UT5_TFU- Ignacio Villarreal, Bruno Albín, Santiago Aurrecochea, Joaquin Gasco, José Varela y Gonzalo Paz.

Patrones de diseño

Desarrollar un backend en C# .NET Core que satisfaga las necesidades del frontend.

Durante este proyecto, el equipo de front será contratado en la modalidad de outsourcing, por lo que la documentación de las APIs y el prototipo son una herramienta de comunicación fundamental.

Identificar y ajustar los casos de uso a partir de diseños existentes.

Asegurar la coherencia entre el modelo de clases, prototipo y la implementación.

1 - Identificación de Casos de Uso:

Revisen el diseño de clases y modelos creados en la UT3 y el prototipo en la UT4. Identifiquen los casos de uso más relevantes y agreguen cualquier caso de uso adicional si lo consideran necesario.

Realizado una revisión de las clases y modelos creados previamente, para identificar y ajustar los casos de uso esenciales, nos dimos cuenta que no hacía falta modificar mucho los casos de uso, y decidimos dejar las funcionalidad de crear campeonatos, asignar disciplinas, inscribir participantes y registrar puntuaciones que creemos que son las más importantes.

- La aplicación debe ser capaz de registrar las disciplinas con datos como el nombre y descripción. Este caso de uso es fundamental porque las disciplinas son la base de las competiciones. También debe permitir crear las propias competiciones.
- Es esencial que la aplicación pueda registrar a los participantes de cada disciplina. Tener un registro completo y preciso de los participantes es importante para gestionar adecuadamente las competiciones y calcular las puntuaciones de manera correcta. Esto incluye información como el nombre del participante, país, categoría y género.
- Además, la aplicación debe permitir registrar las puntuaciones de cada competidor en función de las reglas específicas de cada disciplina.
- La aplicación debe gestionar las distintas categorías de competidores (por edad, género, discapacidad, etc.) y aplicar las reglas correspondientes a cada una de ellas. Gestionar las categorías adecuadamente es necesario para aplicar las reglas y restricciones específicas de cada una.

2 - Desarrollo de la API Rest:

a. Diseñar y desarrollar una API REST que cumpla con los casos de uso identificados.

b. Asegúrense de seguir los principios SOLID y utilizar patrones de diseño donde sean apropiados.

Dado que ninguno de los integrantes del equipo sabía hacer una api en .net con c#, ni los controladores, ni repositorios, ni nada y además tuvimos poco tiempo hicimos lo que pudimos.

Tratamos de seguir los principios SOLID lo más que pudimos:

SRP: El principio de responsabilidad única creemos que lo aplicamos a medias, tratamos de que cada clase, controlador o repositorio solo tenga las mínimas responsabilidades que debería tener asignadas a su funcionalidad propia.

OCP: El principio abierto/cerrado también lo aplicamos en la parte de crear nuevas disciplinas, en estos casos quizás con una modificación mínima de las clases originales y a partir de interfaces se podría agregar nuevas disciplinas con su determinado cálculo de puntuación.

LSP: El principio de sustitución de Liskov creemos que lo cumplimos debido a que si bien usamos interfaces y clases abstractas estas podrían ser sustituidas perfectamente con sus hijos, un ejemplo de esto sería la clase base Disciplina la cual podría ser sustituida por cualquiera de sus hijas perfectamente, como Natación o Halterofilia.

ISP: El principio de segregacion de interfaces tambien lo complimos debido a que las interfaces que utilizamos y las clases que la implementan usan completamente sus métodos declarados y en ningún momento se usa `throw new exception` para especificar que no tiene cuerpo el método implementado. Esto se puede ver en por ejemplo las dos interfaces para calificaciones de disciplinas `ICalificacionPeso` y `ICalificacionTiempo`, donde cada una tiene su manera de calificar diferente y no están todas en una misma interfaz.

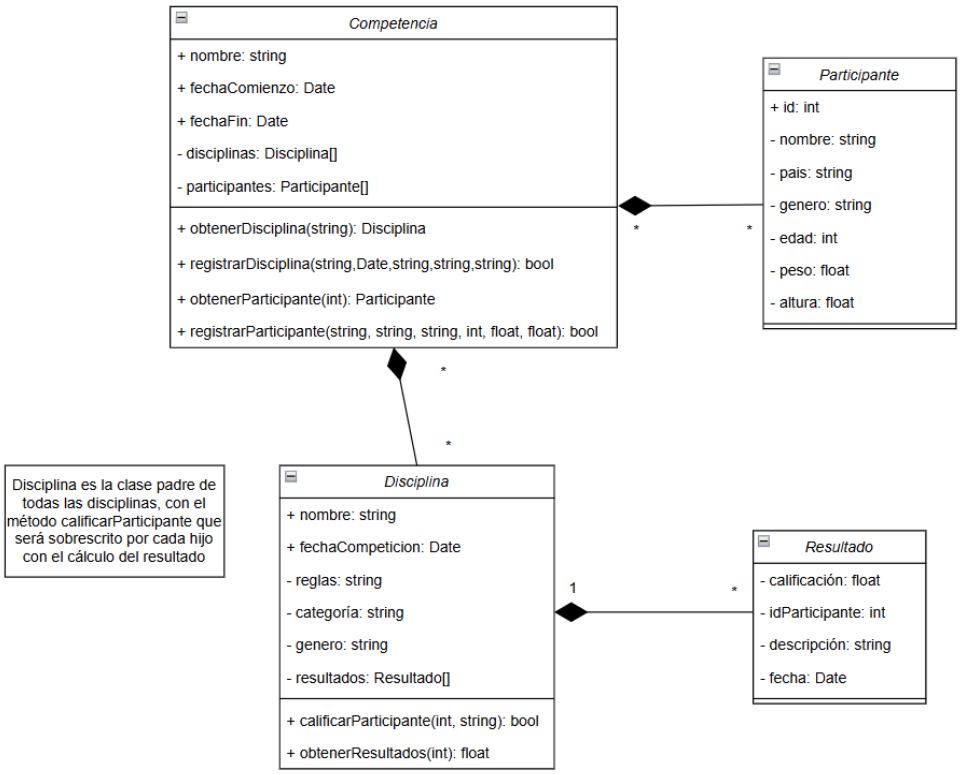
DIP: El principio de inversión de dependencias no estamos seguros de si lo aplicamos.

Otros que nos gustaría implementar:

El patrón Factory Method es muy buena opción en el contexto del problema, tenemos una clase abstracta `Disciplina` que hereda a varios tipos de disciplinas, como `Natación` y `Halterofilia`, cada una con su propia manera de calificar. Si después queremos crear una nueva disciplina, como `Gimnasia`, con el patrón Factory Method podríamos añadir fácilmente nuevas disciplinas sin cambiar el código existente. Además se podría hacer lo mismo para cada sub disciplina.

3 - Modificaciones y Ajustes:

a. Durante el proceso de desarrollo, si identifican que se necesitan cambios en el modelo de clases, hagan las modificaciones correspondientes.



Decidimos dejarlo de la misma manera y después en código agregarle las interfaces, las disciplinas específicas, controladores, repositorios, etc. implementando cada una de las funcionalidades.

b. De igual forma, si encuentran que el prototipo requiere ajustes basados en los cambios realizados o hallazgos durante el desarrollo, cámbienlo.

En el caso de figma como se pide solo una api para el backend no hizo falta cambiarlo.

4 - Documentación y Entrega:

a. Suban el código al repositorio GitHub provisto.

La api se encuentra en:
[ignacioVillarreal2003/WebApi-.NET (github.com)]

b. Crea una presentación en la que expliquen su trabajo, incluyendo los hallazgos, cambios en el modelo de clases, cambios en el prototipo, patrones utilizados y su justificación.

c. Documenta los cambios realizados al modelo, incluye un diagrama de clases actualizado. Es importante que se vean los cambios.

d. Proporcionen un enlace a un prototipo público en Figma con los cambios.

https://www.figma.com/design/8eO8W4CDEua50HAWNPtYIF/Untitled?node-id=0-1&t=8nmCbCWdsn0dvuOc-1

e. Documenten la API (puede ser Swagger).

f. Provean pruebas automatizadas de la API (puede ser con Postman Collections)

En el repositorio de github.