Ingeniería informática

UT5 TFU: Backend y API de los juegos olímpicos

Asignatura: Análisis y Diseño de Aplicaciones

Docentes: Glauco Javier Yannone, Rodrigo Lujambio

20 de junio de 2024

Índice:

Parte 1: Casos de uso

Juez:

- Iniciar sesión
- Ver disciplinas/modalidades/categorías
- Seleccionar categoría
- Ver participantes de dicha categoría y modalidad
- Calificar participante según puntajes de la categoría
- Ver Calificaciones/calificados
- Ver equipos de dicha categoría y modalidad
- Calificar miembros de equipos según puntaje de la categoría

Por temas de tiempo nos centramos en la implementación del Juez, dejando para el futuro los roles de otros usuarios como el Administrador, Voluntario y Atleta.

Parte 2: Desarrollo de la API Rest

Se desarrolló la api con Maven haciendo uso de Spring Boot en parte debido a que ya implementa o facilita la implementación de algunos patrones de diseño, como es el Repository que se encuentra implementado en la API.

Otro tema a consideración es que algunos de los compañeros ya habían usado Maven para la creación de una API, lo cual permitió tener una base de conocimientos a aplicar, y por último Maven cumple con estar programado en Java uno de los requisitos del TFU.

Principios SOLID

SRP (Single Responsibility Principle): DatabaseConnection

Se separó la lógica de la conexión a la base de datos de los repositorios, porque si a futuro se quieren añadir nuevos métodos ya poseemos una razón para cambiar en los Repository pero si además cambia la conexión a la base de datos se poseería una segunda razón para cambiar que violaría el principio SRP.

ISP (Interface Segregation Principle): Controllers

Los controllers dependen de interfaces específicas de los repositorios que dependen, en vez de depender de una interfaz general IRepository que implementaran todos los repositorios. Esto nos permite reducir el acoplamiento y la implementación de métodos innecesarios.

DIP (Dependency Inversion Principle): Controllers

Los controllers dependen de interfaces, es decir abstracciones, en vez de en las clases concretas o de los detalles. Esto nos permite reducir el acoplamiento al no depender de implementaciones concretas.

Patrones

Singleton: DatabaseConnection

Esta clase se encargará de proporcionar una única instancia de conexión a la base de datos, esta implementación nos permite asegurarnos de que sea la única instancia a su vez que dar un punto de acceso global a la misma, se utiliza porque precisamos de un único objeto de conexión a la base de datos compartido en todo el programa.

Repository: Multiples Repository

Estas clases se encargan de encapsular la lógica de conexión, almacenamiento y obtención de datos de la base de datos emulando una colección de datos, por ejemplo en múltiples de los métodos de estos repositorios son getSomething() que devuelve una lista de Something[] que se encuentran en la base de datos, el cliente que usa esta clase no conoce ni tiene porqué conocer los detalles necesarios para la obtención de los datos que precisa, para el implica simplemente obtener una colección de datos.

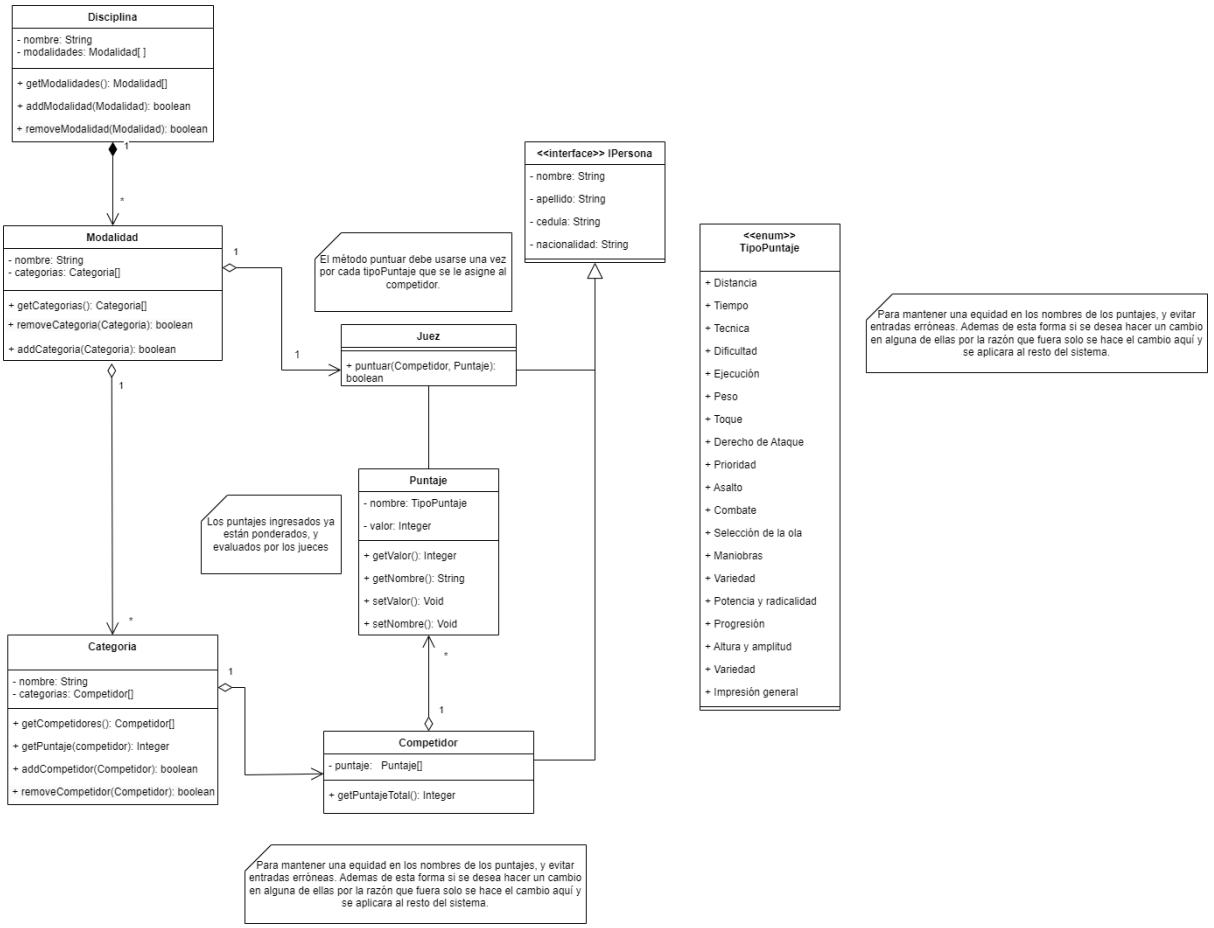
Parte 3: Modificaciones y ajustes

Diagrama de clases

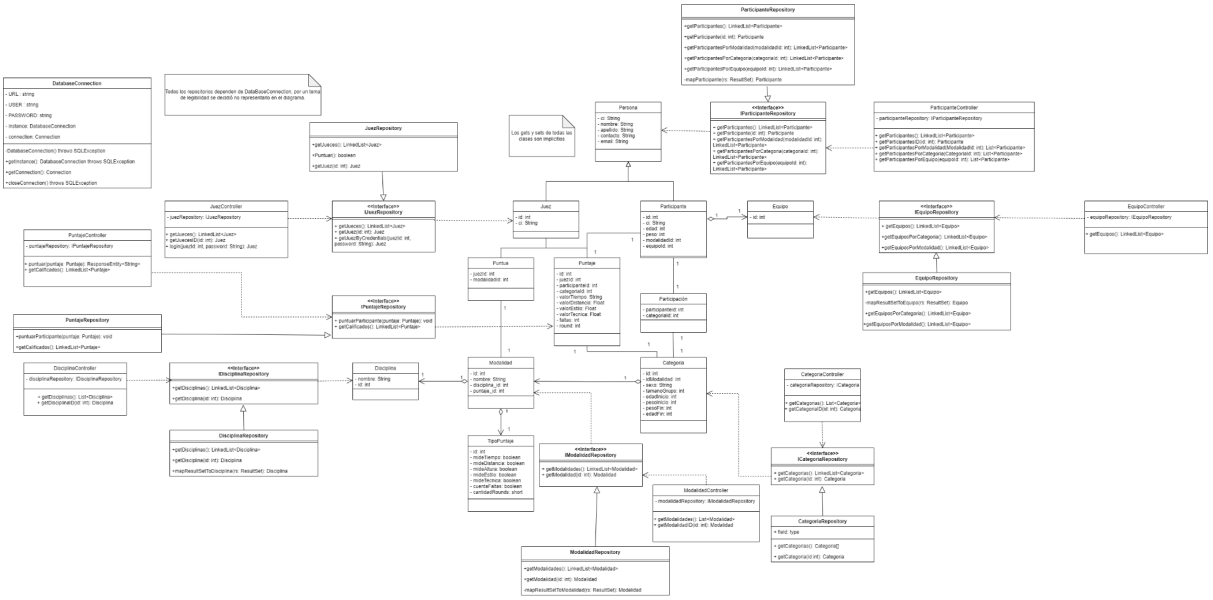
Los cambios del diagrama de clases pueden parecer radicales, y en algunos casos lo es por los nuevos patrones aplicados como los Repository, y los controllers. A la hora de hacer la API nos encontramos con múltiples clases que no pensamos inicialmente, como la de Equipos, y la de distintas conexiones a la base de datos, a su vez que otras necesarias para el apartado de persistencia y realización de pedidos.

También hubieron cambios en el enumerator TipoPuntaje, este paso a ser una clase para así poder conocer los TipoPuntaje de una modalidad, además que TipoPuntaje ahora posee múltiples variables para conocer que tipo de inputs debe poner el juez a la hora de calificar un participante de dicha modalidad. Junto a este cambio se vio la necesidad de hacer que Puntaje sea una clase que posea todos los puntajes del TipoPuntaje, por lo cual el atleta tiene un puntaje en la modalidad en vez de múltiples puntajes divididos.

Antes:


Disciplina
- nombre: String
- modalidades: Modalidad[ ]
+ getModalidades(): Modalidad[]
+ addModalidad(Modalidad): boolean
+ removeModalidad(Modalidad): boolean
1
*
Modalidad
- nombre: String
- categorias: Categoria[]
+ getCategorias(): Categoria[]
+ removeCategoria(Categoria): boolean
+ addCategoria(Categoria): boolean
1
1
El método puntuar debe usarse una vez por cada tipoPuntaje que se le asigne al competidor.
<<interface>> IPersona
- nombre: String
- apellido: String
- cedula: String
- nacionalidad: String
Juez
+ puntuar(Competidor, Puntaje): boolean
<<enum>>
TipoPuntaje
+ Distancia
+ Tiempo
+ Tecnica
+ Dificultad
+ Ejecución
+ Peso
+ Toque
+ Derecho de Ataque
+ Prioridad
+ Asalto
+ Combate
+ Selección de la ola
+ Maniobras
+ Variedad
+ Potencia y radicalidad
+ Progresión
+ Altura y amplitud
+ Variedad
+ Impresión general
Para mantener una equidad en los nombres de los puntajes, y evitar entradas erróneas. Ademas de esta forma si se desea hacer un cambio en alguna de ellas por la razón que fuera solo se hace el cambio aquí y se aplicara al resto del sistema.
Puntaje
- nombre: TipoPuntaje
- valor: Integer
+ getValor(): Integer
+ getNombre(): String
+ setValor(): Void
+ setNombre(): Void
Los puntajes ingresados ya están ponderados, y evaluados por los jueces
Categoria
- nombre: String
- categorias: Competidor[]
+ getCompetidores(): Competidor[]
+ getPuntaje(competidor): Integer
+ addCompetidor(Competidor): boolean
+ removeCompetidor(Competidor): boolean
Competidor
- puntaje: Puntaje[]
+ getPuntajeTotal(): Integer
Para mantener una equidad en los nombres de los puntajes, y evitar entradas erróneas. Ademas de esta forma si se desea hacer un cambio en alguna de ellas por la razón que fuera solo se hace el cambio aquí y se aplicara al resto del sistema.

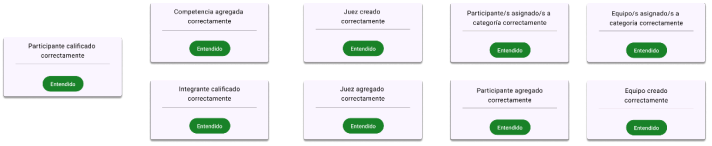
Después:

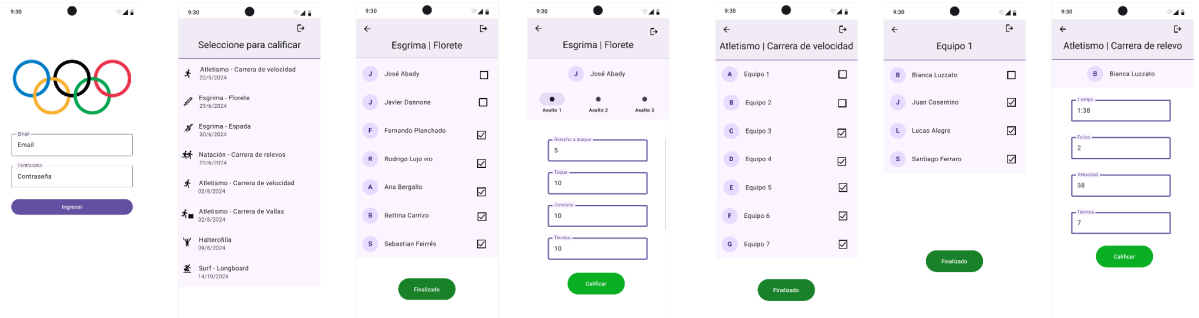


Figma

Por temas de tiempo se decidió reducir el alcance de la aplicación al uso de un único usuario llamado juez, se desarrolló la API teniendo en cuenta que esta sea escalable para permitir a futuro crear nuevos roles y endpoints para los casos de uso que no se implementaron de administradores y voluntarios.

Dejando de lado el tema del alcance, se debe de destacar los pocos problemas o cambios que trajo al figma la creación de la API, esto se debe a que el prototipado original ya captaba la gran mayoría de endpoints que precisaria y tenía una buena lógica detrás para permitir su funcionamiento.



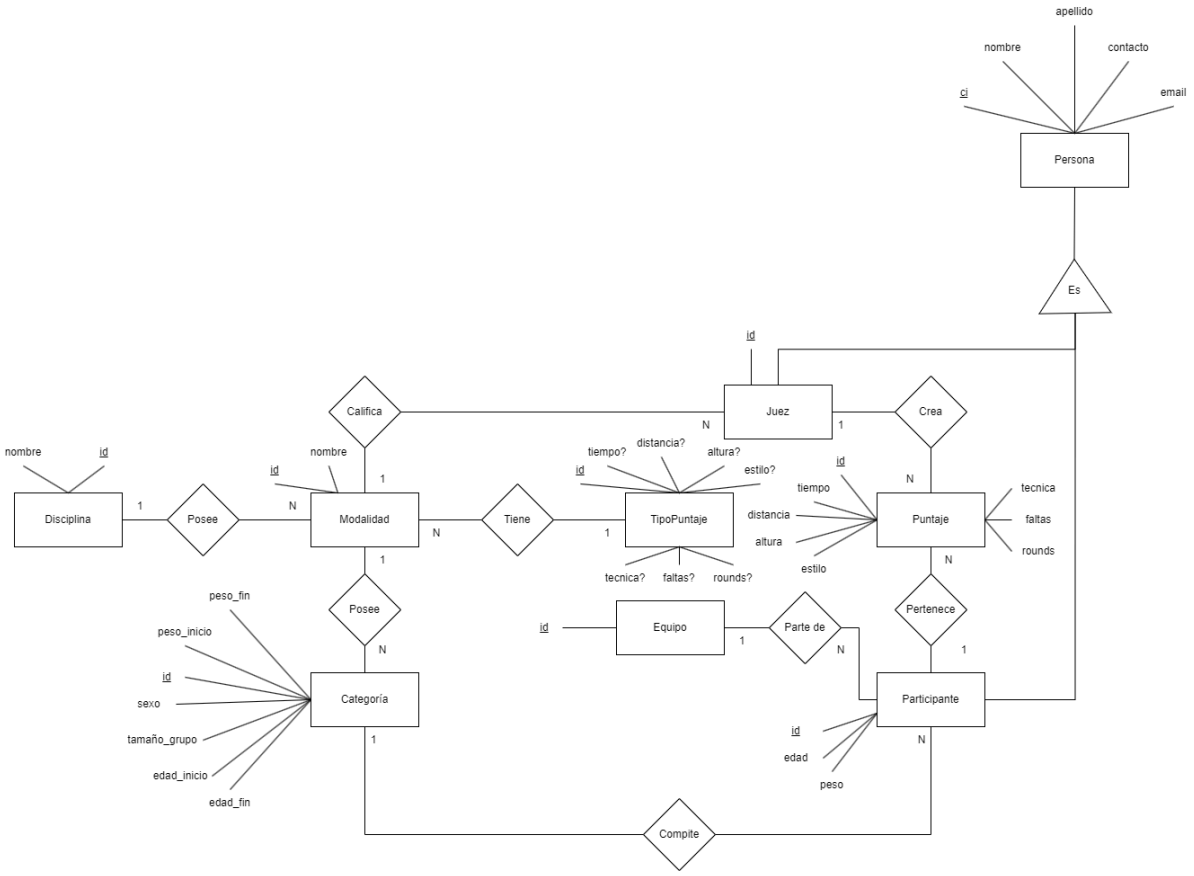


(Link al figma)

MER (Modelo Entidad-Relación)

Como un extra fuera de la consigna original se decidió crear un MER para tener un consenso del modelado de datos para la base de datos, pudiendo asi identificar las entidades que participaron, cómo se relacionan dichas entidades, sus cardinalidades y sus atributos. Esto nos permitió bajar a tierra muchos aspectos que inicialmente no se tuvieron en cuenta en el diagrama de clases.

La creación de este MER facilitó la creación del diseño lógico, es decir del pasaje a tablas, ya que a partir de un MER base se es conocido las técnicas que se deben emplear para traducirlo a tablas, como el uso de claves primarias, foráneas, tablas intermedias para aquellas relaciones de cardinalidad N-N, etcétera.



(Link al MER)

(Lucas Alegre, Franco De Stefano, Santiago Ferraro, Franco Robotti)