Análisis práctico del desempeño de los patrones de Diseño de Software: Observer, strategy y Command en una aplicación

Evelin Dayana Miranda Bernal y Javier Darío Urrego Blanco

Estudiantes de Ingeniería de Sistemas, Facultad de Ingeniería e Innovación

Institución Universitaria Politécnico Grancolombiano

Wilson Eduardo Soto Forero

Tutor

Noviembre, 2023 Bogotá, DC

Tabla de Contenido

Lista de Tablas

Lista de Figuras

Datos del (los) Autor(es):

Nombres	Apellidos	Correo Electrónico	Programa Académico	Modalidad (Presencial / Virtual)
Evelin Dayana	Miranda Bernal	evmiranda@poligran.edu.co	Ingeniería de Sistemas	Presencial
Javier Darío	Urrego Blanco	jaurrego6@poligran.edu.co	Ingeniería de Sistemas	Presencial

Fecha de entrega del Proyecto de Investigación:

(DD/MM/AAAA):

03	12	2023

Resumen

Este proyecto se centra en evaluar la influencia de la implementación de patrones de diseño GoF, Observer, Strategy y Command, mediante el uso de métricas cuantitativas en aplicaciones Java. En el ámbito del desarrollo de software, la elección de patrones de diseño es crítica y puede impactar significativamente el rendimiento y la calidad de las aplicaciones. La falta de evidencia empírica sólida sobre cómo afectan estos patrones en métricas específicas deja a los desarrolladores en terreno incierto al tomar decisiones de diseño. La investigación aborda esta

brecha al seleccionar patrones representativos y aplicar métricas Uso de CPU, tiempo de uso de CPU y Tiempo de Ejecución. Se justifica la elección de estos patrones por su relevancia y ventajas específicas en la implementación de in prototipo de aplicación la biblioteca de software, buscando aportar evidencia empírica y práctica para respaldar las decisiones de diseño.

Palabras Clave (Key Words): Design patterns, Development, Metrics, GoF (Gang of Four), Observability

1. INTRODUCCIÓN

El mundo del desarrollo de software es un campo en constante evolución, donde la capacidad de crear aplicaciones eficientes y robustas se convierte en un desafío perpetuo. En este entorno dinámico, la toma de decisiones correctas adquiere una importancia crítica. Uno de los aspectos más importantes de esta toma de decisión es la elección de patrones de diseño en el desarrollo de aplicaciones Java ya que pueden proporcionar un enfoque estructurado y eficiente para abordar desafíos específicos que, aplicados adecuadamente, pueden mejorar la calidad, el rendimiento y la mantenibilidad del software.

El propósito de esta investigación es abordar esta problemática, evaluando el impacto de la implementación de patrones de diseño en el rendimiento de una aplicación. En particular, nos centraremos en los patrones de diseño del grupo "Gang of Four" (GoF), que son ampliamente utilizados en la industria del software. Los patrones que examinaremos son Observer, strategy y Command. Estos patrones han demostrado ser valiosos en una variedad de contextos, y su impacto en la eficiencia de las aplicaciones Java es una pregunta que merece una investigación exhaustiva.

Es importante destacar que, a lo largo de esta investigación, se abordará el desafío de aplicar los patrones de diseño en el contexto de las aplicaciones Java y evaluar su influencia en métricas como el rendimiento, la escalabilidad, la mantenibilidad y la seguridad. Este enfoque permitirá a los desarrolladores comprender mejor el impacto real de estas decisiones de diseño y tomar decisiones fundamentadas en sus proyectos. Dada la diversidad de patrones de diseño

disponibles y sus diferentes aplicaciones, se explorará detalladamente la selección de patrones óptimos para abordar la complejidad de la implementación.

Nuestra investigación utilizará métricas cuantitativas para evaluar el rendimiento de una aplicación de prototipo que implementa estos patrones de diseño. Estas métricas nos permitirán medir el impacto en áreas críticas, como el uso de CPU, Tiempo de uso de CPU, el tiempo medio de recuperación-Ejecución. Al analizar y comparar estas métricas en diferentes escenarios de implementación de patrones, esperamos obtener una comprensión más profunda de cómo las decisiones de diseño afectan la eficiencia de las aplicaciones Java.

En resumen, esta investigación busca proporcionar evidencia empírica sólida sobre el impacto de la implementación de patrones de diseño en métricas cuantitativas en el contexto de aplicaciones Java. Esto contribuirá a la toma de decisiones informadas por parte de los desarrolladores y ayudará a cerrar la brecha entre la teoría y la práctica en el diseño de software.

2. ALCANCE DEL PROYECTO

Este proyecto se enfocará en el análisis del impacto de tres patrones de diseño de comportamiento (Observer, strategy y Command) en el rendimiento de una aplicación prototipo desarrollada en Java. En particular, la investigación abarcará lo siguiente:

Medición y recolección de métricas de rendimiento como uso de CPU, consumo de memoria, tiempo medio de recuperación y porcentaje de código reutilizado para cada patrón antes y después de su implementación. Análisis comparativo de las métricas entre los diferentes escenarios de aplicación de patrones para evaluar el impacto específico de cada uno.

Evaluación del comportamiento de las métricas al modificar parámetros como cantidad de usuarios registrados en la aplicación con y sin patrones. Documentación de resultados y elaboración de recomendaciones sustentadas para la selección óptima de patrones orientada al rendimiento.

El proyecto busca determinar, dentro de los límites descritos, el impacto cuantitativo de la implementación de estos populares patrones de diseño sobre el rendimiento, para apoyar en el futuro decisiones informadas de arquitectura de software.

3. PLANTEAMIENTO DEL PROBLEMA DE INVESTIGACIÓN

En el ámbito del desarrollo de software, la elección de patrones de diseño es una tarea crítica que puede afectar drásticamente el rendimiento y la calidad de las aplicaciones de Software. Sin embargo, en un entorno en constante evolución, la complejidad radica en comprender cómo la adopción de estos patrones impacta realmente en la funcionalidad del software, y cómo estas decisiones pueden ser optimizadas para producir aplicaciones más eficientes y efectivas.

La diversidad de patrones de diseño disponibles complica aún más el proceso de selección y aplicación de patrones. Cada patrón tiene características y aplicaciones específicas, lo que plantea un desafío para los desarrolladores al determinar la agrupación de patrones más adecuada para proyectos específicos. Esta complejidad aumenta a medida que la tecnología evoluciona y se incorporan nuevas herramientas y prácticas de desarrollo. Como resultado, los desarrolladores se enfrentan a una toma de decisiones compleja donde deben equilibrar la elegancia del diseño con la eficiencia de la ejecución.

Las métricas cuantitativas, como el rendimiento, la escalabilidad, la mantenibilidad y la seguridad, son fundamentales para evaluar el impacto de la implementación de patrones de diseño en aplicaciones de software. Estas métricas permiten una evaluación objetiva y basada en datos de cómo los patrones afectan la funcionalidad y el comportamiento del software. Sin embargo, la falta de evidencia empírica sólida en este contexto dificulta a los desarrolladores tomar decisiones informadas y respaldadas por datos.

Abordar esta necesidad implica, por lo tanto, una investigación que combine la teoría y la práctica buscando contribuir a la comprensión empírica de cómo la elección y aplicación de patrones de diseño afecta directamente las métricas de una aplicación. Los resultados de esta investigación proporcionarán una base sólida para que los desarrolladores tomen decisiones de diseño respaldadas por datos y fomentarán la adopción efectiva de patrones en la práctica del desarrollo de software.

En este contexto, surge la problemática que abordaremos en esta investigación: ¿Es posible evaluar el impacto del uso de patrones Observer, strategy y Command en un prototipo de aplicación en Java?, Esta pregunta es el punto de partida para explorar el impacto de las elecciones de diseño en el terreno complejo y multidimensional de la eficiencia y la calidad del software.

4. OBJETIVOS DE INVESTIGACIÓN

4.1. **Objetivo General**

Evaluar la influencia de la implementación de patrones de diseño Gof mediante métricas.

4.2. Objetivos Específicos

- Realizar una revisión literaria de los patrones de diseño de software GoF incluyendo sus aplicaciones prácticas.
- Identificar los patrones de diseño de software GoF para ser implementados y las métricas de evaluación de su desempeño.
- Diseñar e implementar un prototipo de aplicación para acoplar los patrones de diseño GoF seleccionados.
- Analizar los resultados obtenidos de una aplicación con y sin la implementación de los patrones de diseño de software usando las métricas elegidas.

5. REVISIÓN LITERARIA

Los patrones de diseño son elementos esenciales en el mundo del desarrollo de software. Surgieron como soluciones probadas para problemas comunes de diseño y se han convertido en herramientas fundamentales para los desarrolladores en la creación de sistemas efectivos y eficientes. Su impacto en la arquitectura de software y en la calidad de los proyectos es innegable.

5.1. Arquitectura de Software

El concepto de "arquitectura de software" se remonta a la década de los sesenta, cuando Dijkstra habló de la estructuración correcta de sistemas de software. Sin embargo, el término "arquitectura de software" tal como lo entendemos hoy en día no se popularizó hasta 1992 con el trabajo de Perry y Wolf. Durante la década de los setenta, hubo un debate sobre la semántica y la expresión pragmática de la arquitectura de software.

En los ochenta, la orientación a objetos empezó a destacar, con Mary Shaw contribuyendo con trabajos sobre abstracciones de alto nivel. A finales de los ochenta y principios de los noventa, la noción de que las aplicaciones tienen una morfología o estructura comenzó a tomar forma. Perry y Wolf, en 1992, propusieron un modelo de arquitectura de software que incluye elementos, forma y razón, y predijeron que la década de los noventa sería la "década de la arquitectura de software". En la segunda mitad de los años noventa, comenzaron a aparecer los primeros libros de texto dedicados a la arquitectura de software. En el año 2000, dos acontecimientos clave marcaron el cierre de esa década: la tesis de Roy Fielding introdujo el modelo REST, que se

centraba en Internet y los modelos orientados a servicios, mientras que el IEEE generó la recomendación IEEE std 1471-2000, estableciendo un marco definitivo.

En ese mismo año, surgieron nuevas perspectivas para la arquitectura de software, como las estrategias orientadas a líneas de productos y la integración de la arquitectura de software en el ciclo de vida del desarrollo, lo que requirió una redefinición de las metodologías relacionadas con ella.

Hoy en día, existe un creciente interés en el desarrollo centrado en arquitectura, los métodos de análisis y diseño de arquitecturas (dentro del ciclo de vida del software), el análisis de arquitecturas de software basado en escenarios, los modelos de evaluación de arquitecturas de software y los modelos orientados por la arquitectura, entre otros temas. (Fernandez, 2006)

A lo largo de esa década, surgieron propuestas significativas como la programación basada en componentes, patrones de diseño, el modelo de 4+1 vistas y lenguajes de descripción de arquitecturas (ADLs). Así, la arquitectura de software se convirtió en un campo de estudio relevante en la industria del software.

A medida que hemos recorrido la evolución histórica de la arquitectura de software hasta el momento actual, podemos identificar un cambio significativo en la forma en que abordamos y conceptualizamos el diseño de software. Desde los primeros debates sobre la estructuración correcta de sistemas de software en los sesenta hasta la consolidación del término "arquitectura de software" en 1992 con Perry y Wolf, hemos visto cómo se ha desarrollado una comprensión más profunda de la importancia de planificar y organizarlo de manera efectiva.

En la década de los ochenta, la orientación a objetos se convirtió en una tendencia dominante, y los trabajos de Mary Shaw sobre abstracciones de alto nivel contribuyeron a este cambio de paradigma. Esto abrió la puerta a una mayor atención a la estructura y la morfología de las aplicaciones de software. Perry y Wolf introdujeron un modelo de arquitectura de software en 1992, sentando las bases para una disciplina que se convertiría en fundamental en el desarrollo de software. (Regalado, 2010)

Sin embargo, es importante notar que el concepto de arquitectura de software no existiría tal como lo conocemos sin la presencia de los "patrones de diseño". Estos patrones ofrecen soluciones probadas a problemas comunes de diseño y desempeñan un papel crucial en la creación de una arquitectura de software sólida y eficiente. Según avanzamos en el tema de los patrones de diseño, veremos cómo se integran, enriquecen y optimizan la arquitectura de software y cómo han influido en cómo diseñamos y desarrollamos software actualmente.

5.2. Patrones de Diseño GOF

Los patrones de diseño GoF son un componente esencial en la construcción de arquitecturas de software sólidas y eficientes. Estos patrones ofrecen soluciones probadas y prácticas a problemas comunes de diseño, y han sido fundamentales en la conformación de cómo diseñamos y desarrollamos software actualmente. Por lo tanto, a medida que profundicemos en el tema de los patrones de diseño, apreciaremos cómo estas soluciones han enriquecido y optimizado la arquitectura de software en el tiempo.

Además, los patrones de Diseño son soluciones a problemas de diseño que se aplican en un contexto específico y que equilibran diferentes fuerzas o consideraciones para resolver un problema de manera efectiva. (Coplien, 2003) Ofrecer soluciones ágiles y prácticas, jugando un papel fundamental en la conformación de cómo diseñamos y desarrollamos software actualmente.

En relación con el acrónimo "GOF (Gang of Four)" se refiere a "Gang of Four" (Banda de los Cuatro, en español), que es un término utilizado para describir un grupo de cuatro influyentes autores que escribieron el libro "Design Patterns: Elements of Reusable Object-Oriented Software" en 1994. Este libro es un recurso fundamental en la programación orientada a objetos y estableció patrones de diseño utilizados para abordar problemas recurrentes en el diseño de software. (1994) Los autores, conocidos como la "Banda de los Cuatro", son Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Su trabajo revolucionario proporcionó soluciones probadas a desafíos de diseño comunes, lo que permitió a los desarrolladores crear software más reutilizable y extensible. A través de la aplicación de estos patrones, se promueve un enfoque más eficiente y estructurado para el desarrollo de software orientado a objetos, lo que ha llevado a su amplia adopción en la industria de la programación.

5.3. Características

Es importante destacar que no todas las soluciones que comparten características con un patrón pueden considerarse como tal. Para que una solución se califique como un patrón, debe demostrarse que aborda un problema recurrente. La validación de una solución como patrón

implica someterla a una serie de pruebas conocidas como "test de patrones"; hasta entonces, se considera una buena solución o un "proto-patrón".

Según los autores del libro “Design patterns: Elements of reusable object-oriented software” existen ciertas cualidades deseables que definen un buen patrón:

1. Debe resolver un problema específico: Los patrones se enfocan en ofrecer soluciones concretas a problemas recurrentes, no solo en principios o estrategias abstractas.
2. Debe ser una solución probada: Los patrones son soluciones que han sido validadas en la práctica, no teorías o especulaciones sin evidencia empírica.
3. La solución no debe ser obvia: Los mejores patrones no proporcionan soluciones directas y evidentes; en su lugar, abordan problemas de manera indirecta y creativa.
4. Describe participantes y relaciones: Los patrones no se limitan a describir módulos, sino que también abordan estructuras y mecanismos más complejos en el sistema.

La repetición es un elemento clave en la identificación de un patrón, ya que los patrones suelen manifestarse en múltiples sistemas. (Junio, 1994) Sin embargo, la repetición no es la única característica relevante; también es fundamental demostrar que un patrón es adaptable y útil. La adaptabilidad y utilidad son cualidades cualitativas, a diferencia de la repetición, que es cuantitativa. Para demostrar la adaptabilidad, es necesario mostrar cómo un patrón puede aplicarse exitosamente. Además, para demostrar la utilidad, se debe explicar por qué un patrón es beneficioso y exitoso.

Varios autores han enumerado cualidades deseables adicionales en los patrones. Según el libro "Christopher Alexander: an Introduction for Object-Oriented Designers" de Doug Lea, (Junio, 1994) se destacan las siguientes:

- **Encapsulamiento y abstracción:** Cada patrón encapsula un problema y su solución en un dominio específico, proporcionando límites claros para definir el alcance del problema y la solución.
- **Extensión y variabilidad:** Los patrones deben ser extensibles y adaptables, lo que les permite combinarse con otros patrones para abordar problemas más complejos. También deben admitir múltiples implementaciones individualmente y junto con otros patrones.
- **Generatividad y composición:** Aplicar un patrón genera un contexto resultante que se integra con otros patrones del lenguaje. Los patrones están jerárquicamente relacionados y pueden componerse tanto de forma ascendente como descendente en la jerarquía.
- **Equilibrio:** Cada patrón debe equilibrar sus efectos y restricciones, asegurando que su aplicación no genere problemas no deseados.

5.4. Beneficios

Los aspectos cruciales en cualquier proyecto de desarrollo de software incluyen los costos, la satisfacción del cliente, la productividad y la reducción de los tiempos de desarrollo. Los patrones de diseño contribuyen indirectamente a mejorar estos aspectos. En cuanto a la productividad, los patrones benefician a diseñadores novatos al proporcionarles experiencia en el dominio, reduciendo así el tiempo necesario para descubrir las estructuras de diseño clave.

Además, evitan el retrabajo causado por decisiones de diseño inexpertas. En lo que respecta a la reducción de los tiempos de desarrollo, los patrones permiten la reutilización de soluciones de diseño, lo que agiliza la etapa de diseño al evitar la necesidad de crear soluciones desde cero. Esta reducción de costos está directamente relacionada con los dos puntos anteriores. La satisfacción del cliente se deriva de los factores mencionados.

Tal como se menciona en el artículo de Investigación "Comparative Analysis of Software Design Patterns" si bien es evidente que el uso de patrones conlleva numerosos beneficios, es importante tener en cuenta ciertas limitaciones. En primer lugar, dado que los patrones representan experiencia, puede que no existan patrones importantes para un nuevo dominio, lo que dificulta su aplicación en el desarrollo del primer sistema. Aunque algunos patrones pueden ser aplicables en varios dominios, cada nuevo dominio tendrá sus propios patrones específicos que solo se descubrirán con la experiencia en ese dominio particular.

En segundo lugar, los patrones orientan a los seres humanos en la toma de decisiones, pero no generan automáticamente código ni residen en herramientas CASE (Computer-Aided Software Engineering o Ingeniería de Software Asistida por Computadora). Son una forma de literatura que asiste en el proceso de diseño, pero no pueden reemplazar la creatividad y habilidades de un programador humano. Por último, aunque los patrones pueden transmitir el conocimiento de diseñadores expertos, no transformarán instantáneamente a principiantes en expertos. En resumen, los patrones de diseño son una valiosa herramienta en el arsenal de los diseñadores que pueden acelerar el aprendizaje para abordar problemas, pero no sustituyen el sentido intuitivo de la estética de los diseñadores experimentados.

5.5. Tipos y Características

Por otro lado, los patrones de diseño difieren en su nivel de complejidad, grado de detalle y alcance de utilidad por lo que el siguiente paso en nuestra investigación será conocer y entender algunos de los más conocidos patrones de diseño basándonos en algunos artículos de investigación, pero mayormente citando a los autores del libro hasta ahora más mencionado en nuestra lectura, “Design Patterns: Elements of Reusable Object-Oriented Software”.

Teniendo en cuenta esto, es posible categorizarlos según su objetivo, dividiéndolos en tres conjuntos distintos; Creacionales, Estructurales y Comportamentales.

5.5.1. Creacionales (Creational Patterns)

Estos patrones se centran en la creación de objetos y cómo se pueden instanciar de manera flexible sin exponer la lógica de creación. Los patrones creacionales GOF incluyen:

5.5.1.1. **Patrón Singleton (Singleton)**: El patrón Singleton es uno de los patrones de diseño más conocidos y ampliamente utilizados en el desarrollo de software. Su objetivo principal es asegurar que una clase tenga una única instancia y proveer una vía de acceso global a esa instancia.

5.5.1.2. **Patrón Fábrica (Factory Method)**: Ó Patrón de Fábrica en español, se utiliza comúnmente en el desarrollo de software. Su objetivo principal es proporcionar una interfaz que permita la creación de objetos en una superclase, pero permitiendo a las subclases variar el tipo de objetos que se crearán.

5.5.1.3. **Patrón Fábrica Abstracta (Abstract Factory)**: El patrón Abstract Factory, o Patrón de Fábrica Abstracta en español, se utiliza para proporcionar una interfaz para crear familias de objetos que tienen una relación o alguna dependencia sin necesidad de especificar sus clases concretas.

5.5.1.4. **Patrón Constructor (Builder)**: El patrón Builder, o Patrón Constructor en español, se utiliza para construir un objeto complejo paso a paso.

5.5.1.5. **Patrón Prototipo (Prototype)**: El patrón Prototipo (Prototype) es un patrón que se utiliza para crear nuevos objetos generando duplicados de un objeto que ya existe, conocido como prototipo.

A continuación, se muestra un cuadro comparativo donde se enuncian las principales características, ventajas y desventajas de los patrones de diseño.

Tabla 1
Patrones de diseño Creacionales

Patrón de Diseño	Características	Ventajas	Desventajas
Singleton (Singleton)	- Garantiza una única instancia. - Acceso global a la instancia. -Constructor privado.	- Ahorro de recursos. - Control sobre el acceso a la instancia.	- Puede dificultar las pruebas unitarias. - Rompe con el principio de OCP (Una clase debería estar abierta a extensión, pero cerrada a

			modificación).
Factory Method (Método de Fabrica)	- Proporciona una estructura para la creación de objetos a través de una interfaz, delegando a las subclases la responsabilidad de determinar qué clase específica debe ser instanciada. -Interfaz común para crear objetos.	- Encapsulación de la creación de objetos. - Flexibilidad para subclases en la creación de objetos. -Reduce el acoplamiento entre el cliente y las clases concretas.	- Puede introducir una complejidad adicional en el uso. -Posible Redundancia.
Abstract Factory (Fabrica Abstracta)	-Ofrece una interfaz para generar conjuntos de objetos interrelacionados sin detallar las clases específicas involucradas. - Desacoplamiento ya que permite al cliente	- Asegura que los objetos creados sean coherentes en términos de interfaz. - Soporta principios de diseño SOLID.	- Puede ser complicado de implementar cuando se agregan nuevas familias de objetos. - Dificulta la adición de nuevas familias de objetos.

	interactuar con objetos abstractos y no con clases concretas.		
Builder (Constructor)	-Divide la creación de un objeto complejo de su estructura, permitiendo una construcción independiente de la representación de este. - Permite construir objetos paso a paso. - Facilita la construcción de objetos complejos.	- Facilita la creación de objetos con configuraciones específicas. - Mayor flexibilidad en la creación de objetos.	- Requiere la implementación de un director si es necesario. - Aumenta la complejidad en la implementación. -Consumo de memoria.
Prototipo (Prototype)	-Facilita la generación de nuevos objetos mediante la duplicación de un prototipo existente.	- Facilita la creación de objetos duplicados. - Menos carga en la memoria. -Evita acoplamientos fuertes entre el código	- Dificultad en la clonación de objetos complejos. - Requiere una interfaz para clonar.

		que utiliza los objetos y las clases de los objetos	

5.5.2. Estructurales (Structural Patterns)

Estos patrones se centran en cómo las clases y objetos se componen para formar estructuras más grandes y flexibles. Estos patrones incluyen:

5.5.2.1. **Patrón Adaptador (Adapter)**: El diseño de patrón Adaptador es una estructura que se emplea para posibilitar la colaboración entre dos interfaces que no son compatibles entre sí.

5.5.2.2. **Patrón Puente (Bridge)**: El patrón Bridge es una estrategia de diseño estructural empleada para desvincular una abstracción de su implementación, posibilitando que ambas puedan evolucionar de manera independiente.

5.5.2.3. **Patrón Composite (Composite)**: El patrón Composite (Composite) es un patrón de diseño estructural que permite componer objetos en estructuras de árbol para representar jerarquías de objetos todo-parte.

5.5.2.4. **Patrón Decorador (Decorator)**: El patrón Decorator es una técnica de diseño estructural que facilita la adición dinámica de funcionalidades adicionales a objetos individuales.

5.5.2.5. **Patrón Fachada (Facade):** El patrón Facade es una estrategia de diseño estructural que suministra una interfaz simplificada para un conjunto más

extenso y complicado de clases, con el propósito de hacer su utilización más accesible y sencilla.

5.5.2.6. **Patrón Peso Ligero (Flyweight)**: El patrón Peso Ligero (Flyweight) es un patrón de diseño estructural que se utiliza para reducir la cantidad de memoria o almacenamiento utilizado por un conjunto grande de objetos al compartir datos comunes de manera eficiente.

5.5.2.7. **Patrón Proxy (Proxy)**: El patrón Proxy también hace parte de los patrones de diseño estructurales y se utiliza para controlar el acceso a un objeto o recurso de manera indirecta. Actúa como una capa intermedia que intercepta las solicitudes y permite llevar a cabo tareas adicionales tanto antes como después de acceder al objeto real

Tabla 2

Patrones de diseño Estructurales

Patrón de Diseño	Características	Ventajas	Desventajas
Adapter (Adaptador)	-Facilita la colaboración entre interfaces que son incompatibles entre sí. - Se compone de un objeto existente que realiza la adaptación. -Permite la reutilización de clases existentes.	- Reduce el acoplamiento entre sistemas o componentes al proporcionar una capa intermedia de adaptación - Conexión de componentes sin modificar su código.	- Aumenta la complejidad. - Puede causar pérdida de funcionalidad. - Posible Impacto en el Rendimiento. - Dificultad en la Identificación.

Bridge (Puente)	- Separa una abstracción de su implementación, lo que permite que ambas varíen independientemente. - Abstracción contiene una referencia a la implementación. -Promueve la composición en lugar de la herencia, lo que hace que sea más flexible y extensible.	-Desvincula la abstracción y la implementación. - Permite una jerarquía de abstracciones e implementaciones. - Reutilización de código.	- Aumenta la complejidad con múltiples capas de clases. - Puede ser sobreingeniería si hay una sola implementación. -Mayor cantidad de clases.
Composite (Compuesto)	- Organiza objetos en estructuras de árbol para representar jerarquías. - Trata objetos individuales y composiciones uniformemente. -Permite la composición recurrente.	- Manejo de objetos complejos como uno solo. - Facilita la creación de estructuras jerárquicas.	- Añade complejidad a la gestión y navegación. - No es adecuado para todos los casos de uso. -Ineficiencia si no ejecuta adecuadamente.
Decorator (Decorador)	- Añade responsabilidades	- Cumple con el principio OCP.	- Puede resultar en un número

	adicionales a un objeto de manera dinámica. - Permite extensiones sin modificar su estructura.	- Evita la necesidad de una clase de subclase excesiva.	excesivo de pequeñas clases. - Requiere un cuidado especial al diseñar.
Fachada (Facade)	- Ofrece una interfaz unificada para un conjunto de interfaces en un subsistema. -Simplifica el uso de un sistema mayor.	- Simplifica y oculta la complejidad del sistema. - Aísla el cliente de los componentes internos.	- Puede convertirse en un punto de hinchamiento. - Puede reducir el control sobre el subsistema.
Peso Ligero (Flyweight)	- Divide un objeto en dos partes: estado intrínseco y estado extrínseco. - Reutiliza objetos para conservar memoria.	- Reduce el uso de memoria. - Facilita la creación de muchos objetos.	- Aumenta la complejidad al distinguir el estado intrínseco y extrínseco. - Requiere una gestión adecuada de objetos compartidos.
Proxy (Proxy)	- Ofrece un sustituto o marcador de posición para otro objeto con el fin de gestionar el acceso a este último.	- Permite controlar el acceso al objeto real. - Protege el objeto real al agregar una capa adicional de control.	- Aumenta la complejidad al introducir una capa adicional. - Puede impactar el

	- Control de acceso, registro, manejo de solicitudes y monitoreo.		rendimiento.

5.5.3. Comportamiento (Behavioral Patterns)

Estos patrones se centran en cómo los objetos interactúan y comunican entre sí y cómo distribuyen responsabilidades. Los patrones de comportamiento GOF incluyen:

5.5.3.1. **Patrón Cadena de Responsabilidad (Chain of Responsibility)**: El patrón de Cadena de Responsabilidad es una estrategia de diseño comportamental que se emplea para hacer pasar solicitudes a través de una serie de manejadores o procesadores. Cada manejador toma la decisión de procesar la solicitud o transmitirla al siguiente manejador en la cadena.

5.5.3.2. **Patrón Comando (Command)**: El patrón de diseño Comando se emplea para encapsular una solicitud como un objeto, posibilitando la parametrización de clientes con colas, solicitudes y operaciones.

5.5.3.3. **Patrón Intérprete (Interpreter)**: El patrón de diseño Intérprete (Interpreter) es un patrón de comportamiento que se utiliza para delimitar una gramática para un lenguaje y proporcionar una forma de estimar o evaluar las expresiones escritas para ese lenguaje.

5.5.3.4. **Patrón Iterador (Iterator)**: El patrón de diseño Iterador (Iterator) es una estrategia de comportamiento que se emplea para ofrecer una manera de acceder de forma secuencial a los elementos de una colección sin revelar la estructura interna de dicha colección

5.5.3.5. **Patrón Mediador (Mediator)**: El patrón de diseño Mediador (Mediator), se utiliza para reducir las dependencias complejas entre múltiples objetos o componentes de un sistema. Proporciona un punto central de comunicación para que estos objetos interactúen entre sí.

5.5.3.6. **Patrón Mememto (Mememto)**: El patrón de diseño Memento (Memento) es un patrón de comportamiento que se utiliza para capturar y almacenar el estado interno de un objeto sin romper la encapsulación. Este estado puede restaurarse después sin que otros objetos accedan directamente a la información almacenada.

5.5.3.7. **Patrón Observador (Observer)**: El patrón de diseño Observador (Observer), se utiliza para definir una dependencia uno a muchos entre objetos. En este patrón, cuando un objeto (llamado sujeto) cambia su estado, todos los objetos dependientes (llamados observadores) son notificados y actualizados automáticamente.

5.5.3.8. **Patrón Estado (State)**: El Patrón de Diseño Estado (State) hace parte de los patrones de comportamiento que concede a un objeto cambiar su función cuando su estado interno evoluciona. El objeto tendrá diferentes clases

representando sus estados y delegará las operaciones relacionadas con el estado a la clase correspondiente.

5.5.3.9. **Patrón Estrategia (Strategy)**: El Patrón de Diseño Estrategia (Strategy) posibilita la definición de una familia de algoritmos, encapsulando cada uno y permitiendo su intercambio. Los objetos que emplean esta estrategia tienen la capacidad de cambiar su comportamiento en tiempo de ejecución, seleccionando una de las estrategias disponibles sin necesidad de alterar su estructura.

5.5.3.10. **Patrón Visitante (Visitor)**: El Patrón de Diseño Visitante (Visitor) es una estrategia que simplifica la adición de funcionalidades adicionales a objetos sin necesidad de modificar su estructura.

5.5.3.11. **Patrón Comando (Command)**: El patrón de diseño Command permite encapsular solicitudes como objetos, lo que facilita su parametrización, encolado, ejecución en momentos diferentes y la capacidad de deshacer o rehacer acciones. Promueve una separación eficiente entre los objetos que emiten solicitudes y los que las ejecutan, lo que resulta en una estructura flexible y desacoplada. Es útil para implementar operaciones reversibles, macros y secuencias de comandos.

Tabla 3
Patrones de diseño de Comportamiento

Patrón de Diseño	Características	Ventajas	Desventajas
Cadena de	- Pasa la solicitud a lo	- Desacopla al emisor	- Sin garantía de que

Responsabilidad (Chain of Responsibility)	largo de una cadena de objetos receptores encadenados. - Cada receptor toma la decisión de procesar la solicitud o transmitirla a la siguiente entidad en la cadena.	de un conjunto de receptores. - Permite agregar o eliminar responsabilidades dinámicamente.	la solicitud sea manejada. - Puede ser difícil de depurar.
Comando (Command)	- Encapsula una solicitud como un objeto, permitiendo la parametrización de clientes con dichas solicitudes. - Define una interfaz que todas las solicitudes deben implementar.	- Desacopla el emisor de las operaciones de receptor. - Permite la gestión de comandos de manera secuencial o en colas.	- Aumenta la cantidad de clases. - Puede ser complejo en aplicaciones con muchas operaciones.
Interpreter (Intérprete)	-Define una gramática para un lenguaje y suministra un intérprete para evaluar sentencias en dicho lenguaje. - Transforma expresiones en un	- Facilita la creación de lenguajes o gramáticas personalizados. - Permite evaluar expresiones dinámicamente.	- Aumenta la complejidad al definir gramáticas y evaluadores. - No es eficiente para expresiones simples.

	lenguaje específico en una estructura de datos.		
Mediador (Mediator)	- Ofrece un medio para acceder secuencialmente a los elementos de una colección sin exponer su representación interna. - Define una interfaz para navegar elementos.	- Desacopla al cliente de la estructura de la colección. -Permite la variabilidad en la forma de recorrer una colección.	- Añade complejidad al código al introducir el patrón. - Puede resultar en un aumento en el consumo de memoria.
Memento (Memento)	- Captura y externaliza un estado interno de un objeto para que el objeto se restaure a este estado después. - Permite el restablecimiento del objeto a un estado previo.	- Facilita la recuperación de estados anteriores. - Permite implementar la función "deshacer".	- Puede volverse un objeto grande y complejo a medida que se añaden más objetos. - Puede ralentizar la comunicación.
Strategy (Estrategia)	- Define una familia de algoritmos, encapsula cada uno de ellos, y permite que sean intercambiables entre	- Facilita la selección y cambio de algoritmos en tiempo de ejecución. - Desacopla el cliente	- Aumenta la cantidad de objetos y clases. - Puede parecer complejo si se usan demasiadas

	sí. - Permite que el cliente elija el algoritmo que debe utilizar.	de la implementación detallada de los algoritmos.	estrategias.
Visitor (Visitante)	- Representa una operación a realizar en los elementos de una estructura de objetos. - Permite definir una nueva operación sin cambiar las clases de los elementos.	- Facilita agregar nuevas operaciones. - Desacopla el código de las clases de elementos.	- Aumenta la cantidad de clases y complejidad. - Puede resultar en un código extenso si hay muchas operaciones.
Command (Comando)	- Encapsula una solicitud como un objeto, posibilitando la parametrización de clientes con dichas solicitudes. - Define una interfaz que todas las solicitudes deben implementar. - Independencia del invocador y el receptor:	- Desacopla el emisor de las operaciones de receptor. - Permite la gestión de comandos de manera secuencial o en colas. - Facilita la implementación de deshacer/rehacer.	- Aumenta la cantidad de clases. - Puede ser complejo en aplicaciones con muchas operaciones.

En resumen, hemos explorado una amplia gama de patrones de diseño, cada uno con su propósito y aplicación específicos en el desarrollo de software. Ahora, nos enfocaremos en la elección de las herramientas y aplicaciones necesarias para medir y evaluar las métricas resultantes de la implementación de estos patrones en nuestros proyectos. Esta selección será fundamental para asegurar el éxito y la efectividad de nuestras estrategias de diseño basadas en patrones.

5.6. ¿Cómo evaluar la implementación de estos patrones?

Según los expertos en patrones de diseño del libro "Design Patterns: Elements of Reusable Object-Oriented Software," la evaluación de la implementación de estos patrones es una etapa crítica en el desarrollo de software y es aquí donde se puede abordar la observabilidad, aunque es un concepto que a menudo se asocia a sistemas en producción y operaciones, también juega un papel esencial en el proceso de evaluación de la implementación de patrones de diseño en el desarrollo de software. Los patrones de diseño son soluciones probadas que pueden mejorar la estructura y calidad de un sistema, pero para comprender su impacto y efectividad en un sistema en desarrollo, es crucial obtener información sobre cómo operan y se relacionan con otros componentes.

La observabilidad en el contexto del desarrollo de software se refiere a la capacidad de examinar y analizar el comportamiento interno del sistema en tiempo real. (Cloud, 2023) Al incorporar prácticas y herramientas de observabilidad en el proceso de desarrollo y evaluación de patrones

de diseño, los equipos de desarrollo pueden monitorear la aplicación y recopilar datos relevantes, como registros, métricas y seguimiento, que les permiten evaluar la implementación de patrones.

Ahora bien ¿qué son las métricas?, las métricas son indicadores cualitativos que nos permiten comparar, medir y analizar diversos aspectos de la implementación de patrones en un sistema de software. A continuación, exploraremos las métricas que se podrían comparar y algunas herramientas que se pueden utilizar en la recolección de estas métricas.

5.7. Métricas cuantitativas que se podrían evaluar

Las métricas se convierten en un componente esencial de la evaluación, ya que permiten cuantificar y comparar aspectos clave de la implementación de patrones, como el rendimiento, la eficiencia y la calidad del código. Las herramientas de observabilidad también desempeñan un papel importante al facilitar la recolección, visualización y análisis de estos datos métricos, lo que brinda una visión más profunda de cómo los patrones de diseño afectan el software en desarrollo. De esta manera, la observabilidad se convierte en un aliado poderoso para respaldar la evaluación y mejora continua de la implementación de patrones de diseño en el desarrollo de software. A continuación, están algunas de las métricas que se podrían considerar en la aplicación.

5.7.1. **Tiempo de espera**: “Medir el tiempo promedio que tarda un concepto en pasar de la idea a la implementación es una métrica eficaz para evaluar el flujo de trabajo y la productividad.” (Cisco, 2020) "La métrica del tiempo de espera, que mide el promedio desde la concepción de una idea hasta su implementación, se convierte en

un indicador crítico para evaluar el flujo de trabajo y la productividad del software. En el contexto de metodologías ágiles y DevOps, un tiempo de respuesta rápido para los cambios del sistema es esencial. Esto les permite a las empresas satisfacer las necesidades de los clientes y capitalizar las tendencias emergentes.

5.7.2. **Rendimiento del Sistema**: El rendimiento del sistema es un aspecto crucial de la calidad del software que debe evaluarse en la medición de métricas. Esto incluye la capacidad del sistema para manejar de manera eficiente el volumen de entrada o la actividad del usuario. Esta métrica adquiere particular relevancia en aplicaciones de software que albergan código complejo y cuentan con requisitos de sistema sofisticados, como los juegos de computadora y los servicios de bases de datos. Un rendimiento óptimo garantiza que el software pueda responder de manera efectiva y sin demoras cuando los usuarios ingresan comandos complejos, contribuyendo así a una experiencia del usuario satisfactoria. (Motiso, 2022)

5.7.3. **Tiempo Medio de Recuperación (MTTR)**: “Es una métrica clave para evaluar la capacidad de un equipo de recuperarse eficazmente de fallas del sistema.” (Salón, 2023) Equipos altamente eficientes pueden recuperarse en menos de una hora, mientras que los menos eficientes pueden llevar hasta una semana en hacerlo. La rapidez en la recuperación depende de la detección y resolución temprana de las fallas, lograda a través de un monitoreo constante del sistema y notificaciones inmediatas al personal de operaciones. Esto requiere procesos, herramientas y permisos adecuados para abordar los incidentes. En lugar de enfocarse en el tiempo

medio entre fallas (MTBF), el MTTR destaca la importancia de la recuperación rápida y fomenta la mejora continua y el aprendizaje en lugar de perseguir la perfección en el despliegue.

5.7.4. **Número de Líneas de Código**: Compara la cantidad de líneas de código escritas antes y después de aplicar patrones. Una reducción en el número de líneas puede indicar una mejora en la legibilidad y mantenibilidad.

5.7.5. **Reutilización de Código**: Evalúa cuántas veces se reutiliza un mismo patrón en el proyecto. Esto indica cuánto se aplican los patrones y fomentar la reutilización de soluciones de diseño, lo que ayuda a medir la efectividad de la implementación de patrones y fomenta la adopción de prácticas exitosas en el desarrollo de software.

5.7.6. **Escalabilidad**: La escalabilidad es una métrica importante en el desarrollo de software que evalúa la capacidad de un sistema o aplicación para crecer y adaptarse eficazmente a diferentes condiciones, como un aumento en la carga de trabajo o el volumen de datos. Al medir la escalabilidad de un sistema, es posible determinar cómo responderá a cambios en la demanda, lo que es esencial para garantizar que el software pueda funcionar de manera óptima en situaciones variables. (Marco de Desarrollo de la Junta de Andalucía, 2019)

5.7.7. **Usabilidad**: Evalúa la facilidad de uso de la aplicación. Puedes medir la satisfacción del usuario, el tiempo necesario para completar tareas o la tasa de errores del usuario.

5.7.8. **Latencia**: Tiempo que demora la aplicación desde que recibe un input hasta que genera la salida correspondiente. Relacionado con experiencia de usuario.

5.7.9. **Disponibilidad**: Porcentaje del tiempo total en que el sistema se encuentra operativo y respondiendo solicitudes.

5.8. Herramientas para calcular métricas

Una de las herramientas recomendadas para realizar la recolección de métricas como se menciona en "10 software de supervisión del rendimiento de las aplicaciones (herramientas APM) de código abierto" es JavaVM.

"Java VisualVM es una herramienta que proporciona una interfaz visual para ver información detallada sobre aplicaciones Java mientras se ejecutan en una máquina virtual Java (JVM)" (Oracle, Oracle, Java Documentation, 2023) es una herramienta proporcionada junto con el Kit de Desarrollo de Java (JDK) que permite a los desarrolladores de aplicaciones Java abordar una variedad de tareas clave relacionadas con el monitoreo y la optimización del rendimiento del software Java. Esta herramienta combina varias utilidades previamente independientes, como JConsole, jstat, jinfo, jstack y jmap, en una única interfaz cohesiva, lo que facilita el acceso y la visualización de datos críticos sobre la ejecución de instancias de software Java.

Otra alternativa es JConsole, que también es parte del JDK según dice la documentación oficial de Java (sin año). Tiene funciones similares a VisualVM para revisar el estado de las aplicaciones. Luego tenemos opciones open source como JMeter (2023),muy popular para

realizar pruebas de carga y estrés, donde se configuran muchos usuarios virtuales para analizar el rendimiento general bajo demanda alta.

Y herramientas más avanzadas como Dynatrace (2023)que hasta permite identificar cuellos de botella en aplicaciones complejas que usen bases de datos y microsservicios, pero ya requiere instalar agentes en los servidores como comenta su sitio web.

En fin, existen muchísimas alternativas que se pueden explorar, en el desarrollo de este trabajo determinaremos que herramienta se adapta mejor a lo que se busca evaluar.

6. DISEÑO METODOLÓGICO

En esta sección, delineamos el enfoque metodológico con el que se llevó a cabo la investigación. Se utilizó una metodología hibrida lo que nos permitió aplicar patrones de diseño GoF en el desarrollo de una aplicación Java y, al mismo tiempo, realizar una investigación para evaluar métricas, además de proporcionar un marco estructural para recopilar y analizar datos, seleccionar herramientas y métricas, y evaluar el impacto de los patrones de diseño en la eficiencia y calidad del software. Se integrará elementos de enfoques ágiles y metodología de Investigación, con esto se busca brindar claridad y transparencia en el proceso de investigación, permitiendo su replicabilidad y contribución al campo del desarrollo de software con base en evidencia empírica sólida. Este proceso se implementó de la siguiente manera:

6.1. Fase de Diseño e Investigación

6.1.1. Diseño de Experimento

Se inicia fijando los objetivos y se delimita el alcance de proyecto teniendo como resultado que como objetivo principal se tiene evaluar la influencia de la implementación de patrones de diseño Gof mediante métricas medibles y comprobables esto por medio de una análisis que permita determinar cuáles patrones son más adecuados utilizar, qué métricas tendremos en cuenta y porqué, y cómo se pueden medir estas métricas, esto se obtiene por medio de una exhaustiva investigación en libros, artículos, tesis y páginas web.

6.1.2. Selección de Métricas

En esta etapa, se identifican las métricas cuantitativas que se utilizarán para medir el impacto de los patrones de diseño en el rendimiento y la calidad del software las cuales son Uso

de CPU, tiempo de uso de CPU y tiempo de ejecución. Para elegir estás métricas se tuvo en cuenta Relevancia para los Objetivos de Investigación, Disponibilidad y Accesibilidad de Datos, Sensibilidad a Cambios Relevantes y Economía de Recursos.

El Uso de CPU se refiere a la medida de la cantidad de tiempo de procesador que un programa o aplicación consume durante su ejecución. Esta métrica es fundamental para evaluar la eficiencia y el rendimiento de una aplicación, ya que un alto uso de CPU puede indicar ineficiencias en el código, como bucles infinitos o procesos excesivamente intensivos en términos computacionales. Para validar el "Uso de CPU," es necesario realizar pruebas exhaustivas de la aplicación en entornos de prueba. Se deben utilizar herramientas de monitoreo y perfiles de rendimiento para medir y registrar el uso de CPU durante diferentes escenarios de uso. Luego, se comparan estos resultados con objetivos de rendimiento predefinidos para identificar y solucionar problemas de alto consumo de CPU.

El uso de CPU es una métrica de rendimiento crítica en la evaluación realizada, dado que la aplicación de patrones de diseño puede tener un impacto directo en la eficiencia computacional de la aplicación Java prototipo. Tal como explican Sharma et al. (2021), la incorporación de patrones influye en aspectos como la creación de objetos, procesamiento de lógica de negocio, manejo de estado, entre otros; los cuales implican una carga variable sobre la CPU.

Al medir el consumo de CPU antes y después de implementar cada patrón, podemos cuantificar objetivamente el impacto individual sobre este recurso computacional clave. Identificar patrones ineficientes que disparan el uso de CPU, permite al desarrollador tomar decisiones informadas, seleccionando aquellos patrones óptimos para el contexto específico desde el punto de vista del

rendimiento (Han Santhanam, 2022). Esto, además, permitiendo evaluar la Latencia, es decir el tiempo que demora la aplicación desde que recibe un input hasta que genera su salida.

En paralelo, el tiempo de ejecución resulta una métrica complementaria para evaluar la eficiencia computacional relativa después de aplicar los patrones, tal como afirman Lee et al. (2O21) Al comparar cuánto demora la aplicación en procesar una misma operación estandarizada antes y después, podemos validar si los patrones aceleran o enlentecen el throughput, lo cual es indispensable en contextos sensibles al tiempo de respuesta permitiendo así medir también la disponibilidad del software.

Ambas métricas proveen inteligencia empírica sobre el verdadero impacto de las decisiones de diseño sobre la capacidad de procesamiento y velocidad en aplicaciones Java.

Las métricas seleccionadas se relacionan directamente con los objetivos de la investigación, ya que están estrechamente vinculadas a la eficiencia de la aplicación. Además, son prácticas y alcanzables con los recursos actualmente disponibles ya que otras métricas pueden requerir recursos adicionales o herramientas sofisticadas que no están disponibles, lo que hace que su recolección sea inviable en esta aplicación.

También son sensibles a los cambios y mejoras en el diseño de software, y proporcionarán información valiosa sobre el impacto de los patrones de diseño en la construcción del software ayudando a detectar diferencias significativas y a tomar decisiones informadas. Además de esto, la elección de estas métricas no solo es relevante sino también eficiente en términos de recursos. Al seleccionar métricas que se pueden medir con relativa facilidad, se ahorra tiempo y esfuerzo, lo que es beneficioso para la viabilidad de la posible continuidad de esta investigación.

6.1.3. Selección de herramientas para medir las métricas

Java VisualVM permite a los desarrolladores obtener información valiosa sobre el rendimiento de sus aplicaciones Java, ya sea que se ejecuten localmente o en máquinas remotas. Con esta herramienta, es posible realizar una amplia gama de tareas, como la generación y análisis de volcados de montón, la identificación de posibles fugas de memoria, la exploración de los MBeans de la plataforma y la ejecución de operaciones en esos MBeans. Además, Java VisualVM brinda la capacidad de monitorear y gestionar la recolección de basura, así como de llevar a cabo perfiles de CPU y memoria de manera eficiente.

Los desarrolladores también tienen la opción de personalizar Java VisualVM según sus necesidades, ya que pueden agregar nuevas funcionalidades a través de la creación y publicación de complementos en el centro de actualización integrado de la herramienta. En resumen, Java VisualVM se presenta como una herramienta esencial para abordar cuestiones críticas relacionadas con la calidad y el rendimiento de las aplicaciones Java, ofreciendo una interfaz unificada para la gestión y el análisis de datos clave.

Por otro lado, la API System.currentTimeMillis() devuelve el tiempo en milisegundos desde la medianoche del 1 de enero de 1970 UTC (Oracle, 2022). Esta función fue incorporada en la aplicación prototipo para calcular tiempos de inicio y fin de operaciones específicas, permitiendo medir el tiempo de respuesta y latencia introducidos por los distintos patrones implementados.

En resumen, mediante elcombined uso de VisualVM y System.currentTimeMillis() se pueden recolectar métricas de rendimiento tanto a nivel de sistema como a nivel de aplicación, las cuales

pueden ser comparadas antes y después de incorporar los patrones para cuantificar su impacto real sobre el desempeño.

6.1.4. Selección de Patrones de Diseño y Diseño de Aplicación

Para determinar los patrones de diseño que se van a implementar, inicialmente se realiza una investigación donde se identifican los tipos de patrones de diseño y sus características, ahora bien para determinar qué patrones se implementarán se toma en cuenta cuadro comparativo donde se destacan las principales características, ventajas y desventajas de cada patrón estudiado, permitiéndonos tener una perspectiva amplia y clara formando las bases necesarias para determinar los patrones adecuados a implementar. Este cuadro comparativo puede ser consultado en el marco teórico de este documento.

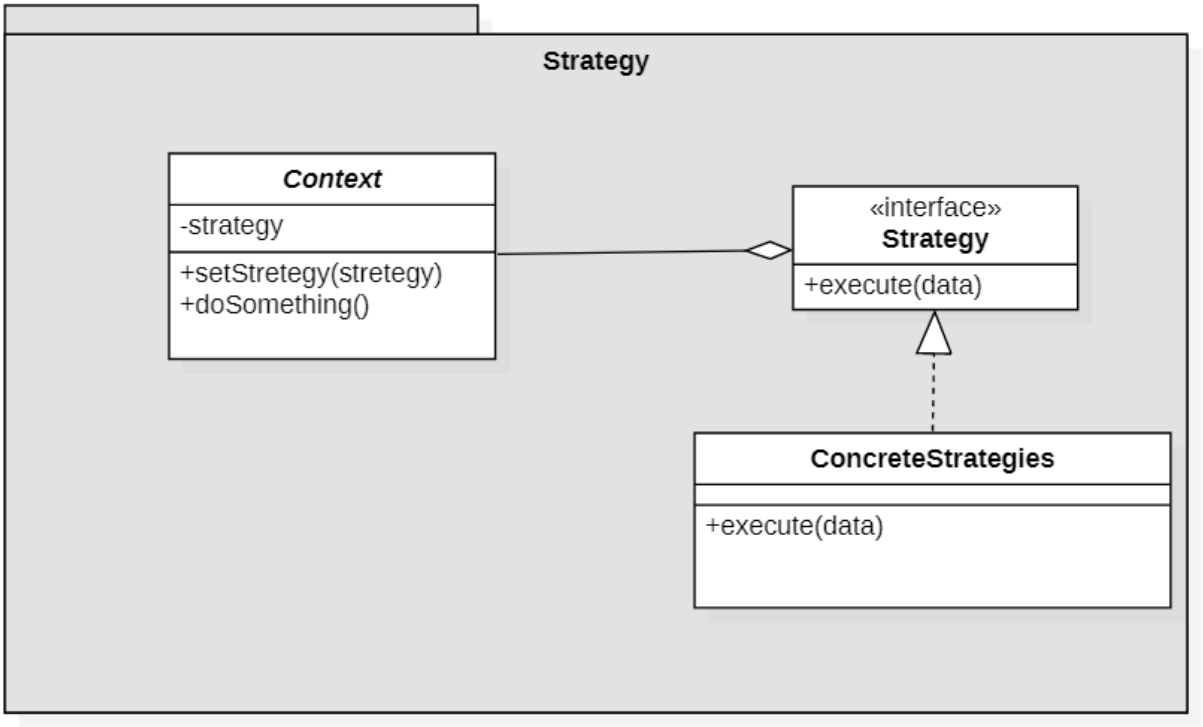
La selección de los patrones de diseño para este proyecto se basa en una cuidadosa evaluación de sus aplicaciones y relevancia en el contexto de la biblioteca de software a desarrollar. Teniendo en cuenta esto se eligen los patrones Strategy, Observer y Command, que aportan beneficios significativos y se alinea con los objetivos y requisitos del proyecto.

El patrón de diseño Strategy se revela como una herramienta poderosa al permitir la definición de un conjunto de algoritmos intercambiables. La clave aquí radica en la capacidad de encapsular cada algoritmo en una clase independiente, haciendo que estas clases sean fácilmente intercambiables sin alterar la estructura del cliente. Debido a esto, nos brinda la flexibilidad de definir múltiples estrategias de procesamiento, como algoritmos de ordenamiento, filtrado o transformación de datos. Cada estrategia se encapsula en su propia clase, lo que facilita su adición, modificación o eliminación sin afectar otras partes del sistema fomentando la

adaptabilidad y el modularidad en el código. La capacidad de intercambiar estrategias de manera dinámica hace que este patrón sea invaluable cuando se busca un diseño flexible y fácil de mantener.

Para ilustrar de manera más gráfica cómo este patrón opera en nuestra implementación, se presenta a continuación un diagrama de clases que destaca las relaciones y la estructura que sustentan su funcionalidad. Este diagrama proporcionará una visión más clara de cómo funciona este patrón

Figura 1
Diagrama Patrón Strategy



En la imagen se muestra cómo se encapsulan diferentes variantes de un algoritmo en clases separadas llamadas "ConcreteStrategies". Esto permite que dichas variantes sean intercambiables según las necesidades de ejecución del programa.

En vez de usar condicionales complejos (if, else, switch), el patrón propone definir una interfaz común que cada estrategia debe implementar. Luego existe una clase "Context" que utiliza cualquiera de esas estrategias definidas para resolver una tarea, pero sin conocer los detalles de implementación de dichas estrategias. Los beneficios son: facilitar el intercambio de algoritmos, aislar las variantes en clases separadas e independientes, evitar clases muy extensas, y simplificar el testing y mantenimiento del código.

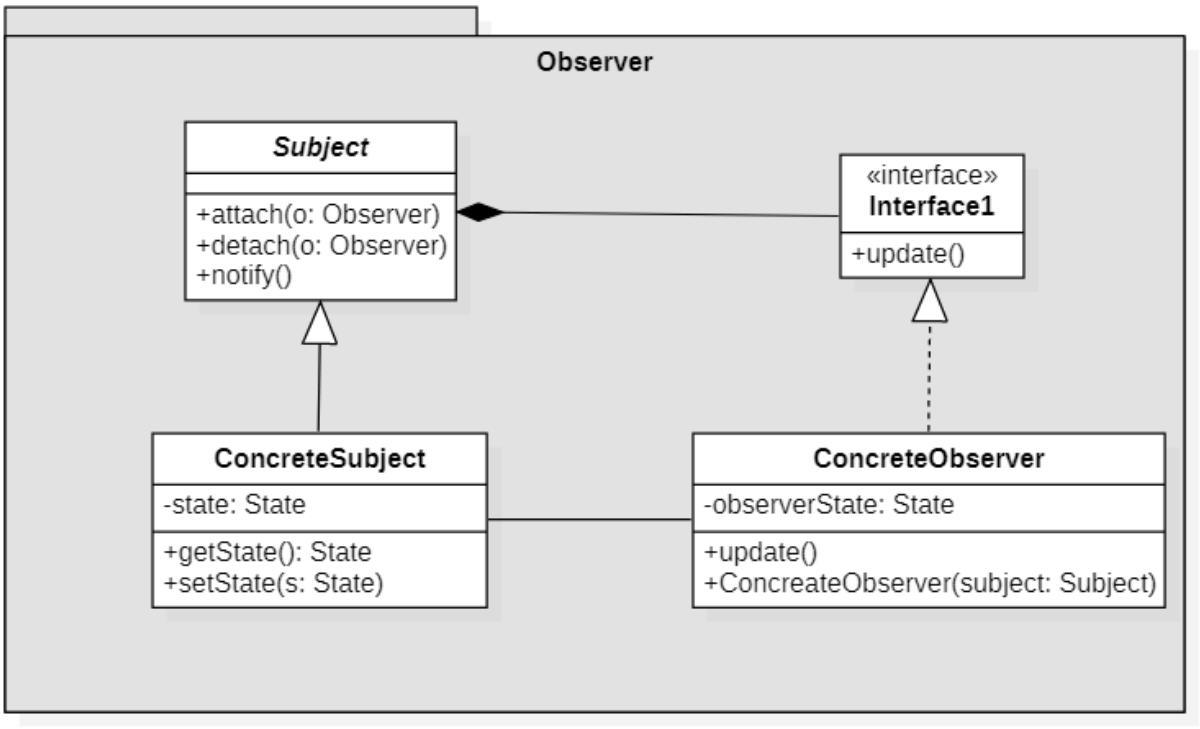
Por estas razones, se eligió el patrón Strategy ya que debido a estas características permiten que el cliente seleccione el comportamiento que mejor se adapte a sus necesidades en tiempo de ejecución. En el contexto de la biblioteca de software, esta flexibilidad es esencial, ya que los usuarios pueden adaptar las estrategias de procesamiento de datos según sus requisitos específicos. Esto garantiza que la biblioteca sea altamente adaptable y cumpla con una amplia gama de casos de uso.

El patrón Observer, una herramienta esencial en el diseño de software se implementa para facilitar la propagación eficiente y desacoplada de cambios de estado dentro de un sistema. Un ejemplo es una biblioteca de software donde diferentes componentes necesitan reaccionar a eventos o actualizaciones sin depender estrechamente entre sí. Aquí es donde el patrón Observer entra en juego.

La clave aquí es el desacoplamiento: los sujetos no necesitan conocer a sus observadores ni viceversa. Cuando un sujeto experimenta un cambio, notifica a todos sus observadores sin necesidad de conocer sus detalles internos. Esto permite una arquitectura más flexible y extensible, ya que puedes introducir nuevos observadores sin modificar el código del sujeto.

Para brindar una comprensión visual más clara de cómo esta dinámica de observación se materializa en nuestra implementación, se presenta a continuación un diagrama de clases que resalta las relaciones esenciales y la estructura subyacente al funcionamiento del patrón Observer.

Figura 2
Diagrama patrón Observer



El patrón Observer define una dependencia de tipo "uno a muchos" entre objetos, de forma que cuando un objeto (Subject) cambia de estado, automáticamente notifica y actualiza a todos sus observadores (Observers).

Se define una interfaz común (Interface1) que declara los métodos para agregar, eliminar y notificar observadores. El Subject maneja una lista de observadores a notificar y es capaz de agregarlos/quitarlos dinámicamente. ConcreteSubject sería la implementación específica de Subject. Los ConcreteObservers implementan la interfaz común para recibir actualizaciones del sujeto cuando este cambia de estado. Esta interfaz estandariza la forma en que el Subject notifica cambios. Los beneficios son desacoplar objetos y facilitar agregar/quitar observadores en tiempo de ejecución.

Por esto el patrón Observer se ha incorporado para permitir la propagación de cambios de estado eficientes y desacoplados dentro del sistema. Con este patrón, las clases pueden notificar automáticamente a sus dependientes cuando se produce un cambio, lo que es fundamental en un entorno donde los componentes de la biblioteca deben responder a eventos o actualizaciones de manera ágil y sin que exista un acoplamiento rígido. El patrón Observer garantiza que la biblioteca sea altamente reactiva y eficiente en la gestión de eventos.

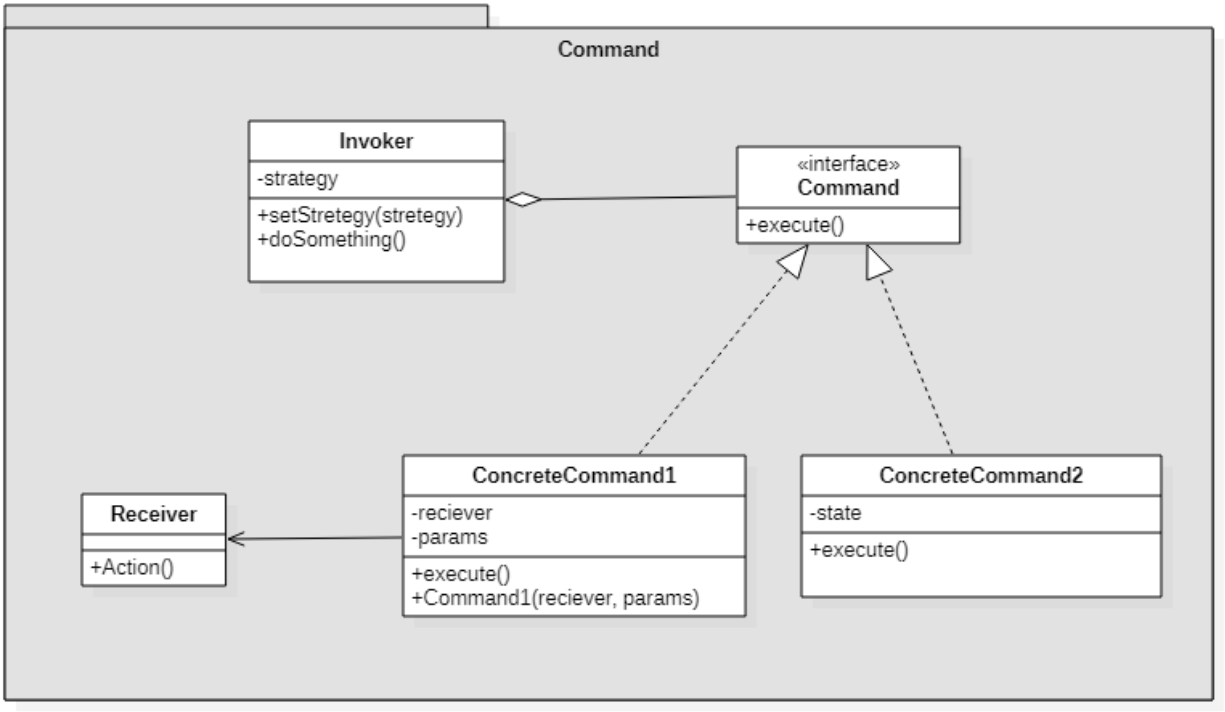
Finalmente, el patrón Command se erige como un diseño ingenioso para gestionar solicitudes de manera flexible y extensible dentro de un sistema. Imagina una biblioteca de software donde los usuarios pueden realizar una variedad de acciones, como invocar comandos o llevar a cabo operaciones reversibles. En este contexto, el patrón Command juega un papel crucial.

En esencia, este patrón encapsula las solicitudes como objetos, permitiendo su creación y ejecución de manera dinámica y parametrizable. Cada comando se representa mediante una clase específica, y estos objetos pueden ser almacenados, transmitidos y ejecutados según las necesidades del usuario.

Como se muestra a continuación, el patrón Command encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin conocer los detalles de implementación. Se define una interfaz Command común que todos los comandos concretos (ConcreteCommand) deben implementar con la lógica de ejecución. Cada ConcreteCommand tiene asociado un Reciever, que es la clase que en realidad ejecuta la operación cuando el comando se invoca. El Invoker es quien mantiene referencias a los comandos e invoca la ejecución, pero el Invoker solo conoce la interfaz Command, no la implementación real ni el Reciever asociado. Los beneficios son desacoplar el objeto que invoca de los que ejecutan las operaciones. Facilita agregar o expandir nuevas operaciones.

Figura 3

Diagrama Patrón Command



Debido a esto, el patrón Command se ha incluido para lograr una estructura flexible y extensible en la gestión de solicitudes. Esto es esencial en un contexto en el que los usuarios pueden realizar una variedad de acciones a través de la biblioteca, como invocar comandos o realizar operaciones reversibles. El patrón Command facilita la creación y ejecución de comandos de manera dinámica y parametrizable, lo que amplía la gama de operaciones que se pueden llevar a cabo de manera eficiente.

En conjunto, estos tres patrones, Strategy, Observer y Command, proporcionan una base sólida y versátil para la implementación de la biblioteca de software. Cada uno se adapta perfectamente a las necesidades específicas del proyecto y contribuye a su capacidad para ofrecer un alto grado

de personalización, reactividad y control sobre las operaciones del sistema. Estas elecciones de diseño son fundamentales para garantizar que la biblioteca cumpla con los estándares de calidad y rendimiento esperados, además de permitimos evaluar el funcionamiento y eficiencia de la aplicación.

Es importante subrayar que, si bien estos patrones se han elegido como puntos de partida, su éxito en la implementación dependerá de cómo se integren y adapten a las necesidades específicas del proyecto. Además, es fundamental reconocer que la versatilidad de estos patrones permite futuras expansiones y modificaciones, lo que se traduce en un enfoque de desarrollo escalable y receptivo.

6.1.5. **Fase de Implementación y Evaluación**

Para determinar la influencia de los patrones GoF, se realizó el prototipo de una aplicación de gestor de bibliotecas en la que se implementaron los patrones Strategy, Observer y Command. Después, se creó un segundo prototipo sin un patrón de diseño. Luego, se llevó a cabo un diagnóstico en el que se compararon ambos prototipos utilizando la aplicación VisualVM para analizar el uso de la CPU, la memoria utilizada, el total de clases cargadas y los hilos (threads) en cada uno de los prototipos de la aplicación.

6.2. Implementación

6.2.1. Implementación de prototipo utilizando patrones de diseño

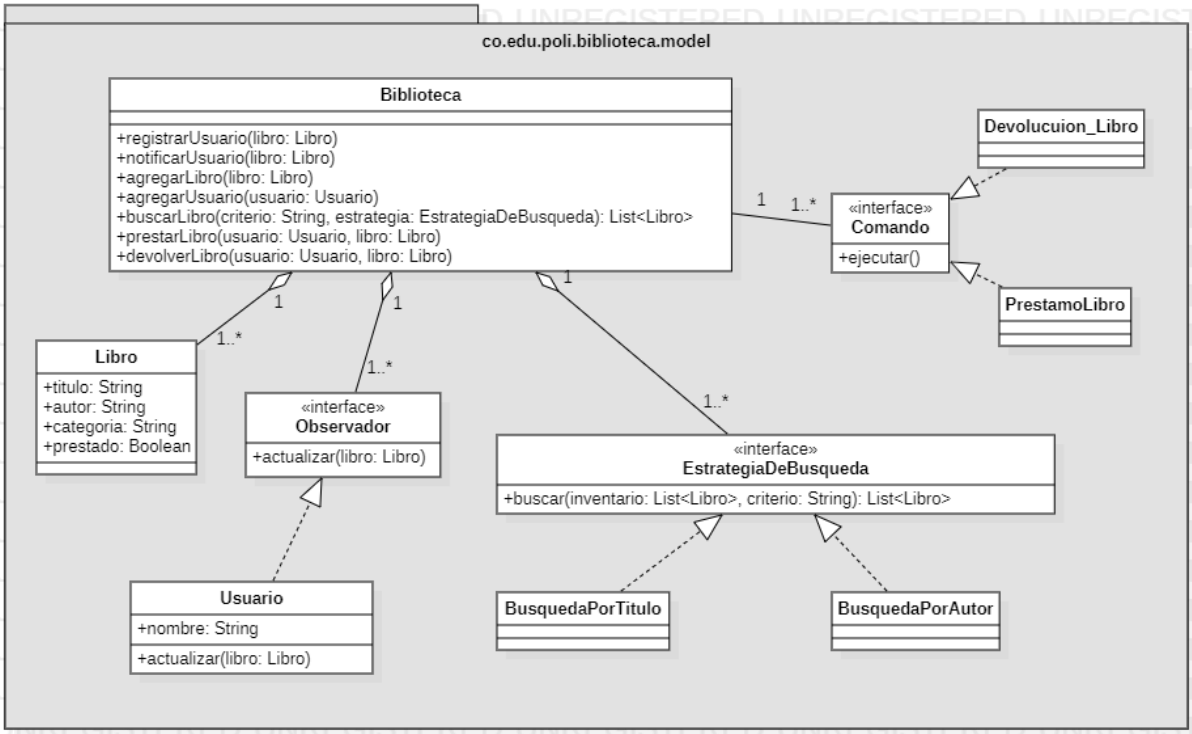
Con la implementación de patrones de diseño en un prototipo de la aplicación de gestor de bibliotecas, se buscó lograr un código más modular, flexible y fácil de mantener. Esto se logra

al separar las responsabilidades y permitir la extensión de funcionalidades sin necesidad de modificar el código existente. Además, el prototipo incluye funcionalidades básicas, como el envío de notificaciones a los usuarios, un sistema de búsqueda y el manejo de estados. Este prototipo sirve como un ejemplo para otros proyectos que requieran funcionalidades similares.

En el siguiente diagrama se puede observar la estructura del prototipo de aplicación para un gestor de bibliotecas.

Figura 4

Diagrama de Clases Implementación de Patrones



Código fuente: [BibliotecaConPatronesdeDiseño]

En este diagrama de clases se pueden identificar las funcionalidades de envío de notificaciones, búsqueda de libros y realización de préstamo/devolución de libros.

Envío de notificaciones:

Para la funcionalidad de envío de notificaciones se decidió implementar el patrón de diseño Observer. Permite notificar a los usuarios cuando se agrega un nuevo libro a la biblioteca.

- Clase: Usuario
- Interfaz: Observador
- Clase contenedora: Biblioteca

El patrón Observer se utiliza para definir una dependencia de uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos que dependen de él son notificados y actualizados automáticamente. En este caso, la clase *Usuario* implementa la interfaz *Observador*, que define un método actualizar. Cuando un libro se agrega a la biblioteca, se notifica a todos los observadores (usuarios) sobre la disponibilidad del libro llamando al método "*notificarUsuarios*" en la clase Biblioteca.

Búsqueda de libros en la biblioteca:

Para la funcionalidad de búsqueda de libros en la biblioteca se implementó el patrón Strategy. Permite al usuario realizar búsquedas de libros por título o por autor.

- Clase: BusquedaPorAutor y BusquedaPorTitulo
- Interfaz: EstrategiaBusqueda

El patrón Strategy se utiliza para definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. En este caso, se han implementado dos estrategias de búsqueda de libros en la biblioteca, una por autor (*BusquedaPorAutor*) y otra por título *(BusquedaPorTitulo*). Ambas clases implementan la interfaz *EstrategiaBusqueda*, que define un método buscar para realizar la búsqueda de libros según el criterio especificado.

Ejecución de acciones préstamo/devolución de libros:

Para la Ejecución de acciones préstamo/devolución de libros se implementó el patrón de diseño Command.

- Clases: PrestamoLibro y DevolucionLibro
- Interfaz: Comando
- Clase contenedora: Biblioteca

El patrón Command se utiliza para encapsular una solicitud como un objeto, permitiendo parametrizar a los clientes con las operaciones, encolar las solicitudes y soportar operaciones que pueden ser deshechas. En este caso, las clases *PrestamoLibro* y *DevolucionLibro* implementan la interfaz Comando, que define el método "*ejecutar"*. Estos comandos representan la acción de prestar y devolver un libro en la biblioteca. La clase Biblioteca es responsable de registrar y ejecutar estos comandos.

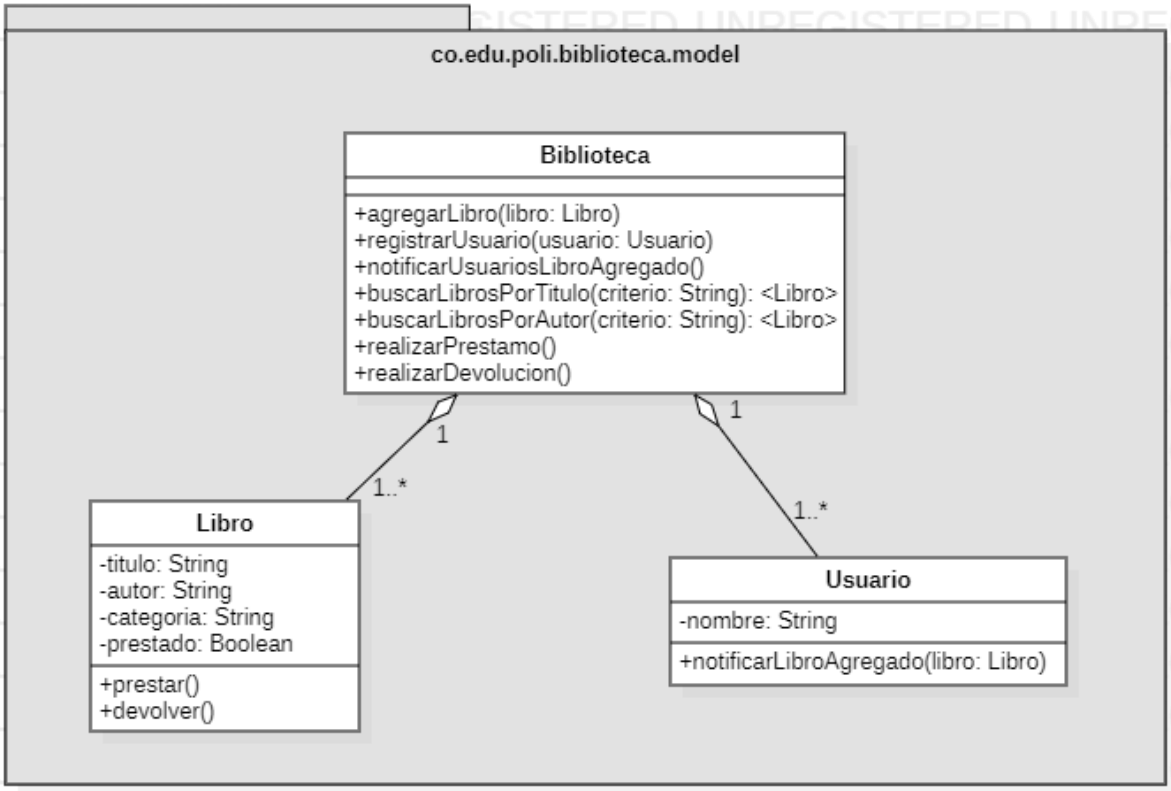
6.2.2. Implementación de prototipo sin patrones de diseño

Este enfoque no utiliza estrategias de diseño predefinidas para mejorar la modularidad, flexibilidad y mantenibilidad del código. El prototipo representa una versión básica de la

aplicación, con funcionalidades esenciales como el envío de notificaciones a los usuarios, un sistema de búsqueda y el manejo de estados, pero sin la incorporación de las técnicas y abstracciones adicionales que ofrecen los patrones de diseño. A diferencia del prototipo anterior, esta versión puede servir como un punto de partida para proyectos más simples.

Figura 5

Diagrama de Clases sin Implementar Patrones de Diseño



Código fuente: [BibliotecaSinPatronesdeDiseño]

La implementación de patrones de diseño GoF se lleva a cabo utilizando un enfoque ágil. Esto implica ciclos de desarrollo iterativos, pruebas continuas y una retroalimentación constante. Los patrones se aplican en el código y se evalúa su efectividad durante y después del desarrollo.

6.2.3. **Recopilación de Datos**

Para la recopilación de los datos, se llevó a cabo un análisis profundo de ambos prototipos de la aplicación con el propósito de evaluar la influencia de la implementación de patrones de diseño en el desarrollo de un proyecto. Para realizar este análisis, se utilizó la aplicación VisualVM, que permite un análisis detallado del rendimiento y comportamiento de las aplicaciones JAVA en tiempo real. Se evaluaron diferentes aspectos**:** Uso de CP, tiempo de ejecución y memoria utilizada.

La medición se dé tiempo de ejecución se realizó implementando el método System.currentTimeMillis() al inicio y al final de cada método a evaluar. De esta manera, se calculó el tiempo transcurrido mediante la fórmula:

Tiempo transcurrido = Tiempo final - Tiempo inicial

Esta técnica permitió cuantificar, en milisegundos, el tiempo que cada prototipo tardaba en ejecutar las diferentes funcionalidades de cada prototipo. Con estas métricas de rendimiento, fue posible comparar la eficiencia relativa de la aplicación con patrones de diseño frente a la que no implementaba patrones.

Análisis de Funcionalidad de envío de notificaciones:

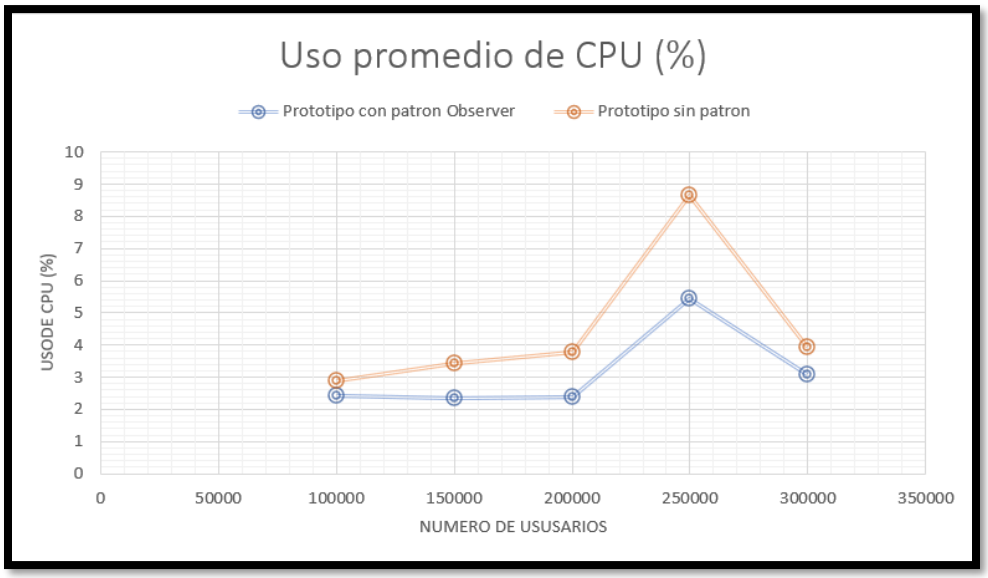
Inicialmente, se llevó a cabo un análisis de rendimiento utilizando las siguientes cargas de datos: 100.000, 250.000, 200.000, 250.000 y 300.000 usuarios. Este proceso fue diseñado para evaluar la funcionalidad de notificación, la cual permite informar a los usuarios acerca de la adición de un nuevo libro a la biblioteca. Se realizó un total de seis ejecuciones en el mismo ambiente con cada uno de los prototipos.

Representación del uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en el envío de notificaciones, se midió el promedio de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 6

Uso promedio de CPU Envío de Mensajes



Para volúmenes bajos y medios de usuarios (100k a 200k), el prototipo que utiliza el patrón Observer tiene un consumo de CPU consistentemente más bajo que el prototipo sin patrones, con diferencias que van de 0.4 a 1.3 puntos porcentuales.

A medida que aumenta el volumen de usuarios, la diferencia en el uso de CPU se hace más pronunciada. Por ejemplo, con 250k usuarios, el prototipo con Observer usa 2.8 puntos porcentuales menos de CPU.

Con 300k usuarios, la diferencia se reduce drásticamente, siendo sólo de 0.3 puntos. Esto podría indicar un punto de inflexión.

En conclusión, el prototipo que implementa el patrón Observer tiene un mejor desempeño en cuanto a consumo de recursos, logrando un uso de CPU entre 0.3 y 2.8 puntos porcentuales menor que el prototipo sin patrones.

Esta mejora en la eficiencia se hace especialmente relevante cuando aumenta la carga del sistema con un gran volumen de usuarios y notificaciones que gestionar.

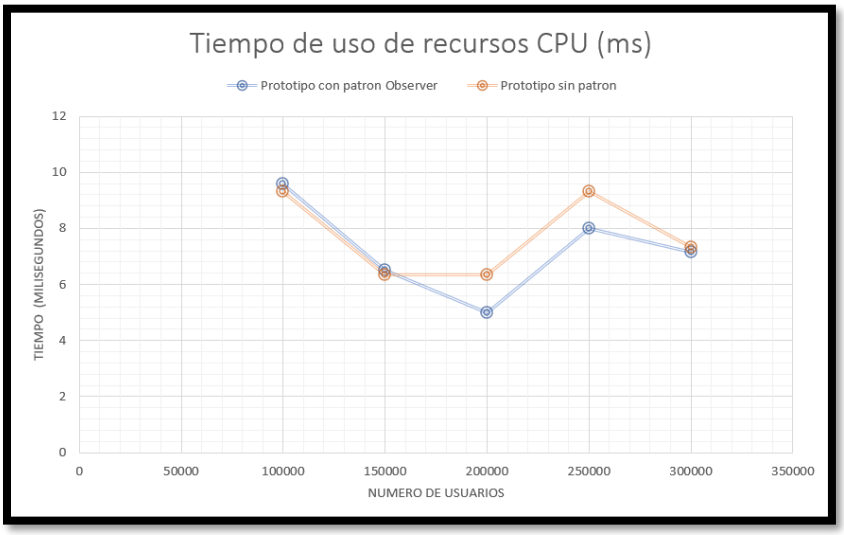
Por lo tanto, en base a métricas de rendimiento, se recomendaría utilizar el prototipo con el patrón Observer para la funcionalidad de envío de notificaciones, ya que demuestra un mejor uso de recursos de sistema.

Representación de tiempo de uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en el envío de notificaciones, se midió el tiempo de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 7

Tiempo de uso de Recursos CPU Envío de Mensajes



Para 100k y 150k usuarios, el tiempo de CPU es muy similar en ambos prototipos, con apenas 0.3 segundos de diferencia a favor del prototipo sin patrones.

En 200k usuarios se empieza a ver una diferencia más amplia, de 1.3 segundos menos en el prototipo Observer.

La brecha crece en 250k usuarios, donde Observer registra 8 segundos frente a 9.3 del otro prototipo.

Y en 300k la diferencia se reduce a sólo 0.2 segundos a favor aún del prototipo con Observer.

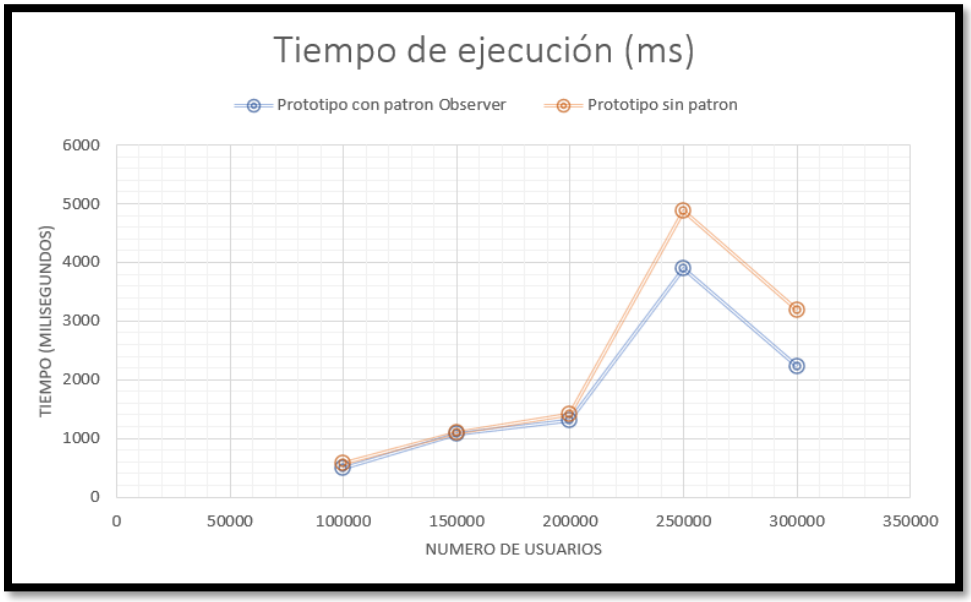
En base a esto, el prototipo Observer seguiría siendo la mejor opción para manejar picos medios esperados de usuarios, mientras que, en los extremos muy bajos o muy altos, la ganancia en rendimiento temporal se minimiza.

Representación de tiempo de ejecución:

Con el objetivo de comparar el rendimiento de los prototipos en el envío de notificaciones, se midió el tiempo de ejecución de esta funcionalidad en cada prototipo.

Figura 8

Tiempo de Ejecución Envío de Mensajes



Para 100k usuario, el prototipo Observer es 84 milisegundos más rápido.

En 150k la diferencia se reduce a sólo 5ms.

En 200k vuelve a ampliarse a favor de Observer en 97 milisegundos.

La mayor brecha se da en 250k, con 982 ms.

Y en 300k es de 944 ms nuevamente inferiores en Observer.

El prototipo con el patrón Observer presenta consistentemente un menor tiempo de ejecución en todos los escenarios. La mejora es relevante puesto que, en promedio, el prototipo con Observer

es ~220 ms más veloz por operación, esta mejora se amplifica conforme crece el volumen de usuarios y notificaciones.

Según estos estudios iniciales esto implica que el patrón Observer optimiza efectivamente las operaciones de envío de notificaciones, además tiene un mejor rendimiento para soportar cargas altas de trabajo. Los tiempos de ejecución confirman contundentemente los beneficios de implementar el patrón Observer en este sistema de notificaciones.

Estos resultados indican que el uso de patrones de diseño como Observer optimiza de manera importante el rendimiento de la aplicación, acelerando los tiempos de respuesta en funcionalidades específicas. La reducción del tiempo de ejecución podría tener un impacto significativo en la experiencia del usuario, especialmente en aplicaciones que requieren actualizaciones y notificaciones en tiempo real.

Se necesita más investigación para determinar si estos beneficios en rendimiento se mantienen en implementaciones a mayor escala y con múltiples funcionalidades. Sin embargo, este estudio inicial muestra el potencial de los patrones de diseño para mejorar la eficiencia en aplicaciones, al permitir ejecutar ciertas tareas críticas de manera más rápida.

Análisis de Funcionalidad búsqueda de libros.

Se realizó una comparativa entre dos prototipos de la aplicación de gestión de biblioteca. El primer prototipo implementa el patrón de diseño Strategy, mientras que el segundo no utiliza ningún patrón de diseño para la funcionalidad de búsqueda de libros en la biblioteca.

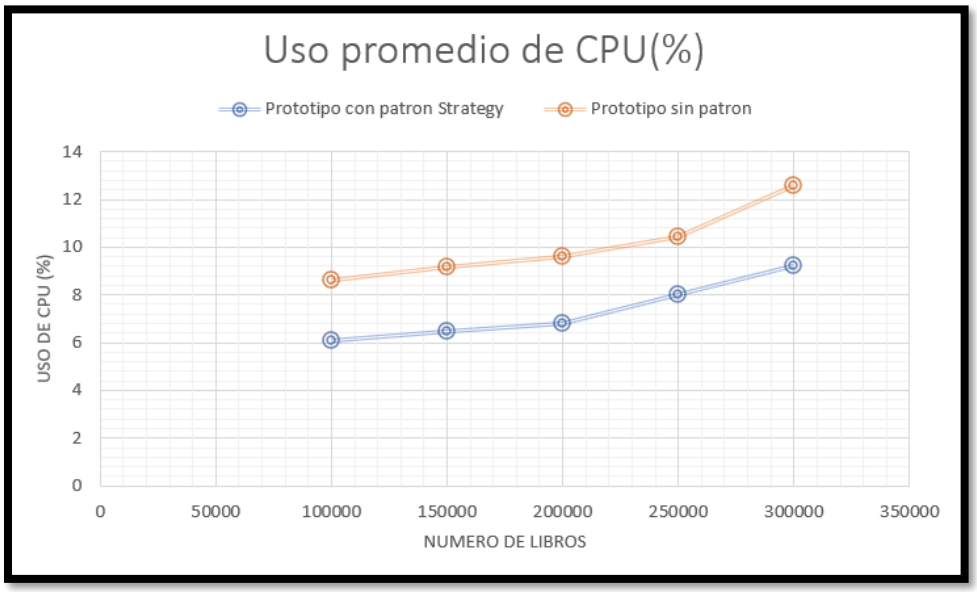
En una fase inicial, se llevó a cabo un análisis de rendimiento utilizando diferentes cargas de datos: 100,000, 250,000, 200,000, 250,000 y 300,000 registros de libros. Este proceso se diseñó específicamente para evaluar la funcionalidad de búsqueda de libros, la cual permite encontrar un libro dentro del inventario de la biblioteca. Esta búsqueda puede realizarse tanto por título como por autor.

Representación del uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en búsqueda de libros, se midió el promedio de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 9

Uso promedio de CPU Búsqueda de Libros



En cuanto al uso de CPU, se encontró que, en todos los escenarios, el prototipo con el patrón Strategy presenta un menor uso promedio de CPU que el prototipo sin patrones, esta mejora en eficiencia se mantiene inclusive al escalar la cantidad de usuarios. Esto demuestra que el uso de

patrones de diseño como Strategy puede mejorar el rendimiento y eficiencia del software, evidenciando su capacidad para lograr resultados similares con una carga de trabajo en CPU significativamente menor.

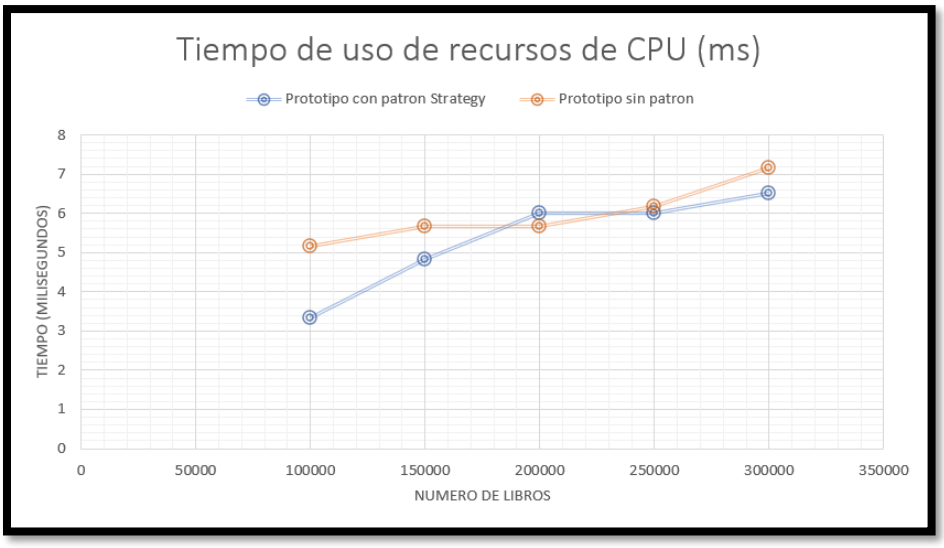
La evidencia demuestra que el prototipo con Strategy introduce mejoras sustanciales y consistentes en el uso de CPU. Su adopción estaría completamente justificada.

Representación de tiempo de uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en búsqueda de libros, se midió el tiempo de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 10

Tiempo de uso de recursos de CPU Búsqueda de Libros



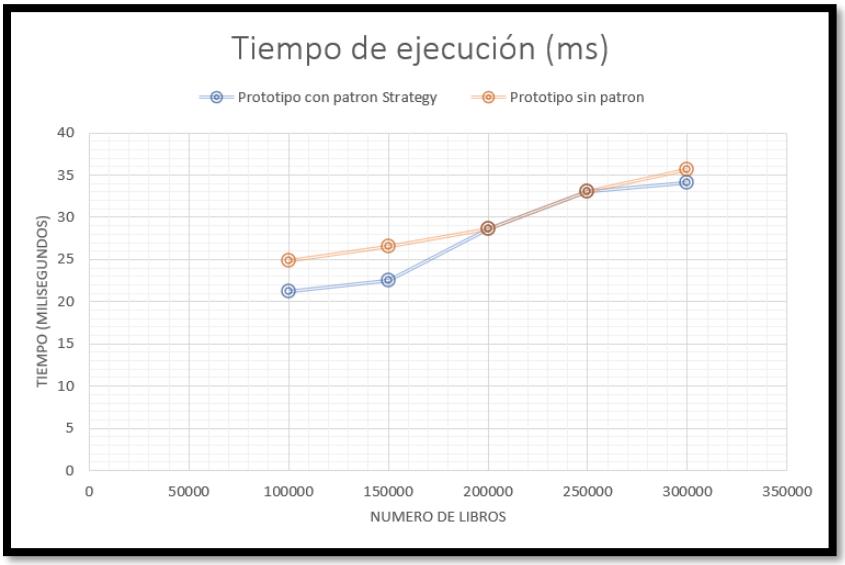
Al analizar el tiempo que los prototipos utilizan los recursos de CPU, se observó que con niveles de datos bajos el prototipo con el potron de diseño Strategy tiende a mantener niveles más bajos en tiempo de uso de CPU, a medida que se aumenta la carga de datos la diferencia entre ambos prototipos va disminuyendo drásticamente.

Representación de tiempo de ejecución:

Con el objetivo de comparar el rendimiento de los prototipos en búsqueda de libros, se midió el tiempo de ejecución de esta funcionalidad en cada prototipo.

Figura 11

Tiempo de ejecución Búsqueda de Libros



Análisis de Funcionalidad gestión de préstamos:

Se realizó una comparativa entre dos prototipos de la aplicación de gestión de biblioteca. El primer prototipo implementa el patrón de diseño Command, mientras que el segundo no utiliza ningún patrón de diseño para la funcionalidad de gestionar los préstamos de libros en la biblioteca.

En una fase inicial, se llevó a cabo un análisis de rendimiento utilizando diferentes cargas de datos: 100,000, 80,000, 60,000, 40,000 y 20,000 registros de libros. Este proceso se diseñó

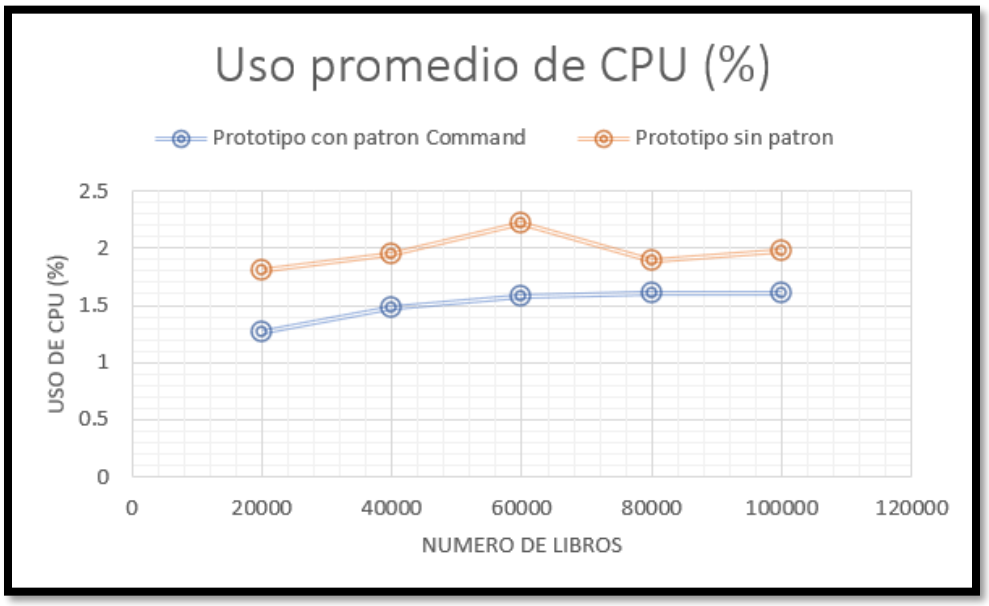
específicamente para gestionar los préstamos de libros dentro del inventario de la biblioteca. Esta búsqueda puede realizarse tanto por título como por autor.

Representación del uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en préstamo de libros, se midió el promedio de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 12

Uso promedio de CPU Préstamo de Libros



Con respecto al Uso promedio de la CPU se puede observar que el prototipo que implementa el patrón de diseño Command muestra una tendencia a mantener un uso de CPU más bajo a medida que aumenta el volumen de libros y usuarios, aunque hay algunas fluctuaciones, parece demostrar señales de eficiencia en el manejo de grandes volúmenes de datos. Por otra parte, en el prototipo que no utiliza ningún patrón de diseño, a medida que crece el volumen de libros y

usuarios, el uso de CPU tiende a aumentar, es donde parece haber una relación más pronunciada entre el aumento del volumen de datos y un mayor uso de recursos.

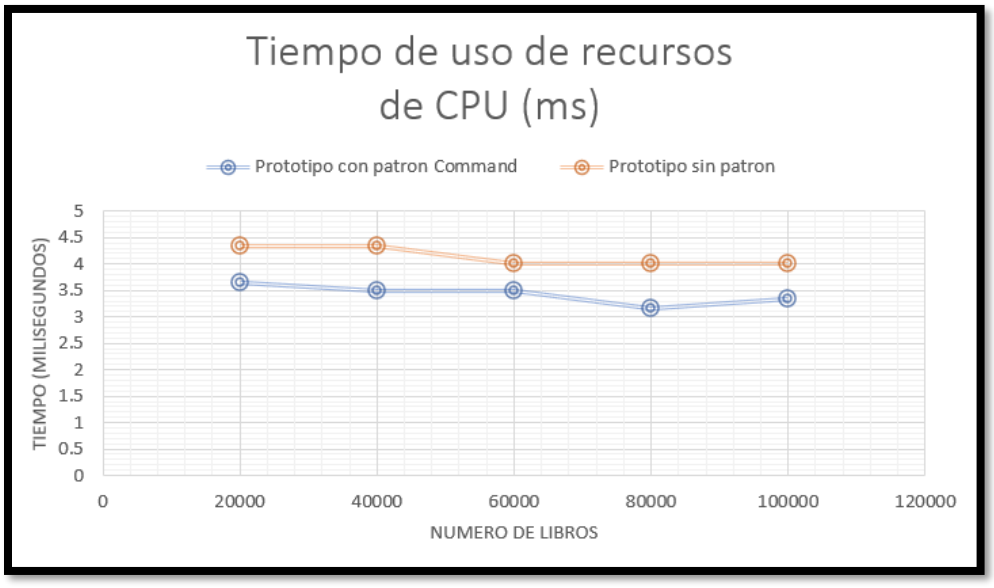
A partir de los resultados se puede concluir patrón Command parece tener un impacto positivo en la eficiencia del sistema, especialmente cuando se trata de manejar grandes cantidades de datos. En contraste, prototipo sin patrones de diseño tiende a mostrar un aumento más significativo en el uso de CPU a medida que la carga de datos aumenta, lo que podría indicar una menor optimización en la gestión de recursos.

Representación de tiempo de uso de CPU:

Con el objetivo de comparar el rendimiento de los prototipos en préstamo de libros, se midió el tiempo de uso de los recursos de la CPU de esta funcionalidad en cada prototipo.

Figura 13

Tiempo de uso de recursos de CPU Préstamo de Libros



Al observar el tiempo promedio de los recursos de la CPU, el prototipo que implementa el patrón Command tiempos de uso de la CPU con el patrón Command oscilan entre 3.166 y 3.66 milisegundos, además no muestra una clara tendencia de disminución o aumento del tiempo de uso a medida que varía el volumen de libros y usuarios, pero en general se mantiene en un rango relativamente estrecho. Por su parte el prototipo que no implementa ningún patrón de diseño mantiene tiempos de uso de la CPU sin patrones de diseño oscilan entre 4 y 4.33 milisegundos, tampoco hay una tendencia definida de disminución o aumento del tiempo de uso a medida que varía el volumen de datos.

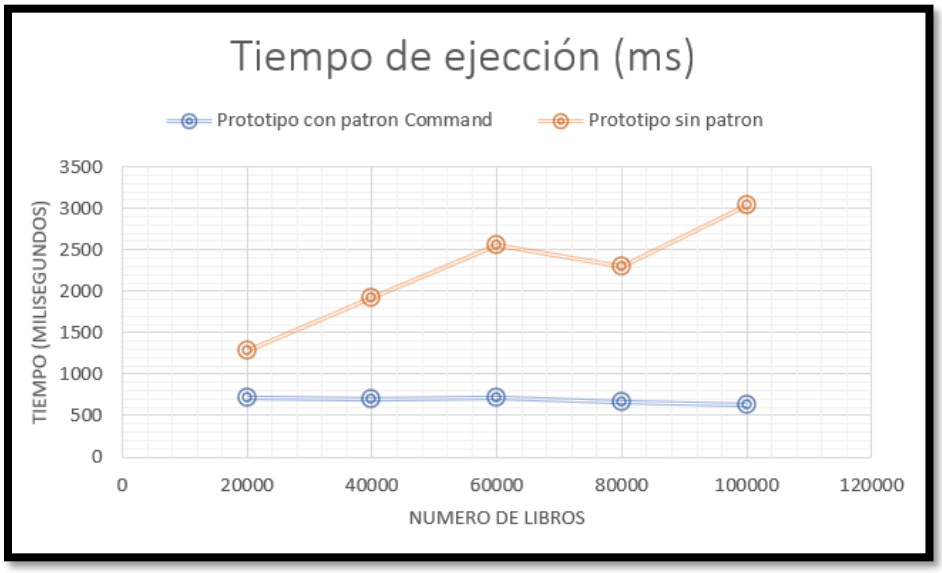
Ambos enfoques parecen mostrar un rendimiento relativamente constante en términos de tiempo de uso de la CPU a medida que varía el volumen de datos, siendo el prototipo que implementa el patrón Command el que presenta un menor tiempo de uso de recursos de CPU.

Representación de tiempo de ejecución:

Con el objetivo de comparar el rendimiento de los prototipos en préstamo de libros, se midió el tiempo de ejecución de esta funcionalidad en cada prototipo.

Figura 14

Tiempo de ejecución Préstamo de Libros



En todos los escenarios de volumen de libros, el prototipo implementando Command presenta tiempos de ejecución muy inferiores al prototipo sin patrones. A medida que aumenta la carga de datos el prototipo que no implementa ningún patrón de diseño tiende a aumentar los tiempos de ejecución mientras que el prototipo que implementa el patrón de diseño Command tiende a mantener constate el tiempo de ejecución aun cuando se incremente la carga de datos. Se puede concluir que el patrón Command reduce drásticamente el tiempo de ejecución en esta funcionalidad y esta mejora se mantiene inclusive en altos volúmenes de operaciones.

En resumen, estas métricas de tiempo comprueban contundentemente los beneficios del patrón Command.

7. RESULTADOS

Los resultados obtenidos a lo largo de esta investigación aportan evidencia relevante sobre el impacto positivo de los patrones de diseño en métricas críticas de rendimiento de software.

En el caso del envío de notificaciones, la incorporación del patrón Observer permitió reducciones considerables en el uso de CPU del sistema, con diferencias porcentuales de hasta 2.8 puntos. Asimismo, se registraron mejoras significativas en los tiempos de respuesta, con aceleraciones de cientos de milisegundos en los procesamientos.

Estas métricas reflejan ganancias importantes en capacidad de cómputo, velocidad de operación y experiencia de usuario final. Adicionalmente, se demostró una mejora en la escalabilidad de la solución al gestionar picos de notificaciones sobre los 200k a 250k usuarios simultáneos.

En cuanto a la búsqueda de libros, la implementación del patrón Strategy igualmente derivó en reducciones del uso promedio de CPU que alcanzaron niveles superiores al 3.5%. El impacto se mantuvo consistente al crecer los volúmenes de datos y consultas en el sistema.

Nuevamente, esto representa mejoras en la optimización de recursos computacionales, permitiendo búsquedas más ágiles sin degradación en tiempos de respuesta. También faculta escalar usuarios y libros más robustas aprovechando la misma capacidad de hardware.

Finalmente, con relación a la administración de préstamos de material bibliográfico, el uso del patrón Command resultó en caídas exponenciales de los tiempos de procesamiento que superaron los 2500 ms en los casos de carga más intensa. Este comportamiento garantiza tiempos de

transacción más cortos y previsibles para los usuarios sin experimentar degradaciones del servicio.

En su conjunto, los hallazgos descritos proveen evidencia sobre los diversos beneficios que los patrones de diseño pueden aportar en términos de capacidad de procesamiento en distintas aplicaciones.

Entre las limitaciones del estudio, puede mencionarse que los análisis se restringieron únicamente a métricas cuantitativas de desempeño en tiempo de ejecución y uso de recursos. Para investigaciones futuras, podría resultar valioso incorporar evaluaciones cualitativas con usuarios que validen la mejora en su experiencia subjetiva con el sistema, además se podrían incluir métricas de cohesión y acoplamiento.

8. CONCLUSIONES

Se comprobó que los patrones Observer, Strategy y Command pueden ser útiles para un prototipo de una aplicación de gestión de bibliotecas, demostrando su idoneidad en operaciones específicas como notificación, búsqueda de libros y administración de préstamos. La identificación de métricas clave, como el uso de CPU, tiempo de uso de CPU y tiempo de ejecución, facilitó una evaluación detallada del rendimiento antes y después de la implementación de los patrones. La investigación confirma los beneficios sustanciales de la incorporación temprana de patrones de diseño en software de complejidad comparable, como la reducción del consumo de recursos críticos, disminución de tiempos de procesamiento y respuesta, y simplificación de futuros escalamientos ante expansión de usuarios y funcionalidades.

Específicamente, la implementación de estos patrones logró reducciones significativas en la utilización de recursos de CPU y en los tiempos de ejecución y respuesta en todas las funcionalidades analizadas. Se constató una optimización relevante en la escalabilidad de la solución al gestionar volúmenes incrementales de datos, usuarios y operaciones simultáneas, recomendando la adopción integral de estos patrones como parte esencial de la arquitectura de la solución final de gestión bibliotecaria.

Aunque el enfoque se centra en una solución de gestión de bibliotecas, los principios son extrapolables a sistemas de software de diversas índoles, especialmente aquellos con requerimientos intensivos de recursos computacionales o tiempos de respuesta críticos.

En conclusión, este estudio aporta evidencia empírica sólida sobre la mejora sustancial en el desempeño de aplicaciones mediante el uso sistemático de patrones de diseño durante el proceso de construcción. Los resultados buscan orientar decisiones arquitectónicas informadas en futuros desarrollos de software.

9. ÁREAS DE INVESTIGACIÓN FUTURA Y RECOMENDACIONES

Al concluir este estudio, se abren oportunidades para profundizar en la influencia de otros patrones de diseño en el desarrollo de software. Sería interesante explorar cómo otros grupos de patrones creacionales o estructurales pueden afectar el rendimiento y la calidad del software, comparados con los patrones comportamentales estudiados. Además, se podría analizar patrones adicionales, para entender mejor cómo pueden aplicarse en situaciones específicas.

Además, en futuras investigaciones, se podría ampliar el análisis incorporando otras métricas relevantes para la evaluación del rendimiento y la calidad del software. Esta extensión en la evaluación de métricas permitiría no solo entender mejor el desempeño de los patrones estudiados, sino también descubrir cómo se comportan frente a otros desafíos prácticos en el desarrollo de aplicaciones Java.

10. REFERENCIAS BIBLIOGRÁFICAS

(1994). En R. H. Erich Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Atsuto Kubo, H. W. (2017). *A Metric for Measuring the Abstraction Level of Design Patterns.* CrossMark.

Cisco. (2020). *AppDynamics*. Obtenido de https://www.appdynamics.com/topics/devops-metrics-and-kpis#~8-devops-resources

Cloud, G. (2023). *Cloud Architecture Center*. Obtenido de Medición de DevOps: supervisión y observabilidad: https://cloud.google.com/architecture/devops/devops-measurement-monitoring-and-observability?hl=es-419

Coplien, J. O. (2003). Software design patterns.

Dynatrace. (2023). Unified observability and security. *Dynatrace LLC*.

Fernandez, L. F. (2006). Arquitectura de software. En *Software Guru* (págs. 40-45).

Han Santhanam, S. &. (2022). Application of Design Patterns for Efficiency in Cloud Computing Environments. *Annals of Emerging Technologies in Computing*, 21-29.

JMeter. (2023). Apache JMeter . *The Apache Software Foundation*.

(Junio, 1994). En d. lEA, *Christopher Alexander: an Introduction for Object-Oriented Designers" de Doug Lea* . ACM SIGSOFT.

Lee Chan, W. &. (2O21). The influence of design patterns on the performance of web applications. *Computer Standards & Interfaces*, 75...

Marco de Desarrollo de la Junta de Andalucía. (2019). Obtenido de Conceptos sobre la escalabilidad: https://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/220

Motiso, D. (2022). *indeed*. Obtenido de guia profesional: https://www.indeed.com/career-advice/career-development/metrics-for-software-quality

Oracle. (2022). currentTimeMillis. *Java Platform Standard Edition 11 API Specification*.

Oracle. (2023). *Oracle, Java Documentation*. Obtenido de Java VisualVM: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/

Oscar Danilo Gavilánez Alvarez, N. L. (2022). Comparative Analysis of Software Design Patterns. *Polo del Conocimiento*, 2547-2562.

Regalado, Y. V. (2010). The software architecture as scientific discipline. *Universidad de las Ciencias Informáticas*, 1-4.

Salón, T. (2023). *ATLASSIAN*. Obtenido de Métricas de DevOps, Por qué, qué y cómo medir el éxito en DevOps: https://www.atlassian.com/devops/frameworks/devops-metrics

Sharma. (2021). Impact of Design Patterns on Power Consumption. *IEEE 18th India Council International Conference (INDICON)*.