a estos métodos.

Etapa 3: Aplicación de estrategias de refactorización

3.1 Refactorización de métodos seleccionados.

- Se aplicará una estrategia de refactorización basada en el análisis realizado en la etapa anterior.
- Se dividirán métodos extensos en funciones más pequeñas cuando sea necesario.
- Se optimizarán estructuras de control complejas para mejorar la legibilidad.

3.2 Reestructuración del código.

- Se mejorará la organización y modularización del código en función de la refactorización aplicada.

Etapa 4: Medición posterior a la refactorización

4.1 Nueva ejecución de análisis estático.

- Se volverá a ejecutar Dart Code Metrics para obtener las métricas después de la refactorización.
- Se registrarán y compararán los valores de CC obtenidos antes y después de la refactorización.

Etapa 5: Análisis de resultados

5.1 Evaluación del impacto de la refactorización

- Se determinará el cambio en la Complejidad Ciclomática (CC) antes y después de la refactorización.
- Se analizará cómo la refactorización afectó la distribución de la CC en los métodos del módulo seleccionado.
- Se compararán los valores de CC antes y después de la refactorización en dos niveles:
    - Funcionalidad de selección de productos dentro del módulo de Ventas.
    - Módulo de Ventas en general, considerando todas sus funcionalidades.
- Se identificará si algunos métodos mantienen valores elevados de CC tras la refactorización y se analizarán los factores que contribuyen a este comportamiento, tales como condiciones anidadas, flujos de ejecución múltiples o dependencias en estructuras de datos.

5.2 Revisión de hallazgos

- Se discutirá cómo la refactorización redistribuyó la CC en los métodos generados.
- Se analizará si la optimización redujo la CC en los métodos críticos o si parte de la lógica compleja se trasladó a métodos auxiliares en lugar de ser completamente eliminada.

Capítulo 4

Resultados

Una métrica ampliamente utilizada para medir la dificultad de un código es la Complejidad Ciclomática (CC), la cual cuantifica la cantidad de caminos lógicos dentro de un módulo o función. Valores elevados de CC pueden indicar un código difícil de entender, modificar y probar, lo que impacta directamente en su mantenibilidad y escalabilidad.

En este capítulo, se presentan los resultados obtenidos en el análisis de CC en los módulos de la aplicación, antes y después de la refactorización. Para ello, se utilizó Dart Code Metrics, una herramienta de análisis estático que permitió calcular automáticamente la CC de los métodos evaluados.

Siguiendo la metodología descrita en el Capítulo 3, se presentan los resultados obtenidos en cada una de las etapas:

4.1 Medición inicial de la Complejidad Ciclomática

El propósito de esta sección es presentar los resultados obtenidos tras la medición inicial de la Complejidad Ciclomática (CC) en los módulos de la aplicación. Para ello, se utilizó Dart Code Metrics, la cual permitió obtener automáticamente los valores de CC en los módulos de Inventario, Ventas, Pedidos y Configuración.

El objetivo de esta fase fue identificar el módulo con el mayor promedio de CC y establecerlo como el sujeto de estudio para el proceso de refactorización.

En la Tabla 1 se presentan los valores iniciales de CC en cada módulo de la aplicación:

Tabla 1. *Métricas iniciales de la Complejidad Ciclomática por módulo*

Módulo	CC
Inventario	3.13
Ordenes	3.41
Ventas	6.41
Configuraciones	2.90

Nota: Fuente de elaboración propia

Como se observa en la Tabla 1, el módulo de Ventas presentó el mayor promedio de CC = 6.41. Debido a esto, se seleccionó como el módulo de estudio para un análisis más detallado.

4.2 Identificación de los métodos con mayor complejidad

Para determinar qué métodos dentro del módulo de Ventas requerían refactorización, se estableció un umbral de CC ≥ 10. Con base en este criterio, se analizaron los métodos dentro de las dos funcionalidades principales de este módulo:

1. Selección de productos
2. Resumen y pago de productos

Las Tablas 2 y 3 muestran los métodos evaluados en cada funcionalidad, identificando aquellos que superan el umbral de CC ≥ 10.

Tabla 2. *Complejidad Ciclomática en la funcionalidad de selección de productos.*

Método	CC

SelectProductsToSaleView	1.00
createState	1.00
initState	1.00
dispose	1.00
initFuture	1.00
retriveAllSubProducts	6.00
retrieveSaleProducts	13.00
saveData	2.00
logicToSelectedCardDefault	4.00
logicToselectedCardKg	18.00
findMainProduct	1.00
changeCheckBoxValue	2.00
chipCategories	5.00
cardProduct	15.00
cardKgProduct	60.00
gramsTextEdit	6.00
build	30.00
Promedios	**9.82**

Nota: Fuente de elaboración propia

Tabla 3. *Complejidad Ciclomática en la funcionalidad de resumen y pago de productos.*

Método	CC
SummarySaleView	1.00
createState	1.00
initState	1.00
saveProducts	6.00
redirectToPayment	6.00
physicalCardPayment	2.00
prepareBodyTransaction	4.00
llenarLista	1.00

getPaymentTypeIcon	4.00
getOUM	4.00
getStyle	1.00
showDialogMessage	1.00
build	3.00
showButtonSheet	7.00
Promedios	**3.00**

Nota: Fuente de elaboración propia

Tras este análisis, se identificaron los métodos con CC ≥ 10 dentro de la funcionalidad de selección de productos (Tabla 2), los cuales presentan una alta complejidad y requieren refactorización. La Tabla 4 agrupa los métodos seleccionados para ser optimizados.

Tabla 4. *Métodos con alta Complejidad Ciclomática seleccionados para refactorización.*

Método	CC
retrieveSaleProducts	13.00
logicToselectedCardKg	18.00
cardProduct	15.00
cardKgProduct	60.00
build	30.00

Nota: Fuente de elaboración propia

4.3 Aplicación de estrategias de refactorización

Con el objetivo de mejorar la estructura del código y reducir la CC de los métodos seleccionados, se aplicó la técnica Extract Method, la cual consiste en dividir métodos

extensos en funciones más pequeñas y especializadas. Esta técnica permite mejorar la legibilidad del código y facilita su mantenimiento futuro.

Tras la aplicación de la refactorización, se observó un incremento en el número de métodos dentro del módulo, ya que la lógica de los métodos originales se distribuyó en varias funciones más específicas. La Tabla 5 muestra la nueva distribución de métodos dentro de las clases afectadas por la refactorización.

Tabla 5. *Nueva distribución de métodos tras la refactorización*

Funcionalidad/Clase	Métodos Antes	Métodos Después
Selección de productos	17	63
Resumen de venta	21	21

Nota: Fuente de elaboración propia

4.4 Análisis después de la refactorización

Para evaluar el impacto de la refactorización, se realizó una comparación entre los valores de CC antes y después del proceso. La Tabla 6 resume estos resultados.

Tabla 6. *Comparación de Complejidad Ciclomática antes y después de la refactorización.*

Método original	CC Antes	Métodos generados	CC después
retrieveSaleProducts	13	retrieveSaleProducts	3
		_fetchAndProcessProducts	1
		_populateProducts	2
		_addCategoryIfNotExists	2
		_addProduct	1

		_recoverPreviousSelection	3
		_restoreSavedProduct	3
		_calculateSaleAmount	4
		_handleError	1
logicToselectedCardKg	18	logicToselectedCardKg	2
		_deselectProduct	2
		_resetNonMeatProduct	3
		_resetMeatProduct	3
		_selectProduct	2
		_selectNonMeatProduct	6
		_selectMeatProduct	6
cardProduct	15	cardProduct	1
		_handleProductTap	4
		_saveProductToDatabase	4
		_showEditProductScreen	1
		_buildProductCard	1
		_buildProductInfo	2
cardKgProduct	60	cardKgProduct	1
		_handleProductTapKg	4
		_handleProductStockAndPrice	4
		_buildProductCardKg	1
		_buildProductImage	1
		_buildProductDetails	2
		_buildProductName	1
		_buildProductPrice	2
		_buildQuantityControls	1
		_buildKgAndGramsButtons	1
		_buildKgButton	4
		_handleKgButtonTap	7
		_buildGramsButton	4
		_handleGramsButtonTap	6

		_buildQuantityAdjustmentButtons	6
		_handleDecreaseQuantity	17
		_handleIncreaseQuantity	11
build	30	_build	1
		_handleWillPop	2
		_buildAppBar	1
		_buildProductListView	7
		_buildProductList	4
		_buildEmptyState	1
		_buildErrorState	1
		_buildBottomNavigationBar	2
		_handleCheckout	8
		_updateProductsAfterCheckout	3
		_calculateProductPrice	3
		_restoreProductStock	4

Nota: Fuente de elaboración propia

Como se observa, la fragmentación de los métodos en funciones más pequeñas disminuyó la CC individual, aunque el número total de métodos incrementó. Este cambio facilita la comprensión del código y permite que cada método tenga una única responsabilidad, lo que mejora la mantenibilidad y escalabilidad del sistema.

4.5 Evaluación del impacto de la refactorización

Para evaluar el impacto general de la refactorización, se compararon los promedios de Complejidad Ciclomática (CC) antes y después de la refactorización en dos niveles:

1. Funcionalidad de selección de productos, donde la CC promedio pasó de 9.82 a 3.14, lo que representa una reducción significativa en la complejidad lógica de los métodos.
2. Módulo de Ventas en general, considerando todas sus funcionalidades, donde la CC promedio disminuyó de 6.41 a 3.07, confirmando que la refactorización contribuyó a mejorar la estructura y mantenibilidad del código.

Además, en la Tabla 7 se presentan los valores individuales de CC en los métodos generados tras la refactorización, lo que permite observar cómo se redistribuyó la complejidad.

Tabla 7. *Distribución de la Complejidad Ciclomática tras la refactorización en la funcionalidad de selección de productos.*

Método	CC
retrieveSaleProducts	3
_fetchAndProcessProducts	1
_populateProducts	2
_addCategoryIfNotExists	2
_addProduct	1
_recoverPreviousSelection	3
_restoreSavedProduct	3
_calculateSaleAmount	4
_handleError	1
saveData	2
logicToSelectedCardDefault	4
logicToselectedCardKg	2
_deselectProduct	2
_resetNonMeatProduct	3

_resetMeatProduct	3
_selectProduct	2
_selectNonMeatProduct	6
_selectMeatProduct	6
findMainProduct	1
changeCheckBoxValue	2
chipCategories	5
cardProduct	1
_handleProductTap	4
_saveProductToDatabase	4
_showEditProductScreen	1
_buildProductCard	1
_buildProductInfo	2
cardKgProduct	1
_handleProductTapKg	4
_handleProductStockAndPrice	4
_buildProductCardKg	1
_buildProductImage	1
_buildProductDetails	2
_buildProductName	1
_buildProductPrice	2
_buildQuantityControls	1
_buildKgAndGramsButtons	1
_handleKgButtonTap	7
_handleGramsButtonTap	6
_buildProductListView	7
_handleCheckout	8
_restoreProductStock	4

Nota: Fuente de elaboración propia

A pesar de la reducción general en la CC, algunos métodos presentaron valores superiores al umbral de CC ≥ 10, como _handleDecreaseQuantity (CC = 17) y _handleIncreaseQuantity (CC = 11), respectivamente, lo que supera el umbral establecido en este estudio.

Este comportamiento se debe a varios factores estructurales en el código:

1. Condiciones anidadas: Ambos métodos contienen múltiples estructuras condicionales para validar diferentes escenarios, lo que aumenta el número de caminos lógicos y, por ende, su CC.
2. Manejo de múltiples flujos de ejecución: La lógica de selección de productos y actualización de cantidades requiere considerar distintas configuraciones de venta (por peso o unidad), lo que introduce bifurcaciones adicionales en el código.
3. Dependencias en otras funciones y estructuras de datos: La actualización de algunas variables como el amountSale, y la validación de regex.hasMatch(product.itemGroup) requieren interacción con varias variables y objetos, lo que incrementa la complejidad de ejecución.

En general, la refactorización redistribuyó la CC dentro del módulo, reduciendo la complejidad en los métodos originalmente seleccionados para optimización. Sin embargo, en algunos casos, la lógica compleja fue trasladada a métodos auxiliares en lugar de ser completamente eliminada.

Conclusión

El presente estudio evaluó el impacto de la refactorización en la Complejidad Ciclomática (CC) dentro de una aplicación de gestión comercial, demostrando que la aplicación de la técnica Extract Method permitió una reducción significativa en la complejidad de los métodos analizados. La CC promedio en la funcionalidad de selección de productos disminuyó de 9.82 a 3.14, mientras que el promedio del módulo de Ventas en general pasó de 6.41 a 3.07, lo que facilitó la modularización del código y su comprensión.

Estos resultados se alinean con estudios previos sobre el impacto de la refactorización en la calidad del software. Investigaciones previas han señalado diferencias en la efectividad de la refactorización entre entornos académicos e industriales, indicando que en el entorno industrial los beneficios pueden ser menos evidentes debido a la complejidad inherente de los sistemas. Sin embargo, este estudio, aplicado en un entorno industrial real, mostró que la refactorización no solo redujo significativamente la CC, sino que también mejoró la organización del código, lo que sugiere que puede ser una estrategia efectiva para optimizar la calidad del software en aplicaciones comerciales.

A pesar de la reducción general en la CC, se identificaron dos métodos nuevos, _handleDecreaseQuantity y _handleIncreaseQuantity, que mantienen valores superiores al umbral de 10 definido en el estudio. Este resultado concuerda con el estudio de (Cardacci, 2016), quien señala que la refactorización no siempre reduce la complejidad de forma uniforme, sino que puede desplazarla a otros métodos o capas del sistema. Sin embargo, a diferencia del código original, ahora la lógica está mejor segmentada y modularizada, lo que facilita futuras optimizaciones.

Además, este estudio contribuye a la brecha existente en la literatura sobre refactorización en entornos industriales, proporcionando evidencia empírica sobre su impacto en una aplicación real. Se demostró que la aplicación de estrategias adecuadas puede mejorar la calidad, mantenibilidad y escalabilidad del código, facilitando la evolución del software.

Finalmente, un aspecto clave de este trabajo es su aplicabilidad a otros sistemas de gestión comercial con características similares. Dado que este tipo de software enfrenta desafíos relacionados con la escalabilidad y mantenibilidad del código, los resultados obtenidos pueden servir como referencia para desarrolladores y empresas que buscan mejorar la calidad estructural de sus aplicaciones mediante refactorización.

Recomendaciones

Con base en los resultados obtenidos, se sugieren las siguientes recomendaciones para futuras refactorizaciones y mejoras en la calidad del código:

- Monitoreo continuo de la Complejidad Ciclomática: Aunque la CC se redujo significativamente, algunos métodos aún presentan valores elevados, como _handleDecreaseQuantity (CC=17) y _handleIncreaseQuantity (CC=11). Se recomienda seguir optimizando estos métodos mediante estrategias adicionales, como la reducción de la profundidad de anidamiento, el uso de estructuras de datos más eficientes o la aplicación de patrones de diseño como Strategy o Factory para simplificar la lógica condicional.
- Balance entre modularidad y eficiencia: La segmentación del código mediante Extract Method facilitó la comprensión y mantenibilidad del sistema. Sin embargo,

es fundamental evaluar el impacto de la refactorización en el rendimiento del software, asegurando que la modularización no afecte el uso de memoria y el tiempo de ejecución.

- Uso de estándares de calidad del código: Implementar métricas de calidad en el flujo de trabajo permitirá detectar posibles problemas antes de que el código se vuelva difícil de mantener. Herramientas como Dart Code Metrics pueden integrarse en los procesos de desarrollo para monitorear automáticamente la CC y otras métricas relevantes.
- Extensión del estudio a otros módulos: Este trabajo se enfocó en el módulo de Ventas, pero aplicar la misma metodología a otros módulos permitiría validar si los beneficios de la refactorización son consistentes en diferentes áreas del sistema.
- Consideración de métricas adicionales: Aunque este estudio se centró en la CC, otras métricas pueden aportar información valiosa sobre la mantenibilidad del software. Se recomienda analizar:
    - Índice de Mantenibilidad (IM): Permite evaluar el esfuerzo requerido para modificar y entender el código a lo largo del tiempo.
    - Acoplamiento y cohesión: Indicadores clave para evaluar qué tan bien estructurado está el código en términos de dependencias entre módulos.
    - Halstead Volume (HV): Puede ser útil para medir la complejidad algorítmica del código y determinar oportunidades adicionales de optimización.

Con estas recomendaciones, se busca optimizar la calidad del código en sistemas similares, asegurando su mantenibilidad, escalabilidad y eficiencia a largo plazo.

Referencias Bibliográficas

Abuchar Porras, A. (2023). *Metodologías ágiles para el desarrollo de software.* Editorial Universidad Distrital Francisco José de Caldas.

Acosta Véliz, M., Salas Narváez, L., Jiménez Cercado, M. E., y Guerra Tejada, A. M. (02 de 2018). *La administración de ventas. Conceptos Clave en el Siglo XXI.* Dialnet: https://3ciencias.com/wp-content/uploads/2018/02/La-administracion-de-ventas.pdf

Almogahed, A., Omar, M., y Zakaria, N. H. (julio de 2018). *Impact of Software Refactoring on Software Quality in the Industrial Environment: A Review of Empirical Studies.* Universiti Utara Malaysia, Malaysia: https://d1wqtxts1xzle7.cloudfront.net/78858663/Impact_of_Software_Refactoring_on_Software_Quality_in_the_Industrial_Environment_A_Review_of_Empirical_Studies-libre.pdf?1642306001=&response-content-disposition=inline%3B+filename%3DImpact_of_Software_Refact

Almogahed, A., Omar, M., y Zakaria, N. H. (01 de junio de 2019). *Categorization Refactoring Techniques based on their Effect on Software Quality Attributes.* Revista Internacional de Tecnología InnovadoraTecnología e Ingeniería Exploratoria (IJITEE): https://www.researchgate.net/profile/Abdullah-Almogahed/publication/334733334_Categorization_Refactoring_Techniques_based_on_their_Effect_on_Software_Quality_Attributes/links/5d3e0df9a6fdcc370a6948f8/Categorization-Refactoring-Techniques-based-on-their-Ef

Amazon. (s.f.). *¿Qué es Flutter?* aws: https://aws.amazon.com/es/what-is/flutter/

Amazon. (s.f.). *¿Qué es una interfaz de programación de aplicaciones (API)?* aws: https://aws.amazon.com/es/what-is/api/

Aurora. (14 de junio de 2024). *¿Qué es un editor de código?* ID boot camps: https://iddigitalschool.com/bootcamps/que-es-un-editor-de-codigo/

AWS. (s.f.). *¿En qué consiste la calidad del código?* AWS: https://aws.amazon.com/es/what-is/code-quality/

AWS. (s.f.). *Aws api gateway*. https://aws.amazon.com/apigateway/

AWS. (s.f.). *Aws lambda*. https://aws.amazon.com/es/lambda/

Calzado-Mesa, Z. (2022). *Proyecto de codificación industrial en la gestión de inventarios.* Ciencias Holguín: https://www.redalyc.org/articulo.oa?id=181572159007

Cardacci, D. G. (noviembre de 2016). *Comportamiento de la complejidad ciclomática en la refactorización de código estructurado a código orientado a objetos.* Congreso Nacional de Ingeniería en Informática / Sistemas de información: https://bibliotecas.ucasal.edu.ar/opac_css/index.php?lvl=cmspage&pageid=24&id_notice=61395

Developers, G. (2021). *Flutter: Beautiful Native Apps in Record Time*. https://flutter.dev

EN-COM. (20 de febrero de 2024). *Los conceptos básicos de la complejidad ciclomática y por qué todo programador debería conocerlos*. IN-COM: https://www.in-com.com/es/blog/cyclomatic-

complexity/#:~:text=La%20complejidad%20ciclom%C3%A1tica%20es%20una%20m%C3%A9trica%20de,muy%20%C3%BAtil%20para%20la%20ingenier%C3%ADa%20de%20software.

Espinosa Sánchez, J. A., Rache Fonseca, J., y Pedraza Rodríguez, L. A. (2022). *Propuesta de diseño de un sistema de gestión de inventarios para Motovalle S.A.S.* Universidad ECCI de Colombia: https://repositorio.ecci.edu.co/handle/001/3008

Estrada Velasco, M. V., Núñez Villacis, J. A., Saltos Chávez, P. R., y Cunuhay Cuchipe, W. C. (09 de enero de 2021). *Revisión Sistemática de la Metodología Scrum para el Desarrollo de Software.* Dominio de las Ciencias: https://dialnet.unirioja.es/servlet/articulo?codigo=8384028

Flores Saca, P. N., y Condori Champi, I. (2022). *Sistema web para la gestión de inventarios y ventas de la Farmacia Multiservicios Santa Ana.* Universidad Tecnológica de los Andes: https://repositorio.utea.edu.pe/server/api/core/bitstreams/045fdeef-d95e-4ce9-b354-ba157a956f6b/content

Fuentes Romero, B. C., y Tovar Giraldo, J. M. (2019). *DISEÑO DE UN SISTEMA DE GESTIÓN DE INVENTARIO PARA MINIMIZAR COSTOS EN UNA EMPRESA COMERCIALIZADORA DE REPUESTOS AUTOMOTRIZ.* Universidad San Ignacio de Loyola: https://repositorio.usil.edu.pe/server/api/core/bitstreams/823dd421-f689-428b-ad6a-a009ba5464e0/content

Gills, S. A. (julio de 2020). *static analysis (static code analysis)*. TechTarget: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis

IBM. (05 de agosto de 2024). *¿Qué es un endpoint de API?* IBM: https://www.ibm.com/mx-es/topics/api-endpoint

IONOS. (29 de septiembre de 2020). *Refactorización: cómo mejorar el código fuente*. IONOS: https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/que-es-la-refactorizacion/

Kaur, S., y Singh, P. (noviembre de 2019). *How does object-oriented code refactoring influence software quality? Research landscape and challenges.* Journal of Systems and Software: https://doi.org/10.1016/j.jss.2019.110394

Latte , B., Henning, S., y Wojcieszak, M. (2019). *Clean Code: On the Use of Practices and Tools to.* https://oceanrep.geomar.de/id/eprint/45829/1/ELMS2019CleanCode.pdf

Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education.

McCabe, T. J. (1976). A Complexity Measure. En *IEEE Transactions on Software Engineering* (pp. 308-320). IEEE.

MDC. (30 de julio de 2024). *¿Por qué DCM?* DCM: https://dcm.dev/docs/introduction/why-dcm/

Microsoft. (29 de noviembre de 2023). *Valores de métrica de código.* Microsoft: https://learn.microsoft.com/es-es/visualstudio/code-quality/code-metrics-values?view=vs-2022

Nolle, T. (marzo de 2023). *Programación estructurada*. TechTarget: https://www-techtarget-com.translate.goog/searchsoftwarequality/definition/structured-programming-modular-programming?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq

Opdyke, W. F. (1992). Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign.

Palma Muñoz, K. A., Garzón García, J. J., Delgado Zambrano, J. D., Zambrano Alcívar, K. G., Párraga Zambrano, L. A., y Mendoza Navarrete, M. L. (08 de 02 de 2021). *EL IMPACTO DE LAS APLICACIONES MÓVILES, ORIENTADO A LAS MIPYMES DE LA CIUDAD DE CHONE.* ULEAM Bahía Magazine (UBM) E-ISSN 2600-6006: https://revistas.uleam.edu.ec/index.php/uleam_bahia_magazine/article/view/84

QindelGroup. (19 de junio de 2023). *¿Qué es Clean Code o código limpio?* Qindel: https://www.qindel.com/que-es-clean-code-o-codigo-limpio/

Quisaguano Collaguazo, L. R., Pallasco Venegas, M. S., Andaluz Guerrero, A. A., Martínez Freire, M. N., y Corrales Beltrán, S. H. (21 de 09 de 2022). *Desarrollo Hibrido con Flutter.* Ciencia Latina Revista Científica Multidisciplinar: https://doi.org/10.37811/cl_rcm.v6i4.2959

Refacturing Guru. (s.f.). *Refactoring Techniques*. Refacturing Guru: https://refactoring.guru/refactoring/techniques

Sangama Oñate, A. F. (20 de diciembre de 2020). *Metodologías ágiles Scrum, XP, SLeSS, Scrumban, HME, Mobile-D y MASAN empleadas en la industria de dispositivos móviles: Un contraste en favor de la industria del desarrollo móvil.* Universidad Peruana Unión: https://repositorio.upeu.edu.pe/server/api/core/bitstreams/4810a4df-9d98-4a1e-a065-0fb0f7fde9e0/content

Slingerland, C. (08 de diciembre de 2021). *5 técnicas de refactorización que puedes utilizar para mejorar tu software*. CloudZero: https://www.cloudzero.com/blog/refactoring-techniques/

UNAM. (s.f.). *Lenguajes de Programación*. Unidad de Apoyo para el Aprendizaje: https://repositorio-uapa.cuaieed.unam.mx/repositorio/moodle/pluginfile.php/2655/mod_resource/content/1/UAPA-Lenguajes-Programacion/index.html

Wright, G. (junio de 2022). *módulo*. TechTarget: https://www-techtarget-com.translate.goog/whatis/definition/module?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq#:~:text=In%20computer%20software%2C%20a%20module,is%20designed%20for%20easy%20reusability.

Anexos

Anexo 1. Configuración de Dart Code Metrics en analysis_options.yaml.