

CAMPEONES

Oliver y Benji



Samuel Gómez Gutiérrez, Carlos Cortés Yagüe y Jesús Cobo Arrogante

1. Presentación del proyecto

En este proyecto se va a realizar el desarrollo de una aplicación de gestión de trabajo para un equipo de fútbol llamado New Team, que requerirá de funcionalidades básicas de creación, modificación y eliminación de registros así como administrar las entradas en memoria y la gestión del almacenamiento local en el equipo leyendo y escribiendo en distintos formatos de archivo.

2. Presentación del equipo y reparto de tareas

El proyecto fue realizado por [Samuel Gómez Gutiérrez](#), [Carlos Cortés Yagüe](#) y [Jesús Cobo Arrogante](#), tres estudiantes de Desarrollo de Aplicaciones Web en el IES Luis Vives de Leganés.

Los integrantes se encargarán de lo siguiente:

Tareas comunes: Creación de modelos, creación de consultas, definición de los requisitos del proyecto y la configuración.

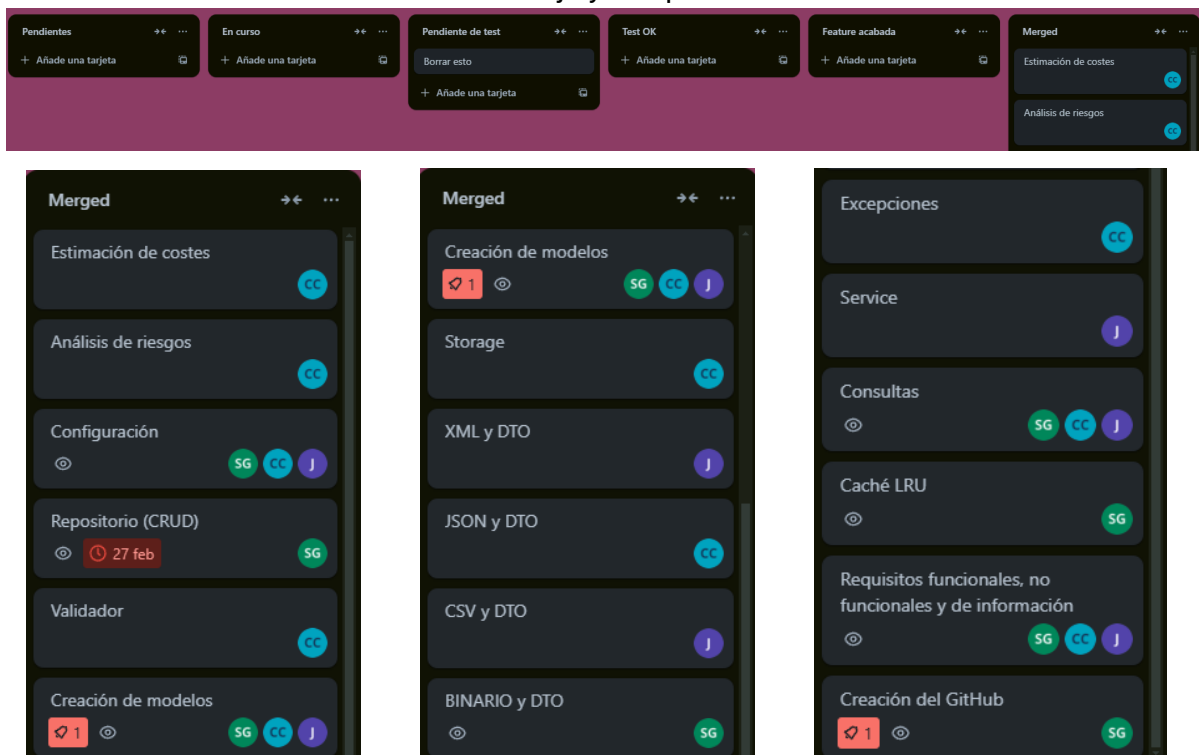
Samuel Gómez: Creación del repositorio con funciones CRUD, gestión de almacenamiento, lectura y escritura de ficheros binarios, creación de la caché con filosofía LRU y la creación del repositorio remoto principal.

Carlos Cortés: Estimación de los costes, análisis de riesgos, validadores de las clases, almacenamiento, gestión de lectura y escritura de ficheros JSON y el árbol de excepciones.

Jesús Cobo: Gestión de almacenamiento, lectura y escritura de ficheros XML y CSV y la creación del servicio que gestiona los distintos componentes de la aplicación.

3. Trello y gestión del trabajo. Análisis de riesgos

Esta era inicialmente la estructura de trabajo y la repartición de tareas en Trello:



Sin embargo, finalmente la repartición de tareas fue diferente debido a las circunstancias de los programadores y fueron redistribuidas.

Se han considerado los riesgos de que alguno de los trabajadores no pudiese realizar las tareas asignadas a su parte del proyecto y en base a ello se ha destinado una partida de la estimación de costes del proyecto en concepto de horas extras para poder abarcar todos los requisitos de la aplicación.

4. Captura del gráfico de donde se muestre el modelo de flujo de trabajo GitFlow y se explique cómo se ha integrado y resuelto los conflictos

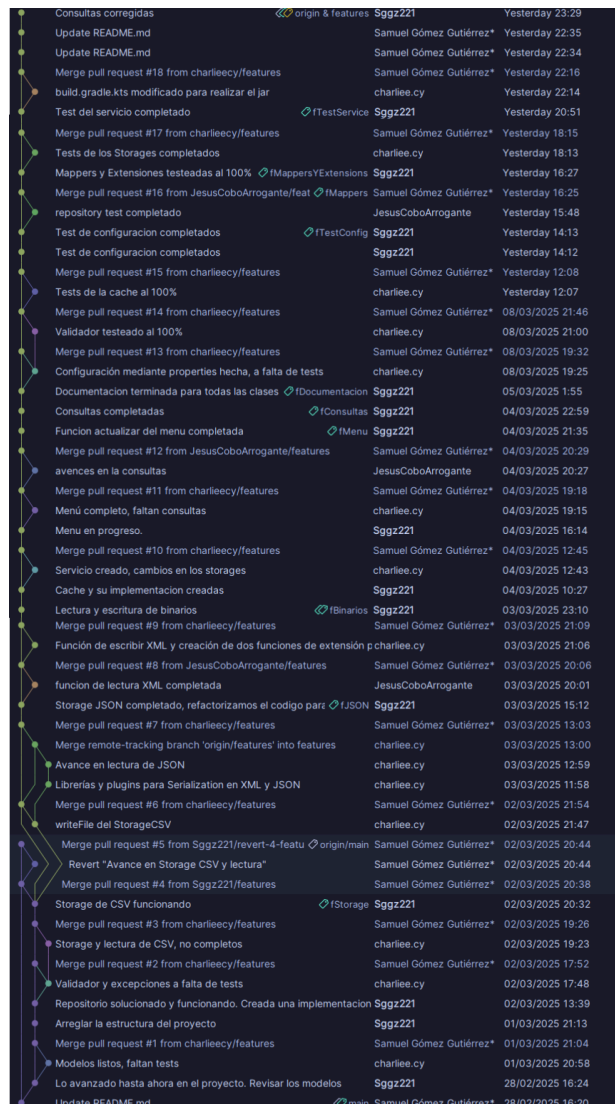


Gráfico de flujo de trabajo de git

Para desarrollar este proyecto llevamos a cabo un modelo en el que creamos una rama para una funcionalidad en concreto y una vez finalizada se hacía merge a la rama features, para finalmente hacer el merge a dev y de dev a main.

5. Especificación de requisitos funcionales, no funcionales y de información de forma razonada

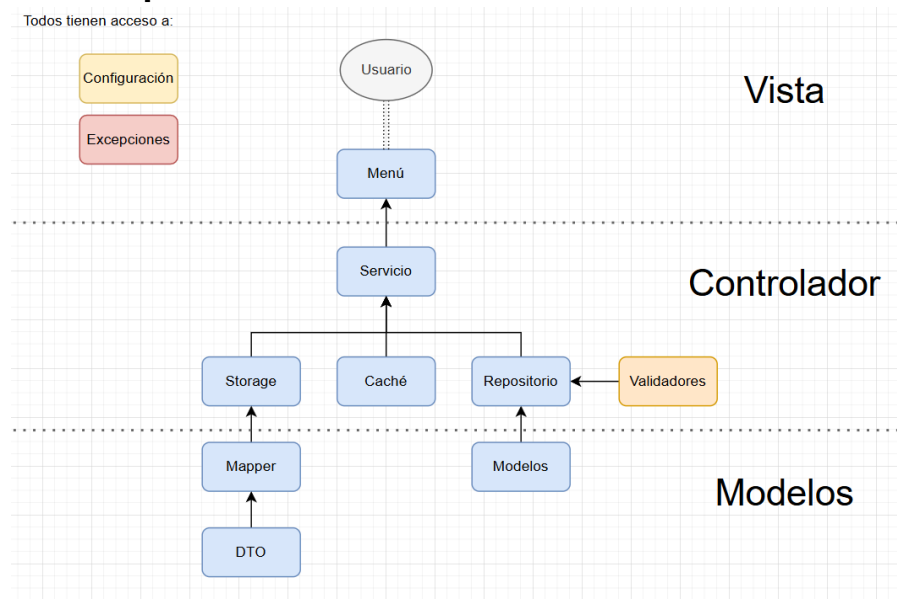
- **Requisitos funcionales** (qué debe hacer la aplicación).
 - **RF1:** Realizar operaciones CRUD (Create, Read, Update y Delete) para gestionar los miembros del equipo.
 - **RF2:** Asignar un id único a cada miembro del equipo guardado.
 - **RF3:** Representar a los miembros del equipo como Jugadores o Entrenadores.
 - **RF4:** De todos los miembros, almacenar nombre, apellidos, fecha de nacimiento, fecha de incorporación, salario y país de origen.
 - **RF5:** De los Jugadores, además, almacenar posición en el campo (portero, defensa, centrocampista o delantero), número de camiseta, altura, peso, número de goles anotados y partidos jugados.
 - **RF6:** De los Entrenadores, además, almacenar su área de especialización (entrenador de porteros, entrenador principal y entrenador asistente).
 - **RF7:** Validar los datos de los miembros antes de guardarlos para evitar errores o inconsistencias.
 - **RF8:** Permitir la importación de miembros desde consola y desde la lectura de ficheros CSV, JSON, XML y BIN.
 - **RF9:** Permitir la exportación de miembros a ficheros CSV, JSON, XML y BIN.
 - **RF10:** Implementar una caché con filosofía LRU con un tamaño máximo de 5 elementos.
 - **RF11:** Localizar la exportación de la información según el fichero de configuración.
 - **RF12:** Implementar un menú para que el usuario maneje la aplicación por consola, con las siguientes opciones:
 - Cargar datos desde fichero según la especificación indicada.
 - Crear miembro del equipo.
 - Actualizar miembro de equipo.
 - Eliminar miembro del equipo.
 - Copiar datos a fichero según la especificación realizada.
 - Realizar consultas indicadas.

- **RF13:** Sobre los datos almacenados, realizar las siguientes consultas:
 - Listados de personal agrupados por entrenadores y jugadores.
 - El delantero más alto.
 - Media de goles de los delanteros.
 - Defensa con más partidos jugados.
 - Jugadores agrupados por su país de origen.
 - Entrenador con el mayor salario.
 - Promedio de altura de los jugadores agrupados por posición.
 - Listado de todos los jugadores que han anotado más de 10 goles.
 - Jugadores con un salario mayor al promedio del equipo.
 - Número total de partidos jugados por todos los jugadores.
 - Jugadores agrupados por el año de su incorporación al club.
 - Entrenadores agrupados por su especialidad.
 - Jugador más joven en el equipo.
 - Promedio de peso de los jugadores por posición.
 - Listado de todos los jugadores que tienen un dorsal par.
 - Jugadores que han jugado menos de 5 partidos.
 - Media de goles por partido de cada jugador.
 - Listado de jugadores que tienen una altura superior a la media del equipo.
 - Entrenadores que se incorporaron al club en los últimos 5 años.
 - Jugadores que han anotado más goles que el promedio de su posición.
 - Por posición, máximo de goles, mínimo de goles y media.
 - Estimación del coste total de la plantilla.
 - Total del salario pagado, agrupados por año de incorporación.
 - Jugadores agrupados por país y, dentro de cada grupo, el jugador con más partidos jugados.

- Promedio de goles por posición, y dentro de cada posición, el jugador con el mayor número de goles.
- Entrenadores agrupados por especialidad, y dentro de cada especialidad, el entrenador con el salario más alto.
- Jugadores agrupados por década de nacimiento, y dentro de cada grupo, el promedio de partidos jugados.
- Salario promedio de los jugadores agrupados por su país de origen, y dentro de cada grupo, el jugador con el salario más bajo y alto.
- **Requisitos no funcionales** (cómo debe funcionar la aplicación).
 - **RNF1:** Los datos de los miembros del equipo deben validarse para evitar errores e inconsistencias.
 - **RNF2:** La caché debe mejorar la velocidad de acceso a los datos.
 - **RNF3:** El menú debe ser intuitivo y facilitar el manejo de la aplicación a los usuarios.
 - **RNF4:** La aplicación debe implementar un sistema de logs para facilitar el seguimiento de la traza del programa y la corrección de errores.
 - **RNF5:** Todo el código debe estar documentado y el proyecto debe incluir la documentación generada por dokka.
 - **RNF6:** Todo el código debe ser testeado.
 - **RNF7:** La aplicación debe poder ejecutarse desde la terminal gracias al archivo .jar
 - **RNF8:** Durante el desarrollo del proyecto, el equipo debe trabajar usando GitFlow/Git/Pull Request.
- **Requisitos de información** (peculiaridades sobre los tipos de datos que se gestionan).
 - **RI1:** Todos los miembros del equipo tendrán los siguientes campos comunes.
 - **Id:** De tipo Long para posibilitar una mayor escalabilidad del proyecto, al permitir un rango más amplio de valores que el tipo Int.
 - **Nombre:** De tipo String. El nombre no puede estar vacío.
 - **Apellidos:** De tipo String. Los apellidos no pueden estar vacíos.
 - **Fecha de nacimiento:** de tipo LocalDate para que las fechas respeten el estándar ISO-8601. La fecha de nacimiento no puede ser posterior ni a la fecha actual ni a la fecha de incorporación.

- **Fecha de incorporación:** de tipo LocalDate para que las fechas respeten el estándar ISO-8601. La fecha de incorporación no puede ser posterior a la fecha actual ni anterior a la fecha de nacimiento.
 - **Salario:** de tipo Double para poder representar los céntimos como cifras decimales. El salario no puede ser negativo.
 - **País de origen:** de tipo String. El país de origen no puede estar vacío.
- **RI2:** Los miembros del equipo que sean Jugadores tendrán los siguientes campos específicos.
- **Posición:** de tipo Posición (una enum class que contiene los valores portero, defensa, centrocampista y delantero).
 - **Dorsal:** De tipo Int, ya que solo se admitirán dorsales del 1 al 99.
 - **Altura:** De tipo Double, para poder representar los cms como cifras decimales. La altura no puede ser negativa ni superior a 3m.
 - **Peso:** De tipo Double, para poder representar los g como cifras decimales. El peso no puede ser negativo.
 - **Goles:** De tipo Int. Los goles no pueden ser negativos.
 - **Partidos jugados:** De tipo Int. Los partidos jugados no pueden ser negativos.
- **RI3:** Los miembros del equipo que sean Entrenadores tendrán el siguiente campo específico.
- **Especialidad:** de tipo Especialidad (una enum class que contiene los valores entrenador de porteros, entrenador principal o entrenador asistente).

6. Diagrama de arquitectura de software



7. Herramientas tecnológicas usadas

Entorno de desarrollo IntelliJ IDEA de JetBrains, librería Logger para el uso de logs en la aplicación, Dokka para generar una documentación profesional, Git para el control de versiones y gestión del flujo de trabajo, GitHub para el trabajo en equipo.

8. Justificación de elecciones de cada uno de los elementos software existente justificando de las opciones posibles y por qué se ha decantado el equipo por ella, justificado pros y contras de la solución parcial aportada por dicho elemento

No hubo muchos problemas a la hora de decidir por un diseño en concreto, pero cabe a destacar que por consecuencia de la herencia en la que se basó el diseño principal de los modelos, se tuvo que decidir si los campos específicos para cada clase (tipo de entrenador, posición) se implementan con interfaces o con *enum class*.

Al final se decidió usar *enum class* debido a que no se iba a poder sacar tanto partido a las interfaces porque las clases no tenían comportamientos específicos dependiendo de su **rol** puesto que es un programa meramente de gestión. Las *enum class* por el contrario proveían de una implementación mucho más sencilla y a la hora de convertir los datos a los ficheros fue un proceso casi directo, cosa que con las interfaces no hubiera sido tan sencillo.

También fue necesario diseñar en base al modelo MVC, por el cual no hubo debate puesto que prácticamente se describe en los requisitos de la aplicación haciendo de su requerimiento algo indispensable.

9. Principios SOLID aplicados y ejemplos de su implementación y uso en el código

Los principios SOLID usados en el proyecto son:

S (Single responsibility principle) Principio de responsabilidad única: Principio que dictamina que cada elemento del software debe realizar una única función y delegar el resto del trabajo a otras funciones. Ejemplo:

```
fun callOperation(option: Int){  Sggz221 +1
    when(option){
        1 → cargarDatos()
        2 → crearMiembro()
        3 → actualizarMiembro()
        4 → borrarMiembro()
        5 → service.getAll().forEach {println(it)}
        6 → exportarDatos()
        7 → data.consultas()
        8 → {}
    }
}
```

La función callOperation() no realiza las funciones de cargar datos, crear miembros, etc. sino que lo delega a otras funciones que se encargan de dicha operación.

O (Open/Close principle) Principio abierto/cerrado: Una clase debe estar cerrada a la su modificación pero abierta a su extensión. Ejemplo:

```
fun Entrenador.copy( Sggz221
    newId: Long= this.id,
    newNombre: String= this.nombre,
    newApellidos: String = this.apellidos,
    new_fecha_nacimiento: LocalDate = this.fecha_nacimiento,
    new_fecha_incorporacion: LocalDate = this.fecha_incorporacion,
    newSalario: Double = this.salario,
    newPais: String = this.pais,
    newIsDeleted: Boolean = this.isDeleted,
    newEspecialidad: Especialidad = this.especialidad,
    timeStamp: LocalDateTime = LocalDateTime.now()
): Entrenador {
    return Entrenador(
        id = newId,
        nombre = newNombre,
        apellidos = newApellidos,
        fecha_nacimiento = new_fecha_nacimiento,
        fecha_incorporacion = new_fecha_incorporacion,
        salario = newSalario,
        pais = newPais,
        createdAt = timeStamp,
        updatedAt = timeStamp,
        isDeleted = newIsDeleted,
        especialidad = newEspecialidad,
    )
}
```

En este ejemplo se ve cómo en lugar de modificar la clase Entrenador, se extiende para añadir un comportamiento.

L (Liskov substitution principle) Principio de sustitución de Liskov: Los objetos de un programa pueden ser reemplazados por sus subtipos sin alterar su correcto funcionamiento. Ejemplo:

- Entrenador
- Especialidad
- Integrante
- Jugador
- Posicion

```
private val logger = logging()
private val equipo = mutableMapOf<Long, Integrante>()
private var nextId = 1L
private var validator = IntegranteValidator()
```

En este ejemplo se muestra una herencia de clases, en la que la superclase **Integrante**, la cual es una clase abstracta y no se puede instanciar, está presente en todo el programa de manera que todas las colecciones de este son de Integrante. Esto es posible debido a que **Jugador** y **Entrenador** son subclases de Integrante y todos los comportamientos de Integrante lo heredan estas dos y donde se espera un Integrante se puede incluir o bien un Jugador o bien un Entrenador. Es una consecuencia del **polimorfismo**.

I (Interface segregation principle) Principio de segregación de interfaces: Es mejor tener varias interfaces con pocas funciones que una interfaz más grande que tenga muchas funciones que no todas las clases pueden usar y deben implementar igualmente. Ejemplo:

```
interface Service {  🐞 charliee.cy
    fun importFromFile(filePath: String)  🐞 charliee.cy
    fun exportToFile(filePath: String)  🐞 charliee.cy

    fun getAll(): List<Integrante>  🐞 charliee.cy
    fun getById(id: Long): Integrante  🐞 charliee.cy
    fun save(integrante: Integrante): Integrante  🐞 charliee.cy
    fun update(id: Long, integrante: Integrante): Integrante  🐞 charliee.cy
    fun delete(id: Long): Integrante  🐞 charliee.cy
    fun deleteLogical(id: Long, integrante: Integrante): Integrante  🐞 charliee.cy
}
```

En el ejemplo se observa como una interfaz Service tiene una serie de métodos que nosotros implementamos en la clase servicio. Si en un futuro la aplicación obtuviese una funcionalidad que requiriese un servicio distinto pero con más métodos, estos métodos deberían de ir en una interfaz a parte en lugar de aumentar el tamaño de ésta puesto que el actual servicio no haría uso de las nuevas funciones.

10. Patrones aportados en la solución y ejemplos de uso

Patrón MVC Model View Controller: Este patrón se basa en el uso de clases de manera que su responsabilidad se pueda diferenciar en las siguientes capas:

- Modelos: Son las abstracciones del programa y es lo que usan las clases del controlador para gestionar el programa. Aquí entrarían las clases Entrenador, Jugador, Integrante, las *enum class*, etc.
- Controlador: Es el encargado de gestionar todo el programa. Realización de cálculos, empleo de los modelos, gestión en memoria, etc.
- Vista: La información extraída del controlador es gestionada en la vista, la cual da un formato a la información, la muestra en pantalla y sirve como intermediario entre el usuario y el programa.

Patrón Singleton: Este patrón se basa en crear una única instancia de un objeto en tiempo de ejecución impidiendo múltiples instancias del mismo objeto.

Este patrón se cumple en la creación del objeto **Configuration**.

11. Ejemplos de ejecución del código

```
---- Aplicación de gestión de un equipo de fútbol ----
¿Qué desea hacer?

|1. Cargar datos desde fichero
|2. Crear un miembro del equipo
|3. Actualizar un miembro del equipo
|4. Eliminar un miembro del equipo
|5. Mostrar miembros
|6. Exportar equipo a fichero
|7. Imprimir consultas del equipo
|8. Salir de la aplicación
```

Menú principal

```
¿De qué fichero importar los datos del equipo?
1| CSV, 2| JSON, 3| XML, 4| BIN:
```

Importación de fichero (opción 1)

```
Escriba el id del jugador
1
10:57:01.572 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo el integrante del equipo con id 1
10:57:01.574 [main] DEBUG org.example.Cache.CacheImpl -- Obteniendo elemento en la cache con clave 1
10:57:01.574 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo integrante del equipo con ID: 1
10:57:01.574 [main] DEBUG org.example.Cache.CacheImpl -- Guardando elemento en la cache con clave 1
10:57:01.574 [main] DEBUG org.example.Cache.CacheImpl -- Alcanzado el tamaño máximo: eliminando elemento más antiguo de la cache
¿Qué campo del jugador deseas actualizar?|1. Nombre |2. Apellidos |3. Fecha de Nacimiento |4. Fecha de incorporacion |5. Salario |6. Pais
```

Actualización de un miembro del equipo (opción 3)

12. Explicación de las consultas y ejemplos de salidas de cada una de ellas

1 - Mediante la función `partition`, se divide a los miembros del equipo entre los que cumplen y los que no cumplen la condición de ser Jugador. Se almacenan los jugadores en `listaJugadores`, los entrenadores en `listaEntrenadores` y se imprime cada una de las listas.

```
println("1. Listados de personal agrupados por entrenadores y jugadores.")
val todos = service.getAll()
val (listaJugadores, listaEntrenadores) = todos.partition { it is Jugador }
println("Lista de JUGADORES:")
listaJugadores.forEach {println(it)}
println("Lista de ENTRENADORES:")
listaEntrenadores.forEach {println(it)}
```

```
1. Listados de personal agrupados por entrenadores y jugadores.
09:55:50.613 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.613 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Lista de JUGADORES:
Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España, createdAt= 2025-03-10T09:
Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania, createdAt= 2025-03-10T0
Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra, createdAt= 2025-03-10T0
Jugador(id= 6, nombre= Bruce, apellidos= Harper, fecha_nacimiento= 1983-08-15, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025-03-10T09
Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = Italia, createdAt= 2025-03-1
Jugador(id= 9, nombre= Johnny, apellidos= Mason, fecha_nacimiento= 1984-01-25, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = España, createdAt= 2025-03-10T09
Jugador(id= 11, nombre= Paul, apellidos= Diamond, fecha_nacimiento= 1983-10-18, fecha_incorporacion= 2001-05-15, salario= 29000.0, pais = España, createdAt= 2025-03-10T0
Jugador(id= 12, nombre= Ed, apellidos= Warner, fecha_nacimiento= 1983-02-02, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T09:5
Jugador(id= 13, nombre= Danny, apellidos= Mello, fecha_nacimiento= 1984-09-09, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = España, createdAt= 2025-03-10T09
Jugador(id= 14, nombre= Ted, apellidos= Carter, fecha_nacimiento= 1983-07-07, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025-03-10T09:
Jugador(id= 15, nombre= Jack, apellidos= Morris, fecha_nacimiento= 1984-11-11, fecha_incorporacion= 2001-05-15, salario= 33000.0, pais = España, createdAt= 2025-03-10T09
Jugador(id= 16, nombre= Ralph, apellidos= Peterson, fecha_nacimiento= 1983-05-05, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10
Jugador(id= 17, nombre= Charlie, apellidos= Custer, fecha_nacimiento= 1984-08-08, fecha_incorporacion= 2001-05-15, salario= 29500.0, pais = España, createdAt= 2025-03-10
Jugador(id= 18, nombre= David, apellidos= Evans, fecha_nacimiento= 1983-12-12, fecha_incorporacion= 2001-05-15, salario= 28500.0, pais = España, createdAt= 2025-03-10T09
Jugador(id= 19, nombre= Patrick, apellidos= Everett, fecha_nacimiento= 1984-06-06, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-1
Jugador(id= 20, nombre= Richard, apellidos= Textex, fecha_nacimiento= 1983-03-03, fecha_incorporacion= 2001-05-15, salario= 27500.0, pais = España, createdAt= 2025-03-10
Lista de ENTRENADORES:
Entrenador(id= 1, nombre= Roberto, apellidos= Hongo, fecha_nacimiento= 1960-07-17, fecha_incorporacion= 2000-01-01, salario= 60000.0, pais = Brasil, createdAt= 2025-03-1
Entrenador(id= 4, nombre= Freddy, apellidos= Marshall, fecha_nacimiento= 1965-09-22, fecha_incorporacion= 2005-04-10, salario= 55000.0, pais = España, createdAt= 2025-03
Entrenador(id= 7, nombre= Jefferson, apellidos= Robles, fecha_nacimiento= 1968-12-12, fecha_incorporacion= 2003-11-11, salario= 58000.0, pais = México, createdAt= 2025-0
Entrenador(id= 10, nombre= Leo, apellidos= Aragones, fecha_nacimiento= 1970-05-30, fecha_incorporacion= 2002-09-01, salario= 57000.0, pais = Argentina, createdAt= 2025-0
```

2 -Se filtran solo los jugadores. Sobre los jugadores, se vuelve a filtrar sólo aquellos que son delanteros. Sobre los delanteros, se obtiene el delantero con el máximo valor en el campo altura.

```
println("2. El delantero más alto.")
println(service.getAll().filterIsInstance<Jugador>().filter { it.posicion == Posicion.DELANTERO }.maxBy { it.altura })

2. El delantero más alto.
09:55:50.614 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.615 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 17, nombre= Charlie, apellidos= Custer, fecha_nacimiento= 1984-08-08, fecha_incorporacion= 2001-05-15, salario= 29500.0, pais = España, createdAt= 2025-03-10
```

3 -Se filtran solo los jugadores. Sobre los jugadores, se vuelve a filtrar sólo aquellos que son delanteros. Se mapean los delanteros para quedarnos solo con el número de goles y se hace la media de ese campo.

```
println("3. Media de goles de los delanteros.")
println(service.getAll().filterIsInstance<Jugador>().filter { it.posicion == Posicion.DELANTERO }.map { it.goles }.average())

3. Media de goles de los delanteros.
09:55:50.615 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.615 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
39.25
```

4 -Se filtran solo los jugadores. Sobre los jugadores, se vuelve a filtrar sólo aquellos que son defensas. Sobre los defensas, se obtiene el defensa con el máximo valor en el campo partidos jugados.

```
println("4. Defensa con más partidos jugados.")
println(service.getAll().filterIsInstance<Jugador>().filter { it.posicion == Posicion.DEFENSA }.maxBy { it.partidos_jugados })

4. Defensa con más partidos jugados.
09:55:50.615 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.615 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 6, nombre= Bruce, apellidos= Harper, fecha_nacimiento= 1983-08-15, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025-03-10T09:55:50.615Z)
```

5 - Se filtran solo los jugadores. Se agrupan los jugadores por país.

```
println("5. Jugadores agrupados por su país de origen.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.pais }.forEach { println(it) }

5. Jugadores agrupados por su país de origen.
09:55:50.615 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.615 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
España=[Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España, createdAt= 2025-03-10T09:55:50.615Z), Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania, createdAt= 2025-03-10T09:55:50.615Z)]
Alemania=[Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania, createdAt= 2025-03-10T09:55:50.615Z)]
Inglaterra=[Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra, createdAt= 2025-03-10T09:55:50.615Z)]
Italia=[Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = Italia, createdAt= 2025-03-10T09:55:50.615Z)]
```

6 - Se filtran los entrenadores. Sobre los entrenadores, se obtiene el entrenador con el máximo valor en el campo salario.

```
println("6. Entrenador con el mayor salario.")
println(service.getAll().filterIsInstance<Entrenador>().maxBy { it.salario })

6. Entrenador con el mayor salario.
09:55:50.616 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.616 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Entrenador(id= 1, nombre= Roberto, apellidos= Hongo, fecha_nacimiento= 1960-07-17, fecha_incorporacion= 2000-01-01, salario= 60000.0, pais = Brasil, createdAt= 2025-03-10T09:55:50.616Z)
```

7 - Se filtran los jugadores. Los jugadores se agrupan por posición. Del mapa resultante, se transforman los valores para quedarnos solo con la altura y, de ella, se obtiene la media.

```
println("7. Promedio de altura de los jugadores agrupados por posición.")
println(service.getAll().filterIsInstance<Jugador>().groupBy { it.posicion }.mapValues { (_, jugadores) -> jugadores.map { it.altura }.average() })

7. Promedio de altura de los jugadores agrupados por posición.
09:55:50.616 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.616 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
{DELANTERO=1.765, PORTERO=1.8450000000000002, CENTROCAMPISTA=1.7349999999999999, DEFENSA=1.795}
```

8 - Se filtran los jugadores. Se vuelven a filtrar solo aquellos jugadores cuyo valor en el campo goles sea mayor a 10.

```
println("8. Listado de todos los jugadores que han anotado más de 10 goles.")
service.getAll().filterIsInstance<Jugador>().filter { it.goles > 10 }.forEach { println(it) }

8. Listado de todos los jugadores que han anotado más de 10 goles.
09:55:50.619 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.619 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 9, nombre= Johnny, apellidos= Mason, fecha_nacimiento= 1984-01-25, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 11, nombre= Paul, apellidos= Diamond, fecha_nacimiento= 1983-10-18, fecha_incorporacion= 2001-05-15, salario= 29000.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 13, nombre= Danny, apellidos= Mello, fecha_nacimiento= 1984-09-09, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 14, nombre= Ted, apellidos= Carter, fecha_nacimiento= 1983-07-07, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 17, nombre= Charlie, apellidos= Custer, fecha_nacimiento= 1984-08-08, fecha_incorporacion= 2001-05-15, salario= 29500.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
Jugador(id= 18, nombre= David, apellidos= Evans, fecha_nacimiento= 1983-12-12, fecha_incorporacion= 2001-05-15, salario= 28500.0, pais = España, createdAt= 2025-03-10T09:55:50.619Z)
```

9 - Filtramos los jugadores. Sobre los jugadores, se aplica un nuevo filtro en el que la condición de filtrado es que el salario propio de cada jugador supere al salario medio de todo el equipo.

Esta consulta no arroja ningún resultado porque ninguno de los jugadores que hay importados desde el archivo .csv (tomado como referencia para ejecutar las consultas) cumple la condición.

```
println("9. Jugadores con un salario mayor al promedio del equipo.")
service.getAll().filterIsInstance<Jugador>().filter { it.salario > service.getAll().map { it.salario }.average() }.forEach { println(it) }
```

10 - Se filtran los jugadores. Solo de los jugadores, se suman los valores del campo partidos jugados para obtener el total.

```
println("10. Número total de partidos jugados por todos los jugadores.")
println("Partidos jugados en total: " + service.getAll().filterIsInstance<Jugador>().sumOf { it.partidos_jugados })
```

10. Número total de partidos jugados por todos los jugadores.
09:55:50.620 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.620 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Partidos jugados en total: 2435

11 - Se filtran los jugadores. Se agrupan por el año de su campo fecha de incorporación.

```
println("11. Jugadores agrupados por el año de su incorporación al club.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.fecha_incorporacion.year }.forEach { println(it) }
```

11. Jugadores agrupados por el año de su incorporación al club.
09:55:50.621 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.621 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
2001=[Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España, createdAt= 2025-03-

12 - Se filtran los entrenadores. Se agrupan por su campo especialidad.

```
println("12. Entrenadores agrupados por su especialidad.")
service.getAll().filterIsInstance<Entrenador>().groupBy { it.especialidad }.forEach { println(it) }
```

12. Entrenadores agrupados por su especialidad.
09:55:50.622 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.622 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
ENTRENADOR_PRINCIPAL=[Entrenador(id= 1, nombre= Roberto, apellidos= Hongo, fecha_nacimiento= 1960-07-17, fecha_incorporacion= 2000-01-01, salario= 60000.0, pais = Brasil,
ENTRENADOR_PORTEROS=[Entrenador(id= 4, nombre= Freddy, apellidos= Marshall, fecha_nacimiento= 1965-09-22, fecha_incorporacion= 2005-04-10, salario= 55000.0, pais = España
ENTRENADOR_ASISTENTE=[Entrenador(id= 7, nombre= Jefferson, apellidos= Robles, fecha_nacimiento= 1968-12-12, fecha_incorporacion= 2003-11-11, salario= 58000.0, pais = México

13 - Se filtran los jugadores. De éstos, se obtiene el jugador con el valor máximo en el campo fecha de nacimiento, es decir, el más joven.

```
println("13. Jugador más joven en el equipo.")
println(service.getAll().filterIsInstance<Jugador>().maxBy { it.fecha_nacimiento })
```

13. Jugador más joven en el equipo.
09:55:50.622 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.622 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 15, nombre= Jack, apellidos= Morris, fecha_nacimiento= 1984-11-11, fecha_incorporacion= 2001-05-15, salario= 33000.0, pais = España, createdAt= 2025-03-10T

14 - Se filtran los jugadores. Una vez filtrados, se agrupan por posición. Del mapa resultante de groupBy, omitimos las claves (nos interesa que la clave sea la posición) y transformamos los valores asociados a dichas claves en la media del valor del campo peso.

```
println("14. Promedio de peso de los jugadores por posición.")
println(service.getAll().filterIsInstance<Jugador>().groupBy { it.posicion }.mapValues { (_,jugadores) -> jugadores.map { it.peso }.average() })
```

```
14. Promedio de peso de los jugadores por posición.
09:55:50.622 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.622 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
{DELANTERO=67.25, PORTERO=79.75, CENTROCAMPISTA=66.0, DEFENSA=71.5}
```

15 - Se filtran los jugadores. Sobre los jugadores, se vuelve a aplicar un filtro para quedarnos con aquellos que cumplan la condición de que su número de dorsal sea par.

```
println("15. Listado de todos los jugadores que tienen un dorsal par.")
service.getAll().filterIsInstance<Jugador>().filter { it.dorsal % 2 == 0 }.forEach { println(it) }
```

```
15. Listado de todos los jugadores que tienen un dorsal par.
09:55:50.623 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.623 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 6, nombre= Bruce, apellidos= Harper, fecha_nacimiento= 1983-08-15, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 9, nombre= Johnny, apellidos= Mason, fecha_nacimiento= 1984-01-25, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 12, nombre= Ed, apellidos= Warner, fecha_nacimiento= 1983-02-02, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 17, nombre= Charlie, apellidos= Custer, fecha_nacimiento= 1984-08-08, fecha_incorporacion= 2001-05-15, salario= 29500.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 19, nombre= Patrick, apellidos= Everett, fecha_nacimiento= 1984-06-06, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
Jugador(id= 20, nombre= Richard, apellidos= Textex, fecha_nacimiento= 1983-03-03, fecha_incorporacion= 2001-05-15, salario= 27500.0, pais = España, createdAt= 2025-03-10T15:00:00.000Z)
```

16 - Se filtran los jugadores. Sobre los jugadores, se vuelve a aplicar un filtro para quedarnos con aquellos que cumplan la condición de que su número de partidos jugados sea menor a 5.

Esta consulta no arroja ningún resultado porque ninguno de los jugadores que hay importados desde el archivo .csv (tomado como referencia para ejecutar las consultas) cumple la condición.

```
println("16. Jugadores que han jugado menos de 5 partidos.") //no muestra nada ya que no hay nada
service.getAll().filterIsInstance<Jugador>().filter { it.partidos_jugados < 5 }.forEach { println(it) }
```

17 - Se filtran los jugadores. Para cada jugador, se crea la variable media, a la que se le asigna el resultado de dividir su número de goles entre su número de partidos jugados. Se imprime el id, el nombre y la media de goles por partido de cada jugador.

```
println("17. Media de goles por partido de cada jugador.")
service.getAll().filterIsInstance<Jugador>().forEach {
    val media:Double = it.goles.toDouble() / it.partidos_jugados
    println(it.id.toString()+ " " + it.nombre + " " + media)
}
```



```

17. Media de goles por partido de cada jugador.
09:55:50.623 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.623 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
2 Oliver 0.4666666666666667
3 Benji 0.0
5 Tom 0.21428571428571427
6 Bruce 0.027777777777777776
8 Phillip 0.0625
9 Johnny 0.11538461538461539
11 Paul 0.27586206896551724
12 Ed 0.0
13 Danny 0.18518518518518517
14 Ted 0.16
15 Jack 0.05161290322580645
16 Ralph 0.0
17 Charlie 0.15714285714285714
18 David 0.1565217391304348
19 Patrick 0.04
20 Richard 0.0

```

18 - Se crea la variable media, a la que se le asigna el valor de la media de altura de los jugadores (se filtran los jugadores, sobre los jugadores filtrados nos quedamos solo con el campo altura y sobre ese campo se calcula la media).

Se filtran los jugadores, se vuelve a filtrar solo aquellos cuya altura es mayor a la media del equipo anteriormente calculada y se imprimen.

```

println("18. Listado de jugadores que tienen una altura superior a la media del equipo.")
val alturaMedia = service.getAll().filterIsInstance<Jugador>().map { it.altura }.average()
service.getAll().filterIsInstance<Jugador>().filter { it.altura > alturaMedia }.forEach { println(it) }

```

```

18. Listado de jugadores que tienen una altura superior a la media del equipo.
09:55:50.626 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.626 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
09:55:50.626 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.626 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = Italia, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 12, nombre= Ed, apellidos= Warner, fecha_nacimiento= 1983-02-02, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 15, nombre= Jack, apellidos= Morris, fecha_nacimiento= 1984-11-11, fecha_incorporacion= 2001-05-15, salario= 33000.0, pais = España, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 16, nombre= Ralph, apellidos= Peterson, fecha_nacimiento= 1983-05-05, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 19, nombre= Patrick, apellidos= Everett, fecha_nacimiento= 1984-06-06, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = España, createdAt= 2025-03-10T12:00:00.000Z)
Jugador(id= 20, nombre= Richard, apellidos= Textex, fecha_nacimiento= 1983-03-03, fecha_incorporacion= 2001-05-15, salario= 27500.0, pais = España, createdAt= 2025-03-10T12:00:00.000Z)

```

19 - Se filtran los entrenadores. Sobre los entrenadores, se vuelve a filtrar solo aquellos que cumplan la condición de que el resultado de restar el año de su fecha de incorporación al año actual sea menor o igual a 5.

Esta consulta no arroja ningún resultado porque ninguno de los entrenadores que hay importados desde el archivo .csv (tomado como referencia para ejecutar las consultas) cumple la condición.

```

println("-----")
println("19. Entrenadores que se incorporaron al club en los últimos 5 años.") //no muestra nada ya que no hay nada
service.getAll().filterIsInstance<Entrenador>().filter { (LocalDate.now().year - it.fecha_incorporacion.year) <=5 }.forEach { println(it) }

```

20 - Se filtran los jugadores. Se agrupan los jugadores por su posición.

Se crea la variable `mediaGolesPosicion`, en la que se almacena el número medio de goles que han anotado cada jugador (ya agrupados por posición).

Del mapa resultante de agrupar, los valores son filtrados de nuevo para seleccionar solo aquellos jugadores cuyo número de goles es mayor a la media de goles por posición.

```
println("20. Jugadores que han anotado más goles que el promedio de su posición.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.posicion }.mapValues { (posicion, jugadores) ->
    var mediaGolesPosicion = jugadores.map { it.goles }.average()
    jugadores.filter{it.goles > mediaGolesPosicion}.forEach { println(posicion.toString() + ": " + it) }
}
```

```
20. Jugadores que han anotado más goles que el promedio de su posición.
09:55:50.627 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.627 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
DELANTERO: Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, p
DELANTERO: Jugador(id= 11, nombre= Paul, apellidos= Diamond, fecha_nacimiento= 1983-10-18, fecha_incorporacion= 2001-05-15, salario= 29000.0
CENTROCAMPISTA: Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0
DEFENSA: Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0
DEFENSA: Jugador(id= 15, nombre= Jack, apellidos= Morris, fecha_nacimiento= 1984-11-11, fecha_incorporacion= 2001-05-15, salario= 33000.0, p
```

21 - Se filtran los jugadores. Se agrupan por posición.

Del mapa resultante de agrupar, se transforman los valores en un la variable `goles`, que solo almacena los goles que ha anotado cada jugador.

Se crea un triple formado por el máximo número de goles, el mínimo número de goles y la media de goles y se imprime.

```
println("21. Por posición, máximo de goles, mínimo de goles y media.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.posicion }.mapValues { (_, jugadores) ->
    val goles = jugadores.map { it.goles }
    Triple(goles.maxOrNull(), goles.minOrNull(), goles.average())}.forEach { println(it) }
```

```
21. Por posición, máximo de goles, mínimo de goles y media.
09:55:50.629 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.629 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
DELANTERO=(70, 22, 39.25)
PORTERO=(0, 0, 0.0)
CENTROCAMPISTA=(30, 15, 20.75)
DEFENSA=(10, 5, 7.25)
```

22 - Se suma el salario de todo el equipo y se muestra por pantalla

```
println("22. Estimación del coste total de la plantilla.")
val sumSalarios = service.getAll().sumOf { it.salario }
println("Coste total de toda la plantilla: $sumSalarios")
```

```
22. Estimación del coste total de la plantilla.
09:55:50.629 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.629 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
Coste total de toda la plantilla: 727500.0
```

23 - Se agrupan los miembros del equipo por el año de su fecha de incorporación. Del mapa obtenido se opera con los valores sumando el salario e imprimiendo por pantalla.

```
println("23. Total del salario pagado, agrupados por año de incorporación.")
println(service.getAll().groupBy { it.fecha_incorporacion.year }.mapValues { (_,jugadores) -> jugadores.sumOf { it.salario } } )

23. Total del salario pagado, agrupados por año de incorporación.
09:55:50.629 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.630 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
{2000=60000.0, 2001=497500.0, 2005=55000.0, 2003=58000.0, 2002=57000.0}
```

24 - Se filtran los miembros del equipo para obtener solo los jugadores. Esta lista se agrupa por posición resultando en un mapa, del cual obtenemos el jugador con el máximo de partidos jugados y se imprime por pantalla.

```
println("24. Jugadores agrupados por país y, dentro de cada grupo, el jugador con más partidos jugados.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.pais }.mapValues { (_,jugadores) -> jugadores.maxBy { it.partidos_jugados } }.forEach { println(it) }

24. Jugadores agrupados por país y, dentro de cada grupo, el jugador con más partidos jugados.
09:55:50.630 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.630 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
España=Jugador(id= 6, nombre= Bruce, apellidos= Harper, fecha_nacimiento= 1983-08-15, fecha_incorporacion= 2001-05-15, salario= 30000.0, pais = España, createdAt= 2025
Alemania=Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania, createdAt= 2
Inglaterra=Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra, createdAt=
Italia=Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = Italia, createdAt=
```

25 - Se filtran los miembros del equipo para obtener solo los jugadores. Esta lista se agrupa por posición resultando en un mapa, del cual almacenamos en la variable maxGoles el jugador con el máximo de goles y en la variable goles se mapean los jugadores en base a sus goles. Se almacena en un triple de manera que:

- Primer valor: Media de goles.
- Segundo valor: El número máximo de goles.
- Tercer valor: La variable maxGoles (que es el jugador con más goles)

```
println("25. Promedio de goles por posición, y dentro de cada posición, el jugador con el mayor número de goles")
service.getAll().filterIsInstance<Jugador>().groupBy { it.posicion }.mapValues { (_,jugadores) ->
    val maxGoles = jugadores.maxBy { it.goles }
    val goles = jugadores.map { it.goles }
    Triple(goles.average(), goles.maxOrNull(), maxGoles)
}.forEach { println(it) }

25. Promedio de goles por posición, y dentro de cada posición, el jugador con el mayor número de goles
09:55:50.630 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.630 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
DELANTERO=(39.25, 70, Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais =
PORTERO=(0.0, 0, Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Ale
CENTROCAMPISTA=(20.75, 30, Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pa
DEFENSA=(7.25, 10, Jugador(id= 8, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pai
```

26 - Se filtran todos los entrenadores y se agrupa por especialidad. Se mapean los valores del mapa resultante y se obtiene el entrenador con el salario más alto.

```
println("26. Entrenadores agrupados por especialidad, y dentro de cada especialidad, el entrenador con el salario más alto.")
service.getAll().filterIsInstance<Entrenador>().groupBy { it.especialidad }.mapValues { (_,entrenadores) -> entrenadores.maxBy { it.salario } }.forEach { println(it) }

26. Entrenadores agrupados por especialidad, y dentro de cada especialidad, el entrenador con el salario más alto.
09:55:50.630 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.630 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
ENTRENADOR_PRINCIPAL=Entrenador(id= 1, nombre= Roberto, apellidos= Hongo, fecha_nacimiento= 1960-07-17, fecha_incorporacion= 2000-01-01, salario= 60000.0, pais = Brasil
ENTRENADOR_PORTEROS=Entrenador(id= 4, nombre= Freddy, apellidos= Marshall, fecha_nacimiento= 1965-09-22, fecha_incorporacion= 2005-04-10, salario= 55000.0, pais = Españ
ENTRENADOR_ASISTENTE=Entrenador(id= 7, nombre= Jefferson, apellidos= Robles, fecha_nacimiento= 1968-12-12, fecha_incorporacion= 2003-11-11, salario= 58000.0, pais = Méx
```

27 - Se filtran los jugadores y se agrupan por la década, es decir, el año de nacimiento y se divide entre 10, y luego se vuelve a multiplicar por 10, de manera que forzamos a que las unidades siempre sean 0. Ejemplo 1985 -> 1980 -> 1980.

```
println("27. Jugadores agrupados por década de nacimiento, y dentro de cada grupo, el promedio de partidos jugados.")
service.getAll().filterIsInstance<Jugador>().groupBy { (it.fecha_nacimiento.year / 10) * 10 }.mapValues { (_, jugadores) -> jugadores.map { it.partidos_jugados }.average() }

27. Jugadores agrupados por década de nacimiento, y dentro de cada grupo, el promedio de partidos jugados.
09:55:50.630 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.630 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
1980=152.1875
```

28 - Se filtran los jugadores. Se agrupan por país de origen. Del mapa resultante de la agrupación, se transforman los valores en un triple formado por:

- Primer valor: Salario medio.
- Segundo valor: Jugador con el salario máximo.
- Tercer valor: Jugador con el salario mínimo.

```
println("28. Salario promedio de los jugadores agrupados por su país de origen, y dentro de cada grupo, el jugador con el salario más bajo y alto.")
service.getAll().filterIsInstance<Jugador>().groupBy { it.pais }.mapValues { (_, jugadores) -> Triple(jugadores.map { it.salario }.average(), jugadores.maxBy { it.salario }, jugadores.minBy { it.salario }) }.forEach { println(it) }

28. Salario promedio de los jugadores agrupados por su país de origen, y dentro de cada grupo, el jugador con el salario más bajo y alto.
09:55:50.631 [main] DEBUG org.example.service.ServiceImpl -- Obteniendo todos los integrantes del equipo
09:55:50.631 [main] DEBUG org.example.repositories.EquipoRepositoryImpl -- Obteniendo todos los integrantes
España=(30807.69230769231, Jugador(id= 2, nombre= Oliver, apellidos= Atom, fecha_nacimiento= 1983-04-10, fecha_incorporacion= 2001-05-15, salario= 35000.0, pais = España), Jugador(id= 3, nombre= Benji, apellidos= Price, fecha_nacimiento= 1983-11-07, fecha_incorporacion= 2001-05-15, salario= 34000.0, pais = Alemania), Jugador(id= 4, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra), Jugador(id= 5, nombre= Tom, apellidos= Baker, fecha_nacimiento= 1984-03-20, fecha_incorporacion= 2001-05-15, salario= 32000.0, pais = Inglaterra), Jugador(id= 6, nombre= Phillip, apellidos= Callahan, fecha_nacimiento= 1983-06-06, fecha_incorporacion= 2001-05-15, salario= 31000.0, pais = Italia))
```

13. Explicación teórico-práctica de cada prueba o conjunto de test relevantes realizados

Tests Cache:

```
@Test
fun cacheLRU() {

    //Llenamos la cache (tiene 3 de tamaño)
    cache.put("clave1", "valor1")
    cache.put("clave2", "valor2")
    cache.put("clave3", "valor3")

    //Consultamos clave2, así es el más recientemente usado
    cache.get("clave2")

    //Ahora, al añadir otro valor, se debería eliminar el menos utilizado (valor 1)
    cache.put("clave4", "valor4")

    assertNull(cache.get("clave1")) //No debería estar
    assertNotNull(cache.get("clave3")) //Debería estar
    assertNotNull(cache.get("clave2")) //Debería estar
    assertNotNull(cache.get("clave4")) //Debería estar
}
```

Verifica que la caché siga la filosofía LRU. Cuando se supera el tamaño máximo, el elemento menos recientemente usado es eliminado, y los demás siguen en la caché.

```
@Test  🐞 charliee.cy
fun eliminarElemento () {
    cache.put("clave1", "valor1")
    cache.remove( key: "clave1")
    assertNull(cache.get("clave1"))
}
```

Comprueba que un elemento específico puede eliminarse correctamente de la caché. Se verifica que después de la eliminación, el elemento no esté presente.

```
@Test  🐞 charliee.cy
fun limpiarCache() {
    cache.put("clave1", "valor1")
    cache.put("clave2", "valor2")
    cache.put("clave3", "valor3")

    cache.clear()

    assertNull(cache.get("clave1"))
    assertNull(cache.get("clave2"))
    assertNull(cache.get("clave3"))
}
```

Verifica que el método clear() vacíe toda la caché. Después de llamar a clear(), todos los elementos deben ser nulos.

```
@Test  🐞 charliee.cy
fun tamanoCache() {
    val size = cache.size()

    assertEquals( expected: 0, size)
}
```

Verifica que el tamaño de la caché sea 0 al inicio, asegurando que la caché está vacía.

Test Configuration:

```
@Test  Sggz221
@DisplayName("Leer la configuracion")
fun cargarConfiguracion() {
    val configurationProperties = Configuration.configurationProperties

    val expectedDataDir = Path.of(System.getProperty("user.dir"), ...more: "data").pathString
    val expectedBackupDir = Path.of(System.getProperty("user.dir"), ...more: "backup").pathString

    assertEquals(expectedDataDir, configurationProperties.dataDirectory)
    assertEquals(expectedBackupDir, configurationProperties.backupDirectory)

    assertTrue(Files.exists(Path.of(configurationProperties.dataDirectory)), message: "El directorio dataTest deberia existir")
    assertTrue(Files.exists(Path.of(configurationProperties.backupDirectory)), message: "El directorio backupTest deberia existir")
}
```



Se compara si los valores de los directorios (dataDirectory y backupDirectory) en configurationProperties coinciden con los valores esperados. Se valida que ambos directorios existan usando Files.exists().


```
@Test  Sggz221
@DisplayName("Leer configuracion con valores por defecto")
fun leerDeafaultValues(){
    val configDeafault = Configuration.configurationProperties

    val expectedDataDir = Path.of(System.getProperty("user.dir"), ...more: "data").pathString
    val expectedBackupDir = Path.of(System.getProperty("user.dir"), ...more: "backup").pathString

    assertEquals(expectedDataDir, configDeafault.dataDirectory)
    assertEquals(expectedBackupDir, configDeafault.backupDirectory)
}
```

Test Extensions:

```
class ExtensionsTest {  Sggz221
    @Test  Sggz221
    @DisplayName("Copy de Jugador y Entrenador")
    fun copyTest(){
        val jugador = Jugador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, month: 8,
        val entrenador = Entrenador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, mon

        val newJugador = jugador.copy()
 val newEntrenador = entrenador.copy()

        // assertAll de Jugador
        assertAll(
            { assertEquals(jugador.id, newJugador.id) },
            { assertEquals(jugador.nombre, newJugador.nombre) },
            { assertEquals(jugador.apellidos, newJugador.apellidos) },
            { assertEquals(jugador.fecha_nacimiento, newJugador.fecha_nacimiento) },
            { assertEquals(jugador.fecha_incorporacion, newJugador.fecha_incorporacion) },
            { assertEquals(jugador.salario, newJugador.salario) },
            { assertEquals(jugador.pais, newJugador.pais) },
            { assertEquals(jugador.posicion, newJugador.posicion) },
            { assertEquals(jugador.dorsal, newJugador.dorsal) },
            { assertEquals(jugador.altura, newJugador.altura) },
            { assertEquals(jugador.peso, newJugador.peso) },
            { assertEquals(jugador.goles, newJugador.goles) },
            { assertEquals(jugador.partidos_jugados, newJugador.partidos_jugados) }
        )

        // assertAll de Entrenador
        assertAll(
            { assertEquals(entrenador.id, newEntrenador.id) },
            { assertEquals(entrenador.nombre, newEntrenador.nombre) },
            { assertEquals(entrenador.apellidos, newEntrenador.apellidos) },
            { assertEquals(entrenador.fecha_nacimiento, newEntrenador.fecha_nacimiento) },
            { assertEquals(entrenador.fecha_incorporacion, newEntrenador.fecha_incorporacion) },
            { assertEquals(entrenador.salario, newEntrenador.salario) },
            { assertEquals(entrenador.pais, newEntrenador.pais) },
            { assertEquals(entrenador.especialidad, newEntrenador.especialidad) },
        )
    }
}
```

Se asegura de que la función `copy()` funcione correctamente, es decir, que no cambie ningún valor en las propiedades al crear una nueva instancia con los mismos datos. Se validan los valores predeterminados de los directorios (`dataDirectory` y `backupDirectory`) y se comparan con los valores esperados.

Test Mapper:

```
class IntegranteMapperTest { @Sggz221
    @DisplayName("Integrante to DTO")
    fun testToDTO() {
        val jugador = Jugador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, month: 8,
        val entrenador = Entrenador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, mon

        val jugadorDTO = jugador.toDto()
        val entrenadorDTO = entrenador.toDto()

        // assertAll de Jugador
        assertAll(
            { assertEquals(jugador.id, jugadorDTO.id) },
            { assertEquals(jugador.nombre, jugadorDTO.nombre) },
            { assertEquals(jugador.apellidos, jugadorDTO.apellidos) },
            { assertEquals(jugador.fecha_nacimiento.toString(), jugadorDTO.fecha_nacimiento) },
            { assertEquals(jugador.fecha_incorporacion.toString(), jugadorDTO.fecha_incorporacion) },
            { assertEquals(jugador.salario, jugadorDTO.salario) },
            { assertEquals(jugador.pais, jugadorDTO.pais) },
            { assertEquals(jugador.posicion.toString(), jugadorDTO.posicion) },
            { assertEquals(jugador.dorsal, jugadorDTO.dorsal) },
            { assertEquals(jugador.altura, jugadorDTO.altura) },
            { assertEquals(jugador.peso, jugadorDTO.peso) },
            { assertEquals(jugador.goles, jugadorDTO.goles) },
            { assertEquals(jugador.partidos_jugados, jugadorDTO.partidos_jugados) }
        )

        // assertAll de Entrenador
        assertAll(
            { assertEquals(entrenador.id, entrenadorDTO.id) },
            { assertEquals(entrenador.nombre, entrenadorDTO.nombre) },
            { assertEquals(entrenador.apellidos, entrenadorDTO.apellidos) },
            { assertEquals(entrenador.fecha_nacimiento.toString(), entrenadorDTO.fecha_nacimiento) },
            { assertEquals(entrenador.fecha_incorporacion.toString(), entrenadorDTO.fecha_incorporacion) },
            { assertEquals(entrenador.salario, entrenadorDTO.salario) },
            { assertEquals(entrenador.pais, entrenadorDTO.pais) },
            { assertEquals(entrenador.especialidad.toString(), entrenadorDTO.especialidad) },
        )
    }
}
```

Para cada uno de los modelos, se verifica que todos los atributos del objeto original coincidan con los del DTO. Se comprueba que las fechas, cadenas de texto, números y enum class sean iguales, y que las conversiones de objetos como Posicion y Especialidad se hagan bien.


```

@Test
@DisplayName("DTO to Model")
fun testToModel() {
    val jugadorDTO = IntegranteDTO( id: 1L, nombre: "Pepe", apellidos: "Garcia", fecha_nacimiento: "2003-08-12", f
    val entrenadorDTO = IntegranteDTO( id: 1L, nombre: "Pepe", apellidos: "Garcia", fecha_nacimiento: "2003-08-12"

    val jugador = jugadorDTO.toModel()
    val entrenador = entrenadorDTO.toModel()

    // assertAll de Jugador
    assertAll(
        { assertEquals(jugador.id, jugadorDTO.id) },
        { assertEquals(jugador.nombre, jugadorDTO.nombre) },
        { assertEquals(jugador.apellidos, jugadorDTO.apellidos) },
        { assertEquals(jugador.fecha_nacimiento.toString(), jugadorDTO.fecha_nacimiento) },
        { assertEquals(jugador.fecha_incorporacion.toString(), jugadorDTO.fecha_incorporacion) },
        { assertEquals(jugador.salario, jugadorDTO.salario) },
        { assertEquals(jugador.pais, jugadorDTO.pais) },
        { assertEquals( expected: "Jugador", jugadorDTO.rol) },
        { assertEquals((jugador as Jugador).posicion.toString(), jugadorDTO.posicion) },
        { assertEquals((jugador as Jugador).dorsal, jugadorDTO.dorsal) },
        { assertEquals((jugador as Jugador).altura, jugadorDTO.altura) },
        { assertEquals((jugador as Jugador).peso, jugadorDTO.peso) },
        { assertEquals((jugador as Jugador).goles, jugadorDTO.goles) },
        { assertEquals((jugador as Jugador).partidos_jugados, jugadorDTO.partidos_jugados) }
    )

    // assertAll de Entrenador
    assertAll(
        { assertEquals(entrenador.id, entrenadorDTO.id) },
        { assertEquals(entrenador.nombre, entrenadorDTO.nombre) },
        { assertEquals(entrenador.apellidos, entrenadorDTO.apellidos) },
        { assertEquals(entrenador.fecha_nacimiento.toString(), entrenadorDTO.fecha_nacimiento) },
        { assertEquals(entrenador.fecha_incorporacion.toString(), entrenadorDTO.fecha_incorporacion) },
        { assertEquals(entrenador.salario, entrenadorDTO.salario) },
        { assertEquals(entrenador.pais, entrenadorDTO.pais) },
        { assertEquals((entrenador as Entrenador).especialidad.toString(), entrenadorDTO.especialidad) },
    )
}

```

Se crea un DTO a partir de un modelo con datos específicos, y luego se verifica que después de convertirlo de nuevo a un objeto modelo, todos los valores sean los mismos.

```

@Test
@DisplayName("IntegranteXML to DTO")
fun testXmlToDTO() {
    val jugador = Jugador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, month: 8, day: 15 ),
    val entrenador = Entrenador( id: 1L, nombre: "Pepe", apellidos: "Garcia", LocalDate.of( year: 2003, month: 8, day: 15 ),

    val jugadorDTO = jugador.toXmlDTO()
    val entrenadorDTO = entrenador.toXmlDTO()

    // assertAll de Jugador
    assertAll(
        { assertEquals(jugador.id, jugadorDTO.id) },
        { assertEquals(jugador.nombre, jugadorDTO.nombre) },
        { assertEquals(jugador.apellidos, jugadorDTO.apellidos) },
        { assertEquals(jugador.fecha_nacimiento.toString(), jugadorDTO.fecha_nacimiento) },
        { assertEquals(jugador.fecha_incorporacion.toString(), jugadorDTO.fecha_incorporacion) },
        { assertEquals(jugador.salario, jugadorDTO.salario) },
        { assertEquals(jugador.pais, jugadorDTO.pais) },
        { assertEquals(jugador.posicion.toString(), jugadorDTO.posicion) },
        { assertEquals(jugador.dorsal.toString(), jugadorDTO.dorsal) },
        { assertEquals(jugador.altura.toString(), jugadorDTO.altura) },
        { assertEquals(jugador.peso.toString(), jugadorDTO.peso) },
        { assertEquals(jugador.goles.toString(), jugadorDTO.goles) },
        { assertEquals(jugador.partidos_jugados.toString(), jugadorDTO.partidos_jugados) }
    )

    // assertAll de Entrenador
    assertAll(
        { assertEquals(entrenador.id, entrenadorDTO.id) },
        { assertEquals(entrenador.nombre, entrenadorDTO.nombre) },
        { assertEquals(entrenador.apellidos, entrenadorDTO.apellidos) },
        { assertEquals(entrenador.fecha_nacimiento.toString(), entrenadorDTO.fecha_nacimiento) },
        { assertEquals(entrenador.fecha_incorporacion.toString(), entrenadorDTO.fecha_incorporacion) },
        { assertEquals(entrenador.salario, entrenadorDTO.salario) },
        { assertEquals(entrenador.pais, entrenadorDTO.pais) },
        { assertEquals(entrenador.especialidad.toString(), entrenadorDTO.especialidad) },
    )
}

```

Se valida que la conversión desde un modelo XML de los objetos Jugador y Entrenador a su DTO se realice correctamente. Se aseguran las conversiones de valores, incluyendo las propiedades que pueden estar representadas de manera diferente en XML.

```

fun testDtoToXML() {
    val jugadorDTO = IntegranteXmlDTO( id: 1L, rol: "Jugador", nombre: "Pepe", apellidos: "Garcia", fecha_nacimiento: ... )
    val entrenadorDTO = IntegranteXmlDTO( id: 1L, rol: "Entrenador", nombre: "Pepe", apellidos: "Garcia", fecha_na...

    val jugador = jugadorDTO.toModel()
    val entrenador = entrenadorDTO.toModel()

    // assertAll de Jugador
    assertAll(
        { assertEquals(jugador.id, jugadorDTO.id) },
        { assertEquals(jugador.nombre, jugadorDTO.nombre) },
        { assertEquals(jugador.apellidos, jugadorDTO.apellidos) },
        { assertEquals(jugador.fecha_nacimiento.toString(), jugadorDTO.fecha_nacimiento) },
        { assertEquals(jugador.fecha_incorporacion.toString(), jugadorDTO.fecha_incorporacion) },
        { assertEquals(jugador.salario, jugadorDTO.salario) },
        { assertEquals(jugador.pais, jugadorDTO.pais) },
        { assertEquals( expected: "Jugador", jugadorDTO.rol) },
        { assertEquals((jugador as Jugador).posicion.toString(), jugadorDTO.posicion) },
        { assertEquals((jugador as Jugador).dorsal.toString(), jugadorDTO.dorsal) },
        { assertEquals((jugador as Jugador).altura.toString(), jugadorDTO.altura) },
        { assertEquals((jugador as Jugador).peso.toString(), jugadorDTO.peso) },
        { assertEquals((jugador as Jugador).goles.toString(), jugadorDTO.goles) },
        { assertEquals((jugador as Jugador).partidos_jugados.toString(), jugadorDTO.partidos_jugados) }
    )

    // assertAll de Entrenador
    assertAll(
        { assertEquals(entrenador.id, entrenadorDTO.id) },
        { assertEquals(entrenador.nombre, entrenadorDTO.nombre) },
        { assertEquals(entrenador.apellidos, entrenadorDTO.apellidos) },
        { assertEquals(entrenador.fecha_nacimiento.toString(), entrenadorDTO.fecha_nacimiento) },
        { assertEquals(entrenador.fecha_incorporacion.toString(), entrenadorDTO.fecha_incorporacion) },
        { assertEquals(entrenador.salario, entrenadorDTO.salario) },
        { assertEquals(entrenador.pais, entrenadorDTO.pais) },
        { assertEquals((entrenador as Entrenador).especialidad.toString(), entrenadorDTO.especialidad) },
    )
}

```

Se valida la conversión de un DTO a su modelo XML. Se asegura de que todos los campos estén correctamente representados, prestando especial atención a las conversiones entre tipos como fechas, cadenas y números, para asegurarse de que el objeto DTO se traduzca correctamente al formato XML.

Test Repositorio:

```

fun save() {
    val equip = EquipoRepositoryImpl()
    val jugadorGuardado = equip.save(jugador)
    val jugadorGuardado1 = equip.save(jugador)

    val entrenadorGuardado = equip.save(entrenador)

    assertEquals(jugadorGuardado1.id, jugadorGuardado.id)
    assertEquals(entrenadorGuardado.id, actual: 3)
}

```

Verifica que los objetos Jugador y Entrenador se guarden correctamente en el repositorio.

```

@Test  🐘 JesusCoboArrogante
fun delete() {
    val equip = EquipoRepositoryImpl()
    val equipoGuardado = equip.save(jugador)
    val equipoEliminado = equip.delete(equipoGuardado.id)
    assertNotNull(equipoEliminado)

    var equipo = equip.getById( id: 1)
    assertNull(equipo)
}

```

Se verifica que un jugador o entrenador se elimine correctamente del repositorio.

```

@Test  🐘 JesusCoboArrogante
fun update() {
    val equip = EquipoRepositoryImpl()
    val cambio1 = equip.save(jugador)
    Thread.sleep( millis: 1)
    val cambio2 = equip.update( id: 1,jugador)
    if (cambio2 != null) {
        assertEquals(cambio1.updatedAt, cambio2.updatedAt)
    }
}

```

Se verifica que un jugador o entrenador se actualice correctamente en el repositorio.

```

@Test  🐘 JesusCoboArrogante
fun actualizarNull(){
    val equip = EquipoRepositoryImpl()
    val cambio1 = equip.save(jugador)

    assertNull(equip.update( id: 2,cambio1))
}

```

Se verifica que cuando se intente actualizar un jugador con un ID no existente (2 en este caso), se retorna null.

```

@Test  🐘 JesusCoboArrogante
fun getAll() {
    val equip = EquipoRepositoryImpl()
    var equipo = equip.getAll()
    assertTrue(equipo.isEmpty())
}

```

Se verifica que el método getAll() devuelve la lista correcta de jugadores y entrenadores.

```

@Test  🐘 JesusCoboArrogante
fun getById() {
    val equip = EquipoRepositoryImpl()
    equip.save(jugador)
    var equipo = equip.getById( id: 1)
    assertNotNull(equipo)

    equipo = equip.getById( id: 5)
    assertNull(equipo)
}

```

Verificar que el método getById() obtenga correctamente un jugador o entrenador por su ID.

Test Service:

```
@Test ㄟ Sggz221
fun importAndExportFromFileCSV() {
    val service = ServiceImpl()

    val fileFrom = File( parent: "data", child: "personal.csv")
    val fileTo = File( parent: "data", child: "personalOutput.csv")
    service.importFromFile(fileFrom.path)
    service.exportToFile(fileTo.path)
}

@Test ㄟ Sggz221
fun importAndExportFromFileXML() {
    val service = ServiceImpl()

    val fileFrom = File( parent: "data", child: "personal.xml")
    val fileTo = File( parent: "data", child: "personalOutput.xml")
    service.importFromFile(fileFrom.path)
    service.exportToFile(fileTo.path)
}

@Test ㄟ Sggz221
fun importAndExportFromFileJSON() {
    val service = ServiceImpl()

    val fileFrom = File( parent: "data", child: "personal.json")
    val fileTo = File( parent: "data", child: "personalOutput.json")
    service.importFromFile(fileFrom.path)
    service.exportToFile(fileTo.path)
}

@Test ㄟ Sggz221
fun importAndExportFromFileBIN() {
    val service = ServiceImpl()

    val fileFrom = File( parent: "data", child: "personal.bin")
    val fileTo = File( parent: "data", child: "personalOutput.bin")
    service.importFromFile(fileFrom.path)
    service.exportToFile(fileTo.path)
}
```

Estos test verifican que los archivos se puedan importar y exportar correctamente.

Test Storage:

```
@Test ㄟ charliee.cy
fun fileReadThrowsFileNotExistsException(){
    val file = File( pathname: "noExiste.txt")

    val expected = "Error en el storage: El fichero no existe, la ruta especificada no es un fichero o no se tienen permisos de lectura"
    val actual = assertThrows<Exceptions.StorageException> { storage.fileRead(file) }

    assertEquals(expected, actual.message)
}
```

Esta prueba verifica si se lanza una excepción cuando intentamos leer un archivo que no existe. Se espera que se lance una excepción de tipo `Exceptions.StorageException` con un mensaje de error específico.

```

fun fileWriteThrowsFileNotExistsException(@TempDir tempDir: File) {
    val file = File(tempDir, child: "fhgdgdh/noExiste.txt")

    val entrenador = Entrenador(
        id = 5,
        nombre = "Tom",
        apellidos = "Baker",
        fecha_nacimiento = LocalDate.parse( text: "1984-03-20"),
        fecha_incorporacion = LocalDate.parse( text: "2001-05-15"),
        salario = 32000.0,
        pais = "Inglaterra",
        especialidad = Especialidad.ENTRENADOR_PRINCIPAL
    )
    val equipo = listOf(entrenador)

    val expected = "Error en el storage: El directorio padre del fichero no existe"
    val actual = assertThrows<Exceptions.StorageException> { storage.fileWrite(equipo, file) }

    assertEquals(expected, actual.message)
}

```

Esta prueba asegura que si intentamos escribir en un directorio que no existe (usando un directorio temporal para las pruebas), se lanza una excepción.

```

fun importOk(@TempDir tempDir: File){
    val equipo = storage.fileRead(file)

    assertEquals( expected: 1, equipo.size)

    val jugadorExpected = Jugador(
        id = 5,
        nombre = "Tom",
        apellidos = "Baker",
        fecha_nacimiento = LocalDate.parse( text: "1984-03-20"),
        fecha_incorporacion = LocalDate.parse( text: "2001-05-15"),
        salario = 32000.0,
        pais = "Inglaterra",
        posicion = Posicion.CENTROCAMPISTA,
        dorsal = 8,
        altura = 1.72,
        peso = 63.0,
        goles = 30,
        partidos_jugados = 140
    )

    val jugadorActual = equipo.first()

    org.junit.jupiter.api.assertAll(
        { assertEquals(jugadorExpected.id, jugadorActual.id) },
        { assertEquals(jugadorExpected.nombre, jugadorActual.nombre) },
        { assertEquals(jugadorExpected.apellidos, jugadorActual.apellidos) },
        { assertEquals(jugadorExpected.fecha_nacimiento, jugadorActual.fecha_nacimiento) },
        { assertEquals(jugadorExpected.fecha_incorporacion, jugadorActual.fecha_incorporacion) },
        { assertEquals(jugadorExpected.salario, jugadorActual.salario) },
        { assertEquals(jugadorExpected.pais, jugadorActual.pais) },
        { assertEquals(jugadorExpected.posicion, (jugadorActual as Jugador).posicion) },
        { assertEquals(jugadorExpected.dorsal, (jugadorActual as Jugador).dorsal) },
        { assertEquals(jugadorExpected.altura, (jugadorActual as Jugador).altura) },
        { assertEquals(jugadorExpected.peso, (jugadorActual as Jugador).peso) },
        { assertEquals(jugadorExpected.goles, (jugadorActual as Jugador).goles) },
        { assertEquals(jugadorExpected.partidos_jugados, (jugadorActual as Jugador).partidos_jugados) }
    )
}

```

Se verifica que el método `fileRead` funcione correctamente. Se escribe un archivo CSV de ejemplo con datos de un jugador (Jugador) y luego se lee en un objeto equipo (equipo). La prueba asegura que los datos leídos del archivo coinciden con el objeto Jugador esperado.

```
fun exportOk (@TempDir tempDir: File){
    fecha_nacimiento = LocalDate.parse( text: "1984-03-20"),
    fecha_incorporacion = LocalDate.parse( text: "2001-05-15"),
    salario = 32000.0,
    pais = "Inglaterra",
    posicion = Posicion.CENTROCAMPISTA,
    dorsal = 8,
    altura = 1.72,
    peso = 63.0,
    goles = 30,
    partidos_jugados = 140
)

val entrenador = Entrenador(
    id = 5,
    nombre = "Tom",
    apellidos = "Baker",
    fecha_nacimiento = LocalDate.parse( text: "1984-03-20"),
    fecha_incorporacion = LocalDate.parse( text: "2001-05-15"),
    salario = 32000.0,
    pais = "Inglaterra",
    especialidad = Especialidad.ENTRENADOR_PRINCIPAL
)

val equipo = listOf(jugador,entrenador)

storage.fileWrite(equipo, file)

val expectedString = "id,nombre,apellidos,fecha_nacimiento,fecha_incorporacion,sala
    "5,Tom,Baker,1984-03-20,2001-05-15,32000.0,Inglaterra,Jugador,,CENTROCAMPISTA
    "5,Tom,Baker,1984-03-20,2001-05-15,32000.0,Inglaterra,Entrenador,ENTRENADOR
    "5,Tom,Baker,1984-03-20,2001-05-15,32000.0,Inglaterra,Entrenador,ENTRENADOR

val actualString = file.readText()

assertEquals(expectedString, actualString)
}
```

Se verifica que el método `fileWrite` funcione correctamente. Se crea un equipo de ejemplo con un jugador (Jugador) y un entrenador (Entrenador), y luego se escribe esta información en un archivo. La prueba asegura que el contenido del archivo coincida con el formato CSV esperado.

El resto de test del storage son las mismas pruebas vistas adaptadas al formato del archivo que maneja ese Storage.

14. Captura de la cobertura del código en test

Element ▾	Class, %	Method, %	Line, %	Branch, %
▼ org.example	87% (28/32)	74% (74/100)	62% (571/9...	32% (128/3...
MainKt	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
> view	0% (0/1)	0% (0/20)	0% (0/216)	0% (0/213)
> validator	100% (1/1)	100% (3/3)	100% (30/3...	100% (34/...
> storage	100% (4/4)	100% (12/12)	98% (152/1...	75% (50/66)
> service	100% (2/2)	100% (10/10)	98% (68/69)	88% (23/26)
> repositories	100% (1/1)	100% (8/8)	100% (36/3...	87% (7/8)
> models	100% (5/5)	71% (5/7)	87% (28/32)	100% (0/0)
> mapper	100% (1/1)	100% (6/6)	100% (112/...	100% (4/4)
> extensions	100% (1/1)	100% (2/2)	100% (54/5...	100% (0/0)
> exceptions	100% (4/4)	100% (4/4)	100% (4/4)	100% (0/0)
> dto	83% (5/6)	88% (8/9)	97% (39/40)	50% (3/6)
> consults	0% (0/1)	0% (0/2)	0% (0/108)	0% (0/24)
> configuration	100% (2/2)	100% (5/5)	96% (25/26)	50% (5/10)
> Cache	100% (2/2)	100% (11/11)	100% (23/23)	100% (2/2)

15. Estimación económica de la ejecución del proyecto

HORAS DE TRABAJO ESTIMADAS		
Requisitos funcionales		
RF	Nombre	Horas estimadas
RF1	Operaciones CRUD	4
RF2	Id único	0,5
RF3	Modelos (jugador y entrenador)	2
RF4	Campos comunes	1
RF5	Campos específicos jugador	2
RF6	Campos específicos entrenador	2
RF7	Validador de datos	4
RF9	Importación y exportación CSV, JSON, XML, BIN	12
RF10	Cache LRU	5
RF11	Localización según fichero de configuración	4
RF12	Menú para el usuario	10
RF 13	Consultas	5
		51,5

Requisitos no funcionales		
RNF	Nombre	Horas estimadas
RNF1	Validación de datos	2
RNF2	Rapidez de acceso mediante caché	4
RNF3	Facilidad de uso del menú	3
RNF4	Logs para debug	2
RNF5	Documentación del código	7
RNF6	Tests	12
RNF7	.jar ejecutable	2
RNF8	Uso de GitFlow/Git/Pull Request	15
		47

Requisitos de información		
RI	Nombre	Horas estimadas
RI1	Campos comunes	5
RI2	Campos de jugador	6
RI3	Campos de entrenador	1
		12

HORAS DE TRABAJO TOTALES	
Suma de los requisitos funcionales, no funcionales y de información	110,5

PRECIO POR HORA DE TRABAJO	
Incluye salario del trabajador y gastos de empresa	135,00 €

PRECIO POR HORA EXTRA DE TRABAJO	
Incluye salario del trabajador y gastos de empresa	160,00 €

COSTE TOTAL DE LAS HORAS DE TRABAJO	
Horas totales * precio / hora	14.917,50 €

ESTIMACIÓN DE COSTES				
Nº	Concepto	Desglose		Coste
		Coste por trabajador	Nº trabajadores	
1	Licencias de software (IDEs, GitHub premium, aplicaciones auxiliares...)	700,00 €	3	2.100,00 €
		Precio por hora	Horas	
2	Horas de trabajo estimadas por requisitos	135,00 €	110,5	14.917,50 €
		Precio por hora extra	15% de las horas totales	
3	Horas extra para cubrir imprevistos	160,00 €	16,58	176,58 €
		Costes antes de margen	15% sobre los costes	
4	Margen de beneficios	17.194,08 €	0,15	2.579,11 €
				19.773,19 €