Memoria Práctica Final Programación Orientada a Objetos

Younes Bouahmed Alavedra
younes.bouahmed
Marc Bernabeu Rodriguez
marc.bernabeu

ÍNDICE

1. Enunciado de la práctica

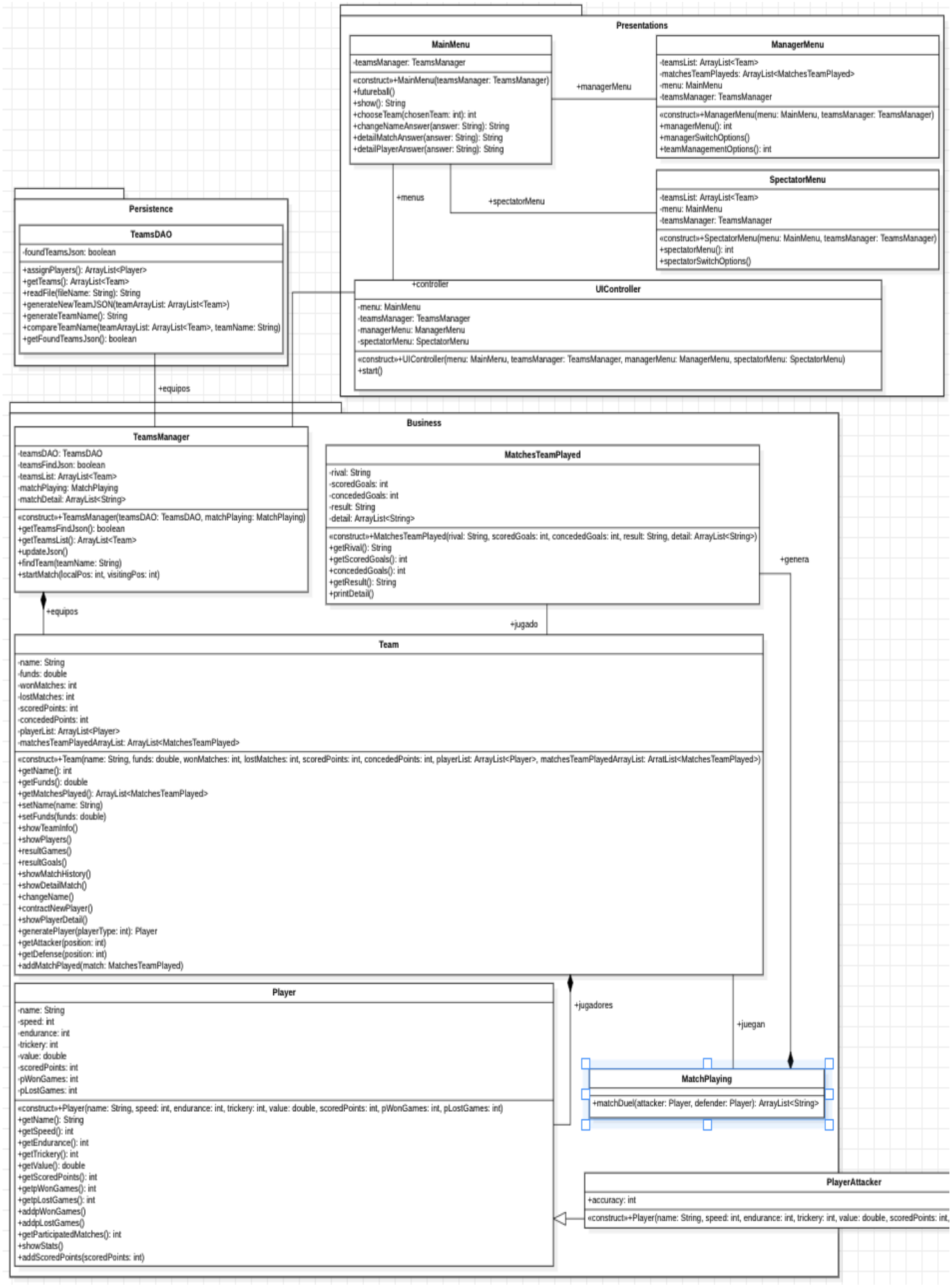
En esta práctica se ha realizado un proyecto en lenguaje de programación Java orientada a objetos.

El objetivo era crear un programa para gestionar partidos entre equipos de futbol. Estos equipos están formados por cuatro jugadores con diferentes capacidades y estadísticas que se nos especifican, dos de ellos son defensas, los otros dos son atacantes.

En los partidos se enfrentan dos equipos, en el cual primero ataca uno y después ataca el otro, y el ganador se determina viendo quien consigue más puntos. Para conseguir estos puntos, los atacantes del equipo que le toca atacar, se enfrentan a los defensores del otro equipo. Los atacantes de forma aleatoria eligen enfrentarse por velocidad, astucia o resistencia con el defensor, y el duelo lo gana el jugador con mejores estadísticas. Si el duelo resulta en que el atacante supera al defensor, este chuta a portería, y según la probabilidad de chute que tenga, podrá marcar o no, esta probabilidad de marcar irá variando a lo largo del partido.

2. Diagrama de clases UML

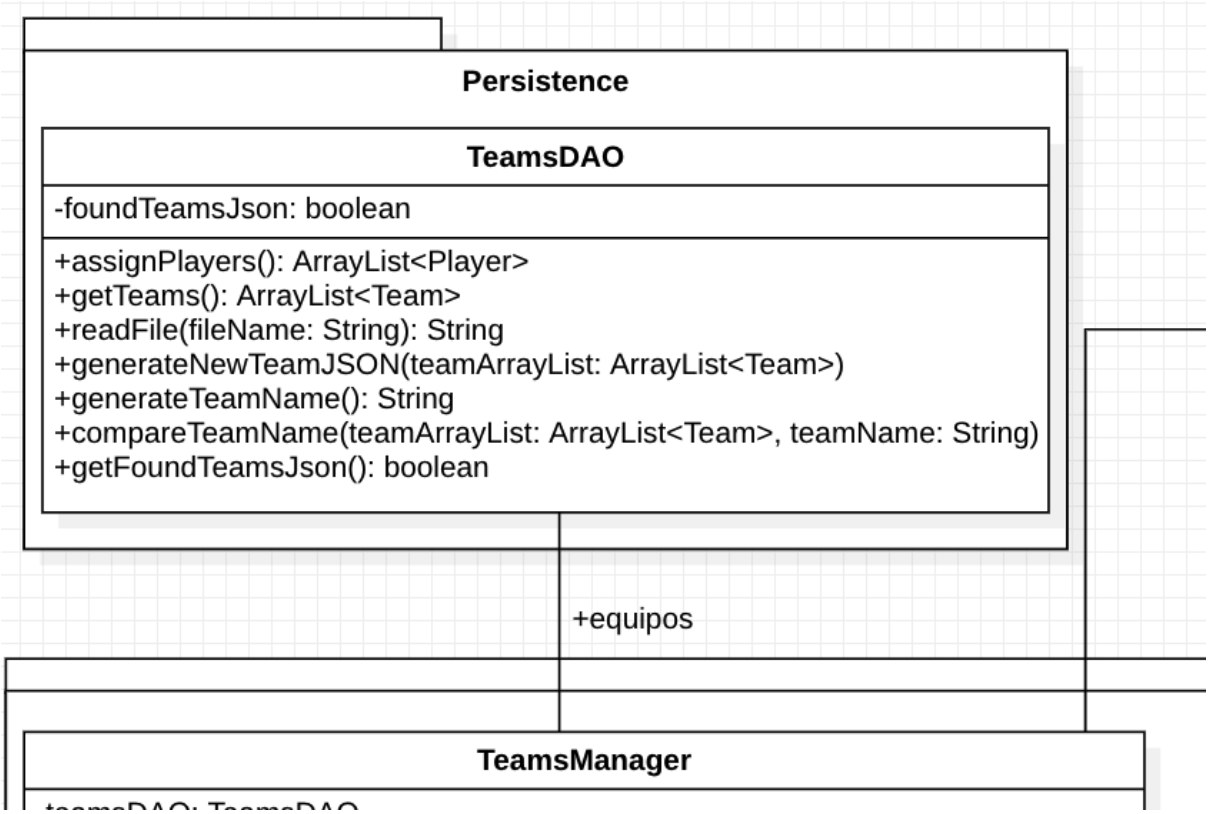
Para realizar el diagrama de clases en UML, usamos el programa StarUML, antes de ponernos con el código realizamos un primer diagrama, que se ha ido modificando a medida que el proyecto ha ido avanzando. En la página siguiente se puede ver el resultado del diagrama final.


Presentations
MainMenu
-teamsManager: TeamsManager
«construct»+MainMenu(teamsManager: TeamsManager)
+futureball()
+show(): String
+chooseTeam(chosenTeam: int): int
+changeNameAnswer(answer: String): String
+detailMatchAnswer(answer: String): String
+detailPlayerAnswer(answer: String): String
+managerMenu
ManagerMenu
-teamsList: ArrayList<Team>
-matchesTeamPlayeds: ArrayList<MatchesTeamPlayed>
-menu: MainMenu
-teamsManager: TeamsManager
«construct»+ManagerMenu(menu: MainMenu, teamsManager: TeamsManager)
+managerMenu(): int
+managerSwitchOptions()
+teamManagementOptions(): int
+menus
+spectatorMenu
SpectatorMenu
-teamsList: ArrayList<Team>
-menu: MainMenu
-teamsManager: TeamsManager
«construct»+SpectatorMenu(menu: MainMenu, teamsManager: TeamsManager)
+spectatorMenu(): int
+spectatorSwitchOptions()
Persistence
TeamsDAO
-foundTeamsJson: boolean
+assignPlayers(): ArrayList<Player>
+getTeams(): ArrayList<Team>
+readFile(fileName: String): String
+generateNewTeamJSON(teamArrayList: ArrayList<Team>)
+generateTeamName(): String
+compareTeamName(teamArrayList: ArrayList<Team>, teamName: String)
+getFoundTeamsJson(): boolean
+controller
UIController
-menu: MainMenu
-teamsManager: TeamsManager
-managerMenu: ManagerMenu
-spectatorMenu: SpectatorMenu
«construct»+UIController(menu: MainMenu, teamsManager: TeamsManager, managerMenu: ManagerMenu, spectatorMenu: SpectatorMenu)
+start()
+equipos
Business
TeamsManager
-teamsDAO: TeamsDAO
-teamsFindJson: boolean
-teamsList: ArrayList<Team>
-matchPlaying: MatchPlaying
-matchDetail: ArrayList<String>
«construct»+TeamsManager(teamsDAO: TeamsDAO, matchPlaying: MatchPlaying)
+getTeamsFindJson(): boolean
+getTeamsList(): ArrayList<Team>
+updateJson()
+findTeam(teamName: String)
+startMatch(localPos: int, visitingPos: int)
MatchesTeamPlayed
-rival: String
-scoredGoals: int
-concededGoals: int
-result: String
-detail: ArrayList<String>
«construct»+MatchesTeamPlayed(rival: String, scoredGoals: int, concededGoals: int, result: String, detail: ArrayList<String>)
+getRival(): String
+getScoredGoals(): int
+concededGoals(): int
+getResult(): String
+printDetail()
+genera
+equipos
+jugado
Team
-name: String
-funds: double
-wonMatches: int
-lostMatches: int
-scoredPoints: int
-concededPoints: int
-playerList: ArrayList<Player>
-matchesTeamPlayedArrayList: ArrayList<MatchesTeamPlayed>
«construct»+Team(name: String, funds: double, wonMatches: int, lostMatches: int, scoredPoints: int, concededPoints: int, playerList: ArrayList<Player>, matchesTeamPlayedArrayList: ArratList<MatchesTeamPlayed>)
+getName(): int
+getFunds(): double
+getMatchesPlayed(): ArrayList<MatchesTeamPlayed>
+setName(name: String)
+setFunds(funds: double)
+showTeamInfo()
+showPlayers()
+resultGames()
+resultGoals()
+showMatchHistory()
+showDetailMatch()
+changeName()
+contractNewPlayer()
+showPlayerDetail()
+generatePlayer(playerType: int): Player
+getAttacker(position: int)
+getDefense(position: int)
+addMatchPlayed(match: MatchesTeamPlayed)
Player
-name: String
-speed: int
-endurance: int
-trickery: int
-value: double
-scoredPoints: int
-pWonGames: int
-pLostGames: int
«construct»+Player(name: String, speed: int, endurance: int, trickery: int, value: double, scoredPoints: int, pWonGames: int, pLostGames: int)
+getName(): String
+getSpeed(): int
+getEndurance(): int
+getTrickery(): int
+getValue(): double
+getScoredPoints(): int
+getpWonGames(): int
+getpLostGames(): int
+addpWonGames()
+addpLostGames()
+getParticipatedMatches(): int
+showStats()
+addScoredPoints(scoredPoints: int)
+jugadores
+juegan
MatchPlaying
+matchDuel(attacker: Player, defender: Player): ArrayList<String>
PlayerAttacker
+accuracy: int
«construct»+Player(name: String, speed: int, endurance: int, trickery: int, value: double, scoredPoints: int,

3. Explicación del diagrama UML

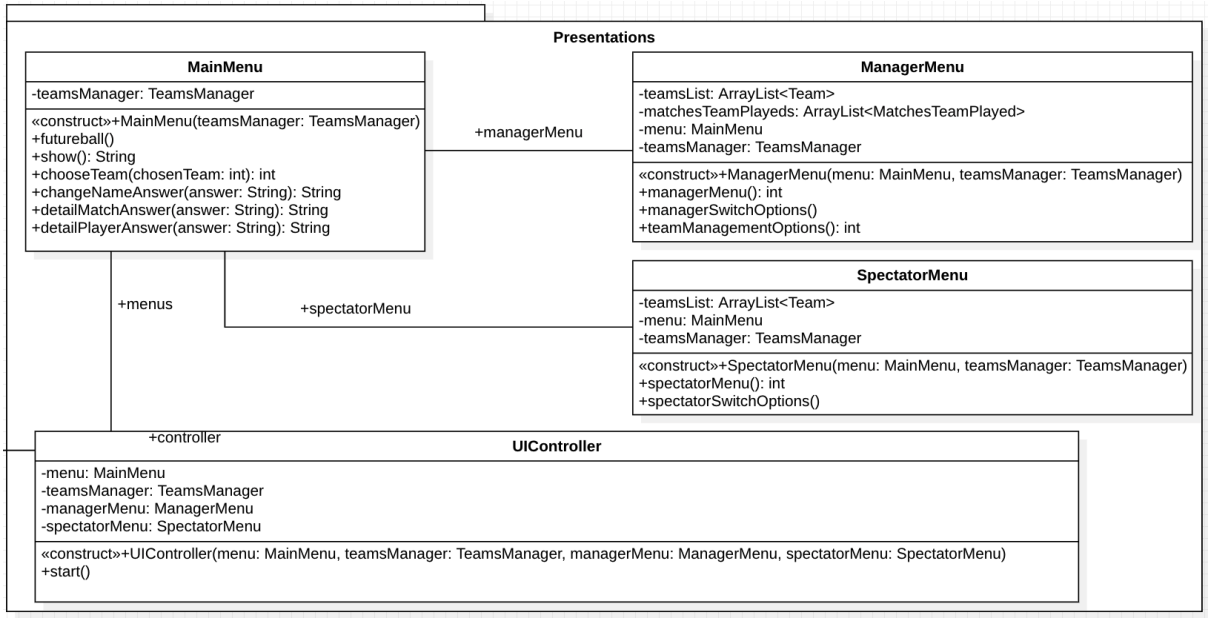
Dado el tamaño del diagrama, para poder explicarlo se hará por partes. De esta forma será más comprensible.
Al realizar este diagrama se nos pidió seguir el patrón de *Layered Architecture* explicado en clase. Con los cual en nuestro diagrama tenemos tres partes que diferencia nuestro código.

En la primera parte tenemos nuestro paquete de *Persistence*, que contiene la clase TeamsDAO, creamos esta clase en su momento para tener una fácil gestión de los datos de los diferentes equipos. Había la posibilidad de que estos datos existieran o, por lo contrario, se tenían que crear, y para eso nos sirve esta clase, que lee un archivo JSON, y si encuentra datos de equipos, accede a ellos, de lo contrario los genera. Al hacer esto tenemos la ventaja de que TeamDAO solo se encargara de comprobar datos y actualizarlos, algo que resulta muy eficiente tener en una sola clase.

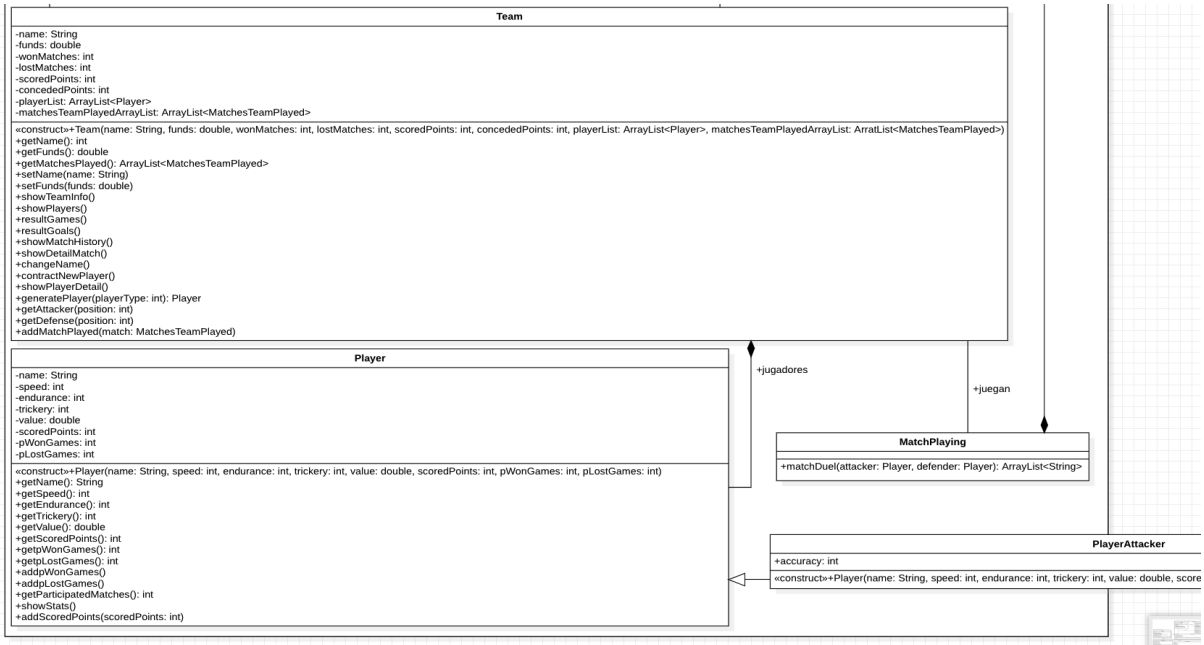


Seguidamente, tenemos el paquete de *Presentation*. Esta parte del código es la que se encargara de mostrar los mensajes correspondientes al usuario una vez que inicie el programa. Al hacer esto conseguimos que nuestro código fuera más eficiente, y eso también facilito el poder gestionar la parte de inputs y output al usuario de forma más eficiente. En este paquete tenemos un total de cuatro clases. Tenemos la *MainMenu* que la hicimos así para el menú principal, y por otra banda creamos *ManagerMenu* y *SpectatorMenu*, esto fue el resultado de ver el programa pedía la posibilidad de ser un espectador de partidos o ser el entrenador que gestiona equipos. Así, pues SpectatorMenu solo ofrece la posibilidad de ver información de equipos o de realizar partidos. Por otra banda, *ManagerMenu* ser encarga de ofrecer al usuario otras opciones como contratar

jugadores o cambiar el nombre de los equipos. Después *UIController* lo creamos para que fuera capaz de gestionar estos diferentes menús, cada vez que tocaba mostrar alguno, de esta forma dividíamos tareas en el código.



Finalmente, tenemos el paquete *Business*, este los creamos para manipular los diferentes datos que llegaban del paquete *Persistence*. Business contiene la clase *Teams* y *Players*, que se relacionan entre sí, por eso las pusimos todas juntas, ya que se necesitan para poder gestionar los datos. También tenemos la clase para jugar y generar partidos, estas no dependen tanto de las otras, pero las creamos para tener una fácil gestión de los partidos y no mezclar datos, porque así nos resultaba más eficiente separar los diferentes objetos, cada uno con sus atributos.



4. Método de pruebas utilizado

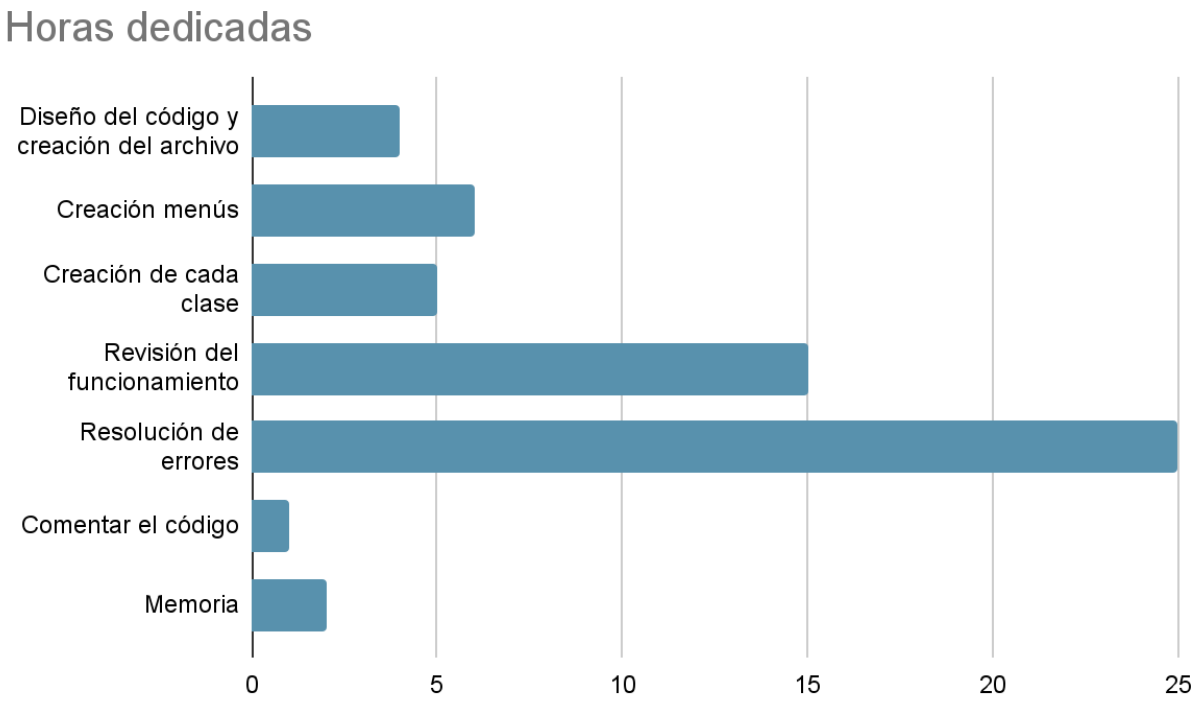
Principalmente, a la hora de realizar las distintas pruebas durante el proceso de la práctica, el método qué hemos usado con más frecuencia ha sido básicamente prueba y error. Dentro del mismo código, cuando nos topábamos con cualquier error, comprobamos los valores obtenidos antes y después de dónde se producía el error para poder ver el cambio del error y sí mostraba correctamente la información qué esperábamos recibir.

En más de una ocasión, llegamos a crear funciones específicas para poder obtener distintas informaciones sobre lo que se estaba ejecutando.

A la hora de gestionar los datos del *TeamsDAO*, nos encontramos con diferentes dificultades que nos impedían seguir. No conseguimos leer los datos bien cuando los teníamos que generar nosotros porque no existían, así que sobre todo usamos bastante la funcionalidad del *tryCatch,* que nos ayudó bastante a la hora de encontrarnos con atributos que eran nulos.

También nos costó el hecho de realizar partidos, ya que en la forma que decidimos gestionarlos, teníamos que preparar los datos de los equipos que iban a jugar, modificar esos datos y después volverlos a pasar al archivo JSON que contenía esa información.

5. Dedicación de horas



6. Conclusiones

A nivel personal es una práctica que nos ha gustado por el enunciado que presentaba, ya que ha sido un tema divertido del cual hacer una práctica. Aunque había partes que nos han resultado complicadas de entender porqué no había una especificación suficientemente clara.

También el hecho de trabajar en pareja nos ha resultado algo complicado porque teníamos diferentes formas de plantear el diseño de lo que íbamos a programar, pero por otra banda ha sido mejor trabajar en pareja porque hemos podido aportar diferentes soluciones y aprendíamos otras formas de diseñar la práctica.

A nivel técnico hemos tenido algunos problemas con la librería de datos que se nos obligaba a implementar a la hora de realizar la práctica. Ya qué hemos tenido que comprender las distintas funcionalidades qué aportaba.
A causa de esta librería, a la hora de obtener la información qué generaba, se añadía como una especie de residuo dentro del nombre del jugador, el cual no entendíamos la causa, haciéndonos perder el tiempo.

En general nos ha gustado realizar esta práctica, aunque con el tiempo justo de nuestra parte. Ya que de haber tenido la posibilidad de empezar antes, habríamos tenido como resultado un código más completo y con la posibilidad de realizar las partes opcionales del enunciado.