

Universidad ORT Uruguay
Facultad de Ingeniería
Ingeniería en Sistemas

Diseño de aplicaciones 2
Obligatorio 2

Itai Miller - 201244
Rafael Alonso - 201523
Grupo M6A

Docente: Eduardo Cuñarro

Índice

1. Descripción general del trabajo	3
2. Descripción de diseño.....	3
2.1 Diagrama de paquetes	3
2.2 Diagrama de clases	4
2.2.1 Capa de la Lógica y el Dominio	4
2.2.2 Capa del Repositorio	8
2.2.3 Capa de la Web API	10
2.3 Diagrama de Componentes	11
2.4 Diagrama de Entrega.....	11
3. Justificación de diseño de las funcionalidades principales.....	12
4. Impacto de cambio de las nuevas funcionalidades.....	17
5. Front-End: Aplicación Angular (SPA)	18
6. Informe de métricas	18
Análisis de la gráfica obtenida	19
6. Endpoints Web API	20
7. Modelo de Tablas	21
Datos de prueba	21
8. Justificación y evidencia de Clean Code.....	21
9. Análisis de las pruebas.....	23
10. Evidencia TDD.....	25
11. Instalación	25
12. Conclusión	25
Anexos	26
User Controller	29
Sport Controller	30
Competitor Controller	31
Matches Controller	32

1. Descripción general del trabajo

El trabajo consiste en el desarrollo de una web api que permite interactuar con un sistema que permita administrar y publicar información sobre los partidos para diferentes deportes.

La aplicación cuenta con dos perfiles o roles: administradores y seguidores.

Los usuarios seguidores pueden conectarse al sistema, empezar a seguir a sus equipos favoritos y poder ver la lista de comentarios de todos los encuentros de los que eligió seguir. También podrán ver el calendario con todos los eventos que se agrupan por deporte. Tendrá la posibilidad de seleccionar un encuentro y añadir comentarios al mismo. También puede aplicar filtro y ordenamiento por nombre al listado y poder visualizar la lista de todos los equipos con sus respectivos datos. Los usuarios tienen la posibilidad de seleccionar uno o más equipos de la lista para seguirlos y que se muestren sus comentarios en la página principal.

En cuanto a los usuarios administradores, tendrán la posibilidad de realizar todas las mismas funcionalidades que los usuarios seguidores, pero también cuentan con la posibilidad de realizar el mantenimiento de todos los usuarios del sistema. Realizar el mantenimiento significa poder dar de alta, baja y modificación de lo siguiente: Usuario, Deporte, Equipo y Encuentro.

Desde nuestro punto de vista, es un programa que no tiene fallas ni bugs conocidos. Que no hayamos encontrado fallas no significa que no las haya por lo que no se puede asegurar en un 100%. Esto se debe a la importancia que le fuimos dando a las pruebas unitarias y de integración durante el proceso de desarrollo. A medida que íbamos avanzando, se nos fueron presentando algunos errores, pero pudimos detectarlos y corregirlos a tiempo.

2. Descripción de diseño

Para poder describir nuestra solución del diseño nos basaremos en diferentes diagramas UML que ayudan a explicar la misma, pero de una forma gráfica.

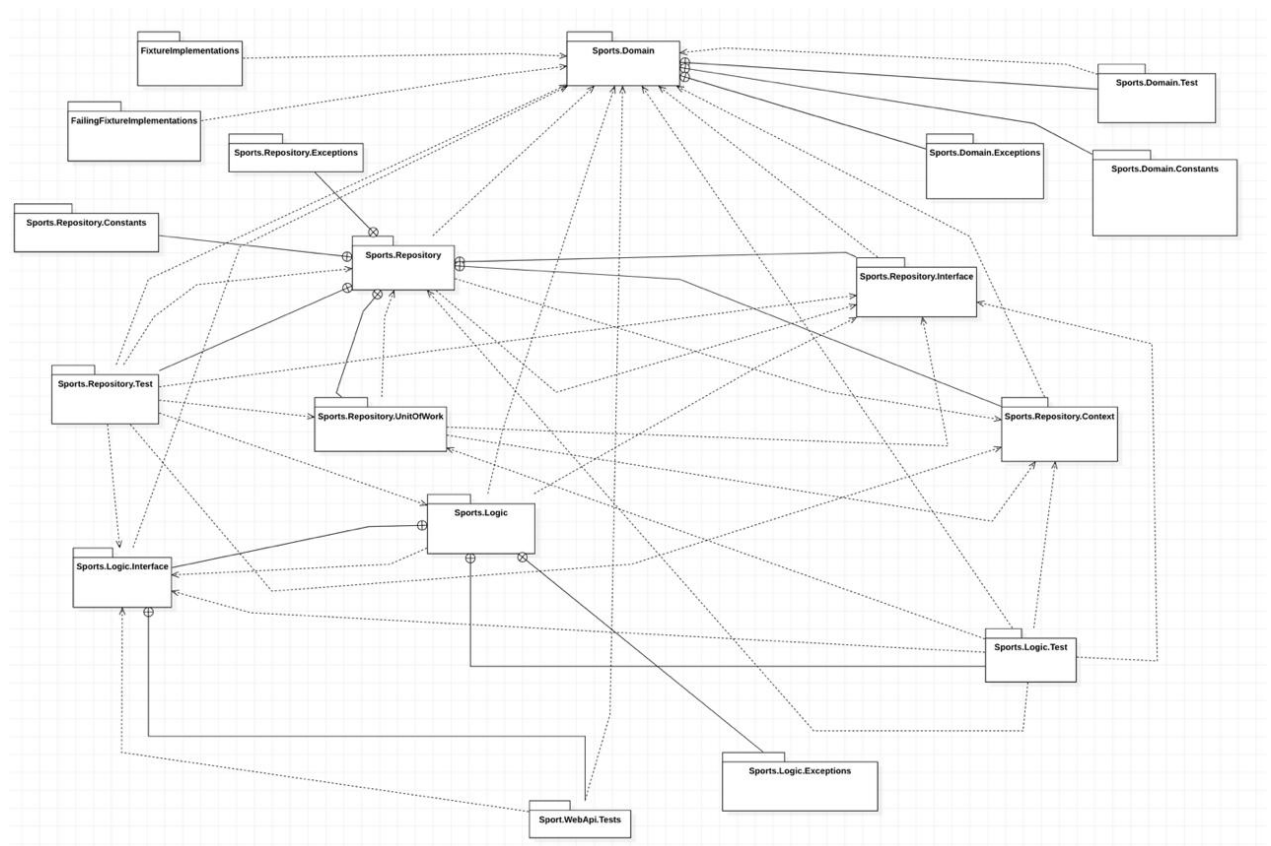
2.1 Diagrama de paquetes

En el siguiente diagrama se puede observar como el sistema está dividido en agrupaciones lógicas y las dependencias entre sí. En el mismo, se puede apreciar que la solución cuenta con 18 paquetes. La idea de separar en diferentes paquetes es porque cada uno cumple con una función específica para la solución.

Dividimos la solución en distintas capas para separar la responsabilidad del sistema. Las mismas son:

- Capa de repositorio.
- Capa de lógica y dominio.
- Capa de WebApi.

Más adelante explicaremos un poco de cada una de las capas.



2.2 Diagrama de clases

En esta parte se incluyen todos los diagramas de clases de los respectivos paquetes que fueron mencionados en la parte anterior. Cabe resaltar que otros diagramas (con otras representaciones) podrán ser vistos más adelante con el fin de adentrarse en explicaciones y descripciones acerca de comportamientos específicos del sistema.

2.2.1 Capa de la Lógica y el Dominio

Diagrama de clases del Dominio

El siguiente diagrama muestra la estructura interna del paquete Sports.Domain. El mismo, es el que contiene a todas las entidades que toman algún rol en el sistema. Las siguientes son:

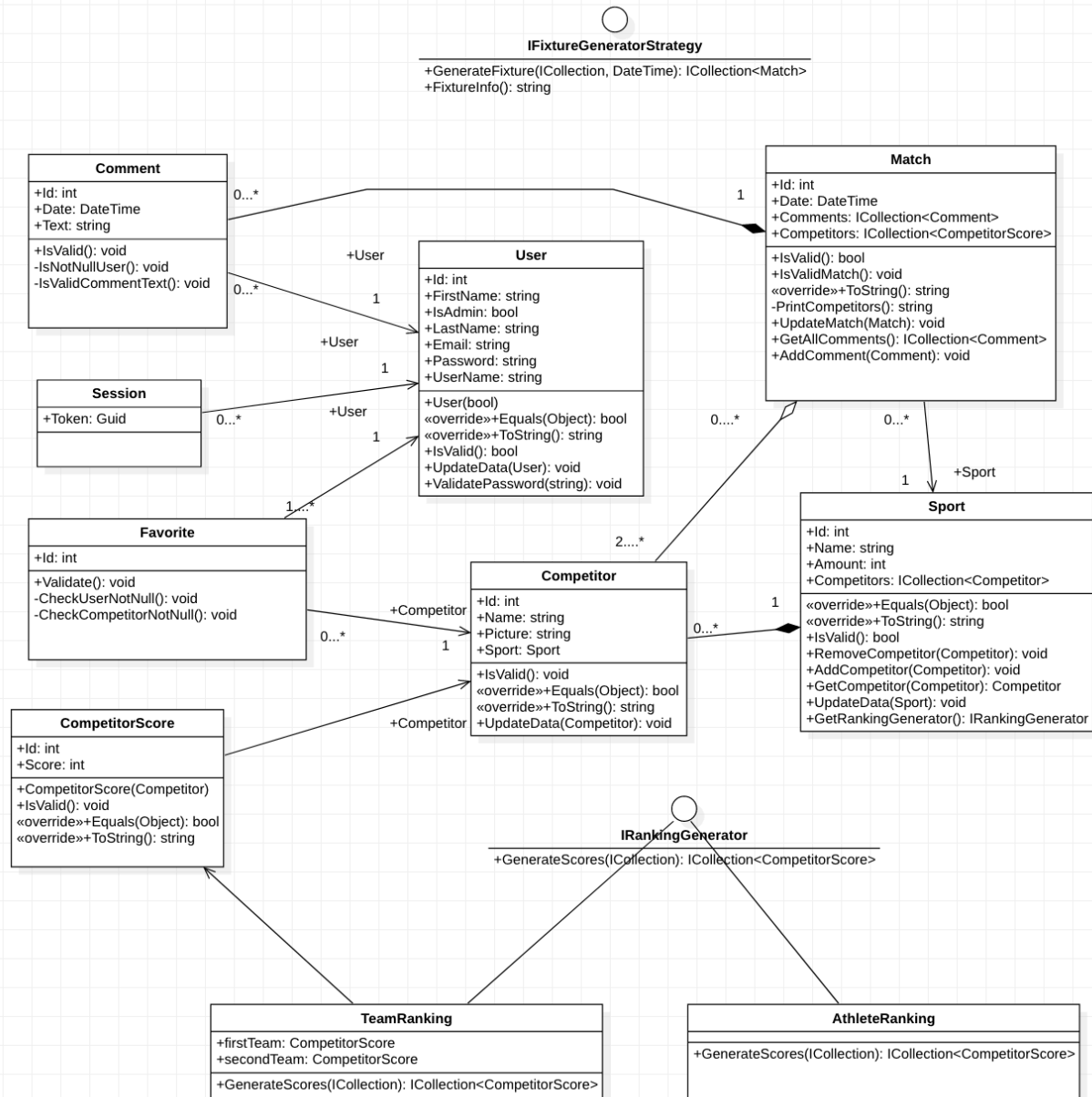
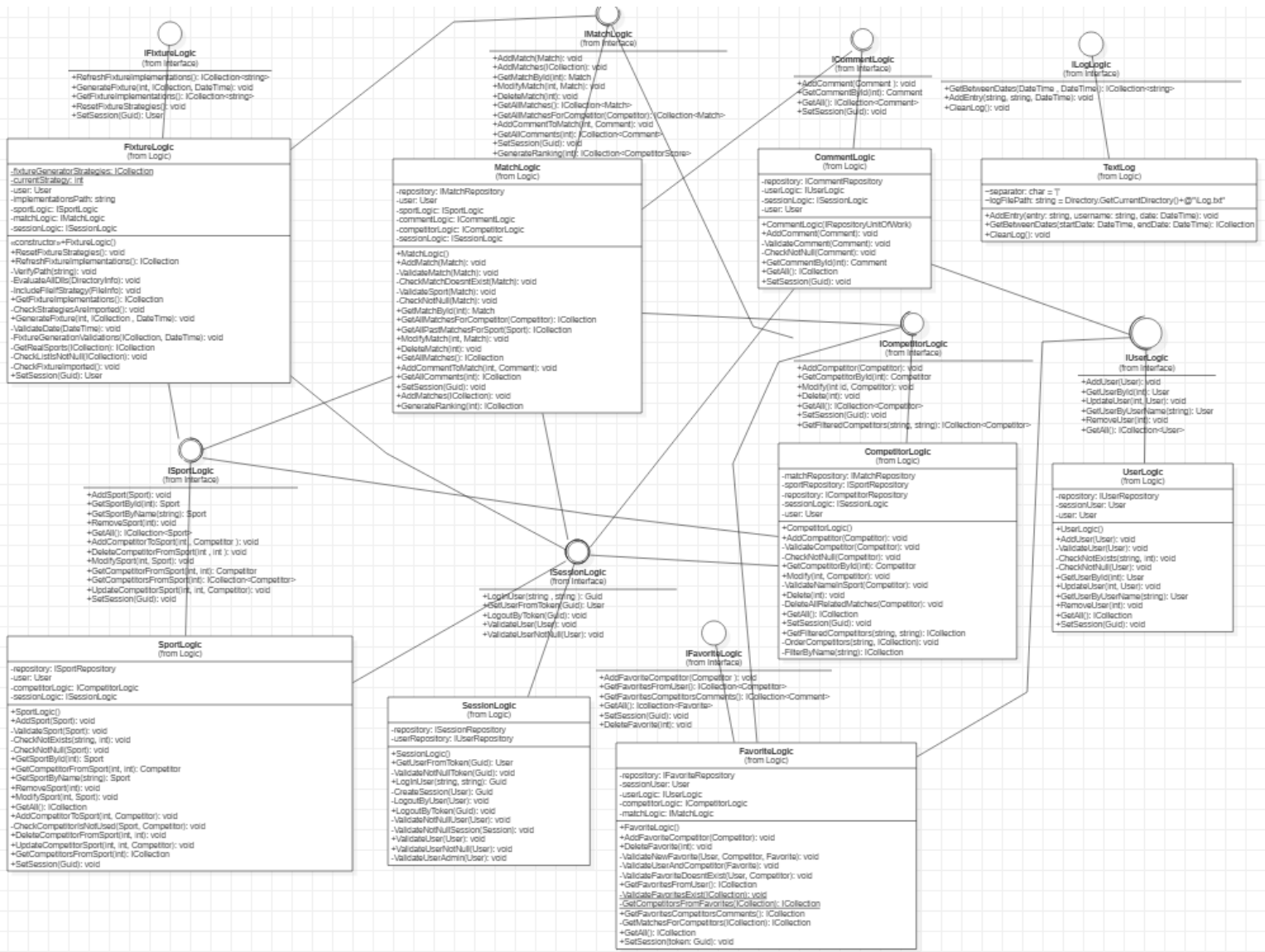


Diagrama de clases de la lógica

El siguiente diagrama muestra la estructura interna del paquete Sports.Logic. El mismo tiene la función de gestionar todos los servicios de las entidades del sistema. La capa lógica es la que separa la Web API del repositorio. Los controladores son los que van a interactuar con la lógica y esta a su vez va a interactuar con el repositorio.

Los controladores van a utilizar la capa de la lógica por medio de las interfaces y no por medio de las implementaciones directas.

Las interfaces se encuentran en el paquete Sports.Logic.Interfaces pero decidimos ponerlas en este diagrama junto con las clases de la lógica ya que cada una de las clases están directamente relacionadas con los contratos que proveen las interfaces. Por ejemplo, MatchLogic implementa IMatchLogic; UserLogic implementa IUserLogic y así sucesivamente con cada clase que se observa en el diagrama.



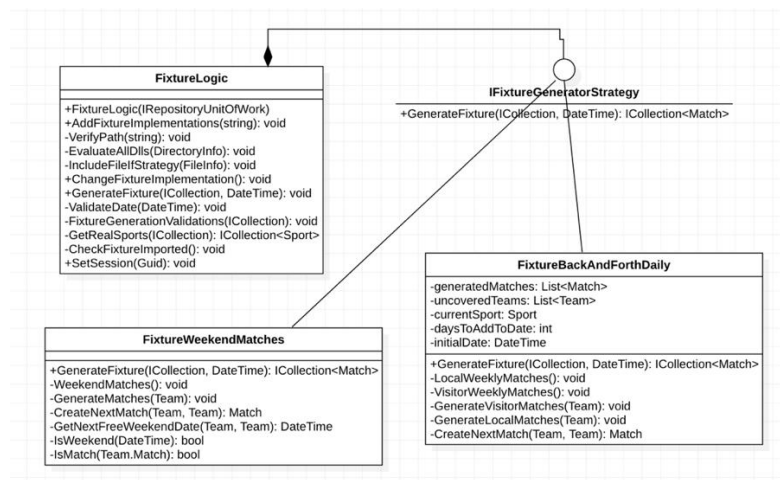
Dentro de esta capa del dominio y de la lógica, creamos dos paquetes para las excepciones. Un paquete llamado Sports.Domain.Exceptions que contiene las excepciones propias del dominio y otro paquete Sports.Logic.Exceptions que contiene las excepciones propias de la lógica. Todas las excepciones heredan de la clase Exception.

Nota: ambos diagramas se pueden ver en la sección de Anexos.

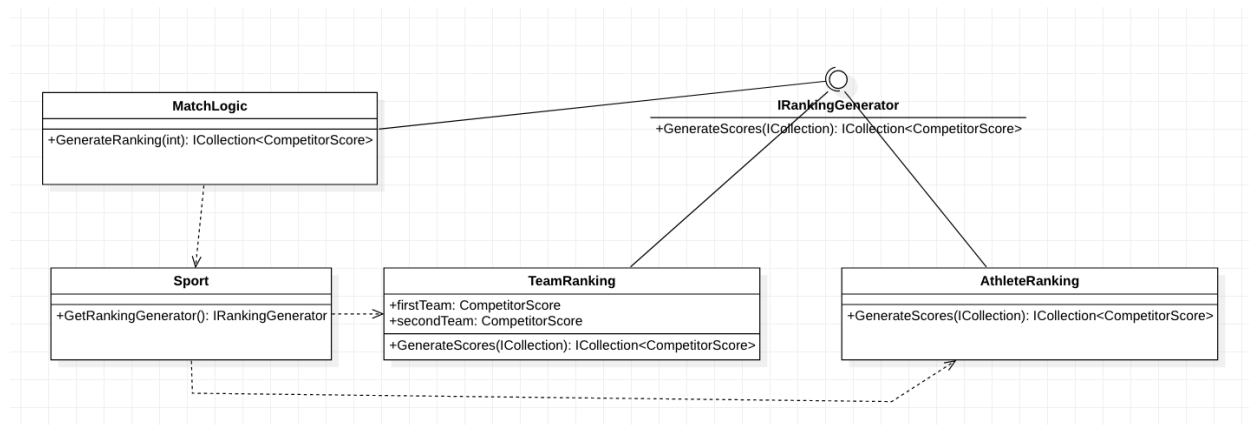
Queremos destacar dos de las funcionalidades muy importantes del sistema y de esta capa lógica y son la generación de fixtures y la generación de ranking.

Para implementar el fixture, nos basamos en dos patrones. Uno es Strategy y el otro es Reflection. Más adelante se justificará la decisión de por qué utilizamos estos patrones. La lógica del fixture utiliza una interfaz IFixtureGeneratorStrategy que es la que tiene el contrato de cómo generar un nuevo fixture.

En el siguiente diagrama se puede observar la implementación de Strategy.



Otra funcionalidad importante es la generación del ranking. Para implementarlo, nos basamos en los patrones Strategy (la jerarquía encapsula uno y solo un algoritmo que es **GenerateScores**) y Factory method en **Sport**. La lógica del ranking utiliza una interfaz **IRankingGenerator** que es la que tiene el contrato de cómo generar el ranking. En el siguiente diagrama se observa la implementación del mismo:

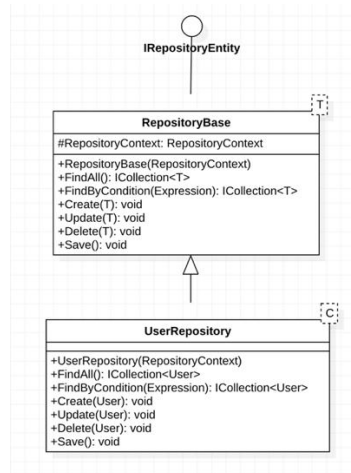


Nota: los diagramas de clases de los tests los pusimos como anexos.

2.2.2 Capa del Repositorio

Para la persistencia de los datos en esta capa, utilizamos el patrón Repositorio. Este patrón permite separar la persistencia y el acceso a los datos de la capa de servicios que es la que contiene a la lógica del sistema. Esto logra que la capa de servicios ignore como los datos son persistidos y accedidos. La capa de servicios accederá a los datos a través de unit of work que contiene a los distintos repositorios. Dentro de los diferentes repositorios, se encuentra la interacción con la base de datos. La clase unit of work es fundamental para asegurar que todos los repositorios usen el mismo contexto.

También decidimos utilizar generics ya que hay ciertas consultas básicas como Create, Update, Delete que se repiten y nos pareció innecesario tener que poner todas estas consultas repetidas en cada clase. Con generics evitamos el código duplicado. Esto lo logramos haciendo que cada repositorio concreto tenga como atributo el repositorio genérico y puedan acceder a través de el a realizar las operaciones. A continuación, se muestra un ejemplo de generics aplicado al sistema:



Creamos repositorios específicos para todas las entidades ya que nos pareció que podía aportar con que la solución sea más extensible. Esto se debe a que si en un futuro se quiere agregar una consulta específica solo se debe agregar la consulta a esa entidad ya que ya está creada.

El repositorio fue separado en dos. Por un lado, la interfaz y por el otro la implementación de la interfaz. La interfaz del unit of work tiene a las distintas interfaces de los repositorios y la implementación del unit of work tiene a las distintas implementaciones de las interfaces.

Diagrama de clases del Repositorio

Las interfaces se encuentran en el paquete `Sports.Repository.Interfaces` pero decidimos ponerlas en este diagrama junto con las clases del repositorio que se encuentran en `Sports.Repository` ya que cada una de las clases están directamente relacionadas con los contratos que proveen las interfaces. Por ejemplo, `MatchRepository` implementa `IMatchRepository`; `UserRepository` implementa `IUserRepository` y así sucesivamente con cada clase que se observa en el diagrama.



2.2.3 Capa de la Web API

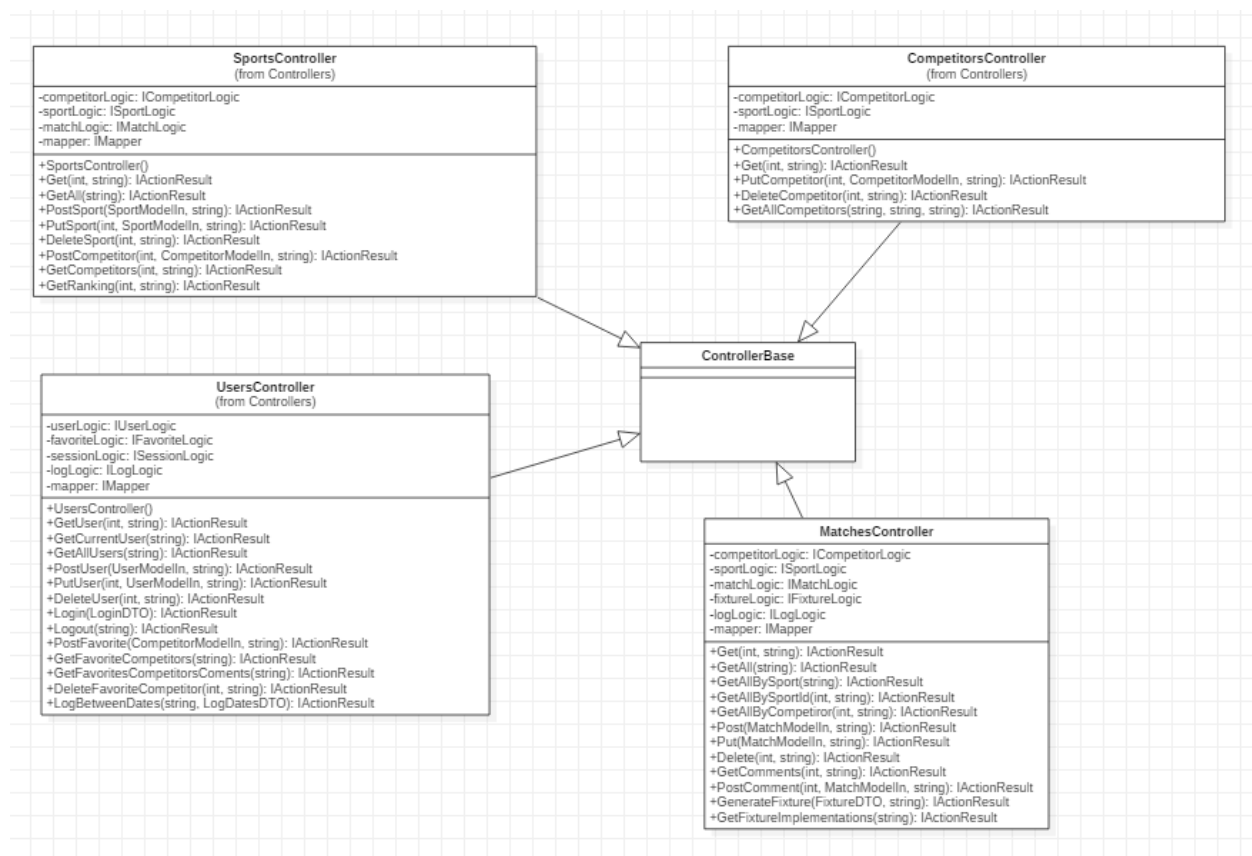
En esta capa se encuentran todos los controllers que definen los distintos end points que es por donde se puede interactuar con el sistema. Es a través de esta capa (Sports.WEBAPI) que los usuarios van a poder tener una interacción con el sistema. Esta va a ser la capa que atrapa las excepciones que el sistema tire devolviendo los códigos de error de HTTP. Cada controller tiene configurado distintas rutas para poder pegarle a la API.

Realizamos una arquitectura que cumple con api REST ya que utilizamos los cuatro verbos más importantes de HTTP que son: GET, POST, DELETE Y PUT.

Hicimos que todos los controllers hereden de ControllerBase que te permite exponer los servicios de la api. Todas las peticiones http que se realizan se delegan a la lógica de negocio y las mismas realizan las acciones correspondientes devolviendo el resultado o en caso contrario mensajes del error encontrado. Más adelante se mostrarán los end points de la Web Api.

En esta capa también decidimos crear modelos para los diferentes controladores. Trabajamos con ModelIn y ModelOut para los diferentes controladores. Más adelante se justifica la elección de los mismos.

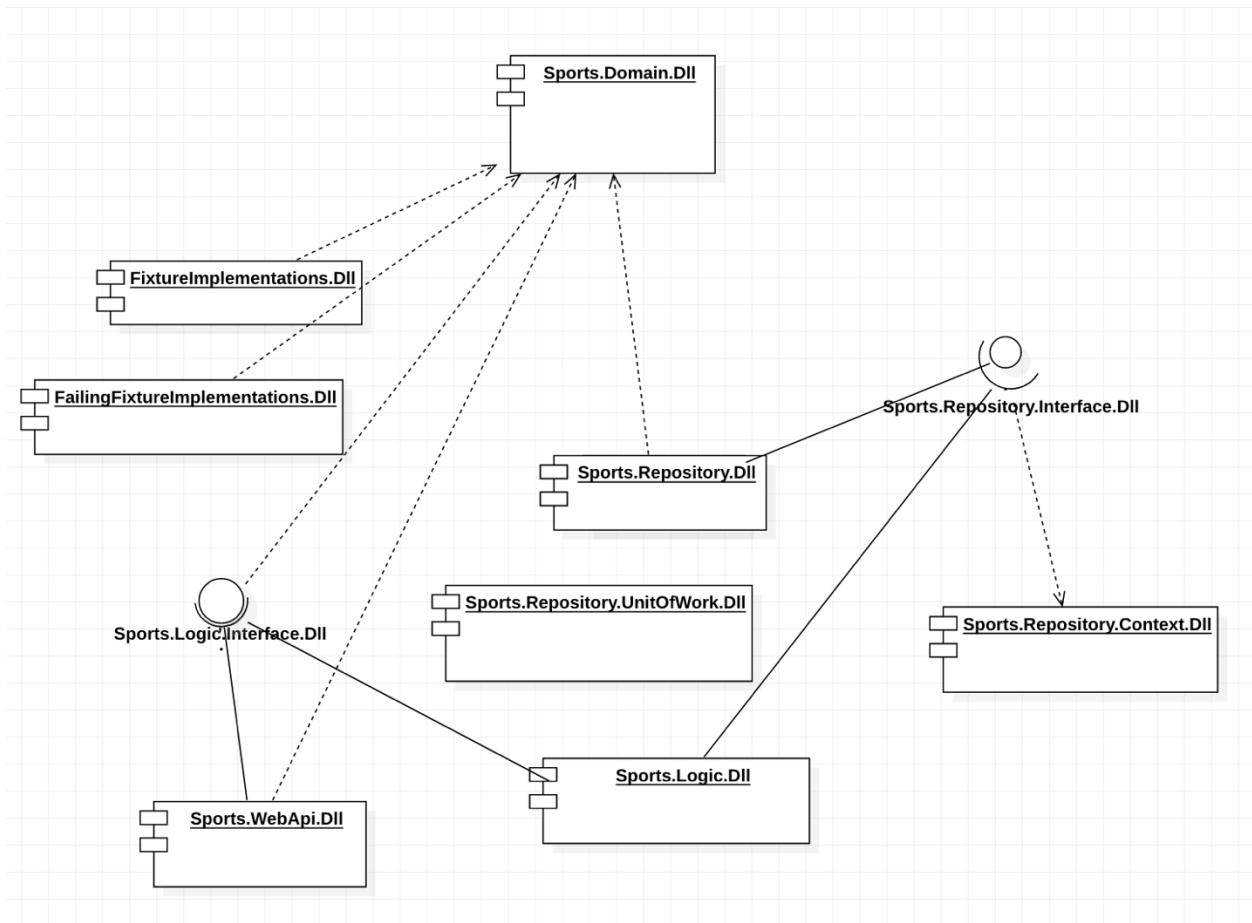
Diagrama de clases de la Web Api



2.3 Diagrama de Componentes

En el siguiente diagrama mostraremos la estructura general del diagrama de componentes del sistema. Se pueden observar como el sistema se divide en distintos componentes mostrando las dependencias entre ellos. Los dlls que se generan luego de haber compilado la aplicación.

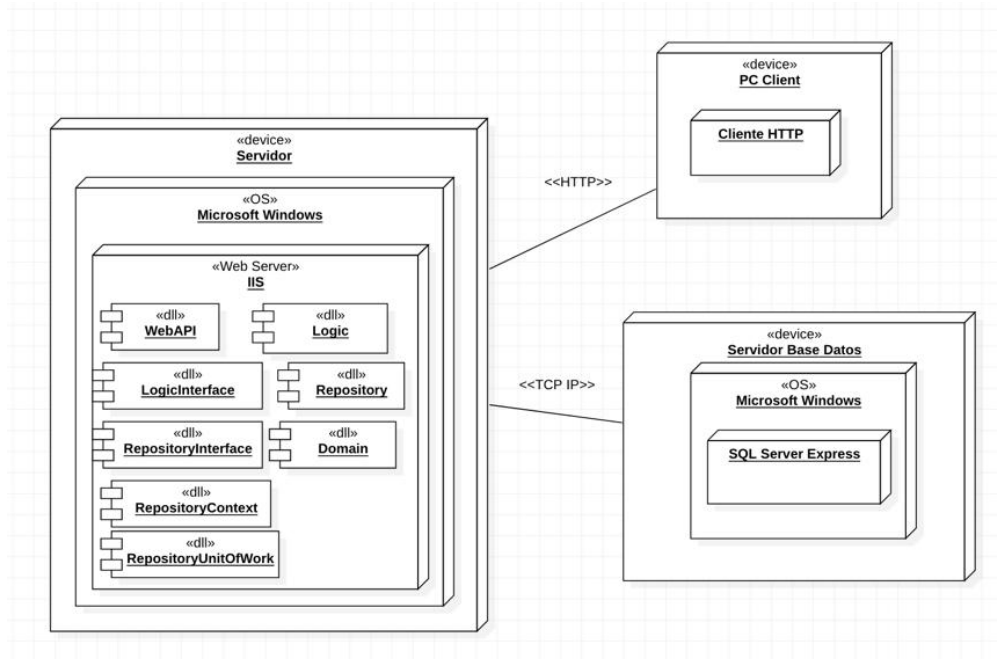
Nota: Los dll de las interfaces tanto de repositorio como de lógica, proveen las distintas interfaces para todas las clases de sus instancias (logic y repository).



2.4 Diagrama de Entrega

Por último, se muestra el diagrama de entrega con los distintos componentes.

No incluimos los paquetes de pruebas ya que consideramos que son relevantes para el que lo desarrolla, pero no aporta ningún valor al cliente.



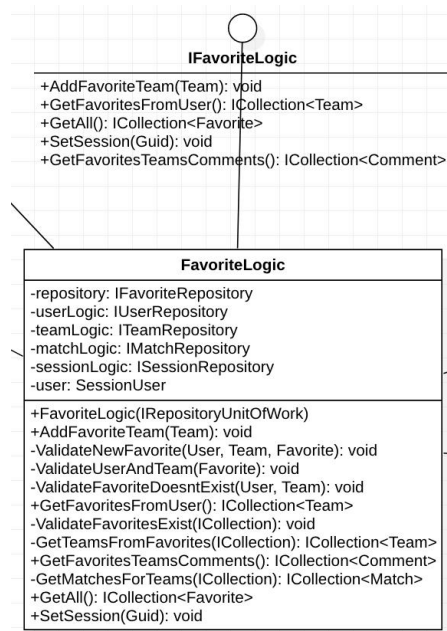
3. Justificación de diseño de las funcionalidades principales

Al momento de realizar el sistema se fueron tomando algunas decisiones de diseño con el objetivo de poder llegar a una solución respetando los principios SOLID y aplicando ciertos patrones de diseño que fuimos aprendiendo en los cursos de diseño 1 y 2. Todas las decisiones que fuimos tomando tenían el fin de lograr una solución que sea lo más extensible y flexible posible para que el día de mañana si se necesita agregar una nueva funcionalidad, esta no tenga gran impacto en el resto del sistema.

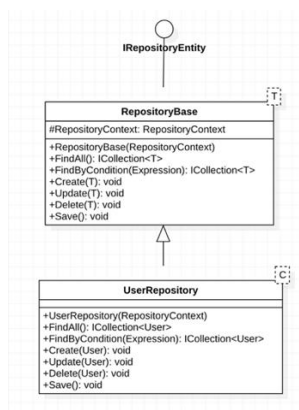
A continuación, se mostrarán las decisiones más importantes junto con sus respectivas justificaciones.

En cuanto a lo que se refiere a la persistencia de los datos, desacoplamos totalmente el mismo de la lógica de negocio. En otras palabras, la lógica no depende de la implementación de la persistencia de datos. Este concepto se refiere al principio conocido como DIP (Dependency Inversion Principle). Este principio dice que es conveniente que se dependa de una abstracción. Las clases de alto nivel no deberían depender de una de bajo nivel de implementaciones directas, sino de abstracciones. Aplicando este principio en nuestro sistema, para poder desacoplar la lógica del negocio de la implementación de la persistencia, utilizamos un conjunto de interfaces, uno para cada uno de los repositorios. Estas interfaces se encuentran en un paquete distinto al de la persistencia, permitiendo que la lógica utilice estas interfaces para que pueda interactuar directamente con ellas y no dependa de la implementación de las mismas. De esta manera, tenemos la implementación de la persistencia de datos de una manera extensible, reduciendo el impacto de cambios para en un futuro.

En la siguiente imagen, se muestra un ejemplo de la relación de la lógica con el repositorio. Se utiliza la lógica de favorito como ejemplo.



Otro principio SOLID que utilizamos fue el de Open / Closed Principle que dice lo siguiente: “el sistema debe ser abierto para la extensión y cerrado para la modificación. Abierto para agregar nuevas funcionalidades al sistema y cerrado para que no cambie la lógica de nuestro sistema”. El uso de generics cumple con este principio y se muestra un ejemplo a continuación:



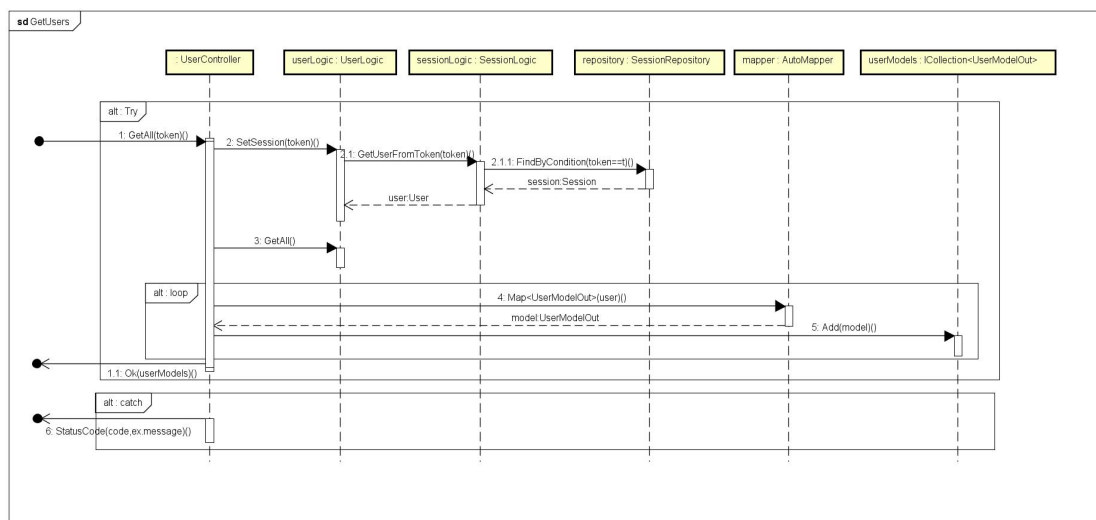
Siguiendo con la persistencia, decidimos hacer el borrado de todas las entidades de manera física por borrado en cascada excepto la entidad de equipo visitante en Match ya que el motor de SqlServer no permite tener dos referencias con borrado en cascada de una entidad a otra. Como solución, decidimos que en la lógica de equipos hacer el borrado físico de forma manual de todos los partidos que contienen a este mismo equipo como visitante. La decisión de hacer borrado físico en cascada se debe a que para obtener los datos y

mantener los mismos nos daría la ventaja de no tener que verificar si las entidades obtenidas fueron borradas o no y así poder evitar errores.

Para los modelos en la Web API decidimos utilizar AutoMapper. Este es un mapeador de objeto a objeto, recibe una instancia de un objeto y me genera otra instancia de otro objeto que tiene diferente estructura. Esto nos permitió ahorrar bastante código. Llevándolo a nuestro código, se muestra un ejemplo con User: nos permitió mapear de UserModelIn a la entidad User y de User a UserModelOut. Lo mismo con las diferentes entidades. La decisión de utilizar ModelIn y ModelOut para los modelos

fue para poder exponer únicamente las properties que consideremos aptas para el usuario. Por ejemplo, en UserModelIn está la password pero en UserModelOut no nos interesa mostrarle la Password. También creamos entidades simples para poder traer sub entidades sin datos redundantes. Por ejemplo, CommentModelOut tiene un UserSimpleModelOut que solo muestra el id y Username de su usuario. Ningun ModelIn requiere todos los datos, esto se debe a que en la lógica los “modify” ignoran todos los campos “null” y la entidad se queda con su dato original en ese campo.

Se muestra el diagrama de secuencia de cómo se traen todos los modelos de usuarios para poder entender de forma más clara la comunicación entre las capas.



Decidimos que las excepciones generadas por inconsistencias de datos vayan en una carpeta dentro del proyecto del dominio, que las excepciones de la lógica vayan en una carpeta dentro del proyecto de la lógica y que las excepciones del repositorio vayan en una carpeta dentro del repositorio. La decisión de tener las excepciones en diferentes proyectos y no en uno solo es para que disminuya el acoplamiento y no tener dependencias innecesarias. El mismo concepto para las constantes. Creamos constantes para el dominio y constantes para la lógica.

Tratamos de que las excepciones sean lo más específicas posibles para que nos permita tener un mayor control sobre el flujo de los mismos. Sin embargo, todas las excepciones de cada componente tienen una excepción “padre” el cual se encuentra en la interfaz de su proyecto para que sea conocido por otros proyectos que usan de este. En la Web API

hacemos catch a todas esas excepciones “padres” ya que por el uso de servicios “Scoped” hacemos reflection de todas las interfaces en el Startup de esta. Más allá de que tengamos excepciones específicas, decidimos tener por último a la Excepción genérica para en el caso de que si nos olvidamos de atrapar alguna genérica tener esa y evitar que se nos caiga el programa.

En cuanto a lo que respecta a la lógica del negocio, se definieron un conjunto de interfaces. Si en el día de mañana se quiere cambiar la implementación de alguna lógica, todos los servicios que usen a esta lógica dependen de una interfaz y no de implementación directa. Estas interfaces las definimos en otro paquete “Sports.Logic.Interface” para disminuir nuevamente el acoplamiento. Nos pareció que no era conveniente poner a estas interfaces ni en el paquete de la lógica ni en el de la Web API.

Como se mencionó anteriormente, una de las principales funcionalidades del sistema es la generación de fixtures. Para esta funcionalidad utilizamos Strategy y Reflection.

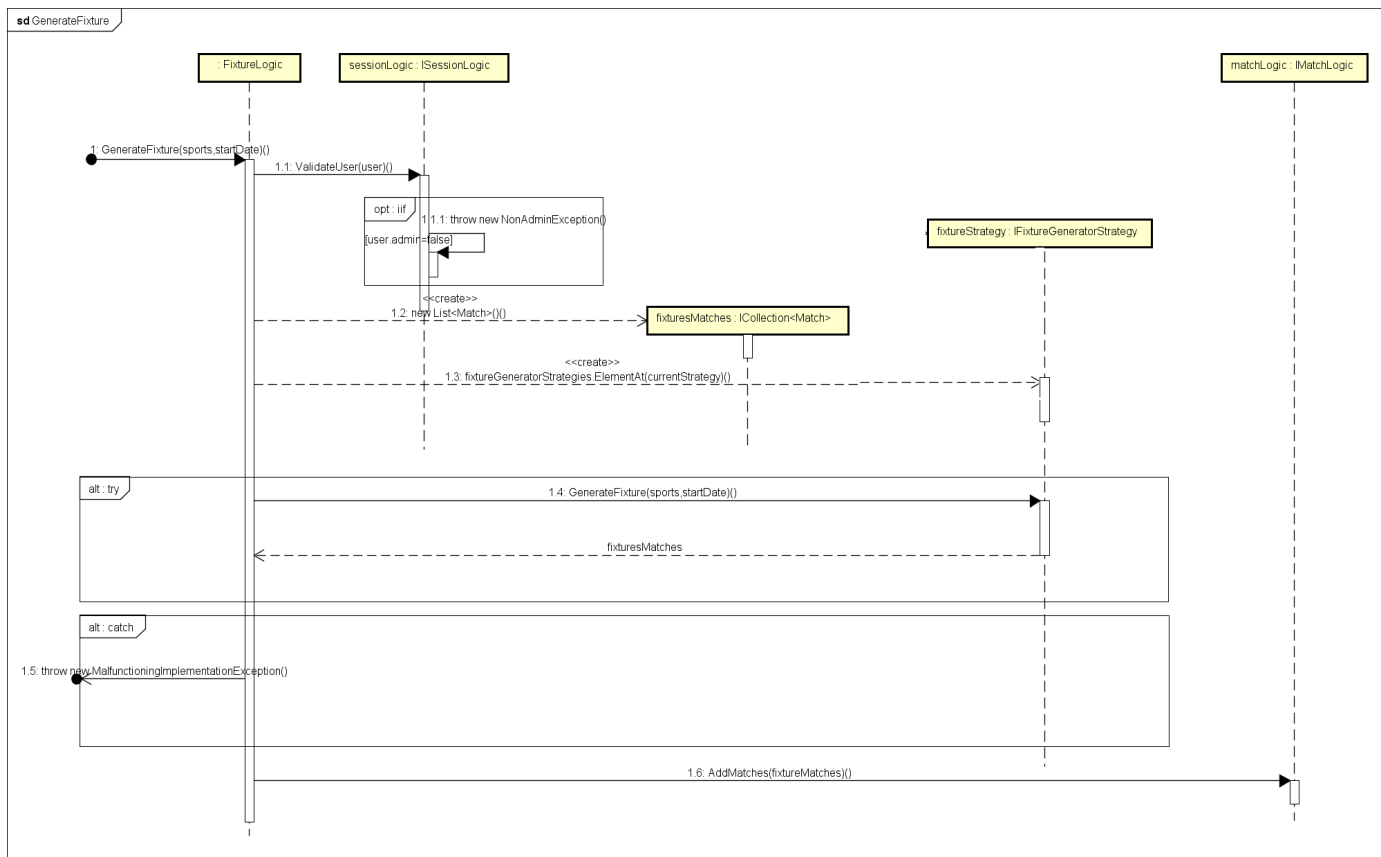
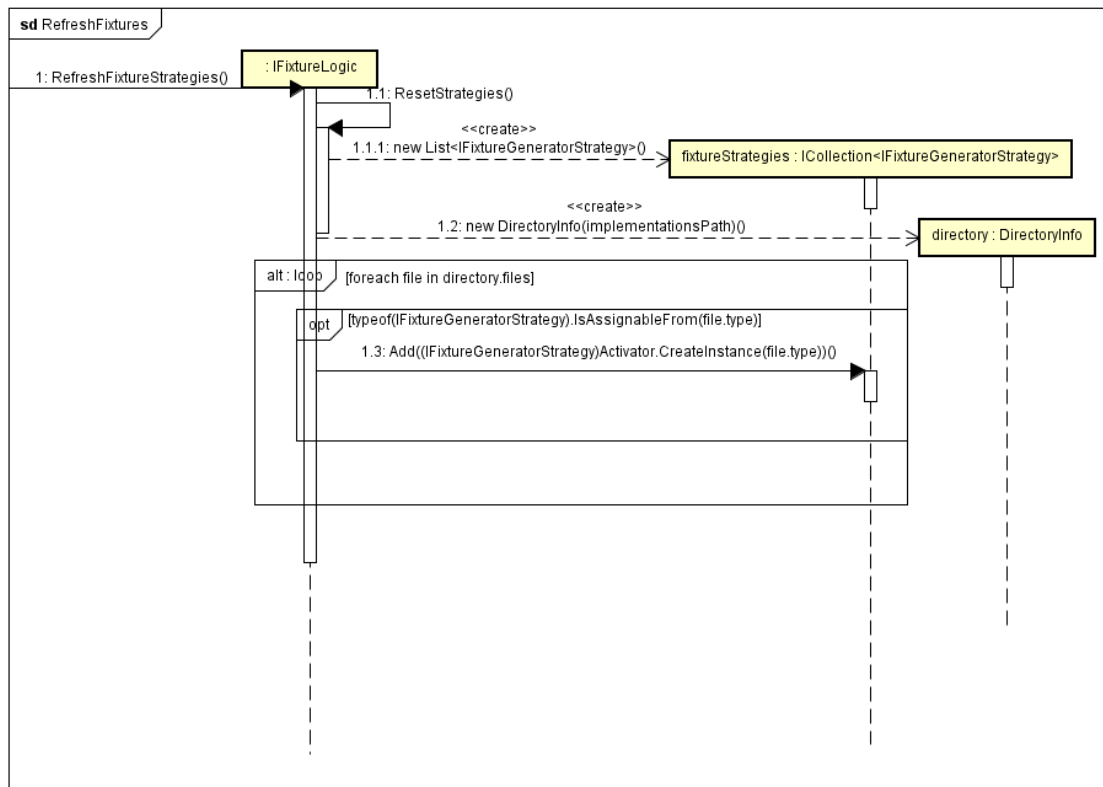
Nos pareció adecuado utilizar el patrón Strategy ya que, si en un futuro se quiere generar un nuevo tipo de fixture, lo único que hay que hacer es que este nuevo fixture concreto implemente la interfaz IFixtureGeneratorStrategy y mediante el polimorfismo, se va a poder ejecutar la generación de este nuevo fixture.

En este caso, la clase FixtureLogic (sería la clase Context) no implementa directamente el algoritmo. FixtureLogic está formado por IFixtureGeneratorStrategy. Esto hace que FixtureLogic sea independiente de cómo se implementa el algoritmo de generación de fixture. FixtureWeekendMatches y FixtureBackAndForthDaily (son los strategies concretos) son lo que implementan IFixtureGeneratorStrategy. En tiempo de ejecución se puede cambiar el atributo y que pase a ser cualquiera de los concreteStrategy.

El patrón reflection no se llegó a enseñar para este obligatorio, pero nos resultó interesante estudiarlo y poder implementarlo en este obligatorio ya que nos permitió independizar la implementación de los fixtures completamente del sistema. Permite que sea extensible sin tener que recompilar el sistema.

Estas dlls cargadas por reflection sin embargo no son persistidas lo cual genera un problema con la arquitectura web api rest que crea una nueva instancia de FixtureLogic por cada vez que se llama. Para poder cargar nuevas estrategias en una llamada y luego poder usarlas en otra, decidimos guardar todas las estrategias en una lista estática dentro de FixtureLogic para que estas no se pierdan por cada llamada.

A continuación, se muestran dos diagramas de secuencia del fixture. Con el primero quisimos mostrar como se cargan los fixtures de un directorio y el segundo como se usa uno de los fixtures previamente cargados.



4. Impacto de cambio de las nuevas funcionalidades

Para este segundo obligatorio tuvimos que incorporar algunas nuevas funcionalidades y realizar ciertos cambios. Estos fueron:

- *El encuentro se puede realizar entre dos o más competidores.* Esta nueva función fue la que tuvo un mayor impacto de cambio en el sistema ya que anteriormente nos referíamos a un equipo local y visitante en cada encuentro. En la versión nueva se maneja una lista de “competidores” en cada encuentro sobre la cual se itera para mostrar la información de cada competidor. Para que los sistemas de fixture sigan funcionando como antes, en la clase deporte agregamos una variable requerida llamada “amount” la cual indica la cantidad de competidores por encuentro para un deporte específico. Esto permite a los algoritmos de fixture saber cuántos competidores se requieren para generar un encuentro.
- *Un encuentro debe contar con un resultado.* Dos alternativas para ingresar el resultado (dos competidores o más de dos competidores). Por la arquitectura de entity framework core, para poder permitir que los encuentros tengan competidores ilimitados, fue necesario crear una clase intermedia la entre competidor y encuentro la cual se llama “CompetitorScore”. Dentro de esta clase se encuentra una referencia a la entidad de competidor y un puntaje que refleja sus puntos en un partido o su posición en una competencia. Debido a que la clase intermedia fue necesaria, el impacto de cambio de esta clase fue mínimo.
- *Generación del ranking.* Debido a que la forma de calcular el puntaje total de los resultados difería según si el encuentro es de 2 o más participantes, creamos una interfaz que implementa el algoritmo de generación de ranking para los partidos de un deporte dependiendo de la cantidad de participantes. Estas clases son independientes del sistema (fuera de deporte que las crea con Abstract Factory) por lo que produjeron un impacto de cambio mínimo. Cada vez que se pide el ranking este se calcula lo cual puede ser un poco ineficiente, pero produce el menor impacto de cambio posible en el sistema.
- *Registrar acciones en el sistema.* Para mantener el log independiente del sistema se creó una interfaz que provee las funciones básicas necesarias para mantener un registro de un sistema. La implementación se independizó del sistema al ser manejada por referencia desde la web api de forma que el sistema de lógica no entre en contacto con ninguna implementación de este. A su vez para hacer que el log sea independiente completamente, este se registra en un archivo de texto independiente de la base de datos para que no dependa de ninguna implementación previa de persistencia como la base de datos. Ya que el log fue instanciado desde la web api, creímos apropiado que este debería de ser manejado desde la misma para que futuras implementaciones de acceso a la lógica no dependan de un log.

- *Generacion de fixtures con reflection.* Esta función ya había sido implementada en la primera entrega por lo cual no hubo impacto de cambio salvo algunas modificaciones aconsejadas por la corrección. Las principales fueron permitir listar los diferentes algoritmos de fixtures para que el usuario pueda elegirlos visualmente con mayor facilidad y también que la dirección de las dlls sea definida en un archivo json que la lógica lea por defecto en lugar de enviar la dirección por la web api.

5. Front-End: Aplicación Angular (SPA)

Utilizamos el framework que nos da angular para realizar la aplicación web (Single Page Application). Uno de los beneficios que posee el framework es el de agregar directivas al HTML.

Consiste en lo siguiente: a partir de los controllers, hacemos llamadas que son asincrónicas al backend para pedirle por determinado recurso. Cuando el backend nos consigue el recurso buscado, se actualiza la vista sin tener que recargar toda la página. El que hace de “intermediario” entre el controller y la vista son las directivas mencionadas anteriormente. El ruteo es importante para poder hacer uso de varios controladores y varias vistas.

A continuación, mencionaremos los principales módulos de una aplicación angular:

- Modulo principal AppModule.ts. Este contiene como componente principal AppComponent. En AppComponent se definen todas las rutas de los distintos componentes del sistema a través del Router Module.
- Un component principal AppComponent que es el componente encargado de comunicarse con el resto de los componentes. Este hace “hice” y “show” de los componentes.
- Carpeta para los servicios
- Carpeta para los modelos
- Carpeta para los guards
- Carpeta para los componentes
- Carpeta para el environment
- Carpeta para las clases con las entidades necesarias

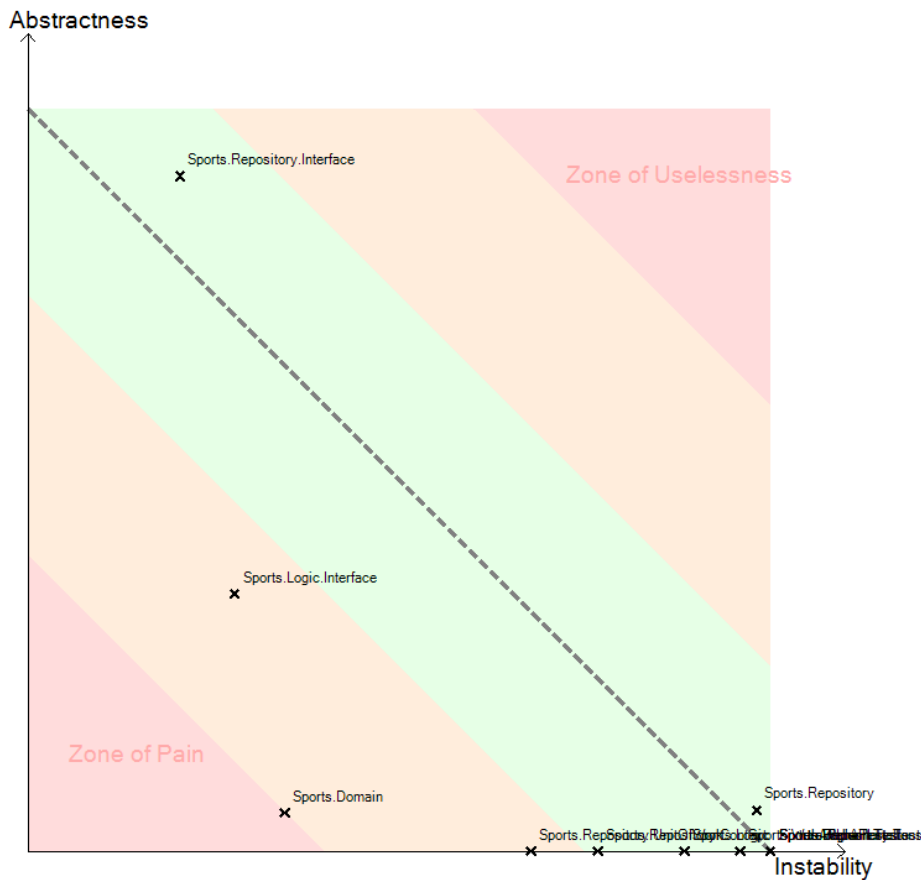
6. Informe de métricas

Para realizar el informe de las metricas, utilizamos la herramienta conocida como NDepend. Esta es una herramienta de analisis estatico para el codigo administrado. Esta herramienta nos permite visualizar las dependencias mediante graficos dirigidos y una matriz de dependencia.

Estas métricas son indicadores que permiten relevar medidas relacionadas a la calidad del software.

Es importante destacar que si las métricas dan dentro del rango esperados no significa que el diseño sea excelente. Lo mismo en el caso que las métricas no estén dentro de lo esperado, no significa que el diseño sea malo.

Para poder realizar un mejor análisis, se muestra una gráfica generada en NDepend donde en el eje x se muestra la inestabilidad y en el eje y se muestra la abstracción a nivel de paquetes.



Análisis de la gráfica obtenida

A partir de la proyección se puede notar que cuanto más cerca de la zona verde estas, mejor son los resultados de abstracción y estabilidad.

Como primer análisis se puede observar el paquete `Sports.Domain` se encuentra dentro de la zona roja ya que casi todos los paquetes dependen de él. A su vez, este no implementa ninguna interfaz por lo tanto su nivel de abstracción es nulo. Para solucionarlo, podríamos hacer que cada clase tenga su propia implementación de modelos de objetos. Ejemplos DTO'S.

`Sports.Logic.Interface` se encuentra en la zona naranja. Esto significa que es un paquete bastante estable. Este paquete no se encuentra dentro de la zona verde ya que no es muy abstracto.

Por último, hablaremos un poco de la zona verde que es la zona de alta inestabilidad y bajo nivel de abstracción. En esta zona se encuentran todos los restantes paquetes incluidos los paquetes de pruebas. Es razonable que en esta zona se encuentren los paquetes de pruebas ya que nadie depende de esas clases y él por el contrario depende de varias. También se encuentran todas las implementaciones de las interfaces. Esto tiene como ventaja de que estas implementaciones se comunican solamente con sus contratos.

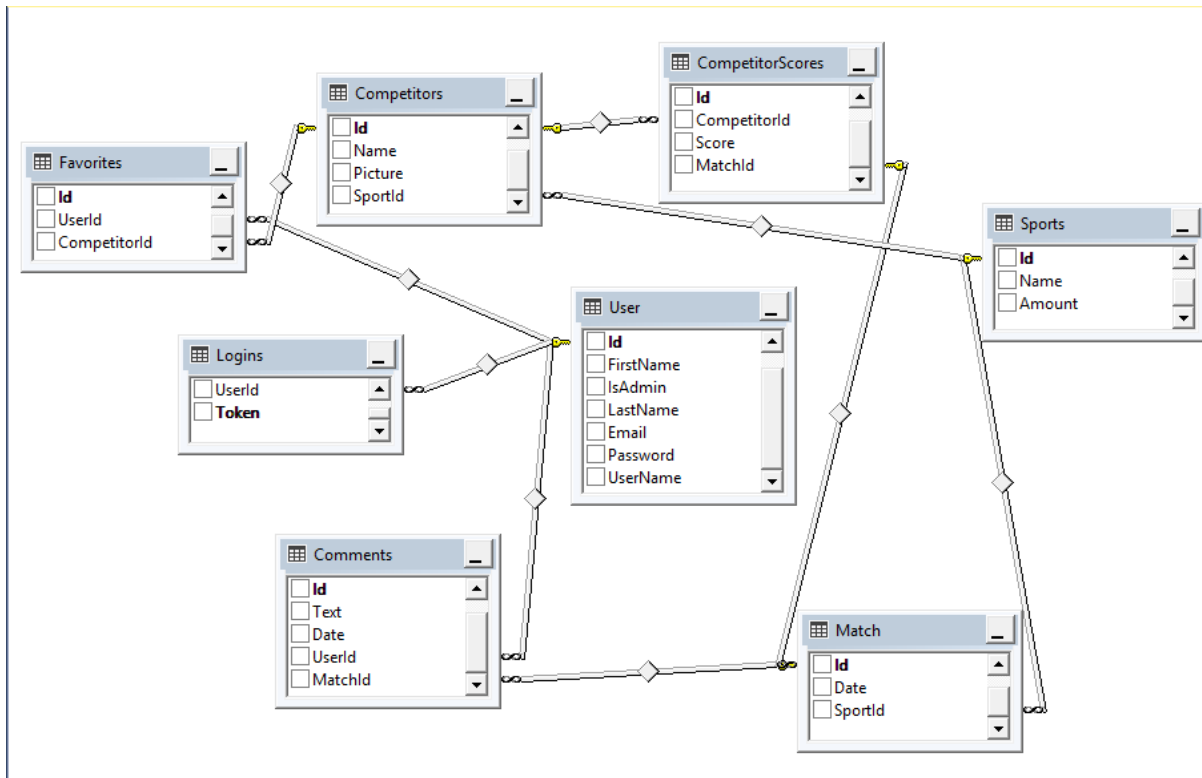
6. Endpoints Web API

Aclaraciones:

- Los Id son de tipo int.
- Los tokens son de tipo Guid y se pasan por header.

7. Modelo de Tablas

En la siguiente imagen se muestra como se mapearon las tablas para persistirlas en la base de datos. Utilizamos “Code First” por lo que las mismas se autogeneraron por el Entity Framework. Las tablas se almacenaron en SQL Server.



Datos de prueba

Nota: En la sección de anexo dejamos una imagen de cada una de las tablas de la base con su datos de prueba.

8. Justificación y evidencia de Clean Code

En esta sección se mostrarán las evidencias del cumplimiento de Clean Code, basándonos en los principios de Robert Martin en su libro “Clean Code”.

Capítulo 2: Nombres

Durante el desarrollo del sistema, se buscó en todo momento de que los nombres de las variables y los métodos sean nemotécnicos para que al leerlos se entienda la intención del mismo y no se pierda tiempo en pensar cuál es su intención.

```
public class User
{
    [Key]
    public int Id { get; set; }
    public string FirstName { get; set; }
    public bool IsAdmin { get; private set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public string UserName { get; set; }
}
```

Capítulo 3: Funciones

Buscamos en todo momento que los métodos sean cortos y entendibles. Nos pasó que, en muchos casos, algunos métodos tenían muchas líneas de código y quizás no eran lo más legibles, pero fuimos aplicando la técnica de refactoring y encapsulando los mismos en otros métodos para que sean más claros a la hora de leerlos.

```
2 references | Rafael Alonso, 18 days ago | 1 author, 4 changes | 0 exceptions
private void ValidateUser(User user)
{
    CheckNotNull(user);
    user.IsValid();
    CheckNotExists(user.UserName, user.Id);
}

1 reference | itaimiller1, 8 days ago | 2 authors, 4 changes | 0 exceptions
private void CheckNotExists(string username, int id = 0)
{
    if (repository.FindByCondition(u => u.UserName == username && u.Id != id).Count != 0)
    {
        throw new UserAlreadyExistException(UniqueUsername.DUPLICATE_USERNAME_MESSAGE);
    }
}

1 reference | itaimiller1, 12 days ago | 2 authors, 4 changes | 0 exceptions
private void CheckNotNull(User user)
{
    if (user == null)
    {
        throw new InvalidNullValueException(NullValue.INVALID_USER_NULL_VALUE_MESSAGE);
    }
}
```

Capítulo 4: Comentarios

En ningún momento tuvimos la necesidad de realizar comentarios en el código ya que todas las variables y métodos revelan su intención.

Capítulo 5: Formateo

Tenemos bien formateado el código tanto verticalmente como horizontalmente. En cuanto al formato vertical se cumple con: una instrucción por línea, indentado para que sea sencillo leer, las instancias de las variables se encuentren al inicio de la clase, las funciones dependientes están lo más cerca posible. En cuanto al formato horizontal se cumple con: líneas cortas para no hacer scroll, indentado para facilitar la lectura.

Capítulo 7: Manejo de errores

Fuimos generando excepciones propias para tener un mayor control del flujo de los mismos e informarle al usuario cual fue el error para que no lo vuelva a cometer. Generamos distintos paquetes de Excepciones. Uno en el dominio, otro en la lógica y otro en el repositorio. También tratamos con los try catch en la capa de Web API comunicándole al usuario del error cometido.

Capítulo 9: Pruebas unitarias

En todo momento se aplicó la metodología conocida como TDD realizando primero el unit test y luego escribir el código para esa prueba. Las pruebas unitarias cumplen con FIRST. Siempre tuvimos la intención de que las pruebas sean lo más legibles posibles.

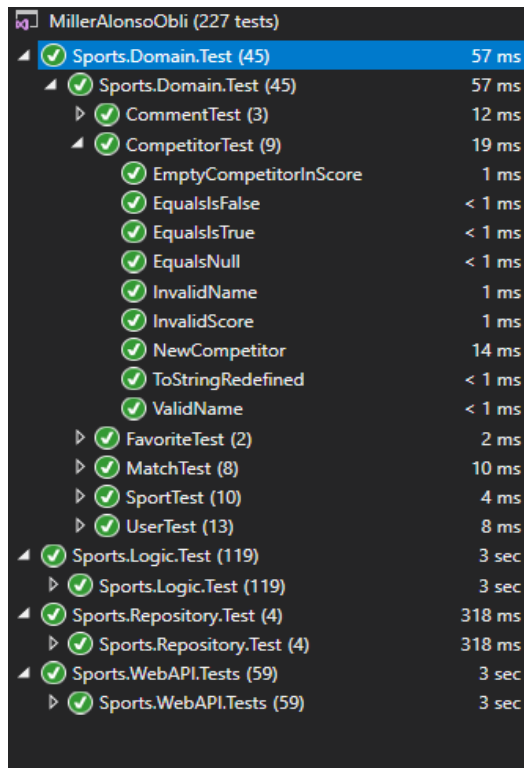
```
[TestMethod]
0 references | Rafael Alonso, 3 days ago | 3 authors, 5 changes | 0 exceptions
public void DeleteUser()
{
    userLogic.AddUser(user);
    userLogic.RemoveUser(user.Id);
    Assert.AreEqual(userLogic.GetAll().Count, 1);
}
```

Capítulo 12: Emergencia

Cumplimos con este capítulo en los siguientes puntos:

- TDD para tener un buen conjunto de pruebas unitarias y tener respaldado todo el código.
- Se evitó el código duplicado.
- Cada variable o método expresan su intención.
- Se minimizó el número de clases y métodos.

9. Análisis de las pruebas



MillerAlonsoObli (227 tests)	
✓ Sports.Domain.Test (45)	57 ms
✓ Sports.Domain.Test (45)	57 ms
✓ CommentTest (3)	12 ms
✓ CompetitorTest (9)	19 ms
✓ EmptyCompetitorInScore	1 ms
✓ EqualsIsFalse	< 1 ms
✓ EqualsIsTrue	< 1 ms
✓ EqualsIsNull	< 1 ms
✓ InvalidName	1 ms
✓ InvalidScore	1 ms
✓ NewCompetitor	14 ms
✓ ToStringRedefined	< 1 ms
✓ ValidName	< 1 ms
✓ FavoriteTest (2)	2 ms
✓ MatchTest (8)	10 ms
✓ SportTest (10)	4 ms
✓ UserTest (13)	8 ms
✓ Sports.Logic.Test (119)	3 sec
✓ Sports.Logic.Test (119)	3 sec
✓ Sports.Repository.Test (4)	318 ms
✓ Sports.Repository.Test (4)	318 ms
✓ Sports.WebAPI.Tests (59)	3 sec
✓ Sports.WebAPI.Tests (59)	3 sec

Como se puede ver en la imagen, las 227 pruebas unitarias devuelven el resultado esperado. Esto nos aseguró de que todos los métodos funcionan de forma correcta cubriendo todos los casos posibles.

Cobertura de las pruebas

Como se puede ver en las siguientes imágenes, se tiene una excelente cobertura de código con las pruebas tanto en el paquete del dominio como en el paquete de la lógica. Que la cobertura de pruebas sea alta, nos asegura una mejor calidad del código ya que está cubierta la gran mayoría del mismo debido a que se utilizó la metodológica TDD para realizar el mismo. Cabe resaltar que los métodos privados no los probamos de manera unitaria, sino que, bajo la integración de otro método, ya que no era posible probarlo de manera independiente debido a su visibilidad.

Decidimos crear archivos json (testFilePaths.json) en las pruebas de la lógica y dominio para importar los dlls de los fixtures y las imágenes de los equipos.

Se muestra la cobertura del dominio: se cubrió en un 98,59%.

└─ sports.domain.dll	8	1,41 %	561	98,59 %
└─ { } Sports.Domain	6	1,11 %	533	98,89 %
└─ AthleteRanking	0	0,00 %	25	100,00 %
└─ AthleteRanking.<>c	0	0,00 %	2	100,00 %
└─ Comment	0	0,00 %	28	100,00 %
└─ Competitor	0	0,00 %	43	100,00 %
└─ CompetitorScore	3	7,14 %	39	92,86 %
└─ Favorite	0	0,00 %	19	100,00 %
└─ Match	1	0,83 %	119	99,17 %
└─ Match.<>c_DisplayClas...	0	0,00 %	4	100,00 %
└─ Session	0	0,00 %	4	100,00 %
└─ Sport	2	2,35 %	83	97,65 %
└─ Sport.<>c_DisplayClass...	0	0,00 %	9	100,00 %
└─ TeamRanking	0	0,00 %	27	100,00 %
└─ User	0	0,00 %	131	100,00 %

Se muestra la cobertura de la lógica: se cubrió en un 98,60%. Lo único que faltó para también cubrirla en 100% fue que en un OrderBy no nos entraba en la expresión lambda en CompetitorLogic

└─ sports.logic.dll	17	1,40 %	1194	98,60 %
└─ { } Sports.Logic	13	1,10 %	1166	98,90 %
└─ CommentLogic	0	0,00 %	42	100,00 %
└─ CompetitorLogic	0	0,00 %	167	100,00 %
└─ CompetitorLogic.<>c	0	0,00 %	4	100,00 %
└─ CompetitorLogic.<>c_...	0	0,00 %	7	100,00 %
└─ FavoriteLogic	0	0,00 %	163	100,00 %
└─ FavoriteLogic.<>c	0	0,00 %	2	100,00 %
└─ FixtureLogic	8	7,34 %	101	92,66 %
└─ MatchLogic	2	0,72 %	276	99,28 %
└─ MatchLogic.<>c_Displa...	0	0,00 %	4	100,00 %
└─ SessionLogic	0	0,00 %	119	100,00 %
└─ SportLogic	0	0,00 %	141	100,00 %
└─ TextLog	3	6,25 %	45	93,75 %
└─ UserLogic	0	0,00 %	95	100,00 %
└─ { } Sports.Logic.Exceptions	4	12,50 %	28	87,50 %

La

web api no se cubrió en un 100% ya que hicimos pruebas de integración con Postman. La cobertura de los controladores de la web api estaban cerca del 100% pero luego al incluir todos los catch disminuyo ya que probar todos estos con mock nos pareció innecesario. De todas formas, en la clase UserControllerTest agregamos varias pruebas mock para todos los catch de excepciones (los cuales son los mismos para todos los métodos) por formalidad de TDD.

De la misma forma, el repositorio no se cubrió al 100% debido a que por limitaciones de mock no se podían generar excepciones genéricas aplicadas al patrón repository, de esta forma hay varias excepciones que no pueden ser cubiertas. Los porcentajes varían según el repositorio ya que algunos tienen más líneas de código por “includes” que otros, sin embargo, lo que no cubren son las mismas dos excepciones de error de base de datos esperado o no lo cuales no pudimos reproducir.

└─ sports.repository.dll	78	18,75 %	338	81,25 %
└─ { } Sports.Repository	76	18,45 %	336	81,55 %
└─ CommentRepository	8	8,42 %	87	91,58 %
└─ CompetitorRepository	8	22,86 %	27	77,14 %
└─ CompetitorScoreReposit...	0	0,00 %	2	100,00 %
└─ FavoriteRepository	8	13,56 %	51	86,44 %
└─ MatchRepository	8	9,64 %	75	90,36 %
└─ RepositoryBase<T>	16	24,24 %	50	75,76 %
└─ SessionRepository	20	57,14 %	15	42,86 %
└─ SportRepository	8	22,86 %	27	77,14 %
└─ UserRepository	0	0,00 %	2	100,00 %
└─ { } Sports.Repository.Exceptions	2	50,00 %	2	50,00 %

10. Evidencia TDD

En la sección de anexos se muestran los distintos commits para evidenciar el uso de TDD de los diferentes proyectos del repositorio de git que fue el utilizado en todo el desarrollo del sistema. Se puede ver claramente cómo se respetan las 3 fases de las pruebas. Primero una prueba que no compila ya que no existe el método a probar [RED], luego agregando lo mínimo para que la prueba pase sin importar si lo hicimos de manera eficiente [GREEN], y por último refactorizando la prueba para que sea eficiente y más legible.

11. Instalación

WebAPP

1. Modificar el archivo appsettings en la carpeta Sports.WebAPI y poner en el ConnectionStrings y cambiar el atributo Database por el nombre de la base de datos que corresponda.
2. Abrir el Sql Server Manangement e importar el respaldo de la base de datos.
3. Luego hacer el deployment de la app para utilizarla. Para ello se debe seguir el instructivo que se encuentra en el siguiente link:
https://aulas.ort.edu.uy/pluginfile.php/332697/mod_resource/content/0/Gu%C3%ADa%20de%20despliegue%20en%20IIS%20-%20OB2.pdf

12. Conclusión

Luego de haber terminado el proyecto pudimos sacar algunas conclusiones.

Primero que nada, nos sirvió haber trabajado con plazos fijos bajo consignas dinámicas (ya que los requerimientos del sistema podían cambiar durante el proceso de desarrollo) para mejorar aspectos de saber seleccionar bien las prioridades y esto pasa mucho en situaciones de la realidad.

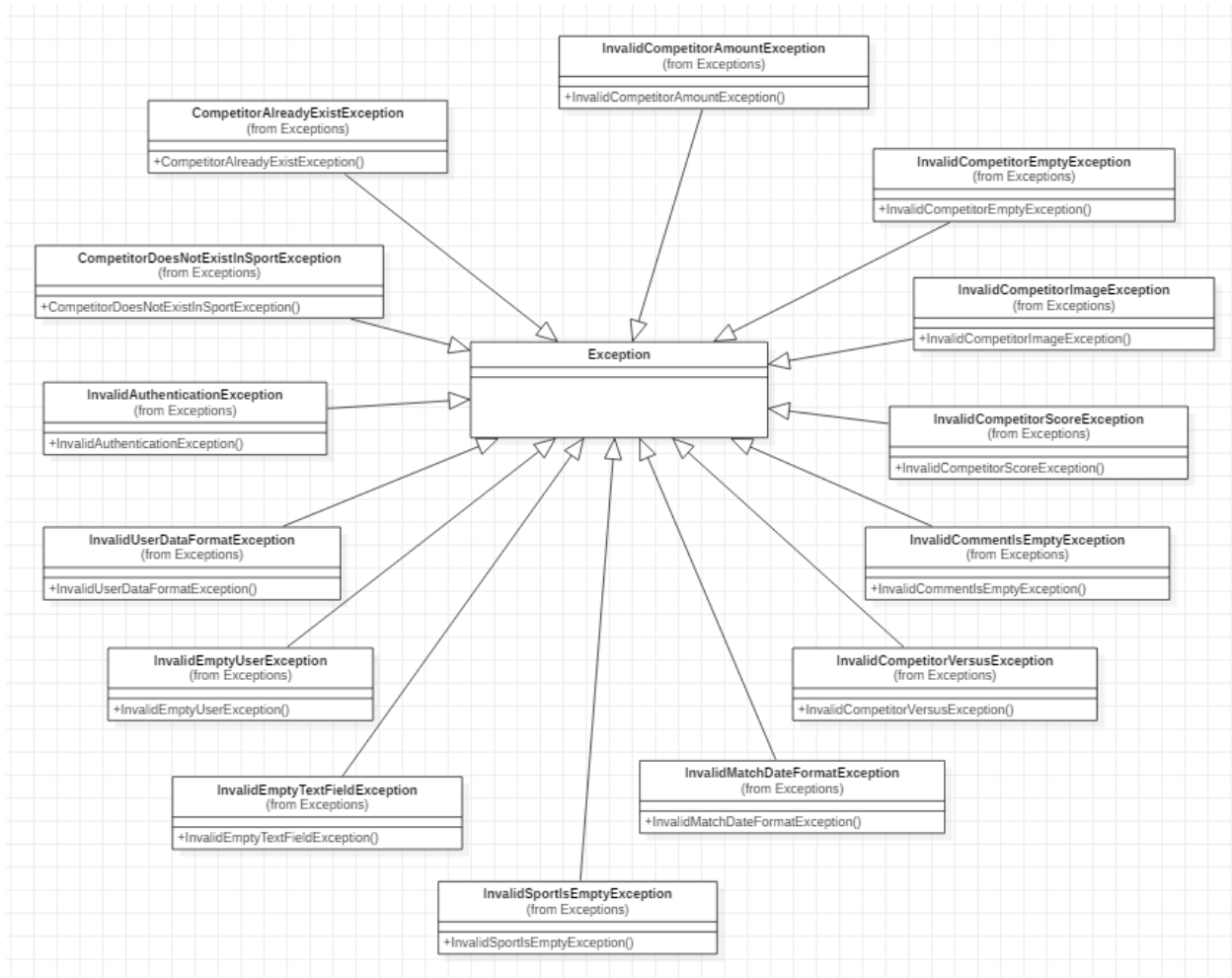
También nos hizo darnos cuenta de la importancia que tiene realizar un buen diseño y arquitectura, más allá de cuales sean los requerimientos. Nos ayudó mucho haber utilizado los patrones de diseño y los principios SOLID. De la misma manera, fue muy importante y le sacamos mucho provecho el haber aplicado Clean Code y la metodología TDD.

Como comentario final, pudimos cumplir con el objetivo de haber realizado un sistema con alto grado de mantenibilidad, flexibilidad y calidad soportando cualquier cambio que vayamos a tener en un futuro ya que no va a impactar mucho en nuestro sistema

Anexos

Diagrama de clases de las exceptions

Sports.Domain.Exceptions:



Sports.Logic.Exceptions:

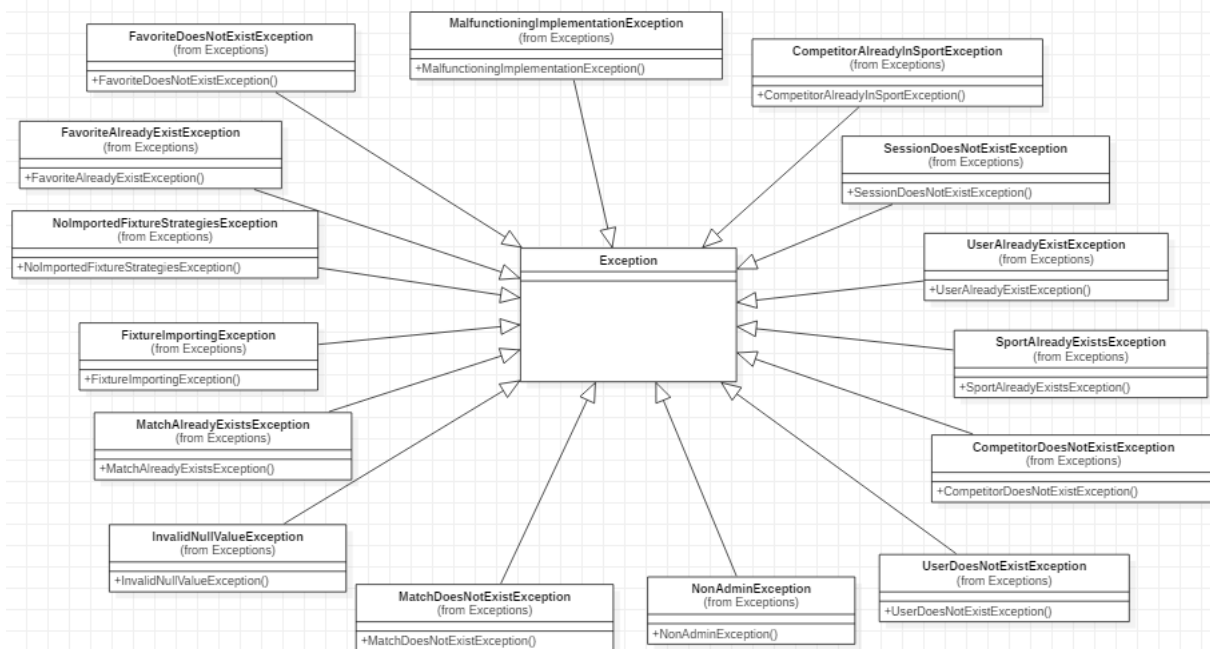


Diagrama de clases de los test del dominio

Sports.Domain.Test es el proyecto que tiene a todas las pruebas del dominio y se pueden apreciar en el siguiente diagrama.

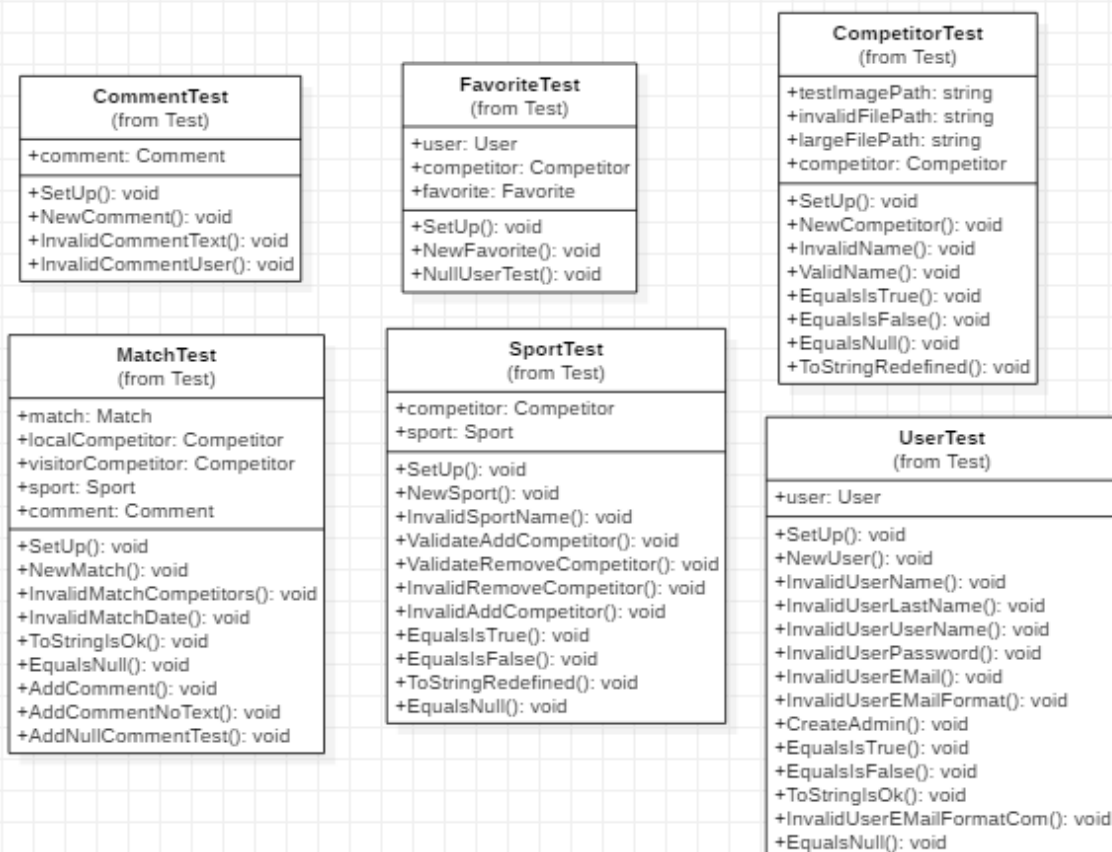
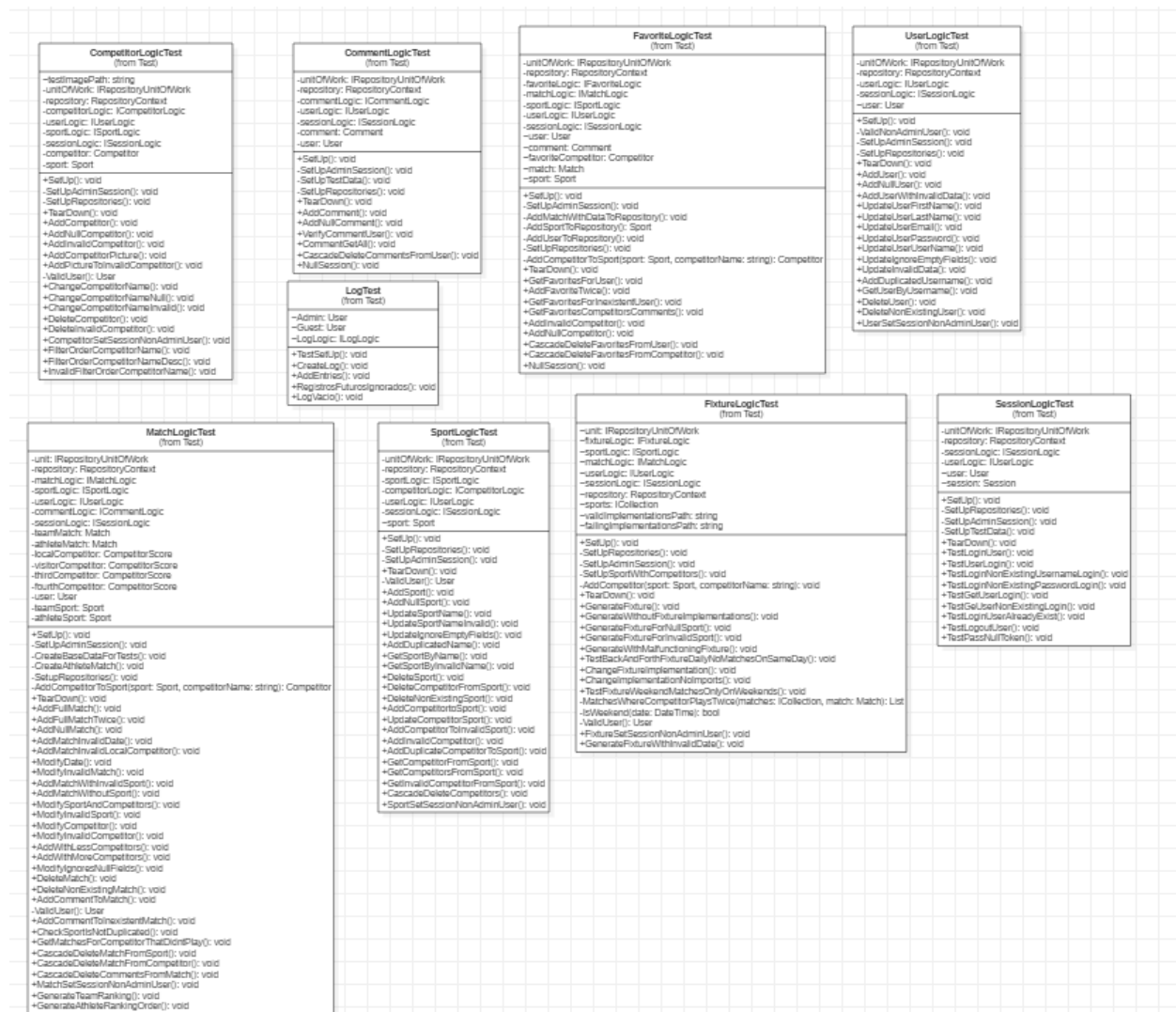


Diagrama de clases de los test de la lógica

Sports.Logic.Test es el proyecto que tiene a todas las pruebas de la lógica y se pueden apreciar en el siguiente diagrama.



User Controller

[POST] "/users/login"

Espera: - username y password.

Respuesta: si el usuario tiene los permisos, le devuelve el token de la sesión.

[POST] "/users"

Espera: UserModelIn y token.

Ej.:

```
{
  "FirstName": "testUser",
  "LastName": "lastName",
  "Email": user@gmail.com,
  "Username": "user2",
  "password": "pass"
}
```

Respuesta: Si el usuario tiene los permisos, te devuelve el usuario creado.

[PUT] "/users/{{userId}}"

Espera: UserModelIn y token.

Ej.:

```
{
  "FirstName": "newName",
  "LastName": "lastNewName",
  "Email": user@gmail.com,
  "Username": "user2",
  "password": "pass"
}
```

Respuesta: Te devuelve el usuario modificado.

[DELETE] "/users/{{userId}}"

Espera: -

Respuesta: Si el usuario tiene los permisos, borra al usuario de la base de datos.

[GET] "/users"

Espera: token

Respuesta: Lista de usuarios sin su password.

[GET] "/users/{{userId}}"

Espera: token

Respuesta: Usuario correspondiente a ese Id sin la password.

[DELETE] “/users/logout”

Espera: token

Respuesta: Si el usuario tiene los permisos, se borra el token de sesión del usuario.

[POST] “/users/favoriteCompetitors”

Espera: CompetitorModelIn y token.

Respuesta: Si el usuario tiene los permisos, te agrega el favorite competidor.

[GET] “/users/favoriteCompetitos”

Espera: token

Respuesta: lista de equipos favoritos.

[GET] “/users/favoriteComments”

Espera: token

Respuesta: lista de comentarios de los equipos favoritos.

[DELETE] “/users/favorite/id”

Espera: token

Respuesta: deja de seguir al competidor.

[POST] “/users/log”

Espera: token y LogDatesDTO.

Respuesta: muestra log entre fecha desde y hasta.

Sport Controller

[POST] “/sports”

Espera: SportModelIn y token.

Ej.:

```
{
  "Name": "Tennis"
}
```

Respuesta: Si tiene los permisos, te devuelve el deporte creado.

[PUT] “/sports/{{sportId}}”

Espera: SportModelIn y token.

Ej.:

```
{
  "Name": "Soccer"
}
```

Respuesta: Si tiene los permisos, te devuelve el deporte modificado.

[DELETE] “/sports/{{sportId}}”

Espera: -

Respuesta: Si tiene los permisos, borra el deporte de la base de datos.

[GET] “/sports”

Espera: token

Respuesta: Lista de deportes.

[GET] “/sports/{{sportId}}”

Espera: token

Respuesta: deporte correspondiente a ese Id.

[POST] “/sports/{{createdCompetitorId}}/competitors”

Espera: CompetitorModelIn y token.

Ej.:

```
{
  "Name": "team1",
  "ImagePath": "(path)"
}
```

Respuesta: Si tiene los permisos, te devuelve el competidor creado.

[GET] “/sports/{{sportId}}/competitors”

Espera: token

Respuesta: los equipos correspondientes a ese Id.

[GET] “/sports/{{sportId}}/ranking”

Espera: token

Respuesta: ranking correspondiente al deporte con ese id.

Competitor Controller

[PUT] “/competitors/{{competitorId}}”

Espera: CompetitorModelIn y token.

Ej.:

```
{
  "Name": "newCompetitor",
  "ImagePath": "(path)"
}
```

Respuesta: Si tiene los permisos, te devuelve el competidor modificado.

[DELETE] “/competitors/{{competitorId}}”

Espera: token

Respuesta: Si tiene los permisos, borra al competidor de la base de datos.

[GET] “/competitors”

Espera: token y posibilidad de filtrar por orden (“asc” o “desc”) y/o el nombre.

Respuesta: Lista de competidores.

[GET] “/competitors/{{createdCompetitorId}}”

Espera: token

Respuesta: competidor correspondiente a ese Id.

Matches Controller

[POST] “/matches”

Espera: MatchModelIn y token.

Ej.:

```
{
  "SportId": "{{createdSportId}}",
  "VisitorId": "{{createdCompetitorId}}",
  "LocalId": "{{otherCompetitorId}}",
  "Date": "01/06/2020 13:14"
}
```

Respuesta: Si tiene los permisos, te devuelve el partido creado.

[PUT] “/matches/{{matchId}}”

Espera: MatchModelIn y token.

Ej.:

```
{
  "SportId": "{{createdSportId}}",
  "Date": "01/06/2021 13:14"
}
```

Respuesta: Si tiene los permisos, te devuelve el partido modificado.

[DELETE] “/matches /{{ matchId }}”

Espera: token

Respuesta: Si tiene los permisos, borra el partido de la base de datos.

[GET] “/matches”

Espera: token

Respuesta: Lista de partidos.

[GET] “/matches /{{createdMatchId}}”

Espera: token

Respuesta: partido correspondiente a ese Id.

[GET] “/matches /{{createdMatchId}}/comments”

Espera: token

Respuesta: todos los comentarios correspondientes a ese Id.

[POST] “/matches/{{createdMatchId}}/comments”

Espera: CommentModelIn y token.

Ej.:

```
{
  "Text": "NewText"
}
```


Respuesta: Si tiene los permisos, te devuelve el comentario agregado.

[POST] “/matches/fixturesImplementations”

Espera: FixturesPathDto y token.

Ej.:

```
{
  "Path": "*path*"
}
```

Respuesta: Si tiene los permisos, se agregan las nuevas implementaciones de fixtures.

[POST] “/matches/generateFixture”

Espera: FixtureDto y token.

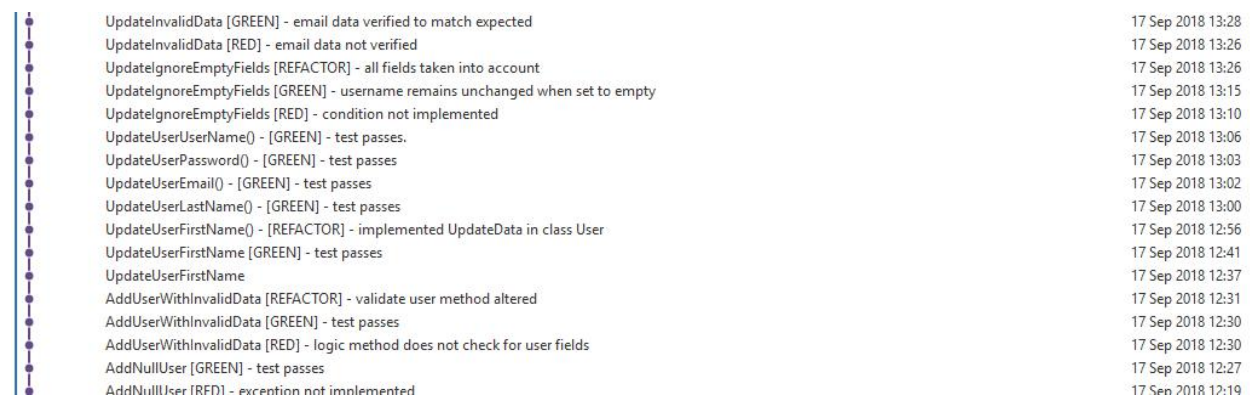
Respuesta: Si tiene los permisos, se genera el fixture.

[PUT] “/matches/nextFixture”

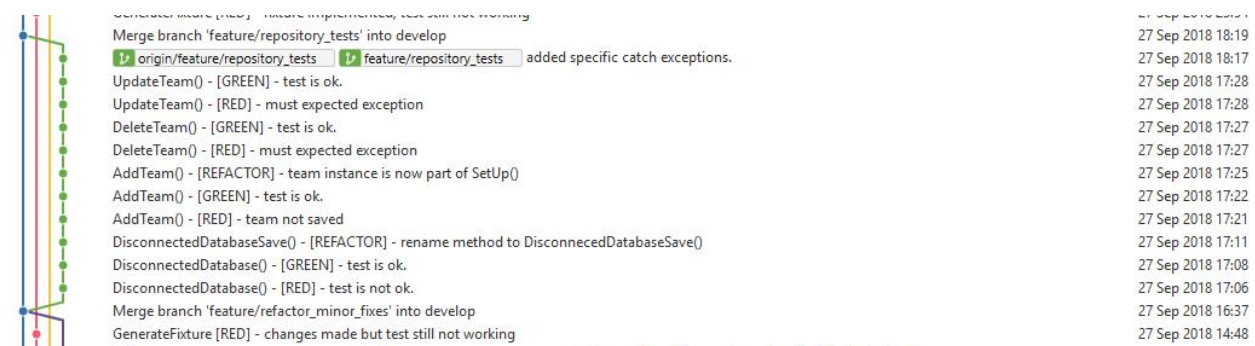
Espera: token.

Respuesta: Si tiene los permisos, cambia la implementación activa de algoritmo de fixture y retorna información sobre lo que hace el algoritmo

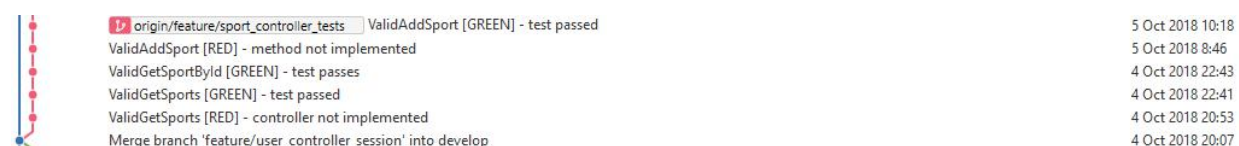
Commits que evidencian el uso de TDD en la lógica:



Commits que evidencian el uso de TDD en el repositorio:



Commits que evidencian el uso de TDD en la Web API:



A continuación, se deja evidencia de todos los datos de prueba de las diferentes tablas.
Hay solo un usuario administrador ingresado y el mismo tiene las siguientes credenciales.
Credenciales: Username: user; Password: pass.

Datos de prueba

Tabla: Users

	Id	FirstName	IsAdmin	LastName	Email	Password	UserName
1	1	Rafael	1	Alonso	pepe@pepe.com	pass	user
2	2	itai	0	miller	itai@hotmail.com	1234	itai

Tabla: Competitors

	Id	Name	Picture	SportId
1	4	roger	data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD...	3
2	5	rafa	data:image/png;base64,iVBORw0KGgoAAAANSUUEUgAAAL...	3
3	6	Nene		3
4	7	Lolo		3

Tabla: CompetitorScore

	Id	CompetitorId	Score	MatchId
1	1	4	0	1
2	2	5	0	1
3	3	5	0	NULL
4	4	4	0	NULL
5	5	5	3	2
6	6	4	2	2
7	7	4	0	3
8	8	5	0	3
9	9	4	0	4
10	10	6	0	4
11	11	4	0	5
12	12	7	0	5
13	13	5	0	6
14	14	6	0	6
15	15	5	0	7
16	16	7	0	7
17	17	6	0	8
18	18	7	0	8

Tabla: Matches

	Id	Date	SportId
1	1	2018-11-20 00:12:00.0000000	3
2	2	2018-11-30 00:00:00.0000000	3
3	3	2018-12-22 00:00:00.0000000	3
4	4	2018-12-23 00:00:00.0000000	3
5	5	2018-12-29 00:00:00.0000000	3
6	6	2018-12-30 00:00:00.0000000	3
7	7	2019-01-05 00:00:00.0000000	3
8	8	2019-01-06 00:00:00.0000000	3

Tabla: Sports

	Id	Name	Amount
1	2	tennis	2
2	3	tenis	2
3	4	pablo	2
4	5	Soccer	2

Tabla: Logins

	UserId	Token
1	1	358FF6E1-1921-4AB9-8478-7EB021E0985A

Tabla: Comments

	Id	Text	Date	UserId	MatchId
1	1	nuevo comment	2018-11-22 13:21:47.6948204	1	2

Tabla: Favorites

	Id	UserId	CompetitorId
1	9	1	5
2	10	1	4

