|  |  |  |
| --- | --- | --- |
| Una caricatura de una persona  Descripción generada automáticamente con confianza baja | UNIVERSIDAD DE BURGOS  ESCUELA POLITÉCNICA SUPERIOR  Grado en Ingeniería Informática |  |

**TFG del Grado en Ingeniería Informática**

**<Sistema de extracción y gestión automatizada de requisitos desde documentos PDF para proyectos de ingeniería>**

Presentado por Estíbalitz Díez en Universidad de Burgos - 15 de enero de 2025

Tutores: Dr. José Manuel Galán Ordax y Dr. José Ignacio Santos Martin

|  |  |  |
| --- | --- | --- |
| Una caricatura de una persona  Descripción generada automáticamente con confianza baja | UNIVERSIDAD DE BURGOS  ESCUELA POLITÉCNICA SUPERIOR  Grado en Ingeniería Informática |  |

D. José Manuel Galán Ordax y D. José Ignacio Santos Martin, profesores del departamento de Ingeniería de Organización, área de Organización de Empresas.

Exponen:

Que el alumno Dña. Estíbalitz Díez, con DNI 78.893.267R, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado “***Sistema de extracción y gestión automatizada de requisitos desde documentos PDF para proyectos de ingeniería***”.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección de los que suscriben, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 15 de enero de 2025

Vº. Bº. del Tutor:

D. José Manuel Galán Ordax

Vº. Bº. del co-tutor:

D. José Ignacio Santos Martin

RESUMEN

Descriptores

ABSTRACT

Keywords

Índice general

[Índice general III](#_Toc182769052)

[Índice de figuras VI](#_Toc182769053)

[Índices de tablas VII](#_Toc182769054)

[1. Introducción 1](#_Toc182769055)

[1.1. Estructura de la memoria 1](#_Toc182769056)

[1.2. Materiales adjuntos 1](#_Toc182769057)

[2. Objetivos del proyecto 3](#_Toc182769058)

[2.1. Objetivos generales 3](#_Toc182769059)

[2.2. Objetivos técnicos 3](#_Toc182769060)

[2.3. Objetivos personales 4](#_Toc182769061)

[3. Conceptos teóricos 5](#_Toc182769062)

[3.1. Procesamiento de texto 5](#_Toc182769063)

[3.1.1 Extracción de Información y Estructuración Jerárquica 5](#_Toc182769064)

[3.1.2 Búsqueda de Tokens y Relación con Subsistemas 6](#_Toc182769065)

[3.2. Base de datos 7](#_Toc182769066)

[3.2.1 Visión General del Diseño 7](#_Toc182769067)

[3.2.2 Principios de Diseño Aplicados 7](#_Toc182769068)

[3.2.3 Decisión Tecnológica: Opciones Evaluadas 8](#_Toc182769069)

[3.2.4 Decisión Final: SQLite 9](#_Toc182769070)

[3.2.5 Relación con la Especificación Técnica (Anexo C) 9](#_Toc182769071)

[3.3. Algoritmo de Búsqueda detección de Subsistema. 10](#_Toc182769072)

[3.3.1 Visión General 10](#_Toc182769073)

[3.3.2 Alternativas Evaluadas 10](#_Toc182769074)

[3.3.3 Decisión Tomada 11](#_Toc182769075)

[3.3.4 Conceptos Relacionados 12](#_Toc182769076)

[3.3.5 Implementación Actual 12](#_Toc182769077)

[3.3.6 Conclusión 13](#_Toc182769078)

[3.4. Diseño de Interfaz y Usabilidad. 13](#_Toc182769079)

[3.4.1 Sencillez e Intuición en el Diseño 13](#_Toc182769080)

[3.4.2 Gestión de Idiomas en un Entorno Internacional 14](#_Toc182769081)

[3.4.3 Decisiones Clave en el Diseño 14](#_Toc182769082)

[3.4.4 Reflexión Basada en la Teoría 15](#_Toc182769083)

[3.4.5 Conclusión 15](#_Toc182769084)

[3.5. Diseño análisis de software 15](#_Toc182769085)

[3.5.1 Decisiones Clave en el Diseño 16](#_Toc182769086)

[3.5.2 Reflexión Teórica 16](#_Toc182769087)

[3.5.3 Conclusión 17](#_Toc182769088)

[4. Técnicas y herramientas 18](#_Toc182769089)

[4.1. Técnicas metodológicas 18](#_Toc182769090)

[4.1.1 Enumeración y Explicación de las Metodologías 18](#_Toc182769091)

[4.1.2 3. Relación entre Metodologías 19](#_Toc182769092)

[4.1.3 4. Conclusión 20](#_Toc182769093)

[4.2. Patrones de diseño 20](#_Toc182769094)

[4.2.1 Arquitectura en Capas 20](#_Toc182769095)

[4.2.2 Ventajas de la Arquitectura en Capas 21](#_Toc182769096)

[4.2.3 Patrones de Diseño Aplicados 21](#_Toc182769097)

[4.2.4 Conclusión 22](#_Toc182769098)

[4.3. Alojamiento de repositorio 22](#_Toc182769099)

[4.4. Control de versiones 22](#_Toc182769100)

[4.5. Entorno de desarrollo integrado (IDE) 23](#_Toc182769101)

[4.6. Gestión del proyecto 24](#_Toc182769102)

[4.7. Comunicación 25](#_Toc182769103)

[4.8. Documentación de memoria 26](#_Toc182769104)

[4.9. Documentación del código 27](#_Toc182769105)

[4.10. Integración y despliegue continuos (CI/CD) 28](#_Toc182769106)

[4.11. Calidad y consistencia del código 30](#_Toc182769107)

[4.12. Cobertura de código 32](#_Toc182769108)

[4.13. Framework escritorio 33](#_Toc182769109)

[4.14. Sistema gestor de bases de datos 34](#_Toc182769110)

[4.15. Bibliotecas y librerías relevantes 35](#_Toc182769111)

[4.16. Otras herramientas 36](#_Toc182769112)

[5. Aspectos relevantes del desarrollo del proyecto 37](#_Toc182769113)

[5.1. Gestión de Documentación Técnica en Proyectos Ferroviarios: Primer Pilar del Desarrollo 37](#_Toc182769114)

[5.1.1 Entrega y Gestión de Documentación Técnica 37](#_Toc182769115)

[5.1.2 Problema Operativo: Bucear en la Documentación 38](#_Toc182769116)

[5.1.3 Impacto de una Mala Gestión de Requisitos 38](#_Toc182769117)

[5.1.4 Solución Propuesta: Herramienta Especializada 39](#_Toc182769118)

[5.1.5 Justificación del Proyecto 39](#_Toc182769119)

[5.1.6 Diseño desde una Perspectiva Global 39](#_Toc182769120)

[5.2. Gestión de tiempo y plazos de entrega 39](#_Toc182769121)

[5.2.1 La Importancia de una Versión Inicial Funcional 39](#_Toc182769122)

[5.2.2 Desafíos del Tiempo Reducido 40](#_Toc182769123)

[5.2.3 Comparativa con un Contexto Profesional 40](#_Toc182769124)

[5.2.4 Milestones y Planificación Estratégica 40](#_Toc182769125)

[5.2.5 Impacto de la Falta de Tiempo de Investigación y Formación 41](#_Toc182769126)

[5.2.6 Reflexión sobre la Planificación 41](#_Toc182769127)

[5.3. Inexperiencia Profesional y Desconocimiento Práctico en la Gestión de un Proyecto de Software 42](#_Toc182769128)

[5.4. Sencillez arquitectónica del proyecto como estrategia fundamental. 44](#_Toc182769129)

[5.5. Revisión y Mejora del Código con Herramientas de Análisis 45](#_Toc182769130)

[5.6. Creación de una Identidad Visual 46](#_Toc182769131)

[5.7. Planificación de una Segunda Versión 46](#_Toc182769132)

[5.8. Revisión y Aplicación de Conceptos Académicos 46](#_Toc182769133)

[5.9. Adaptación a la Metodología Scrum 47](#_Toc182769134)

[5.10. Reflexión sobre la Adaptabilidad y el Aprendizaje Continuo 47](#_Toc182769135)

[5.11. Reflexión Final sobre los Aspectos Relevantes 47](#_Toc182769136)

[6. Trabajos relacionados 48](#_Toc182769137)

[7. Conclusiones y Líneas de trabajo futuras 49](#_Toc182769138)

[7.1. Conclusiones 49](#_Toc182769139)

[7.2. Líneas de trabajo futuras 49](#_Toc182769140)

[Bibliografía 50](#_Toc182769141)

Índice de figuras

[Figura 3.1: Imagen modelo 3](#_Toc177844215)

Índices de tablas

[Tabla 4.1 Tabla modelo 4](#_Toc177844212)

# Introducción

Texto introductorio

## Estructura de la memoria

La memoria sigue la siguiente estructura:

Introducción: breve descripción del problema a resolver y la solución propuesta. Estructura de la memoria y listado de materiales adjuntos.

Objetivos del proyecto: exposición de los objetivos que persigue el proyecto.

Conceptos teóricos: breve explicación de los conceptos teóricos clave para la comprensión de la solución propuesta.

Técnicas y herramientas: listado de técnicas metodológicas y he rramientas utilizadas para gestión y desarrollo del proyecto.

Aspectos relevantes del desarrollo: exposición de aspectos destacables que tuvieron lugar durante la realización del proyecto.

Trabajos relacionados: estado del arte en las aplicaciones y sitios web de bolsa y finanzas.

Conclusiones y líneas de trabajo futuras: conclusiones obtenidas tras la realización del proyecto y posibilidades de mejora o expansión de la solución aportada.

Junto a la memoria se proporcionan los siguientes anexos:

Plan del proyecto software: planificación temporal y estudio de viabilidad del proyecto.

Especificación de requisitos del software: se describe la fase de análisis; los objetivos generales, el catálogo de requisitos del sistema y la especificación de requisitos funcionales y no funcionales.

Especificación de diseño: se describe la fase de diseño; el ámbito del software, el diseño de datos, el diseño procedimental y el diseño arquitectónico.

Manual del programador: recoge los aspectos más relevantes relacionados con el código fuente (estructura, compilación, instalación, ejecución, pruebas, etc.).

Manual de usuario: guía de usuario para el correcto manejo de la aplicación

## Materiales adjuntos

Los materiales que se adjuntan con la memoria son:

Anexos: consultar la documentación técnica.

Herramienta web: este proyecto está especialmente pensado para ser utilizado en un entorno local, pero se ha dispuesto una página web para que el usuario pueda hacer pruebas antes de instalarlo en su equipo. Visitar FAT: Financial Analysis Tool.

Vídeo de presentación: ver presentación del TFG.

Vídeo de demostración: ver demostración funcional.

Además, los siguientes recursos están accesibles a través de internet:

Repositorio: visitar FAT. GitHub.

Documentación del código: visitar FAT. Read The Docs.

Logs de calidad de código: la calidad del código se ha medido de forma automatizada con pylint y GitHub actions. Los informes generados muestran sólo los aspectos mejorables. Es decir, aquello que no está en los logs es porque está con la calidad óptima. Descargar informe de pylint.

Cobertura de código: La cobertura de los tests se ha medido de formaautomatizada con coverage y GitHub actions. Es posible descargar un informe HTML en formato .zip. Se descomprime en un directorio y se lanza desde index.html. Descargar informe de coverage.

# Objetivos del proyecto

Los objetivos de este Trabajo de Fin de Grado podemos englobarlos en tres categorías según el siguiente enfoque:

## Objetivos generales

El proyecto consiste en la creación de una herramienta de software que ayude a automatizar el proceso de manejo de requisitos extraídos de documentos PDF en proyectos de ingeniería. A través de una interfaz sencilla, los documentos serán procesados para extraer la información, permitiendo al usuario gestionar, organizar y visualizar los requisitos, comparar versiones de documentos, y finalmente exportar esta información en formatos estándar como Excel o CSV. El sistema también incluirá funcionalidades adicionales para el seguimiento del cumplimiento de requisitos y la asignación automática de subsistemas.

Objetivos específicos:

• Automatización de la extracción de contenido

• Gestión y almacenamiento de los requisitos

• Interfaz de usuario intuitiva

• Comparación de versiones

• Exportación de datos

• Funcionalidad añadida: seguimiento del estado de los requisitos.

## Objetivos técnicos

Desarrollar un herramienta software de escritorio sencilla a la que le pueda incorporar fácilmente nuevas funcionalidades y pueda gestionar bases de datos de mayor volumen con adaptaciones o migraciones sencillas.

* Crear bases de datos SQLite con gestión desde el propio software.
* Diseñar formularios que permitan la interacción con el usuario para realizar operaciones CRUD en la base de datos principal y operaciones de lectura en las bases de datos de los valores cotizados.
* Utilizar Git como sistema de control de versiones distribuido junto con la plataforma GitHub.
* Hacer uso de herramientas CI/CD integradas en el repositorio con GitHub actions. Por ejemplo, utilizar pyllint como herramienta de control de calidad de código, y SonarQube para chequear de forma continuada el desarrollo del proyecto.
* Aplicar la metodología ágil Scrum junto con Zube como herramienta de gestión de proyectos.

Utilizar un sistema de documentación como Sphinx con el estilo de Read The Docs y con la posibilidad de subir la documentación de forma continua.

## Objetivos personales

Crear una herramienta sencilla para resolver la problemática de gestión de gran volumen de documentación de requisitos de un modo ágil y seguro.

Poner en práctica los conocimientos adquiridos para plantear la solución más eficiente al problema planteado.

Explorar y aprender metodologías, herramientas y estándares utilizados en el mercado laboral.

# Conceptos teóricos

En este proyecto, la toma de decisiones ha estado guiada por un análisis minucioso de las diferentes alternativas teóricas disponibles. Cada problema planteado ha sido abordado evaluando diversas opciones que, podían ser válidas y posteriormente, .se ha seleccionado la solución más adecuada en base a los objetivos específicos de este trabajo.

Se han trabajado seis áreas desde un punto de vista teórico que desarrollamos en los siguientes subapartados. Estas áreas abarcan aspectos clave del diseño y desarrollo del sistema, y reflejan cómo los fundamentos teóricos han guiado las decisiones tomadas durante el proyecto.

## Procesamiento de texto

En el procesamiento de texto resolvimos dos tareas totalmente diferenciadas:

### Extracción de Información y Estructuración Jerárquica

#### Alternativas Evaluadas

En esta primera etapa, se plantearon diferentes enfoques para la extracción de información de los documentos PDF y su organización jerárquica:

1. Procesamiento Lineal:

* Recorrer el texto del documento de principio a fin, identificando capítulos, subcapítulos y párrafos mediante patrones predefinidos, como encabezados (Capítulo X, 1.1.) o viñetas.
* Ventaja: Sencillez en la implementación y adecuación para documentos con estructuras consistentes.
* Inconveniente: Limitado en flexibilidad para manejar documentos con formatos más complejos.

1. Análisis Basado en Árboles Jerárquicos:

* Utilizar un modelo basado en gramáticas para analizar la estructura del texto y generar automáticamente un árbol jerárquico que represente capítulos y párrafos.
* Ventaja: Permite manejar documentos con estructuras complejas.
* Inconveniente: Requiere un esfuerzo considerable en la definición de gramáticas específicas para cada tipo de documento.

1. Segmentación en Bloques por Análisis Visual:

* Procesar el documento analizando visualmente la distribución del texto, detectando espacios, tamaños de fuente o estilos para identificar capítulos y secciones.
* Ventaja: Adecuado para documentos con formato variado.
* Inconveniente: Depende del uso de herramientas adicionales para extraer metadatos visuales de los PDF.

#### Decisión Tomada

Se optó por el Procesamiento Lineal utilizando patrones predefinidos. Esta solución fue elegida debido a que:

* Los documentos utilizados en el proyecto presentan una estructura relativamente consistente, con capítulos claramente delimitados y listas organizadas.
* Este enfoque permite extraer y organizar la información en una jerarquía simple (Documento > Capítulos > Párrafos) sin necesidad de análisis visual complejo ni definición de gramáticas avanzadas.

#### Implementación y Posibilidad de Edición

Una vez extraída, la información se organiza en una estructura jerárquica que:

1. Permite la Edición por Parte del Usuario:

* Antes de almacenar los datos en la base de datos, el sistema muestra los capítulos y párrafos extraídos para que el usuario pueda revisarlos y ajustarlos según sea necesario.

1. Almacena la Información en Formato Jerárquico:

* Cada capítulo y párrafo se almacena con su posición dentro del documento, asegurando que la estructura se mantenga intacta para futuras consultas.

### Búsqueda de Tokens y Relación con Subsistemas

#### Alternativas Evaluadas

Para determinar los subsistemas relacionados con un documento de requisitos, se exploraron las siguientes alternativas:

1. Uso de Modelos de Lenguaje (MLL):

* Aplicar modelos avanzados como BERT o GPT para analizar cada párrafo del documento y predecir automáticamente los subsistemas relacionados.
* Ventaja: Ofrece una solución automatizada basada en aprendizaje profundo.
* Inconveniente:
* Confidencialidad y Seguridad: Los datos procesados podrían estar expuestos a vulnerabilidades.
* Complejidad en la Gestión: Requiere reentrenar modelos si cambian los requisitos o los subsistemas.
* Rendimiento: Este enfoque sería excesivo en relación con la necesidad de asociar documentos completos a subsistemas.

1. Comparación Directa con Listas de Palabras Clave:

* Utilizar listas predefinidas de palabras clave asociadas a cada subsistema para buscar coincidencias en el texto completo del documento.
* Ventaja:
* Simplicidad: Fácil de implementar y gestionar.
* Flexibilidad: Las listas de palabras clave pueden actualizarse fácilmente según cambien los requisitos o subsistemas.
* Inconveniente: Depende de la calidad y exhaustividad de las listas de palabras clave.

#### Decisión Tomada

Se optó por un enfoque basado en listas de palabras clave debido a que:

* Los documentos analizados suelen tratar un tema general (por ejemplo, conectores de energía) aplicable a un subsistema específico. Por tanto, buscar palabras clave en el documento completo es suficiente para establecer asociaciones, sin necesidad de analizar párrafo por párrafo.
* Este método ofrece una solución segura y eficiente, permitiendo que las listas de tokens sean gestionadas y actualizadas directamente por los usuarios.

#### Implementación y Gestión de Tokens

1. Comparación Directa:

* El texto completo del documento se compara con las listas de palabras clave asociadas a cada subsistema.
* Cada coincidencia genera una relación entre el documento y el subsistema correspondiente.

1. Gestión de Tokens:

* Las palabras clave se gestionan mediante archivos de configuración, lo que permite añadir, eliminar o modificar tokens sin afectar el funcionamiento del sistema.
* Esta modularidad asegura que el sistema sea fácilmente adaptable a cambios en los subsistemas o requisitos.

#### Justificación Teórica y Reflexión

Este enfoque se fundamenta en los principios aprendidos en la asignatura de Procesadores de Lenguaje, aplicando conceptos básicos de análisis léxico para identificar patrones simples y establecer asociaciones semánticas directas. A pesar de que los modelos de lenguaje podrían ofrecer más automatización, la simplicidad, flexibilidad y seguridad del enfoque basado en palabras clave lo hacen más adecuado para este proyecto.

## Base de datos

### Visión General del Diseño

El diseño de la base de datos se fundamentó en los requisitos funcionales y no funcionales establecidos en la Especificación de Requisitos (Anexo B). Este sistema debía cumplir con las siguientes necesidades:

Gestión eficiente de los requisitos:

* Organizar los requisitos en una estructura jerárquica que relacione proyectos, documentos y subsistemas.

Adaptabilidad y escalabilidad:

* Permitir la evolución del sistema para futuras ampliaciones, como el almacenamiento de datos adicionales o la migración a bases de datos más robustas.

Simplicidad y portabilidad:

* Garantizar que el sistema pueda ejecutarse sin necesidad de infraestructura compleja, manteniendo la base de datos en un entorno local y controlado.

### Principios de Diseño Aplicados

El diseño de las tablas y relaciones, detallado en la Especificación de Diseño (Anexo C), sigue los siguientes principios:

Modularidad y Cohesión Alta:

* Cada tabla representa un concepto único (proyectos, documentos, requisitos, asociaciones), asegurando una clara separación de responsabilidades.

Normalización:

* Las tablas fueron normalizadas hasta la Tercera Forma Normal (3NF) para evitar redundancias y garantizar la integridad de los datos.

Restricciones y Relaciones:

* Las claves primarias y foráneas aseguran consistencia en las relaciones entre tablas, aplicando integridad referencial en todas las asociaciones.

### Decisión Tecnológica: Opciones Evaluadas

Durante el diseño, se consideraron varias tecnologías de bases de datos, evaluando sus ventajas y desventajas para determinar la solución más adecuada.

#### SQLite

Ventajas:

* Simplicidad: Es una base de datos embebida que no requiere un servidor, ideal para sistemas de escritorio.
* Portabilidad: Almacena los datos en un único archivo, lo que facilita el transporte y la copia.
* Control local: Los datos permanecen en el equipo del usuario, eliminando riesgos de seguridad asociados a conexiones remotas.
* Facilidad de migración: La estructura relacional permite migrar a bases de datos más robustas si el sistema escala.

Desventajas:

* Limitaciones en concurrencia: No es óptima para entornos multiusuario.
* Capacidad limitada: Menos adecuada para grandes volúmenes de datos o transacciones complejas.

#### Oracle

Ventajas:

* Escalabilidad: Maneja grandes volúmenes de datos y usuarios concurrentes.
* Funcionalidades avanzadas: Ofrece particionamiento de datos, auditoría y gestión avanzada de transacciones.
* Fiabilidad: Alta disponibilidad y recuperación ante fallos.

Desventajas:

* Complejidad: Requiere configuración avanzada y personal capacitado.
* Coste: Licencias elevadas en comparación con otras opciones.
* Sobrecarga innecesaria: Muchas de sus características no son necesarias para esta versión del sistema.

#### PostgreSQL

Ventajas:

* Código abierto: Sin costos de licencia.
* Concurrencia y escalabilidad: Soporta múltiples usuarios y grandes volúmenes de datos.
* Funcionalidades avanzadas: Incluye tipos de datos personalizados y transacciones complejas.

Desventajas:

* Complejidad de instalación: Requiere un servidor y configuración adicional.
* Exceso de características para esta versión inicial: Similar a Oracle, sus capacidades superan los requisitos actuales.

#### Bases de Datos NoSQL (como MongoDB)

Ventajas:

* Flexibilidad: Almacena datos con estructuras variables.
* Escalabilidad horizontal: Maneja grandes volúmenes de datos distribuidos.

Desventajas:

* Desalineación con el modelo relacional: El sistema actual requiere un esquema relacional para cumplir con los requisitos.
* Complejidad de integración: Requeriría un replanteamiento completo del diseño actual.

### Decisión Final: SQLite

Tras analizar estas alternativas, SQLite fue seleccionada como la solución más adecuada debido a:

Compatibilidad con el Software de Escritorio:

* Permite operar sin necesidad de servidores, reduciendo la complejidad técnica y los riesgos de seguridad.

Simplicidad y Portabilidad:

* Facilita el despliegue en entornos locales con requisitos mínimos.

Adaptabilidad:

* La estructura relacional diseñada es compatible con bases de datos más robustas, como PostgreSQL u Oracle, para futuras ampliaciones.

### Relación con la Especificación Técnica (Anexo C)

El diseño implementado en SQLite respalda directamente los casos de uso descritos en el Anexo C:

* Consulta de Requisitos y Documentos:
* Las relaciones entre tablas permiten realizar consultas eficientes, como buscar documentos asociados a un proyecto o subsistemas relacionados con un requisito específico.
* Gestión de Documentos y Subsistemas:
* Las claves foráneas garantizan que cada requisito esté vinculado a su documento y proyecto, y opcionalmente a uno o más subsistemas.
* El modelo relacional utilizado para este diseño se ilustra en los diagramas E/R del Anexo C.

## Algoritmo de Búsqueda detección de Subsistema.

### Visión General

El objetivo del algoritmo de búsqueda es identificar subsistemas relacionados con un documento de requisitos, basado en la presencia de palabras clave (tokens) asociadas a cada subsistema. La filosofía detrás del diseño no es restringir, sino sugerir todas las posibles relaciones de alcance, dejando al usuario la decisión final sobre cuáles aplicar. Este enfoque busca maximizar la utilidad de la herramienta sin caer en restricciones innecesarias que puedan omitir relaciones importantes.

Desde este planteamiento, se consideraron diversas alternativas para optimizar la búsqueda, incluyendo técnicas como programación concurrente y algoritmos avanzados de búsqueda. Sin embargo, el tamaño limitado de las listas de tokens y la previsión de crecimiento moderado llevaron a descartar enfoques más complejos, priorizando un diseño simple, claro y suficiente para las necesidades actuales del sistema.

### Alternativas Evaluadas

#### Búsqueda Secuencial por Subsistema (Actual)

En el diseño actual, el algoritmo recorre secuencialmente:

Para cada subsistema, compara todos los tokens con el contenido completo del documento.

La relación con un subsistema se sugiere si se detecta al menos una coincidencia, aunque la decisión final recae siempre en el usuario.

Ventajas:

* Implementación sencilla y eficiente para el tamaño actual de los datos.
* Garantiza una cobertura completa del documento y las palabras clave de todos los subsistemas.
* Es fácilmente adaptable a cambios en la lista de tokens o subsistemas.

Desventajas:

* Escalabilidad limitada si se añaden muchos subsistemas o tokens (aunque este caso es improbable).

#### Búsqueda Concurrente por Subsistema (Valorada)

Se evaluó la posibilidad de implementar programación concurrente, distribuyendo la búsqueda de cada subsistema como una tarea independiente. En este caso:

* Cada tarea procesaría la lista de tokens de un subsistema contra el contenido completo del documento.
* Las tareas se ejecutarían en paralelo, utilizando hilos o procesos para aprovechar múltiples núcleos de CPU.

Ventajas:

* Reducción del tiempo total de búsqueda en sistemas con múltiples núcleos.
* Escalable para futuros escenarios con más subsistemas o documentos más extensos.

Desventajas:

* Overhead innecesario: La lista de tokens y el número de subsistemas actuales no justifican el costo adicional en recursos.
* Complejidad en la Implementación: Introducir concurrencia aumenta la dificultad de desarrollo y mantenimiento del sistema.
* Impacto Reducido en Rendimiento: Dado el volumen moderado de datos, los beneficios de la concurrencia serían marginales.

#### Búsqueda Basada en Índices o Algoritmos Avanzados

Se consideró el uso de técnicas como:

Índices invertidos: Estructuras de datos que permiten encontrar rápidamente palabras en grandes volúmenes de texto.

Algoritmos de búsqueda binaria: Requieren ordenar las listas de tokens previamente.

Ventajas:

* Aceleran la búsqueda en casos con listas muy extensas.
* Reducen la cantidad de comparaciones necesarias.

Desventajas:

* Sobrecoste innecesario: El número reducido de tokens y subsistemas hace que estas optimizaciones sean redundantes.
* Complejidad Adicional: Requieren más preprocesamiento y mantenimiento.

### Decisión Tomada

Se optó por la búsqueda secuencial por subsistema debido a:

Simplicidad y Suficiencia:

* El volumen actual de datos (10 subsistemas, 30 tokens por subsistema) y el tamaño de los documentos (hasta 50 páginas) no representan un desafío significativo para el enfoque secuencial.

Enfoque de Sugerencia No Restrictiva:

* La búsqueda no pretende ser estricta, sino identificar todas las posibles relaciones de alcance para que el usuario decida. Este planteamiento minimiza los riesgos de omisión.

Flexibilidad y Adaptabilidad:

* Este diseño es fácilmente modificable en el futuro para incorporar técnicas concurrentes o algoritmos optimizados si el volumen de datos lo requiere.

Gráfico, Gráfico de barras

Descripción generada automáticamente

### Conceptos Relacionados

Programación Concurrente (Asignatura: Programación Concurrente)

* Se evaluó implementar tareas concurrentes para procesar cada subsistema de manera independiente. Esto habría implicado:
* Distribuir las Tareas: Cada hilo o proceso ejecutaría la búsqueda de tokens para un subsistema completo.
* Sincronización de Resultados: Combinar los resultados al finalizar todas las tareas.
* Aunque técnicamente viable, esta opción fue descartada porque:
* Introduce overhead innecesario dado el volumen actual de datos.
* La búsqueda secuencial ya satisface los requisitos actuales.

Algoritmos de Búsqueda y Estructuras de Datos (Asignatura: Algoritmia)

El diseño también se fundamenta en conceptos clave de búsqueda y optimización, aprendidos en Algoritmia:

* Búsqueda Secuencial: Elegida por su simplicidad y adecuación al tamaño reducido de las listas.
* Búsqueda Binaria e Índices: Técnicas evaluadas pero desestimadas por no justificar su complejidad en el caso actual.

### Implementación Actual

Iteración por Subsistema:

* Por cada subsistema, el algoritmo recorre su lista de tokens y los compara con el texto del documento completo.

Sugerencia de Relaciones:

* Si se encuentra al menos un token, el subsistema se incluye en las sugerencias para revisión por parte del usuario.

Modularidad:

* El diseño permite ampliar o modificar las listas de tokens y subsistemas sin necesidad de cambios estructurales en el código.

### Conclusión

El diseño actual de la búsqueda cumple con el objetivo principal de sugerir relaciones de alcance de manera flexible y eficiente. Las técnicas avanzadas, como programación concurrente o algoritmos optimizados, fueron descartadas por no justificar su complejidad dado el volumen moderado de datos y la filosofía de diseño centrada en la sencillez y la sugerencia no restrictiva.

## Diseño de Interfaz y Usabilidad.

La interfaz de usuario es un elemento esencial de este sistema, especialmente dada su aplicación en proyectos de gestión de requisitos con un alcance internacional. Se diseñó pensando en usuarios de distintas nacionalidades y en la posibilidad de gestionar documentos y requisitos en múltiples idiomas. Por ello, se incorporaron funcionalidades clave como la selección de idioma, un apartado de ajustes personalizables y una sección de ayuda integrada, todas ellas siguiendo los principios de usabilidad aprendidos en la asignatura de Interacción Hombre-Máquina (IHM).

El diseño no solo busca que los usuarios interactúen fácilmente con el sistema, sino también que puedan adaptarlo a sus necesidades específicas, facilitando la gestión de requisitos y documentos en entornos multiculturales y multilingües. Además, se priorizó una experiencia altamente intuitiva, que minimice la necesidad de explicaciones o formación adicional, gracias a una interfaz minimalista y bien estructurada.

### Sencillez e Intuición en el Diseño

Uno de los objetivos principales fue diseñar una interfaz que los usuarios puedan comprender y utilizar de manera inmediata, sin necesidad de manuales extensos ni asistencia técnica frecuente. Esto se logró mediante:

Diseño Minimalista:

* Se eliminaron elementos innecesarios para evitar sobrecargar al usuario con opciones o información superflua.
* Los menús se organizaron de forma jerárquica pero plana, mostrando solo las opciones relevantes en cada momento.

Elementos Familiares:

* Los botones, iconos y etiquetas siguen convenciones ampliamente conocidas, como el uso de un icono de engranaje para ajustes o de una lupa para búsquedas.

Feedback Continuo:

* La interfaz proporciona retroalimentación inmediata para cada acción del usuario, como mensajes de confirmación al guardar datos durante el procesamiento de documentos.

Diseño Centrado en Tareas:

* Las funcionalidades principales (carga de documentos, asignación de subsistemas, gestión de tokens) están organizadas de manera que sigan el flujo natural del trabajo del usuario.

### Gestión de Idiomas en un Entorno Internacional

En proyectos de alcance internacional, la posibilidad de trabajar con múltiples idiomas en la interfaz y en los documentos era una necesidad crítica. Para ello, se incorporaron funcionalidades específicas:

Selección de Idioma (Botón de Ajustes):

* Los usuarios pueden configurar la interfaz en su idioma preferido, independientemente del idioma en el que estén redactados los documentos.
* Esto es especialmente útil en proyectos donde equipos de diferentes países gestionan la misma documentación.
* Modularidad: Los archivos de idioma son independientes, lo que permite añadir nuevos idiomas fácilmente sin modificar el código base.

Sección de Ayuda Multilingüe (Botón de Ayuda):

* La ayuda integrada proporciona explicaciones detalladas sobre cada funcionalidad del sistema.
* Está completamente traducida a los idiomas disponibles (español, inglés y francés), y su diseño modular permite añadir más idiomas en futuras versiones.
* El contenido incluye:
* Instrucciones paso a paso para las principales funcionalidades, como cargar documentos, gestionar tokens y asignar subsistemas.
* Respuestas a preguntas frecuentes, diseñadas para resolver dudas comunes sin necesidad de soporte técnico.

Interfaz Multilingüe y Documentación Cruzada:

* La interfaz asegura que los usuarios puedan trabajar de manera fluida incluso si la documentación está en un idioma diferente al configurado en la interfaz. Por ejemplo, los tokens y términos clave de los documentos pueden mostrarse en su idioma original junto con traducciones o equivalencias.

### Decisiones Clave en el Diseño

El diseño no solo se centró en la simplicidad, sino también en garantizar que el sistema fuera accesible y adaptable:

Simplicidad sin Compromisos:

* Cada elemento de la interfaz fue diseñado para ser lo suficientemente intuitivo como para no requerir explicaciones. Por ejemplo:
* El botón de "Cargar Documento" está claramente etiquetado y accesible en todo momento.
* Los flujos de trabajo guían al usuario de manera natural, minimizando la posibilidad de errores.

Diseño Consistente y Coherente:

* La ubicación de los botones y menús es uniforme en todos los módulos, permitiendo a los usuarios generalizar rápidamente su conocimiento del sistema.

Preparación para el Futuro:

* El módulo de ajustes está diseñado para incorporar nuevas funcionalidades relacionadas con la personalización del sistema, como configuraciones de accesos directos o preferencias visuales.

### Reflexión Basada en la Teoría

Los principios aprendidos en la asignatura de Interacción Hombre-Máquina (IHM) se reflejan directamente en las decisiones tomadas para el diseño de esta interfaz:

Reconocimiento sobre Memorización:

* Los iconos y etiquetas están diseñados para ser reconocidos fácilmente sin necesidad de memorizar su función.

Eficiencia para el Usuario:

* Los flujos de trabajo siguen un diseño lógico que reduce los pasos necesarios para completar tareas comunes.

Satisfacción del Usuario:

* Una interfaz limpia, clara y bien estructurada genera una experiencia positiva, incluso para usuarios no técnicos.

### Conclusión

La interfaz combina simplicidad, modularidad y adaptabilidad, destacando funcionalidades como los botones de ajustes y ayuda, que no solo mejoran la experiencia actual del usuario, sino que también preparan el sistema para futuras expansiones. Este enfoque asegura que el software sea accesible, intuitivo y eficiente, incluso en un entorno internacional y multicultural.

## Diseño análisis de software

El diseño del sistema fue guiado por principios fundamentales de la ingeniería de software, con un enfoque claro en garantizar la modularidad, la simplicidad y la escalabilidad del proyecto. Desde las etapas iniciales, se buscó crear un software que no solo cumpliera con los requisitos funcionales actuales, sino que también pudiera evolucionar y adaptarse a futuras necesidades sin requerir una reestructuración completa.

El sistema se organiza en módulos claramente definidos, cada uno con responsabilidades específicas. Por ejemplo, las funcionalidades relacionadas con el procesamiento de texto, la gestión de datos y la detección de subsistemas están separadas para garantizar que los cambios o ampliaciones en una parte no afecten al resto del sistema. Este enfoque modular refleja la aplicación de los principios de cohesión alta y acoplamiento bajo, aprendidos en las asignaturas de Ingeniería de Software y Arquitectura de Software.

Desde el principio, el diseño estuvo marcado por la necesidad de encontrar un equilibrio entre simplicidad y flexibilidad. Dado que el sistema está pensado para ejecutarse localmente y manejar un número limitado de subsistemas y tokens, se optó por evitar arquitecturas complejas, como microservicios, y se centró en una estructura monolítica. Esta decisión no solo simplifica el desarrollo y mantenimiento, sino que también se adapta perfectamente a las necesidades de un entorno autónomo, donde no es necesaria una infraestructura distribuida.

### Decisiones Clave en el Diseño

Uno de los aspectos más importantes del diseño fue priorizar la facilidad de escalabilidad. Aunque la versión actual del sistema maneja un volumen moderado de datos (como máximo 10 subsistemas y 30 tokens por subsistema), el diseño considera posibles ampliaciones futuras. Por ejemplo, la estructura modular permite agregar nuevos subsistemas o extender las funcionalidades relacionadas con el procesamiento de texto sin comprometer la estabilidad del sistema.

Otro punto fundamental fue la separación de responsabilidades. La lógica de negocio, centrada en tareas como la búsqueda de tokens o la gestión de asociaciones, se mantiene independiente de la interfaz de usuario. Esta decisión se tomó para garantizar que el sistema sea fácilmente adaptable a otros entornos o plataformas. Por ejemplo, la misma lógica puede reutilizarse en un sistema web o móvil sin cambios significativos.

Durante el diseño, también se identificaron riesgos específicos, como el crecimiento descontrolado de tokens o subsistemas, lo que podría complicar la gestión de datos. Para abordar estos riesgos, se implementaron estrategias como el uso de una estructura jerárquica sencilla para los tokens y un diseño eficiente de la base de datos, permitiendo un control claro y predecible de la información almacenada.

### Reflexión Teórica

El diseño del software no solo responde a necesidades prácticas, sino que también aplica conceptos clave aprendidos durante la carrera. Por ejemplo:

Cohesión Alta y Acoplamiento Bajo:

Cada módulo tiene una responsabilidad específica. Este principio asegura que las partes del sistema sean independientes y fáciles de mantener. Por ejemplo, el módulo encargado de procesar documentos no tiene conocimiento de cómo se almacenan los datos, delegando esta tarea al módulo de almacenamiento.

Separación de Capas:

Aunque el sistema no sigue un patrón formal como MVC, adopta sus principios básicos. Las capas de interfaz de usuario, lógica de negocio y acceso a datos están claramente diferenciadas, lo que facilita la adaptabilidad del sistema.

Simplicidad como Principio de Diseño:

La decisión de evitar soluciones complejas, como programación concurrente o arquitecturas distribuidas, refleja el principio de diseñar para satisfacer las necesidades actuales sin sobrecargar el sistema con características innecesarias.

Lecciones Aprendidas

Este proyecto destacó la importancia de diseñar sistemas escalables y mantenibles desde las etapas iniciales. Algunas decisiones clave, como la elección de SQLite y la modularidad del sistema, fueron fundamentales para garantizar que el software sea eficiente y adaptable.

Sin embargo, este diseño también reveló retos específicos, como encontrar el equilibrio entre flexibilidad y simplicidad. Por ejemplo, aunque el sistema está preparado para ampliarse, algunas decisiones, como la elección de algoritmos secuenciales para la búsqueda de tokens, podrían revisarse en el futuro si las necesidades del sistema crecen significativamente.

### Conclusión

El diseño y análisis del software reflejan un enfoque práctico y fundamentado, donde cada decisión fue guiada por principios de ingeniería de software y por las necesidades específicas del proyecto. Aunque se priorizó la simplicidad en esta versión inicial, el sistema está preparado para crecer y adaptarse en el futuro, manteniendo siempre un diseño modular, claro y eficiente.

# Técnicas y herramientas

Textp general

## Técnicas metodológicas

Durante el desarrollo de este proyecto se utilizaron diversas metodologías para organizar y estructurar el trabajo. Estas incluyen Scrum y Kanban, complementadas por enfoques como el prototipado, el desarrollo iterativo, y un enfoque adaptativo para ajustarse a las necesidades del proyecto. A continuación, se describen cada una de estas metodologías y su implementación específica en el proyecto.

### Enumeración y Explicación de las Metodologías

#### Scrum

Descripción teórica:

Scrum es una metodología ágil basada en ciclos de trabajo cortos llamados sprints, donde se busca entregar incrementos funcionales del producto. Incluye elementos clave como:

* Backlog: Lista priorizada de tareas.
* Sprints: Ciclos iterativos de 1 a 4 semanas.
* Revisión y adaptación: Al final de cada sprint, se evalúa el progreso para planificar los siguientes pasos.

Uso en el proyecto:

En este proyecto, Scrum fue la metodología principal, siguiendo los sprints como ciclos de desarrollo de 2 semanas. Se utilizó Zube para gestionar el backlog y planificar las tareas, enfocándose en objetivos específicos como el diseño de la base de datos y la implementación de algoritmos.

#### Kanban

Descripción teórica:

Kanban es un método visual de gestión del trabajo que organiza las tareas en un tablero dividido en columnas, como "Pendiente", "En Progreso" y "Completado". Permite una entrega continua de trabajo terminado y prioriza la optimización del flujo de tareas.

Uso en el proyecto:

Kanban se integró para gestionar tareas fuera de los sprints, proporcionando flexibilidad para resolver tareas adicionales y adaptaciones necesarias. Las tareas se visualizaron en el tablero de Zube, asegurando claridad en el estado del proyecto.

#### Prototipado

Descripción teórica:

El prototipado consiste en crear versiones preliminares del sistema para probar ideas y obtener retroalimentación antes de implementar el diseño final. Este enfoque permite evaluar decisiones tempranas y realizar ajustes con bajo costo.

Uso en el proyecto:

Se desarrollaron prototipos funcionales en fases iniciales y se ampliaron iterativamente. Por ejemplo:

* Se creó un prototipo inicial para la gestión básica de documentos y la base de datos.
* En iteraciones posteriores, se añadieron funcionalidades como la búsqueda de tokens y la asociación con subsistemas.

#### Desarrollo Iterativo

Descripción teórica:

En el desarrollo iterativo, el sistema se construye por partes pequeñas y funcionales que se prueban antes de avanzar. Esto permite identificar errores o ajustes necesarios en etapas tempranas.

Uso en el proyecto:

El sistema se desarrolló iterativamente, probando cada componente antes de integrarlo al sistema completo. Por ejemplo:

* La base de datos fue diseñada y probada antes de conectar la interfaz.
* Los algoritmos de búsqueda fueron optimizados en iteraciones sucesivas para mejorar su rendimiento.

#### Enfoque Adaptativo

Descripción teórica:

Este enfoque implica ajustar el diseño y las funcionalidades del sistema según las necesidades reales detectadas durante su desarrollo. Es útil cuando ciertos requisitos no están completamente definidos al inicio.

Uso en el proyecto:

Al tratarse de un proyecto individual con detalles no completamente definidos, se realizaron ajustes sobre la marcha. Por ejemplo:

* Se optimizó la búsqueda de tokens tras evaluar el tiempo de procesamiento en documentos reales.
* La estructura jerárquica de los capítulos se ajustó para simplificar la asociación con subsistemas.

### 3. Relación entre Metodologías

La combinación de estas metodologías permitió gestionar el desarrollo del proyecto de manera eficiente. Scrum proporcionó la estructura para el trabajo iterativo, Kanban complementó con flexibilidad en la gestión de tareas adicionales, y los enfoques de prototipado, iteración y adaptación garantizaron un sistema funcional ajustado a las necesidades reales. Esta integración fue clave para alcanzar los objetivos del proyecto.

### 4. Conclusión

El uso combinado de metodologías ágiles como Scrum y Kanban, junto con enfoques complementarios como el prototipado y el desarrollo iterativo, permitió un desarrollo estructurado y flexible. Esto resultó fundamental para gestionar un proyecto con detalles no completamente definidos desde el inicio, permitiendo ajustes continuos sin perder de vista los objetivos principales.

## Patrones de diseño

El diseño del sistema se fundamentó en una arquitectura en capas, un enfoque ampliamente adoptado en el desarrollo de software debido a sus ventajas para la organización y la modularidad. La arquitectura en capas divide el sistema en tres componentes principales: Interfaz de Usuario (UI), Lógica de Negocio, y Persistencia de Datos, cada uno desempeñando un rol claramente definido.

### Arquitectura en Capas

Motivación para Elegir la Arquitectura en Capas

Este proyecto implicaba la gestión de múltiples procesos: desde la interacción con el usuario, pasando por la extracción y análisis de información, hasta el almacenamiento persistente de los datos. Para abordar esta complejidad, se optó por la arquitectura en capas debido a:

* Separación de responsabilidades: Permite desacoplar las funciones de la interfaz gráfica, la lógica de procesamiento y la base de datos.
* Escalabilidad y flexibilidad: Las capas son independientes, lo que facilita futuras ampliaciones o modificaciones sin afectar al sistema global.
* Facilidad de pruebas y mantenimiento: Cada capa puede probarse y depurarse de manera aislada.

Estructura de las Capas

1. Capa de Interfaz de Usuario (UI):

* Construida con Tkinter, esta capa es responsable de la interacción con el usuario.
* Incluye elementos como la carga de documentos, la consulta de requisitos y la asignación de subsistemas.
* Ejemplo: Un usuario selecciona un archivo PDF desde una ventana gráfica, lo que inicia un flujo de trabajo a través de las capas inferiores.

1. Capa de Lógica de Negocio:

* Gestiona las operaciones de procesamiento de documentos, la extracción de texto y la asociación de subsistemas mediante palabras clave.
* Esta capa actúa como intermediaria entre la interfaz y la base de datos, asegurando que la lógica de negocio esté aislada y no dependa de los detalles de implementación de otras capas.
* Ejemplo: Un documento cargado se analiza para extraer capítulos y requisitos, que luego se envían a la base de datos.

1. Capa de Persistencia de Datos:

* Implementada con SQLite, esta capa gestiona el almacenamiento y la recuperación de documentos, requisitos y asociaciones.
* Proporciona métodos abstractos para interactuar con los datos, lo que simplifica su manipulación sin exponer directamente las consultas SQL.
* Ejemplo: Almacenamiento de un documento junto con sus requisitos y asociaciones, garantizando la consistencia referencial.

### Ventajas de la Arquitectura en Capas

1. Modularidad

Cada capa tiene un propósito claramente definido. Esto facilita el trabajo en equipo (cada desarrollador puede enfocarse en una capa específica) y simplifica la ampliación de funcionalidades. Por ejemplo, añadir un nuevo tipo de archivo para procesar solo requiere modificaciones en la lógica de negocio.

2. Separación de Responsabilidades

* La interfaz de usuario, la lógica de negocio y la base de datos son independientes. Esto permite:
* Cambiar la interfaz gráfica (por ejemplo, reemplazar Tkinter por otro framework) sin modificar la lógica subyacente.
* Optimizar o sustituir el motor de base de datos sin afectar a las capas superiores.

3. Escalabilidad

* La estructura modular permite escalar el sistema agregando nuevas funcionalidades o ampliando la base de datos sin comprometer el diseño existente.

4. Facilidad de Mantenimiento

* La separación clara de funciones facilita la depuración de errores y el mantenimiento continuo. Si ocurre un problema en la lógica de procesamiento, puede corregirse sin alterar la interfaz gráfica.

### Patrones de Diseño Aplicados

1. Patrón en Capas

* Este patrón organiza el sistema en módulos independientes que interactúan de manera controlada. Las capas garantizan que cada parte del sistema tenga un rol específico y no interactúe directamente con partes que no sean sus vecinas inmediatas.

2. Encapsulación (Repository Simplificado)

* El acceso a la base de datos se encapsula mediante métodos como guardar\_documento o consultar\_requisitos. Esto oculta los detalles de implementación de SQLite y permite modificar las consultas SQL sin afectar otras partes del sistema.

3. Separación de Interfaz y Lógica

* Aunque no se implementó un patrón MVC formal, la lógica de negocio está claramente separada de la interfaz gráfica. Esto asegura que cualquier cambio en la GUI no afecte la lógica del sistema.

4. Modularidad y Delegación

Cada bloque del sistema tiene una tarea bien definida, como:

* cargar\_documento.py: Procesamiento de documentos.
* seleccionar\_archivo.py: Gestión de selección de archivos.
* asignar\_subsistemas.py: Asociación de subsistemas.

### Conclusión

La elección de una arquitectura en capas, junto con la implementación de patrones de diseño, ha sido clave para lograr un sistema organizado, escalable y fácil de mantener. Este enfoque permite ampliar el sistema y adaptarlo a nuevos requisitos sin comprometer su estructura actual, asegurando una experiencia de desarrollo eficiente y confiable.

## Alojamiento de repositorio

GitHub es una plataforma basada en la web para el alojamiento y gestión de código fuente. Utiliza Git como sistema de control de versiones y facilita la colaboración en proyectos de software mediante herramientas avanzadas como pull requests, issues y revisión de código.

Características principales:

* Control de versiones remoto: Permite gestionar los cambios realizados en el proyecto desde cualquier lugar con conexión a internet.
* Colaboración: Soporta equipos de desarrollo mediante herramientas de revisión y notificación de cambios.
* Seguridad y accesibilidad: Proporciona repositorios públicos y privados para garantizar la confidencialidad del código cuando sea necesario.
* Integración continua: Permite automatizar pruebas y despliegues mediante flujos de CI/CD.

Motivo de Elección

El uso de GitHub como plataforma de alojamiento fue una indicación del tutor del proyecto, dado su amplio uso en la industria del software y su integración con herramientas como Git. GitHub ofrece un entorno confiable y profesional, ideal para gestionar proyectos individuales o colaborativos.

Aunque existen otras alternativas en el mercado, como GitLab o Bitbucket, no se consideraron en este proyecto debido a la recomendación inicial y la facilidad de uso de GitHub.

## Control de versiones

Git es un sistema de control de versiones distribuido que permite a los desarrolladores rastrear y gestionar los cambios en el código fuente de un proyecto. Es ampliamente utilizado en la industria por su velocidad, flexibilidad y capacidades avanzadas de fusión y ramificación.

Características principales:

* Distribución: Cada copia local de un repositorio contiene el historial completo del proyecto, lo que permite trabajar sin conexión.
* Historial completo: Cada cambio en el código queda registrado, facilitando la trazabilidad y la posibilidad de revertir modificaciones no deseadas.
* Flujos de trabajo flexibles: Soporta distintos estilos de colaboración, como GitHub Flow o Gitflow.

Motivo de Elección

Git fue seleccionado debido a su integración nativa con GitHub, lo que simplifica la sincronización entre el repositorio local y el remoto. Es además una herramienta estándar en la industria, lo que garantiza soporte y documentación abundantes.

## Entorno de desarrollo integrado (IDE)

Visual Studio Code (VS Code) es un editor de código fuente desarrollado por Microsoft, conocido por su ligereza, flexibilidad y amplia capacidad de personalización. Es compatible con múltiples lenguajes de programación, incluyendo Python, y se integra fácilmente con herramientas de control de versiones como Git y plataformas como GitHub.

Características principales:

* Extensiones: Ofrece un marketplace con extensiones para depuración, soporte de lenguajes, integración con bases de datos y más.
* Integración con Git: Incluye una interfaz gráfica para realizar operaciones como commits, merge y push directamente desde el editor.
* Multi-plataforma: Funciona en Windows, macOS y Linux.
* Ligero y rápido: Diseñado para ser eficiente incluso en sistemas con recursos limitados.
* Personalización: Permite ajustar el entorno con configuraciones y plugins adaptados a las necesidades del proyecto.

Motivo de Elección

Al seleccionar un entorno de desarrollo integrado (IDE) para este proyecto, se evaluaron las siguientes necesidades:

* Compatibilidad con Python: El IDE debía proporcionar un soporte sólido para Python, incluyendo autocompletado, resaltado de sintaxis y herramientas de depuración.
* Integración con GitHub: Era esencial que el IDE ofreciera soporte nativo para trabajar con Git y sincronizar cambios con GitHub de forma eficiente.
* Ligereza: Se requería un entorno que no fuera pesado ni complicado de instalar, dado que el proyecto se desarrolla en un equipo con recursos limitados.

Visual Studio Code fue elegido por cumplir con todas estas necesidades:

* Proporciona un excelente soporte para Python mediante extensiones como Python Extension Pack.
* Ofrece integración nativa con Git y GitHub, lo que facilita el flujo de trabajo del proyecto.
* Su diseño ligero lo hace rápido y fácil de configurar.

Alternativas Consideradas

Aunque se eligió Visual Studio Code, también existen otras opciones viables para el desarrollo en Python:

* PyCharm (Community Edition): Una opción robusta con herramientas avanzadas para Python, pero más pesada en comparación con VS Code.
* Sublime Text: Un editor ligero y rápido, pero con capacidades limitadas para la integración con GitHub.
* Atom: Similar a VS Code, pero con menor soporte de extensiones y actualizaciones más lentas.

Sin embargo, ninguna de estas alternativas cumplía con todos los requisitos de ligereza, integración con GitHub y soporte para Python de manera tan eficiente como Visual Studio Code.

Flujo de Trabajo con Visual Studio Code

El flujo de trabajo en este proyecto sigue estos pasos:

* Desarrollo en local: Se utiliza Visual Studio Code para editar y probar el código en un entorno controlado.
* Gestión de cambios: La extensión de control de versiones detecta automáticamente las modificaciones realizadas.
* Commits y sincronización: Una vez aprobados los cambios, se realiza un commit local y se sincronizan con el repositorio remoto en GitHub.

## Gestión del proyecto

Zube es una herramienta de gestión de proyectos diseñada para integrarse con GitHub y otros sistemas de control de versiones. Permite organizar tareas, gestionar flujos de trabajo y realizar un seguimiento del progreso del proyecto mediante tableros Kanban y otros elementos visuales.

Características principales:

* Integración con GitHub: Permite sincronizar issues, pull requests y commits directamente desde el repositorio.
* Gestión visual: Ofrece tableros Kanban que facilitan la organización de tareas y su seguimiento.
* Colaboración: Aunque este proyecto es individual, Zube admite la asignación de tareas a miembros del equipo y proporciona herramientas para la comunicación interna.
* Priorización: Permite categorizar y priorizar tareas según su importancia y urgencia.

Motivo de Elección

El uso de Zube fue recomendado por el tutor del proyecto debido a su integración fluida con GitHub y su enfoque visual para la gestión de tareas. Esta herramienta simplifica la organización y el control del progreso, especialmente en proyectos que utilizan Git como sistema de control de versiones.

En este caso, Zube se utiliza para:

* Planificar tareas: Las funcionalidades de Kanban permiten dividir el proyecto en tareas manejables, organizándolas en columnas como "Por Hacer", "En Progreso" y "Completado".
* Realizar seguimiento: Cada tarea está vinculada a issues o commits en GitHub, lo que garantiza que el progreso esté documentado y actualizado.
* Gestión de prioridades: Las tareas se pueden clasificar según su prioridad, ayudando a identificar cuáles requieren atención inmediata.

Alternativas Consideradas

Aunque no se evaluaron otras herramientas directamente debido a la recomendación del tutor, existen opciones que podrían haberse utilizado para la gestión de este proyecto, como:

* Trello: Una herramienta basada en tableros Kanban, pero sin integración directa con GitHub.
* Asana: Orientada a la gestión de proyectos en equipo, con soporte limitado para integración con Git.
* GitHub Projects: Una solución integrada en GitHub que permite crear tableros y gestionar issues, pero con menos funcionalidades avanzadas que Zube.

Ventajas de Usar Zube

* Integración directa con GitHub: Evita duplicar información entre el sistema de gestión y el repositorio de código.
* Organización visual: Facilita el seguimiento del progreso del proyecto de forma clara y estructurada.
* Adaptabilidad: Aunque el desarrollo del proyecto es individual, la herramienta está preparada para gestionar equipos, lo que permite su escalabilidad en futuros proyectos colaborativos.

Conclusión

Zube ha sido una herramienta clave para la planificación y el seguimiento del progreso en este proyecto, proporcionando un entorno organizado y vinculado directamente con el repositorio de código. Su uso ha permitido dividir el desarrollo en tareas claras y manejables, asegurando que cada etapa del proyecto se lleve a cabo de manera eficiente.

## Comunicación

Aunque el desarrollo de este proyecto se ha llevado a cabo de forma individual, en un entorno colaborativo sería necesario establecer canales de comunicación efectivos para coordinar tareas, resolver problemas y compartir información entre los miembros del equipo. A continuación, se describen herramientas y estrategias que podrían emplearse para garantizar una comunicación eficiente:

Heramientas:

* Microsoft Teams:
* Integra mensajería instantánea, videollamadas y almacenamiento de archivos en un solo entorno.
* Ofrece integración con herramientas como GitHub y Visual Studio Code, permitiendo coordinar tareas técnicas.
* Slack:

Una plataforma de mensajería instantánea ampliamente utilizada en entornos de trabajo.

Permite crear canales específicos para proyectos o temas, además de integrar aplicaciones como GitHub, Zube o Google Drive.

Ideal para comunicación en tiempo real y gestión de notificaciones relacionadas con el proyecto.

* Emails (Gmail, Outlook, etc.):

Útil para comunicaciones más formales o envío de documentación.

Aunque menos dinámico que otras opciones, sigue siendo una herramienta importante para coordinar entregas y recibir actualizaciones.

* Google Meet / Zoom:

Herramientas para videoconferencias en tiempo real, ideales para reuniones de equipo, presentaciones o resolución de problemas complejos.

Conclusión

Aunque este proyecto no requirió comunicación colaborativa, herramientas como Slack, Microsoft Teams o Discord serían opciones ideales para un equipo de desarrollo. Estas herramientas, combinadas con estrategias como reuniones periódicas y documentación centralizada, garantizarían una comunicación efectiva y la coordinación adecuada entre los miembros del equipo.

## Documentación de memoria

La elaboración de la memoria del proyecto requirió elegir una herramienta adecuada para redactar y estructurar la documentación. Se consideraron dos opciones principales: LaTeX y Microsoft Word, evaluando sus características, ventajas y desventajas. Aunque ambas herramientas son ampliamente utilizadas en contextos académicos y técnicos, la decisión final se basó en el nivel de experiencia personal y la eficiencia requerida para completar el trabajo dentro de los plazos establecidos.

LaTex

LaTeX es un sistema de preparación de documentos basado en texto plano que permite a los usuarios diseñar documentos técnicos y científicos con un alto nivel de precisión tipográfica.

* Características principales:
* Control total sobre el formato: Ideal para documentos con fórmulas matemáticas, bibliografías extensas o referencias cruzadas complejas.
* Automatización de índices y bibliografías: Genera automáticamente listas de tablas, figuras e índices.
* Plantillas predefinidas: Facilita el diseño de documentos siguiendo estándares académicos o corporativos.
* Requiere conocimientos de programación: Los documentos se escriben en un lenguaje de marcado, lo que implica aprender comandos específicos.
* Ventajas:
* Genera documentos con una calidad tipográfica excepcional.
* Es ideal para proyectos colaborativos, ya que los archivos de texto plano pueden integrarse fácilmente con herramientas como Git.
* Soporta grandes documentos técnicos de manera eficiente.
* Desventajas:
* Curva de aprendizaje elevada para nuevos usuarios.
* Los cambios de diseño pueden ser complicados si no se tiene experiencia.
* El proceso de compilación (de texto plano a PDF) requiere herramientas adicionales.

Word

Microsoft Word es un procesador de texto ampliamente conocido y utilizado, diseñado para la creación de documentos con una interfaz gráfica intuitiva.

* Características principales:
* Edición en tiempo real: Permite realizar cambios visuales en el documento directamente desde la interfaz.
* Manejo de referencias y control de cambios: Incluye herramientas para crear índices, gestionar bibliografías y realizar un seguimiento de las modificaciones.
* Compatibilidad: Exporta documentos a una amplia variedad de formatos, incluyendo PDF.
* Ventajas:
* Curva de aprendizaje baja, con una interfaz intuitiva que facilita la edición.
* Amplio soporte para plantillas, gráficos, tablas e imágenes.
* Muy útil para documentos técnicos que requieren control de cambios, anotaciones y revisiones.
* Desventajas:
* La consistencia del formato puede ser un desafío en documentos muy largos.
* Menos automatización que LaTeX en bibliografías y referencias cruzadas.
* Dependencia de un software propietario.

Decisión Final

La decisión de utilizar Microsoft Word para la redacción de la memoria se tomó basándome en:

* Nivel de experiencia: Tengo un nivel experto en el manejo de Word, adquirido a través de mi experiencia profesional en la adecuación de documentación técnica de ingeniería. Esto incluye habilidades avanzadas como control de cambios, referencias cruzadas, índices automáticos y manejo de plantillas complejas.
* Plazos ajustados: Aprender a utilizar LaTeX hubiera requerido un tiempo adicional que no era compatible con los tiempos establecidos para la entrega del proyecto.
* Entrega en formato PDF: Aunque la memoria se entrega en formato PDF, Word permite exportar directamente el documento en este formato sin comprometer el diseño ni la estructura.
* Alternativas Consideradas
* Si hubiera tenido experiencia previa en LaTeX, esta herramienta podría haber sido una opción válida debido a:
* Su capacidad para manejar documentos extensos con muchas referencias.
* Su compatibilidad con estándares académicos.
* Sin embargo, dado mi nivel experto en Microsoft Word y la necesidad de optimizar el tiempo, esta fue la herramienta más adecuada para cumplir con los objetivos del proyecto.

## Documentación del código

La documentación del código es un elemento esencial para facilitar la comprensión del proyecto, su mantenimiento y posible ampliación. Este apartado describe las herramientas consideradas para documentar el código y los motivos que llevaron a la selección final.

Opciones Valoradas

Durante el análisis de alternativas para la generación de la documentación del código, se consideraron las siguientes herramientas:

1. Sphinx:

* Ventajas:
* Integración nativa con Python, lo que facilita el análisis de docstrings.
* Capacidad de generar documentación en múltiples formatos (HTML, PDF, etc.).
* Amplia comunidad de soporte y compatibilidad con plataformas como Read the Docs.
* Desventajas:
* Requiere una curva de aprendizaje inicial para configurar correctamente.
* Dependencia de herramientas externas para generar el contenido.

1. Doxygen:

* Ventajas:
* Admite múltiples lenguajes de programación, lo que podría ser útil si en el futuro se incorporaran otras tecnologías.
* Generación de gráficos y diagramas para representar las relaciones entre clases y módulos.
* Desventajas:

Su uso no está tan extendido en proyectos exclusivamente en Python.

Necesidad de configuración adicional para trabajar con docstrings de Python.

1. MkDocs:

* Ventajas:
* Interfaz intuitiva y documentación clara.
* Ideal para crear documentación orientada a usuarios finales más que a desarrolladores.
* Desventajas:
* Menor soporte para documentación técnica directamente extraída del código.
* Requiere mantener archivos de documentación separados del código fuente.

Decisión Final

Se optó por utilizar Sphinx como herramienta principal para la documentación del código debido a su capacidad de extraer directamente información desde los docstrings del código Python, su flexibilidad para personalizar el diseño y su compatibilidad con plataformas de despliegue como Read the Docs.

La integración de Sphinx permite centralizar la documentación técnica y mantenerla actualizada conforme se realizan modificaciones en el código, asegurando coherencia entre el contenido generado y el funcionamiento del proyecto.

Proceso de Documentación

El proceso implementado incluye los siguientes pasos:

* Estandarización de Docstrings: Se han comentado todas las clases, funciones y módulos siguiendo los estándares de Python.
* Configuración de Sphinx: Se ha inicializado un proyecto de Sphinx para automatizar la creación de documentación.
* Generación de Documentación: Usando comandos como sphinx-apidoc y make html, se generó documentación en formato HTML para consulta local y futura distribución.

La decisión de usar Sphinx asegura que la documentación del código esté integrada de manera eficiente con el flujo de trabajo del proyecto, simplificando futuras revisiones y ampliaciones.s.

## Integración y despliegue continuos (CI/CD)

En el desarrollo de este proyecto, el flujo de trabajo ha sido adaptado a mis necesidades como desarrolladora única. Actualmente, sigo un proceso manual de verificación antes de subir los cambios al repositorio de GitHub, lo que asegura que únicamente se almacene código funcional. Esta metodología, aunque sencilla, resulta eficiente en un entorno individual y ha sido suficiente para garantizar la estabilidad del proyecto.

Flujo de trabajo actual

1. Desarrollo y pruebas locales:

Cada nueva funcionalidad se desarrolla y prueba en mi máquina. Si el programa funciona correctamente, considero el cambio listo para su inclusión en el repositorio.

1. Gestión de versiones en GitHub:

Uso Git para registrar los cambios (commits) y subo al repositorio únicamente versiones estables y funcionales. Este método asegura que la rama principal (main) siempre contiene un código limpio y operativo.

1. Evitar errores:

Al no usar un sistema de Integración Continua (CI), los errores potenciales que podrían detectarse en un servidor remoto se eliminan antes de subir el código. Esto es posible gracias a un flujo de trabajo manual y meticuloso.

Limitaciones del enfoque actual

* Sin automatización de pruebas en múltiples entornos:
* Las pruebas solo se realizan en mi máquina local, por lo que el funcionamiento en otros entornos (diferentes sistemas operativos o configuraciones) no se garantiza automáticamente.
* Escalabilidad limitada:
* Si en el futuro el proyecto crece o requiere colaboración de más desarrolladores, este flujo manual podría no ser suficiente para garantizar la calidad y consistencia del código.

Integración y Despliegue Continuos para futuros proyectos

Aunque no he implementado sistemas de Integración Continua (CI) ni Despliegue Continuo (CD) en este proyecto, entiendo su importancia para entornos colaborativos o proyectos más grandes. Estas herramientas ofrecen ventajas clave:

* Pruebas automáticas:
* Ejecutan pruebas unitarias y de integración cada vez que se suben cambios al repositorio.
* Automatización de despliegues:
* Facilitan la publicación de nuevas versiones en entornos de producción o preproducción.
* Protección de ramas:
* Bloquean la inclusión de código defectuoso en la rama principal, garantizando que todo lo integrado ha pasado pruebas automáticas.

Entre las herramientas populares para CI/CD están GitHub Actions, Jenkins, y GitLab CI/CD, las cuales podrían ser útiles en proyectos colaborativos o con requisitos más complejos.

Conclusión

Mi flujo actual es adecuado para un desarrollo individual y me permite mantener el control manual sobre el proceso de desarrollo y pruebas. Sin embargo, para futuros proyectos o colaboraciones, consideraré la implementación de CI/CD para aprovechar la automatización y escalabilidad que ofrecen estas herramientas.

## Calidad y consistencia del código

Garantizar la calidad y consistencia del código es fundamental en el desarrollo de cualquier proyecto. En este trabajo, se han empleado diversas herramientas para analizar y mejorar el código fuente, asegurando que cumple con estándares de desarrollo modernos. Estas herramientas se seleccionaron basándose en su eficacia para detectar errores, optimizar el estilo y mantener las mejores prácticas en Python.

Herramientas utilizadas

1. SonarCloud

* Plataforma en la nube para análisis estático del código, diseñada para evaluar aspectos como mantenibilidad, duplicación, complejidad, seguridad y buenas prácticas.
* Por qué se usó:
* Integración sencilla con GitHub, permitiendo análisis automáticos en cada commit.
* Reportes visuales que facilitan identificar y priorizar mejoras en el código.
* Resultados obtenidos:
* Identificación de áreas clave de mejora relacionadas con duplicación y complejidad.
* Aseguró que los cambios realizados no introdujeran nuevos errores.

1. Pylint

* Una herramienta para el análisis estático del código en Python, diseñada para evaluar problemas de estilo y adherencia a las normas PEP8.
* Por qué se usó:
* Proporciona una calificación cuantitativa que refleja la calidad del código, facilitando la identificación de áreas de mejora.
* Detecta errores comunes, como variables no utilizadas o malas prácticas.
* Resultados obtenidos:
* Se logró una mejora en los módulos analizados, corrigiendo problemas de estilo y eliminando importaciones innecesarias.

1. Black

* Una herramienta para el formateo automático de código Python, que asegura una escritura uniforme y consistente en todo el proyecto.
* Por qué se usó:
* Ayuda a eliminar inconsistencias en el estilo de código.
* Permite centrarse en la funcionalidad en lugar de detalles estéticos durante las revisiones.
* Resultados obtenidos:
* Reformateo de todo el proyecto para garantizar una presentación uniforme y clara.

Otras herramientas consideradas

Aunque no todas las herramientas fueron utilizadas, se valoraron varias opciones que podrían haber contribuido al proceso de análisis del código. Entre ellas:

* Flake8
* Una herramienta combinada para análisis de estilo y errores en Python, que incluye funcionalidades similares a Pylint.
* Aunque útil, se prefirió Pylint por su capacidad de realizar análisis más detallados y proporcionar puntuaciones.
* DeepSource
* Una alternativa a SonarCloud para análisis de código estático, con soporte para múltiples lenguajes.
* SonarCloud ofrecía mayor integración con GitHub y métricas específicas para Python que resultaban más relevantes para el proyecto.
* Prettier
* Una herramienta de formateo automático similar a Black, pero más orientada a JavaScript y otros lenguajes.
* Black es específica para Python y se integra de forma más natural con el flujo de trabajo del proyecto.
* Bandit
* Una herramienta especializada en detectar problemas de seguridad en código Python.
* No se consideró necesario en esta versión del proyecto, ya que no se manejan datos sensibles ni operaciones críticas que justifiquen un análisis de seguridad detallado.

Decisión tecnológica y balance de herramientas

La combinación de SonarCloud, Pylint y Black permitió cubrir diferentes aspectos de la calidad del código:

* SonarCloud proporcionó una visión general del estado del proyecto y de áreas clave a mejorar.
* Pylint ayudó a refinar los detalles en el cumplimiento de estándares de Python.
* Black garantizó la uniformidad en el estilo, facilitando tanto la colaboración como el mantenimiento futuro.

Cada herramienta aportó un valor único al proceso de desarrollo, optimizando tanto la calidad técnica como la presentación del código.

Captura de pantalla de computadora

Descripción generada automáticamente con confianza media

Imagen que contiene Texto

Descripción generada automáticamente

## Cobertura de código

La cobertura de código es un concepto esencial en el desarrollo de software, ya que permite medir qué porcentaje del código ha sido ejecutado durante las pruebas. Este indicador es clave para identificar áreas del programa que no han sido verificadas y pueden contener errores.

Estado Actual en el Proyecto

En el marco de este proyecto, no se han implementado pruebas automatizadas ni herramientas de cobertura de código. Todas las verificaciones se han realizado mediante pruebas manuales, ejecutando el software y comprobando sus resultados en distintos escenarios.

Este enfoque ha sido suficiente para asegurar la funcionalidad básica del sistema, pero tiene limitaciones significativas:

* Es difícil detectar errores en rutas de código menos utilizadas.
* No garantiza una cobertura exhaustiva de todas las funcionalidades.
* Dificulta el mantenimiento y la escalabilidad del software.

Enfoque de Aprendizaje

Dado que no tengo experiencia previa en el diseño e implementación de pruebas automatizadas, este apartado representa una oportunidad de mejora significativa para futuras versiones del proyecto. Sería necesario comenzar por:

* Estudio de Frameworks de Pruebas: Investigar herramientas como unittest o pytest, que son estándar en el ecosistema Python, y aprender a implementarlas en pequeñas funciones.
* Creación de Pruebas Básicas: Iniciar con pruebas simples que verifiquen las funcionalidades clave, como la carga de documentos o las consultas en la base de datos.
* Integración de Cobertura de Código: Utilizar herramientas como Coverage.py para analizar qué partes del código están siendo probadas y cuáles no.

Futuras Aplicaciones

Si el proyecto se amplía o evoluciona, implementar pruebas automatizadas ofrecería beneficios importantes:

* Detectar errores de manera temprana, especialmente en funciones críticas.
* Facilitar el mantenimiento del software, asegurando que nuevas implementaciones no rompan funcionalidades existentes.
* Mejorar la calidad general del código y la confianza en el sistema.

Aunque en esta versión del proyecto no se han integrado pruebas automatizadas, he identificado este aspecto como un área de aprendizaje y mejora futura, alineada con los estándares profesionales del desarrollo de software.

## Framework escritorio

Para el desarrollo de la interfaz gráfica del proyecto, se optó por utilizar Tkinter, el framework de escritorio nativo de Python. Esta elección estuvo motivada por varios factores:

* Simplicidad y Accesibilidad: Tkinter es un framework integrado en Python, lo que elimina la necesidad de instalar dependencias externas y permite comenzar rápidamente con el desarrollo de interfaces gráficas.
* Amplia Documentación: Existe una gran cantidad de documentación, tutoriales y ejemplos disponibles, lo que facilita el aprendizaje y la resolución de problemas.
* Requisitos del Proyecto: Tkinter es suficiente para cubrir las necesidades básicas de este proyecto, que incluyen cargar documentos, consultar información y mostrar resultados en una interfaz amigable.

Alternativas Consideradas

Aunque se seleccionó Tkinter, también se valoraron otras opciones para el desarrollo de la interfaz de escritorio:

* PyQt/PySide: Frameworks más avanzados y potentes que ofrecen soporte para diseños más complejos. Sin embargo, se descartaron debido a:
* Mayor curva de aprendizaje.
* Complejidad innecesaria para un proyecto individual y con funcionalidades simples.
* Licencias que, en algunos casos, pueden ser restrictivas para su uso gratuito.
* Kivy: Orientado a aplicaciones multiplataforma (escritorio y móvil). Aunque es interesante para proyectos futuros, no era relevante para este caso específico.
* WxPython: Una opción menos utilizada que Tkinter, que ofrece diseño más moderno pero requiere más configuración.

Uso de Tkinter en el Proyecto

Tkinter se utilizó para desarrollar una interfaz gráfica sencilla pero funcional, que incluye:

* Carga de Documentos: Un área de interacción donde el usuario puede seleccionar un archivo PDF y asociarlo a un proyecto.
* Consulta y Visualización: Marcos divididos que permiten mostrar resultados de consultas y asociaciones.
* Gestión de Idiomas: Implementación de un selector de idioma para adaptar la interfaz según las necesidades del usuario.

La estructura del código se organizó en módulos que representan bloques de funcionalidad, asegurando un diseño limpio y fácil de mantener. Las ventanas y cuadros de diálogo fueron diseñados para proporcionar una experiencia de usuario intuitiva y clara.

* Ventajas:
* Integrado en Python.
* Fácil de aprender y utilizar.
* Ideal para proyectos pequeños y medianos.
* Desventajas:
* Menor flexibilidad para interfaces complejas.
* Apariencia básica en comparación con otros frameworks modernos.

Reflexión

Tkinter ha demostrado ser una solución adecuada para los requisitos del proyecto. Su simplicidad permitió concentrarse en la lógica del sistema en lugar de en el diseño de interfaces

## Sistema gestor de bases de datos

Para el almacenamiento y gestión de los datos del proyecto, se decidió utilizar SQLite, un sistema de gestión de bases de datos relacional ligero y embebido. La elección de SQLite estuvo fundamentada en los siguientes aspectos:

* Simplicidad y Portabilidad:
* SQLite no requiere un servidor independiente, lo que simplifica su configuración y uso.
* Los datos se almacenan en un archivo local, facilitando la portabilidad del proyecto.
* Adecuado para el Volumen de Datos:
* Dado que el sistema está diseñado para un entorno de escritorio y el volumen de datos esperado es moderado, SQLite ofrece un rendimiento más que suficiente.
* Facilidad de Integración:
* La biblioteca sqlite3 está incluida por defecto en Python, eliminando la necesidad de instalar dependencias adicionales.
* Requisitos del Proyecto:
* La gestión de requisitos, documentos y asociaciones se beneficia de la estructura relacional que ofrece SQLite.

Alternativas Consideradas

Aunque SQLite fue la opción seleccionada, se valoraron otras alternativas como:

* PostgreSQL:
* Ventajas: Escalabilidad, soporte para operaciones avanzadas y manejo de grandes volúmenes de datos.
* Descartada por ser más compleja de configurar y por exceder los requerimientos de un sistema local.
* MySQL:
* Ventajas: Amplio uso en aplicaciones empresariales.
* Descartada debido a la necesidad de un servidor dedicado y la menor portabilidad en un entorno local.
* Bases de Datos NoSQL (como MongoDB):
* Ventajas: Flexibilidad en la estructura de datos y escalabilidad.
* Descartada porque los datos del proyecto son relacionales y no requerían la flexibilidad de un esquema no estructurado.

Implementación en el Proyecto

La base de datos fue diseñada con las siguientes tablas principales:

* Proyectos: Para gestionar los proyectos asociados a los requisitos.
* Documentos: Almacena información sobre los documentos cargados.
* Requisitos: Contiene los requisitos extraídos de los documentos.
* Subsistemas: Lista de subsistemas disponibles.
* Asociaciones: Relación entre documentos y subsistemas.

El esquema de la base de datos fue normalizado hasta la Tercera Forma Normal (3FN) para garantizar la consistencia de los datos y reducir redundancias.

Ventajas y Desventajas de SQLite

* Ventajas:
* Instalación y configuración mínimas.
* Portabilidad extrema: toda la base de datos reside en un único archivo.
* Alto rendimiento para proyectos pequeños y medianos.
* Desventajas:
* No es adecuada para sistemas distribuidos con múltiples usuarios concurrentes.
* Carece de algunas funcionalidades avanzadas como triggers complejos o soporte para usuarios y roles.

Reflexión

SQLite ha demostrado ser una opción ideal para las necesidades actuales del proyecto, permitiendo un desarrollo rápido y eficiente. Aunque cumple perfectamente con los requisitos actuales, su naturaleza local y monousuario podría ser una limitación en una futura versión del sistema con mayores exigencias de concurrencia o escalabilidad.

## Bibliotecas y librerías relevantes

En el desarrollo de este proyecto, se han empleado diversas bibliotecas y librerías que han facilitado la implementación tanto a nivel de procesamiento de datos como en la creación de la interfaz gráfica y la gestión de la base de datos

A continuación, se presenta una lista de las bibliotecas utilizadas, junto con una breve descripción de su propósito principal:

1. os: Manejo de rutas y directorios en el sistema operativo.
2. sys: Ajuste de rutas para importar módulos personalizados.
3. json: Lectura y escritura de archivos de configuración y traducciones.
4. csv: Procesamiento de archivos de tokens en formato CSV.
5. ast: Conversión de cadenas en estructuras Python (listas y diccionarios).
6. sqlite3: Gestión de la base de datos SQLite.
7. logging: Registro y seguimiento de eventos durante la ejecución.
8. re: Uso de expresiones regulares para extracción y manipulación de texto.
9. Tkinter: Creación de la interfaz gráfica de usuario (ventanas y widgets).
10. Pillow: Manipulación de imágenes en la interfaz gráfica.
11. pdfplumber: Extracción de texto de documentos PDF.
12. pandas: Procesamiento y análisis de datos tabulares.
13. Pylint: Evaluación de la calidad del código.
14. Black: Formateo automático del código para uniformidad.
15. SonarQube/SonarCloud: Análisis avanzado de calidad y detección de vulnerabilidades en el código.
16. Sphinx: Generación automatizada de documentación técnica.
17. XlsxWriter: Creación y exportación de datos a hojas de cálculo Excel.
18. Maker: Gestión y manipulación de tokens específicos del proyecto.

## Otras herramientas

Mendeley [33] es una herramienta de gestión de referecnias bibliográficas y colaboración académica. Entre sus características cabe destacar la capacidad de organizar las referencias y exportarlas a un archivo que se puede utilizar desde Word

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Tabla 4.1 Tabla modelo

# Aspectos relevantes del desarrollo del proyecto

El desarrollo de este proyecto no puede entenderse sin considerar el contexto personal, profesional y académico en el que se enmarca. Cada decisión, cada metodología aplicada y cada reto superado están profundamente ligados a mi experiencia, mis conocimientos y las circunstancias concretas que rodean la realización de este Trabajo de Fin de Grado.

A lo largo de este apartado, se desarrollan los pilares fundamentales del proyecto: aspectos clave que no solo han definido el alcance y enfoque del trabajo, sino que también han sido determinantes en su éxito. Estos pilares son un reflejo de quién soy y del contexto en el que se desarrolla este TFG, demostrando cómo las experiencias personales y profesionales pueden integrarse en un proyecto académico para aportar valor práctico y aplicable.

## Gestión de Documentación Técnica en Proyectos Ferroviarios: Primer Pilar del Desarrollo

Contexto: Proyectos Ferroviarios Internacionales

El tema del TFG surge directamente de una necesidad identificada en mi experiencia profesional dentro de una ingeniería ferroviaria que gestiona proyectos internacionales llave en mano. Estos proyectos abarcan múltiples fases, desde la licitación inicial hasta la construcción y el mantenimiento de líneas ferroviarias, e implican la integración de diversos subsistemas: energía, catenaria, obra civil, comunicaciones, ciberseguridad, señalización, entre otros.

La correcta gestión de requisitos en este contexto es crucial. El cliente entrega una enorme cantidad de documentación técnica, organizada en volúmenes, secciones, anexos y revisiones. Estos documentos, fundamentales para comprender las expectativas y requisitos del proyecto, presentan desafíos significativos por su diversidad y volumen. Cada entrega puede incluir:

* Documentos normativos y especificaciones técnicas.
* Planos y anexos técnicos.
* Información relevante para subsistemas específicos.

El volumen de documentación puede superar los 1000 archivos por proyecto, con estructuras y formatos variados. Esto hace que la gestión eficiente de estos documentos sea un desafío recurrente.

### Entrega y Gestión de Documentación Técnica

La entrega de documentación se realiza en dos fases principales:

* Fase de licitación: Durante la preparación de ofertas, donde las empresas compiten en una licitación para ganar un proyecto.
* Fase de ejecución: Una vez adjudicado el proyecto, cuando se deben cumplir los requisitos establecidos por el cliente.

Cada entrega es única, ya que los clientes estructuran la información de manera diferente. Esto genera complejidades adicionales, como:

* Duplicidad de contenido: Documentos similares o idénticos en diferentes versiones.
* Información dispersa: Requisitos distribuidos en varios documentos, dificultando su localización.
* Falta de estandarización: Cada cliente presenta la documentación de forma diferente, complicando la interpretación.

En este contexto, el tiempo juega un papel crítico. Tanto en la fase de licitación como en la ejecución, los plazos suelen ser ajustados, lo que dificulta una revisión exhaustiva de los documentos. Para los subsistemas, esto significa que no pueden perder tiempo buscando manualmente entre miles de páginas para identificar información relevante.

### Problema Operativo: Bucear en la Documentación

El enfoque manual e intuitivo para gestionar esta documentación genera varios problemas operativos:

* Requisitos no atendidos: Requisitos importantes pueden quedar sin asignar o pasar inadvertidos.
* Ambigüedad en responsabilidades: La falta de claridad sobre qué subsistema debe atender un requisito genera conflictos y retrasos.
* Pérdida de tiempo: Revisar manualmente documentos consume tiempo valioso que podría destinarse a tareas más críticas.
* Errores en la interpretación: La organización inconsistente de los documentos puede llevar a malentendidos o a omitir detalles importantes.

### Impacto de una Mala Gestión de Requisitos

Una gestión deficiente de los requisitos tiene implicaciones significativas en proyectos ferroviarios:

Tiempo:

* Los retrasos en la asignación y resolución de requisitos afectan directamente el cronograma del proyecto.

Coste económico:

* Los errores en la gestión de requisitos generan sobrecostes, derivados de correcciones en fases avanzadas del proyecto.

Calidad:

* La falta de trazabilidad afecta la calidad del diseño y construcción, comprometiendo aspectos críticos como la seguridad o funcionalidad.

Responsabilidad contractual:

* Los incumplimientos pueden derivar en penalizaciones contractuales o pérdida de confianza del cliente.

### Solución Propuesta: Herramienta Especializada

El desarrollo de este software responde directamente a estos problemas. La herramienta propuesta tiene como objetivos principales:

* Optimización de la gestión documental: Clasifica y organiza los documentos, facilitando su acceso.
* Trazabilidad de requisitos: Garantiza un seguimiento claro de cada requisito desde su identificación hasta su resolución.
* Reducción de tiempos de análisis: Permite búsquedas rápidas y asignaciones automáticas de requisitos.
* Garantía de calidad: Minimiza errores humanos en la asignación y gestión de requisitos.

### Justificación del Proyecto

Este proyecto no solo aborda un problema técnico, sino que también responde a una necesidad operativa real en proyectos de ingeniería ferroviaria. La experiencia adquirida en mi entorno profesional ha sido clave para identificar esta problemática y diseñar una solución alineada con las necesidades del sector.

El software no se limita a gestionar documentos PDF, sino que integra funcionalidades críticas como:

* Clasificación de requisitos: Ayuda a identificar qué subsistema debe encargarse de cada parte de la documentación.
* Facilitación de la trazabilidad: Permite rastrear cada requisito desde su identificación hasta su resolución final.
* Optimización de recursos: Reduce tiempos y costes asociados a la revisión manual de documentos.

### Diseño desde una Perspectiva Global

El proyecto ha sido concebido como un sistema integral que aborda múltiples dimensiones:

* Impacto en el ciclo de vida del proyecto ferroviario: Desde la licitación hasta el mantenimiento.
* Reducción de riesgos y costes: Identificación temprana de conflictos y errores.
* Colaboración entre equipos: Mejora la comunicación y la eficiencia operativa.

## Gestión de tiempo y plazos de entrega

La gestión del tiempo ha sido un pilar fundamental en este proyecto, condicionado por la necesidad de desarrollar una versión inicial funcional del software y cumplir con los requisitos mínimos de documentación en un plazo extremadamente ajustado. Este enfoque permitió garantizar que el TFG estuviera listo para su entrega en enero de 2025, cumpliendo con las fechas de defensa establecidas por el calendario académico.

### La Importancia de una Versión Inicial Funcional

Desde el inicio, la prioridad ha sido alcanzar una solución mínima viable (MVP) que cumpliera con los objetivos básicos del proyecto. Esta decisión estratégica respondió a dos factores clave:

* Restricción de tiempo: Solo se disponía de 16 semanas para completar el trabajo, incluyendo el desarrollo técnico y la creación de toda la documentación asociada.
* Cumplimiento de plazos académicos: La entrega del TFG debía garantizarse dentro de los tiempos requeridos por la universidad, lo que hacía imprescindible tener una versión funcional que abordara la problemática planteada.

El enfoque MVP permitió centrar los esfuerzos en las funcionalidades esenciales, dejando posibles mejoras y optimizaciones para futuras iteraciones. Este esquema también implicó la creación de una estrategia de milestones que dividiera el trabajo en fases específicas y alcanzables.

### Desafíos del Tiempo Reducido

El tiempo limitado influyó en múltiples aspectos del desarrollo:

* Priorización estricta: Se descartaron funcionalidades adicionales y tareas no críticas para enfocarse únicamente en las necesidades básicas.
* Planificación intensiva: La falta de margen para errores o retrasos exigió una planificación precisa y la ejecución de tareas en paralelo siempre que fue posible.
* Entregables mínimos: En algunos casos, se priorizó el cumplimiento de los requerimientos mínimos sobre la perfección técnica o estética.

### Comparativa con un Contexto Profesional

En un entorno profesional, el desarrollo de un software con características similares podría haberse completado en un plazo más corto debido a la disponibilidad de recursos especializados.

|  |  |  |
| --- | --- | --- |
| **Aspecto** | **TFG Individual** | **Proyecto Profesional** |
| **Duración total** | 16 semanas | 8 semanas |
| **Recursos humanos** | 1 persona | Equipo de 6 especialistas |
| **Distribución del trabajo** | Todas las tareas realizadas por una persona. | División entre diseño, desarrollo, documentación y pruebas. |
| **Iteraciones y mejoras** | Limitadas por tiempo | Amplio margen para optimizaciones |

Esta comparación pone en perspectiva las limitaciones de un proyecto individual en un marco académico frente a las capacidades de un equipo profesional.

### Milestones y Planificación Estratégica

La planificación estratégica fue crucial para avanzar de manera progresiva hacia el objetivo final. El cronograma se estructuró en cuatro fases principales:

1. Septiembre: Investigación y Toma de Decisiones

* Objetivo: Establecer la base conceptual del proyecto.
* Actividades:
* Investigación de tecnologías, herramientas y metodologías aplicables.
* Identificación de los requisitos del software y definición de su alcance.
* Toma de decisiones clave sobre el diseño general y la arquitectura.
* Resultado: Una estructura clara y un plan técnico para las fases posteriores.

1. Octubre: Programación de la Primera Versión

* Objetivo: Desarrollar una versión funcional mínima del software.
* Actividades:
* Creación del código base para las funcionalidades críticas.
* Integración de las diferentes partes del sistema (gestión de archivos, interfaz de usuario, base de datos, etc.).
* Resolución de problemas técnicos fundamentales.
* Resultado: Una primera versión completa con las funcionalidades esenciales.

1. Noviembre: Revisiones, Chequeos y Documentación

* Objetivo: Garantizar que el software cumpliera con los requisitos establecidos y generar la documentación requerida.
* Actividades:
* Pruebas exhaustivas del software para detectar y corregir errores.
* Refinamiento de las funcionalidades existentes.
* Redacción de la memoria y anexos técnicos.
* Resultado: Un software funcional y estable, con una documentación preliminar.

1. Diciembre: Mejoras, Cierres y Segunda Versión

* Objetivo: Entregar una versión final del software y la documentación.
* Actividades:
* Implementación de mejoras y ajustes.
* Finalización de la memoria, anexos y materiales de defensa.
* Resultado: Una segunda versión lista para la entrega final.

### Impacto de la Falta de Tiempo de Investigación y Formación

La falta de tiempo afectó la posibilidad de realizar actividades fundamentales en proyectos de software más amplios, como:

* Pruebas y Testeo Automatizado: Sin conocimiento ni tiempo, no se implementaron sistemas de pruebas automatizadas.
* Integración y Despliegue Continuo (CI/CD): Esta metodología no pudo ser utilizada.
* Formación en Herramientas Avanzadas: Aunque se identificaron herramientas como Sphinx o GitHub Actions, no se tuvo margen para su aprendizaje e implementación.

### Reflexión sobre la Planificación

El esquema de milestones permitió mantener el control sobre el avance del proyecto y alcanzar los objetivos sin comprometer la calidad mínima requerida. Aunque este enfoque sacrificó algunos aspectos, como la posibilidad de explorar funcionalidades adicionales, estableció una base sólida sobre la que se podrían construir futuras iteraciones.

La planificación fue fundamental para garantizar el cumplimiento de los plazos y demostrar que, incluso con recursos y tiempo limitados, es posible entregar un producto funcional y documentado.

## Inexperiencia Profesional y Desconocimiento Práctico en la Gestión de un Proyecto de Software

El tercer pilar del proyecto radica en una limitación significativa: mi falta de experiencia profesional y el desconocimiento práctico sobre la gestión de un proyecto de software a nivel profesional. Este desafío se deriva de mi formación, hasta ahora centrada exclusivamente en un entorno académico universitario.

El Contexto Académico: Nivel Universitario

Mi formación en Ingeniería Informática ha proporcionado una base sólida en teoría de algoritmos, estructuras de datos, y lenguajes de programación. Sin embargo, esta base no incluye una exposición directa a la gestión integral de proyectos de software. La enseñanza académica se ha centrado en aspectos técnicos específicos, dejando de lado:

* Gestión de proyectos: No se han cubierto metodologías como Scrum, Kanban, o la planificación de recursos en profundidad.
* Pruebas y validación: Aunque se menciona la importancia de las pruebas, no se ha enseñado cuándo y cómo diseñarlas ni qué herramientas utilizar.
* Ciclo de vida completo del software: No se ha abordado cómo llevar un proyecto desde la concepción hasta el despliegue y mantenimiento.

Falta de Formación en Validación y Pruebas

Un área especialmente crítica ha sido la ausencia de formación en validación y pruebas. Aunque existe una asignatura optativa en mi plan de estudios dedicada a este tema, no la he cursado. Esto me ha dejado sin conocimientos prácticos sobre:

* Pruebas unitarias y de integración: Cómo diseñarlas e implementarlas.
* Cobertura de código: Métodos para garantizar que todo el software esté adecuadamente probado.
* Automatización de pruebas: Herramientas y técnicas para integrar pruebas en el ciclo de desarrollo.

Este desconocimiento ha condicionado significativamente mi capacidad para implementar procesos de validación robustos en el proyecto, limitándome a pruebas manuales y sin posibilidad de garantizar una cobertura exhaustiva del código.

Falta de Experiencia con Herramientas Profesionales

Aunque en este proyecto he utilizado herramientas como Git, GitHub, y Visual Studio Code, mi uso de estas ha sido básico. Muchas funcionalidades avanzadas, como la integración y despliegue continuo (CI/CD) o la configuración de entornos de desarrollo colaborativos, quedaron fuera de mi alcance debido a:

* Curva de aprendizaje: El tiempo limitado para aprender a utilizarlas en profundidad.
* Desconocimiento inicial: No sabía qué herramientas estaban disponibles ni cuáles eran más adecuadas para mi caso.
* Enfoque en resultados inmediatos: La presión por completar el proyecto en el plazo establecido no permitió dedicar tiempo a una formación más detallada.

Desafíos Asociados al Trabajo Individual

El trabajo en solitario amplificó estas limitaciones. A diferencia de un entorno profesional, donde existen equipos con roles especializados (desarrolladores, testers, project managers), en este proyecto he tenido que asumir todas las responsabilidades:

* Diseño del software: Desde la arquitectura hasta la interfaz gráfica.
* Desarrollo del código: Escribir, probar y corregir el software.
* Documentación: Redactar tanto la memoria como los anexos técnicos.
* Planificación: Organizar las tareas y establecer prioridades sin experiencia previa.

Esta multiplicidad de roles, junto con la falta de experiencia, ha afectado la calidad y eficiencia del proyecto en comparación con lo que podría lograrse en un equipo profesional.

Reflexión sobre la Inexperiencia

Aunque estas limitaciones han representado un reto considerable, también han sido una fuente de aprendizaje. He adquirido un conocimiento básico sobre herramientas y metodologías que no formaban parte de mi formación inicial, y he comprendido la importancia de:

* Planificar pruebas desde el inicio del proyecto.
* Aprender a utilizar herramientas avanzadas para optimizar el desarrollo.
* Buscar formación complementaria en áreas no cubiertas por el plan de estudios.

Sin embargo, la falta de experiencia práctica sigue siendo una barrera significativa que ha condicionado el alcance y la profundidad del proyecto.

Impacto en el Proyecto

La inexperiencia y el enfoque académico han llevado a:

* Ausencia de pruebas automatizadas: Lo que reduce la confianza en la robustez del software.
* Falta de optimización del proceso: Algunas decisiones podrían haberse mejorado con un conocimiento más amplio.
* Un proceso de aprendizaje paralelo: Gran parte del tiempo del proyecto se ha dedicado a aprender conceptos y herramientas sobre la marcha.

Este tercer pilar pone de manifiesto cómo las limitaciones personales y formativas han influido profundamente en el desarrollo del proyecto. A pesar de ello, el esfuerzo por superar estos desafíos y entregar un resultado funcional refleja la capacidad de adaptación y la voluntad de aprendizaje continuo.

Lecciones Aprendidas sobre Pruebas

A lo largo del proyecto, he comprendido que las pruebas deben ser una parte esencial del desarrollo de software. Si bien en este caso las pruebas se han realizado de manera manual y limitada, la experiencia me ha llevado a reflexionar sobre la importancia de:

* Diseñar pruebas desde el inicio: Integrar las pruebas como parte del flujo de trabajo y no como una actividad posterior al desarrollo.
* Automatizar pruebas: Implementar herramientas y frameworks que permitan validar automáticamente el funcionamiento del software con cada nueva iteración.
* Aprender herramientas avanzadas: Profundizar en conocimientos sobre metodologías y herramientas como pytest o frameworks de pruebas específicas para interfaces gráficas y bases de datos.

Aplicación en el Proyecto

Durante el desarrollo del TFG, he tenido que aprender muchas de estas lecciones sobre la marcha, lo que ha ralentizado el proceso y ha limitado la calidad técnica del resultado final. Aunque ahora entiendo mejor la importancia de realizar pruebas en paralelo al desarrollo, esta visión llegó tarde en el proyecto, impidiendo que pudiera ser completamente implementada.

Reflexión sobre la Formación Académica

El enfoque académico ha sido útil para adquirir los fundamentos del desarrollo de software, pero ha dejado lagunas importantes en áreas prácticas clave, como:

* Diseño y ejecución de pruebas: Entender cuándo y cómo implementarlas en el ciclo de desarrollo.
* Gestión profesional de proyectos: Incluir procesos como CI/CD, validaciones de calidad, y optimización colaborativa.
* Metodologías avanzadas: Aunque herramientas como Zube y Trello fueron introducidas, su uso fue orientado únicamente a gestionar tareas individuales o de equipo en proyectos sencillos.

Este aspecto del proyecto subraya la necesidad de una formación más integral que combine teoría y práctica, preparando mejor a los estudiantes para enfrentarse a proyectos reales. A pesar de las dificultades, esta experiencia ha sido clave para comprender las exigencias del desarrollo profesional de software y para plantear mejoras en futuros proyectos.

## Sencillez arquitectónica del proyecto como estrategia fundamental.

Un aspecto relevante del desarrollo de este proyecto ha sido la elección consciente de una arquitectura y solución técnica sencilla. Esta decisión no se basa en una limitación de capacidad, sino en una estrategia deliberada para alinear el alcance del proyecto con mi nivel de experiencia actual y maximizar el aprendizaje global de todo el proceso.

La Importancia de Adaptar la Complejidad al Nivel de Experiencia

Este proyecto representa mi primera incursión en el desarrollo completo de software de manera individual, desde la concepción hasta la entrega final. En este contexto, abordar un problema con soluciones técnicas extremadamente complejas habría desviado mis esfuerzos hacia desafíos específicos, como algoritmos avanzados o integraciones sofisticadas, en detrimento de una visión más amplia del ciclo de vida de un proyecto de software. Al optar por un enfoque más asequible técnicamente, he podido:

* Centrarme en el proceso completo: Desde la planificación, implementación y pruebas, hasta la documentación y presentación final.
* Ganar confianza: Lograr resultados tangibles ha reforzado mi autoconfianza para futuros proyectos más ambiciosos.
* Enfocar el aprendizaje: Adquirir experiencia práctica en herramientas, metodologías y estructuras necesarias para gestionar un proyecto desde una perspectiva general.

Equilibrio entre Simplicidad y Funcionalidad

La simplicidad arquitectónica no significa falta de ambición o un diseño básico. En cambio, se trata de encontrar un equilibrio entre:

* Soluciones funcionales: Resolver de manera eficaz la problemática planteada sin añadir capas innecesarias de complejidad.
* Capacidad de aprendizaje: Crear un proyecto que permita aprender y consolidar conceptos fundamentales del desarrollo de software.
* Tiempo disponible: Adaptar la carga de trabajo al plazo reducido del proyecto, sin comprometer los resultados.

Por ejemplo, en este proyecto, opté por:

* Una estructura en capas para organizar el código y garantizar su mantenimiento.
* Algoritmos sencillos pero efectivos para la gestión de documentos y asignación de requisitos.
* Herramientas accesibles, como GitHub y Visual Studio Code, que son adecuadas para el alcance del proyecto.

Aprendizajes Derivados de la Simplicidad

La elección de una solución sencilla me permitió trabajar con un enfoque integral, abordando aspectos más generales del desarrollo de software que podrían haber quedado relegados en un proyecto más complejo. Algunos aprendizajes clave incluyen:

* Gestión de un proyecto completo: La simplicidad técnica permitió dedicar tiempo a tareas como planificación, documentación y pruebas, que son fundamentales en cualquier desarrollo profesional.
* Comprensión del flujo de trabajo: Realizar un proyecto desde cero me ayudó a entender cómo se interconectan las diferentes etapas del desarrollo.
* Experiencia con herramientas y metodologías: Aunque básicas, las herramientas empleadas me proporcionaron una primera experiencia práctica en la gestión del código, control de versiones y colaboración.

Preparación para Proyectos Más Ambiciosos

Este enfoque equilibrado no solo ha garantizado la finalización exitosa del proyecto, sino que también ha establecido una base sólida para enfrentar futuros desafíos. En proyectos venideros, planeo:

* Aplicar lo aprendido para abordar problemas más complejos con mayor confianza y eficiencia.
* Dedicar más tiempo y esfuerzo a resolver problemas técnicos avanzados, dado que ya tengo experiencia con el resto de elementos del desarrollo.
* Incrementar el nivel de ambición técnica sin comprometer la organización y el flujo de trabajo.

Reflexión Final

La decisión de mantener una arquitectura sencilla ha sido clave para el éxito del proyecto. Más allá de los resultados obtenidos, este enfoque ha permitido transformar una experiencia limitada en un aprendizaje significativo y holístico. Este cuarto pilar subraya la importancia de empezar con proyectos acordes al nivel de experiencia, no como una limitación, sino como una estrategia para construir una base sólida que permita crecer y asumir retos mayores en el futuro.

## Revisión y Mejora del Código con Herramientas de Análisis

Un aspecto relevante del proyecto ha sido la introducción y uso de herramientas para la revisión y mejora del código, algo que nunca había experimentado antes. Herramientas como Pylint, Black, y SonarCloud no solo han ayudado a garantizar la calidad técnica del código, sino que también han introducido conceptos clave como:

* Calidad del código: Medir y mejorar la legibilidad, consistencia y adherencia a buenas prácticas.
* Detección temprana de errores: Identificar fallos potenciales antes de que se conviertan en problemas mayores.
* Refactorización: Ajustar el código para hacerlo más eficiente y claro, sin alterar su funcionalidad.

Impacto en el proyecto:

Estas herramientas han sido fundamentales para introducirme en un estándar profesional, enseñándome a aplicar buenas prácticas que antes desconocía y preparando el proyecto para su mantenimiento y evolución.

## Creación de una Identidad Visual

Para reforzar la profesionalidad y dar cohesión al proyecto, se diseñó una marca con un logo propio que representa su identidad. Este aspecto, aunque no técnico, añade un valor significativo:

* Proyección profesional: Facilita que el proyecto sea percibido como un producto completo y no solo como un ejercicio académico.
* Unificación de materiales: El logo y la identidad visual están presentes en la interfaz gráfica, la documentación y las presentaciones asociadas.
* Aprendizaje en diseño gráfico: Aunque no es mi área de especialización, este proceso implicó investigar y experimentar con herramientas de diseño para crear una imagen visual atractiva.

Impacto en el proyecto:

La creación de una identidad visual ha permitido presentar el proyecto con mayor solidez y profesionalidad, demostrando atención al detalle y una visión integral del desarrollo.

## Planificación de una Segunda Versión

Desde el inicio, se consideró la posibilidad de desarrollar una segunda versión del software, centrada en:

* Cumplir completamente con la metodología Scrum: Incorporando iteraciones adicionales y funcionalidades incrementales.
* Implementar mejoras técnicas: Refinar el código, aumentar la cobertura de pruebas y añadir características más avanzadas.
* Incorporar funcionalidades adicionales: Basadas en el feedback recibido o en oportunidades detectadas durante la primera versión.

Impacto en el proyecto:

La planificación de esta segunda versión no solo refleja un compromiso con la mejora continua, sino que también establece una hoja de ruta para futuras iteraciones que pueden elevar el proyecto a un nivel más profesional.

## Revisión y Aplicación de Conceptos Académicos

Durante el desarrollo del proyecto, ha sido necesario realizar un "repaso" de conceptos aprendidos en asignaturas previas. Este ejercicio ha permitido:

* Consolidar conocimientos: Reforzar áreas clave como estructuras de datos, algoritmos, diseño de bases de datos y patrones de diseño.
* Explorar alternativas: Evaluar distintas herramientas y metodologías antes de tomar decisiones clave en el proyecto.
* Construir una narrativa técnica sólida: Basar la argumentación de las decisiones tomadas en fundamentos teóricos bien establecidos.

Impacto en el proyecto:

Este enfoque ha permitido aplicar el aprendizaje académico de manera práctica, demostrando cómo los conceptos teóricos se convierten en soluciones tangibles.

## Adaptación a la Metodología Scrum

Aunque el proyecto ha seguido un enfoque simplificado de Scrum debido a las restricciones del TFG, este proceso ha introducido elementos clave de la metodología:

* División en sprints: Organizar el trabajo en iteraciones cortas y manejables.
* Priorizar tareas: Fijar objetivos concretos para cada sprint.
* Evaluación continua: Revisar los avances y ajustarlos según las necesidades del proyecto.

Impacto en el proyecto:

Aunque el uso de Scrum fue básico, ha proporcionado una estructura organizativa que ha sido esencial para gestionar el tiempo y priorizar tareas de manera efectiva.

## Reflexión sobre la Adaptabilidad y el Aprendizaje Continuo

Un aspecto intangible pero crucial ha sido la capacidad de adaptarse a las circunstancias y aprender sobre la marcha. Este proyecto ha requerido:

* Flexibilidad: Ajustar objetivos y enfoques según las limitaciones de tiempo y conocimiento.
* Autodidactismo: Investigar, aprender y aplicar herramientas y conceptos nuevos de manera autónoma.
* Resiliencia: Superar desafíos técnicos y personales, manteniendo el foco en la entrega de resultados.

Impacto en el proyecto:

Este proceso de aprendizaje continuo no solo ha enriquecido el proyecto, sino que también ha consolidado habilidades clave para futuros desafíos profesionales.

## Reflexión Final sobre los Aspectos Relevantes

Los elementos descritos en este apartado muestran cómo el desarrollo del proyecto no solo se limitó a escribir código o cumplir con requisitos técnicos. Cada uno de estos aspectos refleja un esfuerzo por construir un proyecto completo, equilibrado y alineado con mis circunstancias personales y profesionales, estableciendo una base sólida para el futuro.

# Trabajos relacionados

# Conclusiones y Líneas de trabajo futuras

Textp general

## Conclusiones

## Líneas de trabajo futuras

Bibliografía