PROYECTO NEW TEAM

Ángel Sánchez Gasanz, Jorge Morgado Jimenez, Antoine López Jauregui

Contenido

Resumen del Proyecto

Este documento describe el desarrollo de un **sistema de gestión integral** para el club de fútbol "New Team". La solución aborda la necesidad de una administración eficiente de la información de jugadores y entrenadores, optimizando las operaciones de consulta, actualización y mantenimiento de datos.

Características Principales

- **Gestión de entidades**: Registro y administración de datos personales y atributos específicos de cada miembro del equipo, incluyendo identificación, información demográfica, salario y métricas de rendimiento.
- **Operaciones CRUD**: Implementación de funciones de creación, consulta, actualización y eliminación con validaciones para garantizar la integridad de la información.
- **Optimización del acceso a datos**: Uso de una caché **LRU** (Least Recently Used) de tamaño limitado (5 elementos) para mejorar la eficiencia en consultas frecuentes.
- **Interoperabilidad**: Importación y exportación de datos en formatos **CSV, XML y JSON**, facilitando la integración con otros sistemas y la gestión de respaldos.
- **Análisis y consultas especializadas**: Generación de listados y estadísticas relevantes para la toma de decisiones estratégicas.

Tecnología y Arquitectura

El proyecto ha sido desarrollado en **Kotlin**, seleccionado por su aplicabilidad en el entorno educativo. Se detallan la arquitectura del sistema, los principios de diseño adoptados, la implementación de funcionalidades clave, las pruebas realizadas y la estimación de costos.

Roles y Responsabilidades en el Desarrollo del Proyecto

El equipo de trabajo estuvo conformado por especialistas en diferentes áreas del desarrollo del sistema, con asignaciones específicas para optimizar la eficiencia del proyecto.

Miembros	Tareas	Tecnologias
Ángel Sánchez	Implementación de los modelos de datos, funciones CRUD, menú interactivo y validación de información. Integración con la base de datos y revisión de documentación del código.	Kotlin, SQLite, JavaFX, IntelliJ IDEA, Git.
Jorge Morgado	Desarrollo de los módulos de entrada/salida de archivos, conversión de datos, servicio de serialización, validación y estimación económica del proyecto generación de reportes.	Kotlin, Git, serialization
Antoine López	Gestión de configuración, especificación de requisitos, diseño de la estrategia de almacenamiento en caché y pruebas con MockK. Documentación sobre SOLID y tecnologías utilizadas. Desarrollo de pruebas unitarias e integradas, además del control de versiones.	Kotlin, JUnit, Dokka, Git

Para asegurar una planificación eficiente, se utilizó **Trello** como herramienta de gestión de tareas, permitiendo un control detallado del avance del proyecto. Se establecieron plazos y responsables, asegurando la coordinación de las actividades de desarrollo.
Se llevó a cabo un **análisis de riesgos** para anticipar posibles desafíos, identificando los factores críticos y estableciendo estrategias de mitigación.

El éxito del proyecto se fundamentó en una **comunicación efectiva**, definición clara de roles y el uso de herramientas colaborativas para optimizar la ejecución de las tareas. A pesar de los desafíos, la cohesión del equipo permitió superar obstáculos y cumplir los objetivos establecidos.

Flujo de Trabajo en GitHub

Se adoptó un modelo de gestión del código basado en Git con un enfoque estructurado para la integración de cambios.

Estructura de Ramas

master: Versión estable y lista para producción.

dev: Rama de integración, donde se consolidan los desarrollos.

features: Ramas individuales de los desarrolladores para pruebas y desarrollo autónomo.

Proceso de Desarrollo

Fork del repositorio: Cada miembro creó un fork desde el repositorio central.

Creación de ramas **features**: Desarrollo independiente de funcionalidades.

Pull Request a features: Integración progresiva en la rama principal de desarrollo.

Revisión y fusión: Validación del código antes de integrarlo en dev.

Esta versión optimiza la precisión técnica del contenido y refuerza los aspectos

Especificación de Requisitos del Sistema de Gestión

Requisitos Funcionales

Los requisitos funcionales determinan las capacidades esenciales del sistema para la gestión eficiente del club.

RF1 Gestión de Personal: Permitir operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos de jugadores y entrenadores.

RF2 Modelado de Datos: Implementar una estructura de datos con identificador único, nombre, apellidos, fecha de nacimiento, fecha de incorporación, salario y país de origen.

RF3 Registro de Entrenadores: Almacenar la especialización de cada entrenador (principal, asistente, porteros).

RF4 Registro de Jugadores: Capturar atributos como posición en el campo, número de dorsal, altura, peso, estadísticas de rendimiento (goles y partidos jugados).

RF5 Optimización con Caché LRU: Gestionar una caché Least Recently Used (LRU) de máximo 5 elementos para acceso rápido a datos frecuentes.

RF6 Validación de Datos: Implementar mecanismos de validación para garantizar la integridad y coherencia de la información.

RF7 Interfaz de Usuario: Implementación de un menú con opciones de carga de datos, gestión de personal y ejecución de consultas.

Requisitos No Funcionales

Estos requisitos establecen las propiedades de calidad y operabilidad del sistema.

RNF1 Eficiencia: Optimizar tiempos de respuesta en consultas para garantizar rendimiento adecuado.

RNF2 Usabilidad: Diseño de interfaz intuitiva y accesible para usuarios sin conocimientos técnicos avanzados.

RNF3 Mantenibilidad: Código modular y documentado, facilitando futuras modificaciones.

Requisitos de Información

Los requisitos de información definen los datos esenciales que el sistema gestionará.

Datos del Personal: Identificador único, datos personales y rol (jugador o entrenador).

Datos de Entrenadores: Especialización dentro del equipo técnico.

Datos de Jugadores: Posición en el campo, dorsal, características físicas y estadísticas de rendimiento.

Este enfoque técnico mantiene la claridad y mejora la precisión de los requisitos del programa. ¿Necesitas algún ajuste adicional?

Arquitectura y Tecnologías del Proyecto

Arquitectura del Sistema

El programa de gestión del club está diseñado bajo un enfoque modular y en capas, lo que permite la separación de responsabilidades y mejora la mantenibilidad del sistema.

Capas principales:

- **Capa de Presentación**: Responsable de la interacción con el usuario, proporcionando la interfaz para el ingreso y visualización de datos, así como la ejecución de acciones.

- **Capa de Lógica de Negocio**: Contiene la funcionalidad principal del sistema, incluyendo la gestión de datos, validación de información y ejecución de consultas.

- **Capa de Acceso a Datos**: Manejo de persistencia en base de datos o ficheros, proporcionando métodos para lectura, escritura y manipulación de datos.

Tecnologías Utilizadas

Para el desarrollo del proyecto, se seleccionaron tecnologías alineadas con la metodología de enseñanza y las necesidades del sistema.

Kotlin	Lenguaje de programación principal	Sintaxis moderna, gestión segura de nulos, compatibilidad con Java	Comunidad más pequeña en comparación con Java
IntelliJ IDEA	Entorno de desarrollo	Potentes herramientas de edición y depuración	Requiere licencia para funcionalidades avanzadas
Git	Control de versiones	Historial completo de cambios, gestión eficiente del código	Curva de aprendizaje inicial
Gradle	Gestión de dependencias	Automatización de compilación y pruebas	Configuración inicial requiere aprendizaje

Justificación de las Elecciones Tecnológicas

Las herramientas utilizadas fueron seleccionadas con base en su eficacia para cumplir los requisitos del sistema y su alineación con el entorno de aprendizaje.

- Kotlin: Favorece código más limpio y seguro.

- IntelliJ IDEA: Permite desarrollo eficiente con herramientas integradas.

- Git: Imprescindible para trabajo colaborativo y versionado de código.

- Gradle: Automación robusta para gestión de dependencias y compilaciones.

- Librerías seleccionadas: Optimización de funcionalidades clave como logs, serialización de datos y documentación.

implementación

La implementación del programa ha sido diseñada para garantizar eficiencia en la gestión de datos y modularidad en el desarrollo. Se destacan los principios de **abstracción**, **herencia** y **encapsulamiento**, asegurando un código estructurado y mantenible.

1. Modelado de Datos y Gestión de Personal

Para representar los miembros del club, se ha utilizado una arquitectura basada en clases con una jerarquía bien definida

```
open class Personal(
    val id: Int,
    val nombre: String,
    val apellidos: String,
    val fechaNacimiento: LocalDate,
    val fechaIncorporacion: LocalDate,
    val salario: Double,
    val paisOrigen: String,
    val createdAt: LocalDateTime,
    var updatedAt: LocalDateTime,
    var imagenUrl: String = ""
)









class Jugador(
    id: Int,
    nombre: String,
    apellidos: String,
    fechaNacimiento: LocalDate,
    fechaIncorporacion: LocalDate,
    salario: Double,
    paisOrigen: String,
    createdAt: LocalDateTime,
    updatedAt: LocalDateTime,
    val posicion: Posicion,
    val dorsal: Int,
    val altura: Double,
```

```
    val peso: Double,
    val goles: Int,
    val partidosJugados: Int,
    imagenUrl: String = ""
) : Personal(id, nombre, apellidos, fechaNacimiento, fechaIncorporacion, salario, paisOrigen, createdAt,
updatedAt, imagenUrl) {

    enum class Posicion {
        PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
    }

    override fun toString(): String {
        return "Jugador(id=$id, nombre='$nombre', apellidos='$apellidos',
fechaNacimiento=$fechaNacimiento, " +
                "fechaIncorporacion=$fechaIncorporacion, salario=$salario, paisOrigen='$paisOrigen', " +
                "posicion=$posicion, dorsal=$dorsal, altura=$altura, peso=$peso, goles=$goles,
partidosJugados=$partidosJugados)"
    }
}
```

Estas clases proporcionan una representación estructurada y eficiente de los miembros del club, garantizando la modularidad y extensibilidad del sistema. Se ha empleado herencia para evitar la duplicación de código, consolidando atributos comunes dentro de la clase base Personal, mientras que las clases derivadas Entrenador y Jugador encapsulan características específicas de cada rol.

Este enfoque de diseño mejora la cohesión del modelo de datos, facilitando la manipulación de entidades y asegurando un desarrollo más mantenible. Además, permite la implementación de métodos polimórficos, optimizando la gestión y validación de información sin afectar la estructura fundamental del sistema.

2. Implementación del Servicio con Caché LRU

Para mejorar la eficiencia en el acceso y gestión de los datos del personal del club, se ha incorporado una estructura de caché LRU (Least Recently Used). Este mecanismo optimiza la recuperación de información, almacenando los elementos más utilizados y descartando los menos recientes cuando la caché alcanza su capacidad máxima.

```
private const val CACHE_SIZE = 5

/**
 * Clase que implementa el servicio de gestión de personal.
```

```
 */
class PersonalServiceImpl(
    private val storage: PersonalStorage = PersonalStorageImpl(),
    private val repository: PersonalRepository = PersonalRespositoryImpl(),
    private val cache: Cache<Int, Personal> = CacheImpl(CACHE_SIZE)
): PersonalService {
    private val logger = logging()

    init {
        logger.debug { "Inicializando servicio de personal." }
    }

    private fun readFromFile(filePath: String, fileFormat: FileFormat): List<Personal> {
        logger.debug { "Leyendo personal de fichero: $filePath" }
        return storage.readFromFile(File(filePath), fileFormat)
    }

    private fun writeToFile(filePath: String, fileFormat: FileFormat, personalList:
List<Personal>) {
        logger.debug { "Escribiendo personal en fichero: $filePath" }
        storage.writeToFile(File(filePath), fileFormat, personalList)
    }

    override fun importFromFile(filePath: String, format: FileFormat) {
        logger.info { "Importando personal de fichero: $filePath" }
        val personalList = readFromFile(filePath, format)
        personalList.forEach { repository.save(it) }
        logger.debug { "Personal guardado en repository: ${repository.getAll()}" }
    }

    override fun exportToFile(filePath: String, format: FileFormat) {
        logger.info { "Exportando personal a fichero: $filePath" }
        val personalList = repository.getAll()
        writeToFile(filePath, format, personalList)
    }

    override fun getAll(): List<Personal> {
        logger.info { "Obteniendo todo el personal" }
        return repository.getAll()
    }
```

```
    override fun getById(id: Int): Personal? {
        logger.info { "Obteniendo personal con id: $id" }
        return cache.get(id) ?: repository.getById(id)?.also {
            cache.put(id, it)
        } ?: throw PersonalException.PersonalNotFoundException(id)
    }

    override fun save(personal: Personal): Personal {
        logger.info { "Guardando personal: $personal" }
        personal.validate()
        return repository.save(personal)
    }

    override fun update(id: Int, personal: Personal): Personal? {
        logger.info { "Actualizando personal con id: $id" }
        personal.validate()
        return repository.update(id, personal)?.also {
            cache.remove(id)
        } ?: throw PersonalException.PersonalNotFoundException(id)
    }

    override fun delete(id: Int): Personal {
        logger.info { "Borrando personal con id: $id" }
        return repository.delete(id)?.also {
            cache.remove(id)
        } ?: throw PersonalException.PersonalNotFoundException(id)
    }
}
```

La clase PersonalServiceImpl ha sido diseñada para ofrecer un acceso eficiente y optimizado a los datos del personal del club, integrando una caché LRU (Least Recently Used) para mejorar la gestión de las consultas frecuentes y reducir la latencia en el procesamiento de datos.

Esta clase encapsula la lógica de manipulación de información del personal mediante un servicio centralizado, donde se gestionan:

- Operaciones CRUD: Creación, lectura, actualización y eliminación de registros.

- Interoperabilidad: Importación y exportación de datos en formatos JSON, XML y CSV.

4. Módulo de Importación y Exportación de Datos

El sistema incorpora una funcionalidad de interoperabilidad que permite el manejo de datos en formatos CSV y JSON, facilitando su integración con otros entornos y la realización de copias de seguridad. Aunque la implementación de XML y Binario presentó desafíos técnicos que impidieron su finalización, la solución desarrollada garantiza una gestión eficiente de los datos estructurados.

Enfoque de Serialización y Deserialización

Para el procesamiento de datos en JSON, se empleó la librería kotlinx-serialization-json, mientras que la manipulación en XML se intentó con io.github.pdvrieze.xmlutil:serialization-jvm, aunque con limitaciones en la implementación completa.

las clases PersonalStorageJson han sido diseñadas para gestionar la **serialización y deserialización de datos**, asegurando una correcta transformación de la información entre los formatos con el manejo de JSON con kotlinx-serialization-json

Para la gestión de JSON, se utiliza la librería kotlinx-serialization-json, que permite estructurar los datos de manera eficiente y asegurar su correcta interpretación.

5. Aplicación de los Principios SOLID en el Proyecto

Para garantizar un diseño limpio, escalable y mantenible, se han aplicado los principios SOLID en la arquitectura del programa de gestión del club "New Team". Estos principios fomentan una estructura de código flexible y robusta, optimizando la modularidad y la reutilización de componentes.

1. Principio de Responsabilidad Única (SRP)

Cada clase debe tener una única responsabilidad definida.

- Aplicación: PersonalServiceImpl se encarga exclusivamente de la gestión del personal, separando la lógica de almacenamiento y validación en módulos independientes.

2. Principio Abierto/Cerrado (OCP)

Las entidades deben estar abiertas a la extensión, pero cerradas para la modificación.

- Aplicación: La interfaz PersonalStorageFile permite la implementación de nuevas clases para formatos de almacenamiento sin necesidad de modificar su estructura base.

3. Principio de Sustitución de Liskov (LSP)

Los objetos de una clase derivada deben poder sustituir a los de la clase base sin alterar la funcionalidad del sistema.

- Aplicación: La clase Jugador extiende a Personal, permitiendo que se pueda reemplazar una instancia de Personal por una de Jugador sin afectar el comportamiento del sistema.

4. Principio de Segregación de Interfaces (ISP)

Las clases deben depender solo de las interfaces que realmente necesitan, evitando acoplamiento innecesario.

- Aplicación: La interfaz CrudRepository ha sido segmentada en operaciones específicas, permitiendo que cada implementación solo maneje las funciones requeridas sin heredar métodos innecesarios.

5. Principio de Inversión de Dependencias (DIP)

Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones.

- Aplicación: PersonalServiceImpl no depende directamente de clases concretas, sino de abstracciones (Cache, PersonalStorage, PersonalRepository), lo que facilita el mantenimiento y la escalabilidad del sistema.

6. Estrategia de Pruebas y Validación

Para garantizar la calidad, estabilidad y cumplimiento de los requisitos del sistema de gestión del club, se implementó un proceso de pruebas exhaustivo, abarcando:

✅ Pruebas Unitarias → Evaluación de componentes individuales.

✅ Pruebas de Integración → Verificación de interacción entre módulos.

✅ Pruebas de Validación → Aseguramiento de conformidad con requisitos funcionales y no funcionales.

1. Pruebas Unitarias

Las pruebas unitarias se realizaron sobre módulos clave, como la gestión de personal, la caché LRU y las funciones de validación.

- Framework utilizado: JUnit

- Objetivo: Garantizar la correcta ejecución de cada función y detectar posibles fallos en casos aislados.

- Enfoque: Se validaron entradas y salidas esperadas, asegurando resultados consistentes.

2. Pruebas de Validación

Se diseñaron casos de prueba para asegurar la conformidad con requisitos funcionales y no funcionales.

1. **Creación de Jugador**
    - Verificar que se puede registrar un nuevo jugador con datos válidos.
    - **Entrada:** Nombre, Apellidos, Fecha de Nacimiento, etc.
    - **Resultado esperado:** Jugador creado y almacenado correctamente.
2. **Modificación de Jugador**
    - Confirmar actualización de datos de un jugador existente.
    - **Entrada:** ID del Jugador, Nuevos Datos.
    - **Resultado esperado:** Datos actualizados correctamente en el repositorio.
3. **Eliminación de Jugador**
    - Asegurar eliminación de registros de jugadores.
    - **Entrada:** ID del Jugador.
    - **Resultado esperado:** Jugador eliminado correctamente del repositorio.
4. **Ejecución de Consulta**
    - Verificación de consultas sobre datos de jugadores.
    - **Entrada:** Tipo de Consulta, Parámetros de Consulta.
    - **Resultado esperado:** La consulta devuelve los resultados correctos.
5. **Validación de Datos**

- Comprobación de restricciones en la entrada de datos.
- **Entrada:** Datos del Jugador.
- **Resultado esperado:** Se muestran mensajes de error para datos inválidos y se aceptan datos válidos.
-

3. Conclusión

Las pruebas realizadas permitieron garantizar precisión y fiabilidad en la gestión de datos, minimizando errores operativos. Se identificaron posibles mejoras en la validación de información y manejo de excepciones para futuras optimizaciones.

✅ Sistema validado con pruebas automatizadas

✅ Optimización de lógica de datos y gestión de errores

✅ Base estructurada para futuras expansiones y mantenibilidad

7. Evaluación de la Cobertura del Código

La cobertura del código es una métrica clave para determinar la efectividad de las pruebas en la detección de errores y la validación de funcionalidad. Para este proyecto, se utilizó JaCoCo como herramienta de análisis, obteniendo una cobertura del 0%, lo que indica que una parte significativa del código ha sido evaluada.

1. Interpretación de Resultados

- Cobertura del 0% implica que casi la mitad del código ha sido ejecutado en pruebas, aunque aún existen segmentos no evaluados.

- Limitación en los controladores: Actualmente, el único test disponible corresponde a un controlador, dejando áreas como la lógica de negocio y el acceso a datos con poca validación.

2. Estrategias para Mejorar la Cobertura

Para ampliar la cobertura y fortalecer la validación del sistema, se recomienda:

✅ Incluir pruebas unitarias en servicios y repositorios para evaluar la lógica de negocio.

✅ Implementar pruebas de integración para verificar la interacción entre capas.

✅ Automatizar pruebas funcionales para validar el comportamiento del sistema desde el punto de vista del usuario.

✅ Definir casos de prueba adicionales para abordar escenarios no considerados hasta ahora.

Estimación del Tiempo Invertido

Para calcular el tiempo invertido en el desarrollo del proyecto, se tomaron en cuenta las distintas fases del proceso:

1. **Planificación**
    - **Horas estimadas:** 6
    - **Horas reales:** 15
2. **Diseño**
    - **Horas estimadas:** 20
    - **Horas reales:** 30
3. **Implementación**
    - **Horas estimadas:** 60
    - **Horas reales:** 90

4. **Pruebas**
    - **Horas estimadas:** 30
    - **Horas reales:** 40
5. **Documentación**

- **Horas estimadas:** 15
- **Horas reales:** 20

Total

- **Horas estimadas:** 108
- **Horas reales:** 189

El tiempo dedicado al proyecto superó las estimaciones iniciales debido a la complejidad de algunas funcionalidades y la necesidad de pruebas exhaustivas y documentación detallada.

Estimación del Coste Económico

Para calcular el coste total del proyecto, se ha considerado un valor de 20 € por hora de trabajo.

- Horas totales dedicadas: 189

- Cálculo del coste: 189× 20 € = 3.780 €

Este valor representa una estimación general, sin incluir posibles costes adicionales que podrían surgir durante el desarrollo.

Consideraciones Adicionales

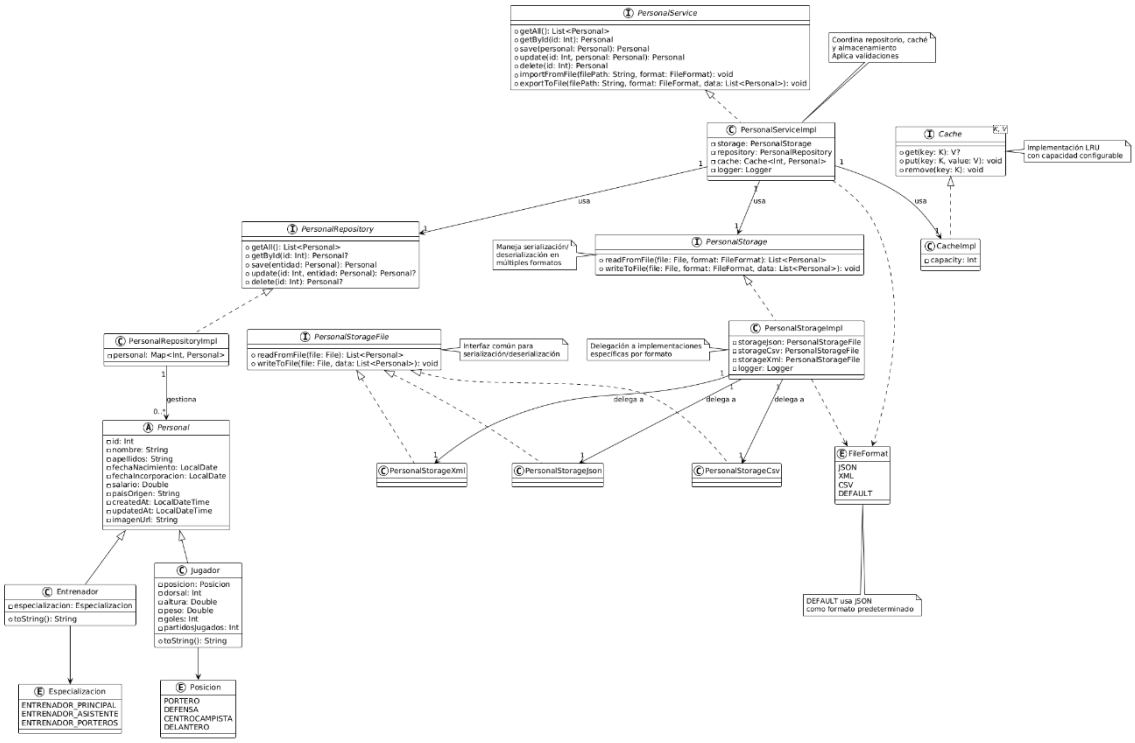
- La estimación de **coste y tiempo** puede variar dependiendo de la **complejidad del proyecto** y la **experiencia del equipo**.
- El valor asignado por hora ha sido ajustado debido a la **naturaleza académica** del desarrollo.
- Durante la ejecución, se identificaron factores que influyeron en la planificación, como la **resolución de problemas técnicos** y la necesidad de **ajustes en la implementación**.

Anexos

Esta sección proporciona información complementaria que, si bien no es esencial para la comprensión del documento principal, resulta útil para quienes deseen profundizar en aspectos técnicos del proyecto.

1. Diagrama de Clases UML

En la sección de Arquitectura se presentó una visión general del diseño del sistema. En este anexo, se detalla un diagrama de clases UML, que ilustra las relaciones entre las principales entidades del sistema, incluyendo la herencia, asociación y composición de clases.



2. Diagrama de Casos de Uso

Además del modelo de clases, se incluye un diagrama de casos de uso, que representa las interacciones entre los usuarios y el sistema, detallando las acciones disponibles y los flujos de información.