

**Universidad ORT Uruguay**  
**Facultad de Ingeniería**

**Obligatorio 2**

Entregado como requisito para la materia Diseño de Aplicaciones 1

[Enlace a Repositorio GitHub](https://github.com/IngSoft-DA1/265403_243045_174039)

[https://github.com/IngSoft-DA1/265403\\_243045\\_174039](https://github.com/IngSoft-DA1/265403_243045_174039)

Florencia Brum – 243045  
Nicolas Rodríguez – 265403  
Rodolfo Presno – 174039

Docentes: Santiago Pérez, Pablo Benítez

20/11/2024

# Índice

Índice .....	2
Declaración de Autoría .....	3
Instalación .....	4
Descripción general del trabajo .....	5
Descripción y justificación del diseño.....	6
Diagrama de paquetes y asignación de responsabilidades .....	6
Responsabilidades DTOs .....	6
Responsabilidades Dominio .....	7
Responsabilidades Services.....	7
Responsabilidades Interfaces .....	8
Diagramas de interacción .....	8
Diagrama Reporte de esfuerzo para una épica. ....	8
Diagramas de clases.....	9
Diagrama completo del proyecto.....	9
Modelo de tablas de la base de datos .....	10
Mecanismos generales y decisiones de diseño .....	11
Criterios seguidos para asignar las responsabilidades.....	12
Análisis de dependencias .....	13
Cobertura de pruebas unitarias.....	14
Bibliografía .....	15
Anexos .....	16
Anexo 1 – Diagramas de Interacción.....	16
Anexo 2 – Diagramas de clases .....	17
Anexo 3 – Cobertura de pruebas unitarias.....	20
Anexo 4 – Instalaciones .....	22

## Declaración de Autoría

Nosotros, Florencia Brum, Nicolas Rodríguez y Rodolfo Presno, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizamos el Obligatorio parte 1 y parte 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado juntamente con otros, hemos explicado claramente qué fue construido por otros, y qué fue construido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

# Instalación

Nuestra aplicación corre sobre .NET Core 6 y Blazor, es necesario SQL Server para implementar la base de datos.

## Ubicación de los ejecutables:

Dentro de la carpeta del release, se encuentra el archivo "InterfazWeb.exe". Para correr la aplicación hay que ejecutarlo habiendo iniciado la implementación de la base de datos con Docker y DBeaver (Anexo 4 - Instalaciones).

-----

## Librerías requeridas:

1. Entity FrameworkCore versión 7.0.9
2. Entity FrameworkCore Design versión 7.0.9
3. Entity FrameworkCore InMemory versión 7.0.9
4. Entity Framework Core SQL Server versión 7.0.9
5. Syncfusion Blazor versión 19.3.0.43
6. EPPlus versión 7.5.0

## Strings de conexión a modificar:

El connection string a modificar se encuentra en el proyecto InterfazWeb en el archivo appsettings.json.

```
"ConnectionStrings": {  
  "DefaultConnection" : "Server=127.0.0.1;Database=TaskPanel;User  
Id=sa;Password=Passw1rd;TrustServerCertificate=true;"  
}
```

## Ubicación de los scripts de la base de datos:

-----

## Usuario inicial de la aplicación:

Usuario: admin@taskpanel.com

Password: Admin123\$

## Descripción general del trabajo

Para esta segunda parte del obligatorio, partimos con la base entregada en la primera parte.

Uno de los grandes desafíos para esta entrega, consiste en la implementación de la persistencia de los datos en una base de datos. En este caso utilizamos como sistema de gestión de base de datos a SQL Server.

Al igual a como lo hicimos con la primera entrega, utilizamos el enfoque de programación dirigida por pruebas (TDD Test-Driven-Development) el cual nos permitió asegurar que cada funcionalidad que se implementó estaba cubierta por pruebas unitarias desde el principio ayudando a prevenir errores, ya que el código se validaba continuamente lo que nos permitió una buena integridad en el sistema.

En cuanto al mantenimiento de la calidad del código, continuamos aplicando la filosofía de Clean Code, para asegurarnos que nuestro código fuera lo más legible y comprensible posible, utilizando nombres claros y descriptivos para las variables y métodos, evitando las funciones grandes y asegurando que cada clase tuviera una única responsabilidad.

Para esta entrega, implementamos la persistencia de todos los datos mediante el uso del ORM .NET Entity Framework Core en conjunto con SQL Server. Este ORM nos simplificó el proceso de trabajar con la base de datos, nos permitiendo mapear las clases de nuestra aplicación en tablas de SQL Server y la inserción de los datos en las mismas sin la necesidad de escribir SQL Manual.

Además, utilizamos Docker como herramienta para virtualizar el servidor de la base de datos, lo que nos proporcionó una solución ágil y flexible donde nos permitió que cada uno de los integrantes tuviéramos un entorno de desarrollo idéntico evitando conflictos entre nuestros entornos.

Para poder visualizar nuestra base de datos utilizamos el programa “DBeaver”, el cual nos permitió explorar tablas y verificar que la persistencia de datos se estaba realizando correctamente como también encontrar errores a la hora de guardar datos.

A lo largo del desarrollo también aplicamos principios SOLID como patrones de diseño de GRASP, promoviendo el bajo acoplamiento y alta cohesión entre las clases.

En cuanto a los requerimientos logramos cumplir con todos los objetivos solicitado de manera satisfactoria, la persistencia de datos fue implementada correctamente y las funcionalidades fueron validadas en varios escenarios.

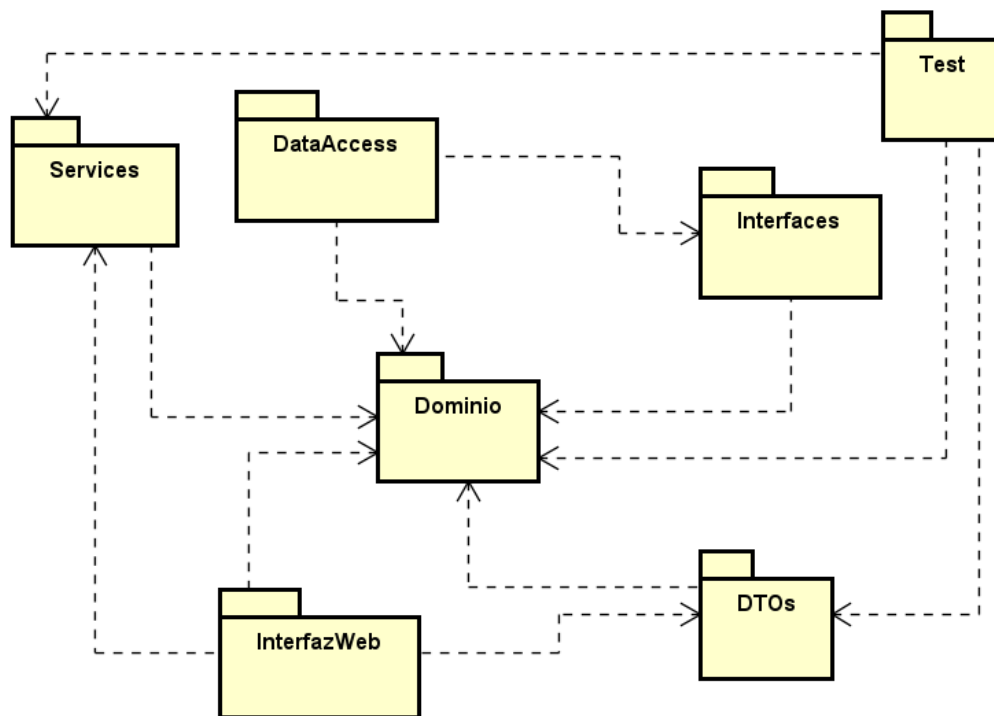
Actualmente, tenemos un bug conocido en nuestra aplicación: si el usuario refresca la página, se ve obligado a volver a iniciar sesión. Este problema radica en la implementación de la clase **Session**, que es responsable de gestionar el inicio de sesión.

Otro bug que ocurre es que cuando se crea un gráfico por primera vez, aparece un error de licencia. Esto se debe a que no compramos la licencia del paquete que utilizamos para los gráficos, Syncfusion.

# Descripción y justificación del diseño

## Diagrama de paquetes y asignación de responsabilidades

En la entrega anterior, teníamos un paquete denominado dominio que incluía varias subcarpetas. Para esta entrega, decidimos reorganizar la estructura de dicho paquete y separar las subcarpetas en paquetes independientes. Esto nos permitió distribuir los componentes de manera más concisa y centralizar mejor las responsabilidades e información de cada paquete, lo cual facilita la mantenibilidad del código. Buscamos aumentar la coherencia en cada paquete.



## Responsabilidades DTOs

Mantuvimos los DTOs implementados para la primera entrega.

Namespace DTOs		
Namespace	Clase	Responsabilidad
DTOs.UserDTOs	UserLoginDTO	Implementa DTO para los datos del login y sus requerimientos.
DTOs.UserDTOs	UserModifyDTO	Implementa DTO para la modificación de datos de usuario.
DTOs.UserDTOs	UserRegisterDTO	Implementa DTO para el registro de nuevos usuarios y sus requerimientos.

## Responsabilidades Dominio

Almacenamos las entidades básicas (previamente Dominio.Models) junto con la clase abstracta IDeletable y Trash para almacenar eliminados. Además, agregamos las clases Epic y Notification para cumplir los nuevos requerimientos.

Namespace Dominio		
Namespace	Clase	Responsabilidad
Dominio	Comment	Almacenar datos de un comentario.
Dominio	Epic	Almacenar datos de una épica.
Dominio	IDeletable	Abstracta, almacena datos básicos de un elemento eliminable. Es implementada por la clase Panel y la clase Task
Dominio	Notification	Almacenar datos de una notificación.
Dominio	Panel	Almacenar datos de un panel y sus tareas.
Dominio	Task	Almacenar datos de una tarea y sus comentarios, y métodos para operar sobre los comentarios.
Dominio	Team	Almacenar datos de equipos, sus usuarios y paneles. Validación de equipos.
Dominio	Trash	Almacenar elementos eliminados por un mismo usuario.
Dominio	User	Almacenar datos de un usuario, y métodos de validación.

## Responsabilidades Services

En el paquete Services almacenamos los métodos de la lógica de negocio. Agregamos aquí los de épicas y notificaciones, además de mover TaskImport por considerar que su responsabilidad de ajusta más a la de un servicio. Agregamos el adaptador xlsx a csv.

Namespace Services		
Namespace	Clase	Responsabilidad
Services	EpicService	Almacenar métodos para obtener y modificar épicas y sus vínculos con paneles y tareas.
Services	NotificationService	Almacenar métodos para el manejo de notificaciones.
Services	PanelService	Almacenar métodos para obtener y modificar vínculos entre equipos y paneles.
Services	PasswordService	Almacenar métodos para validar y generar contraseñas de usuarios.
Services	TaskImportService	Almacenar la lógica para transformar texto con tareas en csv a Tasks.
Services	TaskService	Almacenar métodos para obtener y modificar vínculos entre paneles y tareas.
Services	TeamService	Almacenar métodos para obtener y modificar equipos, sus usuarios y modificar sus vínculos.
Services	UserService	Almacenar métodos auxiliares de usuarios, vínculos con elementos de papelería.
Services	XlsxToCsvAdapter	Extender métodos de TaskImportService para poder procesar archivos .xlsx.

## Responsabilidades Interfaces

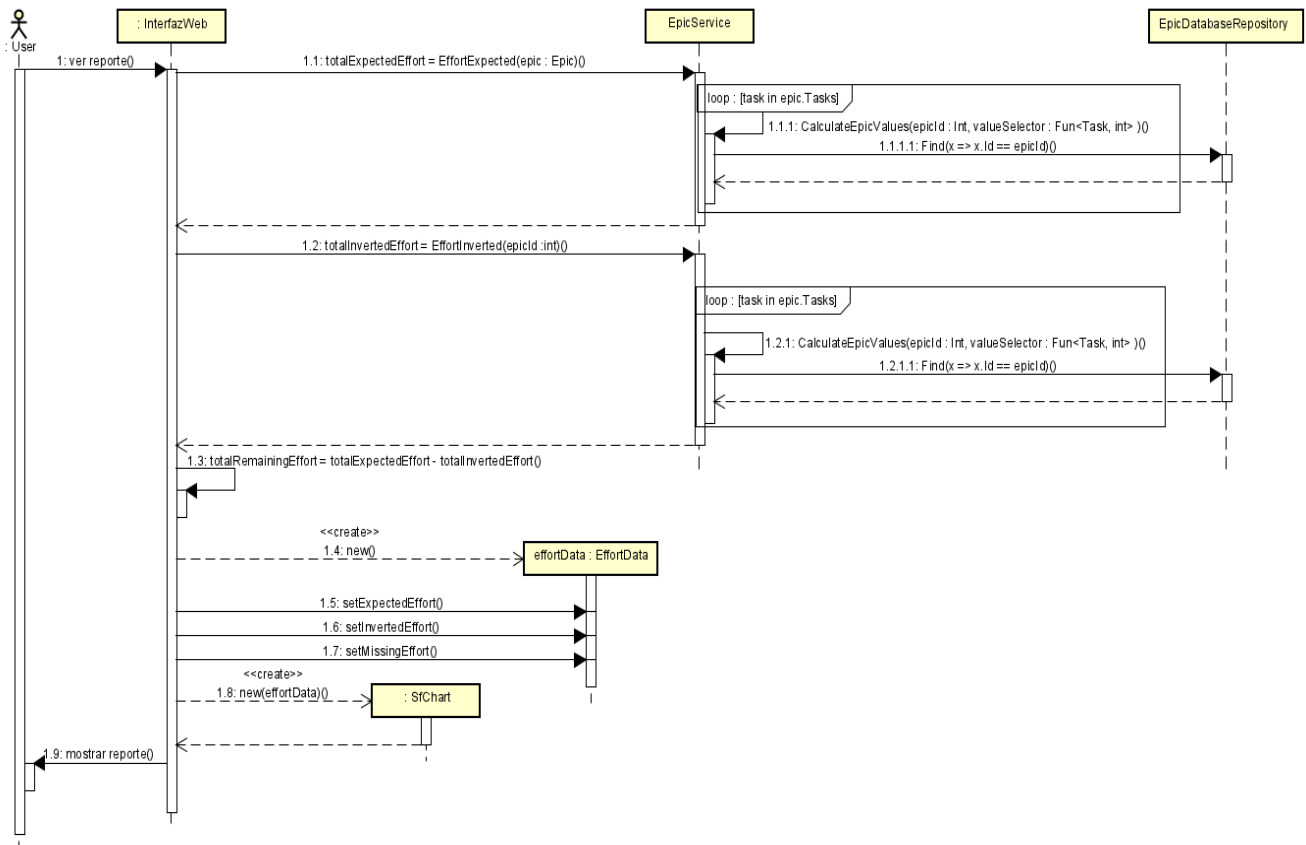
El paquete Interfaces centraliza las interfaces declaradas. Quitamos IPaperBinItem para cambiarlo por una clase abstracta Deleteable y las interfaces que operaban con la implementación anterior de las bases de datos en memoria. También agregamos interfaces para los servicios de épicas, notificaciones, tasks y repositorios.

Namespace Interfaces		
Namespace	Clase	Responsabilidad
Interfaces	IEpicService	Interfaz para EpicService
Interfaces	INotificationService	Interfaz para NotificationService
Interfaces	IPanelService	Interfaz para PanelService
Interfaces	IRepository	Interfaz genérica para los repositorios
Interfaces	ITaskService	Interfaz para TaskService
Interfaces	ITeamService	Interfaz para TeamService
Interfaces	IUserService	Interfaz para UserService

## Diagramas de interacción

En esta sección se encuentra el diagrama de “reporte de esfuerzo de una épica”. El resto de los diagramas puede ser encontrado en el anexo 1

### Diagrama Reporte de esfuerzo para una épica.



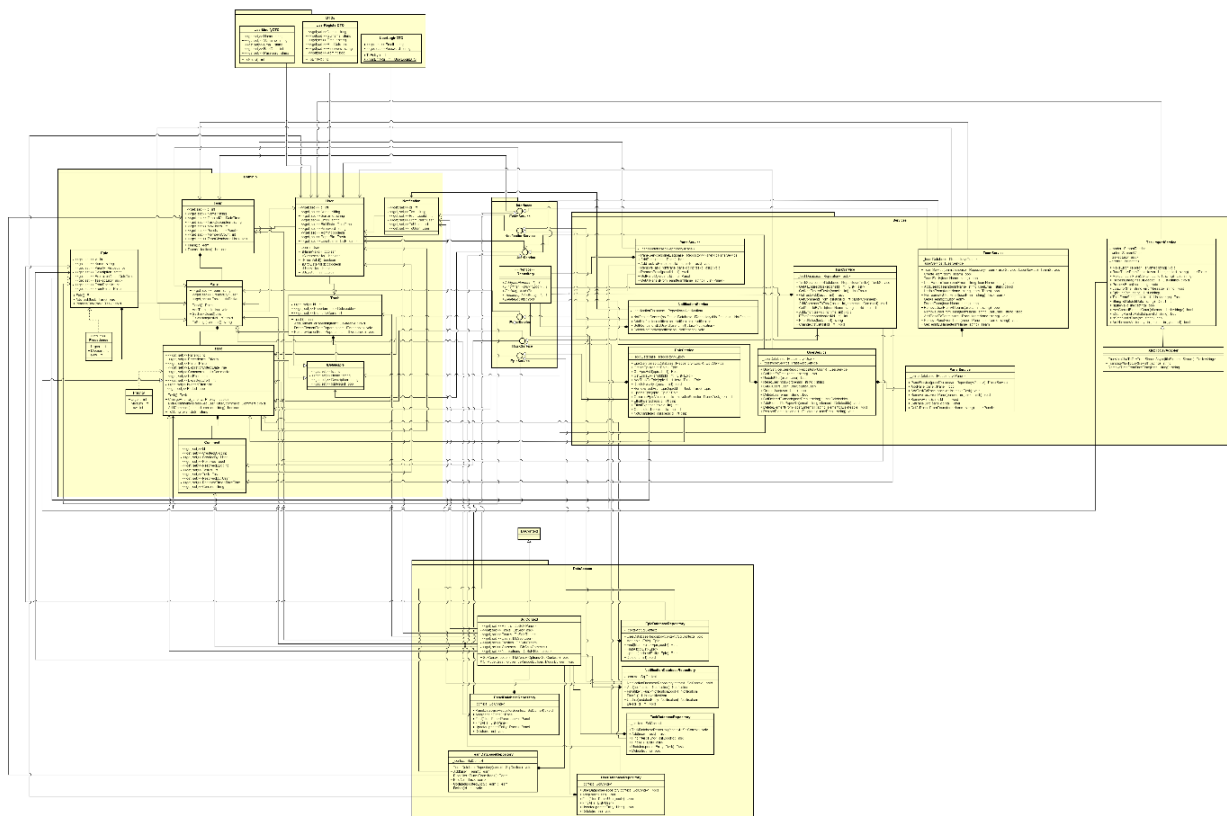


## Diagramas de clases

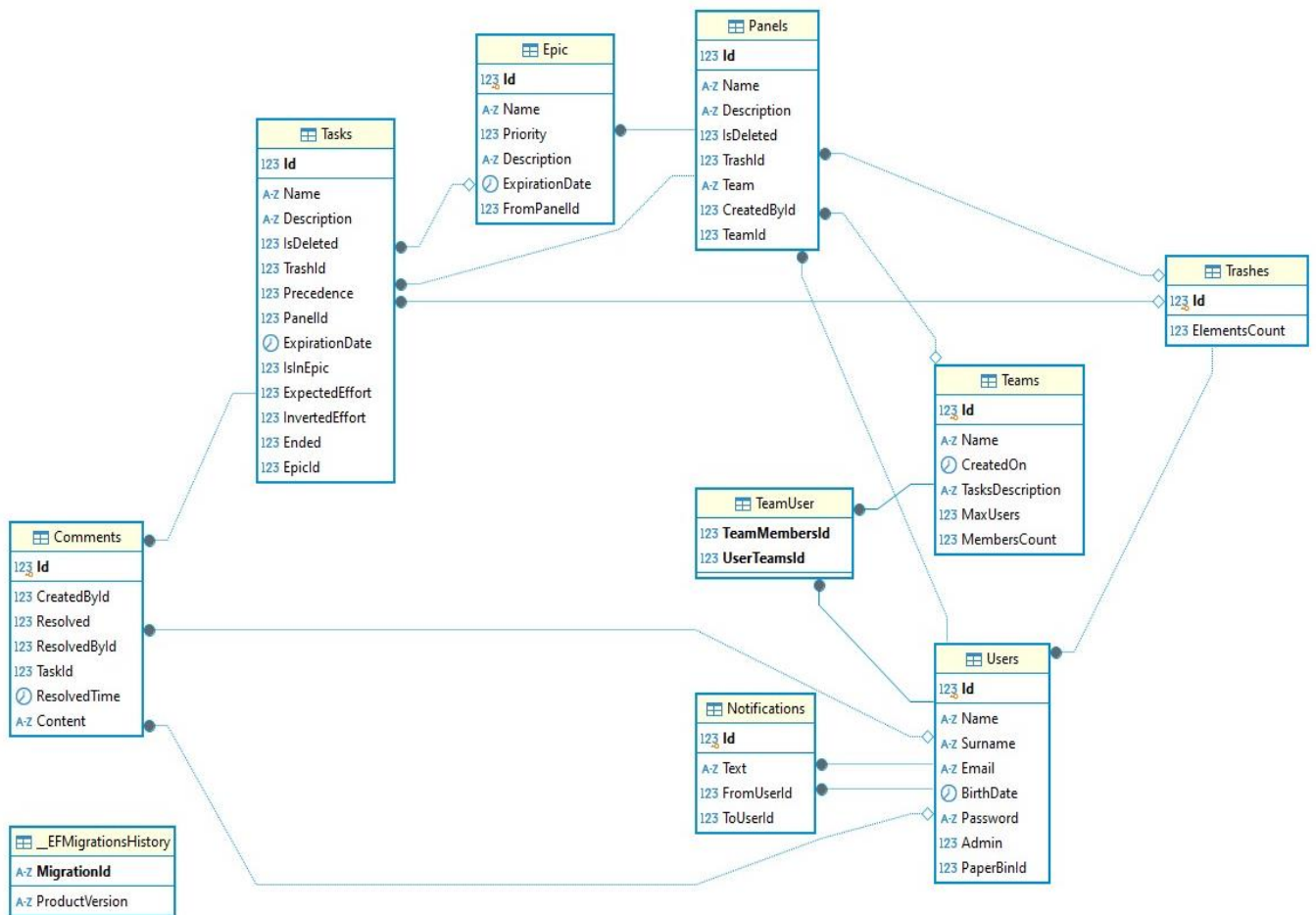
El diagrama de clase que decidimos incluir en esta sección es el diagrama de las clases de todos los paquetes del proyecto. El resto de los diagramas puede ser encontrado en el Anexo 2.

Para mantener la claridad, nos tomamos la libertad de simplificar los diagramas individuales omitiendo las conexiones internas entre las clases de los paquetes secundarios. Por ejemplo, en el diagrama del paquete Service se incluye el paquete Dominio porque es utilizado por Service, pero no se muestran las conexiones internas propias del paquete Dominio, ya que no es el foco principal del diagrama.

## Diagrama completo del proyecto



## Modelo de tablas de la base de datos



## Mecanismos generales y decisiones de diseño

En cuanto a las decisiones de diseño, para esta entrega nos resultó más fácil ya que desde la primera entrega comenzamos a aplicar principios de diseño SOLID.

Entre ellos podemos destacar el principio de Abierto-Cerrado. Consideramos que nuestro código respeta perfectamente este principio y se pueden añadir nuevas funcionalidades sin tocar el código ya existente.

En nuestro diseño, implementamos interfaces y clases abstractas que nos permiten extender la funcionalidad sin cambiar las implementaciones originales. Por ejemplo; hemos creado una clase abstracta “Deleteable” que nos permite definir objetos pueden ser eliminados y enviados a la papelera. Esta clase es implementada por las clases Task y Panel, haciendo uso de Polimorfismo. También usamos polimorfismo para la importación desde archivos.xlsx: el servicio para procesar archivos.xlsx extiende al de archivos.csv.

Cada clase realiza únicamente su tarea específica sin depender de otras clases más allá de lo necesario, por ejemplo, cada servicio no tiene conocimiento sobre el almacenamiento o la persistencia de datos. Esto nos lleva a hablar sobre el bajo acoplamiento de nuestro diseño, ya que utilizamos interfaces para poder desacoplar la lógica de negocio de sobre el acceso a datos, así como también cada clase en el sistema tiene una alta cohesión, ya que cada una se encarga de una única responsabilidad. Por ejemplo, el servicio TaskService se encarga exclusivamente de la lógica de negocio relacionada con las tareas, sin mezclar responsabilidades con otras áreas del sistema. De igual manera, la clase TaskDatabaseRepository está encargada solo de la persistencia de tareas.

Utilizamos interfaces para poder cumplir con el principio de segregación de interfaces, creamos una interfaz para cada servicio para poder utilizarlo luego en la interfaz web. De esta manera definimos un contrato para cada servicio y nos aseguramos de que, si en el día de mañana alguna regla del Task Panel cambia, nos ayude a tener que cambiar el menor código posible.

Se hace uso del Principio de Inversión de Dependencias, esto lo logramos creando un proyecto para las interfaces, y luego la clase que implementa se encuentra en un proyecto, y la clase que utiliza esa interfaz se encuentra en otro proyecto. De esta manera nos aseguramos de que cada proyecto genere su propio DLL.

Continuamos utilizando el modelo de diseño Arquitectura en Capas, donde separamos nuestro programa en 4 capas. Interfaz Web, Servicios, Dominio, y Repositorio, como mencionamos en el entregable anterior, esta separación en capas nos permite asegurar que cada capa se desarrolle y mantenga de forma independiente. Esto nos permite agregar o intercambiar funcionalidades sin afectar otras partes del sistema.

## Criterios seguidos para asignar las responsabilidades

Como mencionamos anteriormente, para este obligatorio utilizamos como base el proyecto anterior. Dicho proyecto ya tenía bien definido el criterio de responsabilidad, estructurando el código en cuatro paquetes principales: uno para los tests, otro para la interfaz web, otro para los DTOs, y un cuarto que gestionaba la parte lógica.

En este nuevo obligatorio, decidimos refinar la organización del paquete lógico separándolo en distintos paquetes más específicos. Esto se hizo con el objetivo de reducir las responsabilidades de cada componente y centralizar mejor las funcionalidades.

Dicha separación terminó resultando en los siguientes paquetes:

- **Dominio:**
  - Responsabilidad: Representar y modelar las entidades principales del sistema y sus atributos.
  - Propósito: Principalmente se utiliza para almacenar información de cada objeto correspondiente.
- **DTOs:**
  - Responsabilidad: Modelar los *Data Transfer Objects* (DTOs) empleados en la transferencia de datos entre capas o componentes.
- **Services:**
  - Responsabilidad: Implementar la lógica de negocio relacionada con cada entidad.
  - Propósito:
    - Manejar operaciones como buscar, crear y eliminar entidades.
    - Actuar como intermediarios entre las entidades y sus repositorios.
    - Incluir servicios específicos como la importación de datos.
- **Interfaces:**
  - Responsabilidad: Definir las interfaces que son empleadas en las distintas capas del sistema.
  - Propósito: Definir el qué hace un componente, sin preocuparse por cómo lo hace. Permite que el código que depende de la interfaz no necesite saber los detalles específicos del servicio que la implementa
- **DataAccess**
  - Responsabilidad: Implementa las clases repositorio.
  - Propósito: Es el encargado de conectar el dominio con la base de datos.
- **Test**
  - Responsabilidad: Implementa los test necesarios para las funciones de cada paquete del programa.

## Análisis de dependencias

Para el reporte de esfuerzo de una épica, se implementaron métodos en la clase **EpicService** que optimizan los cálculos necesarios. La solución incluye un método "principal" y cuatro métodos adicionales que lo utilizan:

1. **Método principal:** se encarga de realizar los cálculos principales, devolviendo un int. Este método recorre las tareas asociadas a la épica y suma los valores según una condición específica pasada como parámetro.
2. **Cuatro métodos derivados:** cada uno llama al método principal para calcular métricas específicas:
  - Esfuerzo esperado de la épica.
  - Esfuerzo invertido en la épica.
  - Cantidad de tareas terminadas.
  - Cantidad de tareas no terminadas.

Esta implementación permite evitar la redundancia de código, ya que, de otra manera, se habrían creado cuatro métodos con lógica prácticamente idéntica.

Las clases de dominio involucradas son **Epic** y **Task**. Epic proporciona la lista de tareas asociadas; y Task es iterada para extraer los datos necesarios para los cálculos.

Los métodos que se encuentran en EpicService tienen alta **cohesión**, ya que cada uno cumple un propósito específico, los cálculos son simples y no combinan responsabilidades, favoreciendo la mantenibilidad.

El acoplamiento entre EpicService y Task es **moderado** ya que desde EpicService se necesita iterar sobre las tareas para calcular las métricas antes mencionadas.

El acoplamiento entre EpicService y Epic es **moderado** ya que es el servicio que maneja la lógica de negocio de Epic.

Para la optimización de código, contamos con la ayuda de chatgpt.

# Cobertura de pruebas unitarias

Al analizar la cobertura de pruebas unitarias (habiendo excluido interfaz gráfica, que no se testea) obtenemos el siguiente resultado inicial: 82%

Symbol	Coverag... ^	Uncovered/Total Stmts.
✓ Total	82%	516/2888
> DataAccess	29%	482/682
> Services	96%	22/581
> Test	99%	12/1354
> Dominio	100%	0/233
> DTOs	100%	0/38

Este es un número que no es nada deseable, y la razón de la baja cobertura es el 29% de DataAccess. Analizando en detalle se puede ver que este es un resultado engañoso, ya que se está teniendo en cuenta la migración empleada para el Entity Framework:

✓ DataAccess	29%	482/682
✓ {} DataAccess	29%	482/682
> {} Migrations	0%	477/477
> DbContext	84%	5/32
> EpicDatabaseRepository	100%	0/26
> NotificationDatabaseRepository	100%	0/23
> PanelDatabaseRepository	100%	0/31
> TaskDatabaseRepository	100%	0/28
> TeamDatabaseRepository	100%	0/33
> UserDatabaseRepository	100%	0/32

Una vez excluida la migración se comprueba que la cobertura es de 98%

Symbol	Coverag... ^	Uncovered/Total Stmts.
✓ Total	98%	39/2411
> Services	96%	22/581
> DataAccess	98%	5/205
> Test	99%	12/1354
> Dominio	100%	0/233
> DTOs	100%	0/38

En los anexos se encuentran las capturas de pantalla de la cobertura para cada uno de los paquetes individuales.

En la captura de la cobertura de Services se puede notar que 16 de 22 líneas no están cubiertas. Estas líneas son del método *TranslateXlsxToCsvFromStreamAsync(Stream fileStream)*. La razón de no cubrirlo es que el parámetro del método es un Stream, y para testearlo habría que generarlo en la clase de prueba.

# Bibliografía

- Aguilar, J. M. (s.f.). *Creando gráficas estadísticas en Blazor con los componentes visuales de Syncfusion*. Obtenido de Variable Not Found:  
<https://www.variablenotfound.com/2021/06/creando-graficas-estadisticas-en-blazor.html>
- ChatGPT. (s.f.). *Blazor File Upload Parsing*. Obtenido de ChatGPT:  
<https://chatgpt.com/share/673d576f-ed10-800c-8623-813e88bfe83d>
- ChatGPT. (s.f.). *Optimización de código C#*. Obtenido de ChatGPT:  
<https://chatgpt.com/share/67386367-6ab0-8002-a6a3-aa3e58be00b3>
- ChatGPT. (s.f.). *Paquetes lectura archivos Excel*. Obtenido de ChatGPT:  
<https://chatgpt.com/share/673ce00a-df58-8002-8e29-8c758db11da7>
- freeCodeCamp.org. (s.f.). *Blazor Server App with .NET 6 and Syncfusion UI Components – Full Course*. Obtenido de Youtube:  
<https://www.youtube.com/watch?v=xO17P9LVkK0>
- Syncfusion. (s.f.). *Add a Blazor Stock Chart to a Blazor Server App in Just 10 Minutes*. Obtenido de Youtube:  
<https://www.youtube.com/watch?v=AxnqK2BnapM>
- Syncfusion. (s.f.). *Create a Blazor Server App and Add Syncfusion Blazor Components*. Obtenido de Youtube:  
<https://www.youtube.com/watch?v=cQZYzOITm0Q>
- Syncfusion. (s.f.). *Create Blazor Charts in Just 10 Minutes*. Obtenido de Youtube:  
<https://www.youtube.com/watch?v=0xwwOvmYQ6E>
- Syncfusion. (s.f.). *Creating A Chart In Blazor Server App*. Obtenido de Syncfusion:  
<https://www.syncfusion.com/code-examples/create-blazor-charts-in-a-blazor-server-app>

# Anexos

## Anexo 1 – Diagramas de Interacción

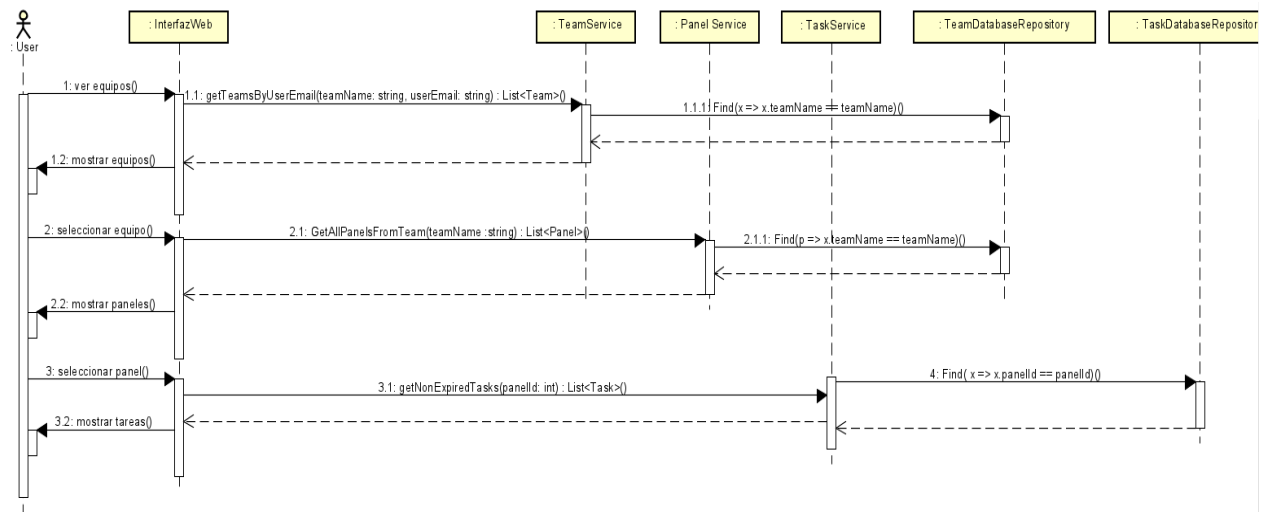


Diagrama de secuencia Mostrar Tareas

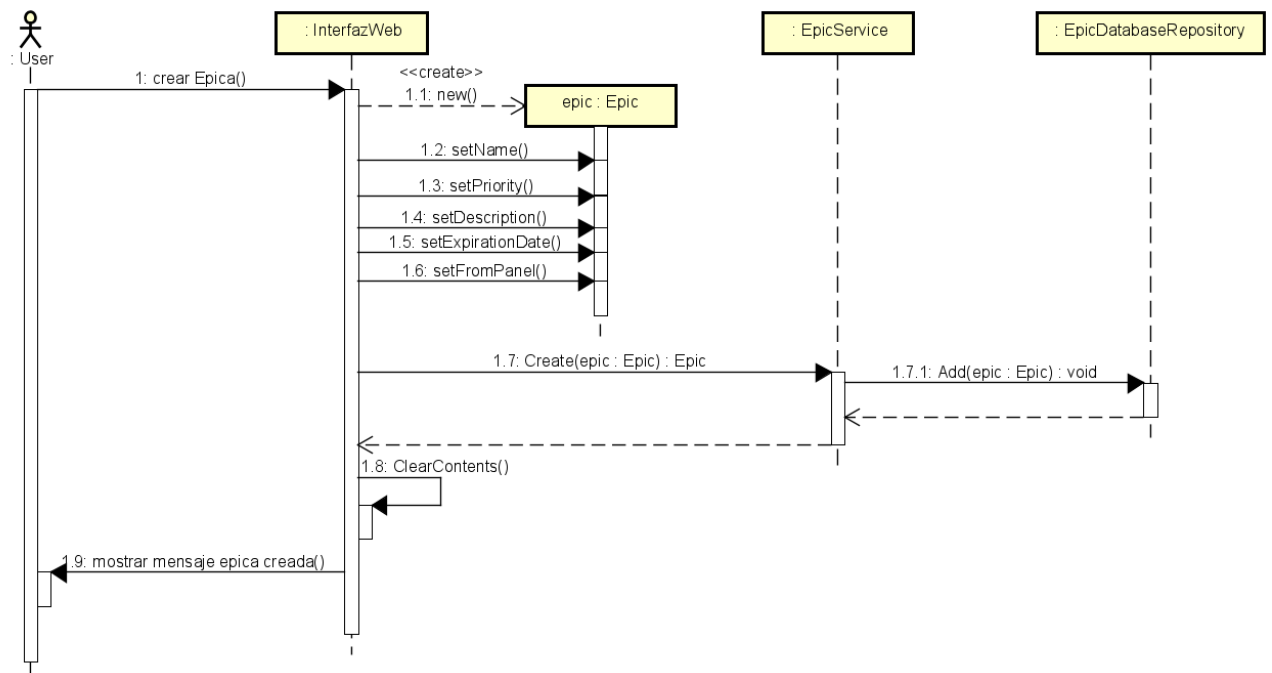
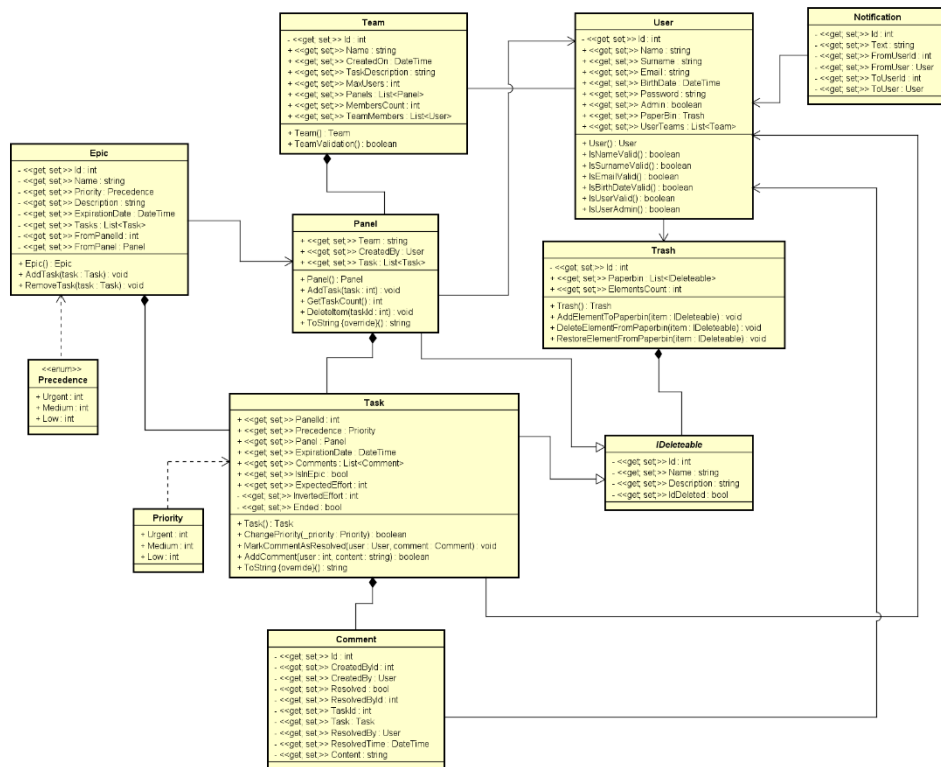


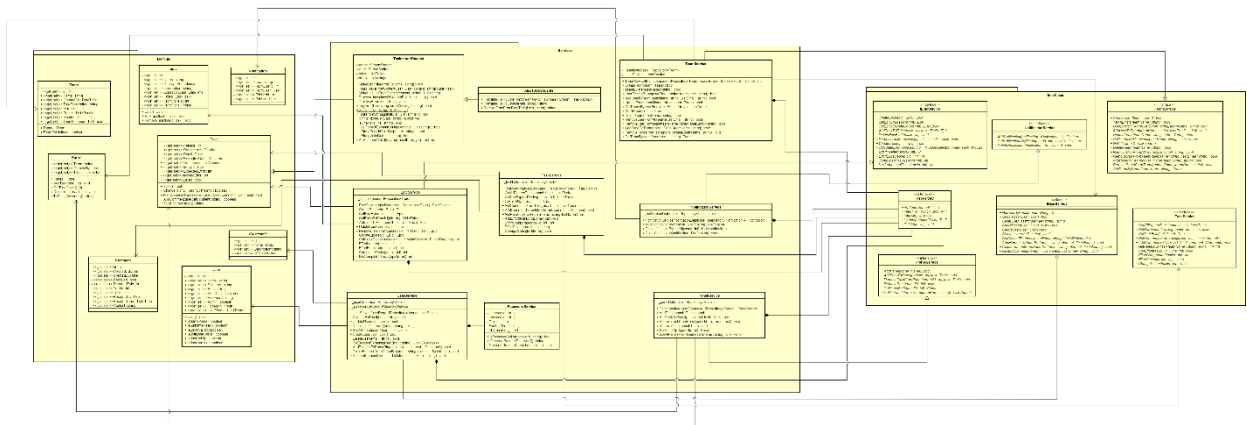
Diagrama Crear Épica



## Anexo 2 – Diagramas de clases



*Diagrama de clases del paquete Dominio*



*Diagrama de clases del paquete Service*

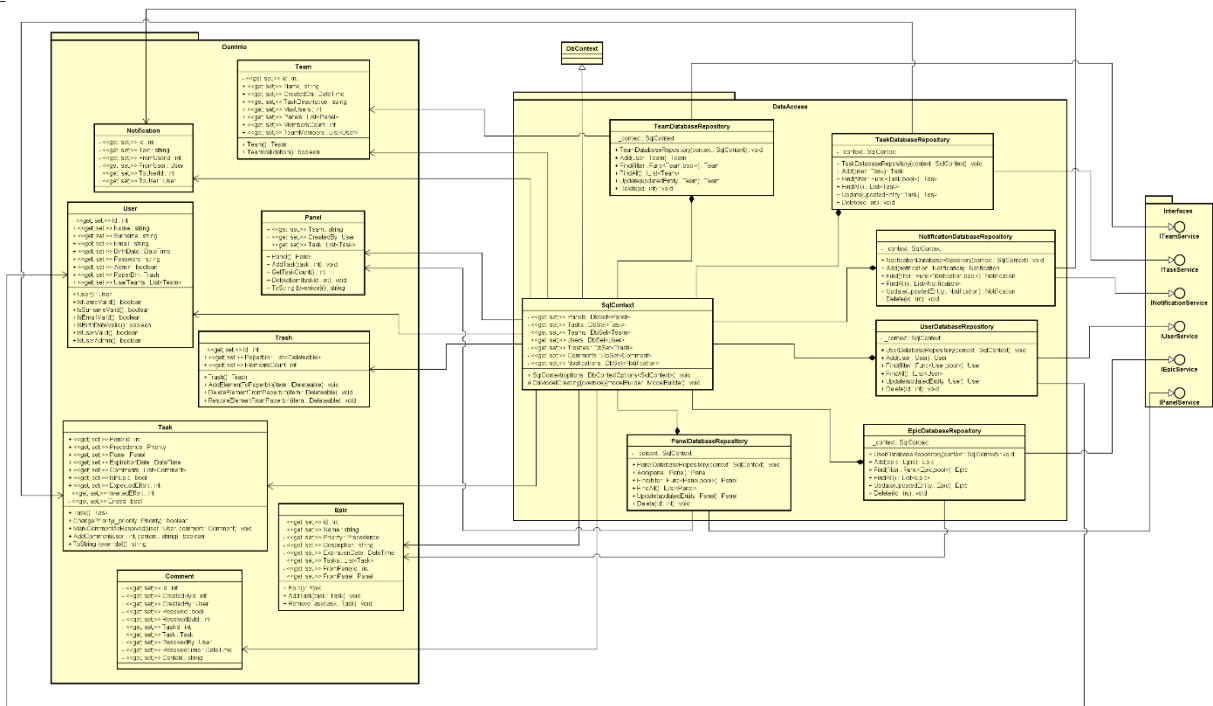


Diagrama de clases del paquete DataAccess

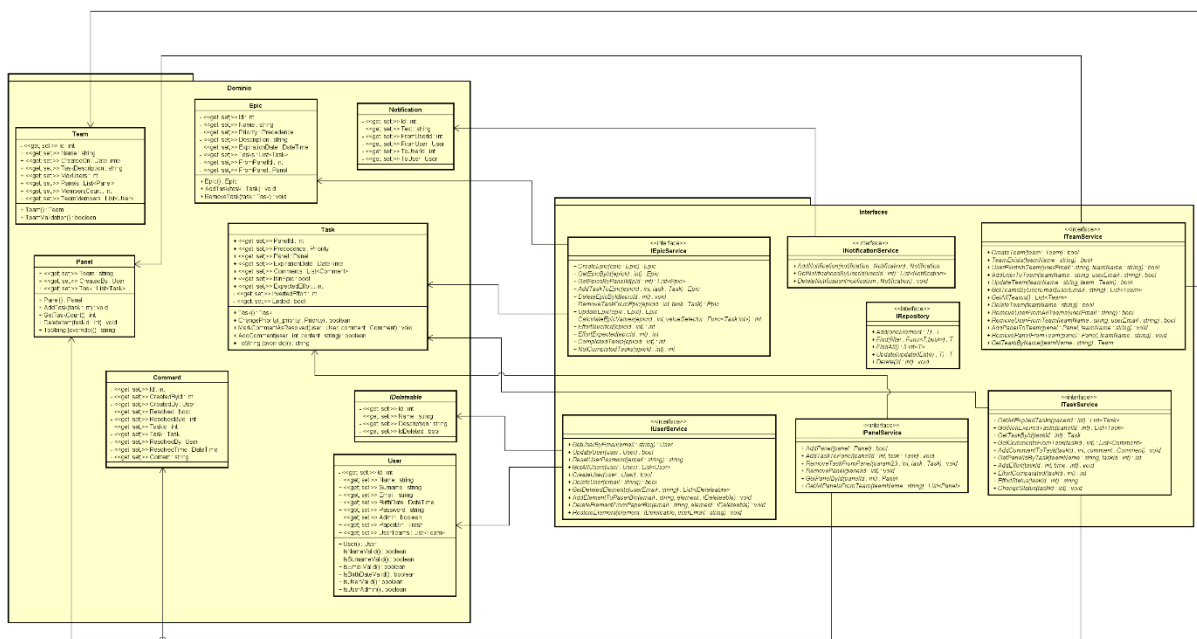
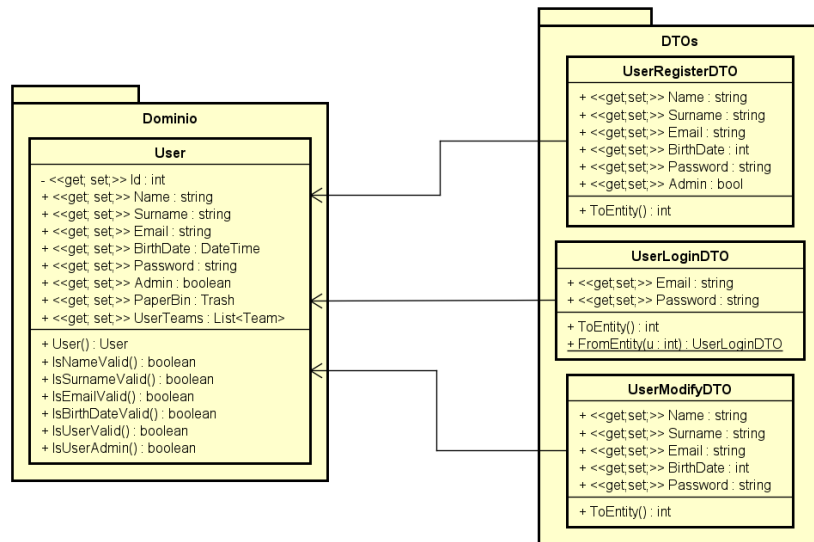










Diagrama de clases del paquete Interfaces



*Diagrama de clases del paquete Interfaces*





## Anexo 3 – Cobertura de pruebas unitarias

▼  DataAccess	98%	5/205
▼ {} DataAccess	98%	5/205
>  SqlContext	84%	5/32
>  EpicDatabaseRepository	100%	0/26
>  NotificationDatabaseRepository	100%	0/23
>  PanelDatabaseRepository	100%	0/31
>  TaskDatabaseRepository	100%	0/28
>  TeamDatabaseRepository	100%	0/33
>  UserDatabaseRepository	100%	0/32

*Cobertura de DataAccess*

▼  Dominio	100%	0/233
▼ {} Dominio	100%	0/233
>  Comment	100%	0/20
>  Epic	100%	0/26
>  IDeleteable	100%	0/8
>  Notification	100%	0/12
>  Panel	100%	0/30
>  Task	100%	0/35
>  Team	100%	0/31
>  Trash	100%	0/30
>  User	100%	0/41

*Cobertura de Dominio*

▼  DTOs	100%	0/38
▼ {} DTOs	100%	0/38
>  UserLoginDTO	100%	0/10
>  UserModifyDTO	100%	0/13
>  UserRegisterDTO	100%	0/15

*Cobertura de DTOs*

▼  Test	99%	12/1354
▼ {} Test	99%	12/1354
> {} ServicesTests	99%	8/724
> {} DataAccessTests	99%	4/321
> {} ModelsTests	100%	0/247
> {} DTOsTest	100%	0/57
> {} Context	100%	0/5

*Cobertura de Test*

Services	96%	22/581
▼ {} Services	96%	22/581
▼  XlsxToCsvAdapter	64%	16/44
>  TranslateXlsxToCsvFromStre	0%	16/16
TranslateXlsxToCsv(string)	100%	0/25
RemoveTimeFromDateTime(s	100%	0/3
>  UserService	96%	3/82
>  PanelService	97%	1/39
>  TeamService	98%	2/95
>  EpicService	100%	0/71
>  NotificationService	100%	0/15
>  PasswordService	100%	0/28
>  TaskImportService	100%	0/141
>  TaskService	100%	0/66

*Cobertura de Services*

## Anexo 4 – Instalaciones

Para preparar el ambiente de trabajo hay que setear el Docker y DBeaver:

**Docker:** Para setear Docker desde PowerShell usamos el comando `"docker pull mcr.microsoft.com/azure-sql-edge"` y luego `"docker run -e 'ACCEPT_EULA=1' -e 'MSSQL_SA_PASSWORD=Passw1rd' -p 1433:1433 --name azuresqledge -d mcr.microsoft.com/azure-sql-edge"`.

**DBeaver:** Luego cuando el contenedor esté funcionando se inicia una nueva conexión SQL Server desde el Dbeaver con el usuario "sa" y contraseña "Passw1rd". Una vez establecida la conexión configuramos el DLL del esquema con el archivo .SQL adjunto. Si se quisiera setear una base de datos con datos de prueba se podrían hacer los inserts en este paso.

Decidimos agregar un comando en los inserts de la base de datos para reiniciar los índices de algunas tablas. Esto se debe a que a veces el contador de las id no se reiniciaba entre iteración de prueba e iteración, entonces observábamos, por ejemplo, un usuario con ID = 15 aunque solo hubiera 5 usuarios.

El comando mencionado es:

```
DBCC CHECKIDENT ('<NombreTabla>', RESEED, 0);
```