Proyecto New Team

Pablo De la Fuente De los Santos

Pablo Zuil Armenteros

Lucía Fuertes Cruz

Introducción al Proyecto “New Team”

Este documento presenta el desarrollo de un programa de gestión diseñado para el club de fútbol "New Team". El proyecto surge de la necesidad de administrar eficientemente la información relativa a los miembros del club, tanto entrenadores como jugadores, facilitando las operaciones de consulta, actualización y mantenimiento de los datos.

El programa implementado permite:

- Almacenar información detallada de cada miembro del equipo, incluyendo datos personales (identificador, nombre, apellidos, fecha de nacimiento, país de origen, salario) y datos específicos según su rol (especialidad para entrenadores, posición, número de camiseta, altura, peso, goles y partidos jugados para jugadores).
- Gestionar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre la información de los miembros del equipo, asegurando la integridad y validez de los datos mediante la implementación de mecanismos de validación.
- Optimizar el acceso a la información mediante la implementación de una caché LRU (*Least Recently Used*) de tamaño limitado (5 elementos), que permite reducir los tiempos de respuesta para las consultas más frecuentes.
- Importar y exportar datos en diversos formatos (*CSV, XML, JSON y binario*), facilitando la interoperabilidad con otros programas y la gestión de copias de seguridad.
- Realizar una serie de consultas específicas sobre la información almacenada, permitiendo obtener listados, estadísticas y datos relevantes para la toma de decisiones.

El desarrollo del proyecto se ha llevado a cabo utilizando el lenguaje de programación *Kotlin*, seleccionando esta tecnología por su implementación en el aula.

El presente documento describe la arquitectura del programa, las tecnologías empleadas, los principios de diseño aplicados, la implementación de las funcionalidades clave, las pruebas realizadas y una estimación económica del proyecto.

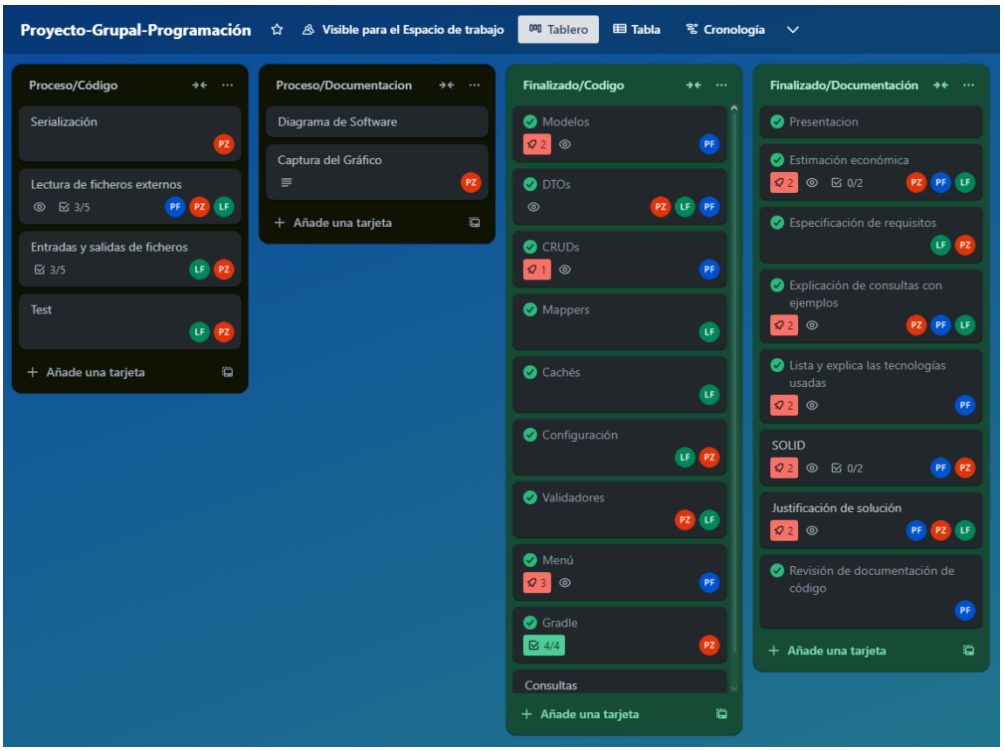
Organización del Equipo de Trabajo

Para llevar a cabo este proyecto, se conformó un equipo de trabajo con roles y responsabilidades definidos. La asignación de tareas se realizó considerando las habilidades y conocimientos de cada miembro, buscando maximizar la eficiencia y el cumplimiento de los objetivos establecidos.

La tabla siguiente resume la estructura del equipo y las principales funciones desempeñadas por cada integrante:

Miembro del Equipo	Tareas Principales	Tecnologías y Herramientas Utilizadas
Pablo De la Fuente De los Santos	Desarrollo de los **Modelos**, **CRUDS**, **Menú** e implementación de la **Justificación de Solución** y la **Revisión de Documentación del Código**. Implementación de la lógica de validación de datos. Integración con la base de datos.	Kotlin, IntelliJ IDEA, Git, GitHub, Dokka
Pablo Zuil Armenteros	Implementación de **Lectura de Ficheros Externos**, **Entradas y Salidas de Ficheros**, **Mappers**, **Service**, desarrollo de los **DTOs**, **Validadores**, **Serialización**, **Gradle** e implementación de la **Captura del Gráfico**, **Explicación de Consultas con Ejemplos** y **Estimación Económica**. Implementación de las consultas solicitadas en el proyecto.	Kotlin, Git, GitHub, JUnit, Dokka, IntelliJ IDEA
Lucía Fuertes Cruz	Implementación de **Lectura de Ficheros Externos**, **Entradas y Salidas de Ficheros**, **Cachés, Configuración,** la **Especificación de Requisitos**, además de la implementación del **Test con mockK**, el **SOLID** y la **Lista de Explicación de las Tecnologías Usadas**. Diseño y ejecución de pruebas unitarias e integrales. Gestión del control de versiones del código utilizando Git.	Kotlin, JUnit, Dokka, Git, GitHub, IntelliJ IDEA

Para la gestión del trabajo y el seguimiento del progreso del proyecto, se utilizó *Trello*. Esta herramienta permitió organizar las tareas, asignar responsables, establecer fechas de entrega y realizar un seguimiento del estado de cada tarea.



Además, se realizó un análisis de riesgos para identificar los posibles problemas que podrían afectar el desarrollo del proyecto y se definieron planes de contingencia para mitigar su impacto.

La tabla siguiente resume los principales riesgos identificados y las acciones implementadas para gestionarlos:

Riesgo Identificado	Probabilidad	Impacto	Plan de Contingencia
Conflictos con Git y sus comandos, ramas y actualizaciones del proyecto (*upstream*).	Alta	Medio	Realizar pruebas adicionales sobre el uso de Git y GitFlow. Establecer unos pasos claros para la gestión de ramas y la resolución de conflictos. Realizar reuniones frecuentes para sincronizar el trabajo y resolver dudas. Designar un miembro del equipo como "Git Master" que en este caso sería Lucía Fuertes Cruz, para asistir en la resolución de problemas.
Problemas con la lectura y escritura de ficheros en formato Binario y XML.	Media	Medio	No se ha solucionado, tras muchos intentos y mucho tiempo invertido, el equipo decidió proseguir con otras tareas para la finalización del proyecto y que este funcionara mínimamente.
Tiempo de entrega insuficiente para completar todas las tareas a la perfección.	Alta	Alto	Como estudiantes inexperimentados nos costó llevar el ritmo de trabajo deseado y por ello, otro equipo solicitó un aplazamiento de la entrega y exposición del proyecto.
Sobrecarga de trabajo y falta de descanso debido al esfuerzo dedicado al proyecto.	Media	Alto	Este apartado por motivos obvios para nuestro nivel de experiencia y profesionalidad no se solucionó. El equipo se ha esforzado duramente y con insistencia en la realización perfecta del proyecto y nos ha afectado negativamente en nuestro nivel regular de vida, tanto en horas de sueño como en proyectos de otros módulos que no han sido posibles de realizar ya que hemos llegado al punto de la obsesión para realizar lo esperado de manera eficiente y óptima.

La organización del equipo y la gestión del trabajo por parte de todos los integrantes fueron factores clave para el éxito del proyecto. La comunicación fluida entre los miembros del equipo, la definición clara de roles y responsabilidades, y la utilización de herramientas de gestión de proyectos permitieron coordinar las tareas, resolver los problemas de forma eficiente y cumplir con los plazos establecidos. Por lo que, en este proyecto grupal nos apoyamos los unos en los otros para lograr alcanzar una meta, y a pesar de tener algunas complicaciones por el camino, siempre salimos adelante juntos.

Flujo de Trabajo con Git

Para la gestión del código fuente del proyecto, se implementó un flujo de trabajo colaborativo basado en el sistema de control de versiones Git. Debido a la estructura del equipo y la forma en que se distribuyeron las tareas, se adoptó un enfoque de trabajo específico que se describe a continuación.

En este proyecto, el repositorio principal reside en la cuenta de GitHub de Pablo Zuil Armenteros. Este repositorio centralizado contiene tres ramas principales:

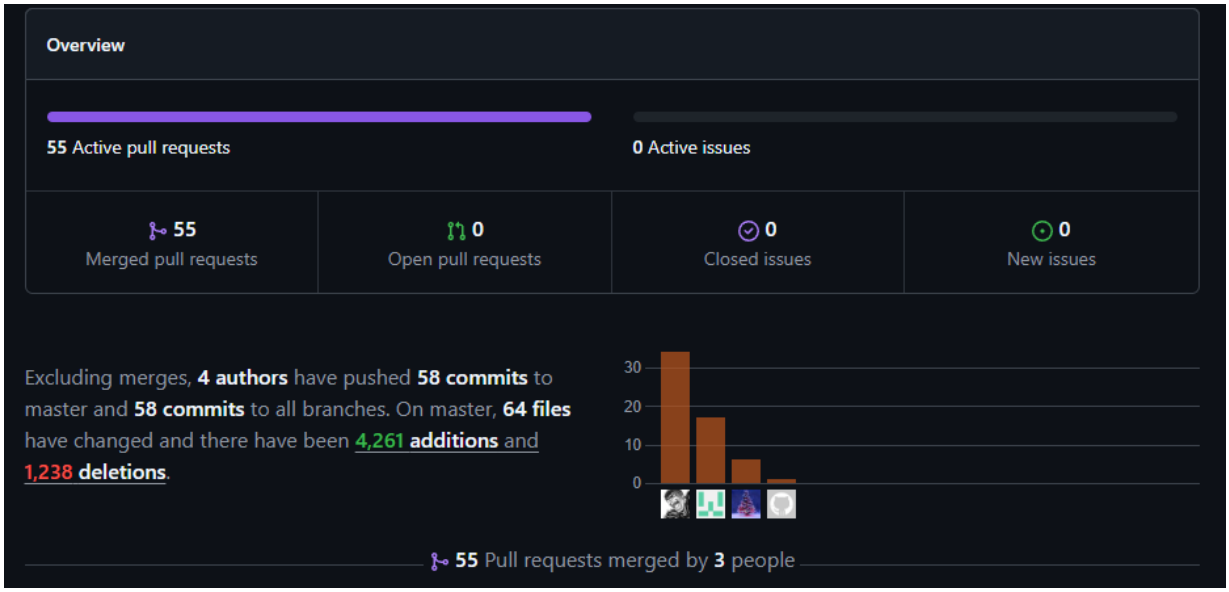
- **master:** Esta rama representa la versión estable del proyecto, lista para su despliegue.
- **dev:** Esta rama actúa como la rama de integración principal, donde se fusionan las contribuciones de todos los miembros del equipo.
- **dev-PabloZuil:** Esta rama es la rama personal de desarrollo de Pablo Zuil. Aquí es donde Pablo Zuil realiza sus propios cambios y pruebas antes de integrarlos en la rama dev.

El flujo de trabajo seguido por los miembros del equipo (Lucía Fuertes Cruz y Pablo De la Fuente De los Santos) es el siguiente:

1. **Creación de un *fork*:** Cada miembro del equipo (Lucía y Pablo De la Fuente) creó un *fork* del repositorio de Pablo Zuil en su propia cuenta de GitHub. Un *fork* es una copia del repositorio que reside en la cuenta personal del desarrollador y le permite trabajar de forma independiente sin afectar el repositorio original.
2. **Creación de una rama personal:** Dentro de su *fork*, cada miembro del equipo creó una rama de desarrollo personal, llamada *dev-LuciaFuertes* y *dev-PabloDLF* respectivamente.
3. **Desarrollo local:** Cada miembro del equipo realizó sus cambios en su rama personal (*dev-LuciaFuertes o dev-PabloDLF*) utilizando su entorno de desarrollo (IntelliJ IDEA). Se realizaron *commits* regulares con mensajes descriptivos para registrar los cambios realizados.
4. **Creación de un *Pull Request* a dev-PabloZuil:** Una vez que un miembro del equipo completaba una tarea o funcionalidad, creaba un *Pull Request* desde su rama

personal (dev-LuciaFuertes o dev-PabloDLF) hacia la rama dev-PabloZuil del repositorio de Pablo Zuil.

5. **Revisión y fusión en dev-PabloZuil:** Pablo Zuil revisaba el código del *Pull Request* y, si lo aprobaba, lo fusionaba en su rama dev-PabloZuil.
6. **Sincronización con dev:** Posteriormente, Pablo Zuil creaba un *Pull Request* desde su rama dev-PabloZuil hacia la rama dev del repositorio principal. Después de la revisión y aprobación, los cambios se fusionaban en la rama dev.



Este flujo de trabajo permitió a cada miembro del equipo trabajar de forma independiente en su propio entorno, garantizando la calidad del código mediante revisiones y facilitando la integración de las diferentes contribuciones en una única versión del proyecto.

En caso de conflictos durante la fusión de ramas, se resolvieron mediante la comunicación y colaboración entre los miembros del equipo, analizando las diferentes versiones del código y determinando la mejor forma de resolver los conflictos, siempre priorizando la integridad y el correcto funcionamiento del programa.

Requisitos del Programa

Para asegurar el correcto funcionamiento del programa de gestión del club "New Team", se han definido una serie de requisitos que se dividen en tres categorías principales: requisitos funcionales, requisitos no funcionales y requisitos de información.

Requisitos Funcionales

Los requisitos funcionales describen las acciones o tareas que el programa debe ser capaz de realizar. En este proyecto, los principales requisitos funcionales son:

ID	Descripción
RF1	**Gestión de Personal:** El programa debe permitir almacenar, modificar y eliminar información de cualquier miembro del club (jugadores y entrenadores).
RF2	**Estructura de Datos:** El programa debe utilizar una estructura de datos que incluya un identificador único, nombre, apellidos, fecha de nacimiento, fecha de incorporación, salario y país de origen para cada miembro del personal.
RF3	**Información Adicional de Entrenadores:** Para cada entrenador, el programa debe permitir registrar su área de especialización (entrenador principal, entrenador asistente, entrenador de porteros).
RF4	**Información Adicional de Jugadores:** Para cada jugador, el programa debe permitir registrar su posición en el campo (portero, defensa, centrocampista, delantero), número de dorsal, altura, peso, número de goles y partidos jugados.
RF5	**Caché LRU:** El programa debe implementar una caché LRU de máximo 5 elementos para gestionar el personal, optimizando el acceso a los datos más frecuentes.
RF6	**Validación de Datos:** El programa debe validar los datos ingresados para evitar errores y asegurar la integridad de la información.
RF7	**Importación/Exportación de Datos:** El programa debe permitir importar y exportar datos en formatos CSV, ~~XML~~, JSON y ~~binario~~.
RF8	**Menú de Opciones:** El programa debe proporcionar un menú con las siguientes opciones: Cargar datos desde fichero, Crear miembro del equipo, Actualizar miembro del equipo, Eliminar miembro del equipo, Copiar datos a fichero y Realizar consultas.
RF9	**Consultas:** El programa debe permitir realizar las siguientes consultas: Listados de personal agrupados por entrenadores y jugadores, El delantero más alto, Media de goles de los delanteros, etc. (Ver lista completa en el PDF proporcionado por el profesor).

Requisitos No Funcionales

Los requisitos no funcionales definen las características de calidad del programa, como el rendimiento, la seguridad, la usabilidad y la mantenibilidad. En este proyecto, los principales requisitos no funcionales son:

ID	Descripción
RNF1	**Rendimiento:** El programa debe responder a las consultas en un tiempo razonable.
RNF2	**Usabilidad:** El programa debe ser fácil de usar e intuitivo.
RNF3	**Mantenibilidad:** El código debe ser fácil de entender y modificar.

Requisitos de Información

Los requisitos de información definen los datos que el programa necesita gestionar. En este proyecto, los principales datos que el programa debe gestionar son:

- **Datos de Personal:** Identificador único, nombre, apellidos, fecha de nacimiento, fecha de incorporación, salario, país de origen, rol (jugador o entrenador).
- **Datos de Entrenadores:** Área de especialización.
- **Datos de Jugadores:** Posición, número de camiseta, altura, peso, número de goles anotados, partidos jugados.

Arquitectura y Tecnologías

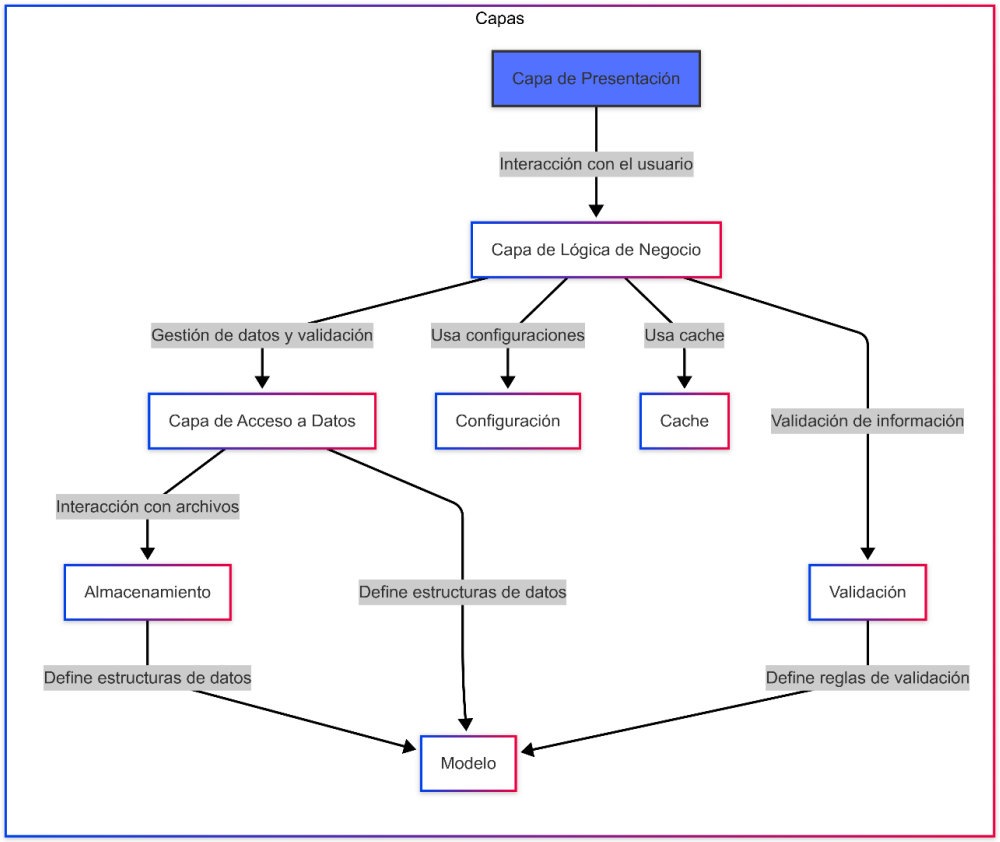
En esta sección, se describen la arquitectura general del programa y las tecnologías que se utilizaron para su desarrollo.

Arquitectura del Programa

El programa de gestión del club "New Team" se ha diseñado siguiendo una arquitectura en capas, que permite separar las diferentes responsabilidades del programa en módulos independientes. Esta arquitectura facilita el desarrollo, la prueba y el mantenimiento del mismo.

El programa se divide en las siguientes capas principales:

- **Capa de Presentación:** Esta capa se encarga de la interacción con el usuario. Proporciona la interfaz para ingresar y mostrar los datos, así como para ejecutar las diferentes acciones del programa.
- **Capa de Lógica de Negocio:** Esta capa contiene la lógica principal del programa, incluyendo la gestión de los datos, la validación de la información y la implementación de las consultas.
- **Capa de Acceso a Datos:** Esta capa se encarga de la interacción con la base de datos (o los ficheros) donde se almacenan los datos del programa. Proporciona las funciones necesarias para leer, escribir, actualizar y eliminar los datos.



Tecnologías Utilizadas

Para el desarrollo de este proyecto, se seleccionaron las siguientes tecnologías:

- **Lenguaje de Programación:** Kotlin
- **Entorno de Desarrollo:** IntelliJ IDEA
- **Sistema de Control de Versiones:** Git
- **Herramienta de Gestión de Dependencias:** Gradle
- **Librerías:**
    - ***org.lighthousegames:logging:1.5.0*:**

      Para implementar el sistema de *logs* de la aplicación.
    - ***ch.qos.logback:logback-classic:1.5.12*:**

      Implementación de la librería *Logger*.
    - ***org.jetbrains.dokka:dokka-gradle-plugin:2.0.0*:**

      Para la generación automática de la documentación del código.
    - ***org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2*:**

      Para la serialización y deserialización de datos en formato *JSON*.

- ***io.github.pdvrieze.xmlutil:serialization-jvm:0.90.3*:**

  Para la serialización y deserialización de datos en formato *XML*.

- ***io.mockk:mockk:1.13.16***

  Para las pruebas y '*testeo*' del programa.

Justificación de las Elecciones Tecnológicas

A continuación, se justifica la elección de cada una de las tecnologías utilizadas en este proyecto:

- **Kotlin:** Se trabajó Kotlin como lenguaje de programación debido a que lo estamos dando en clase. Pero también gracias a su sintaxis sencilla, su seguridad (especialmente en el manejo de nulos) y su compatibilidad con Java. Kotlin permite escribir código más legible, mantenible y limpio, lo que facilita el desarrollo y la prueba del programa.
  - **Pros:** Sintaxis moderna y concisa, seguridad en el manejo de nulos, compatibilidad con Java.
  - **Contras:** Curva de aprendizaje inicial, menor comunidad que Java.

- **IntelliJ IDEA:** Se eligió IntelliJ IDEA como entorno de desarrollo porque lo estamos utilizando en clase. Sin embargo, es preciso recordar sus potentes herramientas de edición, depuración y prueba de código. IntelliJ IDEA facilita el desarrollo en *Kotlin* y proporciona integración con *Git* y *Gradle*.
  - **Pros:** Potentes herramientas de edición y depuración, integración con *Git* y *Gradle*.
  - **Contras:** Requiere una licencia de pago para acceder a todas las funcionalidades.

- **Git:** Se seleccionó Git como sistema de control de versiones porque lo estamos viendo tanto en el módulo de Entornos de Desarrollo con Carmen de Jesús Aguilar como en el de Programación con José Luis González Sánchez. De todos modos, es importante poner de relieve su capacidad para gestionar de forma eficiente los cambios en el código, facilitar la colaboración entre los miembros del equipo y mantener un historial completo de las modificaciones realizadas.
  - **Pros:** Gestión eficiente de versiones, facilidad para la colaboración, historial completo de cambios.
  - **Contras:** Requiere un aprendizaje inicial intenso y crucial para comprender su funcionamiento.

- **Gradle:** Se eligió Gradle como herramienta de gestión de dependencias porque lo utilizamos en el módulo de Programación con José Luis González Sánchez. Sin embargo, es crucial resaltar su flexibilidad y su capacidad para automatizar la compilación, prueba y despliegue del programa. Gradle permite gestionar fácilmente las librerías externas que utiliza el proyecto.
    - **Pros:** Flexibilidad, automatización de tareas, gestión sencilla de dependencias.
    - **Contras:** Requiere un aprendizaje inicial para configurar el *build.gradle*.

- **org.lighthousegames:logging:1.5.0 & ch.qos.logback:logback-classic:1.5.12:** Se utilizaron estas librerías para implementar un sistema de logs robusto que permite registrar la actividad del programa, facilitando la depuración y el seguimiento de errores.
    - **Pros:** Permite registrar la actividad del programa, facilita la depuración y el seguimiento de errores.
    - **Contras:** Requiere una configuración inicial para definir los niveles de log y los destinos de los logs.

- **org.jetbrains.dokka:dokka-gradle-plugin:2.0.0:** Se utilizó Dokka para generar automáticamente la documentación del código a partir de los comentarios en el código fuente. Esto facilita la creación de documentación actualizada y consistente.
    - **Pros:** Genera documentación automática a partir de los comentarios, facilita la creación de documentación actualizada.
    - **Contras:** Requiere que el código esté correctamente documentado con comentarios.

- **org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2:** Se utilizó esta librería para facilitar la serialización y deserialización de datos en formato *JSON*. Esto permite leer y escribir datos en formato *JSON* de forma sencilla y eficiente.
    - **Pros:** Facilita la serialización y deserialización de datos en formato *JSON*, mejora la eficiencia en el manejo de datos del mismo.
    - **Contras:** Requiere el uso de anotaciones específicas en las clases para indicar cómo se deben serializar y deserializar los datos.

- **io.github.pdvrieze.xmlutil:serialization-jvm:0.90.3:** Se utilizó esta librería para facilitar la serialización y deserialización de datos en formato *XML*. Esto permite leer y escribir datos en formato *XML* de forma sencilla y eficiente.
    - **Pros:** Facilita la serialización y deserialización de datos en formato *XML*, mejora la eficiencia en el manejo de datos *XML*.
    - **Contras:** Requiere el uso de anotaciones específicas en las clases para indicar cómo se deben serializar y deserializar los datos.

En resumen, las tecnologías seleccionadas las hemos visto en el módulo de Programación con José Luis González Sánchez. Y, por lo tanto, las tenemos prácticamente todos a la orden del día.

Implementación Clave

En esta sección, se describen los aspectos más relevantes de la implementación del programa de gestión del club "New Team", destacando las soluciones adoptadas para los principales desafíos técnicos.

1. Gestión de Personal y Estructura de Datos

Para representar a los miembros del club (jugadores y entrenadores), se definieron las siguientes clases:

```
class Jugador(
    id: Long,
    nombre: String,
    apellidos: String,
    fechaNacimiento: String,
    fechaIncorporacion: String,
    salario: Double,
    pais: String,
    rol: String,
    var posicion: Posicion,
    var dorsal: Int,
    var altura: Double,
    var peso: Double,
    var goles: Int,
    var partidosJugados: Int
): Personal(id, nombre, apellidos, fechaNacimiento,
fechaIncorporacion, salario, pais, rol) {

    override fun toString(): String {
        return ("Jugador(id=$id, nombre=$nombre, apellidos=$apellidos,
fechaNacimiento=$fechaNacimiento,
```

```
fechaIncorporacion=$fechaIncorporacion, salario=$salario, pais=$pais,
posiciom=$posicion, dorsal=$dorsal, altura=$altura, peso=$peso,
goles=$goles, partidosJugados=$partidosJugados )")
    }

    override fun copy(
        id: Long,
        nombre: String,
        apellidos: String,
        fechaNacimiento: String,
        fechaIncorporacion: String,
        salario: Double,
        pais: String,
        rol: String,
    ): Personal {
        return Jugador(id, nombre, apellidos, fechaNacimiento,
fechaIncorporacion, salario, pais, rol, posicion, dorsal, altura,
peso, goles, partidosJugados)
    }

    @Serializable
    enum class Posicion {
        @SerialName("posicion")
        DEFENSA, CENTROCAMPISTA, DELANTERO, PORTERO, NINGUNO
    }
}
```

Estas clases permiten almacenar la información de cada miembro del club de forma organizada y estructurada. Se utilizó la herencia para evitar la duplicación de código, definiendo una clase base Personal con los atributos comunes a todos los miembros del club, y clases derivadas Entrenador y Jugador con los atributos específicos de cada rol.

2. Servicio con Caché LRU

Para gestionar el personal del club de forma eficiente, se implementó un servicio con caché LRU (Least Recently Used). La caché LRU permite almacenar los elementos más recientemente utilizados, optimizando el acceso a los datos más frecuentes.

```
private const val CACHE_SIZE = 6

//Implementacion de la interfaz PersonalService desarrollando sus
funciones.
class PersonalServiceImpl (

    //Define el tamaño de la cache, el storage, y el repository a
utilizar.
    private val cache : Cache<String, Personal> =
CacheImpl(CACHE_SIZE),
    private val storage: PersonalStorage = PersonalStorageImpl(),
    private val repository : PersonalRepository<Personal> =
```

```
PersonalRepositoryImpl()
) : PersonalService {

    //Logger
    private val logger = logging()

    //Lee el arhivo en el formato indicado
    override fun readFile(filepath: String, format: FileFormat):
List<Personal> {
        logger.info { "Leyendo personal del fichero" }
        return storage.readFile(File(filepath), format)
    }

    //Escribe una lista de objetos de personal en el sitio indicado y
con el formato indicado
    override fun writeFile(filepath: String,format: FileFormat
,personal: List<Personal>) {
        logger.info { "Sobreescribiendo personal del fichero" }
        return storage.writeFile(File(filepath), format, personal)
    }

    //Carga los datos de un archivo en una memoria temporal
    override fun importFile(filePath: String, format: FileFormat) {
        logger.info { "Importando personal del fichero" }
        val personal = readFile(filePath, format)
        personal.forEach {
            repository.save(it)
        }
    }

    //Exporta los que se encuentra de personal a un archivo
    override fun exportFile(filePath: String , fileFormat: FileFormat)
{
        writeFile(filePath, fileFormat, repository.getAll())
    }

    //Muesta todos los registros
    override fun getAll(): List<Personal> {
        return repository.getAll()
    }

    //Busca un registo basandose en el id proporcionado
    override fun getById(id: Long): Personal {
        logger.info { "Obteniendo personal: $id" }
        return cache.get(id.toString())!!

    }

    //Guarda una nueva entidad de personal
    override fun save(personal: Personal): Personal {
        logger.info { "Guardando personal: $personal" }
        return repository.save(personal)
    }
```

```
    //Actualiza el registro en el respoitorio
    override fun update(id: Long, personal: Personal): Personal {
        logger.info { "actualizando personal: $personal" }
        return repository.update(id, personal) ?.also {
cache.remove(id.toString()) }!!
    }

    //Elimina una entidad en base al id obtenido
    override fun delete(id: Long): Personal {
        logger.info { "borrando personal: $id" }
        return repository.delete(id)!!
    }

}
```

La clase *PersonalServiceImpl* implementa el servicio de gestión de personal utilizando una caché LRU para optimizar el acceso a los datos más frecuentemente utilizados. Esta clase define métodos para leer, escribir, importar, exportar, obtener, guardar, actualizar y eliminar datos del personal, interactuando con el repositorio y el almacenamiento de datos. La caché LRU se utiliza para almacenar los elementos más recientemente utilizados, mejorando así la eficiencia en el acceso a los datos.

3. Validación de Datos

Para asegurar la integridad de los datos, se implementaron mecanismos de validación en el programa. Se verifican diferentes aspectos de los datos ingresados, como el formato de las fechas, el rango de los salarios, la validez de las posiciones de los jugadores, etc.

```
class JugadorValidator {

    val logger = logging()

    fun validateJugador(jugador: Jugador) {

        logger.debug { "Validando jugadores" }

        // Validación de nombre
        if (jugador.nombre.isBlank()) {
            throw exceptions.JugadorValidatorException("El nombre no
puede estar en blanco")
        }
        if (jugador.nombre.length !in 1..15) {
            throw exceptions.JugadorValidatorException("El nombre no
puede exceder los 15 caracteres")
        }

        // Validación de apellidos
        if (jugador.apellidos.isBlank()) {
```

```
            throw exceptions.JugadorValidatorException("El apellido no
puede estar en blanco")
        }
        if (jugador.apellidos.length !in 1..30) {
            throw exceptions.JugadorValidatorException("El apellido no
puede exceder los 30 caracteres")
        }

        // Validación de fecha de nacimiento
        if (jugador.fechaNacimiento.isBlank()) {
            throw exceptions.JugadorValidatorException("La fecha de
nacimiento no puede estar en blanco")
        }
        if (jugador.fechaNacimiento <= "1925-01-01") {
            throw exceptions.JugadorValidatorException("La fecha de
nacimiento no puede ser anterior a 1925")
        }

        // Validación de fecha de incorporación
        if (jugador.fechaIncorporacion.isBlank()) {
            throw exceptions.JugadorValidatorException("La fecha de
incorporación no puede estar en blanco")
        }
        if (jugador.fechaIncorporacion <= "1960-01-01") {
            throw exceptions.JugadorValidatorException("La fecha de
incorporación no puede ser anterior a 1960")
        }

        // Validación de salario
        if (jugador.salario!!.isNaN()) {
            throw exceptions.JugadorValidatorException("El salario no
puede ser nulo")
        }
        if (jugador.salario!! < 0) {
            throw exceptions.JugadorValidatorException("El salario no
puede ser negativo")
        }

        // Validación de país
        if (jugador.pais.isBlank()) {
            throw exceptions.JugadorValidatorException("El país no
puede estar en blanco")
        }

        // Validación de rol
        if (jugador.rol.isBlank()) {
            throw exceptions.JugadorValidatorException("El rol no
puede estar en blanco")
        }

        // Validación de posición
        if (jugador.posicion == null) {
            throw exceptions.JugadorValidatorException("Posición no
```

```
puede ser nula")
        }

        // Validación de dorsal
        if (jugador.dorsal == null) {
            throw exceptions.JugadorValidatorException("El dorsal no
puede ser nulo")
        }
        if (jugador.dorsal !in 1..25) {
            throw exceptions.JugadorValidatorException("El dorsal debe
estar entre 1 y 25")
        }

        // Validación de altura
        if (jugador.altura!!.isNaN()) {
            throw exceptions.JugadorValidatorException("La altura no
puede ser nula")
        }
        if (jugador.altura!! !in 0.0..2.5) {
            throw exceptions.JugadorValidatorException("La altura debe
estar entre 0.0 y 2.5 metros")
        }

        // Validación de peso
        if (jugador.peso!!.isNaN()) {
            throw exceptions.JugadorValidatorException("El peso no
puede ser nulo")
        }
        if (jugador.peso!! !in 1.00..90.00) {
            throw exceptions.JugadorValidatorException("El peso debe
estar entre 1.00 y 90.00 kg")
        }

        // Validación de goles
        if (jugador.goles < 0) {
            throw exceptions.JugadorValidatorException("El número de
goles no puede ser negativo")
        }

        // Validación de partidos jugados
        if (jugador.partidosJugados < 0) {
            throw exceptions.JugadorValidatorException("El número de
partidos jugados no puede ser negativo")
        }
    }
}
```

El código de *JugadorValidator* asegura la integridad de los datos de los jugadores mediante diversas validaciones. Se verifica que los nombres y apellidos no estén en blanco y cumplan con la longitud adecuada, que las fechas de nacimiento e incorporación sean

válidas, que los salarios no sean negativos, y que los parámetros específicos como la posición, dorsal, altura, peso y goles sean razonables y dentro de un rango aceptable.

4. Importación/Exportación de Datos

El programa permite importar y exportar datos en formatos CSV y JSON, ya que no logramos alcanzar el objetivo con el XML y el Binario. Para ello, se utilizaron las librerías kotlinx-serialization-json e io.github.pdvrieze.xmlutil:serialization-jvm para la serialización y deserialización de datos en formatos JSON y XML, respectivamente.

Ejemplo en CSV

```
class PersonalStorageCsv : PersonalStorageFile {

    //Implementación del logger
    private val logger = logging()
    init {
        logger.debug { "Iniciando almacenamiento en CSV" }
    }

    //Lee el fichero y lo transforma a una lista del tipo PersonalDto
    override fun readFile(file: File): List<Personal> {
        logger.debug { "Leyendo fichero CSV" }

        //Filtra errores de lectura del archivo
        if (!file.exists() || !file.isFile || !file.canRead() ||
!file.canRead() || file.length() == 0L || !file.name.endsWith(".csv"))
{
            logger.error { "El fichero no existe o no se puede leer:
$file" }
            throw exceptions.PersonalStorageException("El fichero no
existe o no se puede leer")
        }

        //Crea una lista del tipo Personal Dto con todos sus datos,
saltandose la cabecera, separando por la comas,
        //elimina los espacios en blanco del string y por ultimo la
convierte a model.
        val lista = file.readLines()
            .drop(1)
            .map { it.split(",") }
            .map { it.map { it.trim() } }
            .map{
                PersonalDto(
                    id = it[0].toLong(),
                    nombre = it[1],
                    apellidos = it[2],
                    fecha_nacimiento = it[3],
                    fecha_incorporacion = it[4],
                    salario = it[5].toDouble(),
                    pais = it[6],
```

```
                    rol = it[7],
                    especialidad = it[8],
                    posicion = it[9],
                    dorsal = it[10].toIntOrNull(),
                    altura = it[11].toDoubleOrNull(),
                    peso = it[12].toDoubleOrNull(),
                    goles = it[13].toIntOrNull(),
                    partidos_jugados = it[14].toIntOrNull(),
                ).toModel()
            }
        return lista
    }

    //Sobreescribe la lista de personal añadiendo entradas.
    override fun writeFile(personal: List<Personal>, file: File) {

        //Logger
        logger.debug { "Escribiendo fichero CSV" }

        //Filtra errores de lectura del archivo
        if (!file.parentFile.exists() || !file.parentFile.isDirectory
|| !file.name.endsWith(".csv")) {
            logger.error { "El directorio padre del fichero no se
encuentra o no existe" }
            throw exceptions.PersonalStorageCsv("El directorio padre
no existe")
        }

        //Sobreescribe con los datos proporcionados la lista del tipo
personal dependiendo de si es jugador o entrenador
        personal.forEach {
            when (it) {
                is Jugador -> file.appendText("${it.id}, ${it.nombre},
${it.apellidos}, ${it.fechaNacimiento}, ${it.fechaIncorporacion},
${it.salario}, ${it.pais}, ${it.rol}, ${it.posicion}, ${it.dorsal},
${it.altura}, ${it.peso}, ${it.goles}, ${it.partidosJugados} /n")
                is Entrenador -> file.appendText("${it.id},
${it.nombre}, ${it.apellidos}, ${it.fechaNacimiento},
${it.fechaIncorporacion}, ${it.salario}, ${it.pais}, ${it.rol},
${it.especialidad} /n")
            }
        }
    }
}
```

Ejemplo en JSON

```
class PersonalStorageJson : PersonalStorageFile {

    // implementacion del logger
    val logger = logging()
```

```
    init {
        logger.debug { "inicializando PersonalStorageJson" }
    }
    // Lee el archivo json y lo transforma a una cadena
    override fun readFile(file: File): List<Personal> {
        println()
        logger.debug { "Leyendo JSON" }

        if (!file.exists() || !file.isFile || !file.canRead()) {
            throw exceptions.PersonalStorageException("El fichero no
se puede leer, no es un fichero o no se ha encontrado")
        } else {
            val json = Json { ignoreUnknownKeys = true; prettyPrint =
true }
            val imprimirJson = file.readText()
            val listaPersonalDto =
json.decodeFromString<List<PersonalDto>>(imprimirJson)

            val listaPersonalModel = listaPersonalDto.map {
                    when (it.rol) {
                        "Entrenador" -> it.toEntrenador()
                         else -> it.toJugador()
                    }
            }
            return listaPersonalModel
        }
    }

    // Escribe en el archivo json
    override fun writeFile(personal: List<Personal>, file: File) {
        if (!file.parentFile.exists() || !file.parentFile.isDirectory
|| !file.canWrite()) {
            throw exceptions.PersonalStorageException("El fichero json
no se puede sobreescribir o no existe en su directorio padre")
        } else {
            val json = Json { ignoreUnknownKeys = true; prettyPrint =
true }
            val listaPersonalDto = personal.map {
                when (it) {
                    is Jugador -> { it.toDto() }
                    is Entrenador -> { it.toDto() }
                    else -> null
                    }
                }

            val jsonString = json.encodeToString(listaPersonalDto)
            file.writeText(jsonString)
            }
        }
    }
```

El código de *PersonalStorageJson* y *PersonalStorageCsv* proporciona métodos para leer y escribir datos en los formatos JSON y CSV, respectivamente. En el caso de JSON, se utiliza la librería *kotlinx-serialization-json* para serializar y deserializar los datos. En el caso de CSV, se leen y escriben las líneas del archivo manualmente, separando los campos por comas. Ambos métodos aseguran que los datos sean correctamente procesados y almacenados en los archivos correspondientes.

5. Principios SOLID

En este proyecto, se han implementado los principios SOLID para garantizar un código más limpio, mantenible y escalable. Los principios SOLID son un conjunto de cinco recomendaciones de diseño orientadas a objetos que promueven una estructura de código más robusta y flexible. A continuación, se describen los principios SOLID y se proporcionan ejemplos específicos de su aplicación en el proyecto:

Principio de Responsabilidad Única (SRP)

Cada clase debe tener una única responsabilidad.

Ejemplo:

```
class PersonalServiceImpl (
    private val cache : Cache<String, Personal> =
CacheImpl(CACHE_SIZE),
    private val storage: PersonalStorage = PersonalStorageImpl(),
    private val repository : PersonalRepository<Personal> =
PersonalRepositoryImpl()
) : PersonalService {}
```

La clase *PersonalServiceImpl* se encarga exclusivamente de la gestión del personal.

Principio Abierto/Cerrado (OCP)

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

Ejemplo:

```
interface PersonalStorageFile {
    fun readFile(file : File) : List<Personal>
    fun writeFile (personal: List<Personal>, file: File)
}
```

La interfaz *PersonalStorageFile* permite la extensión mediante la implementación de nuevas clases para diferentes formatos de archivo sin modificar la interfaz.

Principio de Sustitución de Liskov (LSP)

Los objetos de una clase derivada deben sustituir a los objetos de la clase base sin alterar el funcionamiento del programa.

Ejemplo:

```
class Jugador(
    id: Long,
    nombre: String,
    apellidos: String,
    fechaNacimiento: String,
    fechaIncorporacion: String,
    salario: Double?,
    pais: String,
    rol: String,
    var posicion: Posicion?,
    var dorsal: Int?,
    var altura: Double?,
    var peso: Double?,
    var goles: Int,
    var partidosJugados: Int
): Personal(id, nombre, apellidos, fechaNacimiento,
fechaIncorporacion, salario, pais, rol) {}
```

La clase *Jugador* extiende a *Personal*, permitiendo la sustitución de objetos de *Personal* por objetos de *Jugador*.

Principio de Segregación de Interfaces (ISP)

Las clases o componentes que dependan de una interfaz, no deben estar obligados a depender de interfaces que no utilizan.

Ejemplo:

```
interface CrudRepository<T, ID> {
    fun getAll(): List<T>
    fun getById(id: ID): T?
    fun save(item: T): T
    fun update(id: ID,item: T): T?
    fun delete(id: ID): T?
}
```

La interfaz *CrudRepository* está segregada en operaciones *CRUD* específicas, permitiendo que las funciones implementen solo las funcionalidades necesarias.

Principio de Inversión de Dependencias (DIP)

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

Ejemplo:

```
class PersonalServiceImpl (
    private val cache : Cache<String, Personal> =
CacheImpl(CACHE_SIZE),
    private val storage: PersonalStorage = PersonalStorageImpl(),
    private val repository : PersonalRepository<Personal> =
PersonalRepositoryImpl()
) : PersonalService {}
```

PersonalServiceImpl depende de abstracciones (*Cache*, *PersonalStorage*, *PersonalRepository*) en lugar de dependencias concretas.

Ejemplo de Consulta: El Delantero Más Alto

Para obtener el delantero más alto del club, se implementó la siguiente consulta:

```
fun consultaDelanteroMasAlto(personalRepository:
PersonalRepositoryImpl) {
    val delanteroMasAlto =
personalRepository.getAll().filterIsInstance<Jugador>().filter {
it.posicion == Jugador.Posicion.DELANTERO }.maxByOrNull { it.altura!!
}
    println("El delantero más alto es: $delanteroMasAlto")
}
```

1. Obtención de Todos los Miembros: La función llama a *personalRepository.getAll()* para obtener una lista de todos los miembros del personal.

2. Filtrado por Jugadores: Utiliza *filterIsInstance<Jugador>()* para filtrar y quedarse solo con los objetos de tipo *Jugador*.

3. Filtrado por Posición: Filtra la lista de jugadores para quedarse solo con aquellos cuya posición es *DELANTERO*.

4. Encontrar el Más Alto: Utiliza *maxByOrNull { it.altura!! }* para encontrar el delantero más alto en la lista filtrada.

5. Impresión del Resultado: Finalmente, imprime el delantero más alto encontrado.

Consultas Destacadas

En esta sección, se presentan algunas de las consultas más relevantes implementadas en el programa de gestión del club "New Team". Estas consultas permiten obtener información valiosa sobre los miembros del club, facilitando la toma de decisiones y la gestión eficiente de la plantilla.

Para cada consulta, se proporciona una descripción de su objetivo, el código Kotlin utilizado para implementarla y un ejemplo de la salida que produce.

1. Listados de Personal Agrupados por Entrenadores y Jugadores

Objetivo: Obtener un listado de todos los miembros del club, separados en dos categorías: entrenadores y jugadores. Esto permite tener una visión general de la composición del equipo técnico y de la plantilla de jugadores.

```
fun consultaListadosPersonal(personalRepository:
PersonalRepositoryImpl) {
    val jugadores =
personalRepository.getAll().filterIsInstance<Jugador>()
    val entrenadores =
personalRepository.getAll().filterIsInstance<Entrenador>()
    println("Jugadores:")
    jugadores.forEach { println(it) }
    println("Entrenadores:")
    entrenadores.forEach { println(it) }
}
```

Ejemplo de Salida:

```
Entrenadores:
Entrenador(id=1, nombre=Juan, apellidos=Pérez, ...)
Entrenador(id=2, nombre=María, apellidos=González, ...)

Jugadores:
Jugador(id=1, nombre=Carlos, apellidos=Martínez, ...)
Jugador(id=2, nombre=Luis, apellidos=Ramírez, ...)
```

2. El Delantero Más Alto

Objetivo: Identificar al jugador que ocupa la posición de delantero y tiene la mayor altura dentro del club. Esta consulta puede ser útil para estrategias de juego que requieran jugadores altos en la delantera.

```
fun consultaListadosPersonal(personalRepository:
PersonalRepositoryImpl) {
    val jugadores =
personalRepository.getAll().filterIsInstance<Jugador>()
```

```
    val entrenadores =
personalRepository.getAll().filterIsInstance<Entrenador>()
    println("Jugadores:")
    jugadores.forEach { println(it) }
    println("Entrenadores:")
    entrenadores.forEach { println(it) }
}
```

Ejemplo de Salida:

```
Entrenadores:

Entrenador(id=1, nombre=Juan, apellidos=Pérez, fechaNacimiento=1970-
05-10, fechaIncorporacion=2015-08-01, salario=50000.0, pais=España,
especialidad=Entrenador Principal)
Entrenador(id=2, nombre=María, apellidos=González,
fechaNacimiento=1980-07-15, fechaIncorporacion=2018-06-10,
salario=45000.0, pais=Argentina, especialidad=Entrenador Asistente)

Jugadores:

Jugador(id=1, nombre=Carlos, apellidos=Martínez, fechaNacimiento=1990-
01-20, fechaIncorporacion=2010-03-15, salario=35000.0, pais=España,
posicion=Delantero, dorsal=9, altura=1.80, peso=75.0, goles=100,
partidosJugados=200)
Jugador(id=2, nombre=Luis, apellidos=Ramírez, fechaNacimiento=1992-04-
10, fechaIncorporacion=2012-07-20, salario=32000.0, pais=Brasil,
posicion=Defensa, dorsal=4, altura=1.85, peso=80.0, goles=10,
partidosJugados=150)
```

3. Media de Goles de los Delanteros

Objetivo: Calcular el promedio de goles anotados por todos los jugadores que ocupan la posición de delantero. Esta estadística puede ser útil para evaluar el rendimiento general de la delantera del equipo.

```
fun consultaMediaGolesDelanteros(personalRepository:
PersonalRepositoryImpl) {
    val delanteros =
personalRepository.getAll().filterIsInstance<Jugador>().filter {
it.posicion == Jugador.Posicion.DELANTERO }
    val mediaGoles = delanteros.map { it.goles }.average()
    println("La media de goles de los delanteros es: $mediaGoles")
}
```

Ejemplo de Salida:

```
El promedio de goles anotados por los delanteros es: 15.75
```

4. Defensa con Más Partidos Jugados

Objetivo: Encontrar al defensor que ha participado en la mayor cantidad de partidos. Esto permite identificar a los jugadores con más experiencia y regularidad en la defensa.

```
fun consultaDefensaMasPartidos(personalRepository:
PersonalRepositoryImpl) {
    val defensaMasPartidos =
personalRepository.getAll().filterIsInstance<Jugador>().filter {
it.posicion == Jugador.Posicion.DEFENSA }.maxByOrNull {
it.partidosJugados }
    println("El defensa con más partidos jugados es:
$defensaMasPartidos")
}
```

Ejemplo de Salida:

```
El defensa con más partidos jugados es: Jugador(id=2, nombre=Luis,
apellidos=Ramírez, fechaNacimiento=1992-04-10,
fechaIncorporacion=2012-07-20, salario=32000.0, pais=Brasil,
posicion=DEFENSA, dorsal=4, altura=1.85, peso=80.0, goles=10,
partidosJugados=150)
```

5. Jugadores Agrupados por su País de Origen

Objetivo: Organizar a los jugadores según su nacionalidad. Esto puede ser útil para analizar la diversidad cultural del equipo y para identificar posibles estrategias de reclutamiento en diferentes países.

```
fun consultaJugadoresPorPais(personalRepository:
PersonalRepositoryImpl) {
    val jugadoresPorPais =
personalRepository.getAll().filterIsInstance<Jugador>().groupBy {
it.pais }
    jugadoresPorPais.forEach { (pais, jugadores) ->
        println("País: $pais")
        jugadores.forEach { println(it) }
    }
}
```

Ejemplo de Salida:

```
País: España
Jugador(id=1, nombre=Carlos, apellidos=Martínez, fechaNacimiento=1990-
01-20, fechaIncorporacion=2010-03-15, salario=35000.0, pais=España,
```

```
posicion=Delantero, dorsal=9, altura=1.80, peso=75.0, goles=100,
partidosJugados=200)

País: Brasil
Jugador(id=2, nombre=Luis, apellidos=Ramírez, fechaNacimiento=1992-04-
10, fechaIncorporacion=2012-07-20, salario=32000.0, pais=Brasil,
posicion=Defensa, dorsal=4, altura=1.85, peso=80.0, goles=10,
partidosJugados=150)
```

[Tabla con resumen de las consultas]

Consulta	Objetivo	Código Kotlin	Ejemplo de Salida
Listados de Personal Agrupados	Visión general de la composición del equipo	`consultaListadosPersonal(personalRepository: PersonalRepositoryImpl) {}`	Listado de jugadores y entrenadores con sus datos
El Delantero Más Alto	Identificar jugadores altos en la delantera	`consultaDelanteroMasAlto(personalRepository: PersonalRepositoryImpl) {}`	Datos del jugador delantero más alto
Media de Goles de los Delanteros	Evaluar el rendimiento de la delantera	`consultaMediaGolesDelanteros(personalRepository: PersonalRepositoryImpl) {}`	Valor numérico del promedio de goles
Defensa con Más Partidos Jugados	Identificar jugadores con más experiencia en la defensa	`consultaDefensaMasPartidos(personalRepository: PersonalRepositoryImpl) {}`	Datos del jugador defensa con más partidos jugados
Jugadores Agrupados por su País de Origen	Analizar la diversidad cultural del equipo e identificar posibles estrategias de reclutamiento	`consultaJugadoresPorPais(personalRepository: PersonalRepositoryImpl) {}`	Listado de jugadores agrupados por país con sus datos

Pruebas y Validación

Para asegurar la calidad y el correcto funcionamiento del programa de gestión del club "New Team", se llevó a cabo un proceso de pruebas. Este proceso incluyó la realización de pruebas unitarias, pruebas de integración y pruebas de validación, con el objetivo de verificar que el programa cumple con los requisitos funcionales y no funcionales definidos.

Pruebas Unitarias

Las pruebas unitarias se centraron en verificar el correcto funcionamiento de cada uno de los componentes individuales del programa, como las clases para la gestión de personal, la *caché LRU* y las funciones de validación de datos.

Para la realización de las pruebas unitarias, se utilizó el *framework JUnit*, que proporciona las herramientas necesarias para escribir y ejecutar las pruebas de forma automatizada.

```
class JugadorValidatorTest {

    @Test
    fun JugadorValidateNombre() {
        val jugador = Jugador(
            id = 0L,
            nombre = "",
            apellidos = "Pérez",
            fechaNacimiento = "1990-01-01",
            fechaIncorporacion = "2010-01-01",
            salario = 1000.0,
            pais = "España",
            rol = "jugador",
            posicion = Jugador.Posicion.CENTROCAMPISTA,
            dorsal = 10,
            altura = 1.75,
            peso = 75.0,
            goles = 5,
            partidosJugados = 20
        )
        val exception =
assertThrows<exceptions.JugadorValidatorException> {
            JugadorValidator().validateJugador(jugador)
        }
        assertEquals("El nombre no puede estar en blanco",
exception.message)
    }

    @Test
    fun JugadorValidateNombreLargo() {
        val jugador = Jugador(
            id = 0L,
            nombre = "NombreMuyLargoQueExcede15Caracteres",
```

```
            apellidos = "Pérez",
            fechaNacimiento = "1990-01-01",
            fechaIncorporacion = "2010-01-01",
            salario = 1000.0,
            pais = "España",
            rol = "jugador",
            posicion = Jugador.Posicion.CENTROCAMPISTA,
            dorsal = 10,
            altura = 1.75,
            peso = 75.0,
            goles = 5,
            partidosJugados = 20
        )
        val exception =
assertThrows<exceptions.JugadorValidatorException> {
            JugadorValidator().validateJugador(jugador)
        }
        assertEquals("El nombre no puede exceder los 15 caracteres",
exception.message)
    }
...
```

Pruebas de Integración

No se han realizado pruebas de integración en el proyecto. Las pruebas de integración se centran en verificar la correcta interacción entre los diferentes componentes del programa, pero en este caso, solo se han llevado a cabo pruebas unitarias para validar el funcionamiento individual de cada componente.

Pruebas de Validación

Las pruebas de validación se centraron en verificar que el programa cumple con los requisitos funcionales y no funcionales definidos. Se probaron escenarios como la creación, modificación y eliminación de miembros del club, la ejecución de las consultas y la validación de los datos ingresados.

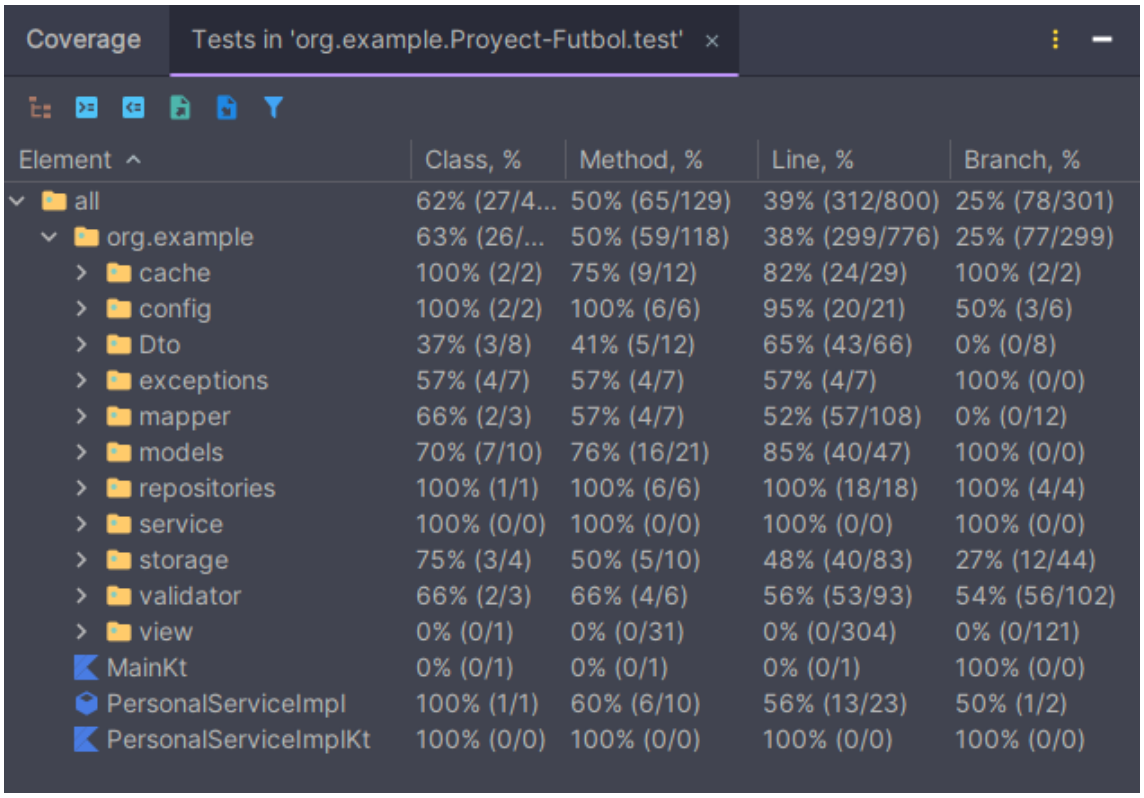
Caso de Prueba	Descripción	Datos de Entrada	Resultado Esperado
Creación de Jugador	Verificar que se puede crear un nuevo jugador con datos válidos	Nombre, Apellidos, Fecha de Nacimiento, etc.	El jugador se crea correctamente y se almacena en el repositorio
Modificación de Jugador	Verificar que se puede modificar los datos de un jugador existente	ID del Jugador, Nuevos Datos	Los datos del jugador se actualizan correctamente en el repositorio

Eliminación de Jugador	Verificar que se puede eliminar un jugador existente	ID del Jugador	El jugador se elimina correctamente del repositorio
Ejecución de Consulta	Verificar que se pueden ejecutar consultas sobre los datos de los jugadores	Tipo de Consulta, Parámetros de Consulta	La consulta devuelve los resultados esperados
Validación de Datos de Jugador	Verificar que se validan correctamente los datos ingresados para un jugador	Datos del Jugador	Se muestran mensajes de error apropiados para datos inválidos, y se aceptan datos válidos
Creación de Entrenador	Verificar que se puede crear un nuevo entrenador con datos válidos	Nombre, Apellidos, Fecha de Nacimiento, etc.	El entrenador se crea correctamente y se almacena en el repositorio
Modificación de Entrenador	Verificar que se puede modificar los datos de un entrenador existente	ID del Entrenador, Nuevos Datos	Los datos del entrenador se actualizan correctamente en el repositorio
Eliminación de Entrenador	Verificar que se puede eliminar un entrenador existente	ID del Entrenador	El entrenador se elimina correctamente del repositorio
Validación de Datos de Entrenador	Verificar que se validan correctamente los datos ingresados para un entrenador	Datos del Entrenador	Se muestran mensajes de error apropiados para datos inválidos, y se aceptan datos válidos

Cobertura del Código

Para medir la efectividad de las pruebas realizadas, se calculó la cobertura del código. La cobertura del código indica el porcentaje de líneas de código que son ejecutadas por las pruebas.

Se utilizó la herramienta *'Coverage'* para calcular la cobertura del código. Los resultados obtenidos muestran que el programa tiene una **cobertura media del 44%**, lo que indica que las pruebas realizadas cubren una parte significativa del código.



Element	Class, %	Method, %	Line, %	Branch, %
all	62% (27/4...	50% (65/129)	39% (312/800)	25% (78/301)
org.example	63% (26/...	50% (59/118)	38% (299/776)	25% (77/299)
cache	100% (2/2)	75% (9/12)	82% (24/29)	100% (2/2)
config	100% (2/2)	100% (6/6)	95% (20/21)	50% (3/6)
Dto	37% (3/8)	41% (5/12)	65% (43/66)	0% (0/8)
exceptions	57% (4/7)	57% (4/7)	57% (4/7)	100% (0/0)
mapper	66% (2/3)	57% (4/7)	52% (57/108)	0% (0/12)
models	70% (7/10)	76% (16/21)	85% (40/47)	100% (0/0)
repositories	100% (1/1)	100% (6/6)	100% (18/18)	100% (4/4)
service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
storage	75% (3/4)	50% (5/10)	48% (40/83)	27% (12/44)
validator	66% (2/3)	66% (4/6)	56% (53/93)	54% (56/102)
view	0% (0/1)	0% (0/31)	0% (0/304)	0% (0/121)
MainKt	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
PersonalServiceImpl	100% (1/1)	60% (6/10)	56% (13/23)	50% (1/2)
PersonalServiceImplKt	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

En resumen, el proceso de pruebas y validación permitió identificar y corregir errores en el programa, asegurando a medias su calidad y su correcto funcionamiento.

Presupuesto y Tiempos

En esta sección, se presenta una estimación del tiempo invertido en el desarrollo del proyecto y una aproximación del coste económico asociado. Es importante tener en cuenta que esta estimación se basa en las horas dedicadas por los miembros del equipo como estudiantes, y no refleja los costes que implicaría un desarrollo profesional a gran escala.

Estimación del Tiempo Invertido

Para estimar el tiempo invertido en el proyecto, se consideraron las diferentes fases del desarrollo, incluyendo la planificación, el diseño, la implementación, las pruebas y la documentación.

La siguiente tabla resume las horas estimadas y las horas reales dedicadas a cada fase:

Fase del Proyecto	Horas Estimadas	Horas Reales
Planificación	3	10
Diseño	15	15

Implementación	60	71
Pruebas	20	30
Documentación	15	24
Total	**120**	**150**

Como se puede observar, el tiempo real dedicado al proyecto superó ligeramente la estimación inicial, principalmente debido a la complejidad de algunas funcionalidades y a la necesidad de realizar pruebas exhaustivas y la creación de la documentación para asegurar la calidad del sistema. El total de horas dedicadas al proyecto fue de aproximadamente **150 horas** entre el 27 de febrero de 2025 y el 10 de marzo de 2025.

Estimación del Coste Económico

Para estimar el coste económico del proyecto, se consideró el valor de la hora de trabajo de cada miembro del equipo. Dado que se trata de estudiantes, se asignó un valor de 10€ por hora.

El coste total del proyecto se calcula multiplicando el número total de horas dedicadas por el valor de la hora de trabajo:

Coste Total = Horas Totales * Valor por Hora = **150 * 10 = 1500 €**

Por lo tanto, el coste estimado del proyecto es de **1500 €**.

Es importante destacar que este coste no incluye otros posibles gastos extraordinarios.

Consideraciones Adicionales

Es importante tener en cuenta que esta estimación es una aproximación y que el coste real del proyecto podría variar en función de diferentes factores, como la complejidad de las funcionalidades, la experiencia del equipo de desarrollo y las posibles incidencias que puedan surgir durante el desarrollo. En este caso, el valor se ha reducido notablemente al ser estudiantes y no profesionales en este sector.

Anexos

En esta sección, se incluyen aquellos elementos que complementan la información presentada en el documento principal, pero que no son esenciales para su comprensión. Estos anexos proporcionan detalles técnicos, ejemplos de configuración y otros recursos que pueden ser de utilidad para aquellos que deseen profundizar en el proyecto.

1. Configuración de Logging

Para facilitar la depuración y el seguimiento del sistema, se implementó un sistema de *logging* utilizando la librería *org.lighthousegames:logging:1.5.0* y *ch.qos.logback:logback-classic:1.5.12* en el *build.gradle.kts*.

2. Diagrama de Clases UML Detallado

Aunque en la sección de Arquitectura se presentó un diagrama de arquitectura general, en este anexo se incluye un diagrama de clases UML más detallado, que muestra las relaciones entre las diferentes clases del proyecto.

DiagramaUML_ProyectoNewTeam.svg

3. Ejemplo de Archivo de Datos (CSV, XML, JSON)

Para ilustrar el formato de los datos que el sistema es capaz de importar y exportar, se incluyen ejemplos de archivos en los formatos CSV, XML y JSON.

Ejemplo de archivo CSV

```
id,nombre,apellidos,fecha_nacimiento,fecha_incorporacion,salario,pais,
rol,especialidad,posicion,dorsal,altura,peso,goles,partidos_jugados
1,Roberto,Hongo,1960-07-17,2000-01-
01,60000.0,Brasil,Entrenador,ENTRENADOR_PRINCIPAL,,,,,,
2,Oliver,Atom,1983-04-10,2001-05-
15,35000.0,España,Jugador,,DELANTERO,10,1.75,65.0,70,150
3,Benji,Price,1983-11-07,2001-05-
15,34000.0,Alemania,Jugador,,PORTERO,1,1.83,78.0,0,200
4,Freddy,Marshall,1965-09-22,2005-04-
10,55000.0,España,Entrenador,ENTRENADOR_PORTEROS,,,,,,
5,Tom,Baker,1984-03-20,2001-05-
15,32000.0,Inglaterra,Jugador,,CENTROCAMPISTA,8,1.72,63.0,30,140
```

Ejemplo de archivo XML

```
<equipo>
    <personal id="31">
        <tipo>Jugador</tipo>
        <nombre>Diego</nombre>
        <apellidos>Martínez</apellidos>
        <fechaNacimiento>1997-11-20</fechaNacimiento>
        <fechaIncorporacion>2023-06-01</fechaIncorporacion>
        <salario>38000.0</salario>
        <pais>Argentina</pais>
        <especialidad/>
```

```xml
        <posicion>DEFENSA</posicion>
        <dorsal>5</dorsal>
        <altura>1.80</altura>
        <peso>75.0</peso>
        <goles>2</goles>
        <partidosJugados>50</partidosJugados>
    </personal>
    <personal id="32">
        <tipo>Jugador</tipo>
        <nombre>Sofía</nombre>
        <apellidos>Gómez</apellidos>
        <fechaNacimiento>2000-07-15</fechaNacimiento>
        <fechaIncorporacion>2024-01-20</fechaIncorporacion>
        <salario>39000.0</salario>
        <pais>España</pais>
        <especialidad/>
        <posicion>DELANTERO</posicion>
        <dorsal>11</dorsal>
        <altura>1.65</altura>
        <peso>60.25</peso>
        <goles>15</goles>
        <partidosJugados>30</partidosJugados>
    </personal>
    <personal id="33">
        <tipo>Entrenador</tipo>
        <nombre>Roberto</nombre>
        <apellidos>Sánchez</apellidos>
        <fechaNacimiento>1975-02-10</fechaNacimiento>
        <fechaIncorporacion>2022-05-15</fechaIncorporacion>
        <salario>60000.0</salario>
        <pais>Uruguay</pais>
        <especialidad>ENTRENADOR_PORTEROS</especialidad>
        <posicion/>
        <dorsal/>
        <altura/>
        <peso/>
        <goles/>
        <partidosJugados/>
    </personal>
</equipo>
```

Ejemplo de archivo JSON

```json
[
    {
        "id": 21,
        "nombre": "Carlos",
        "apellidos": "Santana",
        "fecha_nacimiento": "1985-01-15",
        "fecha_incorporacion": "2002-03-01",
        "salario": 33000.0,
```

```
        "pais": "Brasil",
        "rol": "Jugador",
        "especialidad": "",
        "posicion": "DELANTERO",
        "dorsal": 9,
        "altura": 1.82,
        "peso": 74.0,
        "goles": 50,
        "partidos_jugados": 140
    },
    {
        "id": 22,
        "nombre": "Miguel",
        "apellidos": "Rodriguez",
        "fecha_nacimiento": "1982-11-30",
        "fecha_incorporacion": "2001-07-15",
        "salario": 34000.0,
        "pais": "España",
        "rol": "Jugador",
        "especialidad": "",
        "posicion": "CENTROCAMPISTA",
        "dorsal": 8,
        "altura": 1.76,
        "peso": 70.0,
        "goles": 25,
        "partidos_jugados": 150
    }
...
]
```


4. Enlace al Repositorio de Código Fuente

El código fuente completo del proyecto se encuentra disponible en el repositorio de GitHub https://github.com/karrasmil80/Proyecto-Futbol.git. Este repositorio incluye el código, los archivos de configuración, los scripts de prueba y otros recursos relevantes.

5. Listado de Dependencias *Gradle*

Para facilitar la compilación y ejecución del proyecto, se incluye un listado de las dependencias declaradas en el archivo *build.gradle.kts*. Este listado permite reproducir el entorno de desarrollo y asegurar la compatibilidad de las diferentes librerías utilizadas.

```
plugins {
    kotlin("jvm") version "2.0.21"

    // Plugin para serializar
    kotlin("plugin.serialization") version "1.6.10"
}
```

```
group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))

    // LOGGER
    implementation("org.lighthousegames:logging:1.5.0")
    implementation("ch.qos.logback:logback-classic:1.5.12")

    //DOKKA
    implementation("org.jetbrains.dokka:dokka-gradle-plugin:2.0.0")

    // Serializable JSON
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.3.2")
    // Serializable XML
    implementation("io.github.pdvrieze.xmlutil:serialization-
jvm:0.90.3")
    //MockK
    testImplementation("io.mockk:mockk:1.13.16")
}

tasks.test {
    useJUnitPlatform()
}
kotlin {
    jvmToolchain(21)
}
```

6. Instrucciones de Ejecución

Para ejecutar el proyecto en IntelliJ IDEA, es menester seguir los siguientes pasos:

1. Clonar el repositorio de GitHub: *git clone* https://github.com/karrasmil80/Proyecto-Futbol.git
2. Abrir el proyecto en IntelliJ IDEA.
3. Ejecutar la tarea *gradle build* para compilar el proyecto.
4. Ejecutar el archivo JAR generado en la carpeta *build/libs*: *java -jar Proyecto-Futbol.jar*