



# **SISTEMA DE GESTIÓN DEPORTIVA**

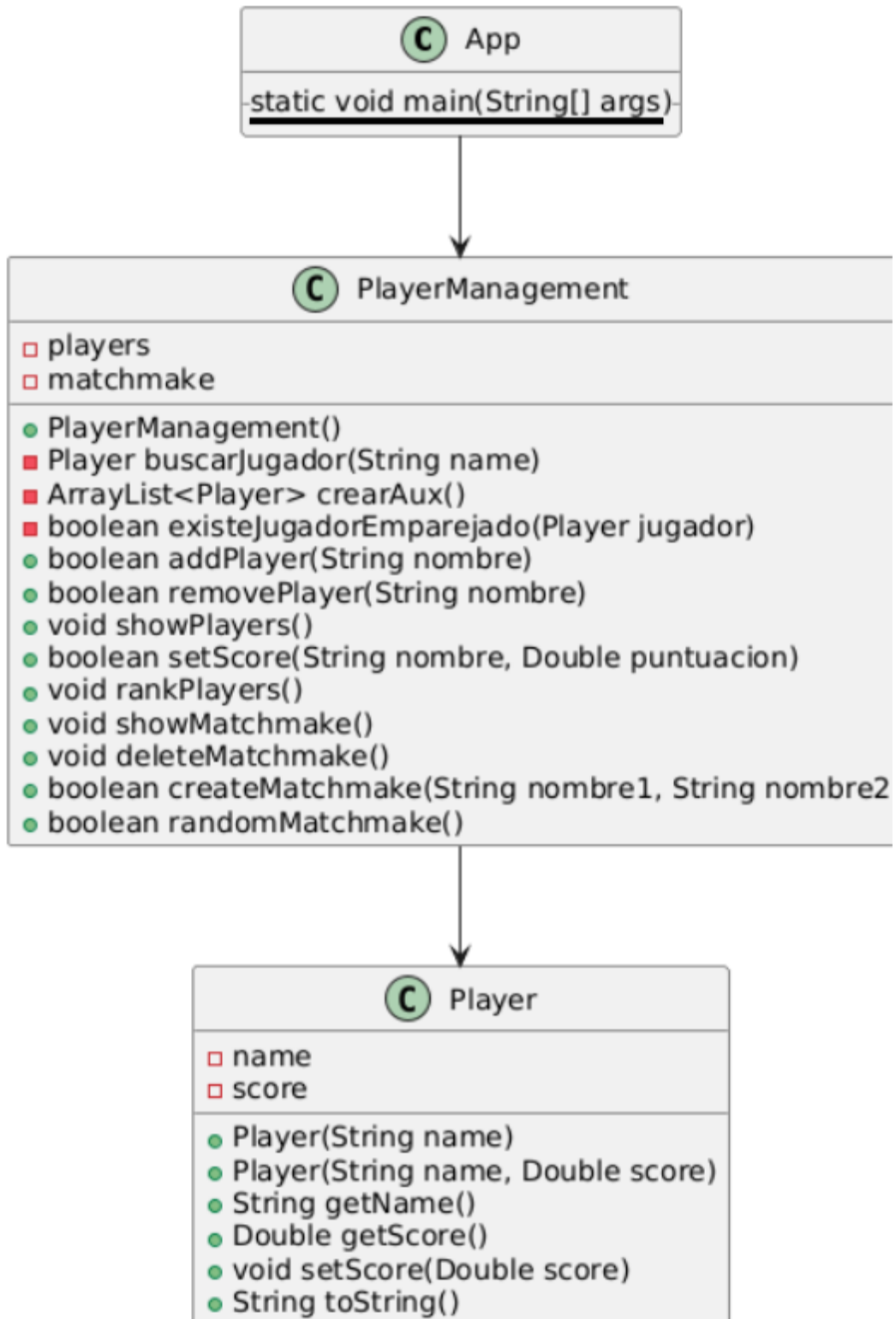
**(ENTREGAS 2 Y 3)**

Por:

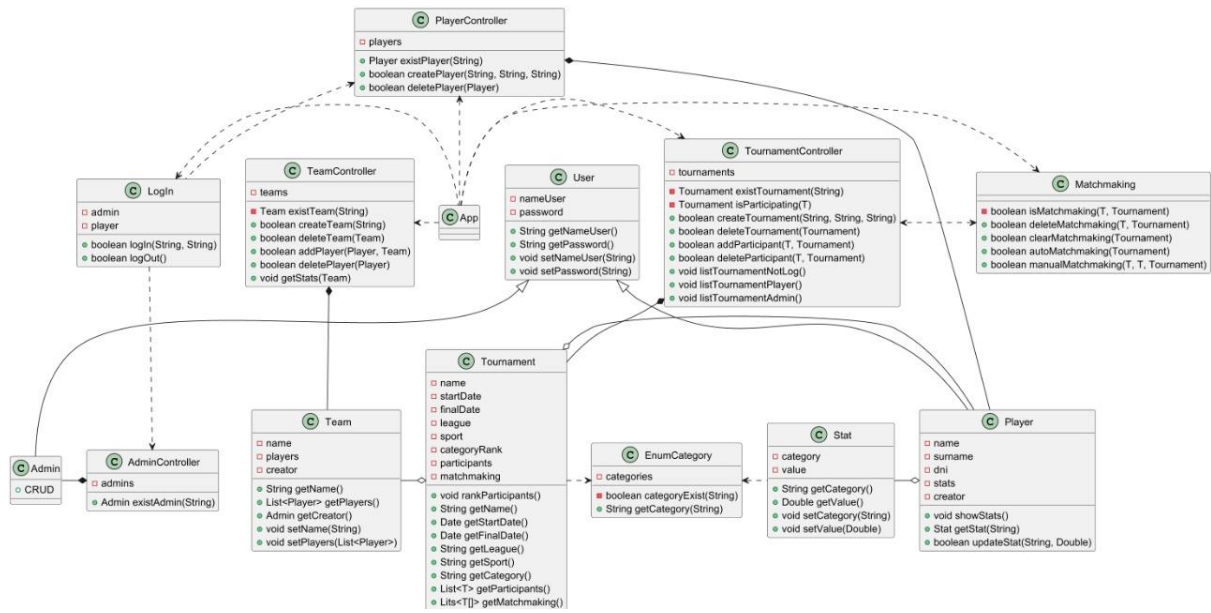
- Francisco Javier Grande Alonso (fj.grande@alumnos.upm.es)
- Adrián Largo Monteagudo (adrian.largo@alumnos.upm.es)
- Alberto Arpa Hervás (a.arpa@alumnos.upm.es)

IWSIM21 – POO

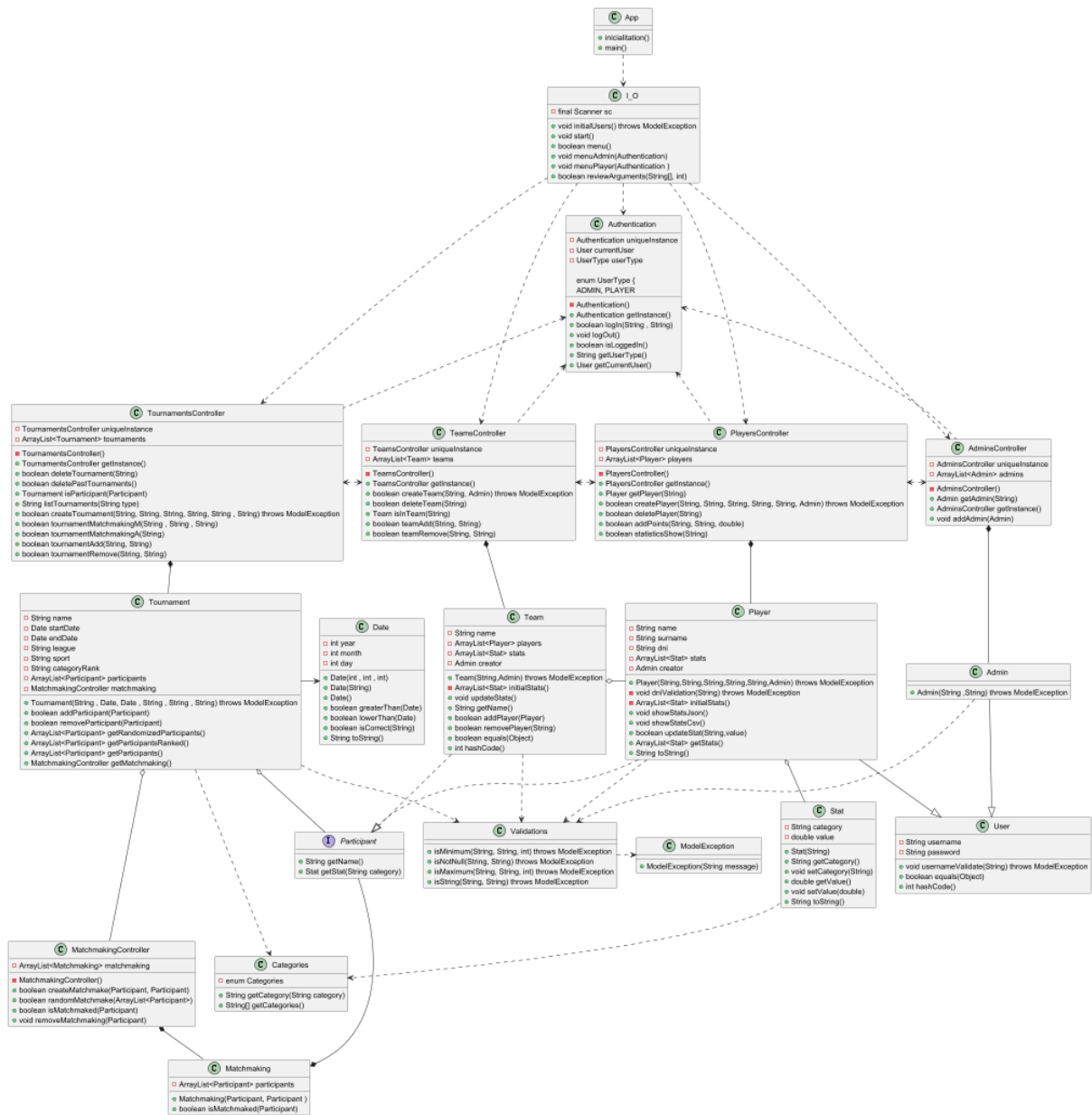
# PRÁCTICA 1:



# PRÁCTICA 2:



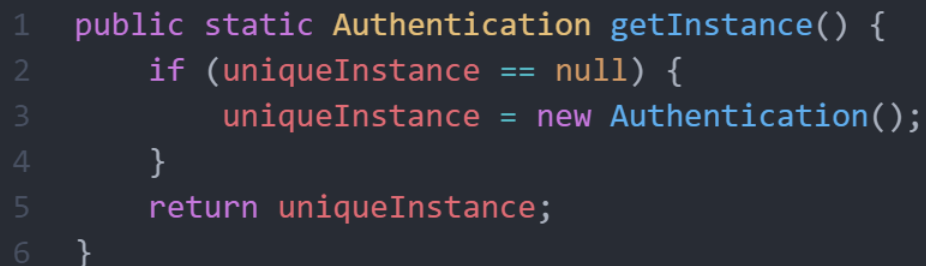
# PRÁCTICA 3:



## Ejemplo y justificación de patrones de diseño en nuestro código.

Nuestro código implementa por ejemplo un patrón de diseño de tipo Singleton. Este patrón es muy utilizado en el mundo del desarrollo permitiendo crear una sola y única instancia de una clase y proporcionar un punto de acceso único a esta, además de una correcta optimización del código.

En nuestro caso específico, hemos incluido este patrón en la clase Authentication con un método getInstance() que comprueba si ya existe una instancia de la clase antes y en caso contrario la crea.

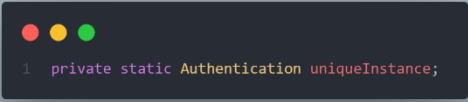


```
1 public static Authentication getInstance() {
2     if (uniqueInstance == null) {
3         uniqueInstance = new Authentication();
4     }
5     return uniqueInstance;
6 }
```

La clase Singleton también debe tener un constructor privado y una variable estática que será la que almacene nuestra instancia de la clase.



```
1 private Authentication() {
2     currentUser = null;
3     userType = null;
4 }
```



```
1 private static Authentication uniqueInstance;
```

# Decisiones de implementación en base al enunciado.

A medida que iba creciendo el sistema, iban surgiendo nuevos problemas de diseño. Desde demasiado acoplamiento hasta métodos repetidos. Las soluciones propuestas cuentan con que cada clase tenga una finalidad, siguiendo los principios SOLID y haciendo que el sistema, aparte de funcionar, cuente con un fácil mantenimiento.

Pese a que en el enunciado no muestre interés por un aporte de puntuación a los jugadores, así como la visualización de los emparejamientos generados, hemos decidido implementar ambos métodos.

```
1 public boolean createMatchmake(Participant participant1, Participant participant2) {
2     if (!isMatchmade(participant1) && !isMatchmade(participant2)) {
3         Matchmaking matchmaking1 = new Matchmaking(participant1, participant2);
4         matchmaking.add(matchmaking1);
5         getMatchmakings();
6         return true;
7     }
8     return false;
9 }
```

```
1 public static boolean addPoints(String username, String stat, double points) {
2     boolean result = false;
3     if (getPlayer(username) != null) {
4         if (TeamsController.isInTeam(username) != null) {
5             TeamsController.isInTeam(username).updateStats();
6         }
7         return getPlayer(username).updateStat(stat, points);
8     }
9     return result;
10 }
```

Los métodos de las clases "Controller" son estáticos para el fácil manejo y relación entre clases. Hemos decidido implementar clases como Validation, Categories o interfaces como Participant que nos ayuden a implementar el código de manera óptima.