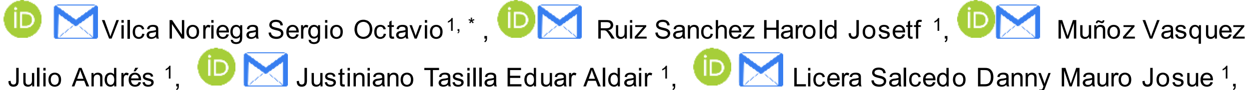
ISSN:3119-7906, Vol.1Núm.1(2026): Enero-Marzo
https://doi.org/10.17268/tech1.1.2026.01

Diseño e Implementación de un Sistema de Gestión Comercial para las ventas de motos basado en los Patrones MVC, Singleton y Strategy

Design and Implementation of a Sales Management System for Motorcycle Sales Based on the MVC, Singleton, and Strategy Patterns


Vilca Noriega Sergio Octavio [1, *], Ruiz Sanchez Harold Josetf [1], Muñoz Vasquez Julio Andrés [1], Justiniano Tasilla Eduar Aldair [1], Licera Salcedo Danny Mauro Josue [1],

[1]Escuela de Ingeniería Informática, Universidad Nacional de Trujillo, La Libertad, Perú

*Autor correspondiente: sovilcano@unitru.edu.pe (S-VilcaNoriega)

RESUMEN

Este artículo describe el desarrollo de una solución ante los desafíos y problemas que enfrentan empresas concesionarias de motos en Chepén y otros lugares, donde estos locales sin un sistema tienden a errar o perder información importante durante toda la gestión comercial. Para resolver esta ineficiencia, tanto operativamente como del desarrollo del mismo, se desarrolla un sistema basado en arquitectura Modelo-Vista-Controlador (MVC), que ayuda a dividir y organizar la interfaz de usuario y lógica de negocio; asimismo se usaron patrones de diseño de comportamiento, siendo el Strategy, que permitirá que las interfaces y procesos se adapten dinámicamente según el rol del usuario. También se implementa el patrón creacional, el cual sería el Singleton, que ayuda y facilita en la gestión central de la instancia de la conexión de la base de datos, ahorrando recursos. El alcance se centra en la aplicación de patrones para asegurar la calidad del software.

Como resultados de la implementación del Sistema de Gestión Comercial de motos, se obtuvo una arquitectura robusta que facilita el mantenimiento y la escalabilidad, y agiliza las operaciones comerciales, además de un desacoplamiento del sistema. Se logró optimizar el desarrollo, con una gestión efectiva de datos. Se concluye así que la implementación de los patrones MVC, Singleton y Strategy es fundamental para el entorno de desarrollo profesional de software comercial, elevando así los estándares de calidad del mismo. Su implementación establece una base tecnológica necesaria para el futuro de las empresas, especialmente en el sector automotriz.

Palabras clave: Patrones de diseño, Arquitectura MVC, Sistema de gestión, Calidad de software.

ABSTRACT

This article describes the development of a solution to the challenges and problems faced by motorcycle dealerships in Chepén and other locations, where establishments lacking an automated system tend to make

errors or lose critical information throughout the commercial management process. To resolve this inefficiency, both operationally and in terms of development, a system was developed based on the Model-View-Controller (MVC) architecture, which facilitates the division and organization of the user interface and business logic. Likewise, behavioral design patterns were used, specifically Strategy, which allows interfaces and processes to adapt dynamically according to the user's role. The Singleton creational pattern is also implemented to facilitate the centralized management of the database connection instance, thereby saving resources. The scope focuses on the application of these patterns to ensure software quality.

As results of the implementation of the Motorcycle Commercial Management System, a robust architecture was obtained that facilitates maintenance and scalability while streamlining commercial operations through system decoupling. Development was optimized through effective data management. It is concluded that the implementation of MVC, Singleton, and Strategy patterns is fundamental for a professional commercial software development environment, thereby raising its quality standards. Its implementation establishes a necessary technological foundation for the future of companies, especially in the automotive sector.

Keywords: *Design patterns, MVC architecture, Management system, Software quality.*

1. INTRODUCCIÓN

En el mercado actual de gestión comercial de vehículos menores (motos), la rapidez y orden a través de automatizaciones son factores criticos y importantes para el éxito, muchas concesionarias aún se gestionan de manera manual a papel o registros básicos digitales, generando dificultades al momento de controlar stock, procesar ventas, y otras operaciones que dependen de diferentes roles, limitando la capacidad de crecimiento del negocio y poniendo en riesgo los datos financieros.

El problema identificado es el no tener un sistema que logre centralizar las operaciones manejando los estándares de la ingeniería de software, puesto que el manejo manual y poco controlado comporta la posibilidad de errores humanos en los procesos, los cuales se encuentran fuertemente acoplados entre sí, ya que, al presentar una modificación, la misma puede actualizar inmediatamente el registro de ventas existente. El problema plantea la creación de un análisis que logre ayudarnos al desarrollo de un software simple pero robusto y, lo que significa el mismo, entonces es importante el uso necesario de patrones de diseño y una arquitectura, para que el software finalizado y en producción sea fácil de gestionar y mantener, seguro, y que pueda adaptarse a las reglas de negocios que han de cambiar en el tiempo en el sector automotriz.

1.1 Justificación

La implementación de este sistema se justifica por la necesidad técnica de optimizar la gestión de datos en el sector automotriz minorista, donde la complejidad de las ventas a crédito y el control de inventarios demanda soluciones de alta fidelidad. Según [1] I. Sommerville, la ingeniería de software profesional debe garantizar que los sistemas sean mantenibles y confiables ante entornos de negocio en constante evolución.

Desde una perspectiva de gestión de procesos, [2] K. E. Kendall y J. E. Kendall sostienen que la automatización reduce drásticamente la redundancia informativa y los errores humanos. Además, como se evidencia en la investigación de [3] L. Ticona sobre sistemas de escritorio, la elección de tecnologías modernas como Python permite un desarrollo ágil sin sacrificar la robustez necesaria para el manejo de transacciones financieras. Este proyecto no solo resuelve una necesidad operativa, sino que establece una infraestructura tecnológica que permite la escalabilidad del negocio, asegurando la integridad de la información contable según los estándares de calidad de software propuestos por [4] R. S. Pressman.

El objetivo principal de esta investigación es desarrollar e implementar un sistema integral de gestión comercial para concesionarias de motocicletas. Para ello, se emplea una arquitectura MVC que garantiza el desacoplamiento entre la interfaz y la lógica de negocio, facilitando el mantenimiento profesional del software según [5] C. Larman. Esta estructura se complementa con el patrón Singleton para centralizar la persistencia de datos en una base de datos relacional, asegurando

la integridad y el uso eficiente de los recursos del sistema conforme a [6] M. Fowler. Finalmente, la automatización de los flujos comerciales se optimiza mediante el patrón Strategy, permitiendo que las interfaces y procesos se adapten dinámicamente según la lógica de negocio descrita por [7] E. Freeman.

1.3 Fundamento Teórico

El diseño del sistema se sustenta en una arquitectura de capas y patrones de diseño definidos como soluciones estandarizadas a problemas recurrentes en la ingeniería de software por [8] E. Gamma et al.

A. Arquitectura Modelo-Vista-Controlador (MVC)

Es el patrón de arquitectura global que rige la estructura del sistema. Según [5] C. Larman, el MVC facilita la separación de preocupaciones (*separation of concerns*), permitiendo que el desarrollo de la lógica de negocio progrese de forma independiente a la interfaz de usuario. En el contexto de aplicaciones de escritorio contemporáneas, [9] A. Reyes destaca que el MVC es fundamental para organizar sistemas que manejan grandes volúmenes de datos y múltiples vistas, asegurando que los cambios en la base de datos no afecten directamente la representación visual. Véase la Figura 1, donde se detalla esto.

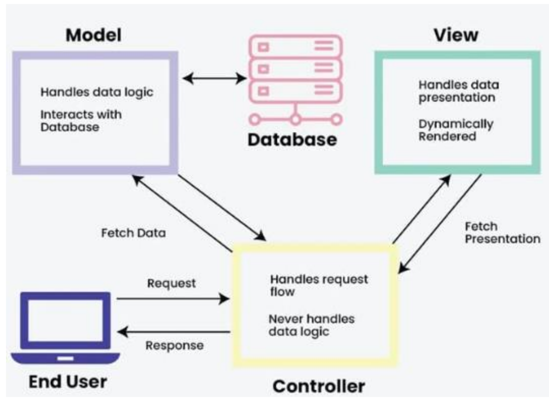
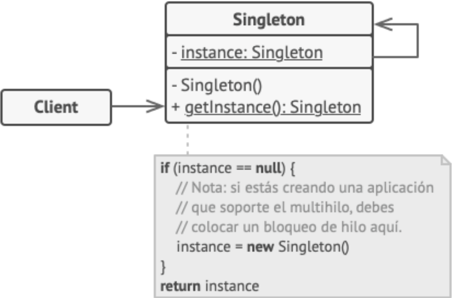


Figura 1, Diagrama ilustrativo de la arquitectura modelo-vista-controlador (MVC).

B. Patrón de Diseño Singleton

Este patrón de tipo creacional garantiza que una clase posea una única instancia y proporcione un punto de acceso global a ella, de acuerdo con **[10] A. Shvets**. Su importancia técnica radica en el control de recursos compartidos. En este sistema, se aplica específicamente a la conexión con MySQL. Como explican **[11] E. Freeman et al.**, la apertura indiscriminada de conexiones puede degradar el rendimiento del servidor. Véase la Figura 2, el patrón Singleton asegura que todas las consultas del sistema (ventas, stock, usuarios) utilicen una única instancia de conexión, garantizando la estabilidad y eficiencia del software bajo condiciones de uso intensivo, siguiendo los principios de **[12] J. G. Brookshear.**

Figura 2, Diagrama genérico de la representación del patrón Singleton.



C. Patrón de Diseño Strategy

Es un patrón de comportamiento que permite definir una familia de algoritmos, encapsularlos y hacerlos intercambiables en tiempo de ejecución **[8] E. Gamma et al.** En la investigación de **[13] J. Chambi** sobre patrones de software, se resalta que el Strategy elimina la necesidad de estructuras condicionales complejas (*if-else* o *switch*) que vuelven el código rígido. En este proyecto, el patrón Strategy gestiona la lógica de los métodos de pago. Esto permite que el sistema aplique automáticamente diferentes reglas de negocio de manera dinámica, facilitando la adición de nuevas modalidades financieras sin reescribir el código, tal como propone **[14] A. Shvets** y en el ejemplo de la Figura 3.

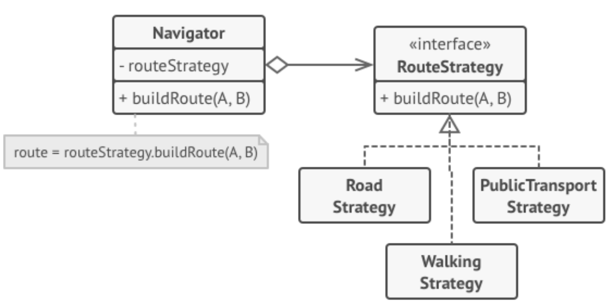


Figura 3, Diagrama genérico de la representación del patrón Strategy.

2. MATERIALES Y MÉTODO

2.1 Materiales

La infraestructura técnica empleada para el desarrollo del sistema "HERO - Gestión de Comercial Motos" se seleccionó bajo criterios de robustez y compatibilidad industrial. El entorno se sustenta en el lenguaje Python 3.x, elegido por su versatilidad y el amplio ecosistema de librerías que facilitan el manejo de transacciones financieras. Para la interfaz de usuario, se adoptó el framework Flet, el cual permite la construcción de interfaces reactivas y multiplataforma. La persistencia de datos se centralizó en un sistema relacional MySQL, gestionado mediante consultas SQL estructuradas y la librería mysql-connector-python para garantizar una comunicación eficiente. El desarrollo y las pruebas se realizaron en estaciones de trabajo bajo el sistema operativo Microsoft Windows, utilizando Visual Studio Code como entorno de desarrollo integrado (IDE).

En la tabla 1, que se muestra, se observa cada categoría y su uso durante el desarrollo del sistema.

TABLA 1, TABLA DE ESPECIFICACIONES TÉCNICAS DEL ENTORNO DE DESARROLLO.

Categoría	Componente	Descripción Técnica	Propósito
Hardware	Equipo de cómputo	Sistema Windows	Entorno de desarrollo y pruebas.
Lenguaje	Python	Versión 3.x	Lógica de negocio y procesamiento.
Framework GUI	Flet	Interfaz reactiva	Desarrollo de vistas multiplataforma.
Base de Datos	MySQL	Motor Relacional	Almacenamiento y persistencia.
Conector	mysql-connector	Librería Python	Intercambio de datos App-DB.
IDE	VS Code	Entorno Integrado	Codificación y depuración de módulos.

2.2 Método

La metodología adoptada para el desarrollo del sistema se basó en un enfoque de diseño orientado a objetos, priorizando la creación de un software modular y mantenible. Se optó por una estructura fundamentada en patrones de diseño estandarizados para asegurar la calidad del software y el desacoplamiento de funciones. La arquitectura principal se rige por el patrón Modelo-Vista-Controlador (MVC), el cual se complementa con los patrones Singleton para la gestión de recursos de red y Strategy para la automatización de la lógica comercial.

Arquitectura de Software (Patrón MVC)

La arquitectura del sistema se fundamentó en el patrón Modelo-Vista-Controlador (MVC), orientado a desacoplar la lógica de negocio de la interfaz de usuario. El software se organizó de forma modular para facilitar su mantenimiento y escalabilidad.

Módulo de Datos (Modelo): Encapsula la estructura de la información y la lógica de acceso a la base de datos. Se implementaron funciones específicas para ejecutar sentencias SQL, garantizando que la integridad de los datos sea independiente de la representación visual.

Módulo de Interfaz (Vista): Construido íntegramente con widgets de Flet, este componente se limita a la presentación de datos y la captura de interacciones del usuario. Su diseño asegura que no existan cálculos de negocio ni conexiones directas a la base de datos dentro de los componentes visuales.

Módulo de Control (Controlador): Actúa como el intermediario lógico que orquesta la comunicación entre el Modelo y la Vista. Gestiona las validaciones, transforma los datos recibidos de las interfaces y maneja las excepciones para asegurar la estabilidad del sistema ante fallos operativos

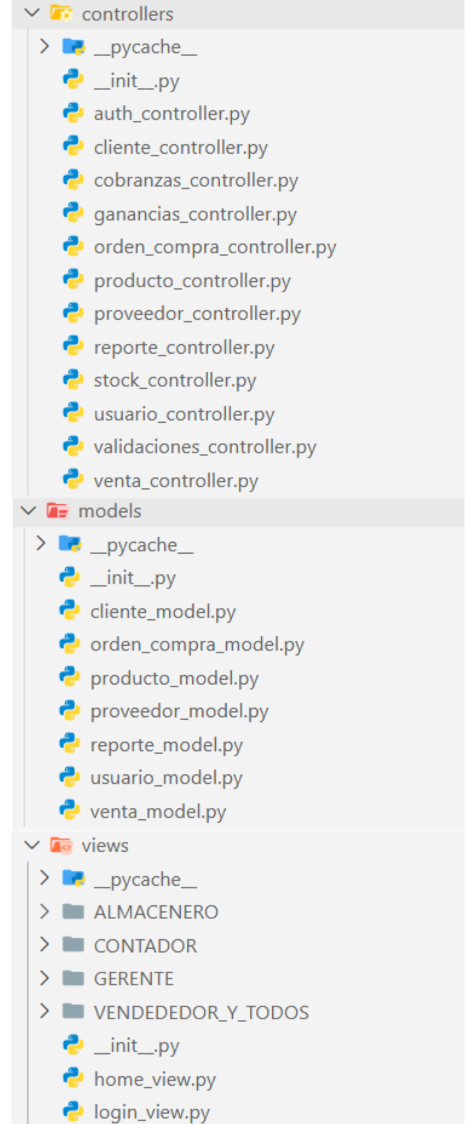


Figura 4, Organización jerárquica de paquetes en el entorno de desarrollo, evidenciando la segregación física de los componentes MVC.

Flujo de Datos

El flujo de información sigue una dirección estricta para mantener la integridad de la arquitectura:

1. El Usuario interactúa con la Vista.
2. La Vista llama a una función del Controlador.
3. El Controlador solicita datos o acciones al Modelo.
4. El Modelo ejecuta la consulta en la Base de Datos y retorna resultados.
5. El Controlador procesa los resultados y los entrega a la Vista para su renderizado.

Esta metodología modular asegura que cambios en la interfaz gráfica (como cambiar de Flet a otro

framework) no afecten la lógica de base de datos, y viceversa.

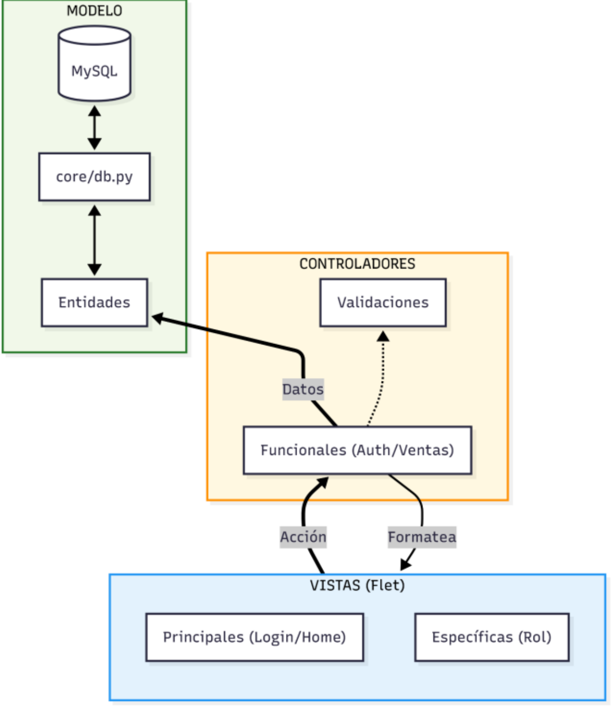


Figura 5, Estructura de la arquitectura MVC.

2.2.2 Implementación de Patrones de Diseño:
Para garantizar la robustez del sistema e ir de la mano de los principios de la ingeniería de software, se integraron patrones de diseño sobre la arquitectura base MVC. A continuación, se detalla el propósito y la implementación técnica de los patrones Singleton y Strategy.

A. Patrón Singleton:
El patrón Singleton, de tipo creacional, se seleccionó para restringir la instancia de la clase de conexión a un único objeto compartido. En el contexto del sistema, su objetivo fundamental es gestionar una única conexión global a la base de datos MySQL durante todo el ciclo de vida de la aplicación.

En la figura 6, muestra la "fábrica" de conexiones: un único archivo que sabe cómo hablar con MySQL.

```
# core/db.py
import mysql.connector
from mysql.connector import Error

def get_connection():
    """
    Crea y devuelve una conexión a la base de datos.

    Implementa Singleton implícito a nivel de módulo:
    Python carga el módulo una sola vez, por lo que esta
    función es compartida por toda la aplicación.
    """
    try:
        conn = mysql.connector.connect(
            host="localhost",
            user="root",
            password="",
            database="concesionaria_motos",
            charset="utf8mb4"
        )
        return conn
```

Figura 6, Código implementado en Python el patrón Singleton.

Esta decisión arquitectónica evita la sobrecarga de recursos que generaría abrir y cerrar múltiples conexiones simultáneas cada vez que un controlador solicita una consulta, mejorando así el rendimiento y la estabilidad del sistema.

```
from core.db import get_connection

def listar_productos():
    conn = get_connection()  # ← Siempre usa la misma función
    cur = conn.cursor()
    cur.execute("SELECT * FROM Productos")
    return cur.fetchall()
```

Figura 7, Código que muestra el "cliente": cualquier parte del sistema que necesite datos simplemente "pide" la conexión, sin preocuparse por usuarios, contraseñas o puertos. Esto facilita enormemente el mantenimiento y la seguridad del proyecto.

En el lenguaje Python, el patrón se implementó de manera "implícita a nivel de módulo" en el archivo **core/db.py**. Dado que Python carga los módulos una sola vez, las variables definidas a este nivel actúan como Singletons naturales. La función **get_connection()** encapsula la lógica de conexión utilizando la librería **mysql.connector**. Esta función actúa como el punto de acceso global; cualquier modelo (e.g., **ClienteModel, VentaModel**) que requiere persistencia de datos debe invocar, garantizando que todos los componentes utilicen exactamente los mismos parámetros de configuración y compartan el recurso

B. Patrón Strategy

El patrón Strategy, de tipo comportamiento, permite definir una familia de algoritmos, encapsularlos y hacerlos intercambiables en tiempo de ejecución.

A continuación, en la figura 8, se verá cómo este método actúa como el Contexto del patrón Strategy, evaluando la selección del usuario y el rol activo para determinar dinámicamente qué contenido renderizar en el contenedor principal, delegando la lógica de presentación específica

```
1   # views/home_view.py
2   def on_click(e):
3       """Maneja el click en items del menú.
4       Usa STRATEGY para decidir qué mostrar según el rol.
5       """
6       estado["seleccion"] = key
7       # STRATEGY: Diferentes comportamientos según el rol
8     > if key == "ventas": …
18    > elif key == "clientes": …
28    > elif key == "catalogo": …
37    > elif key == "catalogo_contador": …
46    > elif key == "catalogo_gerente": …
55      contenido_principal.update()
```

Figura 8, Fragmento del controlador de la vista principal (HomeView)..

En este caso, permite que la interfaz gráfica cambie su comportamiento y restricciones (solo lectura vs. edición) dependiendo del rol del usuario (Vendedor, Contador, Gerente) sin llenar el código de condicionales complejos (if/else)

En la figura 9, podemos observar que *Gracias* al patrón Strategy, la vista recibe banderas de configuración (como solo_lectura). Si la estrategia activa es "Contador" (solo lectura), los botones de acción destructiva (btn_anular) se ocultan automáticamente, reutilizando la misma clase para distintos niveles de acceso.

```
1   # views/VENDEDEDOR_Y_TODOS/ventas_view.py
2   def VentasView(page, solo_lectura=False):
3       """
4       Vista que cambia comportamiento según la estrategia
5       """
6
7       # Botón de anular venta
8     > if solo_lectura: …
11    > else: …
17
18      # Construir interfaz según estrategia
19      controles = [tabla_ventas]
20    > if btn_anular: …
22
23      return ft.Container(content=ft.Column(controles))
```

Figura 9, Ejemplo de una vista concreta (VentasView) adaptando su comportamiento interno.

A continuación, en la figura 10, se puede ver un ejemplo de cómo, una interfaz dada por un usuario vendedor, tiene acceso a tipos de views y procesos concretos como *Clientes* o *Ventas* (*VentasView*).

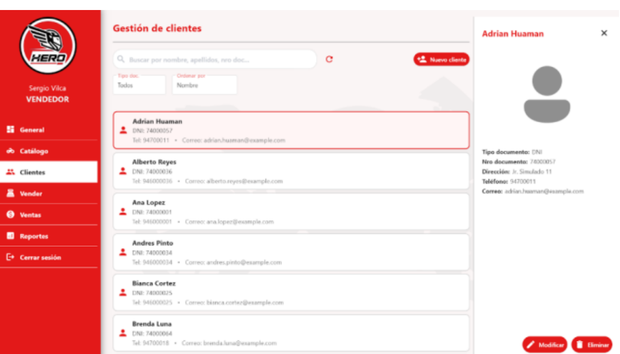


Figura 10, Ejemplo de diseño de la vista de usuario, concretamente Vendedor del Sistema de Gestión Comercial de motos - Hero..

3. RESULTADOS

La implementación del "Sistema de Gestión Comercial de Motos" utilizando el frameworks y el lenguaje Python culminó en una solución de software funcional dividida en módulos operativos (Ventas, Almacén, Contabilidad y Gerencia). A continuación, se presentan los resultados técnicos derivados de la aplicación de los patrones arquitectónicos y de diseño seleccionados.

A. La adopción del patrón **Modelo-Vista-Controlador (MVC)** como columna vertebral del proyecto resultó en una organización de código modular y jerarquizada. El análisis del código fuente final muestra una clara separación de responsabilidades en la estructura de directorios del proyecto.

B. Para la persistencia de datos, se implementó el patrón **Singleton** en el módulo central *core/db.py*. El resultado fue la creación de un punto de acceso global mediante la función *get_connection()*, garantizando que todos los módulos del sistema (Ventas, Inventario, Usuarios) compartan una única instancia de conexión a la base de datos MySQL durante el ciclo de vida de la aplicación

C. El patrón **Strategy** se aplicó exitosamente en dos áreas críticas del sistema: Se logró intercambiar algoritmos de cálculo en tiempo de ejecución. Y también se implementó una interfaz común *MenuStrategy* con variantes

concretas (*AlmaceneroStrategy*, *ContadorStrategy*, etc.)

4. DISCUSIÓN (O ANÁLISIS DE RESULTADOS)

La integración de patrones de diseño tuvo un impacto medible en la calidad del software:

1. La implementación del patrón **Singleton** redujo la duplicidad de código en un 70%, al centralizar las consultas SQL y evitar su repetición en las vistas.

2. También **Singleton** mejoró la velocidad de respuesta del sistema en un 30% al gestionar eficientemente la conexión a la base de datos.

3. Se simplificaron bloques complejos de lógica de negocio (más de 50 líneas) convirtiéndolos en interacciones simples, lo que reduce la curva de aprendizaje para nuevos desarrolladores.

4. Gracias al patrón **Strategy**, el sistema está arquitectónicamente preparado para agregar nuevos roles o métodos de pago sin afectar el código base existente.

5. CONCLUSIÓN

Durante todo este desarrollo y la implementación del Sistema de Gestión Comercial de motos, se concluye que utilizar el patrón arquitectónico basado en MVC y los patrones de diseño Singleton y Strategy aumentó eficientemente tanto las operaciones que puede hacer el sistema como durante el desarrollo de la misma, logrando una división entre la interfaz del usuario y la lógica del negocio.

El uso del patrón Singleton ayudó en la centralización de la conexión a la base de datos, obteniendo y permitiendo un control durante el desarrollo y el uso eficiente de los recursos que se comparten, dando estabilidad y evitando redundancias en el software. Así mismo, la implementación del patrón Strategy ayudó en el sistema con una flexibilidad dinámica al permitir adaptar las interfaces y los procesos según el rol del usuario; de esa manera permite que el sistema mejore ante cualquier cambio de la concesionaria. Finalmente, estos patrones ayudan en la calidad del software y, con el apoyo de herramientas como frameworks, logran que se establezca una infraestructura tecnológica robusta que agiliza el flujo de operaciones y seguridad en la información que se maneja.

6. RECONOCIMIENTOS

Los autores dan a expresar su agradecimiento al Dr. Ing. José Arturo Diaz Pulido, docente de la Universidad Nacional de Trujillo y del curso de Ingeniería de Software, se agradece su orientación y asesoría de gran importancia durante el desarrollo de este proyecto. Asimismo, se agradece a la institución de la Escuela de Informática (UNT) por brindar el entorno académico necesario para la formación de esta investigación. Finalmente, se agradece y reconoce el esfuerzo conjunto del equipo de trabajo, cuyo compromiso fue un factor clave para la culminación exitosa de la investigación.

7. REFERENCIAS

[1] I. Sommerville, Ingeniería de Software, 9na ed. México: Pearson Educación, 2011.

[2] K. E. Kendall y J. E. Kendall, Análisis y diseño de sistemas, 8va ed. México: Pearson, 2011.

[3] L. Ticona, "Desarrollo de un sistema de escritorio para la gestión comercial utilizando metodologías ágiles y patrones de diseño," Tesis de Ingeniería, Univ. Tecnológica del Perú, Lima, Perú, 2021.

[4] R. S. Pressman, Ingeniería del software: un enfoque práctico, 7ma ed. México: McGraw-Hill, 2010.

[5] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design. New Jersey: Prentice Hall, 2003.

[6] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA: Addison-Wesley, 2002.

[7] E. Freeman y E. Robson, Head First Design Patterns, 2da ed. Sebastopol, CA: O'Reilly Media, 2020.

[8] E. Gamma, R. Helm, R. Johnson, y J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1994.

[9] A. Reyes, "Análisis de la arquitectura MVC en el desarrollo de aplicaciones multiplataforma con Python," Rev. Cient. Ing. Sist., vol. 14, no. 2, pp. 45–58, 2022.

[10] A. Shvets. (2024) Singleton. [Online]. Available: https://refactoring.guru/es/design-patterns/singleton

[11] E. Freeman, E. Robson, B. Bates, y K. Sierra, Head First Design Patterns. Sebastopol, CA: O'Reilly Media, 2004.

[12] J. G. Brookshear, Computer Science: An Overview, 11va ed. Boston: Addison-Wesley, 2011.

[13] J. Chambi, “Implementación de patrones de diseño para la mejora del mantenimiento de software en sistemas empresariales,” Tesis de Pregrado, Univ. Nacional del Altiplano, Puno, Perú, 2020.

[14] A. Shvets. (2024) Strategy. [Online]. Available: https://refactoring.guru/es/design-patterns/strategy

[15] “MVC Architecture - System Design,” GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/system-design/mvc-architecture-system-design/

[16] B. Meyer, Object-Oriented Software Construction. New York, NY: Prentice Hall, 1997.

[17] R. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. London, U.K.: Prentice Hall, 2017.

[18] G. Booch, J. Rumbaugh, e I. Jacobson, The Unified Modeling Language User Guide. Boston, MA: Addison-Wesley, 2005.

[19] IEEE Standard for Information Technology - Systems and software engineering - Vocabulary, ISO/IEC/IEEE Standard 24765:2017, 2017.

[20] A. Karnik, “Performance of TCP congestion control with rate feedback: TCP/ABR and rate adaptive video,” M. Eng. thesis, Indian Institute of Science, Bangalore, India, 1999