

Universidad ORT Uruguay

Facultad De Ingeniería

DOCUMENTACIÓN OBLIGATORIO DISEÑO DE APLICACIONES 1

Nicolas Bidenti (305108)
Santiago Canadell (282542)
Felipe Delgado (281987)

Tutor: Luis Simón

2025

https://github.com/IngSoft-DA1/305108_282542_281987

A handwritten signature in black ink, appearing to read "Nicolás B. Scarzela". The letters are cursive and somewhat stylized.

Nicolás Bidenti Scarzela 27/04/2025

A handwritten signature in black ink, appearing to read "Santiago Canadell". The signature is written in a cursive style with a prominent horizontal stroke at the end.

Santiago Canadell 27/04/2025

A handwritten signature in black ink, appearing to read "Felipe Delgado". The signature is written in a cursive style with a long, sweeping horizontal stroke at the end.

Felipe Delgado 15/05/2025

Índice

1) Introducción al proyecto:	4
• Descripción general	4
• Diseño	4
UML	4
• Código	5
Estructura y Dependencias Generadas	5
Explicación de validaciones de negocio	12
Mantenibilidad y extensibilidad	12
-Git	13
Gitflow	13
-Test	14
Cobertura De pruebas	14
Pruebas Funcionales De ABM	15
-TDD	15
-Buenas Prácticas (Mantenibilidad y extensibilidad)	18
-Cálculo De Ruta Crítica	19
-Diagrama de Gantt	21
3) Problemas Enfrentados	21
• Bugs conocidos	21
• Funcionalidades no implementadas	22
4) Anexo	22
• Cálculo De Ruta Crítica: TDD	22
• Dependencias Externas	24
• Uso de IA	24
• Foto coverage de pruebas	25
• Fotos UML	25
SERVICE	25
DATA ACCESS	28
DOMAIN	29
DIAGRAMA DE PAQUETES (todo el proyecto)	31
• Link UML	31

1) Introducción al proyecto:

- Descripción general

Empezamos el proyecto maquetando un UML que tenga sentido en base a la letra del obligatorio. En base a ese UML y la letra comenzamos a diagramar las partes principales del proyecto, dividiéndolas en packages que los cuales especificamos en profundo más abajo en la documentación. Comenzamos por Domain, luego seguimos con Data Access y finalmente Service para poder empezar con el Front-end.

El Front lo intentamos hacer intuitivo, armónico de ver y lo más fácil de usar posible para el usuario. Para eso usamos por ejemplo iconos de bootstrap y los errores en color rojo especificando porque estaba mal el camino que el usuario estaba tomando.

También hicimos que en el momento que a un usuario se le asigna un Rol, este se actualiza instantáneamente, sin necesidad de que el usuario haga logout y login para visualizar los cambios.

Además de estos aspectos, implementamos pruebas unitarias desde el principio, con el objetivo de garantizar la estabilidad y calidad del proyecto, permitiéndonos verificar los cambios que hacíamos en las funciones de forma rápida.

Con respecto al backend, decidimos estructurar la lógica del negocio de forma modular, separando las responsabilidades de cada componente dentro del servicio, como lo vimos en la clase de práctico. Esto no solo facilita la escalabilidad del proyecto, sino que también permite integrar nuevas funcionalidades sin afectar las partes ya implementadas.

2) Estructura del Proyecto

- Diseño

UML

Al UML lo dividimos por Packages. Hicimos uno de Domain, otro de Service y el último de Data Access. Ya que los otros Packages son los de tests y de Interface.

Las fotos del UML se encuentran en el anexo, parte de Fotos UML. Pero dejamos un link a drive con el astah ya que al ser clases tan grandes no se ve casi nada desde el pdf.

Aclaraciones Domain

El proyecto puede tener varios miembros y contiene múltiples tareas. Contiene siempre un administrador que es un usuario con el rol Admin Project.

Usamos IDs como identificador para task y notification. Para user usamos identificador el email y para project el nombre.

Task-Task (agregación reflexiva): Las tareas mantienen referencias a tareas precedentes y simultáneas mediante agregación, lo que permite hacer flujos de trabajo sin crear ciclos de dependencia fuertes.

Task-Resource (agregación): Las tareas pueden utilizar múltiples recursos, modelados como agregación porque los recursos pueden existir independientemente de las tareas.

Aclaraciones Service

Cada servicio se especializa en operaciones relacionadas con una entidad o proceso específico (UserService, ProjectService, TaskService), siguiendo el principio de responsabilidad única.

Conversión DTO-Entidad: Los servicios tienen la lógica de conversión entre DTOs y entidades de dominio, protegiendo el modelo de dominio de exposición directa a el Front-end.

Todos los servicios tienen una relación de asociación con InMemoryDatabase, permitiéndoles acceder a los repositorios necesarios mientras tienen bajo acoplamiento.

En algunos services utilizamos excepciones de Dominio y de Data Access.

Aclaraciones Data Access

Generación Automática de IDs: Los repositorios como NotificationRepository, ResourceRepository y ProjectRepository generan los IDs, gestionando la identidad de entidades.

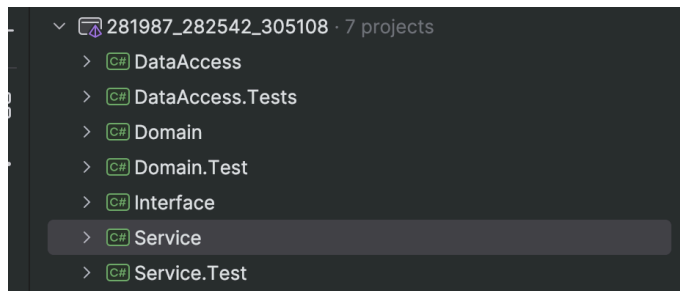
Clonación defensiva: Los repositorios devuelven copias de colecciones (ToList()) en métodos GetAll para evitar modificaciones accidentales.

Búsqueda por Predicado: Los repositorios implementan métodos Get que aceptan Func<T, bool> como parámetros, permitiendo búsquedas flexibles.

Búsqueda por Expresiones: La utilización de Func<T, bool> para búsquedas crea una relación flexible entre los repositorios y los criterios de búsqueda, dejando hacer consultas complejas.

- Código

Estructura y Dependencias Generadas



En el proyecto, la inyección de dependencias es muy importante para conectar las distintas capas del sistema, facilitando la interacción entre el front-end (UI) y el back-end. Para ello, en la capa de Interface (Front-End), se inyectan los servicios necesarios en cada página a través de la inyección de dependencias. Estos servicios, que están definidos en la capa Service, son responsables de la lógica de negocio y de interactuar con el dominio, pero sin que el front-end tenga acceso directo a las entidades del dominio.

Cuando una página necesita realizar alguna operación, como crear un proyecto o actualizar un usuario, obtiene el servicio mediante inyección, ya sea el Login, UserService, AdminPService, etc. Este patrón permite que cada página tenga acceso solo a los servicios necesarios, evitando la creación manual de instancias y ayudando a la reutilización del código.

Los servicios inyectados utilizan DTOs para trasladar los datos entre el front-end y el back-end. Por ejemplo, cuando se crea un proyecto, el front-end utiliza un ProjectDTO para enviar los datos estructurados al back-end, donde se transforman en una entidad de Project para ser procesados. De esta forma, el uso de DTOs separa la lógica del sistema de la representación de datos, lo que permite un mayor mantenimiento.

Asignación De Responsabilidades

DataAccess

Namespace	Clase	Responsabilidad
DataAccess	UserRepository.cs	Manejo de la lista de Users en InMemoryDatabase.
DataAccess	ResourceRepository.cs	Manejo de la lista de Resources en InMemoryDatabase
DataAccess	ProjectRespository.cs	Manejo de la lista de Projects en InMemoryDatabase
DataAccess	NotificationRepository.cs	Manejo de la lista de Notifications en

		InMemoryDatabase
.DataAccess	InMemoryDatabase.cs	Manejador de listas de Users, Notifications, Projects y Resources repositories, es el encargado de crear y de guardar en memoria las mismas.
.DataAccess.Exceptions.UserRepositoryExceptions	Carpeta de exceptions de user repository: -UserEmailsDuplicatedException.cs -UserNotFoundException.cs -UserRepositoryException.cs	Excepciones para casos border inválidos dentro de la clase UserRepository.
.DataAccess.Exceptions.TaskRepositoryExceptions	Carpeta de exceptions que forma parte de project repository: -TaskAlreadyExistsException.cs -TaskNotFoundException.cs -TaskRepositoryExceptions.cs	Excepciones para casos border inválidos que forman parte de la clase ProjectRepository.
.DataAccess.Exceptions.ResourceRepositoryExceptions	Carpeta de exceptions de resource repository: -ResourceIsNullException.cs -ResourceNotFoundException.cs -ResourceRepositoryExceptions.cs	Excepciones para casos border inválidos que forman parte de la clase ResourceRepository.
.DataAccess.Exceptions.ProjectRepositoryExceptions	Carpeta de exceptions de project repository: -DuplicatedProject'sNameException.cs -ProjectNotFoundException	Excepciones para casos border inválidos que forman parte de la clase ProjectRepository.
.DataAccess.Exceptions.NotificationRepositoryExceptions	Carpeta de exceptions de notification repository: -InvalidNotificationOperationException -NotificationNotFoundException -NotificationRepositoryException	Excepciones para casos border inválidos que forman parte de la clase NotificationRepository.

Domain

Namespace	Clase	Responsabilidad
Domain.Exceptions.NotificationExceptions	Carpeta Notification.Exceptions: -NotificationException.cs -NotificationDescriptionException.cs	Excepciones para casos border e inválidos dentro de la clase Notification.cs
Domain.Exceptions.ProjectExceptions	Carpeta Project.Exceptions: -ProjectDescriptionException.cs	Excepciones para casos border e inválidos dentro de la clase

	-ProjectException.cs -ProjectNameException.cs -ProjectStartDateException.cs	Project.cs
Domain.Exceptions.ResourceExceptions	Carpeta Resource.Exceptions: -ResourceException.cs -ResourceNameException.cs -ResourceTypeException.cs	Excepciones para casos border e inválidos dentro de la clase Resource.cs
Domain.Exceptions.TaskExceptions	Carpeta Task.Exceptions: -TaskDescriptionException.cs -TaskDurationException.cs -TaskEndDateException.cs -TaskException.cs -TaskPreviousTaskException.cs -TaskResourceException.cs -TaskTitleException.cs	Excepciones para casos border e inválidos dentro de la clase Task.cs
Domain.Exceptions.UserExceptions	Carpeta User.Exceptions: -UserBirthdayException.cs -UserEmailException.cs -UserException.cs -UserFirstNameException.cs -UserLastNameException.cs -UserPasswordException.cs -UserRoleAlreadyExistsException.cs -UserRoleNotFoundException.cs -UserRolesInvalidAssignmentException.cs -UserTaskException.cs	Excepciones para casos border e inválidos dentro de la clase User.cs
Domain	Notification.cs	Representar una notificación con un estado de lectura y una descripción.
Domain	Project.cs	Representa un proyecto con un nombre, descripción y fecha de inicio. Permite agregar miembros, tareas y notificaciones al proyecto.
Domain	Resource.cs	Representa un recurso con un nombre, tipo y descripción. Tiene una propiedad Id.
Domain	RolesEnum.cs	Define un Rol de los tres posibles: AdminSystem, ProjectMember y Admin Project.
Domain	State.cs	Define un State de los tres posibles: TODO, DOING y DONE.
Domain	Task.cs	Representa una tarea en un

		proyecto, con atributos de título, descripción, fechas de inicio y fin, duración, y estado. También deja la gestión de tareas previas y simultáneas, y recursos asociados
Domain	User.cs	Representa a un usuario con nombre, apellido, correo electrónico, fecha de nacimiento, contraseña, roles y tareas asignadas. Gestiona roles y tareas asociadas al usuario,

Service

Namespaces	Clases	Responsabilidad
Service	UserService .cs	Maneja operaciones de usuarios como agregar, actualizar y obtener usuarios. Valida correos y contraseñas, convierte entre objetos UserDTO y User, y gestiona roles y tareas de los usuarios.
Service	TaskService.cs	Maneja operaciones sobre tareas de proyectos, como agregar, eliminar, actualizar y obtener tareas, además de calcular y actualizar el camino crítico del proyecto.
Service	ResourceService.cs	Gestiona recursos, valida permisos de administrador para modificaciones, y maneja la conversión entre ResourceDTO y Resource
Service	PasswordManager.cs	Valida, hash y verifica contraseñas. Utiliza SHA-256 para el hash, valida contraseñas con un patrón específico y proporciona una contraseña predeterminada.
Service	NotificationService.cs	Gestiona notificaciones en proyectos y usuarios. Permite obtener, agregar, eliminar y marcar como leídas las notificaciones, convirtiendo entre entidades Notification y NotificationDTO
Service	MemberPService.cs	Gestiona proyectos y tareas de miembros, validando roles y permisos para modificar tareas y acceder a proyectos.

Service	Login.cs	Gestiona el inicio y cierre de sesión, valida credenciales y roles del usuario.
Service	GanttService.cs	Convierte tareas y el camino crítico en datos para un diagrama de Gantt, calculando el progreso y creando enlaces entre tareas previas.
Service	CPMService.cs	Calcula las fechas de inicio y fin tempranas y tardías, el margen de tiempo holgura, y determina las tareas críticas. También maneja dependencias entre tareas y calcula la duración del proyecto.
Service	AdminSService.cs	Gestiona operaciones de administración del sistema, como crear, eliminar usuarios, cambiar contraseñas y asignar roles
Service	AdminPService.cs	Gestiona proyectos, miembros y tareas.
Service.Exceptions.UserServiceExceptions	Carpeta de exceptions del service user service: -InvalidUserEmailException -InvalidUserPasswordException -NoUsersFoundException -UserServiceException -UserServiceNotFoundException	Excepciones para casos border e inválidos dentro de la clase UserService.cs
Service.Exceptions.TaskServiceExceptions	Carpeta de exceptions del service task service: -TaskServiceException	Excepciones para casos border e inválidos dentro de la clase TaskService.cs
Service.Exceptions.ResourceServiceExceptions	Carpeta de exceptions del service de resource service: -NoResourceFoundException -ResourceServiceException	Excepciones para casos border e inválidos dentro de la clase ResourceService.cs
Service.Exceptions.NotificationServiceExceptions	Carpeta de exceptions del service de notification service: -NotificationNotFoundException -NotificationServiceException	Excepciones para casos border e inválidos dentro de la clase ResourceService.cs
Service.Exceptions.MemberServiceExceptions	Carpeta de exceptions del service de member p service: -TaskCantBeModifiedByUserException -UserHasNoProjectsException	Excepciones para casos border e inválidos dentro de la clase MemberService.cs
Service.Exceptions.LoginExceptions	Carpeta de exceptions del service de login service: -InvalidLoginCredentialsException	Excepciones para casos border e inválidos dentro de la clase LoginService.cs

	<ul style="list-style-type: none"> -LoginException -UserNotFoundException 	
Service.Exceptions.CPMServiceExceptions	<p>Carpeta de exceptions del service de cpm service:</p> <ul style="list-style-type: none"> -CircularDependencyException -CpmServiceException -CriticalPathCalculationException -EmptyTaskListException -InvalidTaskDependencyException -NullTaskListException 	Excepciones para casos border e inválidos dentro de la clase CPMSERVICE.cs
Service.Exceptions.AdminServiceExceptions	<p>Carpeta de exceptions del service de admin s service:</p> <ul style="list-style-type: none"> -AdminSServiceException -InvalidOldPasswordException -CriticalPathCalculationException -UnauthorizedAdminAccessException 	Excepciones para casos border e inválidos dentro de la clase AdminSService.cs
Service.Exceptions.AdminPServiceExceptions	<p>Carpeta de exceptions del service de admin p service:</p> <ul style="list-style-type: none"> -AdminSServiceException -TaskIsNotFromTheProjectException -UserIsAlreadyAMemberException -UserIsNotAMemberException 	Excepciones para casos border e inválidos dentro de la clase AdminPService.cs
Service.Models	<p>Carpeta que contiene todos los dto de service:</p> <ul style="list-style-type: none"> -CPMResultDTO -GanttData -GanttLink -GanttTask -LoggedUser -NotificationDTO -ProjectDTO -ResourceDTO -RoIDTO -StateDTO -TaskDTO -UserDTO -UserLoginDTO 	Los archivos que se encuentran en la carpeta Models con el sufijo DTO representan objetos de transferencia de datos (Data Transfer Objects) que se utilizan para transportar datos entre capas del sistema.

Service.Interface	Carpeta que contiene todas las interfaces de service: -IAdminPService -IAdminSService -ILogin -IMemberPService -IPasswordManager -IResourceService -IUserService	Aseguran que las clases proporcionen implementaciones específicas para las operaciones necesarias, sin definir cómo se ejecutan esos métodos
-------------------	---	--

Explicación de validaciones de negocio

Las validaciones de negocio se implementaron dentro de los servicios, donde se maneja la lógica de negocio para asegurar la coherencia del sistema. Estas validaciones se asignaron a los servicios para centralizar la lógica y garantizar que las reglas se apliquen de manera uniforme, sin depender de otros componentes.

Ejemplo: En UserService, antes de agregar un usuario, se valida que el email no esté duplicado y que la contraseña cumpla con los requisitos de seguridad. Si alguna validación falla, se lanza una excepción para evitar registros incorrectos.

Mantenibilidad y extensibilidad

Uno de los aspectos clave para asegurar mantenibilidad y extensibilidad es la implementación de inyección de dependencias y el uso de DTOs para la transferencia de datos.

Un ejemplo es el siguiente:

Agregar un nuevo Rol a usuario.

Ej: quiero agregar el rol Manager

```
{
  namespace Domain
  {
    81 usages  nbs282 *  4 exposing APIs
    public enum Rol
    {
      AdminSystem,
      ProjectMember,
      AdminProject,
      Manager // Nuevo rol añadido
    }
  }
}
```

Luego se debería ajustar en UserDTO y en UserService.

```
private List<Rol> ConvertToDomainRoles(List<RolDTO> roleDTOs)
{
    var roles = new List<Rol>();

    foreach (var roleDTO in roleDTOs)
    {
        switch (roleDTO)
        {
            case RolDTO.AdminSystem:
                roles.Add(Rol.AdminSystem);
                break;
            case RolDTO.ProjectMember:
                roles.Add(Rol.ProjectMember);
                break;
            case RolDTO.AdminProject:
                roles.Add(Rol.AdminProject);
                break;
            case RolDTO.Manager: // Agregado el rol Manager
                roles.Add(Rol.Manager);
                break;
        }
    }

    return roles;
}
```

La UI no necesita realizar cambios significativos, ya que solo se han modificado los servicios y los DTOs, que continúan funcionando de manera consistente con el sistema.

La base de datos o el almacenamiento en memoria de usuarios simplemente necesita ser adaptado para manejar este nuevo rol sin afectar la estructura existente.

Cómo usamos servicios y DTOs desacoplamos la lógica de negocio del front-end, lo que facilita la modificación de funcionalidades sin alterar las demás partes del sistema. También agregar nuevos roles, funciones o métodos en los servicios, el impacto se limita solo a las capas correspondientes, sin necesidad de tocar todo el sistema.

-Git

Gitflow

Para implementar Gitflow en nuestro proyecto, seguimos los pasos básicos que vimos en clase y los adaptamos a nuestras necesidades. Primero, nos aseguramos de tener las dos ramas base que son fundamentales en Gitflow: **develop** y **main**.

Luego, cada vez que comenzamos a trabajar en una nueva funcionalidad, creamos una rama a partir de develop. A cada rama le asignamos un nombre que refleja el tipo de tarea

que estamos realizando. Por ejemplo, si estábamos trabajando en una nueva característica, la rama se creaba con el prefijo **feat/**, seguido de una descripción breve de la funcionalidad (ejemplo: feat/login).

Lo mismo ocurrió cuando teníamos que hacer correcciones, ahí usamos el prefijo **fix/** para las ramas de corrección de errores, como **fix/login-bug**, por ejemplo.

Una vez terminada la funcionalidad en cada rama, realizamos un merge de la misma de vuelta a develop. Esto nos aseguró que las nuevas características o correcciones se integren al flujo de trabajo de manera controlada y sin generar conflictos en main.

Es importante destacar que las ramas feature nunca interactúan directamente con main. En lugar de eso, creamos ramas release a partir de develop cuando estábamos listos para integrar las características y preparar una nueva versión.

Este flujo de trabajo nos ayudó a organizar mejor el desarrollo y asegurarnos de que todo el código estuviera en la rama correcta antes de pasar a producción, además de facilitarnos la colaboración entre nosotros.

-Test

Cobertura De pruebas

Nos aseguramos de la calidad de nuestro sistema cumpliendo con el objetivo de tener al menos un 90% de cobertura de pruebas en todas las capas del backend, incluyendo dominio, data access y services.

Las pruebas de dominio, que cubren la lógica principal del proyecto, fueron hechas para garantizar que todas las funcionalidades críticas fueran verificadas. Lo mismo para las pruebas en las clases de Repository que manejan las listas de objetos del dominio.

Por otro lado service, que contienen la lógica para interactuar con otras capas y coordinar el flujo de la aplicación, también nos aseguramos de que estuvieran cubiertas al 90%.

Para lograr la cobertura de pruebas escribimos pruebas unitarias robustas utilizando TDD (Test-Driven Development) en cada uno de estos componentes. Lo que nos permitió verificar que cada función individual cumpliera con su propósito, asegurándonos de que las clases que interactúan con la “base de datos” y los servicios fueran probadas.

Cabe destacar que las partes del frontend, como las clases con ramas que tienen el sufijo -ui y -pov, no fueron cubiertas con pruebas de backend, ya que están relacionadas con la interfaz de usuario, pero sí fueron manejadas dentro del marco de desarrollo de la interfaz.

De esta forma, garantizamos que tanto el dominio, los servicios, como las capas de acceso a los datos estuvieran correctamente testeadas y cubiertas.

La foto del coverage de las pruebas se encuentra en el Anexo.

Pruebas Funcionales De ABM

- <https://youtu.be/D0FBIK4BS7Q>

-TDD

Seguimos el ciclo de Red-Green-Refactor, que nos permitió mantener un desarrollo organizado y asegurar que cada funcionalidad estuviera probada antes de implementarla.

Primero siempre hicimos el RED. Donde implementamos únicamente el test para una nueva funcionalidad sin preocuparnos por la implementación que hiciera que la prueba pasará. El objetivo aquí era asegurarnos de que el test fallara, confirmando que no teníamos esa funcionalidad implementada aún. Una vez hecho esto, realizábamos el primer commit con el mensaje -m "red: información del test".

Después, pasábamos a la fase GREEN, donde implementábamos lo mínimo necesario para que el test pasara. Esto significaba escribir el código justo lo suficiente para que el test fuera exitoso. Al completar esta fase, realizábamos un nuevo commit con el mensaje -m "green: información de la implementación".

Finalmente Refactor, donde, tras haber visto el código funcionando, nos dábamos cuenta de que a veces podíamos mejorar la implementación. Esto podía a veces nos pasaba en el mismo momento de la implementación o después de un tiempo al revisar el código. En esta fase, si detectábamos oportunidades de mejora, hacíamos un refactor en el código, manteniendo su funcionalidad intacta (pasando la prueba como antes), pero optimizándolo. Este paso lo marcábamos con el commit -m "refactor: descripción de la mejora".

Este ciclo de Red-Green-Refactor lo aplicamos de forma consistente en cada una de las ramas del backend.

También implementamos excepciones personalizadas en lugar de utilizar excepciones genéricas como `ArgumentException`.

Lo que nos permitió un mejor control sobre los errores que pueden aparecer en las distintas capas del backend, tirando mensajes de error más claros y específicos.

Un ejemplo de esto es la clase `ProjectException`, que sirve como base para otras excepciones más específicas dentro del contexto del dominio de proyectos. Les mostramos un ejemplo de cómo la hicimos:

```

namespace Domain.Exceptions
{
    3 usages 3 inheritors nbs282
    public class ProjectException : Exception
    {
        3 usages nbs282
        public ProjectException(string message) : base(message) { }

        nbs282
        public override string ToString()
        {
            return $"ProjectException: {this.GetType().Name} - {this.Message}";
        }
    }
}

```

A partir de esta excepción base, creamos excepciones más específicas como la `ProjectNameException`, que maneja el error cuando el nombre de un proyecto no es válido

```

namespace Domain.Exceptions
{
    2 usages nbs282
    public class ProjectNameException : ProjectException
    {
        1 usage nbs282
        public ProjectNameException()
            : base("Project name cannot be null, empty, or whitespace.") { }
    }
}


```

Estas excepciones personalizadas son fáciles y más claras de probar lo que nos facilitó el ciclo de **TDD**, en lugar de poner `ArgumentException` a todo podemos distinguir cual es excepción del nombre, de la descripción o de cualquier otro atributo de la clase.


Cada una de las excepciones personalizadas fue probada como parte de los tests unitarios, para asegurarnos de que se lanzarán de manera correcta en los escenarios adecuados y que su manejo fuera el esperado.

Aquí dejamos un ejemplo de TDD hecho en Gestión de Tareas con Duración y Dependencias, específicamente en la parte de Definición de Tareas.

Green: LatestStart implemented


 SantiCanadell committed 3 days ago

red: LatestStart not implemented


 SantiCanadell committed 3 days ago

Commits on May 5, 2025


green: TaskResourceException implemented

 SantiCanadell committed last week


red: TaskResourceException not implemented

 SantiCanadell committed last week


green: TaskPreviousTaskException implemented

 SantiCanadell committed last week


red: TaskPreviousTaskException not implemented

 SantiCanadell committed last week


fix: id attribute in task

 SantiCanadell committed last week


update taskDomain

 SantiCanadell committed last week

green: implementation of Id_WhenSet_ThenIdsAssigned


 SantiCanadell committed last week

red: implementation of id

 SantiCanadell committed last week

Nosotros utilizamos un Id como clave primaria, y es la manera de identificar los tasks.

red: implementation of id

 SantiCanadell committed last week

```
Domain.Test/TaskTest.cs
@@ -139,9 +139,23 @@ public void State_WhenSet_ThenStateIsAssigned()
139 139     }
140 140
141 141
142 + [TestMethod]
143 + public void Id_WhenSet_ThenIdIsAssigned()
144 + {
145 +
146 +     int expectedId = 123;
147 +     List<Task> previousTasks = new List<Task>();
148 +     List<Task> sameTimeTasks = new List<Task>();
149 +     Task task = new Task("Title", "Description", DateTime.Now, 1, previousTasks, sameTimeTasks);
150
151 +
152 +     task.Id = expectedId;
153
154 +
155 +     Assert.AreEqual(expectedId, task.Id);
156 + }
157
158 +
159
160 }
161 }
```

En el primer commit se logra ver como se crea el test, sin implementar la Id en [Task.cs](#)

green: implementation of Id_WhenSet_ThenIdIsAssigned



SantiCanadell committed last week

```
1 file changed +1 -1 lines changed
Domain/Task.cs
@@ -96,7 +96,7 @@ public void RemoveSameTimeTask(Task task)
96 96     }
97 97     public State State { get; set; }
98 98
99 -
99 + public int? Id {get; set;}
100 100 }
101 101
102 102 }
```

En el segundo commit se ve como luego del red se implementa la funcionalidad en el Domain [Task.cs](#) y este ahora pasa la prueba con lo mínimo e indispensable.

-Buenas Prácticas (Mantenibilidad y extensibilidad)

Organización del Proyecto y Uso de Packages

Organizamos el proyecto en diferentes packages para asegurar una estructura clara y comprensible. Dividimos el proyecto en los siguientes paquetes:

- Domain: Contiene las entidades del dominio, como la clase Task.
- Service: Incluye la lógica de negocio y servicios que interactúan con las entidades del dominio.
- DataAccess: Maneja la persistencia y recuperación de datos.
- UI: Contiene la interfaz de usuario, que se comunica con las demás capas.

Cada una de estas capas tiene su correspondiente package de tests. Lo hicimos para mantener una separación clara de responsabilidades y facilitar la escalabilidad del proyecto, asegurando que cada capa estuviera correctamente testeada y sin dependencias circulares.

Uso de TDD (Test-Driven Development)

Nos permitió mantener un desarrollo organizado y garantizar que todas las funcionalidades fueran probadas antes de ser implementadas.

El ciclo de Red-Green-Refactor fue seguido rigurosamente para cada nueva funcionalidad.

Excepciones Personalizadas

Además, implementamos excepciones personalizadas en lugar de utilizar excepciones genéricas como ArgumentException. Nos sirvió para tener un control sobre los errores que podrían ocurrir en las diferentes capas del backend, proporcionando mensajes de error más específicos para cada tipo de excepción.

Por ejemplo, creamos excepciones como TaskTitleException y TaskResourceException para manejar errores relacionados con las tareas del proyecto.

Convenciones de Nombres

Seguimos las convenciones de nombres de C#, utilizando CamelCase para las variables privadas (por ejemplo, _title, _description) y PascalCase para las propiedades públicas (por ejemplo, Title, Description).

-Cálculo De Ruta Crítica

Hicimos un servicio para el cálculo de la ruta crítica basado en el método CPM (Critical Path Method), desarrollado en la clase CpmService. Lo que hace es analizar las tareas de un proyecto, calcular sus fechas tempranas, tardías, y determinar las tareas críticas, las que no pueden retrasarse sin afectar la duración total del proyecto. La lógica de CPM se ejecuta mediante diversas funciones que se encargan de procesar las tareas del proyecto, estas son las funciones principales que implementamos:

1. Fechas Tempranas:

- La función `CalculateEarlyDates(List<Task> tasks)` calcula las fechas más tempranas de inicio (`StartDate`) y finalización (`EndDate`) para cada tarea. Si una tarea no tiene tareas anteriores devuelve la fecha esperada (`ExpectedStartDate`). Si tiene tareas anteriores, la fecha de inicio será la fecha de finalización de la tarea más tardía de los predecesores.

2. Fechas Tardías:

- La función `CalculateLateDates(List<Task> tasks)` calcula las fechas más tardías de inicio (`LatestStart`) y finalización (`LatestFinish`) para cada tarea sin afectar la duración total del proyecto. Empieza con las tareas finales (que no tienen tareas sucesoras) y luego se recorren las demás tareas en orden inverso, hacemos esto para asegurar que ninguna tarea sucesora se programe después de su fecha más tardía.

3. Slack (Holgura) y Tareas Críticas:

- La función `CalculateSlackAndCriticalTasks(List<Task> tasks)` calcula la holgura (`Slack`) de cada tarea. El `Slack` es la diferencia entre la fecha de inicio más tardía y la fecha de inicio más temprana. Si el `Slack` de una tarea es igual a cero o cercano a cero, hacemos que la tarea se marque como crítica. Son aquellas que su demora afectaría directamente la fecha de finalización del proyecto.

4. Ruta Crítica:

- La función `FindCriticalPath(List<Task> tasks)` determina la ruta crítica del proyecto. Esta ruta está formada por las tareas críticas, las que no tienen `Slack` o tienen un `Slack` muy bajo. La ruta crítica se construye iterando sobre las tareas críticas y uniando las tareas sucesoras.

5. Duración del Proyecto:

- La función `CalculateProjectDuration(List<Task> tasks)` calcula la duración total del proyecto, que es la diferencia entre la fecha de inicio más temprana de las tareas y la fecha de finalización más tardía. Esta duración nos sirve para determinar los plazos de entrega y las metas del proyecto.

El cálculo se hace al ejecutar el método `CalculateCriticalPath(List<Task> tasks)`, que invoca estas funciones y maneja la lógica de cálculo. Este método devuelve un objeto `CpmResult` que contiene:

- AllTasks: Todas las tareas del proyecto con sus respectivas fechas, Slack y estados de criticidad.
- CriticalPath: La lista de tareas que forman la ruta crítica del proyecto.
- CriticalTasks: Las tareas críticas, las que tienen Slack igual a cero.
- ProjectDuration: La duración total del proyecto en días.

Este servicio facilita la integración de la información de la ruta crítica con otras herramientas de visualización, como los diagramas de Gantt, para ofrecer una representación gráfica del progreso del proyecto y sus tareas más importantes.

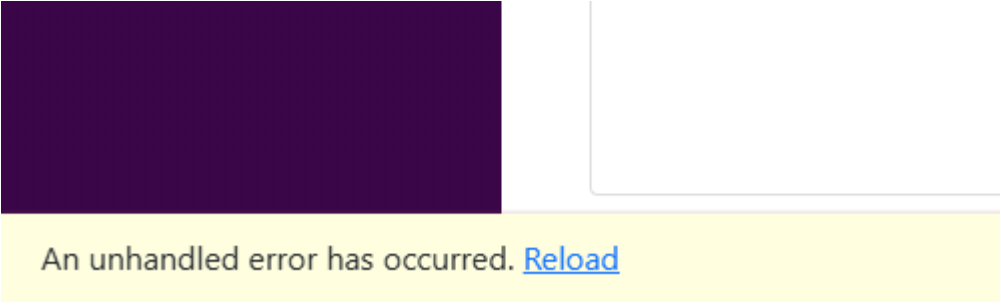
-Diagrama de Gantt

Se implementó un diagrama de Gantt utilizando la librería DHTMLX Gantt para visualizar de forma gráfica las tareas de cada proyecto y sus dependencias. Además permite mostrar el camino crítico de forma gráfica, permitiendo identificar de forma clara las tareas cuya demora afecta directamente la duración total. La lógica se basa en el cálculo CPM (Critical Path Method), desarrollado en la clase CpmService, que analiza las fechas tempranas, tardías y la holgura de cada tarea para determinar cuáles son críticas. Estas tareas se convierten a formato JSON compatible con la librería mediante la clase GanttService, y se renderizan en una página Blazor (Gantt.razor) usando JavaScript. El gráfico incluye todas las tareas de un proyecto, mostrando su progreso, inicio, duración y relación con otras. Además al darle a un botón permite la visualización del camino crítico resaltando las tareas que lo componen con color rojo. La integración funciona completamente offline gracias a la inclusión local de los archivos dhtmlxgantt.js y dhtmlxgantt.css.

3) Problemas Enfrentados

- Bugs conocidos

El único bug que logramos encontrar en el programa es que a veces cuando entramos al Gantt Diagram y se elige el proyecto para visualizar el diagrama. Aparece en la parte inferior izquierda de la página que pide hacer un reload, al apretar en reload, se elige el proyecto a visualizar y se ve todo correctamente.



An unhandled error has occurred. [Reload](#)

- **Funcionalidades no implementadas**

Luego una cosa que nos faltó implementar fue la fecha de ejecución de una tarea al ser terminada por un usuario. Hicimos que una tarea pueda ser marcada como DONE y que una vez que sea terminada recién pueda ser accedida por su sucesor si es que tiene. Pero nos faltó poner esa fecha en la cual fue terminada.

También el tema de recursos sentimos que en la letra estaba un poco ambiguo por lo que nosotros decidimos no tomar los recursos para calcular el CPM ni tomar en cuenta los recursos para completar las tareas.

Las mencionadas anteriormente las penamos tomar en cuenta para la siguiente parte del obligatorio ser arregladas

Las cuestiones mencionadas anteriormente las tomaremos en cuenta para corregirlas en la siguiente parte del obligatorio.

4) Anexo

- **Cálculo De Ruta Crítica: TDD**

Cómo hicimos para las demás partes del proyecto, implementamos el CPMSERVICE utilizando TDD , regido por las etapas red, green y refactor. Esto en la clase CPMSERVICETest.

Pruebas de Excepciones:

Las hicimos para que el sistema maneje correctamente los casos donde los datos de entrada son inválidos o no cumplen con lo pedido en la letra. Se validan las excepciones que deben lanzarse.

Un ejemplo es CalculateCriticalPath_ShouldThrowException_WhenTasksListIsNull verifica que el sistema lance una NullTaskListException cuando se pasa una lista de tareas nula al método CalculateCriticalPath.

Pruebas de Cálculo de Fechas Tempranas:

Estas pruebas verifican que el cálculo de las fechas de inicio y finalización tempranas sea correcto. También la verificación de tareas sin dependencias (que deberían

comenzar en la fecha esperada) y tareas que dependen de otras (que deberían iniciar después de las tareas previas).

Por ejemplo:

`CalculateCriticalPath_ShouldCalculateEarlyDates_ForTaskWithPredecessors` valida que las fechas de inicio y finalización se calculen correctamente para una tarea que depende de otra. La tarea C debe empezar después de la tarea A, y la fecha de finalización de la tarea C debe ser la fecha de inicio de C + su duración.

Pruebas de Cálculo de Fechas Tardías y Slack:

Estas verifican que el sistema calcule las fechas tardías de inicio y finalización correctamente, como el cálculo del Slack de cada tarea. Las tareas críticas deben tener un Slack de 0, y las tareas no críticas deben tener un margen de retraso.

Un ejemplo es `CalculateCriticalPath_ShouldCalculateLateDates_AndSlack` valida que las tareas con dependencias calculen correctamente sus fechas tardías, y además, verifica que el Slack de las tareas críticas sea 0, indicando que no hay margen para retraso. Si una tarea tiene Slack mayor a 0, no es crítica.

Pruebas de Ruta Crítica:

Estas pruebas validan que el algoritmo forme correctamente la ruta crítica, las tareas que su retraso afectará la duración total del proyecto. Hicimos casos de prueba con diferentes estructuras, incluyendo tareas paralelas y dependencias complejas.

Un ejemplo es

`CalculateCriticalPath_ShouldIdentifyCriticalPath_WithComplexDependencies` verifica que la ruta crítica se cree correctamente en un conjunto de tareas con dependencias complejas. La función debe identificar qué tareas son críticas y cuáles no, basándose en las fechas de inicio y finalización calculadas y en las relaciones entre tareas.

Pruebas de Duración del Proyecto:

Aseguran que la duración total del proyecto se calcule correctamente. Esto incluye sumar las duraciones de las tareas y tener en cuenta las dependencias entre ellas para determinar el tiempo total requerido para completar el proyecto.

Por ejemplo, la prueba `CalculateProjectDuration_HandlesEmptyList` valida que si solo hay una tarea en el proyecto, su duración sea correctamente calculada como el tiempo de la tarea misma. Si hay múltiples tareas, la duración total debe considerar las dependencias y el orden de ejecución de las tareas.

- Dependencias Externas

Utilizamos la librería DHTMLX Gantt para hacer el diagrama de Gantt. La descargamos e incluimos al proyecto para hacer que funcione de forma local, permitiendo así acceder al diagrama sin conexión a internet.

Usamos bootstrap para la parte de front-end, principalmente para hacer la página responsive y que se vea bien en todos los dispositivos. También la utilizamos para la parte de los iconos, de manera que la pagina se vea mas estética, amena y entendible para el usuario final.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css">
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.1/font/bootstrap-icons.css">
```

- Uso de IA

Usamos inteligencia artificial para corroborar alguna de las funcionalidades implementadas, para lo que más nos ayudó fue para entender cómo funcionaba un CPM y para implementar partes como el password manager:

<https://chatgpt.com/share/682603dc-50d0-800b-8cb7-47df4307b627>

<https://chatgpt.com/share/e/68260df7-d694-8007-982c-f629f676c058>

<https://chatgpt.com/share/68264e49-f7e8-800e-a717-f915c3c0a145>

Para la parte de CPM primero lo hablamos con chat gpt para entender las bases del mismo y cuales son sus funciones principales pero quien más nos dio una mano con la implementación fue la ia Claude, nos pusimos a buscar el chat en el que nos ayudó con el CPM pero no lo encontramos.

En la parte del diagrama de Gantt se puede ver cómo lo utilizamos para buscar la librería e implementarlo en nuestra app.

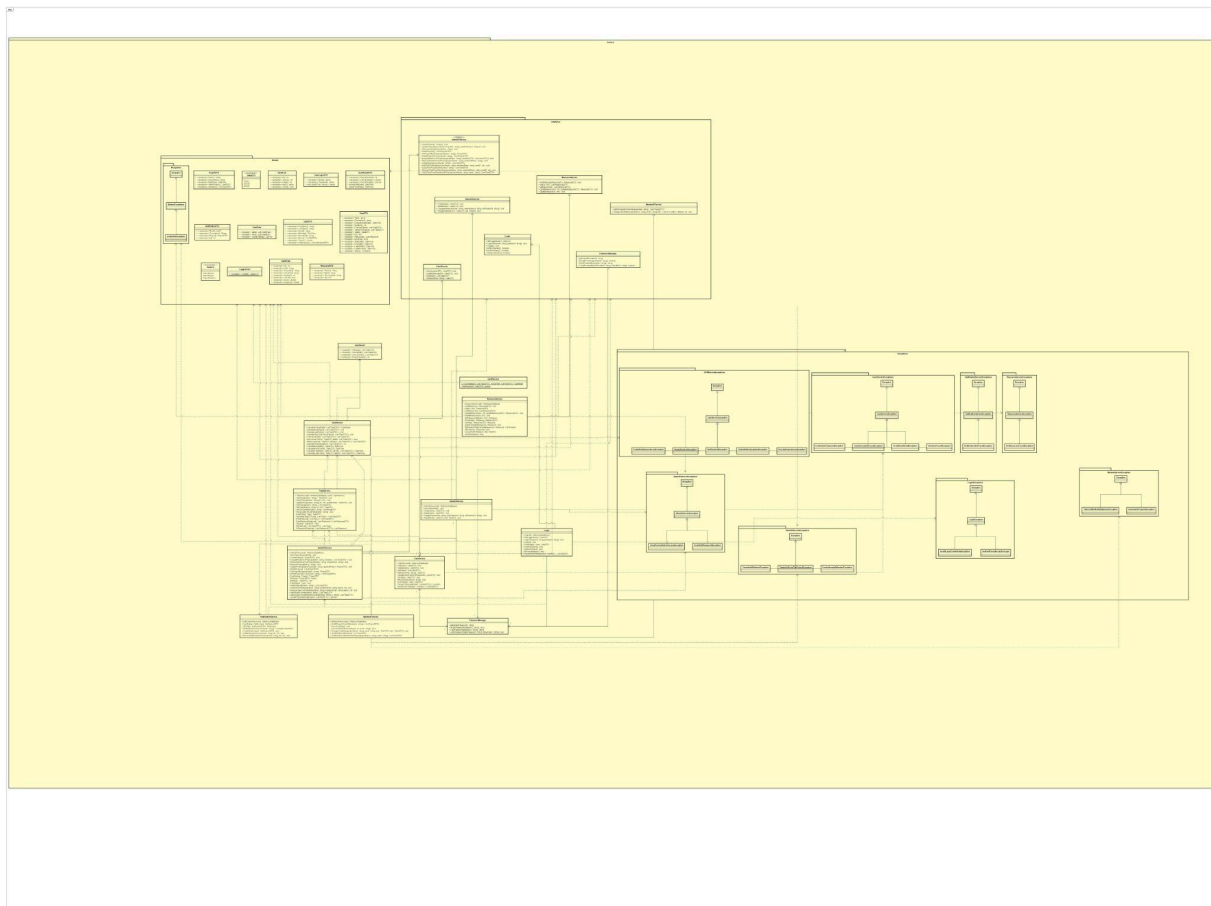
- Foto coverage de pruebas

Type to search		
Symbol	Coverage (%) ▾	Uncovered/Total Stmts.
▼ Total	94%	273/4659
> Service.Test	97%	56/1816
> DataAccess.Tests	95%	20/386
> Domain.Test	94%	29/477
> Domain	94%	21/344
> Service	91%	120/1371
> DataAccess	90%	27/265

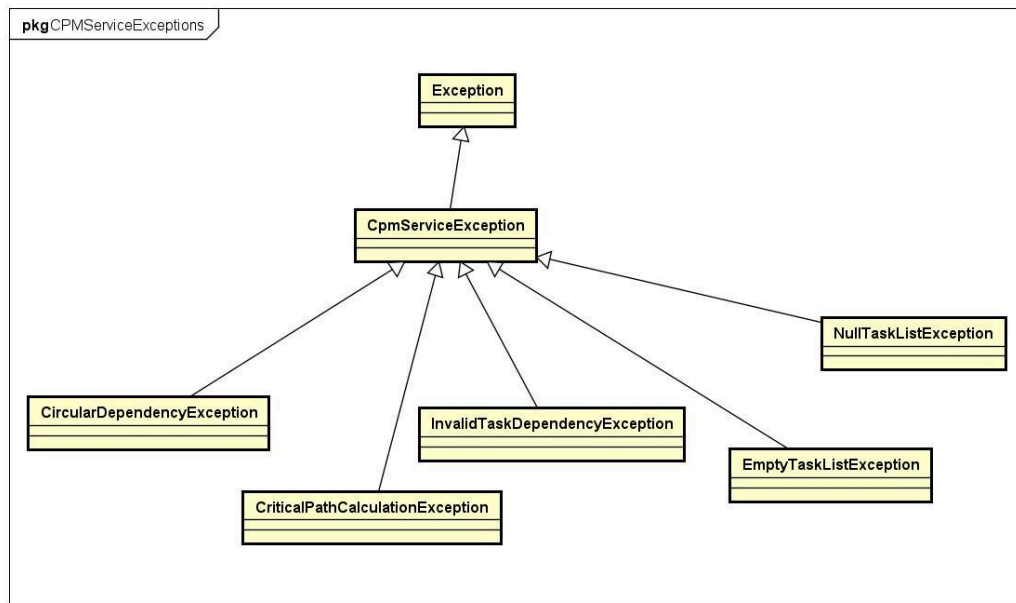
- Fotos UML

SERVICE

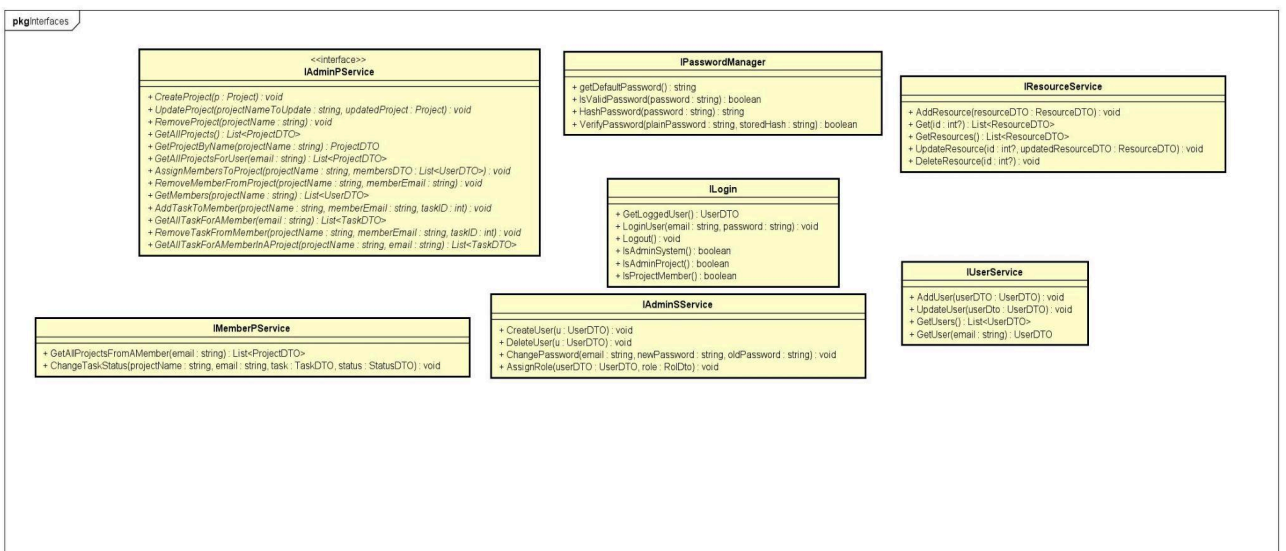
Diagrama packages



Ejemplo de exceptions



Interfaz



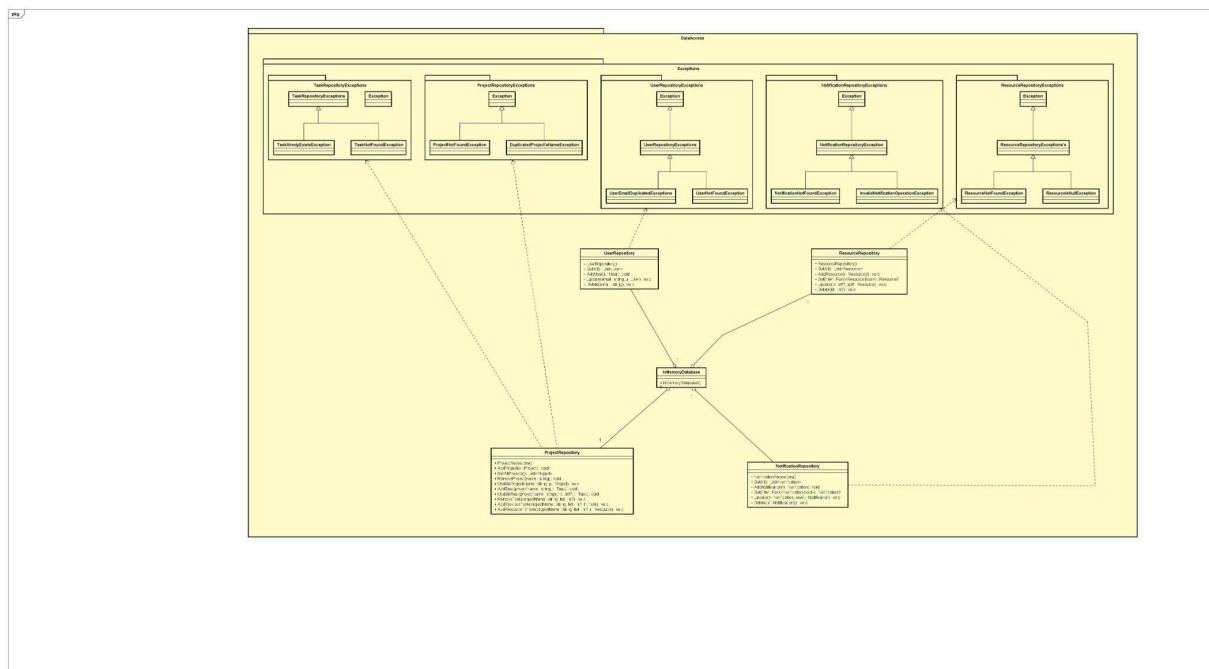
[illegible]

```

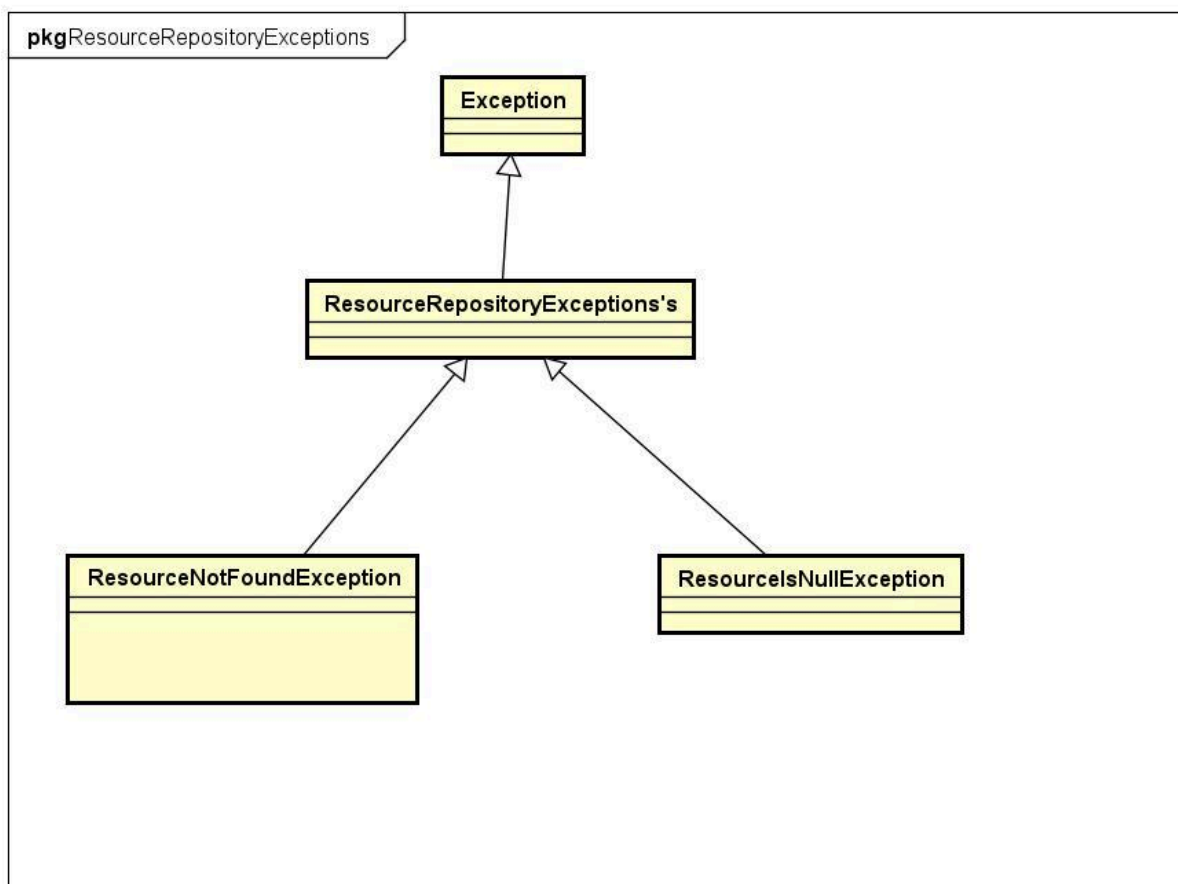
classDiagram
    class CpmResultDTO {
        +ProjectDuration() int
        +CriticalPathIds() List<int>
        +CriticalTasks() List<int>
        +EarliestStartDate() DateTime
        +LatestFinishDate() DateTime
    }
    class GanttLink {
        +Id() int
        +Source() int
        +Target() int
        +Type() string
        +Critical() bool
    }
    class GanttTask {
        +Text() string
        +StartDate() string
        +EndDate() string
        +Duration() int
        +Critical() bool
        +Slack() double
        +Progress() double
    }
    class GanttData {
        +Data() List<GanttTask>
        +Links() List<GanttLink>
        +CriticalPathIds() List<int>
    }
    class ProjectDTO {
        +Name() string
        +Description() string
        +StartDate() DateTime
        +AdminProject() UserLoginDTO
        +Members() List<UserDTO>
    }
    class ResourceDTO {
        +Name() string
        +Type() string
        +Description() string
        +Id() int
    }
    class TaskDTO {
        +Title() string
        +Description() string
        +ExpectedStartDate() DateTime
        +Duration() int
        +PreviousTasks() List<TaskDTO>
        +SameTimeTasks() List<TaskDTO>
        +State() StateDTO
        +Id() int
        +Resources() List<Resource>
        +IsCritical() bool
        +StartDate() DateTime
        +EndDate() DateTime
        +LatestStart() DateTime
        +LatestFinish() DateTime
        +Slack() TimeSpan
    }
    class LoggedUser {
        +Current() UserLoginDTO
    }
    class NotificationDTO {
        +Read() bool
        +Description() string
        +Project() ProjectDTO
        +Id() int
    }
    class UserLoginDTO {
        +Email() string
        +Password() string
        +LoginDTO(e string, p string)
    }
    class UserDTO {
        +FirstName() string
        +LastName() string
        +Email() string
        +Birthday() DateTime
        +Password() string
        +Roles() List<RoleDTO>
        +Tasks() List<int>
        +Notifications() List<NotificationDTO>
    }
    class RoleDTO {
        AdminSystem
        AdminProject
        ProjectMember
    }
    class StateDTO {
        TODO
        DONE
    }

    CpmResultDTO --> GanttData
    GanttData --> GanttLink
    GanttData --> GanttTask
    GanttLink --> GanttTask
    GanttTask --> GanttData
    ProjectDTO --> GanttData
    ProjectDTO --> UserLoginDTO
    ProjectDTO --> UserDTO
    ResourceDTO --> GanttTask
    TaskDTO --> GanttTask
    TaskDTO --> UserLoginDTO
    TaskDTO --> UserDTO
    TaskDTO --> TaskDTO
    TaskDTO --> ResourceDTO
    TaskDTO --> StateDTO
    LoggedUser --> UserLoginDTO
    NotificationDTO --> ProjectDTO
    NotificationDTO --> UserLoginDTO
    NotificationDTO --> UserDTO
    UserLoginDTO --> UserDTO
    UserLoginDTO --> StateDTO
    UserDTO --> StateDTO
    
```

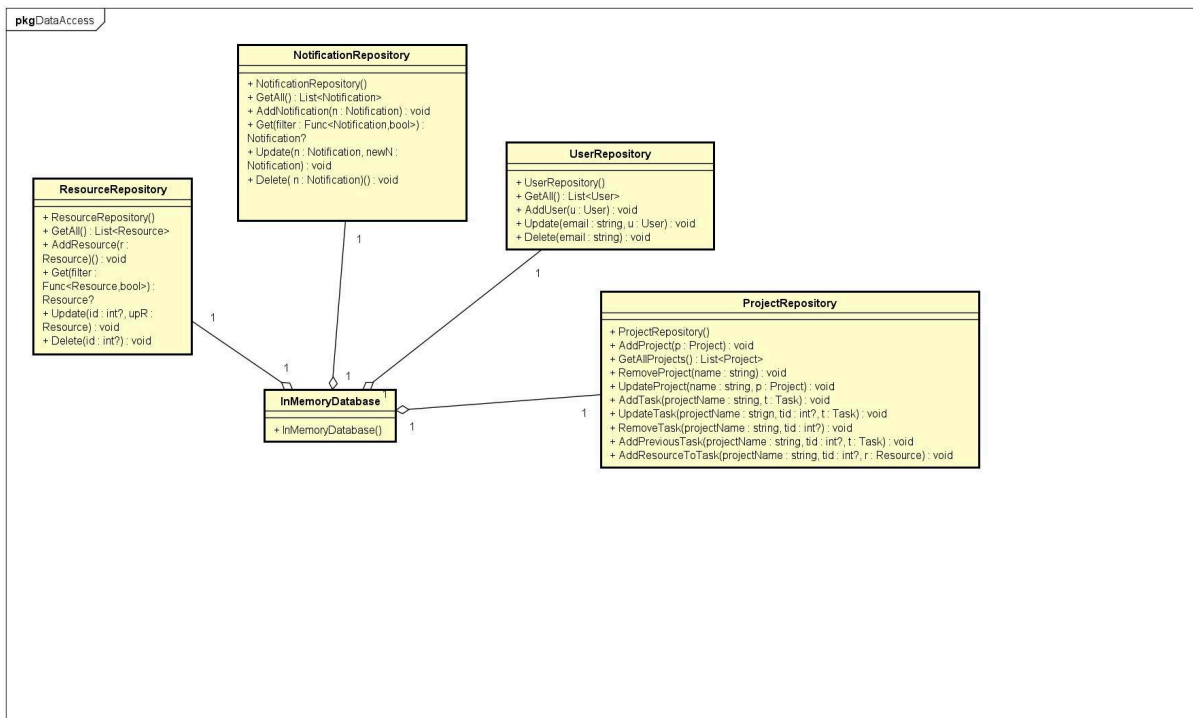
Diagrama packages



Ejemplo de exceptions

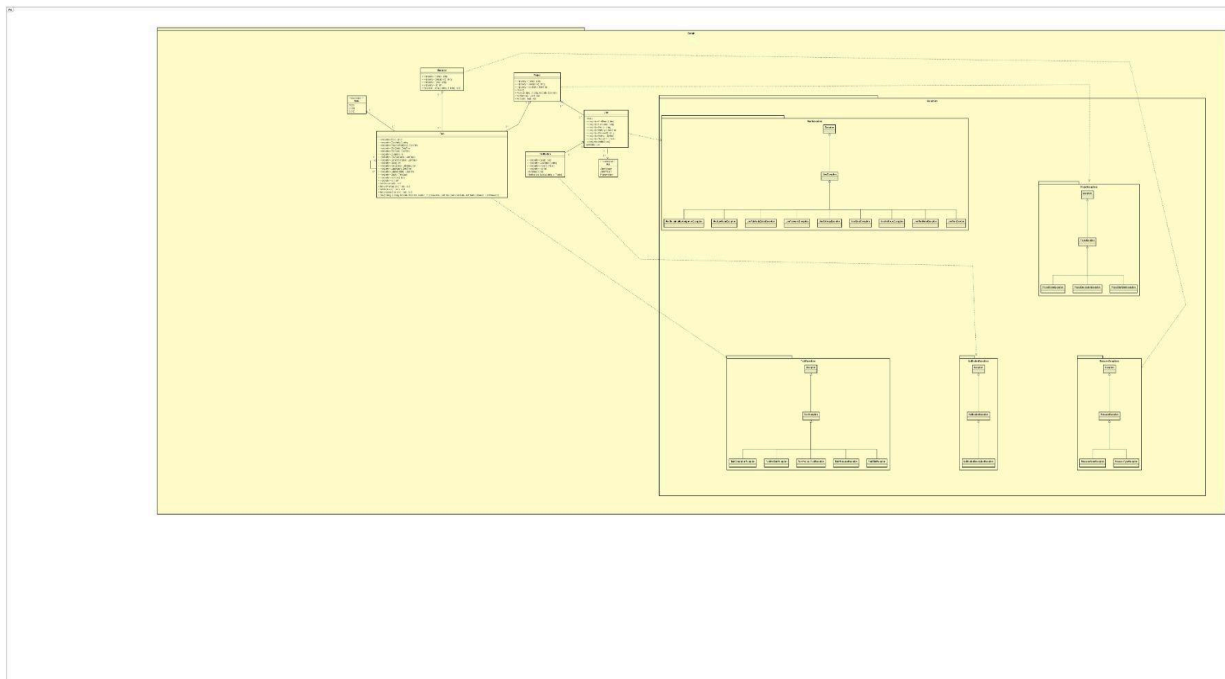


Data Access

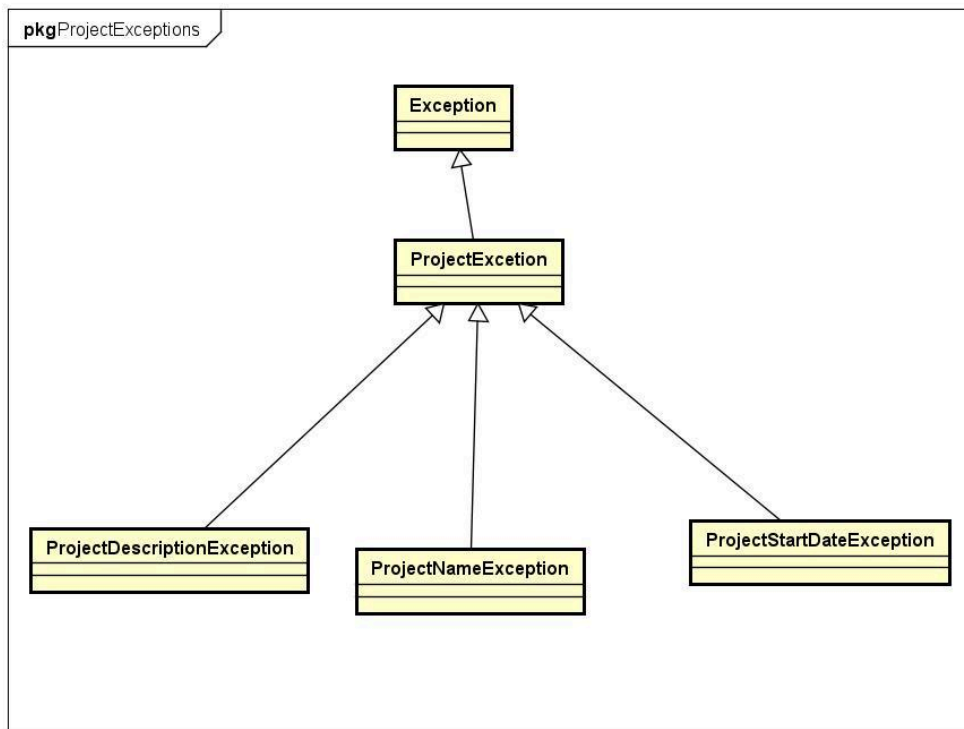


DOMAIN

Diagrama packages



Ejemplo de exceptions



Domain

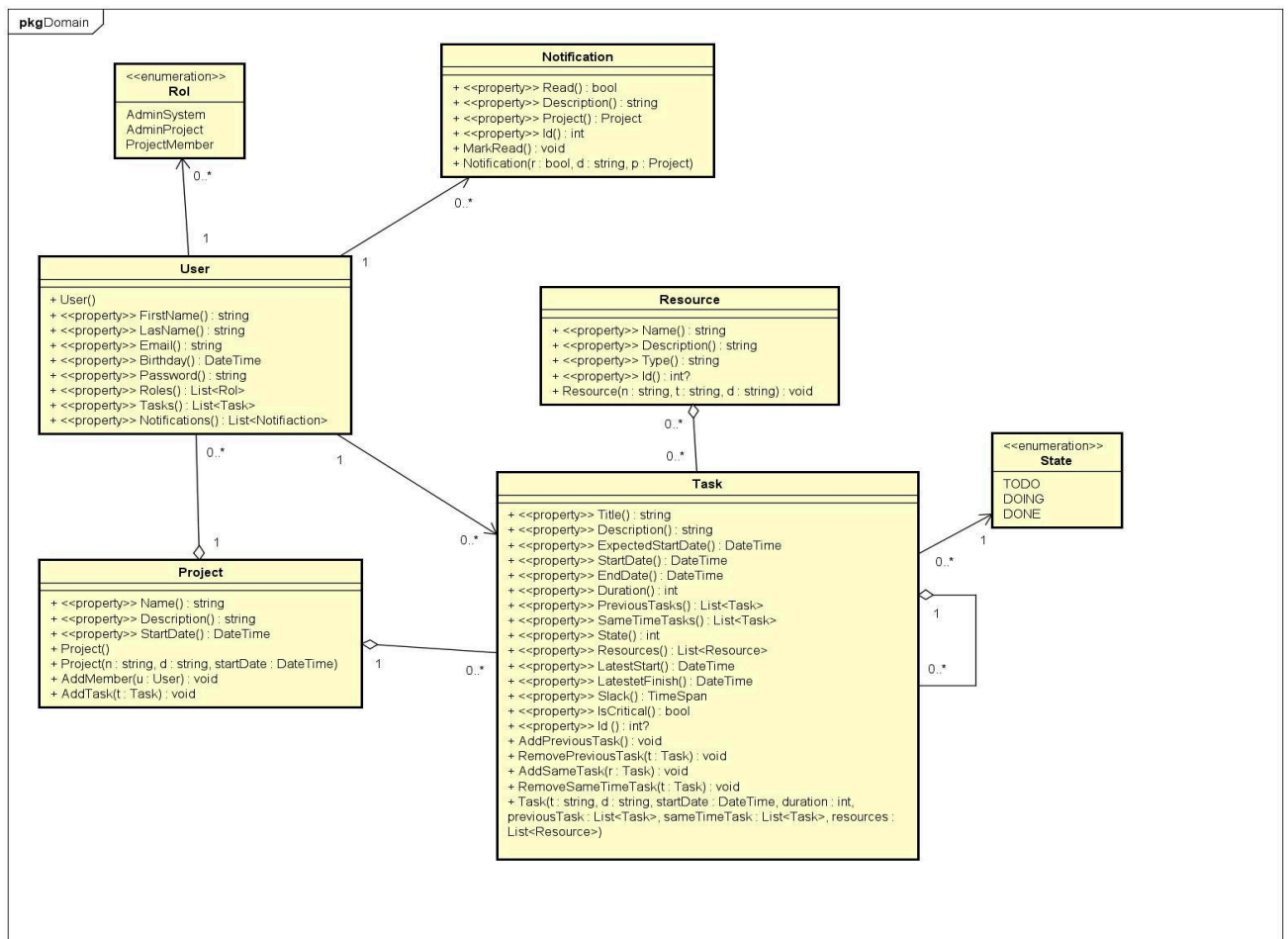
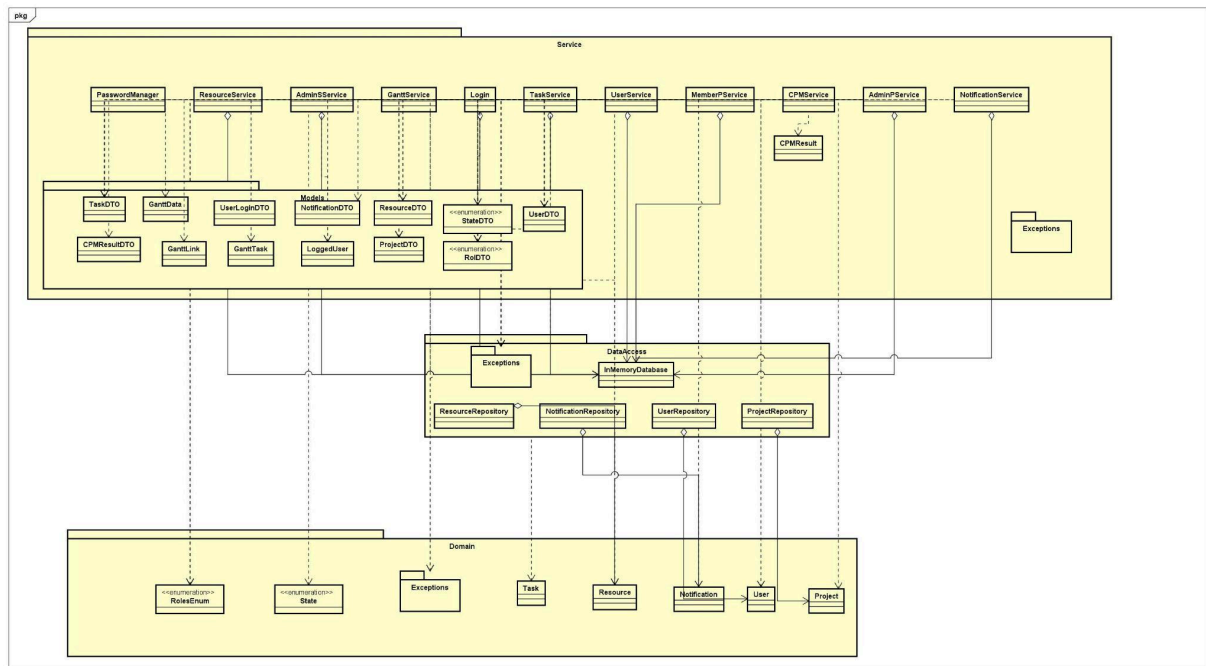


DIAGRAMA DE PAQUETES (todo el proyecto)



- Link UML

<https://drive.google.com/drive/folders/1L778X5ZPe7p-4eQXbsPLbZP8HYq-tpQN>