Facultad Regional Mar del Plata
Carrera: Tecnicatura Universitaria en Programación (TUP)
Materia: Programación II
Trabajo Práctico Grupal Final

Carátula

- **Integrantes del Equipo:**
    - Lang, Rodrigo
    - Santalla, Ezequiel
    - Torres, Jeremías
    - Werner, Marcos
- **Fecha de Entrega:** 19 de noviembre de 2024

Resumen

Descripción del Sistema

Este sistema es una aplicación para gestionar un torneo de tenis, permitiendo crear torneos, registrar jugadores, crear rondas, asignar partidos y consultar estadísticas. Está diseñado para ser fácil de usar y mostrar información clave de manera clara.

Proceso de Desarrollo

El desarrollo del trabajo práctico se realizó de forma colaborativa, dividiendo las tareas según las habilidades y preferencias de cada integrante del grupo:

- **Rodrigo:** Se encargó de implementar la persistencia de datos utilizando archivos JSON y del diseño e implementación de la clase Torneo, asegurando su correcto funcionamiento.
- **Jeremías:** Fue responsable de la gestión de los partidos y rondas, desarrollando las clases y métodos necesarios para manejar estas entidades dentro del sistema.
- **Ezequiel:** Se enfocó en la implementación de los jugadores, diseñando y programando la clase Jugador y su integración con el resto del sistema.
- **Marcos:** Gestionó la metodología de trabajo utilizando Trello, organizando las tareas y asegurándose de que el grupo trabajara de manera coordinada y cumpliendo los plazos establecidos.

Informe Técnico

1. Descripción General del Sistema

El sistema desarrollado es una aplicación para la gestión de un torneo de tenis, diseñada para simplificar la organización y el seguimiento de sus diversas etapas. Permite a los usuarios gestionar jugadores, crear y administrar rondas y partidos, y visualizar estadísticas y rankings de los participantes.

Objetivo Principal

El objetivo del sistema es ofrecer una herramienta intuitiva y eficiente para organizar torneos, permitiendo un manejo centralizado de la información y automatizando procesos clave como la generación de partidos y el cálculo de estadísticas.

Funcionalidades Principales

El sistema incluye las siguientes funcionalidades:

- **Gestión de jugadores:** Registrar, modificar y listar jugadores participantes.
- **Creación y manejo de torneos:** Configurar un torneo con sus atributos principales, como nombre, localidad y superficie.
- **Generación de rondas y partidos:** Administrar el avance del torneo generando enfrentamientos en cada ronda.
- **Estadísticas y rankings:** Consultar estadísticas de los jugadores y visualizar el ranking actualizado según los resultados de los partidos.
- **Persistencia de datos:** Guardar y cargar la información del torneo y los jugadores en archivos JSON.

Audiencia Objetivo

El sistema está diseñado para ser utilizado por organizadores de torneos de tenis de nivel aficionado o profesional que busquen una solución digital para gestionar la información de manera eficiente y precisa.

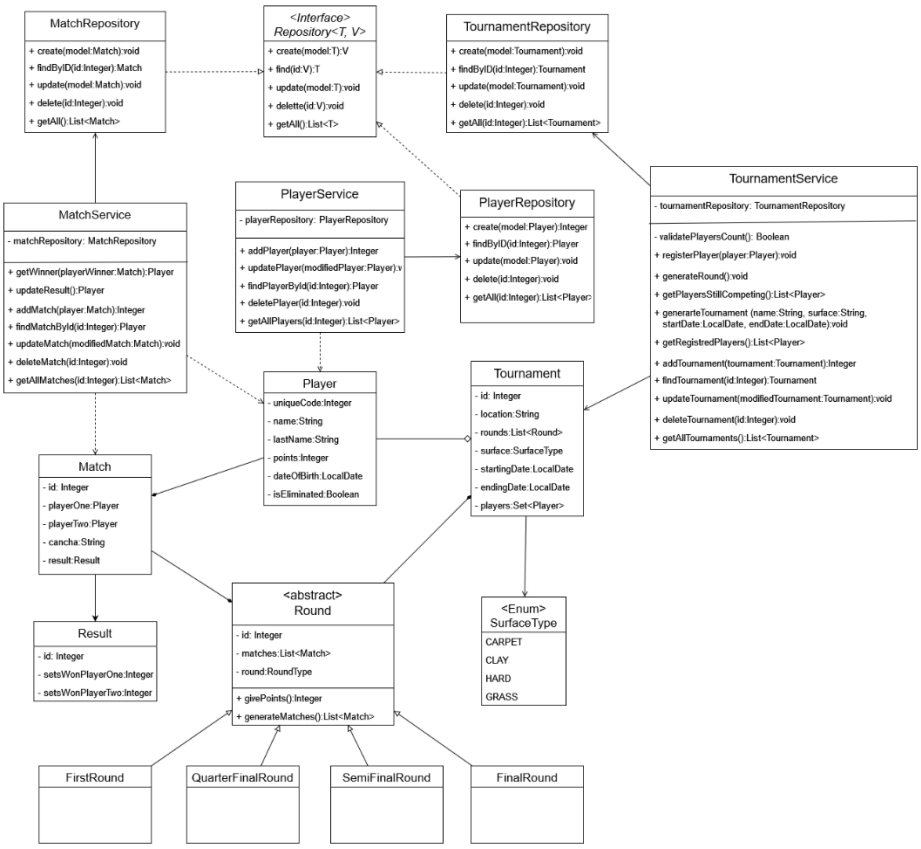
El sistema desarrollado cumple con su objetivo de proporcionar una herramienta práctica y eficiente para la gestión integral de torneos de tenis. Con funcionalidades clave como la administración de jugadores, rondas y partidos, así como la generación de estadísticas y rankings, ofrece una solución digital accesible y efectiva para organizadores, simplificando procesos y centralizando la información en un formato claro y reutilizable.

2. Decisiones de Diseño

2.1 Estructura del Sistema

```
tp-final-programming-II/
├── .idea/
├── data/
│   ├── backUp/
├── libs/
├── src/
│   ├── enums/
│   ├── exceptions/
│   ├── model/
│   ├── repository/
│   ├── service/
│   ├── test/
│   ├── utils/
│   ├── view/
│   ├── Main
├── UML/
```

2.2 Diagrama UML



2.3 Elección de Herramientas y Tecnologías

Durante el desarrollo del sistema, seleccionamos herramientas y tecnologías que se alinearan con los requerimientos del proyecto, garantizando una implementación eficiente y organizada. A continuación, se detalla cada elección:

I. Lenguaje de Programación: Java

- Elegimos **Java** como lenguaje principal por su robustez y orientación a objetos, características esenciales para el diseño e implementación del sistema.
- Su amplia biblioteca estándar y compatibilidad con herramientas adicionales permitieron manejar colecciones, excepciones personalizadas y persistencia de datos de manera efectiva.

II. Formato de Datos: JSON

- Decidimos utilizar **JSON** para la persistencia de datos debido a su formato ligero, fácil de leer y ampliamente utilizado.
- Este formato nos permitió guardar y recuperar información estructurada como jugadores, partidos, rondas y detalles del torneo, facilitando la continuidad del sistema entre ejecuciones.

III. Librería: Jackson

- Para trabajar con JSON, empleamos la librería **Jackson**, que se integró perfectamente con Java.
- Su capacidad para mapear objetos Java a JSON y viceversa simplificó el proceso de lectura y escritura de datos, asegurando la consistencia de la información almacenada.

IV. Herramienta de Gestión: Trello

- Utilizamos **Trello** como herramienta de gestión de tareas para coordinar el trabajo en equipo.
- Organizamos las actividades en un tablero con listas como *Por Hacer*, *En Proceso* y *Completado*. Esto permitió un flujo de trabajo claro y ordenado, asegurando que cada integrante conociera sus responsabilidades y el progreso general del proyecto.

La combinación de estas herramientas y tecnologías nos permitió desarrollar un sistema funcional, bien estructurado y fácil de mantener, cumpliendo con los requisitos establecidos y garantizando una experiencia de desarrollo fluida.

3. Clases y Relación entre Ellas

3.1 Descripción de Clases Principales

Clase Torneo:

- **Rol:** Representa a un torneo.
- **Propiedades:**
  - idTournament (Integer): Identificador único del torneo.
  - name (String): Nombre del torneo.
  - location (String) Lugar donde se lleva a cabo el torneo.
  - surface (Enum): Superficie donde se juega el torneo.
  - startingDate (LocalDate): Fecha que comienza el torneo.
  - endingDate (LocalDate): Fecha que finaliza el torneo.
  - players (Set<Player>): Set de jugadores que compiten en el torneo.
  - rounds (List<Round>): Lista de rondas del torneo.
  - Status (Enum): Estado en el que se encuentra el torneo.
- **Métodos:**
  - registerPlayer(): Registra un jugador al torneo.
  - unsubscribePlayer(): Elimina a un jugador del torneo.
  - startTournament(): Inicia el torneo si están la cantidad correcta de jugadores.
  - advanceCurrentRound(): Avanza de ronda si es que todos los partidos terminaron.
  - getTournamentWinner(): Obtiene el ganador del torneo a través de la ronda final.
  - updateTournamentStatus(): actualiza el estado del torneo según como avanza.

Clase Ronda:

- **Rol:** Representa una ronda del torneo.
- **Propiedades:**
  - matches (List<Match>): Lista de partido en la ronda.
  - givenPoints (Player): Puntos otorgados por la ronda.
- **Métodos:**
  - isCurrentRoundComplete(): Se fija si terminaron todos los partidos de la ronda.
  - getPlayersStillCompeting(): Retorna una lista de jugadores que siguen en el torneo.
  - nextRound(): Genera una nueva ronda y asigna los puntos correspondientes.
  - getCurrentRound(): Devuelve la ronda actual.
  - getFinalMatch(): Devuelve el partido de la final.

Clase Partido:

- **Rol:** Representa un partido del torneo.
- **Propiedades:**
  - idMatch (Integer): Identificador único del partido.
  - playerOne (Player): Jugador 1 del partido.
  - playerTwo (Player): Jugador 2 del partido.
  - result (Result): Obtiene el resultado del partido.
- **Métodos:**
  - getMatchesByPlayer(): Obtiene una lista de partidos dado un jugador.
  - getWinner(): Retorna el jugador ganador de un partido gracias al resultado.
  - assignResult(): Asigna su resultado al partido.
  - modifyResult(): Modifica un resultado de un partido.
  - findMatchById(): Busca un partido dado su ID.

Clase Jugador:

- **Rol:** Representa a un jugador participante del torneo.
- **Propiedades:**
  - idPlayer (Integer): Identificador único del jugador.
  - dni (String): Documento Nacional de Identidad.
  - name (String): Nombre del jugador.
  - lastName (String): Apellido del jugador.
  - nationality (String): Nacionalidad del jugador.
  - dateOfBirth (LocalDate): Fecha de nacimiento del jugador.
  - points (Integer): Puntuación acumulada por el jugador en el torneo, utilizada para rankings y estadísticas.
- **Métodos:**
- showStatsByPlayer(): Modifica las estadísticas del jugador en función de los resultados de los partidos.
- showPlayerRankings(): Muestra el ranking del jugador basado en su puntaje acumulado (points).

Clase Resultado:

- **Rol:** Representa el resultado de un partido.
- **Propiedades:**
  - setsScore (List<SetScore>): Lista de sets ganados por cada jugador.
- **Métodos:**
- addSetScore(): Agrega un Set a un partido.
- thereIsNoWinner(): Devuelve false hasta que un jugador gane 2 sets y finalice el partido.
- getSetsWonPlayerOne(): Devuelve la cantidad de sets ganados por el jugador 1.
- getSetsWonPlayerTwo(): Devuelve la cantidad de sets ganados por el jugador 2.

Clase SetScore:

- **Rol:** Representa el resultado específico de un partido.
- **Propiedades:**
  - playerOneScore (Integer): Numero de games ganados por el jugador 1 en cada set.
  - playerTwoScore (Integer): Numero de games ganados por el jugador 2 en cada set.
- **Métodos:**
- validateFullScore(): Valida que el resultado sea correcto en base a las reglas del tenis.

3.2 Uso de Interfaces y Clases Abstractas

En el diseño de nuestra aplicación, hemos optado por el uso de una **clase abstracta** en lugar de interfaces para modelar las diferentes rondas de un torneo. En particular, hemos creado una clase abstracta llamada Round que actúa como la base para todas las rondas del torneo, desde la primera hasta la última.

3.3 Uso de Colecciones

En nuestro sistema, utilizamos diferentes tipos de colecciones según la necesidad de cada caso. Para los **partidos**, usamos un **List** (como ArrayList), ya que necesitamos mantener el orden de inserción y permitir el acceso por índice. Para los **jugadores**, empleamos un **Set** para garantizar que no haya jugadores duplicados, ya que cada jugador tiene un DNI único. Esta elección optimiza tanto el rendimiento como la integridad de los datos, asegurando que el sistema maneje los partidos y jugadores de manera eficiente.

4. Manejo de Excepciones

Se implementó un conjunto de excepciones personalizadas para manejar errores específicos del sistema, como la duplicación de jugadores (DuplicatePlayerException), la ausencia de entidades (EntityNotFoundException), errores en el procesamiento de archivos (FileProcessingException), resultados inválidos (InvalidResultException) y otros relacionados con el estado de torneos y partidos.

- **DuplicatePlayerException:** Se lanza cuando se intenta agregar un jugador que ya existe en el sistema, es decir, hay un duplicado.
- **EntityNotFoundException:** Indica que no se encontró una entidad específica (jugador, torneo, etc.) al realizar una búsqueda u operación.
- **FileProcessingException:** Ocurre cuando hay un error al procesar un archivo, como al leer o escribir datos.
- **IncompleteMatchException:** Se lanza cuando un partido no tiene toda la información necesaria para ser considerado completo (por ejemplo, falta un resultado).
- **InvalidResultException:** Indica que un resultado ingresado no es válido según las reglas del sistema o del torneo.
- **InvalidTournamentStatusException:** Se produce cuando se intenta realizar una operación en un torneo que no se encuentra en el estado correcto (por ejemplo, intentar inscribir jugadores en un torneo ya iniciado).
- **MatchNotFoundException:** Similar a EntityNotFoundException, pero específicamente para partidos.
- **PlayerNotFoundException:** Indica que no se encontró un jugador al buscarlo.
- **TournamentFullException:** Se lanza cuando se intenta agregar más jugadores a un torneo que ya está lleno.
- **TournamentNotFoundException:** Indica que no se encontró un torneo al buscarlo.

5. JSON y Archivos

En nuestro sistema, utilizamos **JSON** para almacenar y exportar información relacionada con el torneo y los jugadores. En particular, se utiliza la biblioteca **Jackson** para guardar y cargar el torneo y sus rondas, en formato JSON. Esto facilita el almacenamiento y análisis posterior de los datos, permitiendo una fácil exportación e importación de la información de manera estructurada.

Implementación: Utilizamos **Jackson** para serializar los objetos de jugadores, partidos y rondas a archivos JSON y para deserializarlos cuando es necesario cargar los datos. Esto garantiza que los datos se puedan guardar de forma persistente y se puedan recuperar fácilmente para su uso posterior.

---

6. Implementación de Genericidad

La **genericidad** se utilizó en el paquete **repository** para manejar diferentes tipos de objetos de manera flexible. Usamos clases genéricas para las operaciones de almacenamiento y recuperación de datos de manera uniforme, sin importar el tipo de entidad (como Player, Match, etc.).

Ejemplo de su utilidad: Se creó una clase genérica Repository<T, K> que permite trabajar con cualquier tipo de objeto, proporcionando métodos comunes como create(), find(), update(), delete(), getAll() sin necesidad de repetir el código para cada tipo específico de entidad.

Fuentes de Información

- Para desarrollar el proyecto, se utilizaron diversos recursos tanto de **materiales de clase** como de **experiencia previa**. Entre los principales recursos utilizados están:
- **Documentación oficial de Java**: para resolver dudas sobre el uso de colecciones, manejo de excepciones y la implementación de características específicas del lenguaje.
- **Tutoriales de YouTube**: para obtener ejemplos prácticos de cómo utilizar bibliotecas como Jackson para la manipulación de JSON.
- **Apuntes de clase**: proporcionaron la base teórica y práctica necesaria para implementar correctamente la estructura del sistema y los patrones de diseño utilizados.

La combinación de estos recursos fue esencial para abordar los desafíos del proyecto y aplicar soluciones efectivas.