

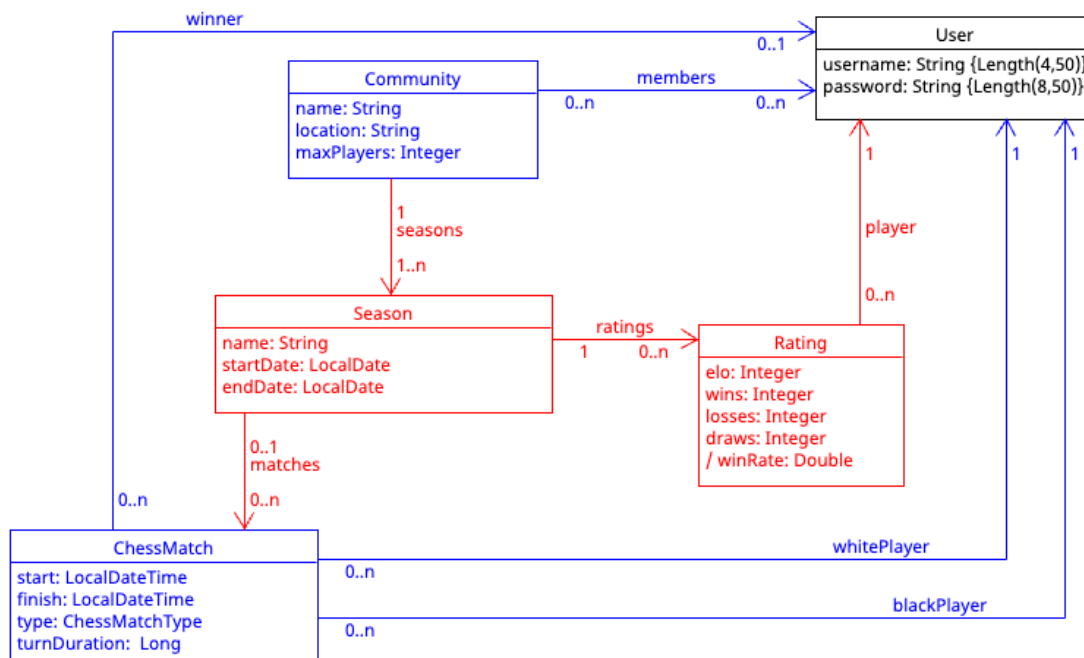
## Control práctico de DP1 2025-2026 (Control-check 1)

### Enunciado

En este ejercicio añadiremos la funcionalidad de **gestión de comunidades de jugadores, temporadas de competición y las puntuaciones obtenidas en estas** dentro de una aplicación web de ajedrez.

Concretamente, se proporciona una clase *ChessMatch* que representa las partidas jugadas entre dos usuarios registrados en la plataforma. Cada partida se asocia a los jugadores que la disputan y registra el ganador de la partida, si existe (la partida puede terminar en empate). Además, se incluyen las clases *User* y *Community*, que representan a los usuarios del sistema (jugadores) y las comunidades de jugadores en las que participan, respectivamente. Para cada comunidad queremos gestionar las puntuaciones obtenidas en cada temporada a modo de estadísticas, representadas mediante las clases *Season* (temporada) y *Rating* (puntuaciones). Finalmente, se ha definido una relación que indica qué partidas se contabilizan dentro de una temporada concreta (llamada *matches*), lo que permitirá gestionar las estadísticas por temporada.

El diagrama UML que describe las clases y relaciones con las que vamos a trabajar es el siguiente:



Las clases para las que realizaremos el mapeo objeto-relacional como entidades JPA se han señalado en rojo. Las clases en azul son clases que se proporcionan ya mapeadas, pero con las que se trabajará durante el control de laboratorio.

Realizaremos una serie de ejercicios basados en funcionalidades que implementaremos en el sistema, y validaremos mediante pruebas unitarias. Si desea ver el resultado que arrojarían las pruebas en backend, puede ejecutarlas (bien mediante su entorno de desarrollo favorito, bien mediante el comando `mvnw test` en la carpeta raíz del proyecto). Cada ejercicio correctamente resuelto valdrá dos puntos, el número de casos de prueba de cada ejercicio puede variar entre uno y otro y la nota se calculará en base al

porcentaje de casos de prueba que pasan. Por ejemplo, si pasan la mitad (50%) de los casos de prueba de un ejercicio, usted obtendrá un punto ( $2*0,5=1$ ), y si pasan un 10% obtendrá 0,2 puntos.

Al aceptar dicha tarea, se creará un repositorio único individual para usted, debe usar dicho repositorio para realizar el control práctico. Debe entregar la actividad en EV asociada al control check proporcionando como texto la dirección url de su repositorio personal. Recuerde que además debe entregar su solución del control.

**La entrega de su solución al control se realizará mediante un único comando “*git push*” a su repositorio individual.** Recuerde que debe hacer push antes de cerrar sesión en la computadora y abandonar el aula, de lo contrario, su intento se evaluará como no presentado. Su primera tarea en este control será clonar (recuerde que si va a usar los equipos del aula para realizar el control necesitará usar un token de autenticación de GitHub como clave, tiene un documento de ayuda a la configuración en el propio repositorio del control). A continuación, deberá importar el proyecto en su entorno de desarrollo favorito y comenzar los ejercicios abajo listados. Al importar el proyecto, el mismo puede presentar errores de compilación. No se preocupe, si existen, dichos errores irán desapareciendo conforme usted vaya implementando los distintos ejercicios del control.

**Nota importante 1:** No modifique los nombres de las clases ni la signature (nombre, tipo de respuesta y parámetros) de los métodos proporcionados como material de base para el control. Las pruebas que se usan para la evaluación dependen de que las clases y los métodos tengan la estructura y nombres proporcionados. Si los modifica probablemente no pueda hacer que pasen las pruebas.

**Nota importante 2:** No modifique las pruebas unitarias proporcionadas como parte del proyecto bajo ningún concepto. Aunque modifique las pruebas en su copia local del proyecto, éstas serán restituidas mediante un comando git previamente a la ejecución de las pruebas para la emisión de la nota final, por lo que sus modificaciones en las pruebas no serán tenidas en cuenta en ningún momento.

**Nota importante 3:** Mientras haya ejercicios no resueltos habrá tests que no funcionen y, por tanto, el comando “*mvnw install*” finalizará con error. Esto es normal debido a la forma en la que está planteado el control y no hay que preocuparse por ello. Si se quiere probar la aplicación se puede ejecutar de la forma habitual pese a que “*mvnw install*” finalice con error.

**Nota importante 4:** La descarga del material de la prueba usando git, y la entrega de su solución con git a través del repositorio GitHub creado a tal efecto forman parte de las competencias evaluadas durante el examen, por lo que no se aceptarán entregas que no hagan uso de este medio, y no se podrá solicitar ayuda a los profesores para realizar estas tareas.

**Nota importante 5:** No se aceptarán como soluciones válidas proyectos cuyo código fuente no compile correctamente o que provoquen fallos al arrancar la aplicación en la inicialización del contexto de Spring. Las soluciones cuyo código fuente no compile o incapaces de arrancar el contexto de Spring serán evaluadas con una nota de 0.

**Nota importante 6:** Excepto el ejercicio 2 (que depende del 1), los ejercicios del examen son independientes y pueden ser resueltos en cualquier orden.

## Test 1 – Creación de las entidades Season y Rating y sus repositorios asociados

### Parte 1A: Entidades y repositorios (1 punto)

Modificar las clases “Season” y “Rating” para que sean entidades. Estas clases están alojadas en el paquete “us.es.dp1.chess.tournament.match”, y deben tener los siguientes atributos y restricciones:

Para la clase Season:

- Un atributo de tipo entero (Integer) llamado “id” actuará como clave primaria en la tabla de la base de datos relacional asociada a la entidad.
- Un atributo de tipo cadena de caracteres (String) llamado “name” obligatorio (no puede ser nulo), que debe tener una longitud mínima de 3 caracteres y máxima de 50 y que no puede estar formada por caracteres vacíos (espacios, tabuladores, etc.).  
Un atributo de tipo fecha (LocalDate) llamado “startDate” obligatorio, que representa la fecha de comienzo de la temporada.
- Un atributo de tipo fecha (LocalDate) llamado “endDate” obligatorio, que representa la fecha de fin de la temporada.

Para la clase Rating:

- Un atributo de tipo entero (Integer) llamado “id” actuará como clave primaria en la tabla de la base de datos relacional asociada a la entidad.
- Un atributo “elo” de tipo entero (Integer) que es obligatorio y mayor o igual que 100, que representa la puntuación del jugador durante una temporada siguiendo el sistema ELO.
- Un atributo “wins” de tipo entero (Integer) que es obligatorio y mayor o igual que 0, que representa el número de partidas ganadas del jugador durante una temporada.
- Un atributo “losses” de tipo entero (Integer) que es obligatorio y mayor o igual que 0, que representa el número de partidas perdidas del jugador durante una temporada.
- Un atributo “draws” de tipo entero (Integer) que es obligatorio y mayor o igual que 0, que representa el número de partidas empatadas (o tablas) del jugador durante una temporada.
- Un método “getWinRate()” de tipo flotante (Double) debe devolver la relación entre partidas ganadas (wins) y partidas totales (wins+losses). Este método representa el atributo derivado “/winRate” del diagrama UML de la primera página, por lo que debe anotarse apropiadamente para que sea volátil (que no se persista en la base de datos).

Modificar las interfaces “SeasonRepository” y “RatingRepository” alojadas en el mismo paquete para que extiendan a CrudRepository. No olvide especificar sus parámetros de tipo.

### Parte 1B: Relaciones/asociaciones entre entidades (1 punto)

Elimine las anotaciones @Transient de los atributos “matches”, “ratings” y “player” de las entidades Season y Rating. Se pide crear las siguientes relaciones entre las entidades. Cree una relación unidireccional desde “Community” hacia “Season” que exprese la que aparece en el diagrama UML (mostrado en la primera página de este enunciado), usando el atributo “seasons”, respetando sus cardinalidades y eliminando la anotación @Transient.

Además, se pide crear una relación unidireccional desde “Season” hacia “ChessMatch” que represente la que aparece en el diagrama UML, tenga en cuenta la cardinalidad que tiene, usando el atributo “matches”

en la clase “Season”. Debe asegurarse de que las relaciones expresan adecuadamente la cardinalidad que muestra el diagrama UML.

Finalmente, se pide crear las dos relaciones unidireccionales desde “Season” hacia “Rating” y desde “Rating” hacia “User” que representen las que aparecen en el diagrama, usando como nombres de atributos “ratings” y “player”, respectivamente. Debe asegurarse de que las relaciones expresan adecuadamente la cardinalidad que muestra el diagrama UML, por ejemplo, algunos atributos pueden ser nulos puesto que la cardinalidad es 0..n pero otros no, porque su cardinalidad en el extremo navegable de la relación es 1..n

## Test 2 – Modificación del script de inicialización de la base de datos para incluir dos temporadas, 6 puntuaciones, y sus relaciones

Modificar el script de inicialización de la base de datos, para que se creen las siguientes temporadas (Season) y puntuaciones (Rating):

Season 1:

- id: 1
- name: “Juegos Municipales 2024-25”
- startDate: 01-09-2024
- endDate: 31-05-2025

Season 2:

- id: 2
- name: “Temporada Oficial 2025-26”
- startDate: 15-10-2025
- endDate: 15-06-2026

Rating 1:

- id: 1
- elo: 1560
- wins: 4
- losses: 2
- draws: 1

Rating 2:

- id: 2
- elo: 1250
- wins: 2
- losses: 2
- draws: 0

Rating 3:

- id: 3

- elo: 1000
- wins: 0
- losses: 0
- draws: 4

Rating 4:

- id: 4
- elo: 2080
- wins: 8
- losses: 1
- draws: 2

Rating 5:

- id: 5
- elo: 1310
- wins: 4
- losses: 6
- draws: 1

Rating 6:

- id: 6
- elo: 1567
- wins: 2
- losses: 0
- draws: 0

Además, debe modificar este script de inicialización de la base de datos para que:

- El *Season* cuyo id es 1 se asocie con los *ratings* con id 1, 2 y 3.
- El *Season* cuyo id es 1 se asocie con las partidas (*ChessMatch*) con id 1, 2 y 3
- El *Season* cuyo id es 2 se asocie con los *ratings* con id 4, 5 y 6
- El *Season* cuyo id es 2 se asocie con las partidas (*ChessMatch*) con id 4, 5 y 6
- La comunidad (*Community*) cuyo id es 1 se asocie con las *seasons* con id 1 y 2.
- El *Rating* cuyo id es 1 tenga como jugador asociado (*User*) aquel cuyo id es 4.
- El *Rating* cuyo id es 2 tenga como jugador asociado (*User*) aquel cuyo id es 5.
- El *Rating* cuyo id es 3 tenga como jugador asociado (*User*) aquel cuyo id es 6.
- El *Rating* cuyo id es 4 tenga como jugador asociado (*User*) aquel cuyo id es 4.
- El *Rating* cuyo id es 5 tenga como jugador asociado (*User*) aquel cuyo id es 5.
- El *Rating* cuyo id es 6 tenga como jugador asociado (*User*) aquel cuyo id es 6.

Tenga en cuenta que el orden en que aparecen los INSERT en el script de inicialización de la base de datos es relevante al definir las asociaciones.

### Test 3 – Modificar el servicio de gestión de Comunidades para que no permita guardar comunidades inválidas (regla de negocio) (2 puntos)

Modificar el método `save` del servicio de gestión de comunidades (*CommunityService*) de manera que se lance la excepción (*MalformedCommunityException* alojada en el paquete *“us.es.dp1.chess.tournament.exceptions”*) en caso de que se intente guardar una comunidad que incumpla alguna de las siguientes reglas de negocio:

- Los jugadores que son miembros de la comunidad (expresado a través de la propiedad *members*) deben haber participado en **al menos dos partidas** asociadas a alguna de las temporadas de esa comunidad (accediendo a través de la propiedad *seasons* de la comunidad y de *matches* para cada temporada). Para considerar que un jugador ha participado en una partida, dicho jugador debe ser uno de los dos jugadores que han participado en ella (expresados a través de las propiedades *whitePlayer* y *blackPlayer*).
- El número de jugadores que son miembros de la comunidad no es superior al máximo especificado (expresado a través de la propiedad *maxPlayers*).

El método debe ser transaccional y hacer *rollback* de cualquier cambio si se lanza dicha excepción.

### Test 4 – Creación de servicios de gestión de temporadas y puntuaciones, y del controlador de temporadas

Parte 4A: Servicio de gestión y repositorio (1 punto):

Modificar la clase *“SeasonService”* y la interfaz *“SeasonRepository”*, para que sean un servicio Spring de lógica de negocio y un repositorio de Spring Data. Además, se debe proporcionar una implementación a los métodos del servicio que use el repositorio:

Para el caso del servicio de gestión de temporadas debe permitirse:

1. Obtener un listado ordenado con la clasificación de los jugadores que participan en una temporada según su puntuación ELO (primero el que más tenga) extraída del campo *elo* de la clase *Rating* (método `getStandings(Season s)`).
2. Obtener todas las temporadas existentes (método `getAll`).
3. Obtener una temporada concreta por id (método `getById`).
4. Grabar una *temporada* en la base de datos (método `save`).

Todos estos métodos de la clase *SeasonService* **deben ser transaccionales**, pero las anotaciones asociadas deben realizarse a nivel de método, no a nivel de clase. El repositorio *“SeasonRepository”* deberá implementar las consultas correspondientes que considere necesarias.

Parte 4B: Controlador para API de temporadas (1 punto):

Modificar la clase *“SeasonController”* para que sea un controlador de Spring. Crear dentro del controlador un método que permita devolver todas las temporadas existentes. El método debe responder a peticiones tipo GET en la URL:

<http://localhost:8080/api/v1/seasons>

Así mismo debe crear un método para devolver la clasificación actual de una temporada concreta, este método debe responder a peticiones en la url <http://localhost:8080/api/v1/seasons/X/standings> donde X

es la Id de la temporada de la que obtener la clasificación, y devolverá un listado ordenado con la clasificación de los jugadores que participan en una temporada según su puntuación ELO (primero el que más tenga). En caso de que no exista una temporada con el id especificado, el sistema debe devolver el código de estado 404 (NOT\_FOUND).

Estos endpoint de la API asociados a la gestión de temporadas deberían estar accesibles únicamente para usuarios de tipo Player (que tengan la authority "PLAYER"), devolviendo un código 403 (FORBIDDEN) en caso contrario.

### Test 5 – Creación de un componente React de listado de clasificación de una temporada (2 puntos)

Modificar el componente React proporcionado en el fichero "frontend/src/standings/index.js" para que muestre la clasificación de una temporada cuyo id X se pase como parámetro. Para ello debe hacer uso de la API lanzando una petición tipo GET contra la URL </api/v1/seasons/X/standings>.

Si el conjunto de puntuaciones (*ratings*) de la temporada no está vacío, este componente debe mostrar una tabla con las puntuaciones de cada jugador en una tabla (se recomienda usar el componente Table de reactstrap) incluyendo columnas para el nombre del jugador (*username*), su puntuación (elo), partidas ganadas (*wins*), partidas perdidas (*losses*), partidas empatadas (*draws*) y porcentaje de victorias (*winRate*). Los títulos de las cabeceras de las columnas de la tabla deben ser "Player", "ELO", "Wins", "Losses", "Draws", y "Win %" (y en ese orden).

Si el conjunto de puntuaciones devuelto por la API es un array vacío, el componente debe mostrar un título de nivel 1 (h1), con el texto "NOT ENOUGH MATCHES", en lugar de la tabla (no debe aparecer la tabla vacía).

Para poder lanzar esta prueba y comprobar su resultado puede colocarse en la carpeta de frontend y ejecutar el comando npm test y pulsar 'a' en el menú de comandos de jest. Nótese que previamente debe haber lanzado al menos una vez el comando npm install para que todas las librerías de node estén instaladas.