UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Inteligencia Artificial Generativa aplicada al diseño de Software.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Haro Limiñana, Iván

Tutor/a: Pelechano Ferragud, Vicente

Director/a Experimental: Mestre Gascón, Antoni

CURSO ACADÉMICO: 2023/2024

Resumen

Este Trabajo Fin de Grado aborda la integración de la Inteligencia Artificial Generativa en la fase de diseño de software. Se explora cómo los LLMs (Large Language Models), como ChatGPT y Gemini, pueden ser utilizados eficientemente en esta fase, identificando buenas prácticas en el uso de LLMs. A partir de los hallazagos, se desarrolla un GPT (Generative Pre-training Transformer) especializado, que puede proporcionar asistencia en la recomendación de patrones de diseño y su posterior implementación.

El objetivo de este TFG es mejorar tanto la productividad como la calidad de los resultados en la fase de diseño de software. Para ello, se creará un proceso o metodología que permita aplicar la IA generativa, en concreto los LLMs, al ámbito del diseño de software. Esta metodología proporcionará un marco estructurado para integrar LLMs en el proceso de diseño, asegurando que se aprovechen al máximo las capacidades de los modelos de lenguaje.

Además, este trabajo busca reforzar la figura del ingeniero de software, cuya relevancia se ha visto cuestionada por los recientes avances en IA. Este respaldo se pretende lograr mediante la continua necesidad de validar los distintos resultados proporcionados por los LLMs, asegurando así la supervisión y calidad del trabajo realizado.

Palabras clave: Ingeniería, Software, Diseño, Patrones, Refactorización, Inteligencia Artificial, LLM, GPT, Gemini

Abstract

This Bachelor's Thesis addresses the integration of Generative Artificial Intelligence in the software design phase. It explores how Large Language Models (LLMs), such as ChatGPT and Gemini, can be efficiently utilized in this phase, identifying best practices in the use of LLMs. Based on the results, a specialized Generative Pre-training Transformer (GPT) is developed, which can provide assistance in recommending design patterns and their subsequent implementation.

The objective of this thesis is to enhance both productivity and the quality of outcomes in the software design phase. For this purpose, a methodology will be created to apply generative AI, specifically LLMs, to the field of software design. This methodology will provide a structured framework for integrating LLMs into the design process, ensuring that the capabilities of language models are fully leveraged.

Additionally, this work aims to reinforce the role of software engineers, whose relevance has been questioned by recent advances in AI. This support is intended to be achieved through the continuous need to validate the various results provided by LLMs, thus ensuring the supervision and quality of the work performed.

Key words: Engineering, Software, Design, Patterns, Refactoring, Artificial Intelligence, LLM, GPT, Gemini

Índice general

Apéndices

CAPÍTULO 1
Introducción

El diseño de software es uno de los ámbitos tecnológicos más complejos y abstractos del siglo XXI. Se fundamenta en la toma de decisiones, en la comprensión y el análisis de problemas no triviales y en muchas más competencias, abarcando áreas dentro y fuera de la ingeniería del software. Este cometido requiere un conocimiento de lo más amplio: arquitecturas de software, patrones de diseño, paradigmas de programación y ciertamente años de experiencia.

Con el fin de mejorar los procesos y los resultados en este campo, se ha decidido emplear la Inteligencia Artificial Generativa utilizando los Large Language Models o LLMs. Estos modelos de lenguaje natural, entrenados con grandes volúmenes de datos, se utilizarán para mejorar las competencias referentes a la materia, proporcionando información útil que podrá servir de intuición o guía a la hora de crear e implementar soluciones que resuelvan los problemas inherentes al diseño de software.

Se pondrá énfasis en cómo los ingenieros de software deben adaptarse a este nuevo paradigma, con el objetivo de reducir la carga de trabajo, optimizar y mejorar procesos, aumentar la productividad y centrar el foco en las tareas más relevantes. Todos estos beneficios están directamente relacionados con la capacidad que tengamos para adaptarnos a lo que ofrece la Inteligencia Artificial, siendo aun así conscientes de que todo el peso del desarrollo seguirá cayendo sobre los hombros de los ingenieros.

1.1 Motivación

Los LLMs representan la punta de lanza del campo de la Inteligencia Artificial Generativa, siendo los modelos desarrollados por OpenAI como ChatGPT, GPT-4 y Dall-E 3 los más populares y generalmente aceptados, soportando una carga diaria de millones de usuarios. Esta afluencia de usuarios se debe a los sorprendentes resultados que pueden ofrecer estos modelos de lenguaje, que día tras día siguen asombrando a la humanidad.

Siendo esta la influencia que propagan estas nuevas tecnologías, es imposible no considerar el diseño de software como un campo sobre el que aplicar estos procedimientos. Por ello, la propuesta de este trabajo, en la cual se intentarán definir ciertos procesos para poder aplicar la Inteligencia Artificial Generativa al Diseño del Software.

Se escoge el diseño de software como disciplina para este proceso por su complejidad. Es un ámbito que está lleno de matices y de pequeños detalles que pueden hacer que una solución que parece correcta, en realidad, no lo sea. El objetivo es llevar a los modelos de lenguaje al límite de lo que son capaces de realizar, abarcando desde la refactorización de código a la implementación de patrones de diseño.

1.2 Objetivos

El objetivo principal del trabajo de fin de grado es conseguir integrar y aplicar la Inteligencia Artificial Generativa al Diseño de Software. Los objetivos restantes que se quieren explorar durante este trabajo son los siguientes:

- Elaborar una estrategia para la creación y construcción de prompts en el ámbito del diseño de software.
- Definir un procedimiento para abordar la refactorización de código mediante modelos de lenguaje natural.
- Explorar las capacidades y limitaciones de los modelos de lenguaje al resolver casos de uso concretos mediante el uso de patrones de diseño.
- Mejorar los resultados obtenidos mediante la construcción de varios modelos GPT especializados en la selección e implementación de patrones de diseño.
- Reforzar la figura del ingeniero de software tras los últimos avances de la Inteligencia Artificial.

1.3 Estructura

Este trabajo de fin de grado, comenzará con una revisión exhaustiva de los conceptos más importantes referentes tanto al diseño de software como a la Inteligencia Artificial y los LLMs. Con el objetivo de garantizar su accesibilidad, se ilustrarán los fundamentos necesarios para comprender totalmente las situaciones planteadas durante el trabajo, detallando además la estructura y arquitectura de estos modelos de lenguaje natural, para así comprender también tanto sus limitaciones como sus capacidades.

Tras haber interiorizado los fundamentos teóricos, se planteará una serie de procedimientos a seguir para conseguir resultados significativos a la hora de utilizar los LLMs. Estas buenas prácticas nos permitirán integrar de una forma estructurada y precisa los distintos ámbitos dentro del diseño de software con los modelos de lenguaje y sus particularidades, lo que resultará en una mejora notable en la calidad de los resultados obtenidos.

Durante esta fase de integración y experimentación se explorarán, en primer caso, procedimientos de refactorización de código mediante LLMs, que servirán para introducir tanto estrategias de prompting como del uso de estos modelos. Este proceso inicial tiene la intención de plantar las bases del trabajo analizando problemas menos complejos para más tarde profundizar y ampliar todos estos aspectos al utilizar estas estrategias en el momento de diseñar, implementar y resolver casos de uso por medio de patrones de diseño.

Tomando como referencia las pautas anteriores, se construirán dos modelos GPT (Generative Pre-training Transformer) especializados, capaces, cada uno, de diseñar e implementar problemas no triviales inherentes al ámbito del diseño de software. Gracias a estos nuevos modelos, se intentará mejorar la calidad de los resultados que se hubiesen obtenido durante la fase anterior.

Finalmente, y como conclusión, se realizará un análisis humano de los resultados que se hayan obtenido a lo largo del trabajo y lo que estos implican. Además de comentar algunas consideraciones personales que conciernan a este trabajo.

CAPÍTULO 2
Fundamentos

Durante este capítulo se exploran las bases del diseño de aoftware y de los modelos de lenguaje natural.

En primer lugar, se abordan los principios del diseño de software. Centrando el análisis en el paradigma de la programación orientada a objetos y profundizando en conceptos como los Patrones de Diseño, Principios SOLID o la filosofía del Clean Code, herramientas indispensables para la creación de software robusto, mantenible y escalable. Estos fundamentos proporcionarán el conocimiento necesario para evaluar la relevancia de los resultados finales del estudio.

Después de haber examinado los principios de la programación orientada a objetos, se explorarán conceptos relacionados con la Inteligencia Artificial Generativa y los Modelos de Lenguaje Natural. Este segmento del estudio no solo abarcará las tecnologías más avanzadas y utilizadas en la actualidad, sino que también ofrecerá una visión detallada sobre su arquitectura y los principios que las sustentan.

Para culminar, se investigarán otros conceptos clave que sean esenciales para la comprensión holística del trabajo, asegurando así una base sólida y completa. Este capítulo, por tanto, no solo servirá como una introducción a importantes conceptos del diseño y la programación, sino que también establecerá un puente hacia la comprensión más profunda de cómo la inteligencia artificial está remodelando el panorama tecnológico actual.

2.1 Diseño de Software

Al abordar el tema del Diseño de Software, es común omitir un aspecto fundamental: el concepto abarca una diversidad de paradigmas dentro del ámbito del desarrollo de software. En este contexto, cuando nos referimos al diseño de software, estamos específicamente hablando del enfoque de programación orientada a objetos.

La programación orientada a objetos es un paradigma que se basa en el concepto de *objetos*. Estos objetos comprenden tanto datos, representados por sus *atributos*, como funcionalidades, expresadas a través de *métodos* o *funciones*. La esencia de este enfoque radica en la capacidad de encapsular datos y comportamientos en unidades cohesivas y reutilizables, lo que facilita la modularidad y organización del código.[1]

2.1.1. Principios de la Programación Orientada a Objetos

La programación orientada a objetos abarca varios principios que son fundamentales para comprender conceptos más complejos que pueden surgir de ellos. Es esencial explicar estos principios antes de adentrarse en aspectos más avanzados del tema.

La ***Herencia*** tiene un papel fundamental en este paradigma, su funcionamiento se basa en la extensión del comportamiento de otra clase u objeto, es decir, una clase que herede de otra mantendrá el comportamiento de la original, pudiendo ser utilizada como esta, añadiendo además funcionalidad extra según sea necesario. Por ejemplo, en un sistema bancario, un cliente puede necesitar diferentes tipos de cuentas, como una cuenta de ahorro y una cuenta de inversión. Estas cuentas comparten un conjunto de operaciones comunes que funcionan de la misma manera. Por lo tanto, a nivel de diseño, este sistema podría representarse con una clase *Cuenta* que contenga el comportamiento común a ambas, y dos clases adicionales, *CuentaAhorro* y *CuentaInversion*, que hereden de ella extendiendo la funcionalidad específica de cada tipo de cuenta. Esta estructura jerárquica no solo promueve la reutilización del código y la coherencia en el diseño, sino que también facilita la mantenibilidad del sistema a medida que evoluciona con nuevos requisitos y funcionalidades.[2]

El ***Encapsulamiento*** es un procedimiento mediante el cual se restringe el acceso tanto a los datos como a los componentes de los objetos y cuyo objetivo es conservar la integridad de los datos. Se lleva a cabo limitando el acceso a los componentes de una clase. Utilizando el encapsulamiento, podríamos proporcionar métodos para permitir que otros componentes del sistema accedan a esta información de manera controlada, sin necesidad de conocer los detalles de cómo se almacena o se procesa internamente. De esta manera, mantenemos la integridad de los datos y evitamos que se modifiquen de manera inapropiada desde el exterior del objeto.[2]

La ***Abstracción*** es el proceso mediante el cual se extraen los aspectos esenciales y significativos de una entidad, ignorando los detalles menos relevantes y concretos. Este concepto facilita así la comprensión y el manejo de sistemas complejos. En esencia, la abstracción permite centrarse en los aspectos más importantes de un objeto o sistema, lo que promueve una visión más clara y general de su funcionamiento. Como caso concreto, para representar un objeto de una clase *Libro*, podríamos definir una interfaz que representase todas las operaciones que deben de implementarse a la hora de utilizar un objeto de esta clase. De esta manera, se estarían extrayendo los elementos más importantes de esta entidad, abstrayendo su comportamiento.

El ***Polimorfismo*** es un concepto fundamental en la programación orientada a objetos que permite que objetos de diferentes clases relacionadas puedan ser tratados de manera uniforme. Esto significa que un objeto de una clase derivada puede ser manipulado utilizando una referencia de la clase base, lo que brinda flexibilidad, reutilización y mantenibilidad al código. A modo de ejemplo, dada una clase abstracta *Empleado* que contase con un método que calculase el salario de este empleado, otras clases que implementasen esta abstracción podrían ser utilizadas como la clase base, de esta forma podríamos calcular el salario medio de todos los empleados de una empresa, sin tener que centrarnos en qué tipo de empleado es y, en consecuencia, en qué salario tiene.[3]

2.1.2. Principios SOLID

En el ámbito del desarrollo de software, ***SOLID*** representa cinco principios fundamentales de diseño que orientan la elaboración de sistemas informáticos eficientes, reutilizables y robustos. Estos principios, concebidos como pilares dentro del diseño de software, se centran en establecer bases que garanticen la robustez, la facilidad de mantenimiento y la escalabilidad en el transcurso del tiempo.[4]

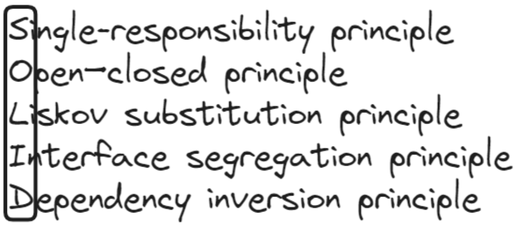


Figura 2.1: Los cinco principios SOLID de la programación orientada a objetos

El objetivo primordial de ***SOLID*** radica en fomentar el desarrollo de código limpio y modular, lo cual facilita su comprensión, extensión y adaptación a medida que los requisitos del proyecto evolucionan. Al adherirse a estos principios, los desarrolladores pueden mitigar la complejidad inherente al desarrollo de software, promoviendo la creación de sistemas más robustos y menos propensos a errores, lo que a su vez contribuye a una mayor eficiencia en el ciclo de vida del software y una reducción de costes a largo plazo.[4]

Los principios ***SOLID*** son los siguientes:

- El ***Principio de responsabilidad única*** establece que una clase debe tener un solo motivo para cambiar, lo que implica que cada clase debe tener una responsabilidad claramente definida y acotada dentro del sistema. Esta regla fomenta la cohesión y el mantenimiento del código al reducir la complejidad y el riesgo de efectos secundarios no deseados. Al adherirse a este principio, se promueve un diseño modular y flexible, donde los cambios en los requisitos del sistema se reflejan en cambios localizados y predecibles en el código. Esto facilita la comprensión, la depuración y la evolución del software a lo largo del tiempo, mejorando su calidad, escalabilidad y mantenibilidad.[5]
- El ***Principio de abierto/cerrado*** postula que una clase debe ser abierta para su extensión pero cerrada para su modificación. Esto significa que el comportamiento de una clase debería poder ampliarse mediante la adición de nuevas funcionalidades, sin necesidad de alterar su código fuente original.

Al adherirse a este principio, se fomenta un diseño que promueve la reutilización del código existente y, además, reduce el riesgo de introducir errores al modificar clases ya probadas y estables. La aplicación de este principio se traduce en un software más flexible, modular y fácilmente mantenible, ya que permite adaptarse a nuevos requisitos o escenarios sin afectar a la integridad de las clases existentes.[5]

- El ***Principio de sustitución de Liskov*** establece que las subclases deben ser sustituibles por su clase base sin afectar al correcto funcionamiento del programa. Esto significa que cualquier instancia de la clase padre debería poder ser reemplazada por una instancia de cualquiera de sus clases descendientes sin alterar el comportamiento esperado del programa. Este principio promueve la coherencia y consistencia en el diseño de clases y jerarquías de herencia, lo que facilita la comprensión y el mantenimiento del código. Al adherirse a este principio, se evita la introducción de errores sutiles y se garantiza la interoperabilidad entre las diferentes partes del sistema. En resumen, la aplicación adecuada del principio de sustitución de Liskov conduce a un diseño de software más robusto, flexible y fácilmente extensible.[5]

- El ***Principio de segregación de la interfaz*** aboga por crear interfaces específicas y dedicadas a cada cliente, en lugar de tener interfaces monolíticas que contengan métodos irrelevantes para las clases que las implementen. Esto implica dividir las interfaces en conjuntos cohesivos de métodos que satisfagan las necesidades particulares de cada cliente, evitando así que las clases dependan de funcionalidades que no necesitan. Al seguir este principio, se promueve un diseño más limpio y coherente, facilitando la implementación de las interfaces y reduciendo el acoplamiento entre componentes del sistema. Además, al proporcionar interfaces más específicas, se mejora la modularidad y la capacidad de evolución del software. Resumiendo, el principio de segregación de la interfaz fomenta una arquitectura más flexible, adaptable y centrada en las necesidades de los clientes.[5]

- El ***Principio de inversión de la dependencia*** establece que las clases de alto nivel no deben depender de clases de bajo nivel, sino de abstracciones. Esto significa que las dependencias entre módulos o componentes de un sistema deben ser definidas mediante interfaces o clases abstractas en lugar de depender directamente de implementaciones concretas. Al adherirse a este principio, se promueve una arquitectura más flexible y fácilmente adaptable, ya que las clases de alto nivel pueden interactuar con múltiples implementaciones a través de una interfaz común, lo que facilita la sustitución y la extensión de componentes sin modificar el código de las clases de alto nivel. Además, este enfoque facilita la realización de pruebas unitarias y la reutilización del código, ya que las dependencias pueden ser fácilmente simuladas o substituidas por objetos *mock* durante las pruebas. En resumen, el principio de inversión de la dependencia ayuda a reducir el acoplamiento entre componentes, fomentando una arquitectura más modular, escalable y mantenible.[5]

2.1.3. Patrones de Diseño

Los ***Patrones de Diseño*** en el contexto del desarrollo de software son soluciones comunes para problemas recurrentes y desafíos inherentes al proceso de diseño y construcción de sistemas informáticos. Estos patrones abarcan las mejores prácticas y el conocimiento acumulado por la comunidad de desarrollo de software a lo largo del tiempo, ofreciendo un enfoque sistemático y efectivo para abordar problemas específicos en el diseño de software.[6]

El concepto de patrones de diseño se originó en el campo de la arquitectura de software con la publicación seminal de *Design Patterns: Elements of Reusable Object-Oriented Software* por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides en 1994. Desde entonces, los patrones de diseño han evolucionado para abarcar una amplia gama de paradigmas de programación y enfoques arquitectónicos, trascendiendo los límites de lenguajes de programación específicos o tecnologías particulares.

La relevancia y la importancia de los patrones de diseño en el diseño de software proviene de varios factores cruciales. Los patrones de diseño representan un compendio de estándares y mejores prácticas en el diseño de software, pulidas a lo largo de décadas de experiencia en la industria. Al adoptar y aplicar estos patrones, los desarrolladores pueden beneficiarse del conocimiento colectivo de la comunidad de desarrollo, evitando errores comunes y adoptando soluciones probadas.

Además, los patrones proporcionan una terminología común para discutir y comunicar fácilmente soluciones de diseño con profesionales del ámbito de la ingeniería del software. Esto facilita la colaboración y el intercambio de ideas, promoviendo una comprensión compartida del diseño del sistema.

Representan un pilar fundamental en la disciplina del diseño de software, ofreciendo un enfoque estructurado para abordar desafíos de diseño complejos. Al adoptar y aplicar estos patrones, los desarrolladores pueden mejorar la calidad, la eficiencia y la mantenibilidad de los sistemas de software, contribuyendo así al éxito y a la sostenibilidad de los proyectos de desarrollo de software en la actualidad.

2.1.4. Clean Code

El término ***Clean Code*** representa un conjunto de prácticas y herramientas destinadas a crear código fuente que no solo sea fácil de leer y comprender, sino que también pueda mantenerse eficazmente a lo largo del tiempo. Esto es especialmente relevante en el contexto de sistemas software, cuya evolución y adaptabilidad frente a cambios en los requisitos, la aparición de nuevas tecnologías y los desafíos relacionados con el rendimiento son aspectos críticos para su sostenibilidad a largo plazo.[7]

La implementación de prácticas asociadas al ***Código Limpio*** exige una dedicación meticulosa desde las fases iniciales de cualquier proyecto software. Este compromiso abarca desde la adhesión a guías de estilo específicas hasta la aplicación de análisis estático de código, pasando por el desarrollo dirigido por pruebas o *Test-Driven Development - TDD* y la integración y despliegue continuos (*CI/CD*, por sus siglas en inglés). La adopción de estas metodologías no solo mejora la legibilidad y mantenibilidad del código sino que también promueve una cultura de colaboración, revisión continua y aprendizaje mutuo dentro de los equipos de desarrollo.[8]

Incorporar el ***Código Limpio*** en el ciclo de vida del desarrollo de software tiene como resultado inmediato una base de código más clara y segura. Facilita las revisiones de código, tanto manuales como automáticas, permitiendo la identificación y corrección temprana de posibles defectos. Esta práctica es esencial no solo para mantener altos estándares de calidad sino también para facilitar la contribución y participación activa de la comunidad en proyectos de código abierto, aumentando así su potencial de adopción y mejora continua.[8]

Desde una perspectiva científica, el enfoque subraya la importancia de la calidad del código en el diseño de software, no solo como un factor técnico sino como un elemento cultural dentro de los equipos de desarrollo. Promueve un entorno donde el aprendizaje continuo, la adaptabilidad y la colaboración son fundamentales para enfrentar los retos inherentes al desarrollo de sistemas software complejos y de larga duración.[8]

La aplicación del ***Clean Code*** contribuye significativamente al éxito del desarrollo de software sostenible, asegurando que los sistemas sean escalables, flexibles y fácilmente actualizables. Esto reduce considerablemente el coste y el esfuerzo requeridos para su mantenimiento y evolución a largo plazo. En el ámbito académico y científico, este enfoque no solo enriquece la formación de futuros ingenieros de software, sino que también establece un marco de referencia para la investigación y el desarrollo de mejores prácticas en la ingeniería de software, promoviendo así la excelencia en el diseño y mantenimiento de sistemas de software robustos y de alta calidad.[8]

2.2 Modelos de Lenguaje Natural

En el campo de la ***Inteligencia Artificial***, los ***Modelos de Lenguaje*** son un conjunto de operaciones probabilísticas del lenguaje natural, diseñados para procesar, comprender y generar lenguaje humano. Estos modelos han experimentado una evolución notable, pasando de sistemas orientados a tareas específicas como la corrección ortográfica o la traducción automática, a soluciones altamente complejas y versátiles. Esta transformación ha sido catalizada, en parte, por el desarrollo y la implementación de los ***Large Language Models***, que escalan en magnitud y capacidad respecto a sus predecesores.[9]

$$P(word(n) \mid word(1), word(2), \ldots, word(n-1))$$

Figura 2.2: Representación de la probabilidad condicional en Modelos de Lenguaje: estimación de la siguiente palabra basada en el contexto previo

Los LLMs se distinguen por su habilidad para procesar y aprender de grandes conjuntos de datos textuales, lo que les permite desarrollar una comprensión más profunda y definida del lenguaje humano. Esta capacidad no solo mejora la precisión en tareas de *Procesamiento de Lenguaje Natural (NLP)*, sino que también amplía las aplicaciones potenciales de estos modelos en campos como el diseño de software, donde pueden contribuir significativamente a la automatización de la generación de código, la depuración y otras tareas relevantes.[9]

2.2.1. Large Language Models

Los ***Large Language Models*** son sistemas de inteligencia artificial diseñados para comprender, generar y traducir textos en lenguaje natural de manera similar a como lo haría un humano. Se caracterizan por su tamaño masivo, medido en número de parámetros, que puede ir desde cientos de millones hasta varios miles de millones. Este tamaño les confiere una capacidad sin precedentes para aprender patrones complejos del lenguaje, gramática, semántica y hasta ciertos aspectos culturales implícitos en los textos.[10]

Los LLMs han revolucionado el campo del *Procesamiento del Lenguaje Natural*, ofreciendo una mejora significativa en tareas como la traducción automática y la generación de textos, entre otros. Su desarrollo y perfeccionamiento han sido posibles gracias al aumento de la capacidad computacional y la disponibilidad de grandes cantidades de datos de texto para el entrenamiento.[10]

Los LLMs se fundamentan en diversas arquitecturas de redes neuronales, siendo las más destacadas:

- ***Redes Neuronales Recurrentes (RNN)***: Especializadas en procesar secuencias de datos, como lo son los textos, al mantener un estado o memoria sobre lo que ya han procesado. Esto les permite manejar información contextual en secuencias de texto. Sin embargo, tienen limitaciones en cuanto a la longitud de las dependencias que pueden manejar efectivamente.[11]
- ***Redes Neuronales Convolucionales (CNN) para NLP***: Aunque originalmente fueron diseñadas para el procesamiento de imágenes, las CNN también se han aplicado en el NLP. Son eficaces para captar patrones locales en los datos, como las relaciones entre palabras adyacentes, pero no son ideales para manejar dependencias a largo plazo en textos.[12]

Ambas arquitecturas han sido fundamentales en el desarrollo inicial de los LLMs, pero tienen sus limitaciones en cuanto a eficiencia y capacidad para manejar secuencias largas de texto.

La ***Arquitectura de Transformadores o Transformers*** representa un avance significativo en este campo, introduciendo un novedoso enfoque para manejar secuencias de datos. Este modelo se basa en *mecanismos de atención* que permiten ponderar la importancia relativa de diferentes partes del texto sin importar su distancia, superando así las limitaciones de las RNN y CNN en el NLP.[13]

El entrenamiento de los LLMs es un proceso intensivo que involucra varias etapas clave:

1. Preprocesamiento de Datos: Antes del entrenamiento, los datos de texto deben ser homogeneizados y estructurados adecuadamente. Esto puede in-

cluir la corrección de errores ortográficos, la tokenización (división del texto en unidades menores como palabras o caracteres), y la normalización.[13]

2. Entrenamiento Supervisado y No Supervisado: Los LLMs pueden entrenarse de forma supervisada y no supervisada. El entrenamiento supervisado utiliza pares de entradas y salidas (como textos y sus respectivas traducciones) para instruir al modelo. En contraste, el entrenamiento no supervisado solo utiliza datos de entrada, y el modelo aprende a generar salidas basándose en patrones internos del lenguaje. Muchos LLMs modernos utilizan un enfoque híbrido para aprovechar las ventajas de ambos métodos.[14]

3. Fine-tuning: Después del entrenamiento inicial, los LLMs suelen ser pulidos en tareas específicas utilizando un conjunto más pequeño de datos que es más relevante. Este proceso ajusta los parámetros del modelo para optimizar su rendimiento en tareas concretas, como la comprensión lectora o la generación de texto en dominios específicos.[15]

El entrenamiento de un LLM es un proceso que consume grandes recursos computacionales y energéticos, requiriendo acceso a robustas infraestructuras de hardware. A pesar de estos desafíos, los avances en la eficiencia de los modelos y las técnicas de entrenamiento continúan expandiendo las capacidades y aplicaciones de los LLMs en diversos campos.[10]

2.2.2. Transformadores

Los ***Transformadores*** han marcado un hito en el campo de la Inteligencia Artificial, transformando fundamentalmente el enfoque hacia el procesamiento del lenguaje natural. Introducida en el paper titulado *Attention is All You Need* por Ashish Vaswani et al. en 2017, esta arquitectura ha permitido que las máquinas procesen y comprendan secuencias de texto con una claridad y profundidad nunca antes vista.[16]

La arquitectura *GPT (Generative Pre-trained Transformer)*, desarrollada por OpenAI, destaca como un ejemplo representativo de esta tecnología. Su capacidad para crear contenido coherente y contextualmente adecuado, desde responder consultas hasta redactar composiciones, ha acercado las capacidades lingüísticas de la IA a las humanas.

La aparición de los transformadores ha impulsado la investigación y desarrollo en el campo de la Inteligencia Artificial, prometiendo sistemas aún más sofisticados y versátiles. Los transformadores no solo han mejorado la capacidad de las máquinas para entender el lenguaje sino que también han ampliado el horizonte de posibles innovaciones en el campo.[17]

La preparación de datos es el primer paso en el funcionamiento de los transformadores. Este proceso implica la conversión de texto en un formato que la red neuronal pueda entender, lo que se logra a través de la *tokenización* y los *embeddings* posicionales y de palabras.[16]

La *tokenización* divide el texto en unidades más pequeñas, conocidas como tokens. Los tokens pueden ser palabras, subpalabras o incluso caracteres, dependiendo de la tarea a realizar y el enfoque deseado. Este proceso es esencial para descomponer el texto en unidades operables para el análisis posterior.[16]

Una vez tokenizado el texto, cada token simboliza un vector en un espacio de múltiples dimensiones. Estos vectores, conocidos como *embeddings de palabras*, representan semánticamente los tokens de manera que palabras con significados similares estén más cercanas en el espacio vectorial. Esta transformación facilita que la red capture relaciones y patrones lingüísticos complejos.[16]

Dado que los transformadores no procesan los tokens de forma secuencial, utilizan *embeddings posicionales* para incorporar la información sobre el orden de los tokens en la secuencia. Estos vectores se suman a los *embeddings de palabras*, proporcionando a la red neuronal la posición relativa de cada token dentro de la secuencia.[16]

Tras comprender cómo la preparación de datos y la codificación de entradas establecen la base para que los transformadores interpreten y procesen el lenguaje, es fundamental adentrarnos en el corazón de lo que realmente potencia su capacidad para manejar complejidades lingüísticas a gran escala: el ***Mecanismo de Atención***. Este componente no solo profundiza en la interpretación contextual del texto, sino que redefine la manera en que la información es ponderada y utilizada a lo largo del proceso.

Mecanismos de Atención

Los ***Mecanismos de Atención*** permiten a los modelos de transformadores ponderar la importancia relativa de diferentes partes de la entrada cuando procesan o generan cada palabra. Es un proceso que, paso a paso, asigna un nivel de relevancia a cada elemento de la secuencia basándose en el contexto.[18]

Para cada token en la entrada, el modelo genera tres representaciones únicas: una *query*, una *key* y un *value*. Cada una de estas representaciones se obtiene multiplicando el vector del token por tres matrices de pesos que el modelo aprende durante el entrenamiento. Las *queries* y *keys* sirven para determinar la importancia relativa de los tokens, mientras que los *values* contienen la información que se utilizará en la salida.[18]

La relación entre cada *query* y todas las *keys* se mide mediante el producto escalar (una forma de calcular la similitud). Este cálculo resulta en un valor para cada par *query - key*, que indica cuánta atención debería prestar el modelo a cada parte de la entrada.[18]

Después de calcular los vectores de atención, estos se normalizan usando la función *softmax*. Esta función transforma los vectores de atención en un conjunto de valores que representan probabilidades. Al hacer esto, el modelo asigna un peso mayor a las partes más relevantes de la entrada, mientras que las menos relevantes reciben un peso menor. [18]

Este proceso de normalización es clave para adaptar el modelo a entradas de cualquier longitud, garantizando que la asignación de importancia a diferentes segmentos de la secuencia se mantenga uniforme y proporcional.

Una vez que se normalizan los valores de atención, estos se aplican a los vectores *values*. Se realiza multiplicando cada *value* por su correspondiente vector de atención normalizado. La suma de estos valores ponderados produce un resultado único para cada posición en la secuencia, lo que permite que cierta información se amplifique mientras que otra se atenúa.[18]

Los mecanismos de atención han sido fundamentales para el éxito de los transformadores en tareas de procesamiento del lenguaje natural. Al ajustar dinámicamente la importancia de diferentes segmentos, los transformadores pueden procesar el lenguaje de manera más natural y efectiva, capturando sutilezas y dependencias complejas.

Arquitectura

La arquitectura de los transformadores funciona mediante la integración de *encoders* y *decoders*. Estos dos componentes trabajan conjuntamente para procesar y generar texto, aprovechando la contextualización proporcionada por el *mecanismo de atención*. La arquitectura de los transformadores, por lo tanto, no solo captura la esencia lingüística en profundidad sino que también genera respuestas coherentes y contextualmente adaptadas.[18]

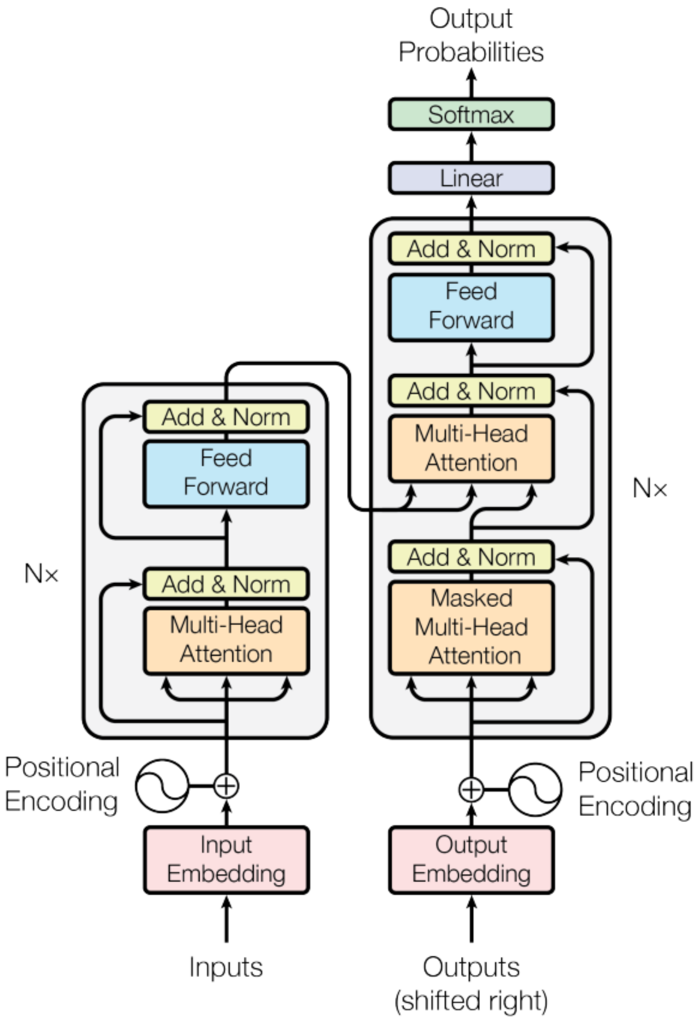


Figura 2.3: Transformador - Arquitectura del modelo

Los *encoders* son los bloques que procesan la entrada textual. Cada encoder en la secuencia realiza una serie de operaciones que incluyen mecanismos de atención y *embeddings*, convirtiendo el texto de entrada en un conjunto de representaciones vectoriales. Estas representaciones capturan no solo el significado de las palabras individuales sino también el contexto proporcionado por la secuencia completa. La habilidad para entender este contexto subyace en la eficacia de los *encoders*, permitiéndoles preparar una base sólida sobre la cual los *decoders* pueden generar la salida.[16]

Por otro lado, los *decoders* asumen la tarea de transformar las representaciones creadas por los *encoders* y generar la secuencia de salida, ya sea en forma de texto traducido, respuesta a una pregunta, o cualquier otro tipo de escrito. Al igual que los *encoders*, los *decoders* se apilan en múltiples capas que aplican mecanismos de atención y *embeddings*, fusionando así, la comprensión del contexto de entrada con el proceso de generación de texto de salida.[16]

La interacción entre *encoders* y *decoders* es lo que permite a los modelos de transformadores realizar tareas complejas de procesamiento del lenguaje. Mientras que los *encoders* encapsulan la comprensión del texto de entrada, los *decoders* utilizan esa comprensión, paso a paso, para construir la salida. Este proceso se lleva a cabo a través de un flujo cuidadosamente orquestado que asegura que cada palabra generada se base en una comprensión integral tanto de la entrada como de lo que ha sido generado hasta el momento.[16]

2.2.3. Limitaciones de los Large Language Models

Modelos como GPT-4 han demostrado capacidades impresionantes, desde escribir historias y ensayos, hasta generar código a partir de especificaciones en lenguaje natural. No obstante, a pesar de su capacidad y versatilidad, estos modelos no están exentos de limitaciones y riesgos que necesitan ser determinados antes de ser utilizados en escenarios del mundo real.

Los LLMs comprenden una serie de competencias extraordinarias, la capacidad que presentan para aprender patrones estadísticos de lenguaje les permite generar lenguaje natural ofreciendo soluciones generativas en distintos ámbitos. Estas características los hacen particularmente adecuados para circunstancias que conciernen la generación consistente y coherente de texto.

A pesar de sus capacidades, los LLMs a menudo erran en tareas que requieren un razonamiento lógico avanzado, un conocimiento documentado o experiencia específica de un dominio. La eficacia de estos modelos puede verse comprometida en situaciones que necesitan una comprensión profunda y contextualizada, siendo necesario en muchos casos complementar las entradas con información significativa para obtener respuestas adecuadas.[19]

Estos modelos también enfrentan obstáculos derivados de los datos con los que se ha entrenado. Problemas como respuestas incorrectas, sesgos o contenido ofensivo o dañino pueden surgir debido a la calidad de estos o incluso por las propias limitaciones del modelo, precisando de mecanismos de detección y corrección eficaces.[20]

Comparados con modelos de *Procesamiento de Lenguaje Natural* especializados, los LLMs pueden no ser la mejor opción en tareas que requieren conocimientos específicos o habilidades particulares. En tareas como análisis de sentimientos o detección de spam, los modelos especializados, entrenados y ajustados con datos relevantes, pueden superar en rendimiento a los LLMs, ofreciendo además mayor eficiencia y exactitud.[19]

Dado que los *Large Language Models* pueden generar texto plausible y convincente sin garantizar su exactitud o fiabilidad, se desaconseja su uso especialmente en contextos de alto riesgo o donde las consecuencias de la desinformación puedan ser significativas.[20]

Es fundamental diseñar aplicaciones de estos modelos que minimicen fallos y proporcionen mecanismos para su detección, corrección e informe. La supervisión humana juega un papel crucial, especialmente en áreas sensibles como pueden ser el asesoramiento médico, legal o financiero, donde se debe garantizar la revisión, edición o aprobación de los resultados por expertos.

Comprender y abordar las limitaciones y riesgos de los LLMs es esencial para tomar decisiones informadas y conscientes sobre sus posibles implementaciones y uso. A pesar de ser una tecnología poderosa y versátil, es vital proceder con cautela, equilibrando sus fortalezas con sus limitaciones para, así, maximizar su potencial y minimizar los posibles perjuicios.

2.2.4. Prompting

La eficacia con la que los *Modelos de Lenguaje* cumplen sus tareas depende en gran medida de la comunicación con estos. Aquí es donde entra en juego el concepto de ***Prompting***, una disciplina clave a la hora de maximizar el potencial de los LLMs. [

El ***Prompting*** se refiere al proceso de formulación y presentación de una pregunta o instrucción específica a un modelo de lenguaje, con el propósito de generar una respuesta o acción deseada. Esta técnica es una forma de comunicación que requiere claridad, precisión y, a menudo, creatividad. Al igual que en cualquier forma de interacción, el éxito del mensaje no solo depende de su contenido sino también de cómo este se presenta.[21]

El ***Prompting*** actúa como un medio a través del cual podemos comunicar nuestras intenciones a un modelo de lenguaje, traduciendo nuestras preguntas y tareas a un formato que el modelo pueda entender y sobre el cual pueda actuar. Por ejemplo, un prompt podría ser tan simple como una pregunta directa *¿Cuál es la capital de Francia?* o tan complejo como una solicitud para generar un artículo científico sobre un tema específico. La clave reside en cómo se formula este para maximizar la probabilidad de obtener una respuesta útil y relevante.[21]

Es esencial recordar que la base de una interacción adecuada radica en nuestra capacidad para comunicar nuestras necesidades de manera clara y concisa. Al dominar el arte del *Prompting*, no solo se mejorará la calidad de las respuestas que recibimos, sino que también se abrirán nuevas vías para la innovación en el uso de la inteligencia artificial.

Principios del Prompting

- ***Especificidad***: La claridad y la concreción son fundamentales. Al formular un *prompt*, se debe procurar eliminar cualquier ambigüedad, para así limitar el alcance de la respuesta esperada. Esto ayuda a que el modelo comprenda con exactitud la operación a realizar.[22]
- ***Descriptividad***: El uso de analogías y descripciones detalladas mejora la comprensión del modelo sobre la tarea a realizar, lo que permite generar respuestas más significativas.[22]
- ***Repetición***: En ocasiones, repetir las instrucciones tanto antes como después del contenido principal para reforzar el entendimiento del modelo puede ser beneficioso. El objetivo es guiar al modelo hacia la generación de la respuesta esperada, subrayando la importancia de las instrucciones al reiterarlas.[22]
- ***Ordenación***: El orden en el que se presenta la información al modelo puede influir significativamente en la respuesta generada. La colocación de las instrucciones, ya sea antes o después del contenido principal, así como el orden de los ejemplos proporcionados, pueden alterar los resultados.[22]
- ***Proporcionar una Alternativa***: Dar al modelo una opción alternativa si no puede completar la tarea asignada puede ser útil. Por ejemplo, indicar que responda con "no encontrado"si la respuesta no está presente en el texto, ayuda a prevenir respuestas falsas o generadas incorrectamente.[22]

El prompting nace como una herramienta indispensable en el diálogo con los *Modelos de Lenguaje*. Su correcta aplicación no solo mejora la calidad de las interacciones, sino que también optimiza los resultados obtenidos, haciendo que la comunicación con estos modelos sea efectiva y eficiente. A través de la comprensión y aplicación de los ***Principios del Prompting***, podemos maximizar el potencial de los LLMs.

CAPÍTULO 3

Diseño de Software con Large Language Models

Los Large Language Models como GPT o Gemini están transformando el campo del Diseño de Software, cambiando la forma en la que los ingenieros abordan sus tareas y remodelando significativamente el proceso de diseño.

Los LLMs ofrecen a los desarrolladores una nueva perspectiva, mediante la cual, pueden orientar la toma de decisiones durante las múltiples etapas del diseño, abarcando desde el diseño conceptual de un sistema hasta la refactorización y mantenimiento del mismo.

Estos modelos pueden incluso generar o refactorizar código, haciéndolo más legible y eficiente. Además, son capaces de identificar errores y sugerir soluciones, ayudando a los ingenieros a evitar problemas comunes y a garantizar la calidad del código desde el principio. Esta capacidad no solo ahorra tiempo, sino que también mejora la mantenibilidad a largo plazo en los proyectos de software.

Igualmente, los modelos de lenguaje pueden ayudar a implementar patrones de diseño, promoviendo la creación de estructuras organizadas y robustas que mejoren la extensibilidad y adaptabilidad del software.

Aunque los LLMs ofrecen numerosos beneficios, es importante reconocer sus limitaciones. La responsabilidad final de cualquier diseño recae en el ingeniero, y los LLMs deben considerarse como herramientas de apoyo en lugar de sustitutos. La interpretación y aplicación de sus sugerencias dependen del juicio crítico del diseñador.

La naturaleza generativa de los modelos de lenguaje implica que sus respuestas pueden carecer de contexto o precisión en algunos casos, por lo que es esencial evaluar cuidadosamente sus sugerencias antes de aplicarlas.

Este capítulo se estructurará en torno a cómo los LLMs pueden contribuir al proceso de diseño de software. Se comenzará explorando cómo pueden ayudar a refactorizar el código para hacerlo más limpio y legible, con ejemplos prácticos que ilustran este proceso.

La refactorización juega un papel muy importante a nivel de diseño, ya que a pesar de que se busca llegar a soluciones óptimas y bien estructuradas, esto no siempre se logra, por lo que contar con un posible proceso de refactorización que ayude al ingeniero a mejorar la estructura y legibilidad del software podría ser muy beneficioso.

Tras el análisis realizado previamente, se discutirá cómo los Large Language Models pueden contribuir en la elección de patrones de diseño, facilitando la creación de estructuras escalables y mantenibles. Los patrones de diseño son el núcleo de la disciplina del diseño de software, por lo que este examen será más profundo, explorando más casos de uso, y cómo los modelos de IA pueden ofrecer alternativas e intuiciones para este proceso.

Además de esto, también se explorará cómo se pueden integrar las soluciones que propongan los modelos de lenguaje en los sistemas software, mediante la implementación de los propios patrones de diseño.

Finalmente, se estudiarán las lecciones aprendidas durante esta sección y se plantarán las bases para la construcción de un modelo GPT especializado, el cual pueda mejorar los resultados obtenidos durante este análisis.

3.1 Refactorización con LLMs

Los LLMs son capaces de descomponer grandes cantidades de código y detectar áreas que requieran mejoras, reduciendo la necesidad de realizar análisis manuales. Estos modelos son capaces de identificar problemas como la duplicación de código, malas prácticas de programación o estructuras ineficientes. Este análisis automatizado permite a los desarrolladores centrarse en los problemas más relevantes, abordando rápidamente las pequeñas áreas de mejora que pueda presentar el código y reduciendo la deuda técnica.

Además, los LLMs pueden proporcionar sugerencias detalladas sobre cómo mejorar la estructura del código. Por ejemplo, pueden detectar patrones de diseño obsoletos o ineficientes, recomendando la adopción de métodos más modernos o efectivos. Este nivel de análisis no solo ahorra tiempo, sino que también eleva la calidad del software al impulsar el hecho de que se mantengan unos estándares de programación.

Un componente esencial, pero a menudo pasado por alto en el proceso de refactorización, es la documentación. Los LLMs pueden desempeñar un papel importante en la generación o mejora de la documentación del código. Después de analizar y refactorizar el código, estos modelos pueden proporcionar explicaciones claras y concisas sobre las funciones y estructuras del software. Esto incluye la generación de comentarios y la creación de documentación detallada, entre otros. Al proporcionar esta asistencia, los LLMs no solo agilizan el proceso de documentación, sino que también apoyan el hecho de que la información sobre el software sea accesible y esté disponible.

Para validar los resultados obtenidos durante este análisis, se seguirán ciertos pasos que garanticen una calidad mínima en las respuestas. Esto es necesario debido a que los modelos de lenguaje son probabilísticos, es decir, no siempre generan respuestas correctas ni de calidad uniforme. Para ello, las peticiones a los modelos se ejecutarán varias veces para asegurar que la variabilidad entre las respuestas no sea demasiado relevante.

Además, se analizará el código proporcionado utilizando la herramienta de software *SonarQube*, que es capaz de medir métricas como la duplicidad del código, la deuda técnica y otros aspectos relevantes. Estas métricas ayudarán a explicar por qué los resultados obtenidos son válidos y a asegurar la calidad de estos. También permitirán comparar distintas métricas antes y después de la refactorización realizada por cada modelo, proporcionando una base sólida para evaluar la efectividad de los LLMs en el proceso de la refactorización de software.[23]

3.1.1. Diseño del Prompt

Un aspecto crítico para obtener resultados precisos de un LLM es el diseño del prompt. Los LLMs responden directamente a la entrada que se les proporciona, por lo que un buen diseño de estos es crucial para asegurar que el modelo genere respuestas relevantes. Un prompt bien diseñado permite a los modelos identificar con precisión las áreas de análisis, sugerir mejoras pertinentes y producir documentación clara.

Además, ayuda a evitar interpretaciones ambiguas o confusas, minimizando el riesgo de respuestas inexactas. Al incluir detalles específicos sobre las necesidades de refactorización, como la simplificación de funciones complejas o la mejora de la estructura de código, los desarrolladores pueden aprovechar al máximo la potencia de los LLMs, obteniendo resultados óptimos en la refactorización y documentación del software.

Diseño Inicial

A la hora de diseñar el prompt, se empleará una estrategia iterativa y de refinamiento. Primero, se redactará un prompt siguiendo las estrategias y recomendaciones discutidas en la sección *2.2.4 Prompting*. Posteriormente, se buscarán mejoras para elevar la calidad de las respuestas obtenidas por los modelos. Para esto, se utilizarán los propios LLMs, que ya realizan este tipo de tareas en otras circunstancias.

Por ejemplo, el modelo DALL-E 3 de OpenAI reestructura, amplía y optimiza los prompts proporcionados por los usuarios para mejorar las respuestas. Asimismo, la herramienta GPT-Builder ayuda a los usuarios a construir modelos especializados, optimizando los prompts para obtener mejores resultados. Esta característica de GPT-Builder se abordará en la siguiente sección, donde se seguirán los pasos para construir un modelo GPT especializado, comprobando así si la calidad de las respuestas que se obtengan es de más calidad.

El prompt inicial que se ha diseñado es el siguiente:

"You are required to refactor the given code into clean and understandable code. The refactored code should adhere to Clean Code and SOLID principles.

Clean code principles:

- *Functions should be small and perform a single task.*
- *Avoid duplication of code.*

- *Use meaningful names for variables, functions, and classes.*
- *Add comments where necessary to explain the code's purpose and functionality.*

SOLID Principles:

- *Single Responsibility Principle (SRP): A class should have only one reason to change.*
- *Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification.*
- *Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types.*
- *Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use.*
- *Dependency Inversion Principle (DIP): Depend on abstractions, not concretions.*

Refactor the provided code following Clean Code and SOLID principles. Remember to make the code clear and understandable. Use meaningful names and ensure each class or function has a single responsibility.

If you cannot refactor the code as instructed, respond with 'refactor not possible'."

Mediante el uso de este prompt se quiere insistir en la necesidad de aplicar las técnicas referentes al Clean Code, y a los principios SOLID. Durante parte del prompt se reiteran estas instrucciones, para así enfatizar y reforzar el entendimiento que pueda tener el modelo sobre la petición de forma que la respuesta sea la esperada.

Refinamiento

Una vez creado un prompt en el cual se especifica la respuesta esperada, se procede a utilizar el modelo GPT-4 para su optimización. Los LLMs pueden identificar áreas de mejora para que el modelo entienda mejor las necesidades del usuario. Además, son capaces de reescribir y mejorar la coherencia y cohesión de la petición.

En este caso, al preguntar al modelo sobre las áreas de mejora en el prompt, esta fue la respuesta:

"The prompt you've written is quite comprehensive and clear in its instructions. However, there are a few areas where you could make improvements to enhance clarity and ensure the AI fully understands the task. Here are some suggestions:

1. *Structure and Clarity: Organize the principles into bullet points to improve readability.*
2. *Specificity: Provide more specific instructions on what kinds of changes are expected.*

3. *Context: If there's any context about the code (like what it does), provide it.*
4. *Example: Include an example of bad code and how it should look after refactoring.*
5. *Handling Complex Code: Give instructions on what to do if the code is too complex to refactor in one step.*
6. *Handling Errors: Suggest checking for and handling potential errors in the code.*

Here's the final version incorporating these additional points:

You are required to refactor the given Java code into clean and understandable code. The refactored code should adhere to Clean Code and SOLID principles. Please follow these guidelines:

Clean Code Principles:

- *Functions should be small and perform a single task.*
- *Avoid duplication of code.*
- *Use meaningful names for variables, functions, and classes.*
- *Add comments where necessary to explain the code's purpose and functionality.*

SOLID Principles:

- *Single Responsibility Principle (SRP): A class should have only one reason to change.*
- *Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification.*
- *Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types.*
- *Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use.*
- *Dependency Inversion Principle (DIP): Depend on abstractions, not concretions.*

Instructions:

1. *Refactor the provided code following the Clean Code and SOLID principles.*
2. *Ensure the refactored code is clear and understandable.*
3. *Use meaningful names for all variables, functions, and classes.*
4. *Ensure each class or function has a single responsibility.*
5. *Add comments where necessary to explain the code's purpose and functionality.*
6. *Check for and handle potential errors in the code.*

Example:

Before Refactoring:

```java
public class User {
    public void saveUserToDatabase() {
        // code to save user to database
    }

    public void generateUserReport() {
        // code to generate user report
    }
}
```

After Refactoring:

```java
public class User {
    // User properties and methods
}

public class UserRepository {
    public void save(User user) {
        // code to save user to database
    }
}

public class UserReportService {
    public void generateReport(User user) {
        // code to generate user report
    }
}
```

If you cannot refactor the code as instructed, respond with refactor not possible”. If the code is too complex to refactor in one step, provide a plan for incremental refactoring.

This revised prompt adds structure, context, and clarity, making it more likely that the AI will understand and execute your instructions effectively.“

Después de implementar las mejoras propuestas, se vuelve a consultar al modelo si hay alguna otra mejora que se pueda implementar en el prompt, sin embargo; la respuesta ofrecida en este caso es muy similar a la anterior. Tras esto, se deben identificar las áreas de interés a considerar para finalmente construir el prompt definitivo.

Se ha decidido mantener las sugerencias sobre los principios SOLID y Clean Code, y se han añadido las instrucciones que se consideran más relevantes para este caso de uso. Además, en las conversaciones con cada modelo, se ha incluido un breve contexto sobre la funcionalidad de cada método que se va a refactorizar. El ejemplo de código se ha eliminado, ya que, aunque puede ser útil en casos donde se necesite una refactorización muy específica, como al actualizar un framework, no es relevante para este caso de uso. Finalmente, el prompt definitivo ha quedado así:

"You are required to refactor the given Java code into clean and understandable code. The refactored code should adhere to Clean Code and SOLID principles. Please follow these guidelines:

***Clean Code Principles:** [...]*

***SOLID Principles:** [...]*

Instructions:

- *Refactor the provided code following the Clean Code and SOLID principles.*
- *Ensure the refactored code is clear and understandable.*
- *Use meaningful names for all variables, functions, and classes.*
- *Ensure each class or function has a single responsibility.*
- *Add comments where necessary to explain the code's purpose and functionality.*
- *Check for and handle potential errors in the code.*
- *Ensure that the refactored code passes all existing tests and improve test coverage if possible.*
- *Ensure that the refactoring does not negatively impact performance.*

If you cannot refactor the code as instructed, respond with 'refactor not possible'. If the code is too complex to refactor in one step, provide a plan for incremental refactoring."

3.1.2. Proceso de Refactorización

Tras haber diseñado el prompt definitivamente, es hora de comprobar la capacidad de refactorización presente en estos modelos de lenguaje. Para ello, se ha decidido evaluar el rendimiento de los dos modelos más potentes y conocidos en el mercado: GPT-4 y Gemini, desarrollados por OpenAI y Google respectivamente.

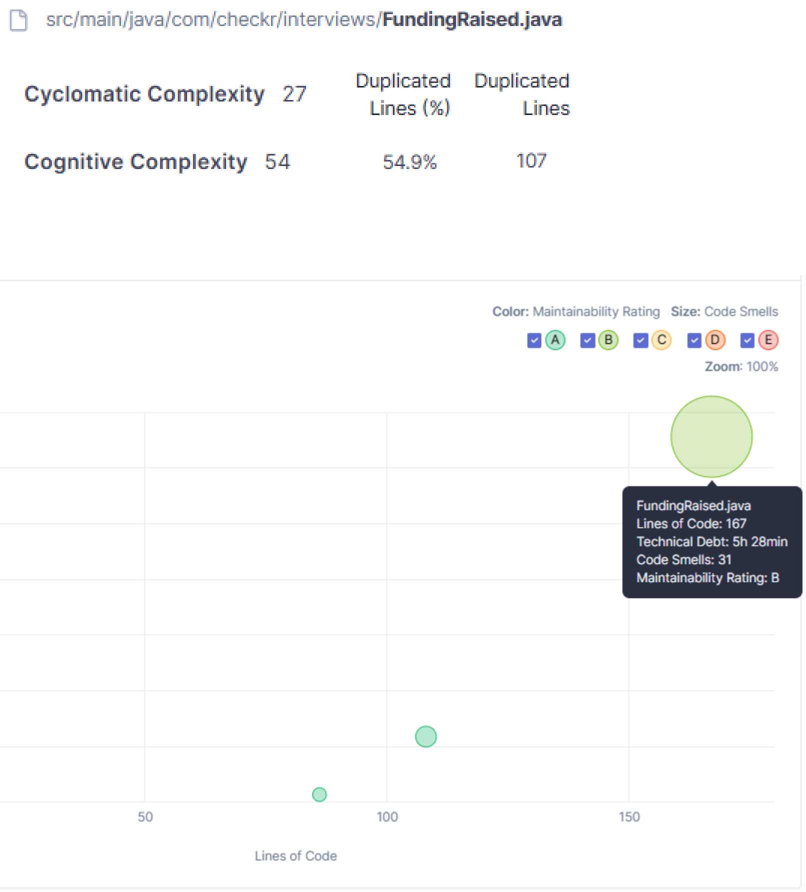
Utilizaremos ambos modelos debido a que, en aspectos más abstractos como el análisis de patrones de diseño, es preferible contar con la mayor asistencia e información posible. Dado que ambos modelos han sido entrenados con conjuntos de datos y objetivos diferentes, las respuestas que proporcionen ofrecerán perspectivas más amplias y significativas, mejorando la toma de decisiones.

Para el proceso de refactorización se utilizará el mismo problema a resolver, el cual, consiste en un repositorio de GitHub que contiene un proyecto en diferentes lenguajes de programación; en este caso, se seleccionará Java. El proyecto incluye una clase con varios métodos que fueron desarrollados utilizando malas prácticas de programación. Contiene mucho código duplicado, redundancias y es difícil de entender y mantener. El código se puede observar en el *Apéndice B.1*. Para verificar que la funcionalidad continúe siendo la misma después del proceso, se ejecutarán unas pruebas unitarias, comprobando así la correcta refactorización de los métodos.[24]

Se ha decidido mostrar el proceso de refactorización con respecto a este proyecto por ser el más representativo de todos con los que se ha experimentado. Además, los test unitarios, la simplicidad del código y la facilidad para analizarlo hacen que se pueda analizar de forma más adecuada, centrando la atención en el proceso y en las mejoras que se generan.

No obstante, este proceso se ha realizado en varios proyectos de código, refactorizando las clases que los formaban método a método, provocando resultados similares a los que se observan en esta demostración.

Utilizando la herramienta *SonarQube* en el proyecto original, se ha detectado un 54.9 % de código duplicado y una deuda técnica de 5 horas y 28 minutos. La deuda técnica es un concepto en la ingeniería de software que refleja el coste implícito de utilizar soluciones a corto plazo. Esto significa desarrollar código que puede funcionar bien a corto plazo, pero que, a largo plazo, dificultará la realización de cambios, llevando más tiempo que si se hubiera hecho un desarrollo adecuado desde el principio.



src/main/java/com/checkr/interviews/**FundingRaised.java**

		Duplicated Lines (%)	Duplicated Lines
Cyclomatic Complexity	27		
Cognitive Complexity	54	54.9%	107

FundingRaised.java
Lines of Code: 167
Technical Debt: 5h 28min
Code Smells: 31
Maintainability Rating: B

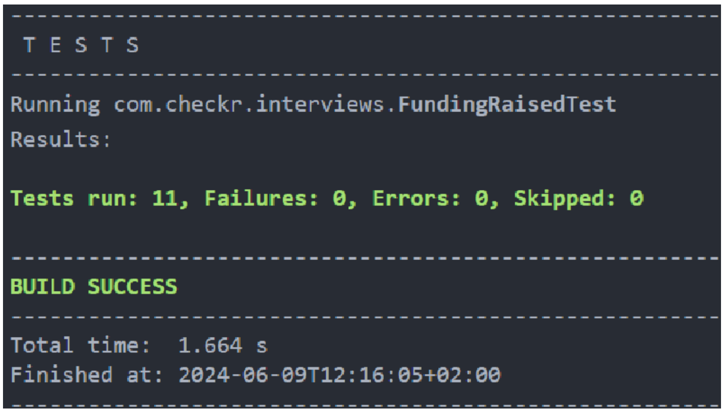
Figura 3.1: Métricas de SonarQube para el código original

Además, otras métricas relevantes que podemos detectar con *SonarQube* son la complejidad ciclomática y la complejidad cognitiva. La complejidad ciclomática es una métrica utilizada para medir la complejidad lógica del software. Este método es ampliamente aceptado, ya que puede aplicarse independientemente del lenguaje de programación. Por otra parte, la complejidad cognitiva mide la dificultad de entender intuitivamente un bloque de código.[25]

3.1.3. Refactorización con GPT-4

Se comenzará analizando el comportamiento de GPT-4, el modelo desarrollado por OpenAI. Usando el prompt diseñado previamente, se le proporcionará al modelo individualmente cada método que se vaya a refactorizar. Esto permite que el análisis realizado por el LLM sea más preciso, de forma que el modelo pueda refactorizar la clase al completo por pasos y de forma incremental. [26]

Para garantizar la consistencia de los resultados entre ejecuciones, se ha decidido refactorizar cada método al menos 5 veces en conversaciones diferentes, utilizando siempre los mismos prompts. En cada ejecución, los resultados han sido similares, con solo pequeños cambios en el código en cada conversación. Es importante destacar que en todas las respuestas, los tests unitarios se han ejecutado sin fallos, manteniendo siempre la funcionalidad intacta.



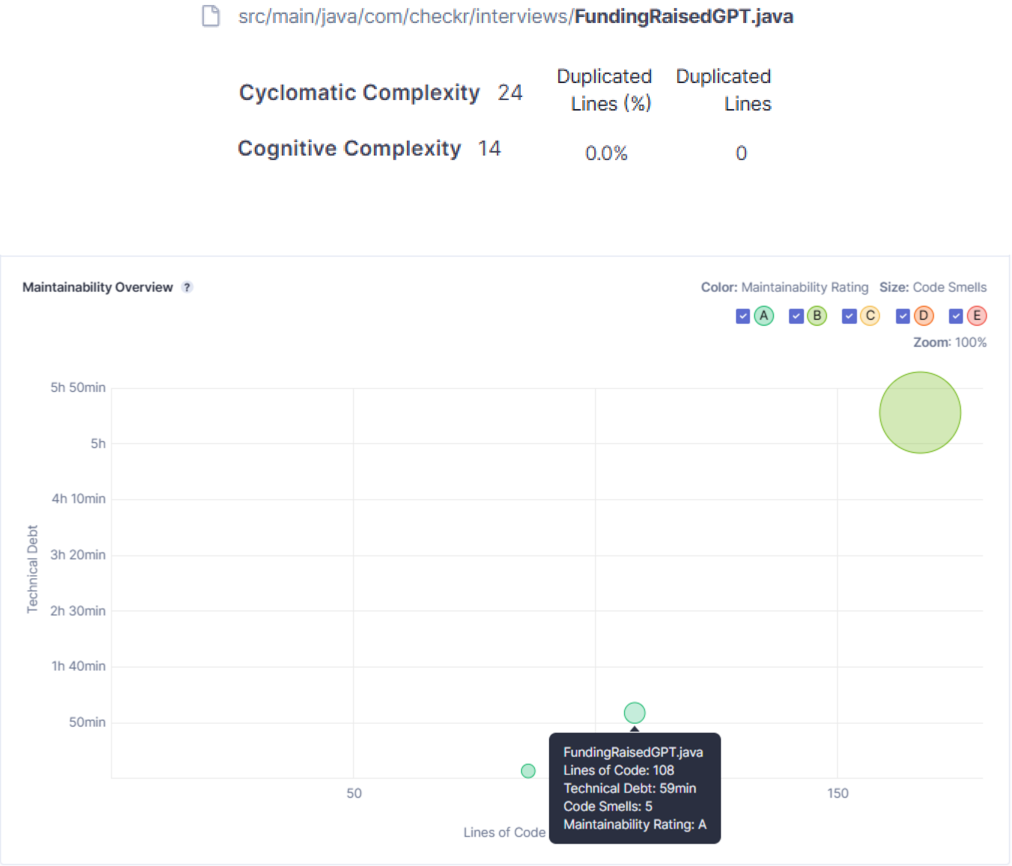
```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.checkr.interviews.FundingRaisedTest
Results:

Tests run: 11, Failures: 0, Errors: 0, Skipped: 0

-------------------------------------------------------
BUILD SUCCESS
-------------------------------------------------------
Total time:  1.664 s
Finished at: 2024-06-09T12:16:05+02:00
-------------------------------------------------------
```

Figura 3.2: Resultados de los tests post-refactorización mediante GPT-4

Esto es un gran primer paso, sin embargo, lo más importante es comprobar cuáles son los resultados de las mediciones realizadas por *SonarQube*, puesto que estas mediciones nos permitirán sacar mejores conclusiones con respecto a si la refactorización ha sido exitosa. Como ejemplo para el análisis, se ha decidido utilizar la respuesta que mejor representa el grueso de las conversaciones, la cual, se puede encontrar en el *Apéndice B.2*.

Tras haber realizado el análisis, se puede observar que la duplicidad del código se ha reducido al 0 %, eliminando totalmente de la clase el código duplicado. Además, la deuda técnica se ha visto reducida a tan solo 59 minutos. Estos resultados son realmente significativos y favorables. No es solo, que la funcionalidad se haya mantenido, sino que la refactorización se ha realizado con éxito.

La complejidad ciclomática también se ha visto reducida en tres unidades, y la complejidad cognitiva ha pasado de 54 a 14. Todas las métricas se han visto mejoradas considerablemente, haciendo que la calidad del código haya aumentado de forma inversamente proporcional.


src/main/java/com/checkr/interviews/**FundingRaisedGPT.java**

Cyclomatic Complexity	24	Duplicated Lines (%)	Duplicated Lines
Cognitive Complexity	14	0.0%	0

FundingRaisedGPT.java
Lines of Code: 108
Technical Debt: 59min
Code Smells: 5
Maintainability Rating: A

Figura 3.3: Métricas de SonarQube para el código refactorizado mediante GPT-4

Al observar el código refactorizado más profundamente, se pueden detectar ciertos fallos o detalles que *SonarQube* no es capaz de identificar. Existen partes del código que se podrían estructurar de forma diferente; en particular, hay dos métodos que realizan una funcionalidad muy similar y que se podrían unificar para reducir aún más la complejidad y duplicidad del código.

Además, a pesar de que en el prompt se especifica que deben incluirse comentarios explicativos sobre el propósito y la funcionalidad del código, solo se ha añadido un comentario en toda la refactorización.

Es cierto que el código generado por estos modelos debe revisarse y cotejarse, y probablemente requiera ajustes y correcciones adicionales para alcanzar una legibilidad y eficiencia óptimas. No obstante, el hecho de que el modelo GPT-4 haya logrado estos resultados es más que significativo.

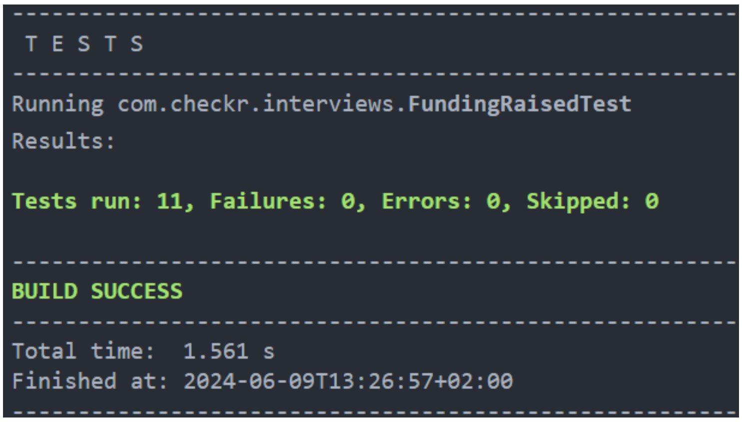
3.1.4. Refactorización con Gemini

En la segunda parte del análisis, utilizaremos el modelo Gemini desarrollado por Google, así como el enfoque incremental empleado anteriormente. Este método se adopta porque si el modelo recibe un conjunto de datos muy amplio, como una clase Java completa, tiende a comportarse erráticamente, no analizando bien el problema y generando respuestas inexactas e incompletas.[27]

Aunque es probable que las respuestas de los modelos sean similares, dado que ambos se enfrentan a ejercicios poco ambiguos, como lo es la refactorización, resulta interesante explorar las diferencias o mejoras que cada modelo puede presentar en sus resultados.

Para el análisis se ha llevado a cabo el mismo procedimiento que antes, escogiendo el código presente en el *Apéndice B.3*. Se ha considerado esta solución por las mismas razones que se han comentado en el análisis anterior, es decir, esta conversación representa la media de todas las respuestas que se han ido obteniendo durante la experimentación.

Los resultados proporcionados por el modelo Gemini muestran que la calidad de la respuesta es superior a la ofrecida por GPT-4. La estructura general del código está mejor organizada, resultando en una mayor legibilidad y ausencia total de código duplicado. El modelo comprende completamente el contexto y la refactorización realizada es adecuada. Los tests también muestran resultados exitosos en cada conversación, a pesar de que, en alguna se ha tenido que realizar algún ajuste menor.



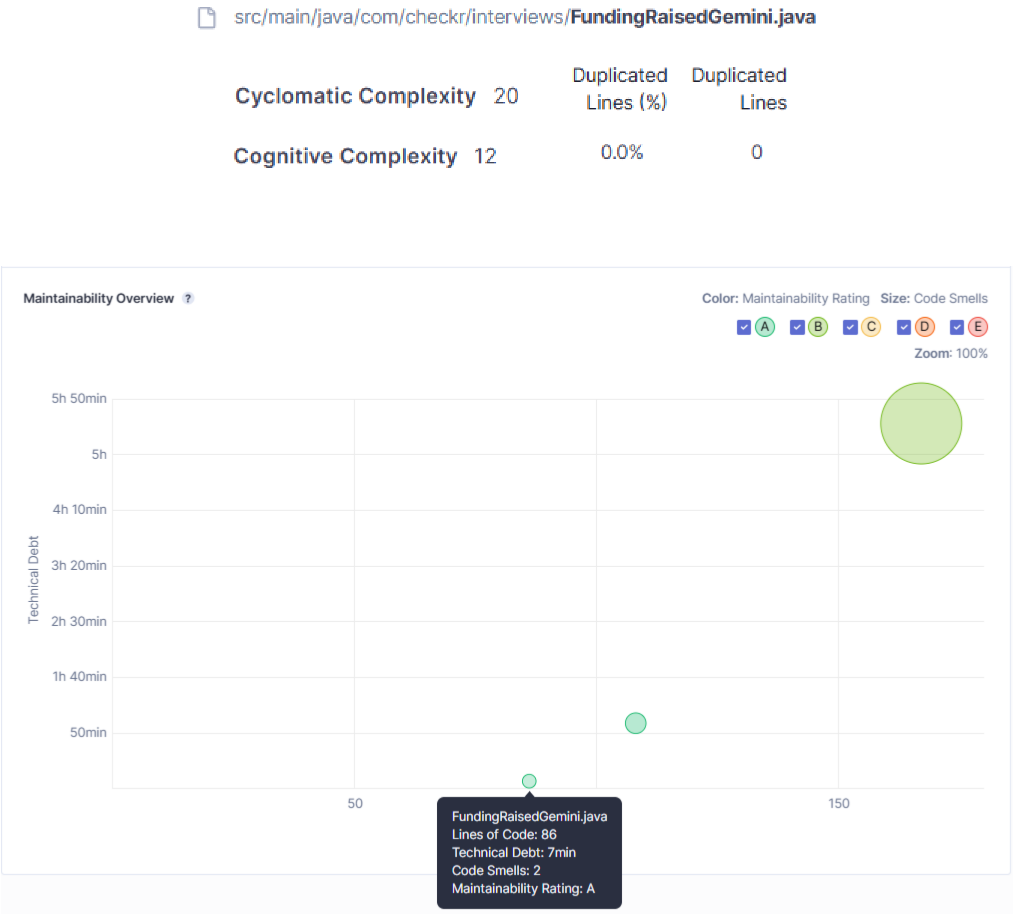
```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.checkr.interviews.FundingRaisedTest
Results:

Tests run: 11, Failures: 0, Errors: 0, Skipped: 0

-------------------------------------------------------
BUILD SUCCESS
-------------------------------------------------------
Total time:  1.561 s
Finished at: 2024-06-09T13:26:57+02:00
-------------------------------------------------------
```

Figura 3.4: Resultados de los tests post-refactorización mediante Gemini

Las cifras arrojadas por el análisis realizado en *SonarQube* muestran que la respuesta ofrecida por Gemini mejora considerablemente a la de GPT-4, como ya se había comentado. La duplicidad del código se mantiene en el 0 %, pero la deuda técnica se ha visto reducida de los 59 minutos de GPT-4 a tan solo 7 minutos, esto, ya de por sí refleja una mejora significativa.

Asimismo, la complejidad ciclomática ha pasado de las 24 unidades de GPT-4 a 20. La complejidad cognitiva también se ha visto reducida a 12 con respecto a las 14 unidades del modelo desarrollado por OpenAI.

Sin embargo, a pesar de que el resultado haya sido mejor que el proporcionado por GPT-4, se continúan encontrando algunos fallos. En este caso, el modelo no ha añadido ningún comentario a pesar de que en el prompt se indicaba explícitamente la necesidad de hacerlo. También se han encontrado otros errores, como un método que contiene un bloque *try-catch* que no maneja adecuadamente las excepciones que puedan ser lanzadas.



src/main/java/com/checkr/interviews/**FundingRaisedGemini.java**

		Duplicated Lines (%)	Duplicated Lines
Cyclomatic Complexity	20	0.0%	0
Cognitive Complexity	12		

FundingRaisedGemini.java
Lines of Code: 86
Technical Debt: 7min
Code Smells: 2
Maintainability Rating: A

Figura 3.5: Métricas de SonarQube para el código refactorizado mediante Gemini

3.2 Patrones de diseño con LLMs

Los Patrones de Diseño son una de las áreas más complejas y abstractas en el ámbito del Diseño de Software. Estos, requieren un amplio conocimiento y experiencia. Existe una gran variedad de patrones disponibles, cada uno diseñado para ofrecer soluciones a problemas no triviales. Esta diversidad implica que es esencial comprender profundamente cada patrón para determinar cuál es la solución más óptima, especialmente cuando varios patrones parecen adecuados para un mismo tipo de problema.[6]

La correcta aplicación de los patrones de diseño presenta varias ventajas significativas, incluyendo mejoras en el rendimiento, la mantenibilidad y la escalabilidad de los sistemas de software. Sin embargo, un uso inadecuado de estos puede tener el efecto contrario, generando problemas en estas mismas áreas. Por lo tanto, es crucial que estos patrones se apliquen con precisión y de manera adecuada.

Los Large Language Models pueden mejorar significativamente la toma de decisiones en el proceso de diseño de software. Aunque no siempre proporcionen la mejor solución para cada caso, ofrecen una variedad de posibles enfoques y perspectivas. Esto permite considerar diferentes patrones que tal vez no se habrían contemplado anteriormente, mejorando así el proceso de diseño. Dado que el diseño de software es una tarea inherentemente compleja, contar con herramientas como los LLMs es muy beneficioso. Actúan como copilotos en el proceso de la toma de decisiones, ofreciendo distintas visiones sobre un mismo problema.

La amplia gama de opciones que ofrecen estos modelos permite que los ingenieros y diseñadores de software tomen decisiones mejor informadas y con mayor precisión. Sin embargo, es crucial reconocer que, aunque los LLMs pueden asistir en la toma de decisiones, la responsabilidad final recae siempre en el ingeniero de software. Es importante subrayar que, aunque los LLMs a menudo proporcionan soluciones útiles, estas pueden ser incorrectas o imprecisas. Por lo tanto, los LLMs deben utilizarse como herramientas de asistencia, pero nunca se debe delegar completamente la responsabilidad en ellos.

Existen tres tipos de patrones de diseño: los patrones creacionales, que proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código; los patrones estructurales, que explican cómo ensamblar objetos y clases en estructuras más grandes, manteniendo la flexibilidad y eficiencia de la estructura; y los patrones de comportamiento, que se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.[28]

Se explorarán tres casos de uso correspondientes a estos tres tipos de patrones y se analizarán las respuestas proporcionadas tanto por GPT-4 como por Gemini. Este análisis no pretende comparar ambos modelos, sino examinar sus respuestas y cómo es posible beneficiarse de las intuiciones ofrecidas por estos modelos. Ambas respuestas pueden ofrecer distintas perspectivas sobre el mismo problema, lo que finalmente hará que el análisis y la propuesta de diseño para resolver el caso de uso sea más preciso y adecuado.

Para el análisis de los patrones de diseño, el proceso se dividirá en dos pasos. El primero, consistirá en obtener la información más relevante con respecto a las propuestas de patrones de diseño que presenten los modelos, para así poder evaluar si estas opciones son adecuadas.

Cada parte de este proceso se llevará a cabo con un prompt distinto, esto se debe a que en la fase inicial, el objetivo principal es poder obtener la máxima información posible para poder tomar una decisión sobre las propuestas de los modelos. Por ello, el prompt inicial estará enfocado en este aspecto y no en la obtención de ejemplos o estructuras de código que muestren cómo se podría implementar el patrón.

El segundo paso, si se considera que la respuesta del modelo es adecuada, se utilizará un segundo prompt. Mediante el cual, junto a la propuesta y el caso de uso correspondiente, se realizará una petición a los LLMs para conseguir una estructura de código que pueda ser útil durante la implementación.

3.2.1. Diseño del Prompt - Selección de Patrones

Diseño Inicial

Para diseñar el prompt, seguiremos una estrategia parecida a la que se había marcado en la refactorización del código. Comenzaremos diseñando un prompt en el cual incluyamos los aspectos críticos que queramos obtener en la respuesta.

En este caso, el objetivo es obtener una respuesta informativa, que sirva de apoyo a la hora de tomar decisiones con respecto al diseño que se vayan a implementar en un sistema. Debido a este propósito se ha diseñado el siguiente prompt inicial.

"You are a Design Pattern Expert. When given a use case, analyze the requirements and recommend suitable design patterns. Include the following in your response:

- *Recommended Pattern: Name and briefly describe the pattern that best fit the use case.*
- *Justification: Explain why this pattern is suitable, considering scalability, maintainability, and flexibility.*
- *Pros and Cons: Outline the advantages and potential drawbacks of the selected pattern.*
- *Alternatives: Mention any alternative patterns and compare them to the chosen pattern.*

Your response should provide a comprehensive solution that addresses the problem effectively while considering future scalability and maintainability. No implementation of the Pattern is needed."

Con este diseño queremos obtener una respuesta en la cual podamos obtener la mayor información posible sobre los aspectos que se han considerado más relevantes a la hora de escoger un patrón de diseño para solucionar un caso de uso.

El primero de estos es la justificación; cómo se está diseñando este prompt para tomar decisiones, es totalmente fundamental conocer por qué este patrón es el adecuado, y cuál es el razonamiento que ha hecho el modelo para llegar a esa conclusión.

Una vez se ha discernido cuál es la justificación de la elección que ha hecho el modelo, es relevante conocer cuáles son las ventajas y desventajas que puede aportar esta elección de diseño. Con esta información, se puede determinar si el patrón es adecuado para el sistema que se está diseñando o, en caso contrario, descartarlo.

Refinamiento

Finalmente, considerar algunas alternativas que quizás puedan encajar mejor en el sistema que se está desarrollando, ya sea porque el modelo no ha realizado la mejor elección, o porque las propiedades de los otros patrones encajan más con lo que se estaba buscando en un principio.

Tras haber seleccionado estas características, se decide refinar el prompt de nuevo usando GPT-4, obteniendo así las posibles áreas de mejora sobre nuestra petición de que encuentre el modelo. Esta es la respuesta que se ha obtenido.

"The prompt provided is well-structured and informative, clearly guiding the response needed from a design pattern expert. However, there are several areas where it could be improved to enhance clarity and comprehensiveness:

1. ***Specificity and Examples**: The prompt could benefit from asking for specific examples or case studies where the recommended patterns have been successfully implemented. This would provide concrete evidence of the patterns' effectiveness and practicality.*
2. ***Criteria for Evaluation**: While scalability, maintainability, and flexibility are mentioned as considerations, the prompt could further specify any additional criteria important to the use case, such as performance, security, or cost implications. This would help in evaluating the suitability of the design patterns more thoroughly.*
3. ***Context and Constraints**: Incorporating a section that asks the expert to consider any specific context or constraints (like technological, business, or regulatory constraints) relevant to the use case can lead to a more tailored and realistic pattern recommendation.*
4. ***Implementation Guidance**: Adding a request for brief guidance on how to implement the recommended patterns could be beneficial. This might include steps, tools, or programming paradigms, offering a more actionable solution to the audience.*
5. ***Impact on Other Architectural Aspects**: The prompt could encourage discussion on how the recommended patterns impact other architectural aspects or interact with other patterns not selected. This helps in understanding the holistic effect of the pattern choice on the system architecture.*
6. ***User-centric Considerations**: It might be beneficial to include a point about how the patterns affect user experience or service delivery. This can make the solution more aligned with user-centric design principles.*
7. ***Feedback Mechanisms**: The prompt could ask for suggestions on how to gather feedback and measure the effectiveness of the implemented design patterns. This would help in continuous improvement and validation of the design decisions made.*

By addressing these areas, the prompt would not only guide the expert to provide a more detailed and practical response but also ensure that the solution is robust, contextually appropriate, and aligned with broader business and technical objectives."

Tras identificar las áreas de mejora en la respuesta del modelo de OpenAI, se ha refinado aún más el prompt, enfocándose en los puntos *1. Specificity and Examples* y *5. Impact on Other Architectural Aspects*, que se ha considerado que son los más relevantes para esta experimentación.

Gracias al primer punto textit1. Specificity and Examples se podrían observar parecidos con otras funcionalidades que sean similares a las que se quieren implementar. Además, visualizar cómo los patrones pueden aplicarse en distintos ámbitos puede hacer que el ingeniero sea más consciente del posible encaje del patrón en el sistema.

Con respecto al segundo punto *5. Impact on Other Architectural Aspects* se podría explorar cómo el patrón puede afectar en otras áreas del sistema, pudiendo prevenir futuros conflictos que no se hayan podido tener en cuenta. También se podría considerar cómo podría combinarse con otros elementos consiguiendo así un diseño más óptimo.

El objetivo es utilizar los modelos de forma informativa ayudando así en la toma de decisiones respecto al diseño de software. Por esta razón, se ha considerado que áreas como la guía de implementación o las consideraciones del usuario no son tan necesarias para este contexto. Además, si la petición abarcara demasiados espectros, la respuesta no llegaría a porfundizar lo suficiente en cada uno de ellos.

Finalmente, tras aplicar los puntos ya mencionados con la ayuda de GPT-4 y refinar así, aún más el prompt, se ha llegado a la siguiente propuesta:

"As a Design Pattern Expert, your expertise is crucial in analyzing specific use cases and recommending the most suitable design patterns. Your response should meticulously address the following points to provide a well-rounded solution:

Recommended Patterns:

- *Identification and Description: Clearly identify and describe the design pattern that best aligns with the given use case. Detail the structure and fundamental principles of the pattern.*
- *Relevant Examples: Provide specific, real-world examples or case studies demonstrating successful implementation of these patterns. Highlight how these implementations addressed similar challenges.*

Justification:

- *Fit for Purpose: Elaborate on why the selected pattern is ideal for the use case, focusing on its scalability, maintainability, and flexibility. Discuss any specific features of the pattern that makes it particularly suitable.*
- *Strategic Impact: Explain how this pattern enhances the system's overall strategy and long-term goals.*

Pros and Cons:

- *Advantages: Outline the key benefits of using the selected design pattern, emphasizing how it contributes to the system's efficiency and robustness.*
- *Drawbacks: Discuss any potential limitations or challenges associated with this pattern, including scenarios where they might not be the best fit.*

Impact on Other Architectural Aspects:

- *System-wide Effects: Analyze how the recommended pattern will interact with other architectural elements of the system. Consider its impact on both the technical architecture and business processes.*

- *Potential Conflicts and Synergies: Identify any possible conflicts or synergies with existing architectural frameworks or patterns. Propose solutions or adjustments to mitigate conflicts and enhance synergies.*

Alternatives:

- *Comparison: Introduce alternative design patterns that could also be applicable. Provide a comparative analysis showing the differences in outcomes, complexity, and other relevant factors.*
- *Selection Rationale: Justify why the primary choice is preferable over these alternatives based on the specific requirements of the use case.*

Your response should synthesize these considerations into a comprehensive solution that not only addresses the immediate needs of the use case but also anticipates future requirements and challenges, thereby ensuring scalability and maintainability within the broader architectural context.“

Iterar

Mencionar que el proceso de validación de un prompt se debe hacer con respecto a los resultados que se obtengan. Es decir, para la creación de un prompt desde cero, se van a tener que pasar por varias propuestas, en las que poco a poco se vayan corrigiendo y refinando los detalles que hacen que las respuestas no sean las adecuadas.

Durante la construcción de este prompt, se ha pasado por varias versiones, comprobando cuáles eran las respuestas que se obtenían al proporcionar los distintos casos de uso. Gracias a las soluciones que han ido proporcionando los modelos, se ha llegado a la versión definitiva, que es la que se muestra en este análisis.

Este proceso es iterativo y mediante la experimentación y las pruebas, se pueden ir detectando cuáles son los detalles que hacen que un prompt sea válido, consiguiendo así refinar a su vez las respuestas que se obtienen de los LLMs.

3.2.2. Diseño del Prompt - Implementación de Patrones

Diseño Inicial

La implementación de patrones de diseño mediante LLMs requiere un prompt mucho más conciso y breve en comparación con otros casos. Esto se debe a que, en esta situación, se busca que el modelo construya un esqueleto de la estructura del patrón para luego completarla con detalles más específicos del sistema, por lo que hay menos instrucciones que se le deben dar al modelo.

En este caso, se busca diseñar un prompt que, dado un caso de uso y un patrón de diseño, sea capaz de implementar la estructura básica de ese patrón. Esto incluye la creación de interfaces, la lógica correspondiente al patrón, y comentarios explicativos que describan la funcionalidad de cada elemento, relacionándolos con las especificaciones del caso de uso.

La estrategia a seguir será la misma que en los casos anteriores, sin embargo, por lo que ya se ha mencionado, esta vez se partirá de un prompt mucho más simple:

"You are a software design pattern expert. Your task is to implement a given design pattern in Java based on a provided use case. Please include the Java code that demonstrates the core structure of the pattern."

Refinamiento

Tras el diseño de este prompt las áreas de mejora más interesantes que encuentra el modelo para refinar el prompt y, que además se deciden aplicar, son las siguientes:

"Your prompt has a clear goal, but it could benefit from some refinements to guide the AI more effectively and to set clearer expectations. Here are a few areas for improvement:

- *Code Comments: Request that the AI includes comments in the Java code. This will make the code more understandable and educational, especially for those who may not be familiar with the design pattern being implemented*
- *Error Handling: Mention if you want the code to include basic error handling to make the implementation more robust."*

Los comentarios son fundamentales en muchos aspectos de la programación, pero para este caso de uso lo son aún más. Contextualizar cada componente dentro de la estructura de código que se quiere obtener es muy importante para poder comprender el funcionamiento del patrón y dónde se deben implementar todos los detalles específicos del sistema.

Además, el manejo de errores en un sistema software es un elemento crucial para hacer el código más robusto, seguro y consistente. El modelo puede ser capaz de detectar dónde pueden haber vulnerabilidades o posibles fallos, por lo que contar con este manejo de errores es muy beneficioso.

Finalmente tras aplicar todos estos aspectos el prompt definitivo queda de la siguiente manera:

"You are a software design pattern expert tasked with implementing the specified design pattern in Java for the following use case: [Insert Use Case Here]. Please select and apply the [Insert Design Pattern Here] appropriately. Include Java code that clearly demonstrates the core structure of the pattern, ensuring the code is well-commented to aid understanding and educational purposes. Additionally, integrate basic error handling to enhance the robustness of the implementation. The comments should explain the functionality of each part of the code and how it addresses the given use case"

Iterar

Comentar que como en el caso anterior este prompt se ha refinado de forma iterativa, es decir, se han estado diseñando y modificando distintos prompts y entre estos se ha decidido utilizar este por los resultados que se han obtenido.

Los prompts se deben validar en base a los resultados que se proporcionan, por lo que si un prompt no genera las respuestas que se esperaban, se pueden optar por dos vías para mejorarlo. La primera y la más común es modificar este prompt cambiando los aspectos que se cree que pueden haber hecho que la respuesta no sea la deseada, tras esto, comprobar de nuevo mediante la experimentación si esta vez los resultados son los supuestos. La segunda opción, en caso de que se quiera optar por una vía más disruptiva sería reformular de nuevo el prompt desde el principio, esto se llevaría a cabo para cambiar aspectos fundamentales del prompt y por lo tanto de la respuesta que se quiere obtener.

Este proceso de diseño, refinamiento y validación del prompt que se realiza de forma iterativa no solo se puede aplicar en este ámbito de diseño de software, sino que se puede aplicar en la mayoría de procesos que implican el uso de Large Language Models. Por ejemplo, durante este trabajo se ha utilizado en la refactorización de código, en la selección de patrones de diseño y en la implementación de estos.

3.2.3. Experimentación con Patrones de Diseño

Durante la siguiente fase de experimentación, se recogerán los distintos resultados sobre qué patrones de diseño son recomendables en cada situación o caso de uso. Tras este análisis, si las elecciones han sido validadas, se obtendrán las estructuras de código que faciliten la implementación de los patrones. Además, comentar que los casos de uso serán descritos sin dar detalles de implementación para que las respuestas obtenidas no estén sesgadas.

Para validar estas respuestas y demostrar que esta metodología puede generar resultados rigurosos, se utilizarán casos de uso donde previamente se conozca cómo deben ser solucionados. De esta manera, podremos verificar si los resultados ofrecidos por los modelos son los esperados.

Cada conversación se llevará a cabo al menos cinco veces para asegurar la consistencia de las respuestas. Además, se utilizará el libro *Design Patterns: Elements of Reusable Object-Oriented Software* para confirmar la exactitud de la información proporcionada. Las soluciones a los casos de uso siempre estarán basadas en patrones de diseño de este libro. En algunos casos, se podrán utilizar distintos patrones para resolver la misma problemática; sin embargo, en otros, solo habrá una elección correcta.[6]

Además, como se mencionó al principio, se utilizarán tanto GPT-4 como Gemini. El uso de ambos modelos puede proporcionar al ingeniero de software diferentes perspectivas y opciones, lo que podría resultar, al tener más información, en un diseño de mayor calidad. Dado que el diseño de software es un campo tan abstracto, contar con estas variantes puede ser realmente útil.

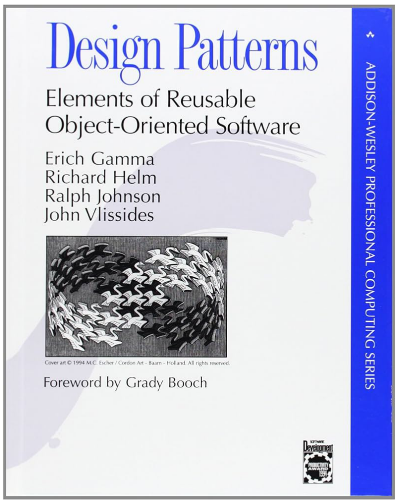


Figura 3.6: Design Patterns: Elements of Reusable Object-Oriented Software

3.2.4. Patrones Creacionales

Los Patrones Creacionales proporcionan soluciones en el ámbito de la creación de objetos, permitiendo desarrollar sistemas que no dependen de cómo se crean sus objetos.

Durante esta sección se analizará el comportamiento de GPT-4 y Gemini al enfrentarse a un caso de uso que requiere la aplicación de un patrón Creacional. El problema a resolver es el siguiente:

Let's consider the use case of a software system for report management in a company. This system needs to be able to generate different types of reports (such as financial reports, employee reports, and inventory reports), and these reports can vary in complexity and format.

Este caso de uso se puede resolver mediante el uso de distintos patrones creacionales. Las dos mejores opciones que se han considerado son el *Builder* y el *Método Fábrica*.

El patrón *Builder* es especialmente útil cuando el proceso de construcción de la entidad debe permitir distintas representaciones del objeto que se está creando. En este caso se podría utilizar este patrón para crear los distintos tipos de reportes, manejando adecuadamente que estos varíen en la complejidad y el formato.

El *Método Fábrica* también es una solución válida mediante la cual, se podrían construir los distintos tipos de reportes. Normalmente estos dos patrones de diseño se suelen poder utilizar en casos de uso parecidos. Sin embargo, dependiendo de cómo esté diseñado el sistema será más adecuado aplicar uno u otro.

GPT-4 - Selección

GPT-4 propone como solución utilizar el Método Fábrica, que como ya hemos visto, es un patrón ideal para el caso de uso propuesto. El método fábrica es un patrón que permite delegar a las subclases el tipo de objetos que se crean. Por lo que gracias a este patrón de diseño se podría construir una estructura que facilitase la creación de los distintos tipos de reportes. La respuesta que ha dado el modelo es la siguiente:[26]

*"**Recommended Pattern: Factory Method***

Identification and Description

The Factory Method pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern involves defining a "creator" interface with a factory method that returns an object of a "product" class. Subclasses of the creator will then override this factory method to create specific product instances. This design effectively separates product construction from its representation, making it easier to introduce new types of products without modifying existing client code.

Relevant Examples

In software development, the Factory Method pattern is commonly used in applications that manage or require the creation of various types of documents or reports. For instance, a business intelligence tool might use this pattern to generate different types of analytics reports such as sales performance, market analysis, or operational efficiency reports. Each type of report would be instantiated by a different subclass, ensuring that the specifics of report generation are encapsulated within each subclass.

Justification

Fit for Purpose

- *Scalability: The Factory Method allows new report types to be added without changing the existing codebase. As the company grows or changes, the system can scale to accommodate new types of reports.*
- *Maintainability: With each report type encapsulated in its own subclass, the system becomes easier to maintain. Changes to a specific report type only affect one subclass.*
- *Flexibility: The pattern provides flexibility in the way reports are created and formatted, enabling customization for client-specific needs without affecting the overall system architecture.*

Strategic Impact

Using the Factory Method enhances the system's strategy by aligning with long-term goals of adaptability and efficiency. It supports ongoing modifications and expansions in reporting capabilities, critical for dynamic business environments that require regular and systematic updates to decision-making tools.

Pros and Cons

Advantages

- *Modularity: Keeps the report creation process modular and independent from the client code that uses the reports.*

- *Extensibility: It is easy to introduce new types of reports without modifying existing factories.*
- *Loose Coupling: Reduces dependencies between the client code and the concrete implementations of reports, enhancing system robustness.*

Drawbacks

- *Complexity: Introducing many subclasses can make the system more complex and harder to understand initially.*
- *Overhead: May introduce unnecessary complexity if the number of report types is small and unlikely to change.*

Impact on Other Architectural Aspects

System-wide Effects

- *Technical Architecture: Can seamlessly integrate with other design patterns like Singleton (for managing a single instance of the report generator) and Builder (for constructing complex report objects).*
- *Business Processes: Supports the dynamic creation of reports tailored to specific business processes without disrupting existing operations.*

Potential Conflicts and Synergies

- *Conflicts: Could conflict with simple direct instantiation methods in parts of the system where such flexibility is not needed, potentially leading to over-engineering.*
- *Synergies: Works well with the Strategy pattern, where different algorithms for report generation could be encapsulated as strategy objects.*

Alternatives

Comparison

- *Abstract Factory: Useful for families of related products but more complex and might be overkill if the types of reports do not have multiple variants themselves.*
- *Builder: Suitable for constructing complex objects with many parameters, which might be more appropriate if the report formats are highly complex.*

Selection Rationale

While Builder and Abstract Factory are viable, the Factory Method is preferred here due to its simplicity and direct applicability to the problem of creating multiple, differing types of reports without requiring a major architectural overhaul. It strikes a balance between flexibility and complexity, making it ideal for the current needs while leaving room for future expansion and complexity.

This comprehensive solution anticipates not only immediate needs but also future requirements and challenges, ensuring that the report management system remains scalable, maintainable, and robust in the broader architectural context.“

Una de las partes más relevantes de la respuesta ofrecida por el modelo es la justificación de por qué se ha seleccionado este patrón de diseño y no otro. En este caso, se explica que el método fábrica es preferible debido a su simplicidad y su aplicación directa al problema sin requerir grandes cambios estructurales en el sistema.

Si bien es cierto que este patrón es fácil de aplicar y desarrollar, es responsabilidad del arquitecto de software tomar la decisión más conveniente a nivel de negocio. Aunque el Método Fábrica es adecuado para abordar la problemática planteada, el enunciado menciona que los reportes pueden variar en complejidad y formato. Esta dificultad adicional podría requerir la aplicación de otro patrón de diseño, como el patrón Builder.

Además, hay partes de la respuesta en las cuales se podría especificar más, es decir, en ocasiones, el modelo proporciona información que quizás es demasiado genérica, pudiendo profundizar más en estos aspectos. Sobre todo, esto se comprueba al comparar la respuesta del modelo con la descripción que hay de este patrón en el libro *Design Patterns: Elements of Reusable Object-Oriented Software*[6].

Cabe mencionar que esta problemática se le ha planteado al modelo varias veces, siempre obteniendo la misma solución. Como ya se ha comentado, los modelos de lenguaje no son deterministas, por lo que pueden producir distintas respuestas para una misma petición. El modelo, en este caso, siempre ha seleccionado el método fábrica frente a los demás patrones.

GPT-4 - Implementación

Como la solución propuesta por el modelo ha sido validada se puede pasar a la segunda parte del proceso. Es hora de ver cómo GPT-4 diseña la estructura de código de este patrón. El lenguaje de programación escogido ha sido Java, por su relación con la programación orientada a objetos y con el grado cursado. La solución propuesta ha sido la siguiente:

```java
// Abstract class defining a general report
abstract class Report {
    // Method to generate the report content, to be implemented by
        subclasses
    public abstract String generateContent();

    // Print the generated report content
    public void displayReport() {
        System.out.println(this.generateContent());
    }
}
```

```java
// Concrete implementation of a Financial Report
class FinancialReport extends Report {
    @Override
```

```java
    public String generateContent() {
        // Detailed financial report generation logic
        return "Financial Report Content";
    }
}

// Concrete implementation of an Employee Report
class EmployeeReport extends Report {
    @Override
    public String generateContent() {
        // Detailed employee report generation logic
        return "Employee Report Content";
    }
}

// Concrete implementation of an Inventory Report
class InventoryReport extends Report {
    @Override
    public String generateContent() {
        // Detailed inventory report generation logic
        return "Inventory Report Content";
    }
}
```

```java
// Interface for creating a report
interface ReportFactory {
    Report createReport();
}
```

```java
// Concrete factory to create Financial Reports
class FinancialReportFactory implements ReportFactory {
    @Override
    public Report createReport() {
        return new FinancialReport();
    }
}

// Concrete factory to create Employee Reports
class EmployeeReportFactory implements ReportFactory {
    @Override
    public Report createReport() {
        return new EmployeeReport();
    }
}

// Concrete factory to create Inventory Reports
class InventoryReportFactory implements ReportFactory {
    @Override
    public Report createReport() {
        return new InventoryReport();
    }
}
```

```java
// Client code using the factories
public class ReportManager {
    public static void main(String[] args) {
        ReportFactory financialFactory = new FinancialReportFactory();
        ReportFactory employeeFactory = new EmployeeReportFactory();
```

```
        ReportFactory inventoryFactory = new InventoryReportFactory();

        try {
            // Generate different types of reports
            Report financialReport = financialFactory.createReport();
            Report employeeReport = employeeFactory.createReport();
            Report inventoryReport = inventoryFactory.createReport();

            // Display reports
            financialReport.displayReport();
            employeeReport.displayReport();
            inventoryReport.displayReport();
        } catch (Exception e) {
            System.err.println("Error generating report: " + e.
                getMessage());
        }
    }
}
```

Al analizar el diseño y la estructura del patrón, se puede destacar que uno de los aspectos más interesantes de este patrón es que la fábrica que crea los productos podría contener lógica adicional que involucre a los objetos que crea. Sin embargo, el modelo no ha conseguido diseñar esto tras varias ejecuciones. A pesar de este detalle, la respuesta es bastante adecuada.

Si bien es cierto que esta solución se tendría que adaptar al sistema en el que se requiera integrar, como visualización de la estructura general del patrón es una respuesta muy correcta.

Los comentarios explican adecuadamente en qué consiste cada elemento, lo que facilita la comprensión del código. Además, si el usuario que lleva a cabo este proceso ha estudiado y conoce los conceptos que se han comentado en la sección **2. *Fundamentos***, le resultará fácil adaptar esta estructura para integrarla con su software.

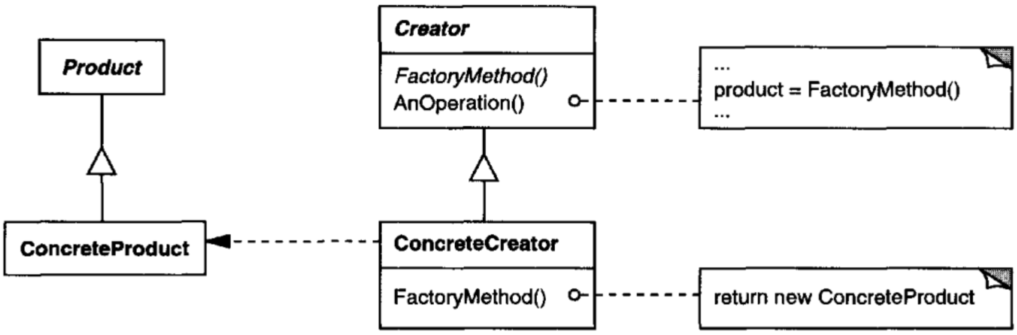


Figura 3.7: Estructura del patrón Factory Method[6]

La estructura del patrón se implementa de forma adecuada relacionando además los elementos del código con las cualidades del caso de uso. Todos estos aspectos hacen que la solución completa que ha propuesto GPT-4 sea muy significativa, incluso a pesar de no ser perfecta.

Gemini - Selección

El modelo de Google propone utilizar el patrón Estrategia, que es un patrón de comportamiento utilizado para definir familias de algoritmos. Sin embargo, este patrón no está destinado a la creación de objetos, por lo que la respuesta en este caso es incorrecta.[27]

El modelo sugiere que el patrón Estrategia podría utilizarse para definir distintos algoritmos que construyan cada tipo de reporte. Aunque es posible aplicar este patrón de esa manera, no sería la solución más adecuada. El patrón Estrategia está diseñado para el intercambio flexible de algoritmos, no para la construcción de objetos complejos. Aquí se puede observar el razonamiento del modelo:

Recommended Pattern: Strategy Pattern

Identification and Description:

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the dynamic selection of an algorithm at runtime.

The structure involves three elements:

- *Strategy Interface: Defines the common interface for all supported algorithms (report types).*
- *Concrete Strategies: Implementations of the Strategy Interface, each representing a specific report type (financial report, employee report, inventory report).*
- *Context: Holds a reference to a Strategy object and uses it to invoke the specific report generation logic.*

Relevant Examples:

- *OpenOffice/LibreOffice: These office suites use the Strategy pattern for various functionalities, including document formatting. Different filters are implemented as concrete strategies, allowing users to choose the desired format (PDF, DOCX, etc.) on the fly.*
- *E-commerce Platforms: Product sorting algorithms in e-commerce platforms can be implemented using the Strategy pattern. Different sorting strategies (price, popularity, etc.) can be offered to users for a customized shopping experience.*

Justification:

Fit for Purpose:

The Strategy pattern is ideal for the report management system because:

- *Scalability: Adding new report types is simple. You only need to create a new concrete strategy implementing the report generation logic. No modifications to the existing context or other strategies are required.*
- *Maintainability: Code specific to each report type is encapsulated within its concrete strategy, making it easier to understand, modify, and test.*

- *Flexibility: The system can dynamically choose the appropriate strategy based on user selection or pre-defined criteria.*

Strategic Impact:

The Strategy pattern promotes long-term maintainability and flexibility. The core system logic remains unchanged with new report types, allowing the system to adapt to evolving reporting needs.

Pros and Cons:

Advantages:

- *Promotes code reusability and reduces code duplication.*
- *Simplifies adding new report types.*
- *Enhances code maintainability and testability.*
- *Provides flexibility by allowing dynamic selection of report generation logic.*

Drawbacks:

- *Introduces an additional layer of abstraction, potentially increasing initial development complexity.*
- *May require additional effort to manage a growing number of concrete strategies.*

Impact on Other Architectural Aspects:

System-wide Effects:

- *Technical Architecture: The Strategy pattern interacts well with layered architectures and separates the report generation logic from the report presentation layer. This allows for independent development and maintenance of each layer.*
- *Business Processes: The pattern facilitates the accommodation of evolving reporting needs without impacting business processes that rely on report data.*

Potential Conflicts and Synergies:

- *Factory Pattern: The Strategy pattern can be combined with the Factory Pattern to automate the creation of concrete strategies based on configuration or user selection.*
- *State Pattern: If reports require different data retrieval or processing logic based on their type, the Strategy pattern can be used in conjunction with the State Pattern to encapsulate state-specific behavior within each concrete strategy.*

Alternatives:

- *If/Else Statements: A basic approach could involve using conditional statements to handle different report types. However, this leads to code duplication, reduces maintainability, and makes adding new report types cumbersome.*

Selection Rationale:

The Strategy pattern is preferable in this scenario due to its scalability, maintainability, and flexibility. It allows for a clean separation of concerns and simplifies the addition of new report types without affecting the core system logic. This is crucial for a report management system that needs to adapt to evolving reporting requirements.

A pesar de que la respuesta no sea correcta, considerar estas opciones puede abrir nuevas líneas de pensamiento y conducir a soluciones mejores. También lleva a reflexionar sobre por qué ciertos patrones no deben utilizarse en determinados escenarios. Como se ha mencionado en diversas ocasiones, estos modelos no son perfectos y pueden cometer errores. Por lo tanto, es crucial verificar y analizar siempre la solución que proponen, ya que podría ser incorrecta.

Esta conversación también se ha ejecutado en diversas ocasiones y sorprende que siempre se haya obtenido la misma solución, siendo claro que el patrón Estrategia no está enfocado en la creación de objetos. Además, se han intentado realizar pequeñas modificaciones en la descripción del caso de uso para intentar que el modelo pueda comprender completamente el problema que se le está planteando. A pesar de esto, no ha sido posible cambiar la respuesta que ofrecía Gemini.

Como la respuesta que ha proporcionado el modelo no es correcta, no es necesario continuar con el proceso para implementar la estructura de este patrón de diseño.

3.2.5. Patrones Estructurales

En esta sección se analizará el comportamiento de GPT-4 y Gemini al enfrentarse a un problema que requiere la aplicación de un patrón Estructural, en concreto el patrón Decorador. El caso de uso a resolver es el siguiente:[26, 27]

In this use case, we are developing a sophisticated graphics editor application. This application will include a variety of basic shapes such as squares, circles, and more. Additionally, it will offer the flexibility to extend these shapes with new functionalities, including options for filling them with colors or adding decorative borders.

En un editor gráfico son muchos los patrones que se podrían implementar, sin embargo, para extender el comportamiento de las formas geométricas que puedan ser utilizadas en el editor, el patrón Decorador es el indicado, puesto que permite añadir nuevas funcionalidades de forma dinámica a un objeto.

Selección

El análisis de los resultados se abordará de forma conjunta, debido a que la media de las respuestas que se han obtenido por ambos modelos son muy similares, en la mayoría de casos, escogiendo el patrón Decorador. Por ello el análisis se centrará en las diferencias y similitudes entre las respuestas ofrecidas por estos modelos, ya que ambos han respondido correctamente a la petición.

Cada respuesta tiene sus ventajas en comparación con el otro modelo. La respuesta ofrecida por GPT-4 se centra más en la argumentación sobre porqué se ha escogido esa solución y no otra estructura que pudiese abordar esa misma problemática. La respuesta muestra cómo el modelo es capaz de racionalizar las decisiones que toma y además justificarlas:

*"**Recommended Pattern: Decorator Pattern***

Identification and Description

The Decorator pattern is ideal for the described graphics editor application. This structural design pattern allows for the addition of new functionality to objects dynamically, without altering the structure of the objects being extended. In essence, the Decorator pattern involves a set of decorator classes that are used to wrap concrete components.

The core principle behind the Decorator pattern is that it uses composition instead of inheritance to extend the functionalities of objects. A base component interface defines the standard functionalities, while concrete implementations of this interface represent basic shapes like squares and circles. Decorator classes implement the same interface and hold a reference to a component object, adding new functionalities to it.

Relevant Examples

1. *Adobe Photoshop: Photoshop uses a similar approach where features like filters and effects are added dynamically to images without changing the original image structure.*
2. *Java I/O Classes: In Java's I/O system, decorators dynamically add responsibilities like buffering or new data handling capabilities to the streams.*

Justification

Fit for Purpose

The Decorator pattern provides the flexibility needed for the application by allowing basic shapes to be extended dynamically with new functionalities such as color filling or decorative borders without modifying the shape classes themselves. This approach ensures:

- *Scalability: New decorators can be added without affecting existing code.*
- *Maintainability: Changes to shape or decoration implementations can be made independently.*
- *Flexibility: Easily mix and match different decorations.*

Strategic Impact

Implementing the Decorator pattern will align with the strategic goal of creating a robust, extensible graphics editor. It supports innovation in functionalities without requiring significant changes to the core application architecture [...]

Alternatives

Comparison

- *Strategy Pattern: Could also be used to change the behavior of shapes dynamically, but it doesn't allow for the stacking of behaviors as Decorator does.*
- *Composite Pattern: Useful for treating individual objects and compositions uniformly, but less focused on adding responsibilities dynamically.*

Selection Rationale

While the Strategy and Composite patterns offer valid solutions for dynamic behavior and grouping objects, respectively, the Decorator pattern is chosen for its superior ability to add responsibilities dynamically without changing the underlying objects, making it the most suitable for the evolving nature of a graphics editor application. This choice reflects a focus on extensibility and adaptability in the application's design."

Los razonamientos ofrecidos por GPT-4 son excelentes, justificando por qué el Decorador es la mejor solución para este caso de uso, y añadiendo además cómo otros patrones podrían mejorar el comportamiento del sistema con respecto a otras funcionalidades, y describiendo cómo este patrón utiliza la composición como estrategia a la hora de añadir nueva funcionalidad a un objeto.

Si bien es cierto que la respuesta del modelo de OpenAI es muy significativa, el modelo de Google tampoco se queda atrás. Una de las partes más interesantes en la respuesta de Gemini, es la siguiente. Aquí se describen algunos detalles de una posible implementación de este patrón. Esto es muy interesante ya que aunque esta estructura es necesaria adaptarla al sistema en el que se vaya a utilizar, esta respuesta puede dar alguna idea de cómo se podría abordar la problemática a nivel de código.

"The Decorator pattern is a structural design pattern that allows you to attach additional functionalities to an object dynamically, without altering its original structure. It provides a flexible way to extend the behavior of an object at runtime.

Here's how it works:

- *Base Shape Class: This class defines the core functionalities of a shape, such as drawing itself.*
- *Concrete Shape Classes: These classes extend the base shape class and represent specific shapes like circles, squares, etc. They implement the drawing behavior specific to their shape.*
- *Decorator Classes: These classes wrap around existing shape objects and add new functionalities. They can be stacked to provide layered functionalities.*

For instance, a ColoredShapeDecorator could extend a circle and add a color property, while a BorderedShapeDecorator could further extend the colored circle to include a border."

Ambos modelos coinciden en la justificación, así como en las ventajas y desventajas de usar ese patrón, ya que los patrones de diseño poseen propiedades inherentes, tanto positivas como negativas. Los patrones son soluciones excelentes que resuelven numerosos problemas de manera efectiva si son utilizados correctamente. Sin embargo, también tienen inconvenientes, por ejemplo, una sobreutilización del patrón Decorador podría llevar a problemas de rendimiento en el sistema.

Durante la experimentación se ha detectado que para este caso de uso tanto el modelo de OpenAI como el de Google no porporcionaban siempre la misma respuesta. El patrón Compuesto y el patrón Estrategia, que son mencionados como posibles alternativas en ambas soluciones, son otros patrones que han sido propuestos.

Como el caso de uso no está descrito en profundidad, los modelos han ofrecido varios patrones para solucionar distintas funcionalidades que se podrían utilizar en un editor gráfico. Esto en una fase temprana del diseño de un sistema podría ser muy útil, ya que puede proporcionar una visión más general del sistema que se va a implementar, además de poder contemplar como cada patrón podría atajar un problema distinto.

Implementación

En la segunda parte del proceso se va a analizar cuáles han sido las respuestas que han proporcionado tanto GPT-4 como Gemini en cuanto a las estructuras de código que han generado. Se comenzará con la solución del modelo de OpenAI:

```java
// Step 1: The 'Shape' interface, representing components that can be
    decorated.
public interface Shape {
    void draw();
}
```

```java
// Step 2: Concrete Components implementing the Shape interface.
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}

public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square.");
    }
}
```

```java
// Step 3: Abstract Decorator class implementing the Shape interface.
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

```

```
    public void draw() {
        decoratedShape.draw();
    }
}
```

```
// Step 4: Concrete Decorator classes adding specific functionalities.
public class ColoredShapeDecorator extends ShapeDecorator {
    private String color;

    public ColoredShapeDecorator(Shape decoratedShape, String color) {
        super(decoratedShape);
        this.color = color;
    }

    @Override
    public void draw() {
        decoratedShape.draw(); // Draw the original shape
        System.out.println("Filling color: " + color); // Additional functionality
    }
}

public class BorderShapeDecorator extends ShapeDecorator {
    private int borderWidth;

    public BorderShapeDecorator(Shape decoratedShape, int borderWidth) {
        super(decoratedShape);
        this.borderWidth = borderWidth;
    }

    @Override
    public void draw() {
        decoratedShape.draw(); // Draw the original shape
        System.out.println("Adding border with width: " + borderWidth + "px"); // Additional functionality
    }
}
```

```
// Main class to demonstrate the use of decorators.
public class DecoratorDemo {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape coloredCircle = new ColoredShapeDecorator(circle, "Red");
        Shape borderedColoredCircle = new BorderShapeDecorator(coloredCircle, 5);

        borderedColoredCircle.draw(); // This will draw a circle with red color and a border.
    }
}
```

La implementación realizada por GPT-4 es prácticamente perfecta, la estructura se aplica y adapta a lo descrito en el caso de uso y construye una estructura que implementa el patrón Decorador al detalle. Como crítica se podría mencionar que los comentarios podrían ser más descriptivos, ya que la descripción de los pasos que realiza son la prácticamente una narración de los nombres de las clases.

A continuación se muestra la implementación que ha hecho el modelo de Google, para posteriormente, comentar qué solución parece más apropiada y que atributos destacan en cada respuesta.

```java
// Defines the core functionalities of a shape
public interface Shape {
  void draw(); // Method to draw the shape
}
```

```java
// Basic implementation of Square shape
public class Square implements Shape {
  private int size;

  public Square(int size) {
    if (size <= 0) {
      throw new IllegalArgumentException("Size must be positive");
    }
    this.size = size;
  }

  @Override
  public void draw() {
    System.out.println("Drawing Square with size: " + size);
    // Draw logic for square using size
  }
}

// Similar implementation for Circle and other basic shapes
```

```java
// Abstract decorator class providing a wrapper for shapes
public abstract class ShapeDecorator implements Shape {
  protected Shape decoratedShape; // Reference to the shape being
      decorated

  public ShapeDecorator(Shape decoratedShape) {
    this.decoratedShape = decoratedShape;
  }

  @Override
  public void draw() {
    decoratedShape.draw(); // Delegate drawing to the decorated shape
  }
}
```

```java
// Decorator for adding color fill
public class ColoredShape extends ShapeDecorator {
  private String color;

  public ColoredShape(Shape decoratedShape, String color) {
    super(decoratedShape);
    this.color = color;
  }

  @Override
```

```java
  public void draw() {
    super.draw(); // Draw the decorated shape first
    System.out.println("Filling the shape with color: " + color);
  }
}

// Decorator for adding borders
public class BorderShape extends ShapeDecorator {
  private int borderWidth;

  public BorderShape(Shape decoratedShape, int borderWidth) {
    super(decoratedShape);
    this.borderWidth = borderWidth;
  }

  @Override
  public void draw() {
    super.draw(); // Draw the decorated shape first
    System.out.println("Adding border with width: " + borderWidth);
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    Shape square = new Square(5); // Create a basic square
    square.draw(); // Output: Drawing Square with size: 5

    Shape coloredSquare = new ColoredShape(square, "Red"); // Add color decoration
    coloredSquare.draw();
    // Output:
    // Drawing Square with size: 5
    // Filling the shape with color: Red

    Shape decoratedSquare = new BorderShape(coloredSquare, 2); // Add border decoration
    decoratedSquare.draw();
    // Output:
    // Drawing Square with size: 5
    // Filling the shape with color: Red
    // Adding border with width: 2
  }
}
```

La respuesta que ofrece Gemini es prácticamente idéntica a la que ha propuesto GPT-4, diferenciando el aspecto de los comentarios que se mencionaba anteriormente, donde el modelo desarrollado por Google, en esta ocasión, es superior. Ambas soluciones son correctas tras haberlas validado, la estructura y el diseño es el indicado implementando al detalle el patrón Decorador con respecto a la misma problemática.

Los dos modelos se han comportado de manera muy similar al enfrentarse al mismo caso de uso, sin embargo, esto no es lo común. Los modelos están entrenados con distintas bases de datos, soportan arquitecturas diferentes y, por ello, otorgan respuestas desiguales en la mayoría de casos.

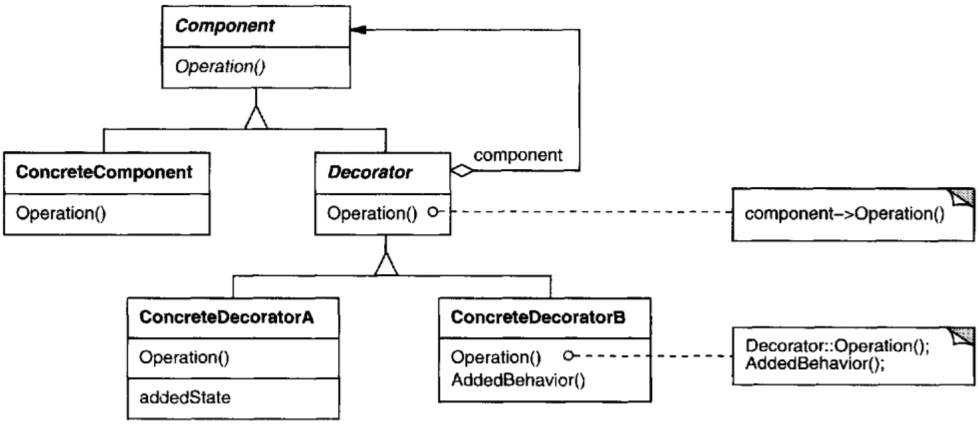


Figura 3.8: Estructura del patrón Decorador[6]

El hecho de que ambos modelos se hayan comportado de forma tan similar, podría reforzar el pensamiento del ingeniero al adaptar la solución que los modelos le proponen. No obstante, este hecho no debe influenciar las decisiones de diseño que realice el ingeniero de software, su opinión y criticidad debe permanecer durante todo el proceso.

3.2.6. Patrones de Comportamiento

En esta sección se analizará el comportamiento de GPT-4 y Gemini al enfrentarse a un problema que requiere la aplicación de un patrón de Comportamiento. El caso de uso a resolver es el siguiente:

Implement a functionality that allows users to undo recent changes in a text editor. This feature should enable users to reverse their latest modifications, whether they are additions, deletions, or alterations of text. By tracking the history of changes, the undo function should provide the capability to step back through these changes one by one, restoring the text to previous states. This will enhance the user's ability to correct mistakes and refine their text without losing any recent work. The implementation should be efficient and seamless, ensuring a smooth user experience.

Este caso de uso específico presenta dos posibles soluciones. La primera, y también la más intuitiva, es la utilización del patrón Memento. Este patrón de diseño permite almacenar el estado de un objeto, facilitando su restauración en un momento posterior. La segunda solución es el uso del patrón Comando, que encapsula acciones en un objeto, permitiendo tanto ejecutar como deshacer dichas acciones mediante los propios Comandos.

Escoger un patrón u otro dependerá del sistema en el que se quiera integrar esta funcionalidad. En un sistema donde los cambios sean simples o lineales el patrón Comando destacará, sin embargo si lo que se busca es guardar estados complejos o difíciles de reconstruir, Memento es la mejor elección.

GPT-4 - Selección

El patrón de diseño propuesto por el modelo desarrollado por OpenAI es el patrón Comando. La respuesta es clara y concisa, explicando las ventajas y desventajas inherentes a este patrón. Se destaca la última parte, donde el modelo explica por qué el patrón Comando debería ser considerado como una opción a destacar.[26]

Después de haber realizado varias solicitudes en relación con este caso de uso, el modelo GPT-4 ha propuesto tanto la solución mediante el patrón Memento como mediante el uso del patrón Comando. En este caso, se ha decidido analizar esta última opción, ya que ha sido la más frecuentemente recomendada. La respuesta media es la siguiente:

*"**Recommended Pattern: Command Pattern***

Identification and Description

The Command pattern is a behavioral design pattern that encapsulates all the information needed to perform an action or trigger an event into a single command object. This pattern decouples the object that invokes the operation from the one that knows how to execute it, which enables a wide range of features like transactional behavior and undo functionality.

In the context of a text editor, the Command pattern involves creating command objects for each action that can modify the text, such as InsertText, DeleteText, and ReplaceText. Each command object stores the state necessary to both perform the action and reverse it (undo).

Relevant Examples

Microsoft Word and Google Docs implement undo functionalities that are very likely based on the Command pattern. For instance, when a user types, deletes, or formats text, each of these actions can be encapsulated in command objects that can be executed to apply the change and also undone to revert the document to its previous state.

Justification

Fit for Purpose

- *Scalability: The Command pattern allows easy addition of new commands without changing the existing codebase, making the system highly scalable.*
- *Maintainability: Commands centralize all action logic in individual command classes, simplifying maintenance and enhancements.*
- *Flexibility: The pattern supports the dynamic addition and removal of commands, enabling flexible operations such as macro recording and complex undo sequences.*

Strategic Impact

Using the Command pattern aligns with long-term strategies of enhancing user experience and robustness in software applications, particularly in content creation tools like text editors. It facilitates the implementation of additional features like redo, repeat last action, and even complex scripted actions without significant modifications to the core application architecture.

Pros and Cons

Advantages

- *Undo/Redo Capability: Fundamental for user error correction.*
- *Separation of Concerns: Decouples the code that issues a command from the code that executes it, enhancing code organization and reducing dependencies.*
- *Extensibility: New commands can be added with minimal impact on the existing system.*

Drawbacks

- *Complexity: The implementation of the Command pattern can add complexity to the code, particularly in tracking the history and state needed for undo operations.*
- *Overhead: Storing command objects for every action might increase memory usage, especially with large documents or long editing sessions.*

Impact on Other Architectural Aspects

System-wide Effects

The Command pattern interacts harmoniously with other architectural elements, such as Composite patterns (for macro commands) and Memento patterns (for enhanced state saving and restoration). It generally fits well within MVC architectures, where it can be used to handle user input and manipulate model state.

Potential Conflicts and Synergies

- *Conflicts: If not well managed, the command history can grow large and affect performance. Proper management strategies must be devised to mitigate this.*
- *Synergies: Combining the Command pattern with the Observer pattern can enhance responsiveness, allowing other parts of the system to react to state changes initiated by commands.*

Alternatives

Comparison

- *Memento Pattern: Instead of commands storing the actions, Memento captures and restores the object's state. This might involve more overhead in capturing the entire state rather than incremental changes.*
- *Strategy Pattern: Could be used to change the behavior of the text editor dynamically but does not inherently support undo functionality.*

Selection Rationale

The Command pattern is preferred over alternatives primarily due to its direct support for undoable actions and its ability to encapsulate all action-related data in a modular way, which the alternatives lack. This makes it the most suitable choice for implementing *a robust and user-friendly undo feature in a text editor.“*

GPT-4 ofrece una respuesta muy significativa en la cual razona y justifica porqué la elección de este patrón de diseño puede solucionar el problema propuesto. Menciona las ventajas y desventajas de forma correcta, y además, proporciona intuiciones sobre cómo se podría implementar este patrón en este caso de uso. Por ejemplo, se comenta la necesidad de mantener un historial con las operaciones que se han realizado, para así, poder deshacerlas.

GPT-4 - Implementación

Gracias a las intuiciones conseguidas durante esta experimentación, será más fácil poder validar la segunda parte del proceso en la que el modelo de OpenAI ha proporcionado el código siguiente:

```java
interface Command {
    void execute();
    void undo();
}
```

```java
// Adds text to the document
class AddCommand implements Command {
    private TextEditor editor;
    private String text;

    public AddCommand(TextEditor editor, String text) {
        this.editor = editor;
        this.text = text;
    }

    public void execute() {
        editor.addText(text);
    }

    public void undo() {
        editor.removeText(text.length());
    }
}

// Deletes text from the document
class DeleteCommand implements Command {
    private TextEditor editor;
    private String removedText;

    public DeleteCommand(TextEditor editor, int length) {
        this.editor = editor;
        this.removedText = editor.getText().substring(editor.getText().length() - length);
    }

    public void execute() {
        editor.removeText(removedText.length());
    }

```

```
    public void undo() {
        editor.addText(removedText);
    }
}
```

```
class TextEditor {
    private StringBuilder text = new StringBuilder();

    public void addText(String text) {
        this.text.append(text);
    }

    public void removeText(int length) {
        if (length > text.length()) {
            throw new IllegalArgumentException("Remove length exceeds text length.");
        }
        text.delete(text.length() - length, text.length());
    }

    public String getText() {
        return text.toString();
    }
}
```

```
import java.util.Stack;

class CommandManager {
    private Stack<Command> commandStack = new Stack<>();

    public void executeCommand(Command command) {
        command.execute();
        commandStack.push(command);
    }

    public void undo() {
        if (!commandStack.isEmpty()) {
            Command command = commandStack.pop();
            command.undo();
        } else {
            System.out.println("No commands to undo.");
        }
    }
}
```

```
public class TextEditorApp {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        CommandManager manager = new CommandManager();

        // Execute some commands
        manager.executeCommand(new AddCommand(editor, "Hello"));
        manager.executeCommand(new AddCommand(editor, " World"));
        System.out.println("Text after additions: " + editor.getText());

        // Undo last add
        manager.undo();
```

```
        System.out.println("Text after undo: " + editor.getText());

        // Continue with other operations
    }
}
```

GPT-4 ha resuelto este caso de uso realizando una implementación casi completa de cómo debería integrarse este patrón de diseño en un editor de texto. No solo ha sido capaz de diseñar una estructura del patrón, sino que ha adaptado este diseño al caso de uso que se estaba proponiendo, cambiando parte de la estructura original para que la implementación sea funcional.

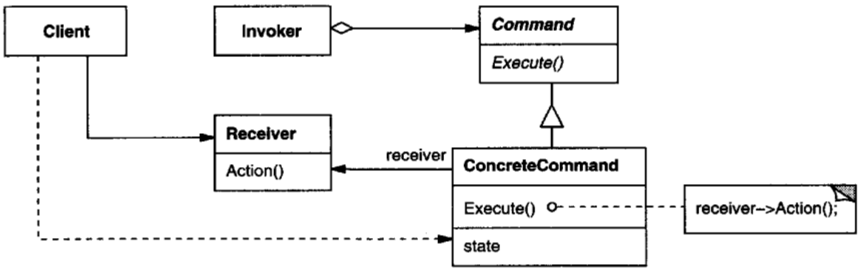


Figura 3.9: Estructura del patrón Comando[6]

La respuesta es adecuada, hay divergencias con respecto a cómo se deben implementar algunos aspectos de este patrón, sin embargo, esto no significa que la solución propuesta por el modelo sea incorrecta. En numerosas ocasiones, es necesario adaptar la estructura de un patrón de diseño para que este se pueda integrar de manera apropiada en los sistemas que se estén desarrollando.

De nuevo como aspecto crítico que no está cumpliendo el modelo GPT-4, es la adición de comentarios que expliquen qué funcionalidad cumple cada elemento que se está desarrollando. Si fuese un código en producción se podría apreciar que el código no estuviese repleto de comentarios explicativos, no obstante, durante este proceso se busca la visualización de cómo estas estructuras podrían integrarse en el software, siendo los comentarios de gran ayuda.

Gemini - Selección

Gemini, el modelo diseñado por Google, presenta una alternativa distinta. El patrón de diseño propuesto es el Memento, que es capaz de almacenar el estado de un objeto, permitiendo restablecerlo en otro momento. Este patrón ofrece una estrategia diferente a la propuesta por el modelo GPT-4, pero no por ello es menos efectiva. Esta solución se adecúa perfectamente al caso de uso propuesto y, por lo tanto, debe ser considerada.[27]

*"**Recommended Pattern: Memento Pattern***

Identification and Description:

The Memento pattern is a behavioral design pattern that encapsulates an object's internal state and provides a way to restore it to that state later. In simpler terms, it creates *a snapshot of an object's state that can be used to revert to a previous state without modifying the object itself.*

The Memento pattern involves three main elements:

- *Originator: The object whose state needs to be stored. In our case, it's the text editor.*
- *Memento: The object that stores the snapshot of the originator's state. This could be a simple data structure containing the text content and other relevant information (e.g., cursor position).*
- *Caretaker: The object responsible for managing the Memento objects. It typically provides methods to create, store, and retrieve Memento objects.*

Relevant Examples:

- *Many text editors and word processing applications like Microsoft Word and Google Docs leverage the Memento pattern to implement undo/redo functionality. Each keystroke or action creates a new Memento object, allowing users to revert to previous states.*
- *Photo editing software like Adobe Photoshop utilizes the Memento pattern to store different versions of an image during editing. This allows users to go back to a previous version without losing work.*

Justification:

Fit for Purpose:

The Memento pattern is ideal for the undo/redo functionality in a text editor due to several reasons:

- *Scalability: The Memento pattern allows storing multiple Memento objects, enabling users to undo several actions. This provides a flexible way to manage undo history without modifying the core text editor functionality.*
- *Maintainability: By separating the state management from the core editing logic, the Memento pattern promotes cleaner code and simplifies future modifications to the undo/redo system.*
- *Flexibility: The Memento object can be designed to store different levels of detail, depending on the specific needs. It can encapsulate the entire text content or just the changes made in a single action.*

Strategic Impact:

The Memento pattern enhances the text editor's usability and user experience by providing a robust undo/redo mechanism. This allows users to experiment and explore different edits without the fear of losing work, fostering a more iterative and efficient editing workflow.

Pros and Cons

Advantages:

- *Easy to implement: The Memento pattern has a relatively simple structure and can be implemented efficiently in most programming languages.*
- *Decouples state management: It separates the state management logic from the core editing functionality, improving code maintainability.*
- *Flexible undo/redo history: The pattern allows for storing multiple Memento objects, enabling users to undo/redo multiple actions.*

Drawbacks:

- *Memory overhead: Storing a large number of Memento objects can consume memory, especially for complex documents with frequent edits.*
- *Performance considerations: Retrieving and applying Memento objects can introduce some overhead, impacting performance in highly intensive editing scenarios.*

Impact on Other Architectural Aspects:

System-wide Effects:

The Memento pattern primarily affects the text editor component but can have broader implications:

- *Technical Architecture: The Memento pattern can be integrated with the text editor's data persistence layer to allow saving and loading documents with undo/redo history.*
- *Business Processes: A robust undo/redo functionality can improve user productivity and reduce errors, potentially impacting business processes that rely on text editing tasks.*

Potential Conflicts and Synergies:

The Memento pattern generally works well with other design patterns. Here are some potential interactions:

- *Command Pattern: The Memento pattern can be combined with the Command pattern to store additional information about the editing actions within the Memento object. This can be useful for more granular undo/redo functionality.*
- *Observer Pattern: The Caretaker object can notify other parts of the system (e.g., a status bar) when the undo/redo state changes using the Observer pattern.*

Alternatives:

- *Command Pattern: While the Command pattern can be used to track editing actions, it doesn't directly address storing and restoring states. It would require additional mechanisms to achieve undo/redo functionality.*
- *State Pattern: The State pattern could be used to represent different states of the text editor (e.g., with formatting applied, without formatting). However, it wouldn't be ideal for managing a history of changes for undo/redo.*

Selection Rationale:

The Memento pattern is the most suitable choice for this use case due to its focus on state management and efficient undo/redo functionality. It provides a clear separation of concerns, scalability, and ease of implementation compared to the alternative patterns mentioned.“

La explicación y los argumentos presentes en la respuesta son correctos, con una estructura similar a la del modelo de OpenAI, aunque con algunas diferencias sutiles. Es notable cómo ambos modelos son capaces de justificar el uso de ciertos patrones, ofrecer sus ventajas y desventajas, y además proponer alternativas viables.

Ambos modelos han sido entrenados con grandes cantidades de datos, lo cual es un punto a favor. Al estar buscando la solución a un problema, es probable que en los datos con los que se entrenaron los modelos se encuentren varios casos similares, de los cuales el modelo puede extraer la solución más común y efectiva para el problema propuesto.

Para un ingeniero de software con experiencia en este tipo de problemas, estas soluciones pueden no resultar muy reveladoras. No obstante, el coste de oportunidad de utilizar estos modelos es tan bajo que merece la pena explorar las posibles alternativas o líneas de pensamiento que pueden ofrecer, ya que esto podría conducir a la consideración de otras alternativas y finalmente a la consecución de una solución más óptima y adecuada.

Gemini - Implementación

Dado que la respuesta obtenida también ha sido validada, se procede a realizar el segundo paso del proceso en el cual se analizará el código y la estructura que ha creado Gemini para implementar este patrón de diseño con respecto al caso de uso.

```java
public class TextEditor {

  private String text;
  private Stack<TextMemento> history; // Use Stack for efficient undo (LIFO)
  private int maxHistorySize; // Limit the number of undo states stored

  public TextEditor(int maxHistorySize) {
    this.text = "";
    this.history = new Stack<>();
    this.maxHistorySize = maxHistorySize;
  }

  public void setText(String text) {
    this.text = text;
    clearHistory(); // Clear history when setting new text
  }

  public String getText() {
    return text;
```

```
  }

  public void appendText(String newText) {
    createMemento(); // Capture state before append
    this.text += newText;
  }

  public void deleteText(int start, int end) throws
      IndexOutOfBoundsException {
    if (start < 0 || end > text.length() || start > end) {
      throw new IndexOutOfBoundsException("Invalid deletion range");
    }
    createMemento(); // Capture state before deletion
    this.text = text.substring(0, start) + text.substring(end);
  }

  public void undo() {
    if (!history.isEmpty()) {
      TextMemento memento = history.pop();
      this.text = memento.getText();
    } else {
      System.out.println("No more undo states available.");
    }
  }

  private void createMemento() {
    if (history.size() == maxHistorySize) {
      history.remove(0); // Maintain maximum history size
    }
    history.push(new TextMemento(text));
  }

  private static class TextMemento {
    private final String text;

    public TextMemento(String text) {
      this.text = text;
    }

    public String getText() {
      return text;
    }
  }
}
```

La implementación que ha realizado Gemini de este patrón no es la más apropiada, esto se debe a que ha realizado todo este patrón en una única clase. Es cierto que es una práctica común utilizar una clase anidada que mantenga el estado del objeto que se quiere poder restaurar. Sin embargo, toda la lógica que respecta al almacenamiento y administración de los Mementos podría encontrarse en una clase diferente, adhiriendo el código al principio de responsabilidad única.

A pesar de todo esto, la implementación podría servir perfectamente a un ingeniero experimentado para poder llevar a cabo la integración de este patrón con su sistema software. El hecho de que estos modelos proporcionen muchas respuestas idóneas es un aspecto muy positivo, pues relativizan la carga de trabajo

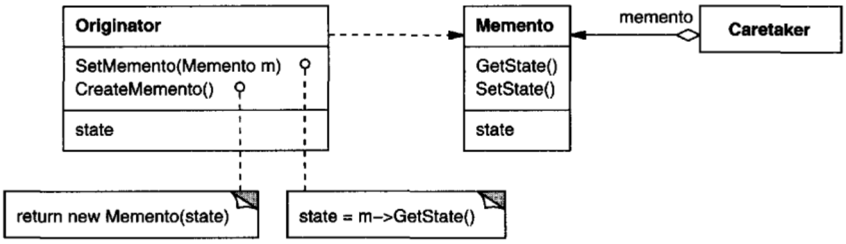


Figura 3.10: Estructura del patrón Memento[6]

y permiten diversificar esfuerzos. No obstante, siempre se ha de tener en cuenta que las respuestas pueden no ser del todo adecuadas, por este motivo, los ingenieros deben estar alerta ante las respuestas que puedan obtener por parte de estos modelos.

3.3 Consideraciones

La Inteligencia Artificial es una tecnología que agiliza procesos, optimiza recursos y ofrece resultados sorprendentes. Sin embargo, aunque muchas de las respuestas generadas por IA son sorprendentemente adecuadas, esto no implica que los ingenieros de software vayan a perder relevancia.

Aunque la situación global puede parecer compleja, estas tecnologías pueden conducir a nuevos avances, diseños e ideas, y por ello, deben ser aprovechadas. La IA debe percibirse como un copiloto, una herramienta en la cual apoyarse y con quien compartir la carga de trabajo.

Durante esta sección se ha destacado la importancia de los ingenieros y arquitectos de software porque la responsabilidad principal sigue recayendo en sus hombros. Deben adaptarse a las innovaciones tecnológicas y aprovecharlas para agilizar tareas, permitiendo que se realicen de manera más precisa y eficiente. Así, los ingenieros podrán dedicarse progresivamente a tareas más complejas. Estos avances deben capacitar al desarrollador para abordar y resolver problemas cada vez más complicados.

Además, se ha podido observar cómo los LLMs son capaces de refactorizar y elevar la calidad del código, eliminando la duplicidad y reduciendo métricas como la complejidad ciclomática. Y cómo es posible utilizarlos durante fases de diseño para asistir y brindar nuevas ideas y líneas de pensamiento que puedan mejorar la toma de decisiones.

Sin embargo, también se ha comprobado que no siempre tienen un comportamiento perfecto, especialmente con respecto a la elección de patrones de diseño, donde a pesar de que las respuestas en su gran mayoría eran correctas, algunas de las descripciones, argumentos o justificaciones podían ser algo genéricas.

Para intentar mejorar ese comportamiento, durante la siguiente sección se intentará refinar aún más este proceso mediante la construcción de un modelo GPT-4 especializado en patrones de diseño. Tras esta construcción, se revistarán uno a uno los casos de uso para explorar las distintas mejoras que pueden aparecer con respecto a los resultados vistos en este análisis.

CAPÍTULO 4
Construcción de modelos GPT

Después de analizar el comportamiento de los modelos GPT-4 y Gemini en diversas situaciones relacionadas con el diseño de software, se propone una estrategia diferente para lograr soluciones más descriptivas y precisas. En esta nueva aproximación, se desarrollarán varios modelos especializados en los patrones de diseño para lograr este objetivo.

También se ha considerado abordar la refactorización, sin embargo, se requiere centrar el foco en una de estas dos áreas por lo amplias que son. Tras considerarlo, finalmente, se ha seleccionado el estudio de los patrones de diseño por su mayor cercanía y relación con el ámbito del diseño de software.

Para la construcción de estos modelos, se utilizará una herramienta desarrollada por OpenAI llamada GPT Builder. Esta herramienta es un modelo de lenguaje que está especializado en la construcción de modelos GPT. El objetivo es conseguir mediante este nuevo proceso mejorar la calidad general de los resultados en aspectos como la descripción de los patrones de diseño o su implementación, buscando respuestas más específicas y consistentes.[29]

El capítulo comenzará con la construcción de dos modelos GPT. El primero estará basado en la selección de patrones de diseño, imitando el comportamiento que se buscaba en el capítulo anterior al seleccionar un patrón para un caso de uso concreto.

El segundo modelo que se construye, busca complementar el proceso definido en el capítulo anterior, estando especializado en la implementación de las soluciones que se obtengan del primer modelo. De esta forma, mediante la construcción de estos modelos especializados en estos ámbitos, se pretende incrementar la calidad de las soluciones que se obtuvieron anteriormente.

El objetivo de estos modelos es proporcionar respuestas más específicas, detalladas y consistentes, asegurando que la información transmitida sea más relevante y contextualizada. Aunque los resultados obtenidos anteriormente son interesantes, algunas respuestas omitían conceptos importantes, careciendo en ocasiones de especificidad. Una respuesta ideal debería profundizar más en la aplicación de estos patrones y su impacto en la mantenibilidad y escalabilidad del software.

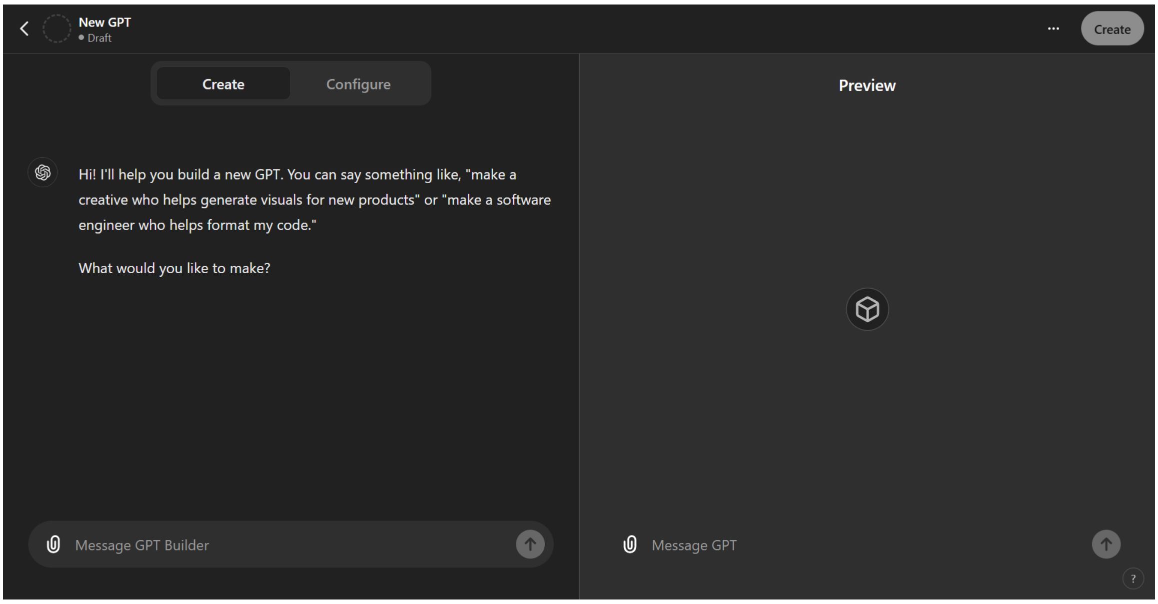
Tras la construcción de estos modelos, se revisitarán los casos de uso vistos en el capítulo anterior, analizando las soluciones que propongan estos modelos utilizando patrones de diseño creacionales, estructurales y de comportamiento respectivamente. Todo esto con el objetivo de observar cuáles son las mejoras con respecto a los resultados obtenidos anteriormente, los aspectos más relevantes de las nuevas respuestas y cómo estos nuevos modelos pueden ser capaces de ofrecer unos resultados más precisos.

Finalmente, se sacarán las conclusiones pertinentes de toda esta experimentación, explorando si los resultados obtenidos han logrado cumplir las expectativas, cumpliendo así uno de los objetivos de este trabajo.

Cabe mencionar que los resultados que se vayan a obtener con este nuevo proceso no distarán demasiado de los que ya se han obtenido en la sección anterior. El LLM que va a procesar las respuestas seguirá siendo GPT-4, por lo que el contenido será similar, sin embargo, se busca conseguir esa calidad extra en las respuestas que consiga hacer la tarea del ingeniero de software más liviana.

4.1 Proceso de construcción - Selección de Patrones

El proceso de construcción del modelo especializado GPT-4 se realizará utilizando la herramienta ya mencionada GPTBuilder. Esta herramienta permite crear modelos personalizados mediante una conversación, en la que GPTBuilder asistirá en su desarrollo. A través de las solicitudes, se especificará el rol que debe asumir el modelo, su propósito, las funciones que debe cumplir y el tono con el que debe responder, entre otros atributos.



Hi! I'll help you build a new GPT. You can say something like, "make a creative who helps generate visuals for new products" or "make a software engineer who helps format my code."

What would you like to make?

Figura 4.1: Interfaz para la construcción de modelos GPT

Construcción Inicial

Todos los procesos de construcción siempre comienzan con la misma estructura en la cual GPTBuilder te guía mediante una conversación para que proporciones los detalles necesarios para construir adecuadamente un modelo. La primera pregunta siempre es: *"What would you like to make?"*. Durante estas fases es preferible responder de forma clara y concisa, ya que será el propio GPTBuilder el que se encargue de reformular las instrucciones que le demos para que el modelo final cumpla con lo esperado.

A la pregunta de *"What would you like to make?"* se ha respondido lo siguiente:

"Make a pattern design expert, proficient in identifying and recommending the most suitable design patterns for various software use cases. When presented with a specific use case, provide detailed advice on the best design pattern to use, explaining the reasons behind your choice. Consider factors such as maintainability, scalability, and simplicity in your recommendations."

Durante estas fases iniciales de configuración es preferible ser breve, directo y conciso. Se debe indicar de forma muy precisa qué es lo que se quiere construir y cuál es su propósito. Tras esta petición, será GPTBuilder el encargado de proporcionar el contexto a nuestro modelo reinterpretando las instrucciones que se le han dado.

Gracias al contexto proporcionado, GPTBuilder iniciará nuestro modelo por primera vez, dándole además un nombre y una foto que lo identifique. En este caso se ha optado por el nombre ***Design Pattern Advisor*** y la foto que se ha utilizado en honor al libro *Design Patterns: Elements of Reusable Object-Oriented Software*, es la siguiente:

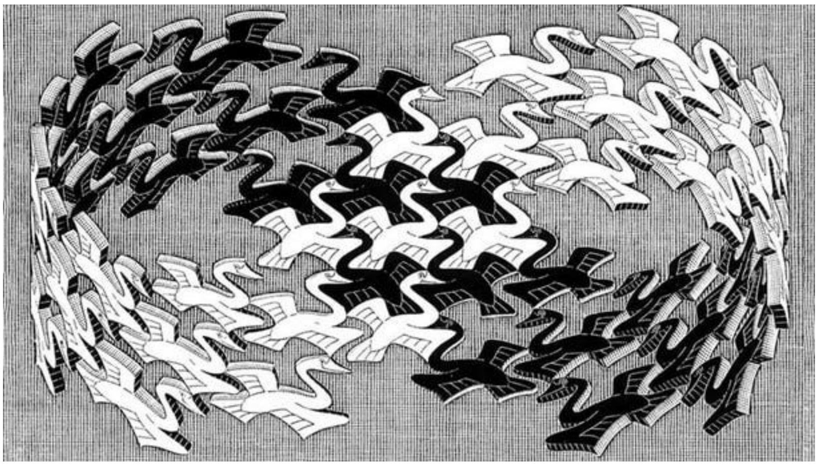


Figura 4.2: Foto identificativa para el modelo ***Design Pattern Advisor***[6]

Refinamiento

Una vez el modelo está creado, comienza el proceso de refinamiento, donde GPTBuilder continúa guiando al usuario sobre qué elementos se deberían enfatizar o evitar a través de la siguiente pregunta, a lo que se le ha respondido:

"When presented with a specific use case, provide detailed advice on the best design pattern to use, explaining the reasons behind your choice. Ensuring that the design pat-

tern directly addresses the core problem of the use case. Focus on providing high-level guidance and reasoning, and avoid getting into detailed implementations.“

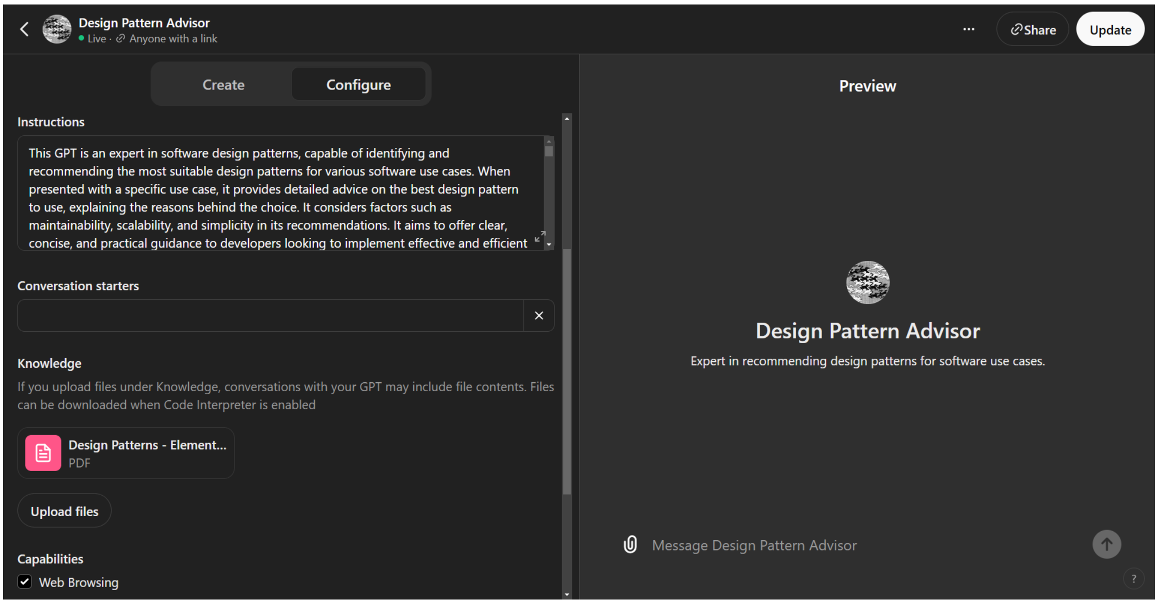
Como se está desarrollando este modelo especializado con el objetivo de utilizarlo como un consejero que sea capaz de proponer patrones que puedan solucionar los distintos casos de uso que se le presenten, se enfatiza que el modelo debe poner el foco en el razonamiento que realiza para proponer la solución y se le pide que evite cualquier detalle de implementación del patrón, ya que esto se abordará con otro modelo.

Finalmente, GPTBuilder solicita que se le indique con qué estilo de comunicación debe responder a las peticiones, es decir, si las respuestas deben contar con un tono formal, casual, de carácter técnico, etc. A lo que se la ha dicho lo siguiente:

“You should communicate in a formal tone, asking clarifying questions if necessary, as if the user were seeking the best detailed yet understandable information. Your role is to provide expert advice on design patterns, ensuring that your recommendations are clear, concise, and well-reasoned.“

Iterar

A partir de este punto, GPTBuilder recalca que el modelo está completamente listo a falta de algún refinamiento que se requiera. Sin embargo, a pesar de que GPTBuilder diga esto, el modelo aún debe terminar de ser refinado. Al igual que la validación del prompt se basaba en los resultados que se obtenían de este, con la construcción de los modelos GPT pasa exactamente lo mismo. El modelo no está construido hasta que las respuestas que se obtienen son las esperadas.



Instructions

This GPT is an expert in software design patterns, capable of identifying and recommending the most suitable design patterns for various software use cases. When presented with a specific use case, it provides detailed advice on the best design pattern to use, explaining the reasons behind the choice. It considers factors such as maintainability, scalability, and simplicity in its recommendations. It aims to offer clear, concise, and practical guidance to developers looking to implement effective and efficient

Conversation starters

Knowledge

If you upload files under Knowledge, conversations with your GPT may include file contents. Files can be downloaded when Code Interpreter is enabled

Design Patterns - Element...
PDF

Upload files

Capabilities

☑ Web Browsing

Design Pattern Advisor

Expert in recommending design patterns for software use cases.

Figura 4.3: *Design Pattern Advisor*

Para refinar y validar el modelo, se puede seguir haciendo uso de GPTBuilder, mediante el cual, a partir de las peticiones que se le hagan, se pueden ir perfeccionando las respuestas que se obtienen. Este proceso iterativo es muy similar al que se seguía al diseñar los prompts durante el capítulo anterior, sin embargo, gracias a GPTBuilder este proceso se puede realizar de forma mucho más ágil y rápida, ya que a medida que se van incorporando las nuevas instrucciones, se puede ir comprobando cómo afectan a las respuestas del modelo. Entre todas las peticiones de refinamiento que se han realizado, la más destacada, es sin duda la siguiente:

"Your response should meticulously address the following points to provide a well-rounded solution:

Recommended Patterns:

- *Identification and Description: Clearly identify and describe the design pattern that best aligns with the given use case. Detail the structure and fundamental principles of the pattern.*
- *Relevant Examples: Provide specific, real-world examples or case studies demonstrating successful implementation of these patterns. Highlight how these implementations addressed similar challenges.*

Justification:

- *Fit for Purpose: Elaborate on why the selected pattern is ideal for the use case, focusing on its scalability, maintainability, and flexibility. Discuss any specific features of the pattern that makes it particularly suitable.*
- *Strategic Impact: Explain how this pattern enhances the system's overall strategy and long-term goals.*

Pros and Cons:

- *Advantages: Outline the key benefits of using the selected design pattern, emphasizing how it contributes to the system's efficiency and robustness.*
- *Drawbacks: Discuss any potential limitations or challenges associated with this pattern, including scenarios where they might not be the best fit.*

Impact on Other Architectural Aspects:

- *System-wide Effects: Analyze how the recommended pattern will interact with other architectural elements of the system. Consider its impact on both the technical architecture and business processes.*
- *Potential Conflicts and Synergies: Identify any possible conflicts or synergies with existing architectural frameworks or patterns. Propose solutions or adjustments to mitigate conflicts and enhance synergies.*

Alternatives:

- *Comparison: Introduce alternative design patterns that could also be applicable. Provide a comparative analysis showing the differences in outcomes, complexity, and other relevant factors.*

- *Selection Rationale: Justify why the primary choice is preferable over these alternatives based on the specific requirements of the use case.*

Your response should synthesize these considerations into a comprehensive solution that not only addresses the immediate needs of the use case but also anticipates future requirements and challenges, thereby ensuring scalability and maintainability within the broader architectural context."

Esta petición es la misma que se utilizó en el capítulo anterior y gracias a la cual se consigue dar formato a las respuestas que el modelo ofrecerá.

Para finalizar la construcción de este GPT, se le adjunta en la base de conocimientos un PDF con el contenido de *Design Patterns: Elements of Reusable Object-Oriented Software*. Esta base de conocimientos consta de archivos en los que el modelo se apoyará a la hora de dar las respuestas, por lo que proveerle de este libro hará que las soluciones que proponga sean mucho más significativas.

4.2 Proceso de construcción - Implementación de patrones

Construcción Inicial

Siguiendo el mismo proceso de construcción utilizado para *Design Pattern Advisor* se definirá el modelo *Design Pattern Architect*. Este nuevo modelo se encargará de implementar mediante código la estructura que define la solución propuesta por el modelo inicial. En este caso, a la pregunta "*What would you like to make?*" se ha respondido lo siguiente:

"*Make a GPT that assists in implementing the core structure of software design patterns. Given a use case and a specified design pattern, it should create the necessary interfaces, define the core structure (including main classes, methods, and relationships), and provide clear documentation explaining the purpose and usage of each component. Inputs will include a description of the scenario (use case) and the specific design pattern to be implemented. Outputs should consist of defined interfaces for the pattern, the core structure with main classes and methods, and documentation detailing the purpose and usage of each component.*"

Con este prompt se quiere definir cuál debe ser el comportamiento fundamental de *Design Pattern Architect*. La conducta del modelo debe asentarse en la implementación de patrones de diseño, creando toda la estructura necesaria para la correcta integración de la solución. Este proceso pasa por la definición de interfaces, clases y métodos que repliquen este comportamiento.

Además, se enfatiza la necesidad de documentar el propósito y el uso de cada componente que forma esta estructura, elemento muy necesario para que el ingeniero de software pueda visualizar correctamente si la solución propuesta es la adecuada.

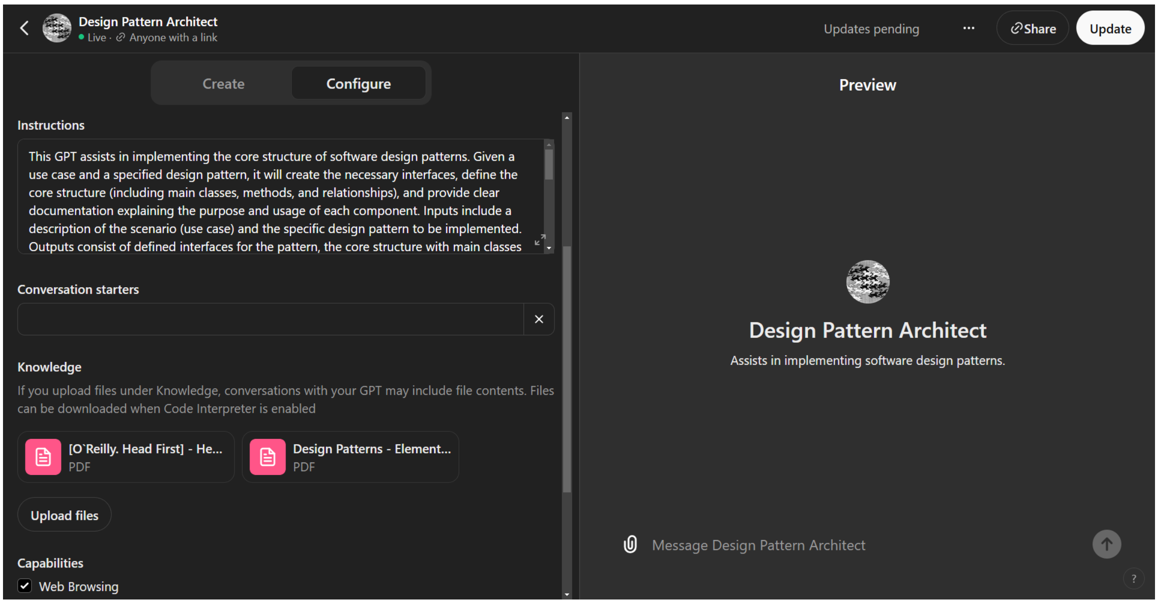
Refinamiento

Tras estas especificaciones y una vez el modelo se ha iniciado por primera vez, se le enfatiza a GPTBuilder cuáles son las áreas en las que el modelo se debe enfocar y cuáles debe evitar.

> *“To start, the GPT should emphasize clarity, accuracy, and simplicity in its responses. It should provide clear and concise definitions and avoid unnecessary complexity or jargon. The focus should be on practical, implementable solutions that align with the given use case and design pattern. Avoid overly verbose explanations and ensure that the core elements are addressed directly and efficiently. The responses should be structured in a way that makes it easy for users to understand and apply the provided information without extraneous details or deviations from the main objective.“*

Para las tareas de implementación se requiere de claridad, precisión y simplicidad, que es lo que fundamentalmente se le está pidiendo a este modelo. Se busca que la respuesta sea directa y concisa, abordando la solución de la forma más práctica posible. Las explicaciones deben seguir la misma estructura, pues lo que se busca aquí no es la comprensión detallada del patrón. Se asume este conocimiento y se busca un entendimiento más ágil y riguroso.

Finalmente, el tono y la forma que debe seguir este modelo será idéntica a la del modelo anterior, ya que se busca una continuidad en el estilo de comunicación de ambos modelos.



This GPT assists in implementing the core structure of software design patterns. Given a use case and a specified design pattern, it will create the necessary interfaces, define the core structure (including main classes, methods, and relationships), and provide clear documentation explaining the purpose and usage of each component. Inputs include a description of the scenario (use case) and the specific design pattern to be implemented. Outputs consist of defined interfaces for the pattern, the core structure with main classes

If you upload files under Knowledge, conversations with your GPT may include file contents. Files can be downloaded when Code Interpreter is enabled

☑ Web Browsing

Figura 4.4: *Design Pattern Architect*

Iterar

Tras establecer las bases del modelo, comienza el proceso de validación del modelo gracias a las posibilidades que ofrece GPTBuilder. Para mejorar la implementación que realiza este modelo sobre los distintos patrones de diseño, no se ha optado por mejorar la configuración del modelo mediante más prompts, debido a que la forma de las respuestas era desde un principio lo que se buscaba. Sin embargo, se han añadido ejemplos de código a la base de conocimiento del modelo, en los cuales se programa la estructura de los diferentes patrones de diseño.

Los ejemplos más relevantes que se han añadido son los presentes en el libro *Head First Design Patterns*, ya que demuestran cómo se pueden implementar diversos patrones mediante código Java. Lo que beneficia e influencia directamente las implementaciones que pueda hacer el modelo.[30]

Figura 4.5: ***Head First Design Patterns (A Brain Friendly Guide)***[30]

El proceso de construcción de este modelo se ha validado mediante la experimentación y los resultados obtenidos. Durante todo el trabajo se ha comentado que este era el método más efectivo para conseguir tener las soluciones que se esperan de cada proceso, y así se ha ido aplicando durante cada construcción y diseño tanto de prompts como de modelos GPT.

4.3 Patrones Creacionales

Durante esta sección se reanalizará el comportamiento del modelo de lenguaje desarrollado por OpenAI al enfrentarse a un problema que requiere de un patrón creacional:

"Let's consider the use case of a software system for report management in a company. This system needs to be able to generate different types of reports (such as financial reports, employee reports, and inventory reports), and these reports can vary in complexity and format."

Sin embargo, esta vez se utilizarán los modelos GPT *Design Pattern Advisor* y *Design Pattern Architect* para comprobar si las respuestas provistas son de mayor calidad que las ofrecidas por GPT-4.

En primer lugar, se utilizará el modelo *Design Pattern Advisor* para analizar el caso de uso propuesto. A partir de la respuesta que se obtenga del modelo especializado, se considerará su implementación usando el modelo *Design Pattern Architect*. Además, se compararán los resultados con los que se obtuvieron durante el capítulo anterior, pretendiendo analizar la mejoría de las soluciones.

Selección

La solución que ha ofrecido el modelo *Design Pattern Advisor* es la siguiente:

*"**Recommended Pattern: Factory Method***

Identification and Description

The Factory Method pattern is a creational pattern that defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. This pattern involves a superclass that contains the logic to invoke the factory method, which is implemented by subclasses to create specific types of objects.

Structure

- *Creator Class: Declares the factory method, which returns an object of type Product. The Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.*
- *ConcreteCreator Classes: Override the factory method to return a different ConcreteProduct instance.*
- *Product Interface: Defines the interface of objects the factory method creates.*
- *ConcreteProduct Classes: Implement the Product interface.*

Justification

Fit for Purpose

- *Scalability: The Factory Method pattern supports the creation of a variety of reports (financial, employee, inventory) by defining a common interface for creating these reports. New types of reports can be added without modifying existing code, enhancing scalability.*
- *Maintainability: The separation of report creation logic from the report processing logic makes the system easier to maintain. Each ConcreteCreator is responsible for the instantiation of specific report types, making it easier to manage changes in report formats or structures.*
- *Flexibility: This pattern allows the creation of reports with different formats and complexities by simply extending the ConcreteCreator class. This flexibility is essential for accommodating varying reporting requirements.*

Strategic Impact

- *Long-term Goals: By utilizing the Factory Method pattern, the system can easily adapt to future requirements such as the addition of new report types or changes in report specifications. This aligns with the long-term goal of maintaining an adaptable and extensible system architecture.*

Pros and Cons

Advantages

- *Encapsulation of Object Creation: It encapsulates the instantiation of report objects, promoting loose coupling between client code and concrete classes.*
- *Ease of Extension: New report types can be introduced by adding new ConcreteCreator classes without altering existing code, enhancing the extensibility of the system.*
- *Single Responsibility Principle: Each creator class has a single responsibility, i.e., to instantiate a particular type of report, making the system more modular and easier to manage.*

Drawbacks

- *Class Proliferation: The Factory Method pattern may lead to an increase in the number of classes, as each new report type requires a new ConcreteCreator class.*
- *Complexity: For simple systems, the Factory Method pattern can add unnecessary complexity. However, for the given use case with varying report types and formats, this complexity is justified.*

Impact on Other Architectural Aspects

System-wide Effects

- *Interoperability: The pattern ensures that new report types can be integrated seamlessly without affecting other parts of the system, promoting a modular architecture.*
- *Consistency: By standardizing the process of report creation, it ensures consistency across different types of reports generated by the system.*

Potential Conflicts and Synergies

- *Synergies: The Factory Method pattern can synergize well with the Abstract Factory pattern if there is a need to create families of related objects (e.g., different types of reports with different formatting components).*
- *Conflicts: There could be potential conflicts if the system requires a high degree of customization for each report type, as this might necessitate a more flexible approach than the Factory Method can offer.*

Alternatives

Comparison

- *Abstract Factory: This pattern could be considered if there is a need to create families of related or dependent objects. However, it might introduce additional complexity if the sole requirement is to instantiate different types of reports.*

- *Builder: This pattern is useful for constructing complex objects step by step. It might be beneficial if the report creation process involves multiple steps or if the reports have a very complex structure.*

Selection Rationale

The Factory Method pattern is preferable for the given use case because it provides a straightforward and scalable solution for creating different types of reports while maintaining low coupling and high cohesion within the system. It strikes a balance between flexibility and simplicity, making it an ideal choice for a report management system that needs to accommodate diverse reporting requirements.

By implementing the Factory Method pattern, the report management system will be well-positioned to handle current and future reporting needs efficiently and effectively.
"

De nuevo, el modelo propone el patrón Método Fábrica para resolver el problema, sin embargo, esta vez, la respuesta es de mucha más calidad.

En primer lugar, identifica todos los elementos que construyen la estructura de la solución, indicando qué papel toma cada elemento. Esto es muy interesante, ya que complementándolo con la implementación del patrón que se verá a continuación, se contemplan todos los elementos para que la integración del patrón sea lo más sencilla posible.

Además de esto, las descripciones, explicaciones e indicaciones son mucho más específicas y precisas, esto se debe sobre todo a la base de conocimiento que se ha construido en el modelo, gracias a la cual *Design Pattern Advisor* es capaz de dar respuestas con mucho más rigor.

Al comparar ambas respuestas, sobresalen las explicaciones que proporciona este nuevo modelo, especialmente en términos de justificaciones, ventajas y desventajas. Las respuestas de GPT-4 eran a menudo demasiado genéricas y podían aplicarse a cualquier patrón de diseño. En cambio, *Design Pattern Advisor* ofrece soluciones con mucho más contexto y especificidad, lo que resulta en una respuesta más precisa y útil.

Implementación

Para la implementación de esta solución se usará el modelo *Design Pattern Architect*, modelo que será capaz de programar la estructura y el núcleo del método fábrica para integrarlo en un sistema de generación de reportes. El código que ha creado el modelo es el siguiente:

```java
// IReport.java
public interface IReport {
    void generate();
}
```

```java
// FinancialReport.java
public class FinancialReport implements IReport {
    @Override
    public void generate() {
```

```java
        // Logic to generate financial report
        System.out.println("Generating Financial Report");
    }
}

// EmployeeReport.java
public class EmployeeReport implements IReport {
    @Override
    public void generate() {
        // Logic to generate employee report
        System.out.println("Generating Employee Report");
    }
}

// InventoryReport.java
public class InventoryReport implements IReport {
    @Override
    public void generate() {
        // Logic to generate inventory report
        System.out.println("Generating Inventory Report");
    }
}
```

```java
// ReportFactory.java
public abstract class ReportFactory {
    public abstract IReport createReport();

    public void generateReport() {
        IReport report = createReport();
        report.generate();
    }
}
```

```java
// FinancialReportFactory.java
public class FinancialReportFactory extends ReportFactory {
    @Override
    public IReport createReport() {
        return new FinancialReport();
    }
}

// EmployeeReportFactory.java
public class EmployeeReportFactory extends ReportFactory {
    @Override
    public IReport createReport() {
        return new EmployeeReport();
    }
}

// InventoryReportFactory.java
public class InventoryReportFactory extends ReportFactory {
    @Override
    public IReport createReport() {
        return new InventoryReport();
    }
}
```

```java
// Main.java
```

```
public class Main {
    public static void main(String[] args) {
        ReportFactory financialFactory = new FinancialReportFactory();
        financialFactory.generateReport();

        ReportFactory employeeFactory = new EmployeeReportFactory();
        employeeFactory.generateReport();

        ReportFactory inventoryFactory = new InventoryReportFactory();
        inventoryFactory.generateReport();
    }
}
```

La implementación que ha realizado *Design Pattern Architect* es correcta. Comparando el código con el diseñado por el modelo GPT-4, se puede observar que el defecto que se detectaba en estas respuestas se ha corregido.

Esta nueva solución deja de usar las fábricas únicamente para delegar la responsabilidad de la creación del reporte. En esta nueva implementación, el modelo implementa funcionalidad en la clase *ReportFactory* que hace uso del método *createReport()* para así realmente utilizar el reporte que generen las subclases.

Esta implementación del patrón es prácticamente perfecta, es cierto que falta documentación en el código, porque los comentarios únicamente definen la estructura de clases. Sin embargo, el modelo ha ofrecido parte de la documentación como explicación fuera del código. Este es un comportamiento que se ha observado a lo largo de todo el trabajo tanto por GPT-4 como por *Design Pattern Architect*. Para desplegar código en producción puede ser la opción más recomendada, pero en estas circunstancias es más interesante que hubiese más documentación en el código.

4.4 Patrones Estructurales

Siguiendo con el análisis de estos dos nuevos modelos y utilizando el mismo procedimiento que en la sección anterior, se examinarán las soluciones que proponen para el siguiente caso de uso, en el cual es necesario implementar un patrón de diseño estructural.

"In this use case, we are developing a sophisticated graphics editor application. This application will include a variety of basic shapes such as squares, circles, and more. Additionally, it will offer the flexibility to extend these shapes with new functionalities, including options for filling them with colors or adding decorative borders."

Selección

La solución propuesta por el modelo *Design Pattern Advisor* es la siguiente:

*"**Recommended Pattern: Decorator***

Identification and Description:

The Decorator design pattern is suitable for the graphics editor application where shapes can be extended with additional functionalities like filling with colors or adding borders. The Decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

Structure and Fundamental Principles:

- *The Decorator pattern involves a set of decorator classes that are used to wrap concrete components.*
- *Decorators are typically designed to be interchangeable, allowing for flexible combinations of behaviors.*
- *It adheres to the principle of class inheritance and composition, where new functionality is added by composing objects rather than inheriting from a base class.*

Relevant Examples:

- *Java I/O Streams: In Java, the I/O Streams library uses the Decorator pattern extensively to add functionality like buffering, data conversion, and more.*
- *GUI Frameworks: In graphical user interface (GUI) frameworks, visual components often use the Decorator pattern to add functionalities such as borders, scrolling, and shadowing.*

Justification:

Fit for Purpose:

- *Scalability: The Decorator pattern allows you to create new functionality combinations at runtime, which is essential for a graphics editor where users might frequently change how shapes are rendered.*
- *Maintainability: Adding new functionalities, such as different border styles or fill patterns, can be done without modifying existing code. This reduces the risk of introducing bugs into the system.*
- *Flexibility: By using decorators, you can extend the functionality of shapes in a flexible and reusable manner. For instance, you can add multiple decorators to a shape to combine effects.*

Strategic Impact:

- *Extensibility: The Decorator pattern enhances the system's ability to grow with new requirements without requiring significant refactoring.*
- *Reusability: Since the core shape classes remain unchanged, they can be reused across different parts of the application, and new decorators can be developed independently.*

Pros and Cons:

Advantages:

- *Open/Closed Principle: Classes are open for extension but closed for modification.*
- *Dynamic Behavior: Decorators can be added or removed at runtime, allowing dynamic changes to object behavior.*
- *Simplification: Reduces the need for complex inheritance hierarchies.*

Drawbacks:

- *Complexity: Can lead to a large number of small classes that can be overwhelming.*
- *Performance: Adding multiple layers of decorators can introduce performance overhead due to the increased number of objects and the indirection involved.*

Impact on Other Architectural Aspects:

System-wide Effects:

- *Consistency: Ensures that extensions to functionality are applied consistently across all shapes.*
- *Integration: Decorators must be carefully integrated with the shape rendering system to avoid unexpected interactions or performance issues.*

Potential Conflicts and Synergies:

- *Synergies: Works well with patterns like Composite (to treat individual objects and compositions uniformly) and Strategy (to encapsulate varying behaviors).*
- *Conflicts: Overuse can lead to complex decorator chains that are difficult to manage and debug.*

Alternatives:

Composite Pattern:

- *Comparison: Composite would allow shapes to be grouped into tree structures, enabling part-whole hierarchies.*
- *Selection Rationale: While Composite is useful for grouping shapes, it doesn't provide the flexibility to dynamically add behaviors like Decorator.*

Strategy Pattern:

- *Comparison: Strategy encapsulates an algorithm inside a class, allowing different algorithms to be swapped in and out.*
- *Selection Rationale: Strategy is more suited for behaviors that can be changed independently of the objects they operate on, whereas Decorator is ideal for adding responsibilities to objects.*

In summary, the Decorator pattern is the most suitable choice for extending shapes with additional functionalities in a sophisticated graphics editor due to its scalability, maintainability, and flexibility. This pattern supports dynamic behavior changes and adheres to key object-oriented principles, ensuring a robust and extensible system.“

Siguiendo la línea de lo que se había vislumbrado en la experimentación anterior, aquí se sigue pudiendo observar que los razonamientos y las explicaciones que expone este modelo son de un carácter superior.

Al contrastar ambas soluciones, se advierte que el contenido es en realidad muy similar. Es decir, lo que ambos modelos intentan transmitir es esencialmente lo mismo. Ahora bien, la forma y la precisión con la que logra hacerlo *Design Pattern Advisor* es significativamente mejor. Provocando que un ingeniero con experiencia comprenda en menos tiempo y con más profundidad qué es lo que el modelo le está transmitiendo.

Otro aspecto interesante es que esta vez *Design Pattern Advisor* no ha optado por definir cada elemento que compone la estructura del patrón, sino que en la mayoría de conversaciones ha descrito los aspectos fundamentales en los que se basa el patrón de diseño. Ambas estrategias le añaden un valor relevante a la solución que propone el modelo.

Implementación

Debido a que la respuesta es la adecuada, se sigue el mismo proceso que en las secciones anteriores, procediendo ahora con la implementación del patrón de diseño:

```java
// The base interface for shapes
public interface Shape {
    void draw();
}
```

```java
// Concrete Component: Circle
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

// Concrete Component: Square
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}
```

```java
// The base decorator class implementing the Shape interface
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

```

```java
    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    @Override
    public void draw() {
        decoratedShape.draw();
    }
}
```

```java
// Concrete Decorator: ColorfulShape
public class ColorfulShape extends ShapeDecorator {
    private String color;

    public ColorfulShape(Shape decoratedShape, String color) {
        super(decoratedShape);
        this.color = color;
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setColor(decoratedShape, color);
    }

    private void setColor(Shape decoratedShape, String color) {
        System.out.println("Filling the shape with color: " + color);
    }
}

// Concrete Decorator: BorderedShape
public class BorderedShape extends ShapeDecorator {
    private String borderStyle;

    public BorderedShape(Shape decoratedShape, String borderStyle) {
        super(decoratedShape);
        this.borderStyle = borderStyle;
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setBorder(decoratedShape, borderStyle);
    }

    private void setBorder(Shape decoratedShape, String borderStyle) {
        System.out.println("Adding border style: " + borderStyle);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape redCircle = new ColorfulShape(circle, "Red");
        Shape redCircleWithBorder = new BorderedShape(redCircle, "Dashed");

        System.out.println("Drawing plain Circle:");
        circle.draw();
```

```

        System.out.println("\nDrawing Red Circle:");
        redCircle.draw();

        System.out.println("\nDrawing Red Circle with Dashed Border:");
        redCircleWithBorder.draw();
    }
}
```

Esta implementación del patrón es muy similar a la que había realizado GPT-4 en la sección anterior. El código ya era adecuado anteriormente y *Design Pattern Architect* lo ha replicado de forma prácticamente idéntica.

Esto no debería sorprender, pues en realidad, ambos modelos son, en esencia, lo mismo. Tras varias ejecuciones de la misma situación, las respuestas han sido muy consistentes, siendo esto uno de los aspectos más a destacar de Design Pattern Advisor y Design Pattern Architect.

La consistencia está presente en este nuevo proceso y esto es desde luego un aspecto muy positivo, pues por lo que se está observando, los resultados en su mayoría son correctos y rigurosos. GPT-4 y Gemini, en cambio, presentaban bastante más variabilidad, y a pesar de que esto le podía otorgar al ingeniero distintas posibilidades, en consecuencia también podía provocar más dudas, pues no había ninguna solución clara.

4.5 Patrones de comportamiento

Finalmente, se explorará cuál es el comportamiento de ambos modelos al enfrentarse a un caso de uso que requiere de un patrón de comportamiento como solución. El problema a resolver es el siguiente:

“Implement a functionality that allows users to undo recent changes in a text editor. This feature should enable users to reverse their latest modifications, whether they are additions, deletions, or alterations of text. By tracking the history of changes, the undo function should provide the capability to step back through these changes one by one, restoring the text to previous states. This will enhance the user's ability to correct mistakes and refine their text without losing any recent work. The implementation should be efficient and seamless, ensuring a smooth user experience“

Selección

Design Pattern Advisor ha formulado la siguiente salida:

*“**Recommended Pattern: Memento***

Identification and Description:

The Memento design pattern is ideal for implementing the undo functionality in a text editor. This pattern captures and externalizes an object's internal state, so that the object can be restored to this state later without violating encapsulation.

The structure of the Memento pattern includes three main participants:

- *Originator: The object whose state needs to be saved. In this case, it is the text editor.*
- *Memento: The object that stores the internal state of the Originator.*
- *Caretaker: The object that keeps track of the mementos, managing the undo stack.*

Relevant Examples:

- *Text Editors: Many text editors like Sublime Text and Microsoft Word implement undo functionality using a variation of the Memento pattern to save and restore text states.*
- *Graphic Editing Software: Programs like Adobe Photoshop use this pattern to allow users to revert changes made to graphics.*

Justification:

Fit for Purpose:

- *Scalability: The Memento pattern scales well with the number of changes, as it allows storing each state change as a separate memento. This ensures efficient state management and retrieval.*
- *Maintainability: By encapsulating the state within mementos, the pattern maintains the principle of encapsulation. The text editor's internal state remains hidden from the Caretaker, which only handles mementos.*
- *Flexibility: The pattern allows multiple undo and redo operations, making it easy to implement features like multi-level undo.*

Strategic Impact:

- *Enhanced User Experience: Providing a robust undo feature enhances the user experience by allowing users to easily correct mistakes without losing their work.*
- *Long-term Usability: As the application evolves, the Memento pattern can be extended to accommodate additional features like selective undo, where users can undo specific changes out of sequence.*

Pros and Cons:

Advantages:

- *Encapsulation: Keeps the text editor's state encapsulated within the memento.*
- *Simplicity: Offers a straightforward mechanism for saving and restoring states.*
- *Separation of Concerns: Clearly separates the responsibility of managing states (Caretaker) from the functionality of the text editor (Originator).*

Drawbacks:

- *Memory Overhead: Storing multiple states can consume significant memory, especially for large documents.*
- *Complexity in Implementation: Ensuring that all relevant aspects of the editor's state are captured and restored correctly can be complex.*

Impact on Other Architectural Aspects:

System-wide Effects:

- *Performance: The performance might be affected due to the memory overhead of storing multiple mementos. Optimization techniques, such as only storing deltas (differences between states), can mitigate this.*
- *Consistency: Ensuring consistency between the memento states and the current state of the text editor is crucial for reliable undo functionality.*

Potential Conflicts and Synergies:

- *Conflicts: The Memento pattern could conflict with other patterns that manipulate the state directly if not carefully managed.*
- *Synergies: Works well with the Command pattern, where each command can create a memento before execution, allowing undo operations to be managed efficiently.*

Alternatives:

Comparison:

- *Command Pattern: The Command pattern can also be used for implementing undo functionality by encapsulating all changes as commands. Each command knows how to undo its action.*
- *State Pattern: This pattern could be used to represent different states of the text editor, allowing transitions between states, but it may not be as efficient for undo functionality compared to the Memento pattern.*

Selection Rationale:

Why Memento is Preferred: The Memento pattern is more suitable for this specific use case as it directly addresses the need to capture and restore the state of the text editor efficiently and encapsulates state information better than the Command pattern for this scenario.

In conclusion, the Memento design pattern offers a robust solution for implementing the undo functionality in a text editor, ensuring encapsulation, scalability, and maintainability while enhancing the user experience and supporting long-term usability.

Durante toda esta segunda fase de experimentación, se asumía que lo más razonable era que *Design Pattern Advisor* seleccionase los mismos patrones que había escogido GPT-4 al tratarse del mismo modelo. Sin embargo, para este caso de uso ha sido propuesto como solución el patrón Memento, en contraposición al patrón Comando, que fue el escogido por GPT-4.

Tanto el patrón Comando como el Memento son soluciones válidas para esta problemática, y la utilización de un patrón u otro dependerá de los requisitos no funcionales de cada sistema. El cambio de elección que realiza el modelo probablemente esté influenciado por la base de conocimiento que se le ha introducido al modelo. Además, el patrón Memento está diseñado específicamente para almacenar el estado de un sistema, por lo que es la elección más lógica e intuitiva.

Es cierto que las respuestas que formulaba GPT-4 contaban con mucha variabilidad, no obstante, el patrón Comando era la solución que más veces se manifestaba. En cambio, un aspecto que se considera de gran relevancia de esta segunda de experimentación es que *Design Pattern Advisor* ha sido mucho más consistente que GPT-4.

Para todos los casos de uso, siempre ha optado por la misma solución, explicando una y otra vez la estrategia que se debía seguir, a pesar del número de veces que se ejecutase la conversación.

Esta mejora de la consistencia y de la forma en las soluciones que propone *Design Pattern Advisor*, no debe provocar una sobreestimación de las capacidades y limitaciones con las que sigue contando este tipo de modelos. Se definen estos procesos para que los resultados mejoren y sean lo más minuciosos posible, pero es fundamental tener en cuenta cuál sigue siendo la fiabilidad de las respuestas, requiriendo siempre la revisión de estas.

Implementación

La implementación que ha hecho *Design Pattern Arhitect* se comparará con la que hizo Gemini, ya que fue el modelo desarrollado por Google, el que construyó la estructura del patrón Memento.

```java
public class TextMemento {
    private final String state;

    public TextMemento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}
```

```java
public class TextEditor {
    private String text;

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public TextMemento save() {
```

```java
        return new TextMemento(text);
    }

    public void restore(TextMemento memento) {
        this.text = memento.getState();
    }
}
```

```java
import java.util.Stack;

public class TextCaretaker {
    private Stack<TextMemento> mementoStack = new Stack<>();

    public void save(TextMemento memento) {
        mementoStack.push(memento);
    }

    public TextMemento undo() {
        if (!mementoStack.isEmpty()) {
            return mementoStack.pop();
        }
        return null;
    }
}
```

```java
public class TextEditorApp {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        TextCaretaker caretaker = new TextCaretaker();

        // Initial state
        editor.setText("Hello");
        caretaker.save(editor.save());

        // Modify text
        editor.setText("Hello, World!");
        caretaker.save(editor.save());

        // Another modification
        editor.setText("Hello, Design Patterns!");
        caretaker.save(editor.save());

        System.out.println("Current Text: " + editor.getText());

        // Undo last change
        editor.restore(caretaker.undo());
        System.out.println("After Undo: " + editor.getText());

        // Undo another change
        editor.restore(caretaker.undo());
        System.out.println("After Second Undo: " + editor.getText());
    }
}
```

Al comparar el código con el de Gemini se puede observar que esta nueva implementación es totalmente diferente. Gemini concentraba todo el patrón en una única clase. No obstante, *Design Pattern Architect* estructura el código de una forma totalmente distinta.

Separando completamente las responsabilidades de cada clase, se aplica correctamente el principio de responsabilidad única, haciendo que el código y el patrón sean a su vez más comprensibles. Además, la implementación es correcta, permitiendo la restauración de estados como definía el caso de uso.

4.6 Consideraciones

Durante todo este capítulo se ha podido observar cuál es la potencia y las posibilidades de los modelos GPT especializados. Las respuestas que se han conseguido han cumplido con las expectativas, mejorando considerablemente la soluciones que se obtuvieron durante el capítulo *3. Diseño de Software con Large Language Models*.

Design Pattern Advisor ha sido capaz de aumentar la calidad con respecto a la selección de patrones de diseño. Las explicaciones en cuanto a la justificación, el razonamiento y las propiedades de las soluciones son más específicas, precisas y descriptivas. Todo esto ha hecho que las respuestas que ofrecía el modelo fuesen más significativas, mejorando así la asistencia que pueda proveer este modelo a los ingenieros de software.

Además, el proceso de implementación de patrones también ha mejorado sustancialmente. *Design Pattern Architect* ha sido capaz de corregir la mayoría de errores que se encontraron durante el capítulo anterior, proponiendo unas estructuras de código más similares a las que se buscaban en un principio. De esta forma, el proceso de integración de estas soluciones en un sistema software sería más liviano y ágil, al contar con el diseño de las clases que se pretenden implementar.

Gracias a estos dos nuevos modelos se ha conseguido un incremento en la calidad de las respuestas que ofrecía GPT-4. Asimismo, uno de los aspectos más importantes es que no solo ha aumentado la calidad, sino que los resultados eran altamente consistentes, lo que provoca una confianza mayor en las respuestas obtenidas.

A pesar de todos estos resultados tan favorables, es necesario seguir insistiendo en la importancia que tiene el papel del ingeniero de software en este ámbito. En primer lugar, estos modelos no se podrían refinar ni validar sin un ingeniero que fuese capaz de detectar si los resultados que se obtienen son los esperados. Pero aún más importante, es que por la naturaleza probabilística de los *Large Language Models*, jamás se va a poder garantizar la corrección de las respuestas.

Aquí es donde juegan un papel fundamental los ingenieros y desarrolladores de software, que deben ser capaces de validar y verificar las soluciones que les puedan proponer los modelos de lenguaje. Esta revisión y análisis de las respuestas que proporcionan los modelos son obligatorias si se quieren garantizar unos resultados adecuados. Como ya se ha mencionado durante todo este trabajo, se ha de ser consciente de que estos modelos son asistentes y deben ser usados de esta misma manera.

CAPÍTULO 5
Conclusiones

Durante el transcurso de este trabajo, se han explorado distintas metodologías para poder integrar y aplicar la Inteligencia Artificial Generativa en el Diseño de Software. Esto se ha llevado a cabo mediante la utilización de *Large Language Models*, con los cuales se pretendía mejorar y facilitar el trabajo de un ingeniero de software al enfrentarse a problemas de diseño.

Los resultados que se han obtenido durante las diferentes fases de experimentación respaldan la eficacia de los procesos que se han definido. Estos resultados sostienen que los modelos de lenguaje, en particular GPT-4 y Gemini, cuentan con el potencial para los procesos y las soluciones en el ámbito del diseño de software.

En este capítulo se explorará cuáles han sido los objetivos de este trabajo y si se han llegado a cumplir. Además, se pretende mencionar todo lo que se ha aprendido y conseguido a lo largo de este trabajo, relacionándolo con los contenidos del grado de Ingeniería Informática. Finalmente, se propondrán futuras áreas de exploración relacionadas con el campo del diseño de software y la inteligencia artificial.

5.1 Revisión de los Objetivos

El objetivo principal del trabajo era conseguir integrar y aplicar Inteligencia Artificial Generativa en el ámbito del Diseño de Software. Considero que este objetivo se ha cumplido con creces, ya que hemos conseguido aplicar estas tecnologías a problemas de diseño del mundo real, proponiendo nuevos procesos y metodologías para hacerlo correctamente.

Es cierto que este objetivo es algo general, la integración de estos dos ámbitos de estudio se puede llevar a cabo de muchas maneras diferentes. Sin embargo, quería hacerlo mediante el uso de *Large Language Models* por el contexto global de la situación actual, donde son utilizados a diario por millones de usuarios.

Con respecto a los objetivos restantes, en primer lugar, se pretendía elaborar una estrategia en la creación y construcción de prompts. Este objetivo se ha conseguido de forma efectiva mediante la definición del proceso en el cual contábamos con un diseño inicial, que posteriormente se iba refinando de forma iterativa para conseguir satisfacer las necesidades que teníamos.

Otro de los objetivos que se pretendía era la definición de un procedimiento para la refactorización de código utilizando los LLMs. Finalmente, este objetivo se ha consumado de forma satisfactoria, gracias en parte al objetivo que se mencionaba anteriormente. La metodología definida para la construcción de prompts nos ha llevado a conseguir que la refactorización de código fuera una tarea más sencilla, complementándola con la validación y evaluación de la calidad de los resultados obtenidos mediante *SonarQube*.

Uno de los objetivos que considero que no se ha llegado a completar, es el referente a la exploración de las capacidades y limitaciones de los modelos de lenguaje al resolver casos de uso por medio de la utilización de patrones de diseño. Si bien es cierto que se han intentado diversificar los análisis lo máximo posible para explorar tanto estos límites como capacidades, creo que aún queda mucho por explorar en este ámbito, ya que los LLMs ofrecen posibilidades casi infinitas. Asimismo, hay funcionalidades con respecto a estos modelos que tampoco se han explorado debido a que el trabajo se debía acotar y adaptar a unos límites para poder ser abordable, como por ejemplo la generación de imágenes.

El objetivo más destacado de este trabajo era intentar mejorar los resultados que obteníamos mediante los LLMs, construyendo y afinando estos mismos modelos para, así, poder obtener unos resultados a medida. Este objetivo se ha cumplido ampliamente, y se puede comprobar con los resultados que han proporcionado estos modelos especializados al enfrentarse a los mismos problemas que GPT-4 y Gemini.

Finalmente, el objetivo que se ha ido recalcando a lo largo del trabajo y que era además una de las motivaciones para realizarlo era reforzar y respaldar la figura del ingeniero de software en la sociedad actual. Con el auge de la Inteligencia Artificial se ha buscado desde varios sectores mermar la labor de los desarrolladores e ingenieros de software, y creo que tras este trabajo esta figura sale reforzada. Los modelos de lenguaje son un avance muy significativo en nuestra sociedad, pero los resultados que ofrecen siguen siendo imperfectos, llenos de matices y detalles que hay que revisar y mejorar.

Estas herramientas pueden ser de gran utilidad para ingenieros que ya conocen cuáles son las limitaciones y capacidades de estos modelos, y que, además, sepan distinguir unos resultados adecuados de unos que no lo son. Sin embargo, para otras figuras con poca experiencia pueden resultar en una pérdida de tiempo y conocimiento.

5.2 Lecciones Aprendidas

A lo largo de este trabajo se han ido aprendiendo muchísimas lecciones, de las que destaco las siguientes. La primera lección es el orden, todos los procesos, investigaciones y experimentos deben llevar un orden de pasos bien definidos. Al principio del trabajo no sabía cómo iba a abordar todas las ideas que había considerado. Para llevar todo esto a cabo, mi principal aliado fue el orden. El hecho de definir metodologías, procesos, y pasos, hace que todo el proceso de experimentación sea replicable, por mi y por otros usuarios, permitiendo añadir rigor y metodología a la investigación.

Esta lección la relaciono con todas las metodologías que se describen en el grado. Estos procesos buscan poner un orden a todos los pasos que se deben seguir para lograr un objetivo o resultado. Sean metodologías ágiles o tradicionales, la finalidad es la misma. Definir unos pasos y un orden a seguir para obtener los mejores resultados posibles.

Otra lección que he aprendido es la validación y consistencia de los resultados que se obtienen durante todo el proceso de experimentación. Todos los resultados que se obtengan se han de validar y cotejar. Además, la calidad de estos se ha de medir o comparar de alguna manera. Los LLMs por naturaleza ofrecen respuestas basadas en probabilidades, por lo que la validación y corrección de estos resultados es algo fundamental en este trabajo, ya que las respuestas podrían ser erróneas. Evaluar la calidad del código en la refactorización, fue una de las medidas que se tomaron para poder demostrar la viabilidad de los resultados.

Durante el grado, en la rama de ingeniería del software, se nos hace hincapié en que es importante medir los atributos del software (mantenibilidad, complejidad ciclomática, etc.) para así poder analizar y validar de forma más efectiva el código. Todo esto lo relaciono directamente con la lección que acabo de mencionar.

Finalmente, la lección que considero más importante es la adaptabilidad. El mundo del software es un mundo cambiante, abstracto e innovador. Creo que prácticamente todo el contenido del grado está enfocado en esto, es decir, en crear personas capaces de adaptarse a cualquier circunstancia, independientemente de la tecnología, metodología o ámbito de la informática. Este grado busca formar ingenieros capaces de moldearse a cualquier situación, y esto es lo que he intentado conseguir, mediante la aplicación de los LLMs, un área innovadora, al tradicional y complejo ámbito del diseño del software.

Esta y otras muchas más lecciones que he aprendido, me han permitido realizar este trabajo de fin de grado aplicando todos los conocimientos que he adquirido. Realizar este trabajo no ha sido fácil, requiere de sacrificio, trabajo y conocimiento, pero tras este camino, me siento finalmente realizado.

5.3 Propuestas

Como ya he comentado en la sección de los objetivos, este trabajo no ha conseguido explorar todos los límites y capacidades de los modelos de lenguaje. Con la situación actual y los avances que se realizan día a día, considero que este se debe continuar por este camino. Quedan muchos ámbitos y propuestas por explorar, los modelos de lenguaje son capaces de generar imágenes, vídeos, analizar datos, generar código y más cosas de las que seremos conscientes en un futuro.

Además, las capacidades de estos modelos irán en aumento, incrementando así las posibilidades de experimentar con ellos. Mis propuestas a futuro van en consonancia con esto, sigamos estudiando la Inteligencia Artificial, busquemos formas de integrarlas en más ámbitos de la ingeniería informática, mejoremos la calidad de los procesos y los resultados. Todo con el objetivo final, de, efectivamente, ser mejores ingenieros de software.

Bibliografía

[1] *Object-Oriented Programming Concepts: A Primer*. Disponible en: https://web.mit.edu/java_v1.0.2/www/tutorial/java/objects/index.html. [Consulta: 21 marzo 2024].

[2] Snyder, A.. *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA.

[3] Microsoft. *Polimorfismo - C Sharp | Microsoft Learn*. Disponible en: https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/object-oriented/polymorphism. [Consulta: 23 marzo 2024].

[4] Martin, R. C.. *Principles of Object Oriented Design*. Disponible en: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. [Consulta: 27 marzo 2024].

[5] Madasu, V. K., Venna, T. V. S. N., y Eltaeib, T.. *SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP*. Journal of Multidisciplinary Engineering Science and Technology (JMEST), Vol. 2, Issue 2, febrero 2015.

[6] Gamma, E., Helm, R., Johnson, R., y Vlissides, J.. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[7] Latte, B., Henning, S., y Wojcieszak, M.. *Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living Software*. wobe-systems GmbH y Software Engineering Group, Kiel University, 2019.

[8] Martin, R. C.. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[9] Google Developers. *Introduction to Large Language Models*. Disponible en: https://developers.google.com/machine-learning/resources/intro-llms. [Consulta: 1 abril 2024].

[10] Amazon Web Services (AWS). *What are Large Language Models?*. Disponible en: https://aws.amazon.com/what-is/large-language-model/. [Consulta: 1 abril 2024].

[11] Amazon Web Services (AWS). *¿Qué son las RNN?*. Disponible en: https://aws.amazon.com/what-is/recurrent-neural-network/. [Consulta: 3 abril 2024].

[12] IBM. *What are Convolutional Neural Networks?*. Disponible en: https://www.ibm.com/topics/convolutional-neural-networks. [Consulta: 3 abril 2024].

[13] Amazon Web Services (AWS). *¿Qué son los transformadores en la inteligencia artificial?*. Disponible en: https://aws.amazon.com/es/what-is/transformers-in-artificial-intelligence/. [Consulta: 3 abril 2024].

[14] Google Cloud. *The Difference Between Supervised and Unsupervised Learning*. Disponible en: https://cloud.google.com/discover/supervised-vs-unsupervised-learning. [Consulta: 8 abril 2024].

[15] OpenAI. *Guide to Fine-Tuning*. Disponible en: https://platform.openai.com/docs/guides/fine-tuning. [Consulta: 8 abril 2024].

[16] Riedl, M.. *A Very Gentle Introduction to Large Language Models Without the Hype*. Medium. Disponible en: https://mark-riedl.medium.com/a-very-gentle-introduction-to-large-language-models-without-the-hype-5f67941fa [Consulta: 10 abril 2024].

[17] NVIDIA. *¿Qué es un Modelo Transformer?*. Disponible en: https://la.blogs.nvidia.com/blog/que-es-un-modelo-transformer/. [Consulta: 10 abril 2024].

[18] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L·L y Polosukhin, I.. *Attention Is All You Need*. Google Brain y Google Research, 2017.

[19] Microsoft. *Understanding strengths and weaknesses of Large Language Models*. Disponible en: https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/use-case-recommend. [Consulta: 15 abril 2024].

[20] Glovguy. *Large Language Models: Reasoning Capabilities and Limitations*. Medium. Disponible en: https://medium.com/@glovguy/large-language-models-reasoning-capabilities-and-limitations-951cee0ac642. [Consulta: 15 abril 2024].

[21] Microsoft. *Introduction to Prompt Engineering*. Disponible en: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering. [Consulta: 20 abril 2024].

[22] OpenAI. *Six Strategies for Getting Better Results in Prompt Engineering*. Disponible en: https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results. [Consulta: 20 abril 2024].

[23] SonarSource. *SonarQube: Code Quality, Security and Static Analysis Tool*. Disponible en: https://www.sonarsource.com/products/sonarqube/. [Consulta: 5 Junio 2024].

[24] Lamchau. *Refactoring Exercise. GitHub*. Disponible en: `https://github.com/lamchau/refactoring-exercise`. [Consulta: 4 mayo 2024].

[25] Microsoft. *Code Metrics - Cyclomatic Complexity*. Disponible en: `https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022`. [Consulta: 7 junio 2024].

[26] ChatGPT. *ChatGPT Official Website*. Disponible en: `https://chatgpt.com/`.

[27] Google. *Google Gemini App*. Disponible en: `https://gemini.google.com/app`.

[28] Refactoring.Guru. *Classification of Patterns*. Disponible en: `https://refactoring.guru/design-patterns/classification`. [Consulta: 20 mayo 2024].

[29] OpenAI. *GPT Builder*. Disponible en: `https://help.openai.com/en/articles/8770868-gpt-builder`. [Consulta: 15 junio 2024].

[30] Freeman, E., y Robson, E.. *Head First Design Patterns: A Brain-Friendly Guide*. 10th Anniversary Edition, O'Reilly Media, 2020.

APÉNDICE A

Conversaciones con Large Language Models

[C 1] Refinamiento del prompt para la refactorización.
https://chatgpt.com/share/df819196-6265-4701-84f7-016bcfa26623

[C 2] Refactorización del código con GPT-4.
https://chatgpt.com/share/d517b8fa-4ae6-4718-a9fe-31e53fd86aa6

[C 3] Refactorización del código con Gemini.
https://g.co/gemini/share/38b88891c293

[C 4] Refinamiento del prompt para la selección de patrones.
https://chatgpt.com/share/bb376c82-4154-4d49-8be5-c9cb142452b5

[C 5] Refinamiento del prompt para la implementación de patrones.
https://chatgpt.com/share/b38177f3-9979-418e-9d46-138fbd805fa0

[C 6] Patrones Creacionales con GPT-4: Factory Method.
https://chatgpt.com/share/2956c8e0-65d8-42b9-a024-910a35d7a49c

[C 7] Patrones Creacionales con GPT-4: Implementación de Factory Method.
https://chatgpt.com/share/e6c091ce-1c9e-420c-95c9-51611c8bc736

[C 8] Patrones Creacionales con Gemini: Strategy.
https://g.co/gemini/share/45261182c330

[C 9] Patrones Estructurales con GPT-4: Decorator.
https://chatgpt.com/share/de19dfdb-03fc-4caf-849c-bcf928c3e894

[C 10] Patrones Estructurales con GPT-4: Implementación de Decorator.
https://chatgpt.com/share/3567cf73-ebd2-45ca-806c-baf2714328b2

[C 11] Patrones Estructurales con Gemini: Decorator.
https://g.co/gemini/share/25e9731bfa6d

[C 12] Patrones Estructurales con Gemini: Implementación de Decorator.
https://g.co/gemini/share/ce51e881c8ca

[C 13] Patrones de Comportamiento con GPT-4: Command.
https://chatgpt.com/share/ec267747-e0e4-4df1-bc84-bc77a5c1d83d

[C 14] Patrones de Comportamiento con GPT-4: Implementación de Command.
https://chatgpt.com/share/0afd8294-3fc4-4a7b-bcc4-bf0fd08e1d4a

[C 15] Patrones de Comportamiento con Gemini: Memento.
https://g.co/gemini/share/5a8ec8a8a08f

[C 16] Patrones de Comportamiento con Gemini: Implementación de Memento.
https://g.co/gemini/share/bdb73dfa01b5

[C 17] Design Pattern Advisor: Factory Method.
https://chatgpt.com/share/455b26bc-4534-4ebb-8fef-28eb98f9bcbf

[C 18] Design Pattern Architect: Factory Method.
https://chatgpt.com/share/f96c03c1-3b94-42bd-bf94-ac908b31fb31

[C 19] Design Pattern Advisor: Decorator.
https://chatgpt.com/share/f962a11e-acfb-4b5a-91cc-335b04b5dc80

[C 20] Design Pattern Architect: Decorator.
https://chatgpt.com/share/b4af0207-ca0d-459e-b0c3-aa8ef3c172cf

[C 21] Design Pattern Advisor: Memento.
https://chatgpt.com/share/151d11cf-9db7-463f-a3d8-e8213b33a96c

[C 22] Design Pattern Architect: Memento.
https://chatgpt.com/share/a049574d-d4d8-4db3-97ae-ee2641cdf9cc

APÉNDICE B

Código en el Proceso de Refactorización

B.1 Código original

```

package com.checkr.interviews;

import java.util.*;
import com.opencsv.CSVReader;
import java.io.FileReader;
import java.io.IOException;

public class FundingRaised {

public static List<Map<String, String>> where(Map<String, String>
    options) throws IOException {

        List<String[]> csvData = new ArrayList<String[]>();
        CSVReader reader = new CSVReader(
          new FileReader("startup_funding.csv"));
        String[] row = null;

        while((row = reader.readNext()) != null) {
            csvData.add(row);
        }

        reader.close();
        csvData.remove(0);

        if(options.containsKey("company_name")) {
            List<String[]> results = new ArrayList<String[]> ();

            for(int i = 0; i < csvData.size(); i++) {
                if(csvData.get(i)[1].equals(options.get("company_name")
                    )) {
                    results.add(csvData.get(i));
                }
            }
            csvData = results;
        }

```

```
        if(options.containsKey("city")) {
            List<String[]> results = new ArrayList<String[]> ();

            for(int i = 0; i < csvData.size(); i++) {
                if(csvData.get(i)[4].equals(options.get("city"))) {
                    results.add(csvData.get(i));
                }
            }
            csvData = results;
        }

        if(options.containsKey("state")) {
            List<String[]> results = new ArrayList<String[]> ();

            for(int i = 0; i < csvData.size(); i++) {
                if(csvData.get(i)[5].equals(options.get("state"))) {
                    results.add(csvData.get(i));
                }
            }
            csvData = results;
        }

        if(options.containsKey("round")) {
            List<String[]> results = new ArrayList<String[]> ();

            for(int i = 0; i < csvData.size(); i++) {
                if(csvData.get(i)[9].equals(options.get("round"))) {
                    results.add(csvData.get(i));
                }
            }
            csvData = results;
        }

        List<Map<String, String>> output = new ArrayList<Map<String,
            String>>();

        for(int i = 0; i < csvData.size(); i++) {
            Map<String, String> mapped = new HashMap<String, String>()
                ;
            mapped.put("permalink", csvData.get(i)[0]);
            mapped.put("company_name", csvData.get(i)[1]);
            mapped.put("number_employees", csvData.get(i)[2]);
            mapped.put("category", csvData.get(i)[3]);
            mapped.put("city", csvData.get(i)[4]);
            mapped.put("state", csvData.get(i)[5]);
            mapped.put("funded_date", csvData.get(i)[6]);
            mapped.put("raised_amount", csvData.get(i)[7]);
            mapped.put("raised_currency", csvData.get(i)[8]);
            mapped.put("round", csvData.get(i)[9]);
            output.add(mapped);
        }

        return output;
    }

    public static Map<String, String> findBy(Map<String, String>
        options) throws IOException, NoSuchEntryException {

        List<String[]> csvData = new ArrayList<String[]>();
```

```java
        CSVReader reader = new CSVReader(
          new FileReader("startup_funding.csv"));
        String[] row = null;

        while((row = reader.readNext()) != null) {
            csvData.add(row);
        }

        reader.close();
        csvData.remove(0);
        Map<String, String> mapped = new HashMap<String, String>();

        for(int i = 0; i < csvData.size(); i++) {
            if(options.containsKey("company_name")) {
                if(csvData.get(i)[1].equals(options.get("company_name")
                    )) {
                    mapped.put("permalink", csvData.get(i)[0]);
                    mapped.put("company_name", csvData.get(i)[1]);
                    mapped.put("number_employees", csvData.get(i)[2]);
                    mapped.put("category", csvData.get(i)[3]);
                    mapped.put("city", csvData.get(i)[4]);
                    mapped.put("state", csvData.get(i)[5]);
                    mapped.put("funded_date", csvData.get(i)[6]);
                    mapped.put("raised_amount", csvData.get(i)[7]);
                    mapped.put("raised_currency", csvData.get(i)[8]);
                    mapped.put("round", csvData.get(i)[9]);
                } else {
                    continue;
                }
            }

            if(options.containsKey("city")) {
                if(csvData.get(i)[4].equals(options.get("city"))) {
                    mapped.put("permalink", csvData.get(i)[0]);
                    mapped.put("company_name", csvData.get(i)[1]);
                    mapped.put("number_employees", csvData.get(i)[2]);
                    mapped.put("category", csvData.get(i)[3]);
                    mapped.put("city", csvData.get(i)[4]);
                    mapped.put("state", csvData.get(i)[5]);
                    mapped.put("funded_date", csvData.get(i)[6]);
                    mapped.put("raised_amount", csvData.get(i)[7]);
                    mapped.put("raised_currency", csvData.get(i)[8]);
                    mapped.put("round", csvData.get(i)[9]);
                } else {
                    continue;
                }
            }

            if(options.containsKey("state")) {
                if(csvData.get(i)[5].equals(options.get("state"))) {
                    mapped.put("permalink", csvData.get(i)[0]);
                    mapped.put("company_name", csvData.get(i)[1]);
                    mapped.put("number_employees", csvData.get(i)[2]);
                    mapped.put("category", csvData.get(i)[3]);
                    mapped.put("city", csvData.get(i)[4]);
                    mapped.put("state", csvData.get(i)[5]);
                    mapped.put("funded_date", csvData.get(i)[6]);
                    mapped.put("raised_amount", csvData.get(i)[7]);
                    mapped.put("raised_currency", csvData.get(i)[8]);
```

```java
                    mapped.put("round", csvData.get(i)[9]);
                } else {
                    continue;
                }
            }

            if(options.containsKey("round")) {
                if(csvData.get(i)[9].equals(options.get("round"))) {
                    mapped.put("permalink", csvData.get(i)[0]);
                    mapped.put("company_name", csvData.get(i)[1]);
                    mapped.put("number_employees", csvData.get(i)[2]);
                    mapped.put("category", csvData.get(i)[3]);
                    mapped.put("city", csvData.get(i)[4]);
                    mapped.put("state", csvData.get(i)[5]);
                    mapped.put("funded_date", csvData.get(i)[6]);
                    mapped.put("raised_amount", csvData.get(i)[7]);
                    mapped.put("raised_currency", csvData.get(i)[8]);
                    mapped.put("round", csvData.get(i)[9]);
                } else {
                    continue;
                }
            }

            return mapped;
        }

        throw new NoSuchEntryException();
    }

}
```

B.2 Código Refactorizado por GPT-4

```java
package com.checkr.interviews;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.opencsv.CSVReader;

public class FundingRaisedGPT {

  private static final String CSV_FILE = "startup_funding.csv";

    public static List<Map<String, String>> where(Map<String, String> options) throws IOException {
        List<String[]> csvData = readCSV(CSV_FILE);
        applyFilters(csvData, options);
        return convertToMapList(csvData);
    }

```

```
    private static List<String[]> readCSV(String filePath) throws IOException {
        List<String[]> records = new ArrayList<>();
        try (CSVReader reader = new CSVReader(new FileReader(filePath))) {
            String[] row;
            while ((row = reader.readNext()) != null) {
                records.add(row);
            }
        }
        records.remove(0); // Remove header
        return records;
    }

    private static void applyFilters(List<String[]> data, Map<String, String> options) {
        for (Map.Entry<String, String> entry : options.entrySet()) {
            data.retainAll(filterByCriteria(data, entry.getKey(), entry.getValue()));
        }
    }

    private static List<String[]> filterByCriteria(List<String[]> data, String key, String value) {
        int index = getColumnIndex(key);
        List<String[]> filtered = new ArrayList<>();
        for (String[] row : data) {
            if (row[index].equals(value)) {
                filtered.add(row);
            }
        }
        return filtered;
    }

    private static List<Map<String, String>> convertToMapList(List<String[]> data) {
        List<Map<String, String>> list = new ArrayList<>();
        for (String[] row : data) {
            Map<String, String> mappedRow = new HashMap<>();
            mappedRow.put("permalink", row[0]);
            mappedRow.put("company_name", row[1]);
            mappedRow.put("number_employees", row[2]);
            mappedRow.put("category", row[3]);
            mappedRow.put("city", row[4]);
            mappedRow.put("state", row[5]);
            mappedRow.put("funded_date", row[6]);
            mappedRow.put("raised_amount", row[7]);
            mappedRow.put("raised_currency", row[8]);
            mappedRow.put("round", row[9]);
            list.add(mappedRow);
        }
        return list;
    }

    public static Map<String, String> findBy(Map<String, String> options) throws IOException, NoSuchEntryException {
        List<String[]> csvData = readCSV(CSV_FILE);
        String[] matchingEntry = findFirstMatchingEntry(csvData, options);
```

```
        if (matchingEntry != null) {
            return convertRowToMap(matchingEntry);
        }
        throw new NoSuchEntryException();
    }

    private static String[] findFirstMatchingEntry(List<String[]> data,
        Map<String, String> options) {
        for (String[] row : data) {
            if (matchesCriteria(row, options)) {
                return row;
            }
        }
        return null;
    }

    private static boolean matchesCriteria(String[] row, Map<String,
       String> options) {
        for (Map.Entry<String, String> option : options.entrySet()) {
            int index = getColumnIndex(option.getKey());
            if (!row[index].equals(option.getValue())) {
                return false;
            }
        }
        return true;
    }

    private static int getColumnIndex(String columnName) {
        switch (columnName) {
            case "company_name": return 1;
            case "city": return 4;
            case "state": return 5;
            case "round": return 9;
            default: throw new IllegalArgumentException("Unknown filter
                key: " + columnName);
        }
    }

    private static Map<String, String> convertRowToMap(String[] row) {
        Map<String, String> mapped = new HashMap<>();
        mapped.put("permalink", row[0]);
        mapped.put("company_name", row[1]);
        mapped.put("number_employees", row[2]);
        mapped.put("category", row[3]);
        mapped.put("city", row[4]);
        mapped.put("state", row[5]);
        mapped.put("funded_date", row[6]);
        mapped.put("raised_amount", row[7]);
        mapped.put("raised_currency", row[8]);
        mapped.put("round", row[9]);
        return mapped;
    }

}
```

B.3 Código Refactorizado por Gemini

```java

package com.checkr.interviews;

import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.opencsv.CSVReader;

public class FundingRaisedGemini {

    private static final String DATA_FILE = "startup_funding.csv";

    public static List<Map<String, String>> filterFundingData(Map<String, String> options) throws IOException {
      try (CSVReader reader = new CSVReader(new FileReader(DATA_FILE))) {
        List<String[]> rawData = reader.readAll();
        List<String[]> filteredData = filterByOptions(rawData, options);
        return convertToListOfMaps(filteredData);
      }
    }

    private static List<String[]> filterByOptions(List<String[]> data, Map<String, String> options) {
      List<String[]> filtered = new ArrayList<>();
      for (String[] row : data) {
        if (matchesOptions(row, options)) {
          filtered.add(row);
        }
      }
      return filtered;
    }

    private static boolean matchesOptions(String[] row, Map<String, String> options) {
      for (Map.Entry<String, String> option : options.entrySet()) {
        int index = getOptionIndex(option.getKey());
        if (index != -1 && !row[index].equals(option.getValue())) {
          return false;
        }
      }
      return true;
    }

    private static int getOptionIndex(String option) {
      switch (option) {
        case "company_name":
          return 1;
        case "city":
          return 4;
        case "state":
```

```
          return 5;
        case "round":
          return 9;
        default:
          return -1;
      }
    }

    private static List<Map<String, String>> convertToListOfMaps(List<
       String[]> data) {
      List<Map<String, String>> mappedData = new ArrayList<>();
      for (String[] row : data) {
        mappedData.add(convertRowToMap(row));
      }
      return mappedData;
    }

    private static Map<String, String> convertRowToMap(String[] row) {
      Map<String, String> map = new HashMap<>();
      map.put("permalink", row[0]);
      map.put("company_name", row[1]);
      map.put("number_employees", row[2]);
      map.put("category", row[3]);
      map.put("city", row[4]);
      map.put("state", row[5]);
      map.put("funded_date", row[6]);
      map.put("raised_amount", row[7]);
      map.put("raised_currency", row[8]);
      map.put("round", row[9]);
      return map;
    }

    public static Map<String, String> findOneBy(Map<String, String>
       options) throws IOException, NoSuchEntryException {
        try (CSVReader reader = new CSVReader(new FileReader(DATA_FILE)
           )) {
            List<String[]> data = reader.readAll();
            return findMatchingEntry(data, options);
        }
    }

    private static Map<String, String> findMatchingEntry(List<String[]>
        data, Map<String, String> options) throws NoSuchEntryException
        {
        for (String[] row : data) {
            if (matchesOptions(row, options)) {
                Map<String, String> map = convertRowToMap(row);
                return map;
            }
        }
        throw new NoSuchEntryException();
    }

}
```

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**	X			
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**			X	
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**		X		

Relación del TFG con los Objetivos de Desarrollo Sostenible

El Trabajo de Fin de Grado aborda la integración de la Inteligencia Artificial Generativa en el diseño de software mediante el uso de Modelos de Lenguaje Natural (LLMs) como GPT-4 y Gemini. Esta investigación se centra en mejorar la calidad y eficiencia de los procesos de diseño de software, lo cual tiene una relación directa e indirecta con varios Objetivos de Desarrollo Sostenible (ODS). A continuación, se detalla cómo este TFG se relaciona con algunos de los ODS más relevantes.

ODS 4: Educación de Calidad

El **ODS 4** busca garantizar una educación inclusiva, equitativa y de calidad, y promover oportunidades de aprendizaje durante toda la vida para todos. Este TFG contribuye a este objetivo de varias maneras:

- El TFG proporciona una base sólida para la educación en el campo de la ingeniería del software. La documentación detallada de metodologías y procesos de integración de LLMs puede ser utilizada en cursos y programas de formación, ayudando a futuros ingenieros a adaptarse a las nuevas tecnologías y mejorar sus competencias. Este enfoque no solo mejora la calidad de la educación, sino que también facilita el aprendizaje continuo, un aspecto fundamental del ODS 4 .
- La implementación de LLMs y la exploración de sus capacidades y limitaciones en el diseño de software promueven el aprendizaje continuo entre los ingenieros de software. Al aprender a utilizar estas herramientas avanzadas, los profesionales se mantienen actualizados con las últimas tecnologías, mejorando su habilidad y competencia en el campo.

ODS 9: Industria, Innovación e Infraestructura

El **ODS 9** busca construir infraestructuras resilientes, promover la industrialización inclusiva y sostenible y fomentar la innovación. El TFG se relaciona con este objetivo de las siguientes maneras:

- La innovación es el núcleo del TFG, ya que explora nuevas aplicaciones de LLMs en el diseño de software. Al desarrollar y documentar nuevas metodologías, el trabajo no solo mejora las prácticas actuales sino que también establece una base para futuras investigaciones y desarrollos en la industria del software. Este enfoque promueve una industrialización más avanzada y la creación de infraestructuras tecnológicas más resilientes y eficientes.
- La investigación destaca el uso innovador de LLMs para resolver problemas complejos en el diseño de software. Al explorar nuevas metodologías y procesos, el trabajo fomenta la innovación dentro del campo de la ingeniería del software, proporcionando nuevas soluciones y mejoras a problemas existentes.
- La implementación y refinamiento de modelos GPT especializados para el diseño de software representan un avance significativo en la infraestructura tecnológica. Estas herramientas mejoran la capacidad de las empresas para desarrollar software de alta calidad de manera más eficiente y efectiva.

ODS 8: Trabajo Decente y Crecimiento Económico

El **ODS 8** tiene como objetivo promover el crecimiento económico sostenido, inclusivo y sostenible, el empleo pleno y productivo y el trabajo decente para todos. El TFG contribuye a este objetivo de las siguientes maneras:

- La investigación sobre la aplicación de LLMs en el diseño de software demuestra cómo estas herramientas pueden reducir la carga de trabajo de los ingenieros y mejorar la eficiencia de los procesos de desarrollo. Esto, a su vez, puede incrementar la productividad en las empresas de software, contribuyendo al crecimiento económico del sector.
- Al fomentar la adopción de tecnologías avanzadas como la IA en el diseño de software, se crean nuevas oportunidades de empleo en el sector tecnológico. Los ingenieros de software deben aprender a interactuar y trabajar con estas nuevas herramientas, lo que requiere formación y habilidades avanzadas, elevando el nivel de calidad del empleo en la industria.

ODS 12: Producción y Consumo Responsables

El **ODS 12** tiene como objetivo garantizar modalidades de consumo y producción sostenibles. El TFG contribuye a este objetivo al mejorar la eficiencia y sostenibilidad en el desarrollo de software:

- La investigación sobre la refactorización de código y la implementación de patrones de diseño mediante LLMs ayuda a crear software más mantenible y eficiente. Esto reduce el desperdicio de recursos y el tiempo dedicado a la corrección de errores y mantenimiento, promoviendo prácticas de desarrollo más sostenibles.
- Al enfatizar la importancia del *Clean Code* y los principios SOLID en el desarrollo de software, el TFG promueve una cultura de calidad y sostenibilidad en la ingeniería del software. Esto no solo mejora la calidad del software producido, sino que también garantiza que los sistemas sean duraderos y fáciles de mantener.

Conclusiones

En conclusión, el TFG guarda una relación significativa con varios ODS. A través de la educación, la mejora de la productividad, la promoción de la innovación y el uso responsable de los recursos, este trabajo contribuye de manera significativa a los objetivos globales de desarrollo sostenible. La integración de LLMs en el diseño de software no solo mejora las prácticas actuales, sino que también sienta las bases para un futuro más sostenible y eficiente en la industria tecnológica .
La relevancia de estos ODS en el contexto del TFG subraya la importancia de continuar explorando y aplicando tecnologías avanzadas como la IA en diversos campos, para así contribuir al logro de los objetivos globales de sostenibilidad y desarrollo.