

**Universidad ORT Uruguay**

**Facultad de ingeniería**



**Diseño de Aplicaciones 1**

**Obligatorio 1**

**M5A**

**Profesores: Guillermo Areosa, Bruno Balduccio**

**Renzo Donagrandi - 315078**

**Joaquin Kalichman - 257303**

**Juan Pedro Michelini - 286845**

**[https://github.com/IngSoft-DA1/315078\\_257303\\_286845.git](https://github.com/IngSoft-DA1/315078_257303_286845.git)**

# Índice

<b>Descripción general del trabajo y sistema.....</b>	<b>3</b>
Descripción general.....	3
Documentación de errores.....	3
Instructivo de instalación.....	4
<b>Justificación de Diseño.....</b>	<b>5</b>
Diagrama de paquetes.....	5
Diagrama de clases.....	10
Diagrama de interacción.....	13
Diagrama de secuencia para la funcionalidad Agregar Equipo:.....	13
Modelo de tablas.....	15
Descripción de las principales decisiones de diseño tomadas.....	16
Criterios seguidos para asignación de responsabilidades.....	17
<b>Análisis de dependencias.....</b>	<b>18</b>
<b>Informe de cobertura de los casos de prueba.....</b>	<b>18</b>
<b>Conclusión.....</b>	<b>19</b>
<b>Anexo.....</b>	<b>19</b>

# Descripción general del trabajo y sistema

## Descripción general

En este Obligatorio, se pidió que implementáremos nuevas funcionalidades de la aplicación “TaskPanel”, y corregir errores mediante la devolución de la primera entrega de esta.

En esta nueva instancia, se pidió que los datos del sistema sean guardados en una base de datos, dando paso a la posibilidad de que cada vez que se abra la aplicación, esta se ejecute con el último estado guardado antes de cerrarla. Esto significa que los datos se tienen que ir guardando en la base de datos mientras el usuario da uso a la solución.

Esta información, debe ser almacenada en una base de datos SQL Server, corriendo en un contenedor de Docker. El diseño debe contemplar el modelado de una solución de persistencia adecuada para el problema utilizando Entity Framework Core (Code First).

Todas estas funcionalidades se implementan correctamente. No obstante, durante el desarrollo del proyecto, surgieron algunos desafíos.

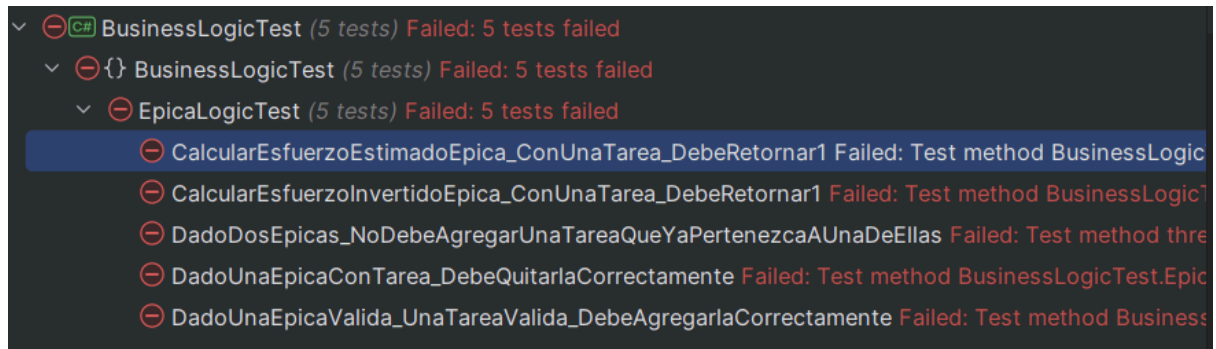
Uno de los cuales fue a la hora de implementar los cambios hechos en el back, hacia el front. Se nos rompían los botones de la interfaz que en la primera entrega andaban, lo supimos resolver ya que era un tema de cambiar los repositorios usados a los nuevos en la Lógica de Negocio, que nos costó darnos cuenta.

A pesar de esto, esta instancia del obligatorio fue una oportunidad valiosa de aprendizaje. Sumado a lo aprendido en la primera instancia, sumamos conocimientos sobre implementación con bases de datos y aspectos relacionados, cómo y dónde ejecutarlas, el uso de herramientas que permiten visualizarlas de forma gráfica, sus métodos, y cómo trabajar con ellas, entre otros temas.

## Documentación de errores

**Asunto:** - Error de Repositorio en memoria en Tests de Epica.

**Explicación:** - El error consiste en que el repositorio en memoria de los test de Épica, no funciona en 5 de los test. Lo que sucede está relacionado a cómo trackea la base de datos inMemory, elementos que son un caso límite y requieren del uso de “update”, se le están pasando 2 Épicas de las cuales solo consigue trackear 1.



```
BusinessLogicTest (5 tests) Failed: 5 tests failed
  BusinessLogicTest (5 tests) Failed: 5 tests failed
    EpicaLogicTest (5 tests) Failed: 5 tests failed
      CalcularEsfuerzoEstimadoEpica_ConUnaTarea_DebeRetornar1 Failed: Test method BusinessLogic
      CalcularEsfuerzoInvertidoEpica_ConUnaTarea_DebeRetornar1 Failed: Test method BusinessLogic
      DadoDosEpicas_NoDebeAgregarUnaTareaQueYaPertenezcaAUnaDeEllas Failed: Test method thre
      DadoUnaEpicaConTarea_DebeQuitarlaCorrectamente Failed: Test method BusinessLogicTest.Epic
      DadoUnaEpicaValida_UnaTareaValida_DebeAgregarlaCorrectamente Failed: Test method Business
```

**Asunto:** - Error en agregado a la papelera eliminando una tarea de una épica

**Explicación:** - Es un pequeño detalle que nos dimos cuenta al final de la entrega, lo que pasa es que al agregar una tarea vencida a una épica, luego eliminarla y que vaya a la papelera, restaurarla y que aparezca en las tareas vencidas y en la épica, si la modificas cambiándola a activa y la vuelves a eliminar se elimina dos veces, lo que debería de pasar es que básicamente no se añada dos veces, porque la fecha y todo se actualiza bien pero queda guardada en la lista de tareas vencidas, aunque no aparezca.

## Instructivo de instalación

Como en el primer Obligatorio, esto fue una tremenda experiencia, ya que nos abrió las puertas al uso de las Bases de Datos en un proyecto. Nos pareció muy interesante y realista este proyecto, y lo tomamos como una enriquecedora experiencia para nuestro aprendizaje, ya que es la primera vez que trabajamos con una base de datos y nos ayudó mucho a entender el uso de esta.

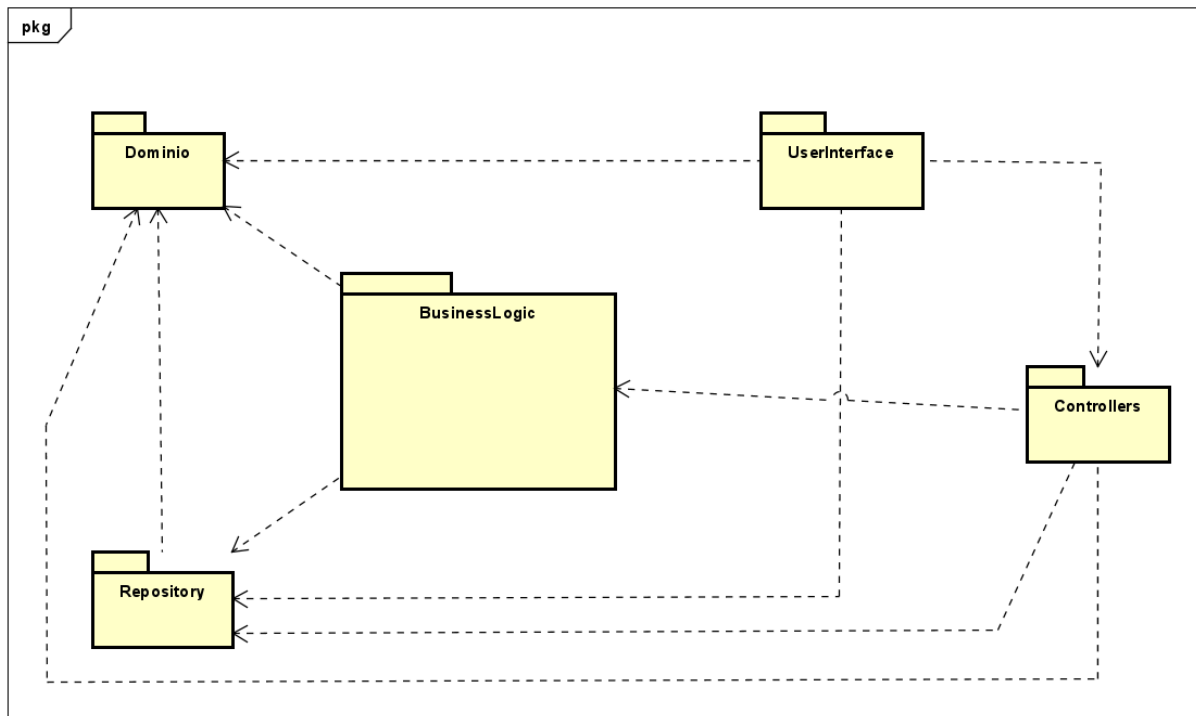
Para ejecutar el programa, es necesario tener instalados DBeaver y Docker Desktop. Primero, se debe iniciar el contenedor utilizando los comandos que se detallan a continuación en una consola de PowerShell. Luego, en DBeaver, se debe conectar a una nueva base de datos de SQL Server, usando el nombre de usuario "sa" y la contraseña "Passw1rd". Sobre esta base de datos "master", se crea una nueva base de datos llamada "TaskPanelDB", y se emplean los scripts SQL para crear las tablas e ingresar los datos correspondientes.

Los comandos son los siguientes:

- `docker pull mcr.microsoft.com/azure-sql-edge`
- `docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=Passw1rd' -p 1433:1433 --name sqlserver -d mcr.microsoft.com/azure-sql-edge`

# Justificación de Diseño

## Diagrama de paquetes



TaskPanel se organiza en varios proyectos o paquetes para poder asegurarnos de tener una clara separación de responsabilidades y un buen encapsulamiento para poder facilitar y mejorar el mantenimiento y la extensibilidad del código. Este concepto fue visto en clase, cuando trabajamos con Clean Code, en donde la mantenibilidad y la legibilidad del código eran más importantes que arreglar los errores de lógica.

Los proyectos son:

→ Dominio: En este paquete están contenidas las clases que representan las entidades principales del proyecto junto con sus respectivos atributos y es también donde se definieron las estructuras de datos.

→ BusinessLogic: Como el nombre lo dice, este paquete contiene las clases que poseen toda la lógica de negocio acerca de las entidades, como también las operaciones que no están relacionadas a un única entidad del dominio. Es la capa intermedia entre la interfaz y el repositorio.

→ Repository: Este paquete gestiona los datos y la persistencia de los mismos. Su propósito principal es abstraer y encapsular el acceso a la base de datos, permitiendo que otras partes de la aplicación interactúen con los datos sin preocuparse por los detalles específicos de cómo se almacenan y recuperan.

→ UserInterface: Por último el paquete del front-end. Este contiene todo lo relacionado a la interfaz de usuario y la presentación de los datos y acciones del sistema a los usuarios. Su propósito principal es presentar la información y procesar las entradas del usuario. Esta capa comunica las solicitudes al back-end.

→ **Controllers**: Paquete que contiene los controladores, aprendidos en el tema GRASP. Estos funcionan como una capa de abstracción extra entre la Interfaz de Usuario y la Logica de Negocios, es una capa intermedia.

Los primeros 3 paquetes van relacionados a sus respectivos paquetes de tests, en donde se almacenan las pruebas unitarias de cada uno (**DominioTest**, **BusinessLogicTest** y **RepoTest**).

→ **BusinessLogicTest**: Se guardan las pruebas unitarias de las clases de **BusinessLogic**

→ **DominioTest**: Se guardan las pruebas unitarias de las clases de **Dominio**.

→ **RepoTest**: Se guardan las pruebas unitarias de las clases de **Repository**.

Dentro de las principales carpetas que actúan como paquetes dentro de estos proyectos existen otras carpetas, las cuales fueron claves para el desarrollo pero no se muestran aquí. Estas son:

→ **DTO (Data Transfer Objects)**: Para encapsular un conjunto de datos y facilitar el intercambio de los mismos entre las distintas capas. Nosotros implementamos 2 clases de DTO, una para el **UsuarioEnviar**, que es el que está conectado a la pagina, osea, el que muestra la interfaz, y en el cual excluimos la property **Contraseña**, y por otro lado el **UsuarioRecibir** que sería el usuario que va a ingresar el que está en sesion, osea el nuevo usuario, en el cual si se incluye la property **Contraseña**.

→ **BusinessLogic.Exceptions**: Gestiona las excepciones específicas que pueden ocurrir dentro de la lógica de negocios de la aplicación.

→ **Dominio.Exceptions**: Gestiona las excepciones específicas que pueden ocurrir dentro del dominio de la aplicación.

→ **Repository.Exceptions**: Gestiona las excepciones específicas que pueden ocurrir dentro del repositorio de la aplicación.

→ **Interfaces**: Contiene todas las interfaces que extienden las clases del repositorio, permitiendo generar una capa de abstracción para mejorar y favorecer a la mantenibilidad y extensión del sistema.

→ **Migrations**: Es donde se guardan todas las migraciones que son utilizadas por el DBMS para crear las tablas y sus atributos

→ **SqlContext**: Esta carpeta contiene la clase **SqlContext** en donde se hacen los **DbSet** y el **OnModelCreating** para la parte de la base de datos.

Namespace	Clase	Responsabilidad
Dominio	Usuario	Representar la entidad de un usuario del sistema. Se encarga de validar la correctitud de tanto los usuarios normales como los administradores y su contraseña.
Dominio	Equipo	Representa a un equipo compuesto por usuarios. Está encargado de validar que el mismo esté apropiadamente conformado.
Dominio	Tarea	Representar una tarea a realizar por un equipo y sus propiedades. Se encarga de validar la correctitud de su entidad.
Dominio	Panel	Representa un panel de tareas. Chequea y valida su conformación y correctitud.
Dominio	Comentarios	Representa a los comentarios que se le puede dejar a las tareas. Se valida una correcta conformación del mismo.
Dominio	Epica	Representa una épica que agrupa tareas relacionadas. Valida su correcta conformación y calcula esfuerzos estimados e invertidos.
Dominio	Notificacion	Representa las notificaciones enviadas a los usuarios. Se encarga de validar su formato y contenido.
Dominio.Exceptions	Una clase de excepciones por cada clase del	Referenciar las excepciones lanzadas

	dominio.	por las validaciones de las clases del dominio.
Repository	UsuarioDBRepository	Administra el almacenamiento, retención y consistencia de los datos de los usuarios en la base de datos.
Repository	EquipoDBRepository	Administra el almacenamiento, retención y consistencia de los datos de los equipos en la base de datos.
Repository	TareasDBRepository	Administra el almacenamiento, retención y consistencia de los datos de las tareas en la base de datos.
Repository	PanelDBRepository	Administra el almacenamiento, retención y consistencia de los datos de los paneles en la base de datos.
Repository	ComentariosDBRepository	Administra el almacenamiento, retención y consistencia de los datos de los comentarios en la base de datos.
Repository	EpicaDBRepository	Administra el almacenamiento, retención y consistencia de los datos de las épicas en la base de datos.
Repository	NotificacionDBRepository	Administra el almacenamiento, retención y consistencia de los datos de las notificaciones en la base



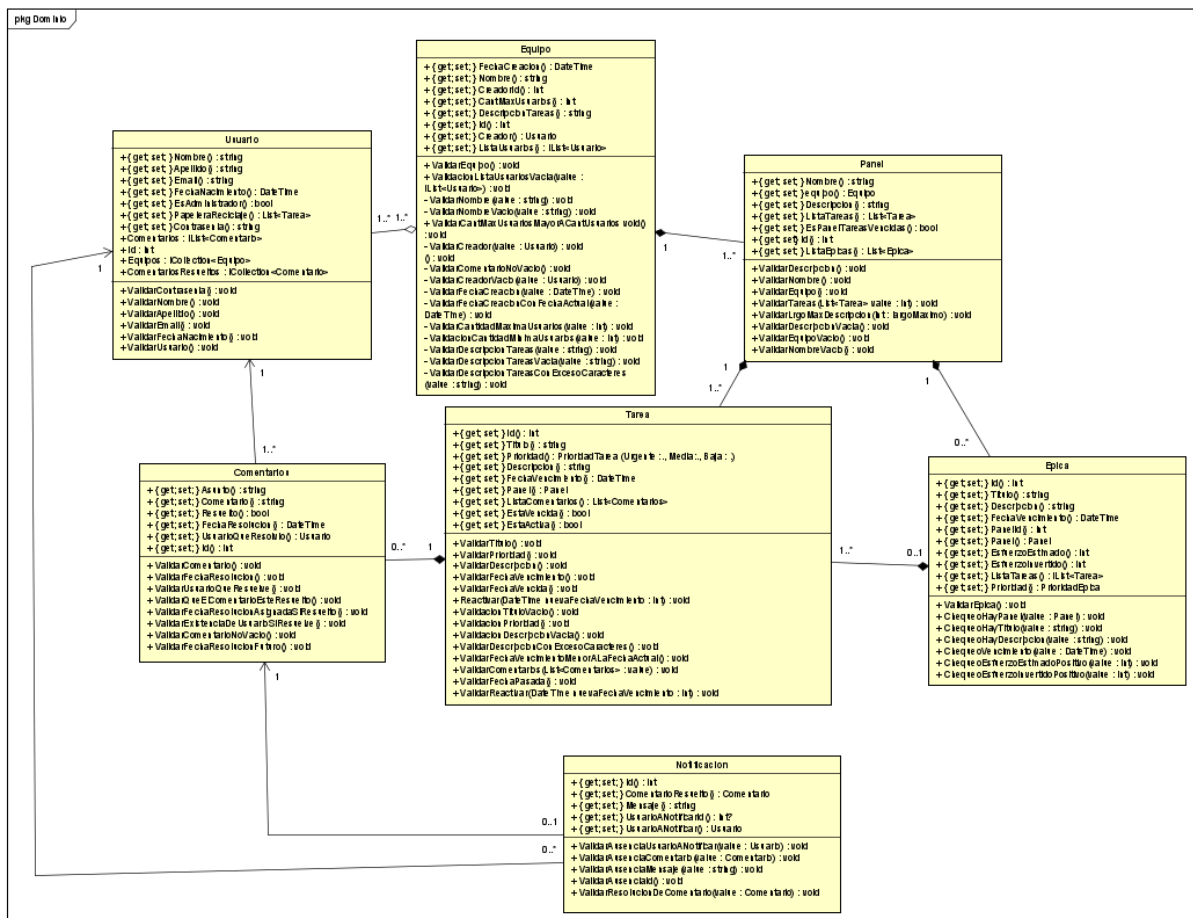
		de datos.
BusinessLogic	SesionLogic	Valida la autenticidad y las credenciales de un usuario a la hora de iniciar sesión.
BusinessLogic	UsuarioLogic	Maneja la lógica de la creación, modificación y eliminación de usuarios. También administra la papelera de reciclaje para paneles y tareas.
BusinessLogic	EquipoLogic	Gestiona la lógica de creación y modificación de los equipos.
BusinessLogic	TareaLogic	Gestiona la lógica de la creación, modificación y administración de tareas además de la reactivación de tareas vencidas. Permite agregar comentarios e importar tareas desde archivos.

BusinessLogic	PanelLogic	Gestiona la lógica de la creación, modificación y eliminación de paneles.
BusinessLogic	NotificacionLogic	Gestiona la lógica de la creación y eliminación de notificaciones de resolución de comentarios.
BusinessLogic	EpicaLogic	Gestiona la lógica de creación, modificación y administración de épicas. Calcula los esfuerzos estimados e invertidos de las épicas, y permite asociar o eliminar tareas de ellas.
BusinessLogic	ComentariosLogic	Gestiona la lógica de la adición y resolución de

		comentarios
UI.Data	SessionService	Gestiona la autenticación y el estado del usuario en la sesión, permitiendo iniciar, verificar y cerrar la misma.

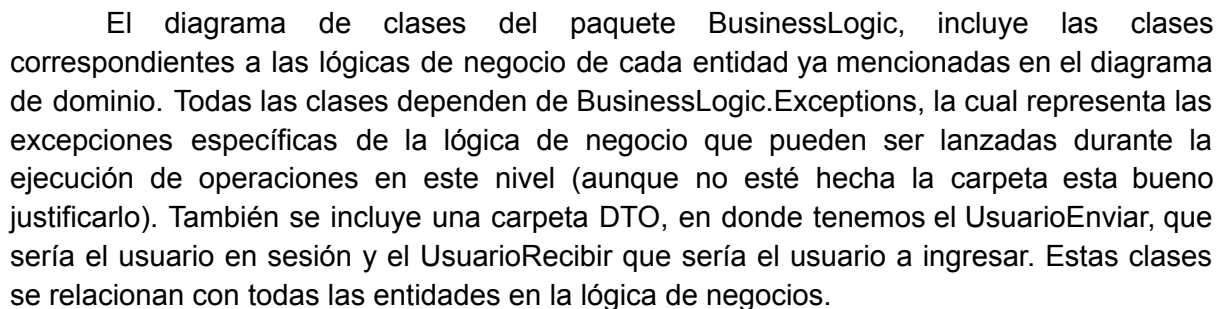
## Diagrama de clases

### Diagrama del Dominio:

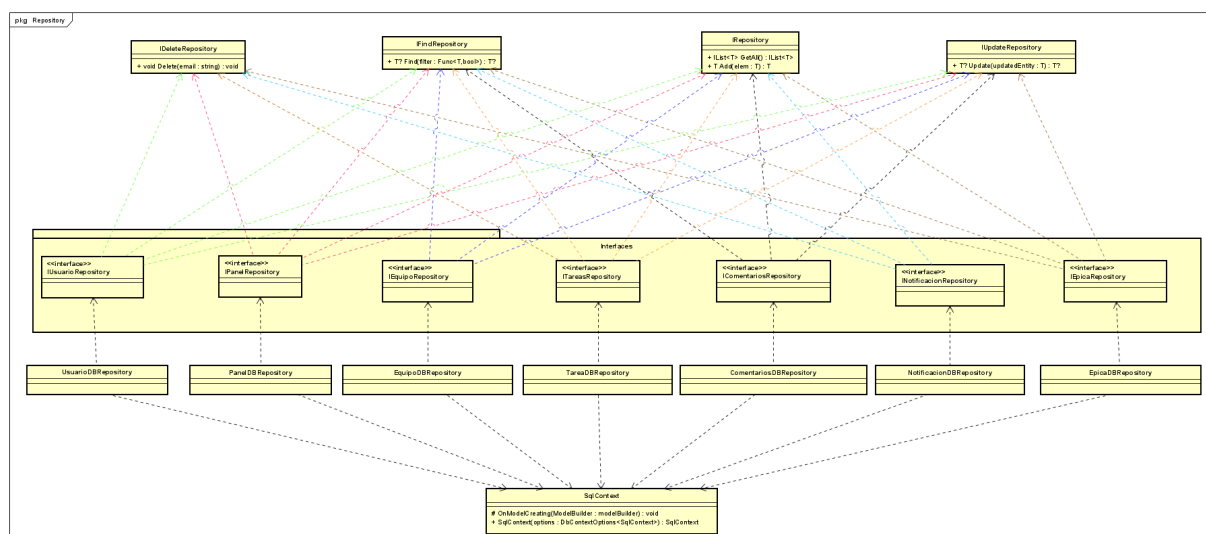


El diagrama de clases del paquete Dominio, incluye las clases correspondientes a las entidades de nuestro obligatorio, las cuales son: Comentario, Épica, Equipo, Notificación, Panel, Tarea y Usuario. Estas clases no están en una carpeta ya que son el principal componente de este paquete. Por otra parte tenemos la carpeta de Dominio.Exceptions la cual contiene una clase para cada entidad de exception, es decir, tenemos UsuarioExceptions, TareaExceptions, etc. En las relaciones entre las distintas clases, se agregó la multiplicidad de la relación entre ellas. A modo de ejemplo, se puede observar como la relación entre Usuario y Equipo es una de muchos a muchos (N-N), ya

## Diagrama del BusinessLogic



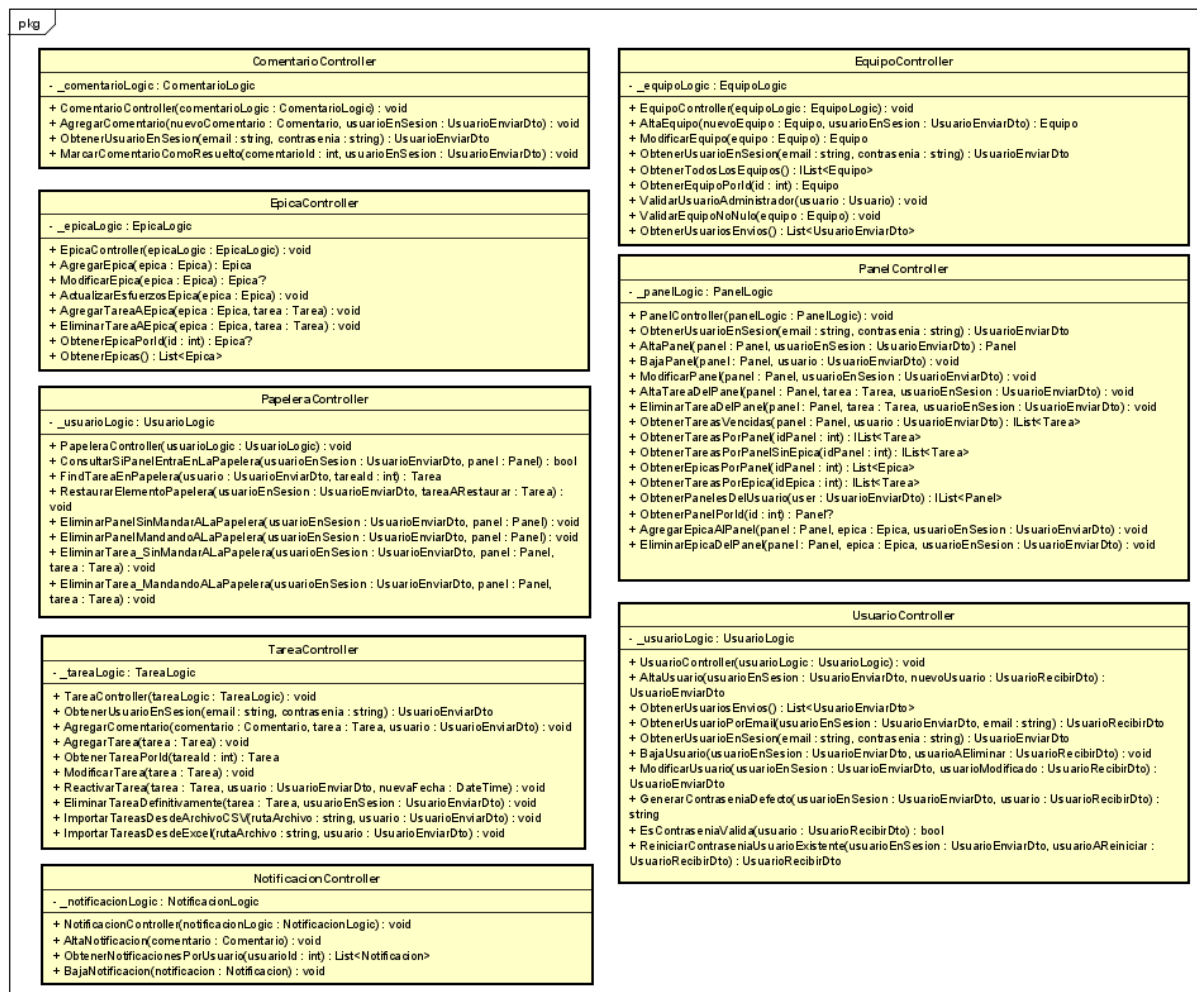
El diagrama de clases del paquete Repository muestra la relación entre las diferentes clases del mismo (respetando las entidades ya mencionadas) y el contexto de



base de datos (SqlContext, en la carpeta SqlContext) en la aplicación, destacando la centralización del acceso a datos a través de SqlContext (en donde está ubicado el método OnModelCreating) y el uso de repositorios específicos para manejar las diferentes entidades. También en este paquete destacamos la carpeta de interfaces, en donde las acciones fueron separadas por interfaz siguiendo el principio 4 de SOLID. Este diagrama fue hecho aparte para que todo sea más claro. Por esto se debe tener en cuenta que entre cada repositorio y su respectiva interfaz hay una relación de dependencia, que aquí solo se muestra con la carpeta de Interfaces. Por otro lado, agregamos la carpeta de Repository.Exceptions la cual será utilizada para realizar los tests de Repository.

El diagrama de interfaces del paquete Repository, como podemos observar, el mismo está separado por acciones ya que no todas las entidades realizan todas las acciones posibles. Cada una de estas interfaces de acciones va relacionada a la interfaz correspondiente según las acciones que se realicen con dicha entidad. Esta fue la forma de asegurarnos que no hayan métodos en las interfaces que no son utilizados o son innecesarios, de no repetir código y que sea una capa más abstracta o encapsulada a todo lo relacionado con el acceso a datos y su manipulación.

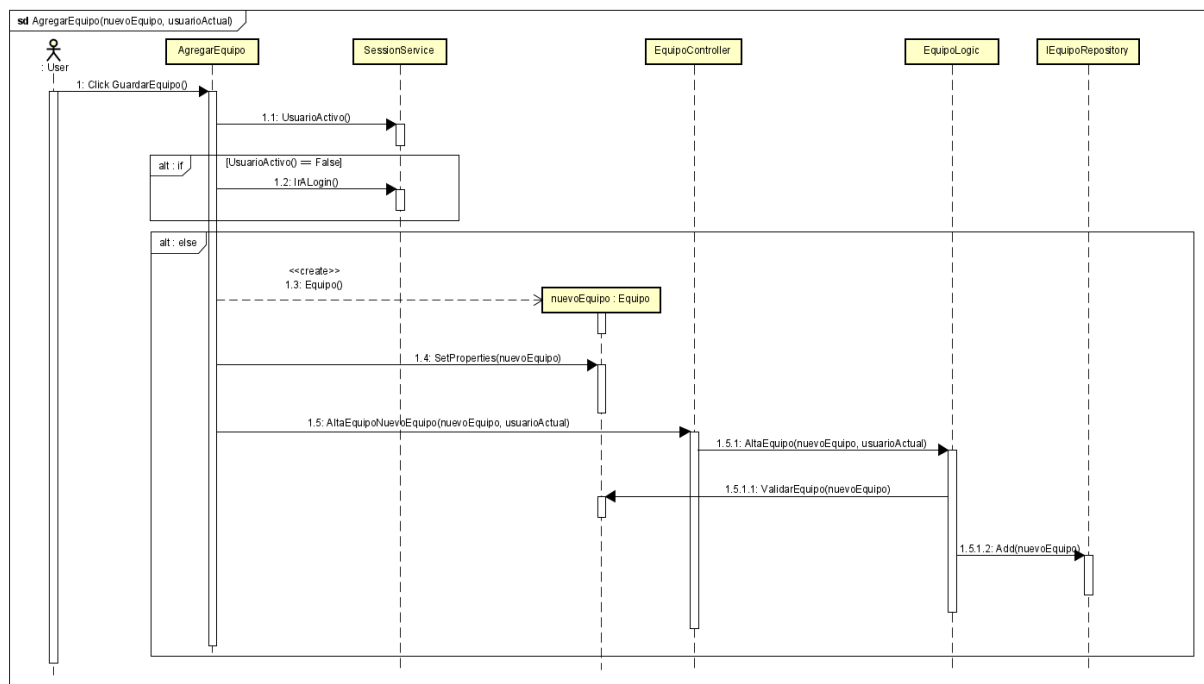
## Diagrama de Controllers



Por último, está el paquete de los controladores. El diagrama es sencillo ya que no existe relación alguna entre las distintas clases. Esta capa se creó simplemente para generar más abstracción entre la interfaz de usuario y la lógica de negocios, lo que se fundamentará más adelante.

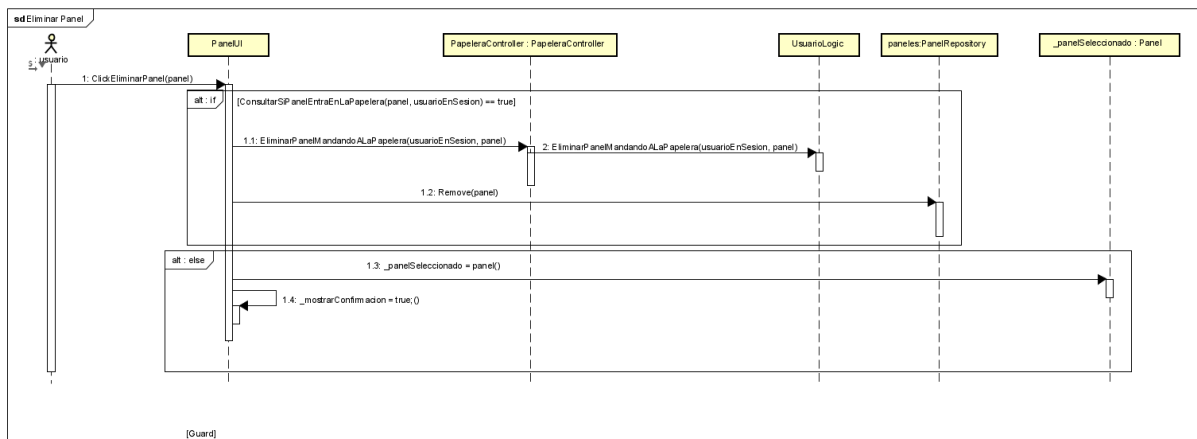
## Diagrama de interacción

### Diagrama de secuencia para la funcionalidad Agregar Equipo:



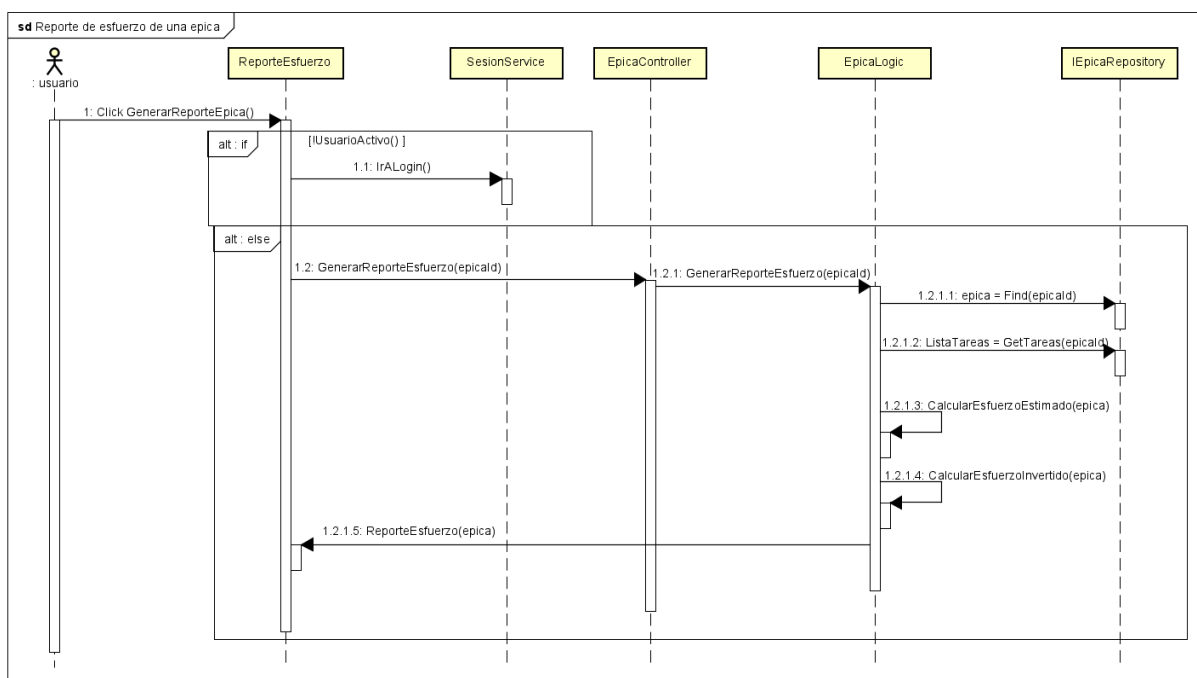
El diagrama de secuencia muestra el proceso de cómo agregar un equipo desde la interfaz de usuario hasta la agregación del equipo al repositorio. Esto sucede cuando el usuario en sesión da click en el botón agregar equipo. Al clicar se va a pedir el ingreso de los datos del equipo, tanto nombre como descripción, lista de usuarios entre otros y se realiza la implementación adecuada. A través del controlador se accede a la lógica de equipo para finalmente agregar el mismo al repositorio correspondiente.

## Diagrama de secuencia para la funcionalidad Eliminar Panel:



El diagrama de secuencia muestra el proceso de como eliminar un panel desde la interfaz de usuario hasta la eliminación del mismo del repositorio. Esto sucede cuando el usuario en sesión da click en el botón baja. Al clickear se pregunta si el mismo entra en la papetera, sino se elimina definitivamente. A través del controlador se accede a la lógica de panel para finalmente eliminar el mismo del repositorio correspondiente.

## Diagrama de secuencia para la funcionalidad Reporte de una épica:



El último diagrama de secuencia muestra el proceso de cómo se genera el reporte de una épica desde la interfaz del usuario. El mismo se genera cuando a una épica se le agrega al menos una tarea, la cual debe contener un esfuerzo estimado mayo al invertido para generar el reporte y ver cuánto esfuerzo falta para llegar al estimado. También se genera una comparativa para las tareas dependiendo de cuánto esfuerzo estimado e

## Modelo de tablas

[illegible]

- Diferencia entre los tipos de datos utilizados, por ejemplo, en dominio para decir si un usuario era administrados, era una booleana (true o false), en cambio en el modelo de tablas se identificó como 1 para true, 0 para false en forma de int.
- En dominio se tiene referencia directa a los otros objetos que tenga dicha clase, para facilitar las operaciones de negocio y la navegabilidad entre objetos, en cambio en el modelo de tablas, mediante algunas técnicas, como las claves foráneas, se intentan evitar redundancias y mejorar la integridad de los datos
- También destacar el uso del DbContext para la creación de tablas, lo que en el dominio se presenta como una lista del tipo correspondiente para almacenar los objetos necesarios, el DBMS ya entiende que es necesario crear una tabla separada que vincule de cierta manera (a través de las claves

foráneas) estos dos objetos (por ejemplo en el caso usuario equipo, con la tabla EquipoUsuario)

## Descripción de las principales decisiones de diseño tomadas

Se ha intentado seguir los principios de diseño SOLID y GRASP para asegurar la mantenibilidad y la extensibilidad del sistema.

La UI o Interfaz de Usuario interactúa directamente con el Dominio, lo cual incluye instanciar las clases del dominio directamente en la misma UI. Esto se debe ya que consideramos que la UI es el “experto” según GRASP en dichas clases, ya que ahí es donde se tienen los datos. Al haber aplicado DTO para la entidad usuario, pudimos encapsular un conjunto de datos y facilitar el intercambio de estos entre las distintas capas.

Previo a iniciar con el desarrollo del proyecto, dedicamos un tiempo a pensar cuales iban a ser las clases principales para el mismo, además de que responsabilidades iban a tener las clases. Con esto y con las pruebas TDD (Test-Driven-Development) poco a poco se fueron definiendo sus atributos y su armado.

Decidimos separar el código en secciones: Dominio, Lógica, Controllers, Repositorio e Interfaz, cada uno con su responsabilidad, bajando el acoplamiento y la cohesión entre paquetes. Luego sabiendo que el proyecto iba a tener que ser realizado utilizando TDD discutimos y fuimos identificando cuáles eran las pruebas que íbamos a tener que realizar para verificar los requerimientos de cada parte de la letra. Para el manejo de errores, los paquetes Dominio y Lógica tienen sus respectivos tipos de excepciones: el nombre de la clase seguido de exception para las clases del dominio y BusinessException para las clases de lógica. Estas son lanzadas ya sea al haber un ingreso erróneo de datos o una operación incorrecta sobre la lógica respectivamente. Se diferencian los tipos de excepciones para tener certeza de donde ocurren, así como con la asignación de un mensaje descriptivo a cada una para poder informar cual es el error que se ha cometido.

Para asegurar la aplicación de los principios SOLID, fue necesario tomar diferentes medidas:

**Single Responsibility Principle:** En general en nuestro código las diversas clases no contaban con motivos por el cual cambiar, ya que desde un principio nuestras clases estaban divididas según las funcionalidades o intereses de los distintos actores de negocio gracias al obligatorio anterior. En cuanto a los nuevos requerimientos funcionales, los mismos se pudieron incluir en clases ya existentes (ya que formaban parte del mismo actor de negocio) o en clases nuevas, por ejemplo notificación y épicas.

**Open Closed Principle:** Se buscó que las clases estén abiertas para extensión y cerradas para modificación gracias a la implementación de interfaces. En el caso de la importación de archivos, se agregó la funcionalidad para procesar archivos XLSX sin modificar la implementación existente para archivos CSV. Esto se debe a que, tras la investigación realizada, se concluyó que ambos tipos de archivos requieren un tratamiento distinto en el backend debido a las librerías y métodos de importación utilizados. Siguiendo el mismo enfoque conceptual empleado para los archivos CSV, se implementaron funciones separadas para cada formato. En la interfaz, se estructuró la lógica para manejar de forma independiente la importación de cada tipo de archivo.



**Liskov substitution:** Las clases derivadas son sustituibles por sus clases base. Las implementaciones de los repositorios pueden ser usadas sin instanciar la clase concreta y llamando a la interfaz correspondiente.

**Dependency Inversion Principle:** Para garantizar el cumplimiento de este principio, estructuramos el diseño de manera que los módulos de alto nivel, como las clases de lógica de negocio, no dependan directamente de implementaciones concretas de los repositorios. En lugar de eso, dependen de abstracciones, específicamente interfaces, que actúan como contratos estables entre las capas del sistema.

**Interface Segregation Principle:** Las interfaces están divididas en interfaces más pequeñas y específicas (ya aplicado en el obligatorio 1 con el tema de las acciones de cada entidad). Esto asegura que las clases implementen solo los métodos que se consideren necesarios.

También se aplicaron los principios de GRASP para la asignación de responsabilidades. Aquí van los principales utilizados:

**Creador/Experto:** El ejemplo más claro es la interfaz de usuario, donde se instancian las clases de dominio porque allí se tienen los datos. Las clases que tienen la información necesaria para cumplir con una responsabilidad son las que realizan las operaciones.

**Controladores:** Los controladores son responsables de manejar las solicitudes del usuario y delegar el trabajo a otros objetos. Son esa capa intermedia entre la UI y la BusinessLogic.

**Bajo acoplamiento/Alta cohesión:** Si bien estos dos puntos no están lo más optimizados posibles debido a una planeación no tan correcta en el tema de dependencias y/o responsabilidades, se intentó minimizar las dependencias entre clases lo más posible para que las clases no tengan tantas responsabilidades.

Las inyecciones de dependencia fueron todas agregadas y de manera correcta. Todos los servicios fueron agregados de forma "Scoped" menos el que guarda al usuario en sesión en la aplicación (Active user) que al solo poder ingresar de a un usuario a la vez, son creados una única vez y son reutilizados a lo largo de toda la aplicación, que fue agregado de forma "Singleton".

## **Criterios seguidos para asignación de responsabilidades**

En nuestro proyecto, hemos organizado las clases de manera que cada una tenga una responsabilidad clara y única, siguiendo el principio de responsabilidad única. Las clases del dominio se encargan exclusivamente de gestionar sus propios atributos, incluyendo su lectura, modificación y validación.

Por otro lado, las clases de la lógica del negocio son responsables de administrar el almacenamiento de las entidades del dominio, definiendo cómo deben interactuar con los repositorios y realizando operaciones que dependen de la creación previa de estas entidades o de su relación con otras dentro de la lógica.

Los repositorios, diseñados para trabajar con bases de datos, tienen la tarea de garantizar la persistencia de los datos y de proporcionar las operaciones necesarias para gestionar y manipular la información almacenada. Estos repositorios interactúan directamente con el contexto SQL, que representa la conexión activa con la base de datos.

En cuanto a la interfaz de usuario, esta no tiene conocimiento sobre la lógica de negocio ni interviene en ella. Su única función es ejecutar las operaciones disponibles a través de las instancias de la lógica, capturar las excepciones que puedan generarse y notificar al usuario de cualquier error o incidencia de manera adecuada.

## Análisis de dependencias

El análisis de dependencias para el requerimiento de reporte de esfuerzo de una épica involucra varias clases del dominio y sus relaciones. La clase principal es Épica, que representa la entidad central en esta funcionalidad. Sus atributos incluyen el esfuerzo estimado y el esfuerzo invertido, que se calculan a partir de las tareas asociadas en la lista ListaTareas. Cada tarea contribuye directamente a los esfuerzos acumulados de la épica, estableciendo una dependencia clara entre ambas clases. Además, la épica está relacionada con un panel a través de su propiedad PanelId, lo que establece el vínculo con un contexto organizacional.

En la lógica de negocio, EpicaLogic coordina estas relaciones y procesos. Incluye métodos como ActualizarEsfuerzosEpica, que recalcula los valores de esfuerzo estimado e invertido, y métodos auxiliares como CalcularEsfuerzoEstimado y CalcularEsfuerzoInvertido, que suman los valores individuales de las tareas. Esta clase también utiliza repositorios como IEpicaRepository e IPanelRepository para manejar la persistencia de datos y se apoya en PanelLogic y SesionLogic para manejar aspectos específicos de paneles y contexto de sesión, respectivamente.

En cuanto a cohesión, la clase Épica muestra un diseño altamente cohesivo, ya que todos sus atributos y métodos están enfocados en gestionar la información y las validaciones relacionadas con una épica. De manera similar, EpicaLogic mantiene su cohesión al centralizar la lógica relacionada con las épicas. Respecto al acoplamiento, existe una relación necesaria y bien controlada entre Épica y Tarea, dado que las tareas son esenciales para calcular los esfuerzos. Por otro lado, el acoplamiento de EpicaLogic con múltiples clases y repositorios es moderado, pero justificado por su rol como coordinador de diferentes aspectos funcionales.

## Informe de cobertura de los casos de prueba












En el equipo siempre estuvo como objetivo realizar la mayor cantidad de tests unitarios posibles, para asegurarnos de tener una buena cobertura del código. Igualmente, en muchos casos tuvimos que continuar con la lógica e implementación del proyecto para retomar luego la elaboración de test y poder aspirar a tener la cobertura de la totalidad del código. En total hubo **339** Test Unitarios, de los cuales fallaron los 5 test documentados.

**(Ver anexo)**

# Conclusión

Para terminar este proyecto, nos pareció una experiencia muy enriquecedora en cuanto a trabajo en grupo compete, con un aprendizaje muy intenso a lo largo de todo el plazo. Por momentos se hizo confuso el pasar de lo dado en teórico en clase a ponerlo en práctica nosotros, más que nada a la hora de afrontar el pasaje entre el uso de repositorios en memoria y el uso de una verdadera Base de Datos, esto nos complicó mucho en varias instancias, igualmente supimos afrontarlo de buena manera y terminamos aclarando nuestras dudas. Este obligatorio nos ha enseñado mucho en cuanto a prácticas de programación como TDD, Clean Code, GRASP y SOLID lo cual nos ha aportado mucho a nuestras capacidades. Nos quedamos con todo lo aprendido y los desafíos que hemos enfrentado y logrado superar.

# Anexo

▼  Total	94%	115/1941
▼  Repository	99%	1/252
▼  Repository	99%	1/252
>  ComentarioDBRepository	100%	0/27
>  EquipoDBRepository	100%	0/33
>  NotificacionDBRepository	100%	0/20
>  PanelDBRepository	100%	0/31
>  TareaDBRepository	100%	0/38
>  UsuarioDBRepository	100%	0/36
>  SqlContext	100%	0/35
>  EpicaDBRepository	97%	1/32

