

Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 2

Diseño de aplicaciones 1

Fecha:16/11/2023

Maximiliano Giménez - 294598

Ana Betina Kadessian - 221509

Mateo Arias - 274857

Docentes: Diego Nicolás Balbi Alves

Docente: Guzmán Vigliecca Frank

Grupo: M5A

[Repositorio Github](#)

Índice

Descripción general.....	3
Justificación de diseño	4
Diagramas de paquetes:	4
Diagramas de clases	6
Diagrama de clases de paquete Dominio.....	6
Diagrama de clases de paquete EspacioReporte	8
Diagrama de clases de paquete Repositorio.....	9
Diagrama de clases de paquete LogicaNegocio	11
Diagrama de clases de paquete Excepcion	12
Diagrama de clases de paquete DTO:.....	12
Base de datos Entity Framework.....	13
Controladores e Interfaz	16
Detalle de cobertura	17
Diagramas de interacción	18
Anexo:.....	20
Evidencia de pruebas funcionales:	20
UML Interfaces Controladores.....	21

Descripción general

El propósito de este proyecto consistió en desarrollar una aplicación denominada "FinTrac" destinada a la gestión de finanzas personales. La aplicación se ha diseñado y construido de acuerdo con todos los requisitos establecidos en el enunciado del obligatorio.

Gitflow:

Se ha empleado el enfoque de Git Flow en todas las etapas del proyecto. Esta metodología permite trabajar en paralelo en diferentes funcionalidades de forma independiente, lo que facilita la creación de características por separado para luego integrarlas en el proyecto principal una vez que estén finalizadas.

El proceso comenzó en la rama principal, desde donde se creó una rama denominada "dev" destinada a la integración de nuevas características y mejoras. A partir de esta rama, se crearon nuevas ramas específicas para implementar diversas funcionalidades (features).

Posteriormente, se mantuvo una rama principal llamada "main" que sirvió para mantener una versión estable de la aplicación. La fusión de cambios en esta rama solo se realizó al finalizar un ciclo con todos los cambios.

Test Driven Development (TDD):

El Desarrollo Guiado por Pruebas (TDD) se aplicó en todas las partes del proyecto, a excepción de la interfaz de usuario. Este enfoque se basa en la creación de pruebas automatizadas antes de escribir el código. El proceso se divide en tres fases:

- 1)Escribir una prueba (Fase "Red"): En esta etapa, se diseñaron pruebas que verifican el comportamiento esperado de una característica o componente.
- 2)Escribir el código (Fase "Green"): Luego de definir las pruebas, se procedió a escribir el código que cumple con los requisitos planteados por las pruebas.
- 3)Refactorizar el código (Fase "Refactor"): La fase final involucra la optimización y mejora del código escrito, garantizando su calidad y legibilidad.

Este enfoque permitió desarrollar la aplicación de manera incremental y garantizar su correcto funcionamiento al haber sido sometida a pruebas exhaustivas en cada etapa del proceso.

Clean Code:

En el proyecto, se ha dado importancia a los principios de Clean Code, asegurando que el código sea legible, fácil de mantener y libre de redundancias. La aplicación de estos principios mejora la calidad del código, facilita su comprensión y reduce la posibilidad de errores, contribuyendo así a la eficiencia y sostenibilidad a largo plazo del proyecto.

Aplicamos varias prácticas de Clean Code. Por ejemplo, nos esforzamos por escribir funciones y métodos pequeños y enfocados, y por dar nombres descriptivos a nuestras variables y funciones.

Justificación de diseño

Diagramas de paquetes:

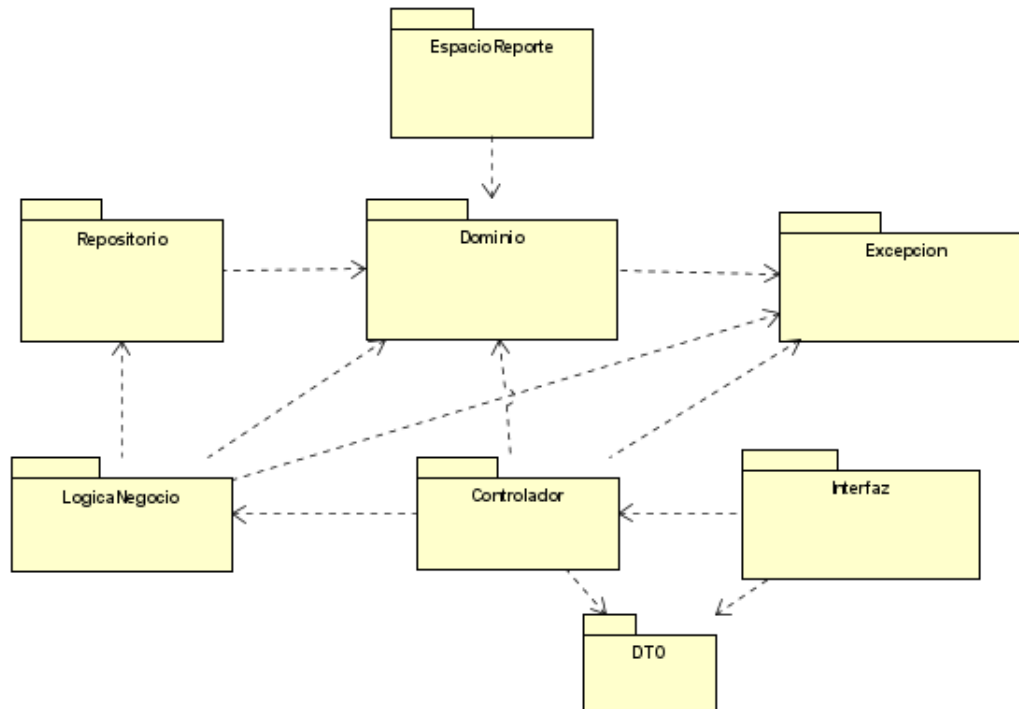


Imagen Diagrama 1.0

En el Diagrama 1.0 podemos apreciar el paquete Dominio contiene las clases de modelo o entidades que representan los datos con los que trabajará la aplicación, estas clases representan objetos de dominio o entidades del mundo real, representan los datos que guardamos en la base de datos.

En el paquete EspacioReporte generamos un conjunto de cuatro clases (Reporte, ObjetivoGasto, CategoriaGasto, IngresoEgreso) Reporte es una clase auxiliar que toma un atributo Espacio y a través de sus métodos definidos generan la funcionalidad requerida de la aplicación de analizar los datos de "Dominio", lo creamos en un paquete diferente para seguir los principios de modularidad, los cuales hacen el código más mantenible separando responsabilidades.

Como seguimos el patrón de diseño Repositorio creamos un paquete con dicho nombre el cual contiene una interface IRepository, que implementa un contrato con la lógica de un CURD.

El paquete "LogicaNegocio" es el motor detrás del funcionamiento de la aplicación. Define las reglas de negocio, donde se establecen y ejecutan procesos que permiten que la aplicación funcione de manera coherente y efectiva, por ejemplo, validación, registro de usuarios, obtener los espacios en los que se encuentra cada usuario etc.

El paquete Excepciones es donde manejamos todas las excepciones de nuestro sistema, estas nos aportan una forma adecuada para distinguir donde se encuentra el problema que pueda surgir, y realizar el manejo apropiado de las mismas.

En el paquete Interfaz se encuentra alojado nuestro servidor de blazor server con el conjunto de páginas que serán el front-end de la aplicación.

El paquete DTO contiene clases similares a las del dominio es para intercambiar datos entre la interfaz y controladores.

El paquete Controlador lo creamos para desacoplar el paquete de la interfaz de clases de dominio excepciones y LogicaNegocio siguiendo el patrón GRASP controlador, este se comunica con la interfaz por medio de clases de DTO, a pesar de que la clase program.cs queda acoplada a estas es una excepción el resto de la funcionalidad se desacopló.

Diagramas de clases

Diagrama de clases de paquete Dominio

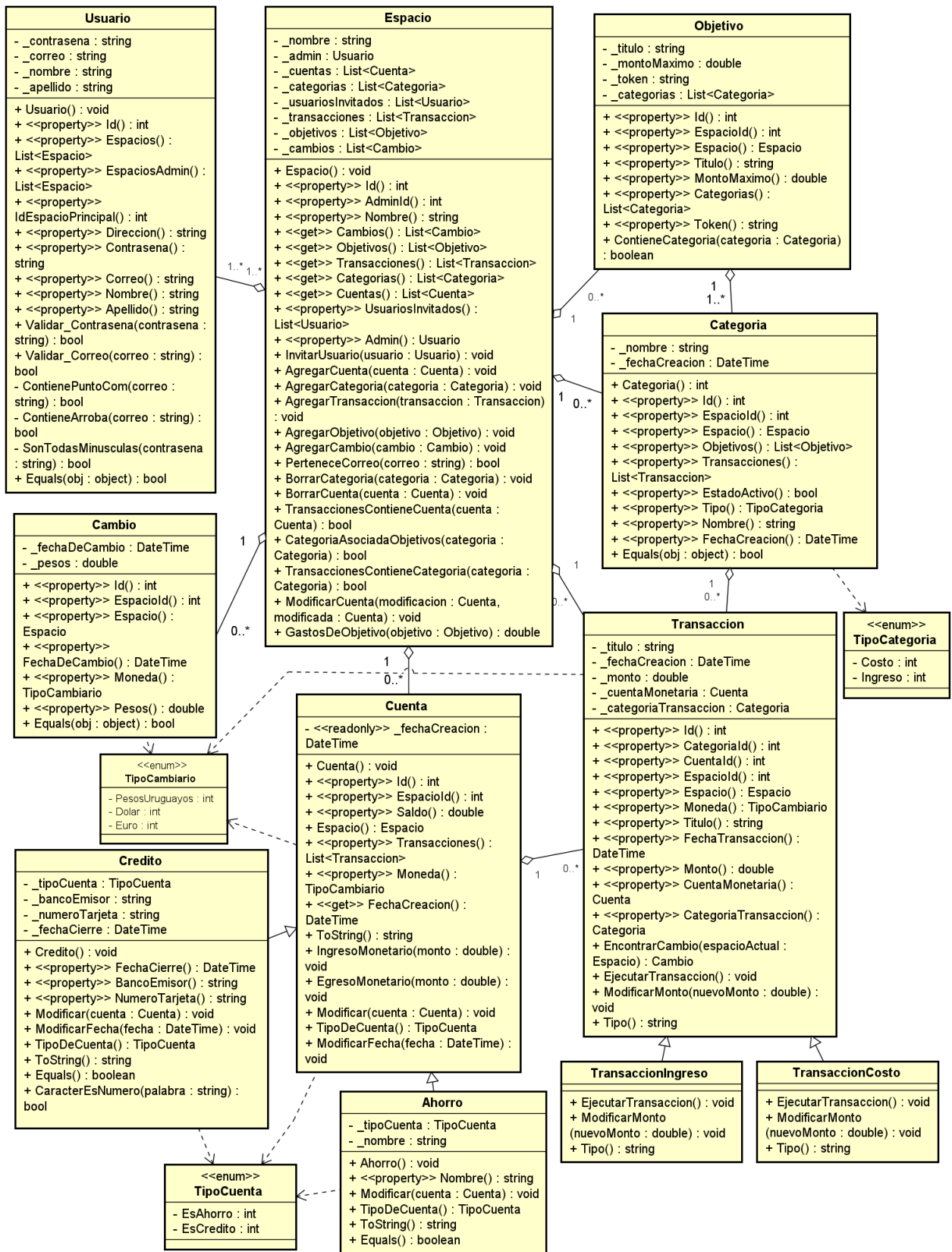


Imagen Diagrama 2.0

En el diagrama de la imagen 2.0 mostramos el modelado UML del paquete Dominio.

Tratamos de que las clases del dominio cumplan con el principio GRASP de Alta Cohesion y Bajo Acoplamiento.

En el definimos Usuario, el cual tiene la responsabilidad de validar sus datos, contraseña y correo que contengan el formato correcto.

Clase Cambio que contiene el tipo cambiario del dólar y euro para determinada fecha, el tipo de moneda es un enumerado (TipoCambiario) para mejorar la mantenibilidad por si a futuro se implementan más monedas. Lo cual resulta sencillo de implementar, ya que con ponerlo en el enumerado y agregándolo en la función de cuenta ToString(), ya se puede utilizar y agregarle su cambio correspondiente. Desde la interfaz, todas las monedas en el enumerado se muestran en una lista, permitiendo al usuario ver y seleccionar entre los diferentes tipos de cambio disponibles.

Clase Cuenta aquí utilizamos el patrón GRASP polimorfismo para reutilizar código favorece la mantenibilidad ni calidad de código. Con esto se heredan los atributos y métodos (públicos) definidos en cuenta, de él heredan Credito y Ahorro.

Clase Transacción aquí también utilizamos el patrón GRASP polimorfismo, en el definimos dos subclases TransaccionIngreso y TransaccionCosto las cuales implementan métodos con override que por ejemplo en una transacción de costo ejecuta la transacción le resta el monto a el saldo de la cuenta asociada a el por el contrario transacción ingreso lo suma, además sobrescribimos un método para modificar el monto de una transacción para q se realice con el cálculo correcto en cada subclase, y también uno para modificar determinado atributo a cada hijo.

Categoría esta clase contiene la fecha de creación que se genera al crear una instancia, para el tipo de categoría creamos un enumerado (TipoCategoria) que contiene tipo Costo e Ingreso, a futuro si se quieren implementar más tipos de categorías será sencillo definirlo.

Objetivo contiene una lista de categorías, un monto y un título, en sus property incluimos excepciones de tipo DominioEspacioExcepcion para que se lancen si estos son vacíos.

Para cumplir con el requerimiento de compartir objetivos de gastos le agregamos una property token que desde la Interfaz se carga, al momento de tocar el botón de compartir objetivo, con un token autogenerado y único. Esta property al dejar de compartir el objetivo vuelve al valor null, y desde la interfaz no se podrá ver el objetivo, en cambio se muestra una página de error.

Espacio, es la que contiene toda la lista de todos los objetos mencionados anteriormente.

Las multiplicidades cambiaron respecto al obligatorio anterior porque agregamos propiedades de navegabilidad que indican el tipo de relaciones de Entity Framework, esto generó más acoplamiento (**Ver Base de datos Entity Framework**).

Diagrama de clases de paquete EspacioReporte

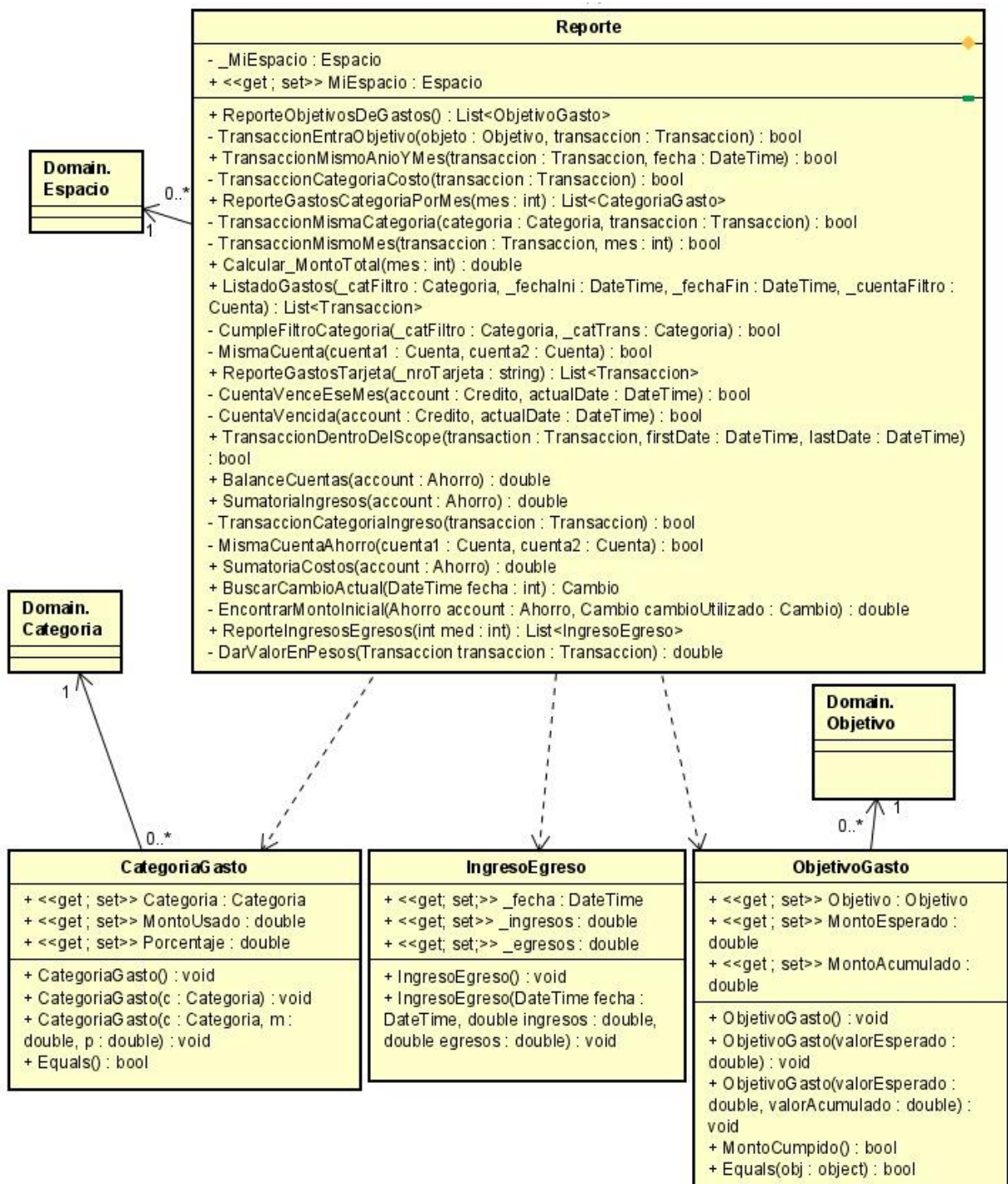


Imagen Diagrama 2.1

En el Diagrama 2.1 se observa el diagrama de clases del paquete EspacioReporte cabe destacar que además de lo graficado la clase Reporte tiene dependencia con todas las clases de Dominio menos con usuario, ya que esta clase se encarga de calcular los reportes de movimiento de un espacio, en el cual las cuentas, transacciones, cambios, objetivos y categorías que son pertenecientes a todo el espacio no a un usuario. Creemos que a futuro deberíamos refactorizar los métodos (ListadoGastos,

TransaccionDentroDelScope), reducir la cantidad de parámetros de su constructor para que el código sea más limpio y mantenible, también deberíamos refactorizar el hecho de aprovechar mejor las propiedades del POO, para que haya menor cantidad de métodos.

La clase reporte toma un Espacio como atributo y realiza operaciones con él en sus métodos retornando los cinco tipos de reportes de los requerimientos, para Reporte de objetivos de gastos utilizamos el método “ReporteObjetivosGasto()” que otorga una lista de ObjetivoGasto, para reporte de gastos de categoría utilizamos el método “ReporteGastosCategoriasPorMes” el cual retorna la lista de CategoriaGasto de ese mes pasado por parámetro, para el listado de gastos utilizamos “ListadoGastos” que de pasada por parámetro una Categoría y un rango de fechas muestra el listado de gastos del rango de fechas, para reporte de Gastos de tarjeta utilizamos “ReporteGastosTarjeta” al cual le pasamos un número de tarjeta y muestra los gastos del mes actual (solo transacciones costo) y para el balance de cuentas utilizamos el método “BalanceCuentas” que recibe la cuenta de ahorro por parámetro y retorna un número double cumpliendo dicho requerimiento.

Diagrama de clases de paquete Repositorio

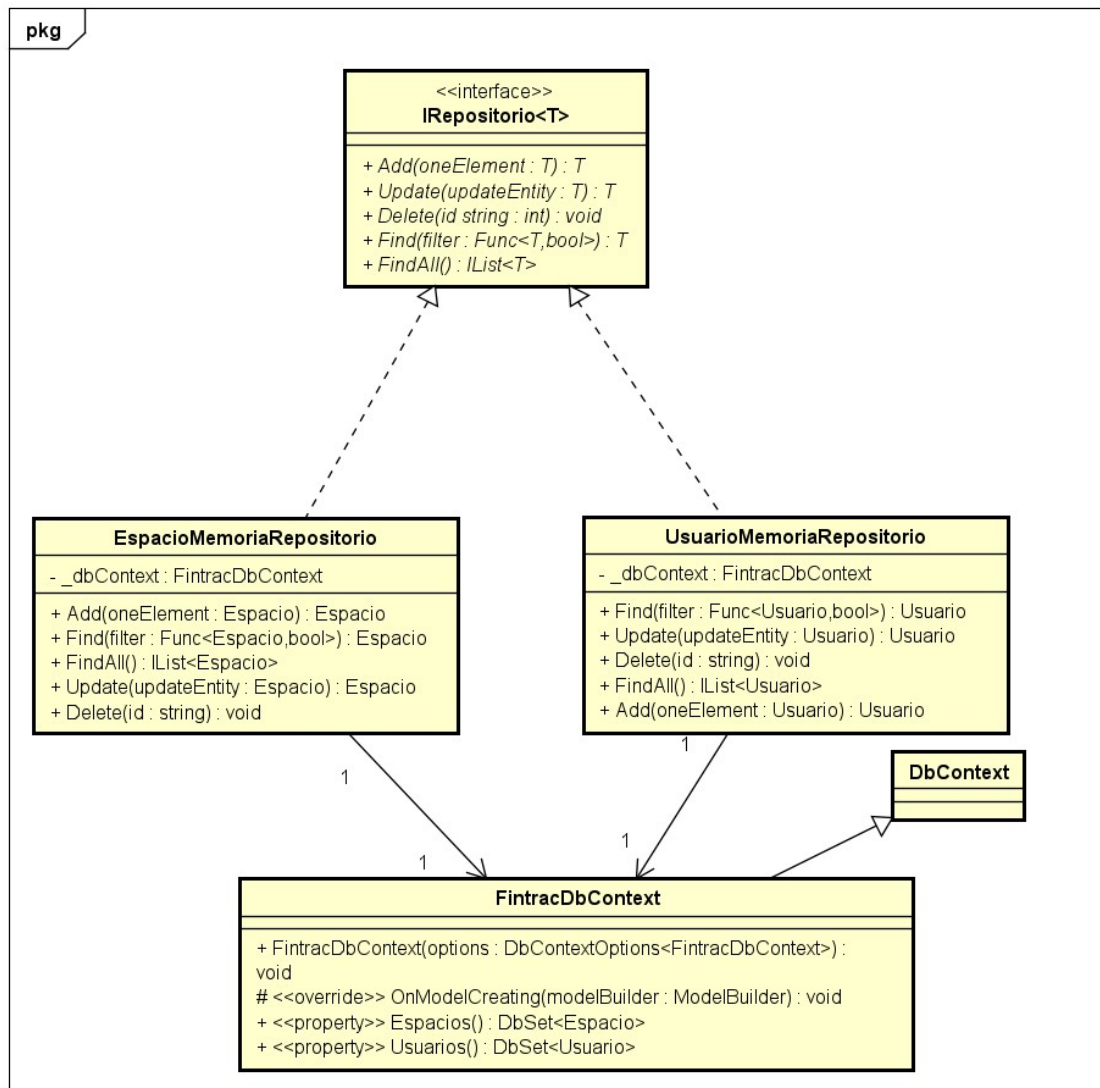


Imagen Diagrama 2.2

En este paquete definimos una interface la cual contiene un contrato con cinco operaciones que utilizaremos para realizar el CRUD (Crear, Leer, Actualizar, Eliminar). Definimos dos entidades UsuarioMemoriaRepositorio que almacena los objetos de Usuario en una clase que hereda de DbContext que nos provee Entity Framework Core para el mapeo de clases a base de datos. Lo mismo para EspacioMemoriaRepositorio pero con Espacio. Escogimos estas entidades para la base de datos Usuario y Espacio porque son los elementos fundamentales en el contexto de nuestra aplicación, ya que un usuario crea y pertenece a múltiples espacios y un espacio almacena toda la información inherente a las funcionalidades requeridas.

Estas clases surgen para favorecer el principio GRASP de Fabricación pura y SOLID Single Responsibility Principle, la única responsabilidad de estas interfaces es la de persistir y permitir la recuperación de la información en la base de datos, esto encapsula la lógica de acceso a los datos, también favorece el patrón SOLID Abierto/Cerrado ya que esta abierta para su extensión pero cerrada para su modificación, ósea podemos ampliar y agregar más bases de datos, pero no deberíamos tener que cambiar el código ya existente, esto hace que nuestra solución sea más mantenible, otro patrón SOLID que favorece esta implementación es el Principio de Inversión de Dependencia al introducir una interfaz que define la abstracción del repositorio, los módulos de alto nivel (por ejemplo LogicaNegocio) dependen de esta abstracción en lugar de depender directamente de la implementación concreta del repositorio, entonces si queremos modificar algo de repositorio no va a cambiar la lógica de negocio.

Diagrama de clases de paquete LogicaNegocio

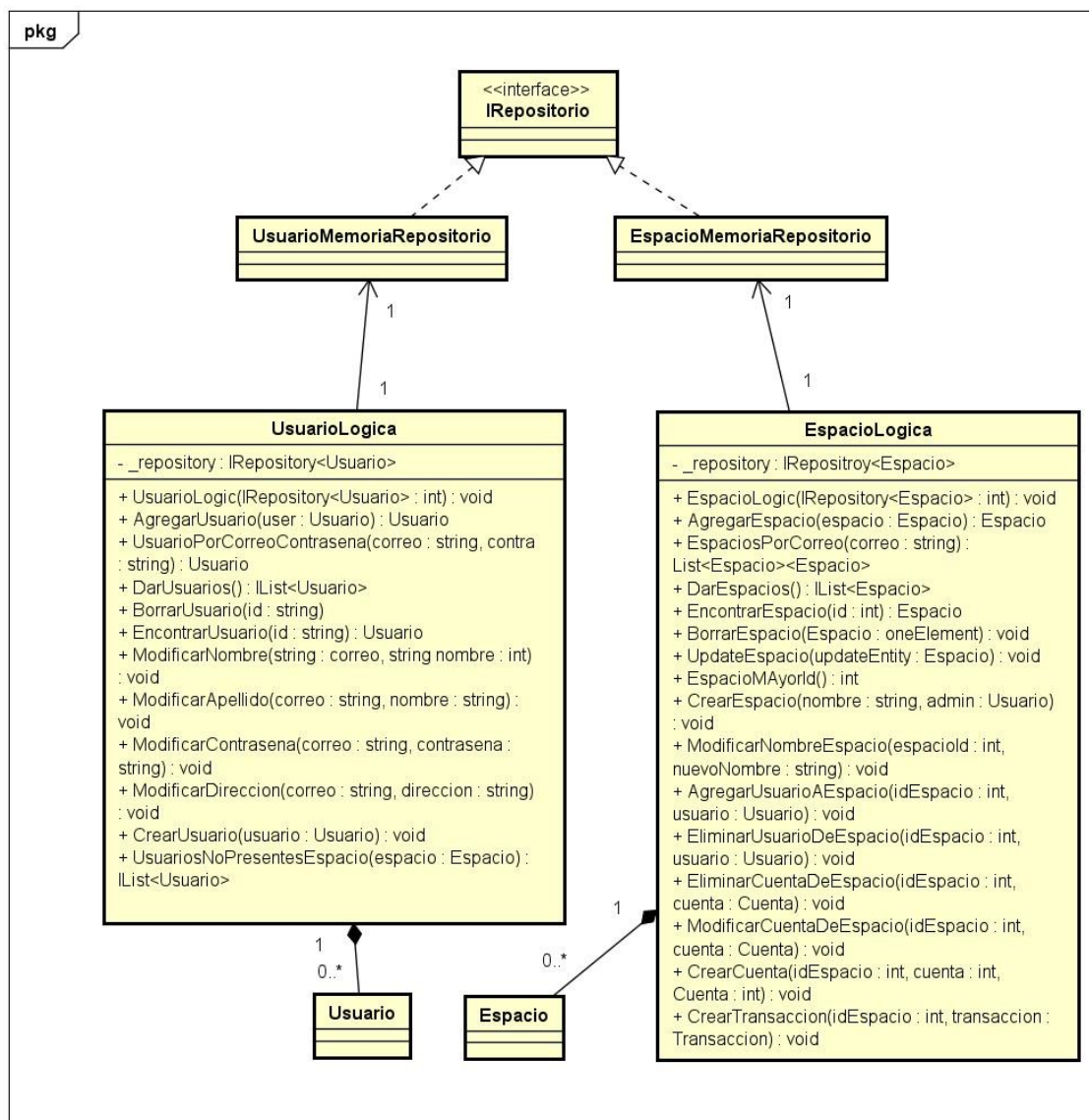


Imagen Diagrama 2.3

Este diagrama corresponde a el namespace de LogicaNegocio, nos pareció adecuado hacer una clase para la lógica de negocio espacio y otra para la de usuario.

En estas clases se puede agregar, borrar, devolver una lista con todos los elementos, devolver un elemento solo y además para el espacio se puede devolver una lista con los espacios que contengan el mismo correo (todos los espacios de un usuario) y para el usuario se puede devolver el elemento que contenga el correo y la contraseña buscada (un usuario en específico).

Además, usamos dos clases **LogicaNegocioEspacioExcepcion** y **LogicaNegocioUsuarioExcepcion** para capturar las excepciones específicas de la lógica de usuario y espacio. Estas clases heredan de **Exception**.

También favorecimos los patrones GRASP creador siendo que como **EspacioLogica** es experto en espacios tiene todos sus datos y toda su lógica le otorgamos la responsabilidad de crearlo. Esto mejora la encapsulación y mantiene el principio de ocultación de información a otras clases.

Diagrama de clases de paquete Excepcion

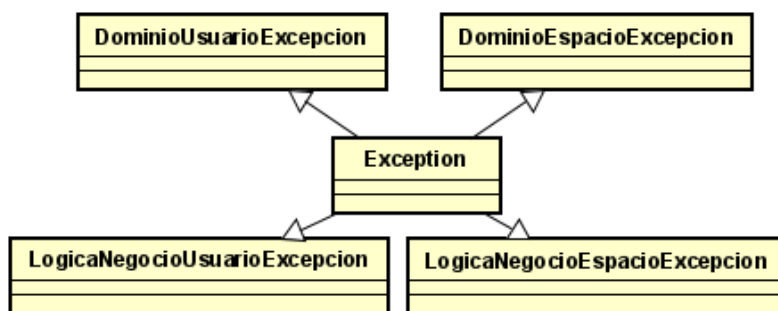


Imagen Diagrama 2.4

Como se muestra en el diagrama 2.4 creamos cuatro excepciones que heredan de **Exception**, su función es capturar excepciones, definidas por nosotros, estas excepciones nos ayudan a entender en donde se encuentra el problema en caso de una excepción. Las manejamos en “Controlador” con las clases controladoras aplicamos try y catch, y retornamos dentro de esos métodos un string con el mensaje de la excepción en caso de que los haya.

Diagrama de clases de paquete DTO:

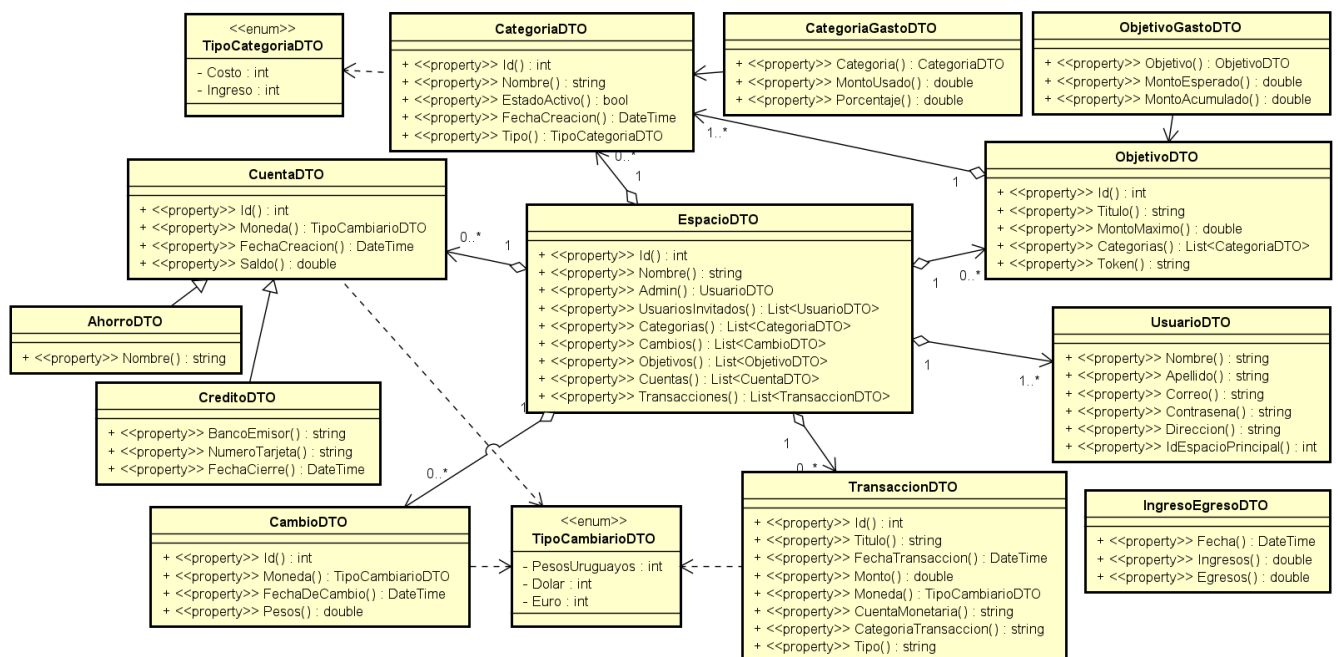


Imagen Diagrama 2.5

Un Data Transfer Object (DTO) es un objeto simple que se utiliza para transportar datos entre procesos o subsistemas de la aplicación. No contiene ninguna lógica de negocio y su propósito principal es agrupar un conjunto de datos para su transferencia.

En el contexto de una aplicación de varias capas, los DTOs se utilizan comúnmente para transferir datos entre la capa de presentación (interfaz) y la capa de negocio (dominio) como en nuestro caso, o entre la capa de negocio y la capa de acceso a datos.

El uso de DTOs ayuda a mantener la separación de responsabilidades entre las capas de la aplicación, mejora la eficiencia al reducir el número de llamadas entre las capas y proporciona un control más preciso sobre qué datos se exponen a cada capa, ayudando a proteger los datos.

Base de datos Entity Framework

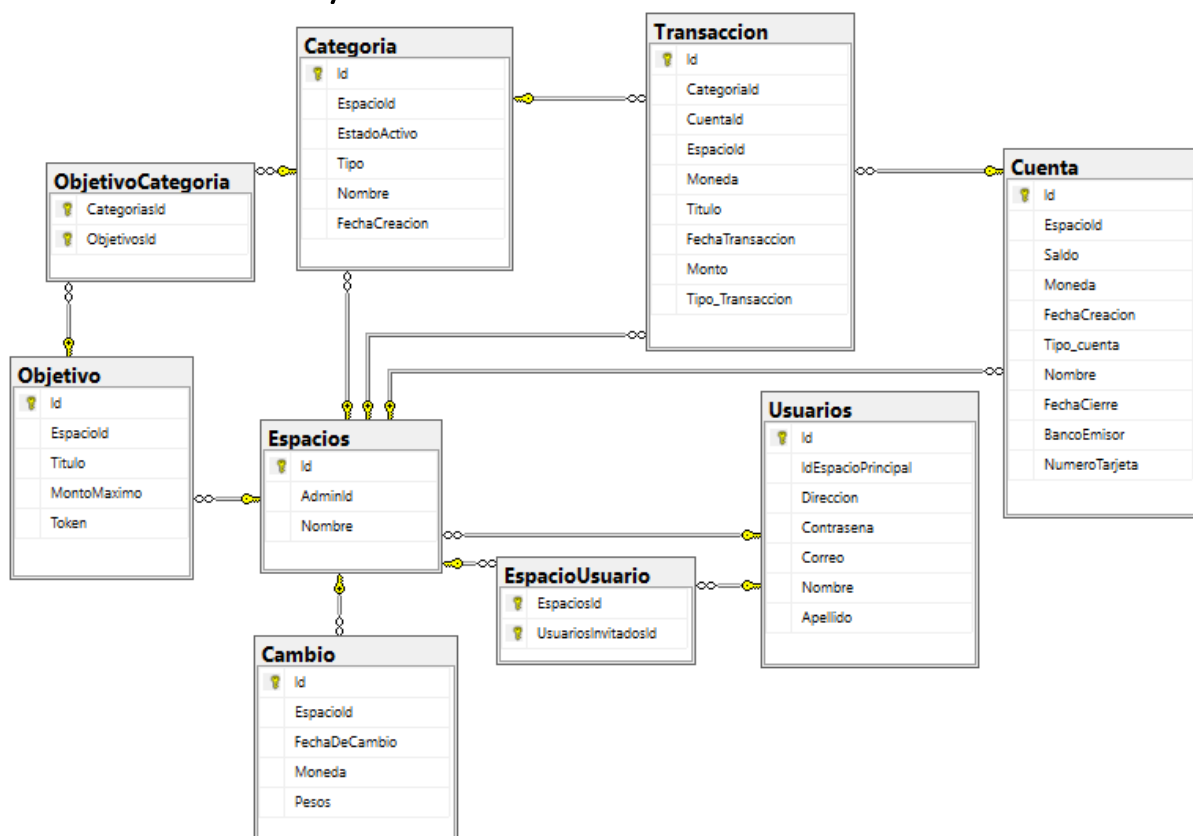


Diagrama de tablas de la base de datos 3.0.

Para conseguir la persistencia de datos en nuestro proyecto utilizamos **Entity Framework Core** un ORM que simplifica el mapeo entre clases y tablas de una base de datos y para administrar esa base de datos utilizamos **Microsoft SQL Server Management Studio 2019**. Utilizamos la técnica Code First ya que ya teníamos las clases del dominio y en base a eso Entity Framework Core genero el esquema de las tablas correspondientes para mapearlas.

Para lograr esta generación de un esquema de base de datos con Entity Framework Core tuvimos que agregar ciertos atributos en cada clase del dominio, las denominadas propiedades de navegación, así Entity Framework Core sabe que tipos de relaciones se establecerán entre objetos, también utilizamos **Fluent API** para configurarlas más detalladamente.

Configuración de la base de datos:

Para la configuración de la base de datos en el paquete donde se realiza el CRUD de la base de datos “Repositorio” implementamos una clase que hereda de DbContext, llamada FintracDbContext, la cual en Entity Framework Core es una parte esencial de la infraestructura que permite interactuar con una base de datos relacional. En ella declaramos dos properties DbSet<Espacio> Espacios y otro DbSet<Usuario> Usuarios, decidimos declarar estas dos properties ya que habíamos establecido dos bases de datos una para espacios “EspacioMemoriaRepositorio” y otra para “UsuarioMemoriaRepositorio” las cuales implementan una interfaz con el CRUD con dichos atributos (Espacio y Usuario) respectivamente. De esta forma en dichas clases establecimos el atributo del tipo FintracDbContext pasado en el constructor por parámetro, con operaciones de dicha clase heredadas de DbContext guardamos y extraemos los datos de la base de datos para el mapeo de clases.

Para el mapeo de la base de datos a clases del dominio en “UsuarioMemoriaRepositorio” y “EspacioMemoriaRepositorio” utilizamos Eagerly Loading, con los métodos Include(atributo) ThenInclude(atributo del atributo anterior) .ToList(), con esto mapeamos todos los objetos de el FintracDbContext con los atributos que deseamos convertir, algunas propiedades de navegación no las mapeamos aquí. Escogimos este método de carga porque nos resultó más sencillo a pesar de que disminuye la performance el cargar todos los atributos todo el tiempo, pero al ser un sistema pequeño nos resultó conveniente.

Luego para guardar los datos de clase a base de datos en los CURD de guardado y actualización de tuplas utilizamos los métodos del DbContext:

Para agregar:

_context.Espacios.Add(oneElement);

_context.SaveChanges();

Para actualizar:

_context.Entry(updateEntity).State = EntityState.Modified;

_context.SaveChanges();

Esto implementado en las dos interfaces con Espacios y Usuarios respectivamente.

Fluent API y configuración de tablas:

En el método OnModelCreating de FintracDbContext configuramos el Fluent Api donde establecimos las relaciones de todas las clases del dominio que deseábamos mapear y el tipo de relación que tenían dichos atributos en dicho lenguaje establecimos las siguientes relaciones y agregamos los siguientes atributos para llevar esta configuración a cabo:

En la clase **Espacio** agregamos un atributo “Id” *int* el cual es la Primary Key de Espacio y otro *int* “AdminId” el cual es clave foránea de la relación 1 a N entre Espacios y Usuarios, esta relación es para mapear que dentro de un Espacio hay un admin Usuario y que un espacio solo tiene un “Usuario” administrador y un Usuario puede ser administrador de muchos espacios. Tabla generada con los datos de la clase espacio “**Espacios**”

En la clase **Usuario** agregamos un atributo “Id” *int* el cual es su Primary Key, *List<Espacio>* “Espacios” la cual es la propiedad de navegación para configurar una relación N a N entre *List<Usuario>* UsuariosInvitados de la clase Espacio y Usuarios, esta propiedad es en representación de que un usuario puede ser invitado a varios espacios y un espacio puede tener varios usuarios invitados, esta relación N a N se representa en la base de datos en la tabla “EspacioUsuario”, también incluimos en esta clase un atributo *List<Espacio>* “EspaciosAdmin” el cual representa la relación 1 a N entre Usuario y Espacio ya que un usuario puede ser admin de múltiples espacios pero un espacio solo tiene un administrador. Tabla generada con los datos de los usuarios “**Usuarios**”, tabla generada con los datos de los usuarios invitados en cada espacio “**EspacioUsuario**”.

En la clase **Categoria** Agregamos un atributo Id *int* como Primary Key, EspacioId *int* como la clave foránea al espacio que pertenece (Relación 1 a N entre lista Categorías de espacio y la Categoría), un atributo *Espacio* Espacio ya que una categoría pertenece a un espacio, una *List<Objetivo>* Objetivos ya que hay una relación N a N entre Objetivo y Categoría (Esta relación se estableció creando una tabla ObjetivoCategoría con Api Fluent) , y una *List<Transaccion>* “Transacciones” ya que una categoría puede pertenecer a muchas transacciones y una transacción solo tiene una categoría, esto es para establecer esa relación 1 a N. Tabla generada con los datos de las categorías “**Categoría**”, tabla generada con los datos de categorías que pertenecen a cada objetivo “**ObjetivoCategoría**”

En la clase **Objetivo** establecimos un atributo Id int como primary key, un atributo int Espacioid como clave foránea de Espacio ya que un objetivo pertenece solo a un espacio y un espacio tiene muchos objetivos, referente a eso también agregamos un atributo Espacio Espacio que es una propiedad de navegabilidad que indica esa cardinalidad de la relación entre Objetivo y Espacio 1 a N. Tabla generada con los datos de cada objetivo, "**Objetivo**".

En la clase **Cambio** agregamos un atributo Primary Key Id int, Espacioid int clave foránea del espacio al que pertenece, y la propiedad de navegabilidad Espacio Espacio la cual establece esa relación 1 a N entre Cambio y Espacio ya que un cambio solo pertenece a un espacio y un espacio tiene múltiples cambios. Tabla generada con los datos de cambio "**Cambio**".

En la clase **Cuenta** ya que tiene dos subclases que heredan de **Ahorro** y **Credito** para mapearlas utilizamos la forma "Table Per Hierarchy" (TPH) la cual hace que todas las clases de la jerarquía compartan todas la misma tabla, dejando en NULL los atributos que no contiene esa clase, y poniendo un discriminador que determina que tipo es cada clase, en nuestro caso aplicamos dos discriminantes Ahorro y Credito ya que clases Cuenta no persistimos. Escogimos esta forma porque nos pareció la más sencilla y porque no nos preocupa dejar atributos en null, puesto que son pocos datos. Los atributos de navegación los establecimos en la clase padre Cuenta, como clave primaria implementamos un atributo int "Id", como clave foránea con espacio un atributo Espacioid int, y como hay una relación entre Cuenta y Espacio de 1 a N ósea una cuenta pertenece a un espacio y un espacio tiene múltiples cuentas, establecimos un atributo Espacio Espacio que establece esta relacion 1 a N. La tabla generada con estos datos es "**Cuenta**".

En la clase **Transaccion** la implementamos con TPH también, con los discriminantes llamados como sus hijos **TransaccionIngreso** y **TransaccionCosto**, en la clase padre Transaccion establecimos los atributos Id int clave primaria, y tres claves foráneas Categorioid int, Cuentaid int, Espacioid int, ya que una transacción tiene una categoría una cuenta y un espacio, relación 1 a N con todas ellas, como propiedad de navegabilidad también agregamos un atributo Espacio Espacio para establecer esa relación 1 a N con este (ya que Categoria y Cuenta ya tenía esto no lo tuvimos que agregar). La tabla generada con estos datos es "**Transaccion**".

Configuración de la Cadena de Conexión:

En el paquete Interfaz, en el archivo llamado **appsettings.json** configuramos una cadena de conexión agregándole el siguiente atributo "**ConnectionStrings**":

```
"FintracsDbConection": "Data Source = (localdb)\\MSSQLLocalDB; Initial Catalog = FinTracDb;  
Integrated Security = True; Connect Timeout = 30; Encrypt = False"
```

Configuración del DbContext y Registro de Servicios:

En el archivo **Program.cs** de la carpeta Interfaz, realizamos el registro del contexto de la base de datos (FintracDbContext) en el sistema de inyección de dependencias, proporcionándole la cadena de conexión mencionada anteriormente mediante el siguiente comando:

```
builder.Services.AddDbContext<FintracDbContext>  
(options => options.UseSqlServer  
(builder.Configuration.GetConnectionString("FintracsDbConection"),  
providerOptions => providerOptions.EnableRetryOnFailure()));
```

De este modo, hemos configurado el servicio del DbContext para que esté conectado a nuestra cadena de conexión, y esté disponible para otros servicios que lo utilicen, como las interfaces UsuarioMemoriaRepositorio y EspacioMemoriaRepositorio, las cuales requieren un DbContext como parámetro.

```
builder.Services.AddScoped<IRepositorio<Usuario>, UsuarioMemoriaRepositorio>();
```

```
builder.Services.AddScoped<IRepositorio<Espacio>, EspacioMemoriaRepositorio>();
```

Estos registros indican que UsuarioMemoriaRepositorio y EspacioMemoriaRepositorio serán utilizados como implementaciones de las interfaces genéricas IRepositorio<Usuario> e IRepositorio<Espacio>, respectivamente, permitiendo la interacción con el DbContext y la base de datos.

```
builder.Services.AddScoped<UsuarioLogica>();
```

```
builder.Services.AddScoped<EspacioLogica>();
```

Estos registros indican que UsuarioLogica y EspacioLogica son servicios disponibles en la inyección de dependencias, encapsulando la lógica de la aplicación y utilizando las interfaces de repositorio para interactuar con la base de datos. Utilizamos Scoped ya que la instancia del servicio se creará una vez por cada solicitud http.

Base de datos en memoria para realización de tests:

Para la realización de pruebas unitarias como las de clases de LogicaNegocio y Repositorio que utilizan una base de datos tuvimos que implementar una base de datos en memoria, la cual implementamos en la clase **InMemoryDbContextFactory** en el paquete Repositorio el cual crea una instancia de FintracDbContext en memoria para que en los test podamos instanciar las clases de los paquetes antes mencionados.

Controladores e Interfaz

Para favorecer la mantenibilidad y buenas prácticas de programación implementamos el patrón GRASP controlador, así desacoplamos el paquete interfaz de LogicaNegocio, Dominio y Excepciones. A pesar de que necesitamos el uso de Dominio y LogicaNegocio en la clase **Program.cs** en el paquete interfaz para la generación de las instancias de los servicios, se desacoplo del resto de interfaces de usuario. **(En el anexo adjuntamos los diagramas UML de dichas interacciones).**

Para utilizar los controladores los inyectamos como servicios en Program.cs el cual instancia todas las variables necesarias para su uso.

Allí tomo las clases de LogicaNegocio los atributos que toman los controladores, y se instancian luego los inyectamos en las páginas que correspondan.


```
builder.Services.AddScoped<IRepositorio<Usuario>, UsuarioMemoriaRepositorio>();
builder.Services.AddScoped<UsuarioLogica>();
builder.Services.AddScoped<IRepositorio<Espacio>, EspacioMemoriaRepositorio>();
builder.Services.AddScoped<EspacioLogica>();
builder.Services.AddSingleton<Persistencia>();
builder.Services.AddMudServices();
builder.Services.AddScoped<ControladorRegistro>();
builder.Services.AddScoped<ControladorUsuarios>();
builder.Services.AddScoped<ControladorEspacios>();
builder.Services.AddScoped<ControladorCuenta>();
builder.Services.AddScoped<ControladorTransaccion>();
builder.Services.AddScoped<ControladorHome>();
builder.Services.AddScoped<ControladorObjetivos>();
builder.Services.AddScoped<ControladorCategorias>();
builder.Services.AddScoped<ControladorCambios>();
builder.Services.AddScoped<ControladorSesion>();
builder.Services.AddScoped<ControladorReporte>();
```

Imagen de program.cs

Luego implementamos los using globales de la interfaz en **_imports.razor** los cuales verifican las dependencias antes mencionadas:

```
@inject NavigationManager NavigationManager;
@using DTO;
@using DTO.EnumsDTO;
@using Controlador;
@inject ControladorSesion controladorSesion;
@inject Persistencia Persistencia;
```

Globalmente en la interfaz inyectamos la Persistencia la cual contiene una instancia singleton con un Id (un int del id del espacio actual), y un Correo (un string con el correo del usuario en la sesión actual). ControladorSesion es una instancia que usamos tanto para validar el inicio de sesión de un usuario como para retornar el espacio actual del programa, lo pusimos globalmente porque lo utilizamos en ciertos mensajes para mostrar datos del espacio en el que se encuentra el usuario. Luego NavigatonManager es una clase con métodos que permite redireccionar a otras páginas fácilmente.

Por último, se observa que utilizamos solo los DTO, DTO.EnumsDTO y Controlador.

Detalle de cobertura

Como se aprecian en las imágenes, todos los paquetes Dominio, EspacioReporte, LogicaNegocio, DTO y Controlador fueron testeados aplicando TDD y logrando una cobertura de 100%. Repositorio no está al 100% ya que en ese paquete se encuentran las migraciones, bajando así la cobertura.

Imagen de cobertura general:

Hierarchy	Covered	Not Covered (Covered	Partially Covered	Not Covered	Covered (%Lines)
maxi_DESKTOP-EGBLJ55_2023-11-16.15_05.coverage	12850	511	13121	23	843	93,81%
controlador.dll	1354	0	1272	0	0	100,00%
dtotest.dll	470	0	471	0	0	100,00%
espacioreporte.dll	588	0	397	0	0	100,00%
logicanegocio.dll	167	0	159	0	0	100,00%
dto.dll	116	0	114	0	0	100,00%
excepcion.dll	8	0	12	0	0	100,00%
controladortest.dll	3683	0	3820	0	0	100,00%
dominio.dll	666	0	567	0	0	100,00%
test.dll	263	2	195	0	1	99,49%
espacioreportetest.dll	1961	26	2215	15	20	98,44%
dominiotest.dll	1695	60	1909	8	56	96,76%
logicanegociotest.dll	650	29	661	0	32	95,38%
repositorio.dll	1229	394	1329	0	734	64,42%

Este porcentaje lo tuvimos muy en cuenta a lo largo de todo el proyecto, asegurándonos de que con los test cubriéramos todos los posibles escenarios de cada funcionalidad.

Los detalles de todos los paquetes se encuentran en el anexo.

Diagramas de interacción

Diagrama de secuencia de la interfaz index.razor al logearse a la aplicación.

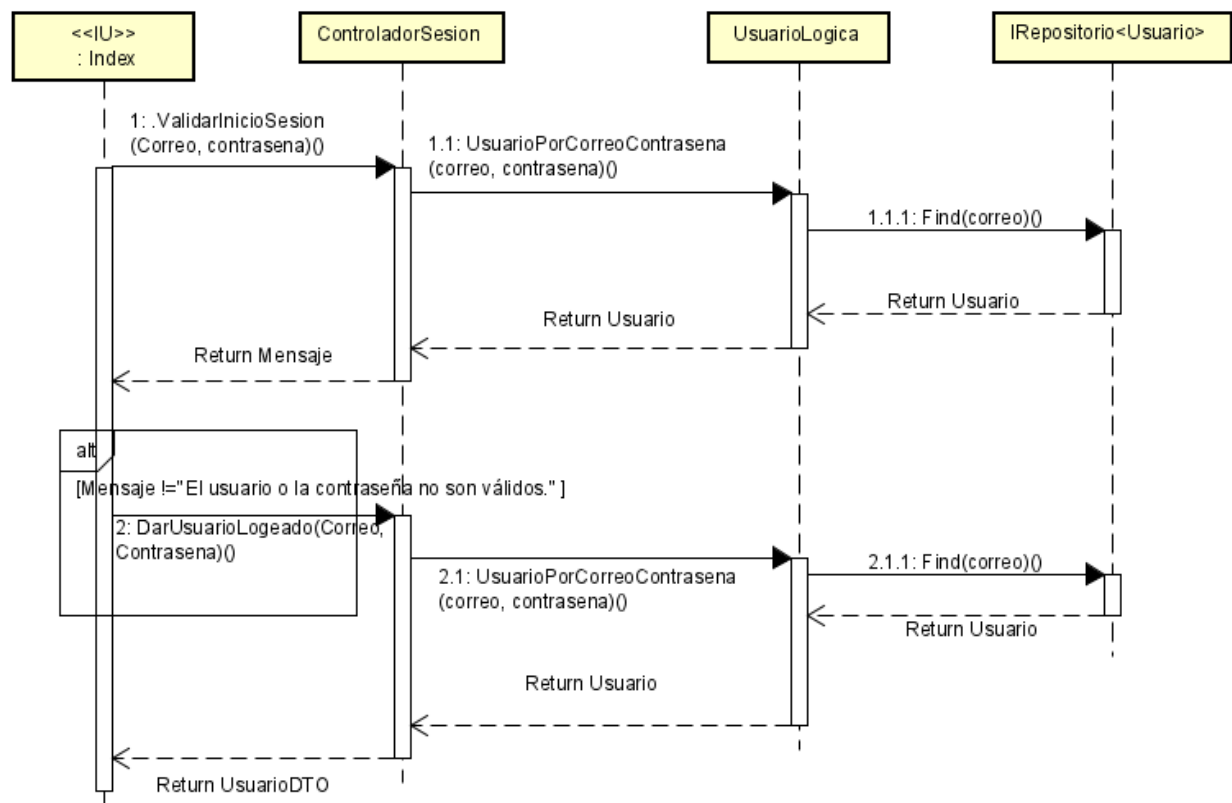
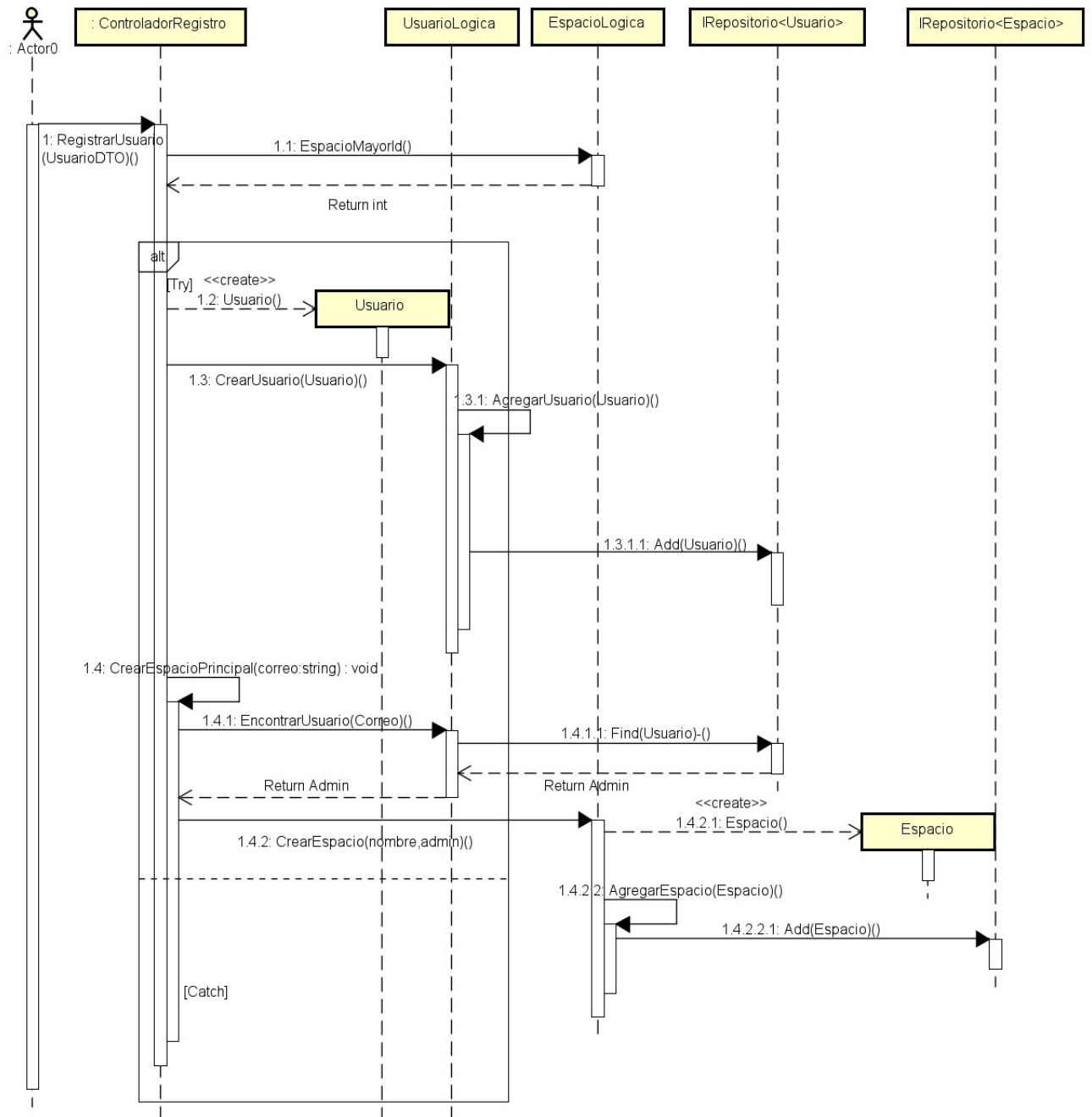
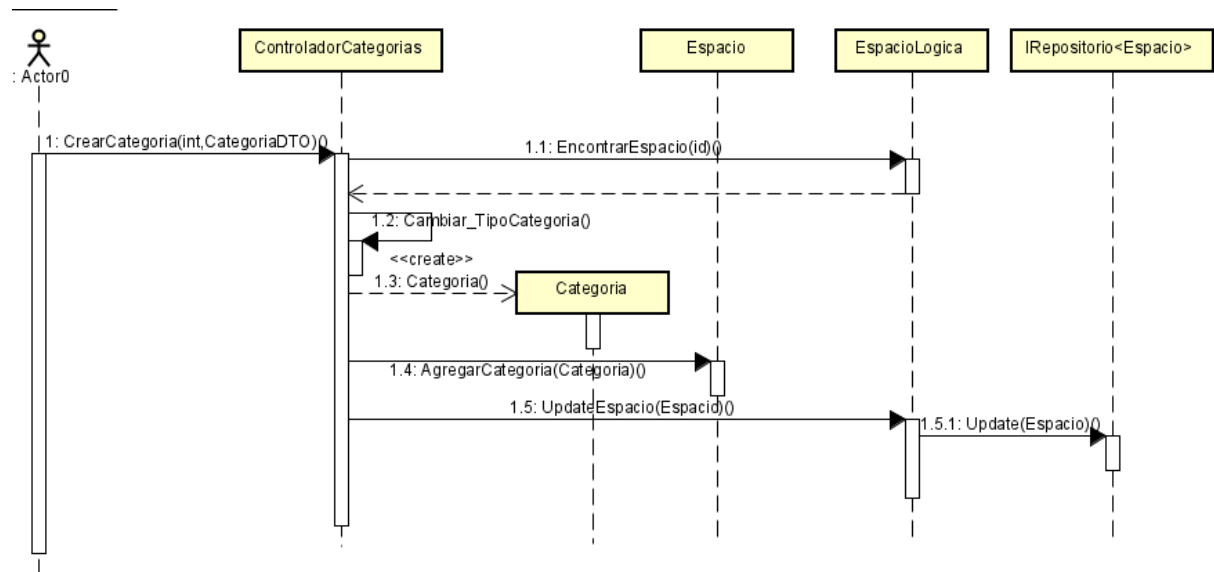


Diagrama de registro de nuevo usuario en registro.razor:



{Nota* Si el diagrama entra en Catch acaba la ejecución del método}

Diagrama Creación de categoría:



Anexo:

La base de datos de prueba se encuentra en el repositorio de GitHub en la carpeta BackUp.

Los datos de acceso son:

- 1- Correo: betina@gmail.com Contraseña: HOLAholo123
- 2- Correo: maxi@gmail.com Contraseña: HOLAholo123

Evidencia de pruebas funcionales:

URL a videos en YouTube:

Parte 1 <https://youtu.be/oOqG0jKQLPM>

Parte 2 <https://youtu.be/dHYM93eWMol>

Parte 3 <https://youtu.be/A01g106ANjU>

UML Interfaces Controladores

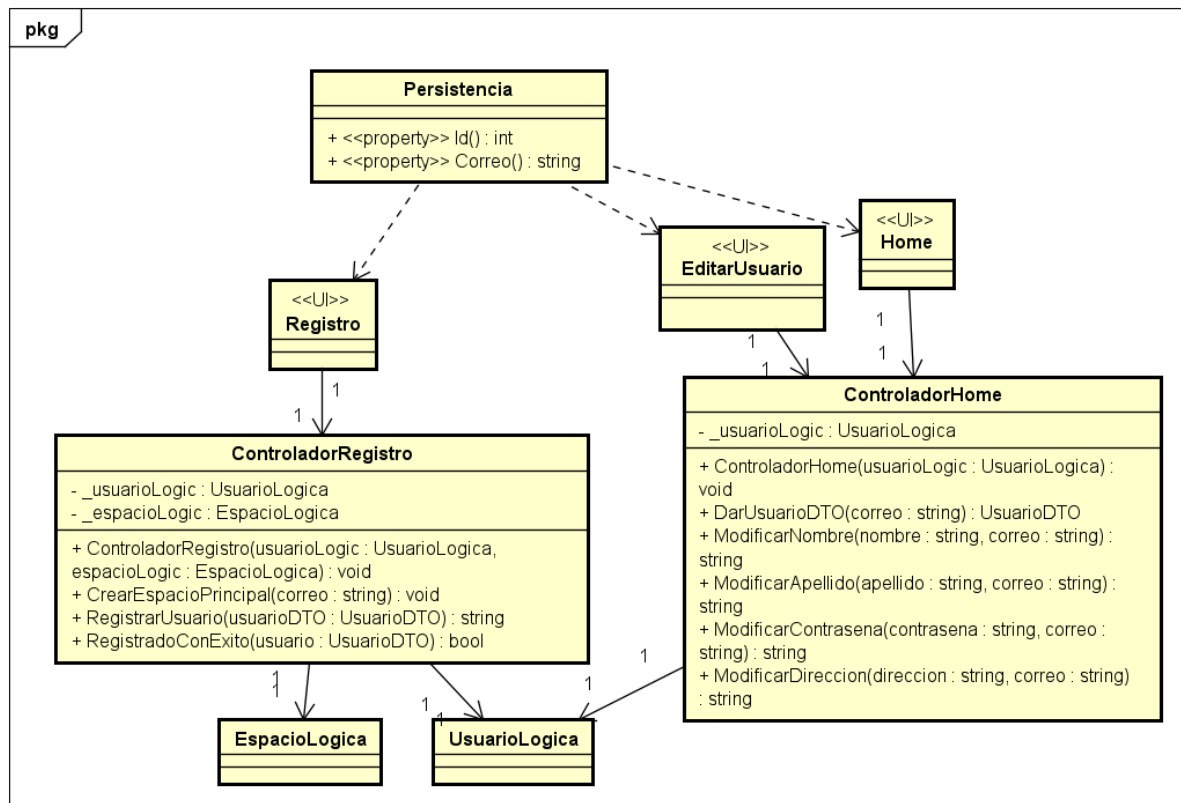


Diagrama 4.0

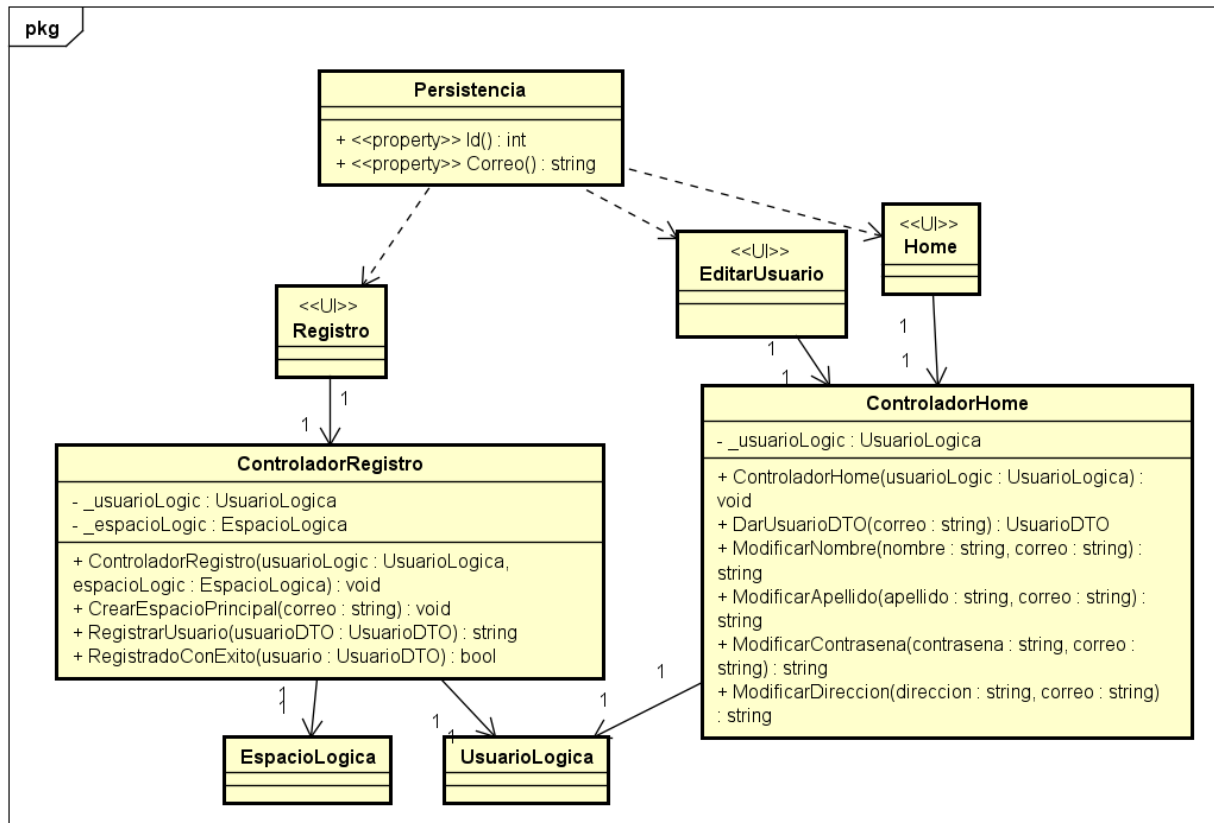


Diagrama 4.1

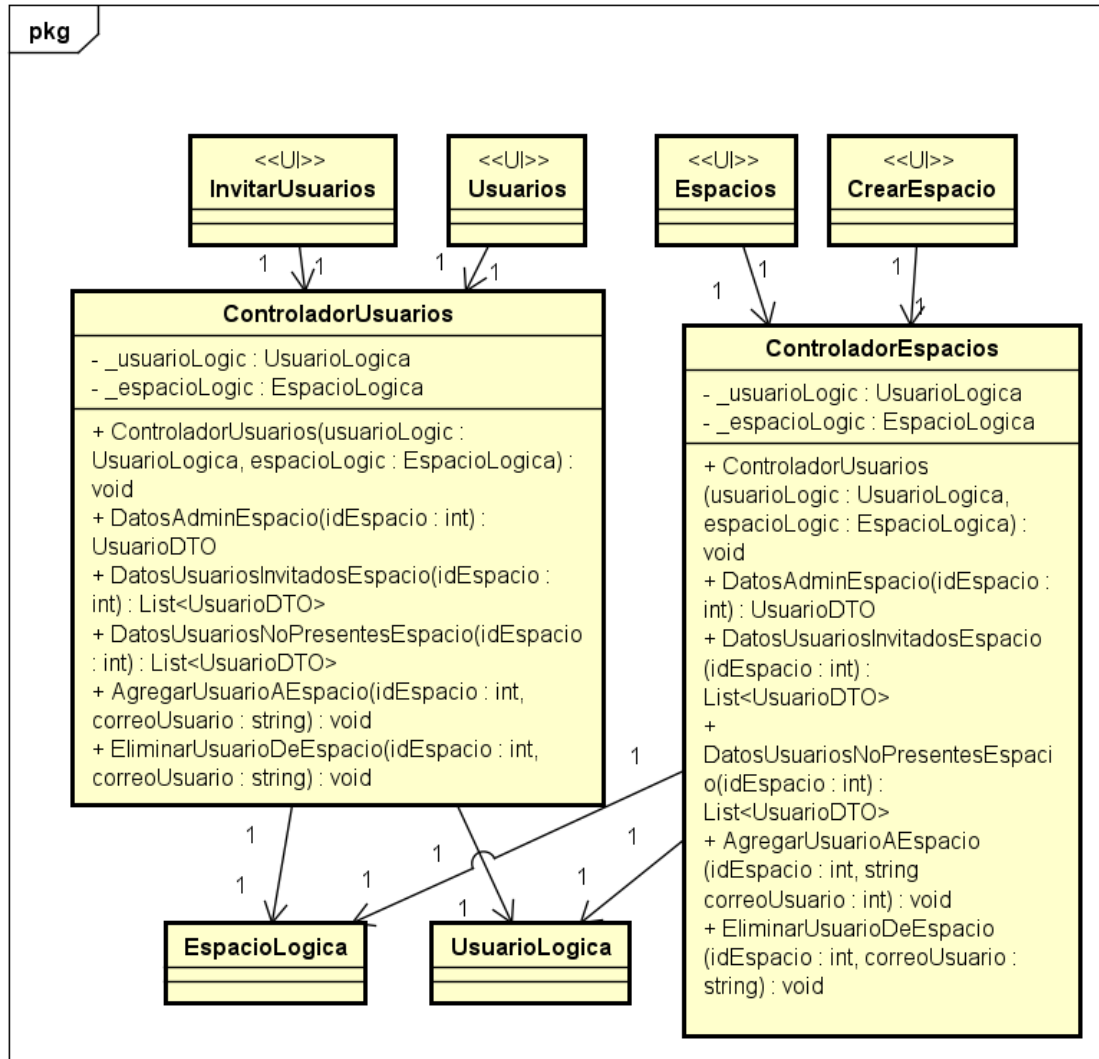


Diagrama 4.2

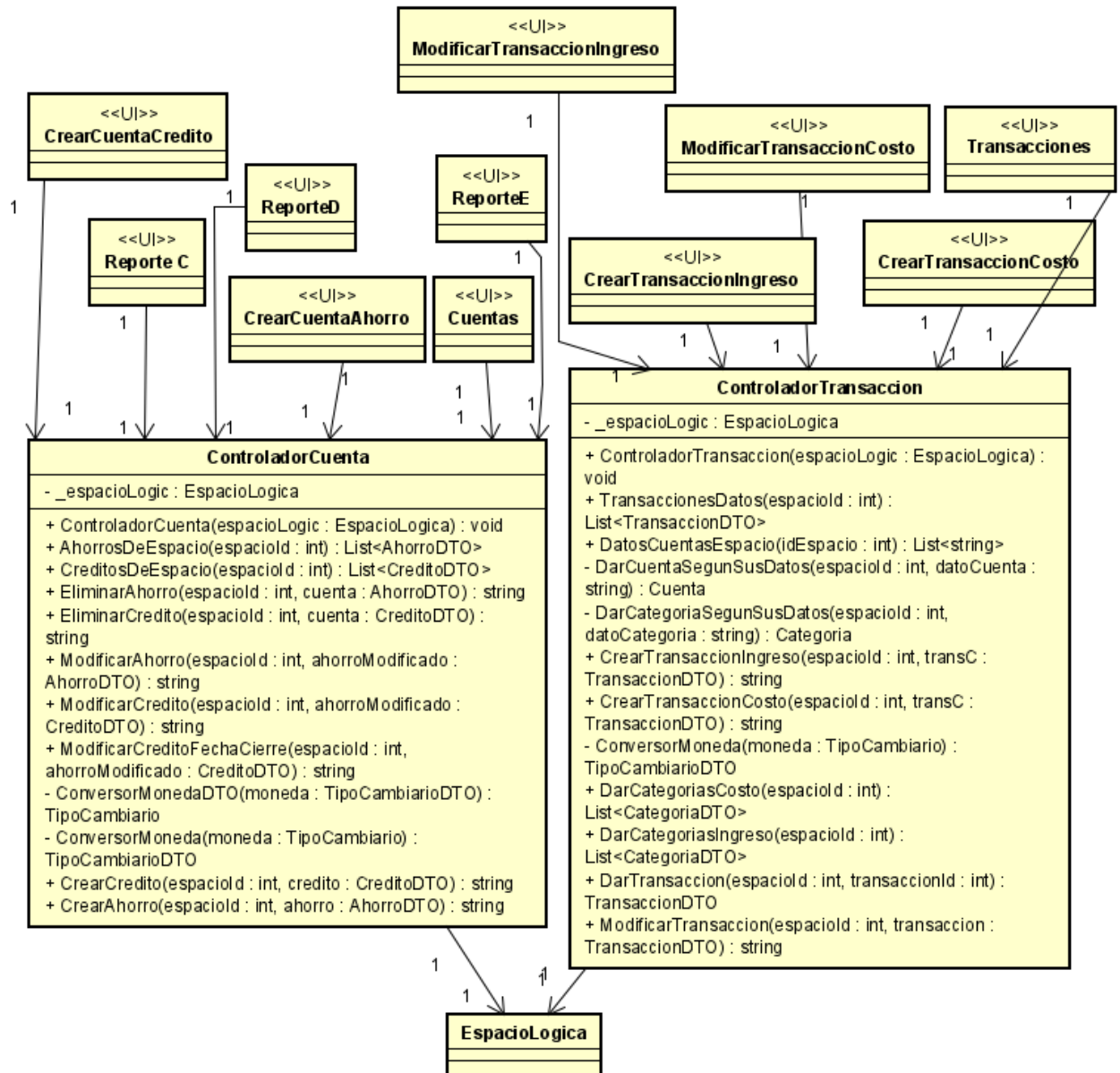


Diagrama 4.3

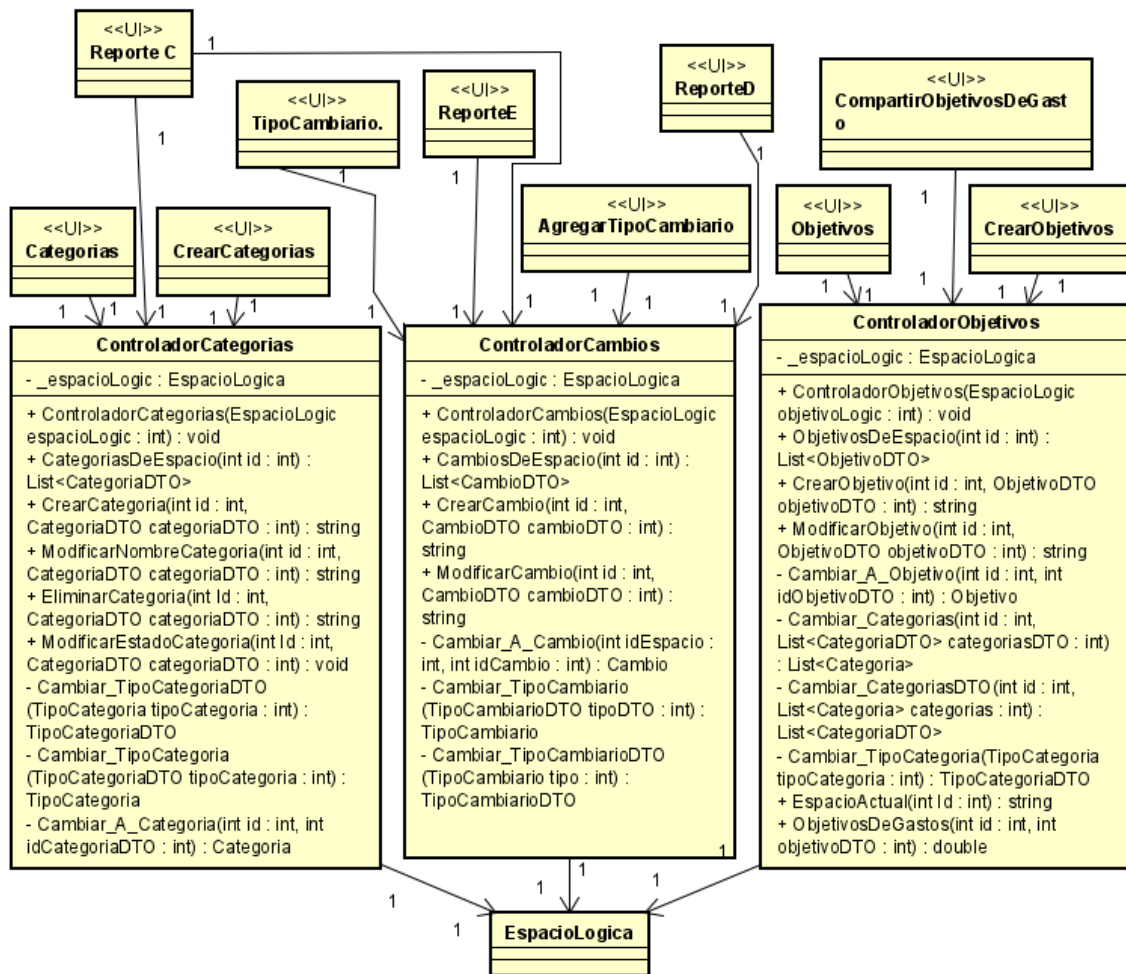


Diagrama 4.4

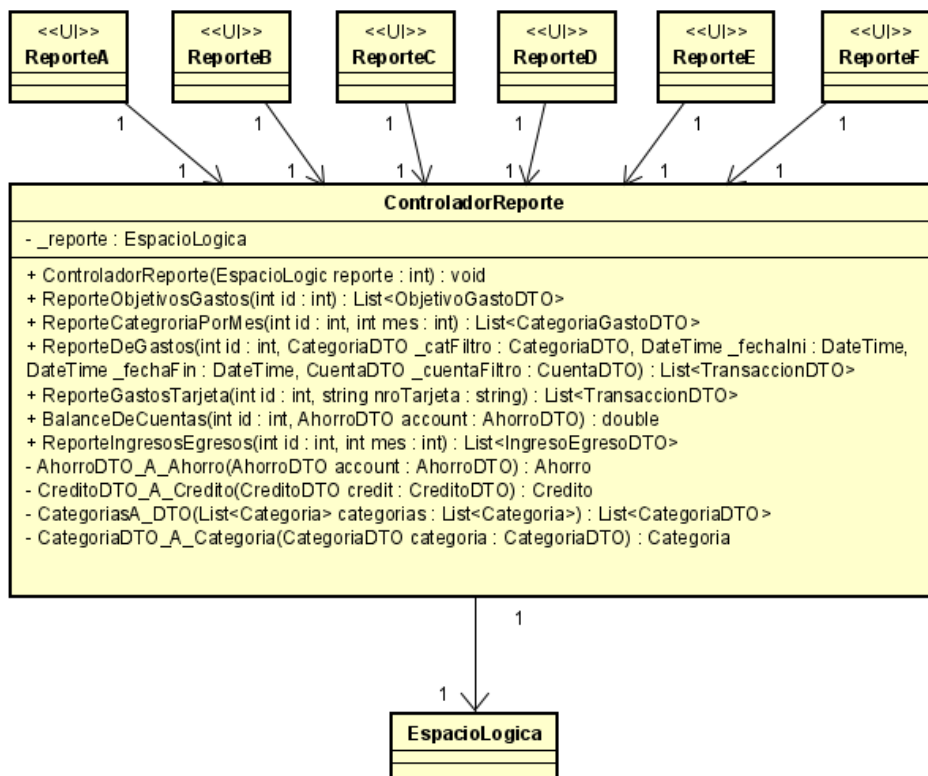


Diagrama 4.5

Los controladores tienen dependencia uso con clases de los paquetes DTO, DTO.DTOEnums, Dominio y Excepcion como se puede observar en el diagrama 4.0, se omitieron estos para mostrar con más claridad el diagrama. Como se puede observar se desacoplo Dominio, LogicaNegocio y Excepciones de las páginas de la interfaz de usuario, las UI tienen dependencia de Persistencia.cs ya que tiene la información de la sesión actual, en algunos diagramas se omitió para que se vea con mejor claridad el uml del controlador, y obviamente que todas las instancias de dichos controladores se generaron en la clase Program.cs de la que depende todo. Las interfaces de usuario tienen dependencia con las clases de DTO (Tampoco se graficó por falta de espacio).

Librerías requeridas:

Microsoft.EntityFrameworkCore: Proporciona las funcionalidades básicas para trabajar con bases de datos.

Microsoft.EntityFrameworkCore.Design: Se utiliza para la creación y configuración de migraciones, que son una forma de mantener actualizada la base de datos con los cambios en el modelo de datos.

Microsoft.EntityFrameworkCore.SqlServer: Es el proveedor de base de datos de SQL Server para EF Core, permitiendo interactuar con una base de datos SQL Server.

Microsoft.EntityFrameworkCore.Tools: Proporciona las herramientas de la consola del Administrador de paquetes para EF Core, utilizadas para realizar tareas como la creación de migraciones, la actualización de la base de datos y la generación de scripts SQL.

Estas son necesarias tanto en Interfaz como en Repository.

Además, en Repository se utilizó:

Microsoft.EntityFrameworkCore.InMemory: Esta librería es un proveedor de base de datos en memoria para EF Core. Es útil para pruebas unitarias y desarrollo, ya que permite trabajar con una base de datos en memoria que se inicializa cada vez que se ejecuta la aplicación.

También en Interfaz, para el requerimiento de mostrar una gráfica con los reportes de ingreso y egreso se utilizó:

MudBlazor: biblioteca de componentes de Blazor que proporciona una amplia gama de componentes de interfaz de usuario predefinidos y personalizables.

MudBlazor.ThemeManager: una parte de la biblioteca MudBlazor que se utiliza para gestionar el tema de la aplicación, permitiendo cambiar entre diferentes temas y proporcionando una interfaz de usuario para que los usuarios de la aplicación puedan cambiar el tema por sí mismos.

Cobertura detallada de Dominio:

dominio.dll	666	0	567	0	0	100,00%
{ } Dominio	666	0	567	0	0	100,00%
▶ Ahorro	38	0	30	0	0	100,00%
▶ Cambio	41	0	26	0	0	100,00%
▶ Categoria	40	0	38	0	0	100,00%
▶ Credito	79	0	63	0	0	100,00%
▶ Cuenta	39	0	45	0	0	100,00%
▶ Espacio	187	0	146	0	0	100,00%
▶ Objetivo	38	0	40	0	0	100,00%
▶ Transaccion	81	0	69	0	0	100,00%
▶ TransaccionCosto	11	0	11	0	0	100,00%
▶ TransaccionIngreso	11	0	11	0	0	100,00%
▶ Usuario	81	0	83	0	0	100,00%
▶ Espacio.<>c__DisplayClass42_0	3	0	1	0	0	100,00%
▶ Espacio.<>c__DisplayClass45_0	3	0	1	0	0	100,00%
▶ Espacio.<>c__DisplayClass47_0	3	0	1	0	0	100,00%
▶ Espacio.<>c__DisplayClass48_0	3	0	1	0	0	100,00%
▶ Espacio.<>c__DisplayClass49_0	8	0	1	0	0	100,00%

Cobertura detallada EspacioReporte:

espacioreporte.dll	588	0	397	0	0	100,00%
{ } EspacioReporte	588	0	397	0	0	100,00%
CategoriaGasto	36	0	27	0	0	100,00%
IngresoEgreso	18	0	16	0	0	100,00%
ObjetivoGasto	46	0	32	0	0	100,00%
Reporte	488	0	322	0	0	100,00%

Cobertura detallada de Repositorio:

repositorio.dll	1229	394	1329	0	734	64,42%
{ } Repositorio.Migrations	679	394	1147	0	734	60,98%
FintracDbContextModelSnapshot	0	104	0	0	422	0,00%
FintracDbContextModelSnapshot.<>c	0	280	0	0	292	0,00%
InitialEspacioDataBase	178	10	673	0	20	97,11%
InitialEspacioDataBase.<>c	501	0	474	0	0	100,00%
{ } Repositorio	550	0	182	0	0	100,00%
EspacioMemoriaRepositorio	178	0	54	0	0	100,00%
FintracDbContext	315	0	81	0	0	100,00%
InMemoryDbContextFactory	6	0	5	0	0	100,00%
UsuarioMemoriaRepositorio	42	0	39	0	0	100,00%
FintracDbContext.<>c	2	0	1	0	0	100,00%
UsuarioMemoriaRepositorio.<>c__DisplayClass4_0	4	0	1	0	0	100,00%
UsuarioMemoriaRepositorio.<>c__DisplayClass5_0	3	0	1	0	0	100,00%























Cobertura detallada de LogicaNegocio:

logicanegocio.dll	167	0	159	0	0	100,00%
{ } LogicaNegocio	167	0	159	0	0	100,00%
EspacioLogica	92	0	90	0	0	100,00%
UsuarioLogica	65	0	65	0	0	100,00%
EspacioLogica.<>c	2	0	1	0	0	100,00%
EspacioLogica.<>c__DisplayClass6_0	2	0	1	0	0	100,00%
UsuarioLogica.<>c__DisplayClass3_0	3	0	1	0	0	100,00%
UsuarioLogica.<>c__DisplayClass6_0	3	0	1	0	0	100,00%

Cobertura detallada de Dto:

dto.dll	116	0	114	0	0	100,00%
{ } DTO	116	0	114	0	0	100,00%
AhorroDTO	2	0	2	0	0	100,00%
CambioDTO	8	0	8	0	0	100,00%
CategoriaDTO	10	0	10	0	0	100,00%
CategoriaGastoDTO	6	0	6	0	0	100,00%
CreditoDTO	6	0	6	0	0	100,00%
CuentaDTO	8	0	8	0	0	100,00%
EspacioDTO	26	0	24	0	0	100,00%
IngresoEgresoDTO	6	0	6	0	0	100,00%
ObjetivoDTO	10	0	10	0	0	100,00%
ObjetivoGastoDTO	6	0	6	0	0	100,00%
TransaccionDTO	16	0	16	0	0	100,00%
UsuarioDTO	12	0	12	0	0	100,00%

Cobertura detallada de Controlador:

 controlador.dll	1354	0	1272	0	0	100,00%
{ } Controlador	1354	0	1272	0	0	100,00%
▶  ControladorCambios	96	0	89	0	0	100,00%
▶  ControladorCategorias	99	0	101	0	0	100,00%
▶  ControladorCuenta	175	0	212	0	0	100,00%
▶  ControladorEspacios	46	0	52	0	0	100,00%
▶  ControladorHome	40	0	54	0	0	100,00%
▶  ControladorObjetivos	116	0	103	0	0	100,00%
▶  ControladorRegistro	39	0	44	0	0	100,00%
▶  ControladorReporte	393	0	321	0	0	100,00%
▶  ControladorSesion	53	0	51	0	0	100,00%
▶  ControladorTransaccion	209	0	181	0	0	100,00%
▶  ControladorUsuarios	61	0	54	0	0	100,00%
▶  ControladorCambios.<>c__DisplayClass5_0	2	0	1	0	0	100,00%
▶  ControladorCategorias.<>c__DisplayClass4_0	4	0	1	0	0	100,00%
▶  ControladorCategorias.<>c__DisplayClass9_0	2	0	1	0	0	100,00%
▶  ControladorCuenta.<>c__DisplayClass8_0	3	0	1	0	0	100,00%
▶  ControladorObjetivos.<>c__DisplayClass5_0	2	0	1	0	0	100,00%
▶  ControladorObjetivos.<>c__DisplayClass6_0	3	0	1	0	0	100,00%
▶  ControladorTransaccion.<>c__DisplayClass11_0	2	0	1	0	0	100,00%
▶  ControladorTransaccion.<>c__DisplayClass12_0	3	0	1	0	0	100,00%
▶  ControladorTransaccion.<>c__DisplayClass4_0	3	0	1	0	0	100,00%
▶  ControladorTransaccion.<>c__DisplayClass5_0	3	0	1	0	0	100,00%