Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 1
Diseño de aplicaciones 1

Fecha:12/10/2023

Maximiliano Giménez - 294598

Ana Betina Kadessian - 221509

Mateo Arias - 274857

Docentes: Diego Nicolás Balbi Alves

Docente: Guzmán Viglieca Frank

Grupo: M5A

Repositorio Github

Índice

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.

Justificación de diseño

Diagramas de paquetes:

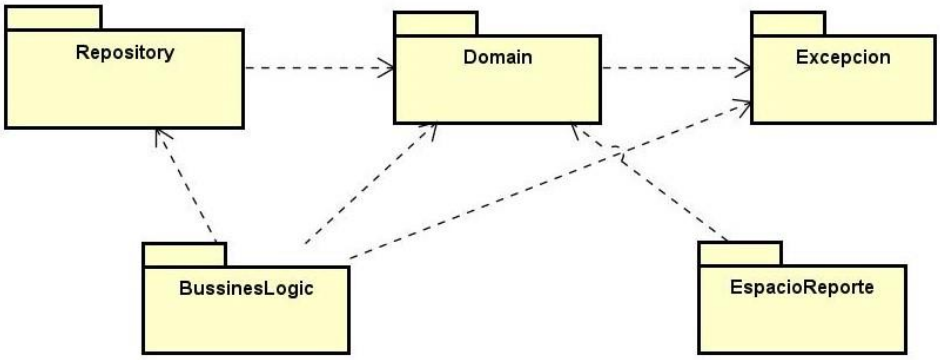


Imagen Diagrama 1.0

En el Diagrama 1.0 podemos apreciar el paquete Domain 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, estas entidades se encuentran en el paquete que luego en un futuro convertiremos en tablas de una base de datos.

En el paquete EspacioReporte generamos un conjunto de tres clases (Reporte, ObjetivoGasto, CategoriaGasto) 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 “Domain”, ObjetivoGato y CategoriaGasto son clases auxiliares a esta última clase, facilitando las operaciones y mostrado de datos de “Reporte”, 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 Repository creamos un paquete con dicho nombre el cual contiene una interface IRepository, que implementa un contrato con la lógica para interactuar con la base de datos en la próxima implementación, las clases “EspacioMemoryRepository” y “UsuarioMemoryRepository” contendrán dicha lógica de acceso a datos específica para cada entidad y proporcionará métodos para interactuar con la base de datos.

El paquete "BusinessLogic" 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.

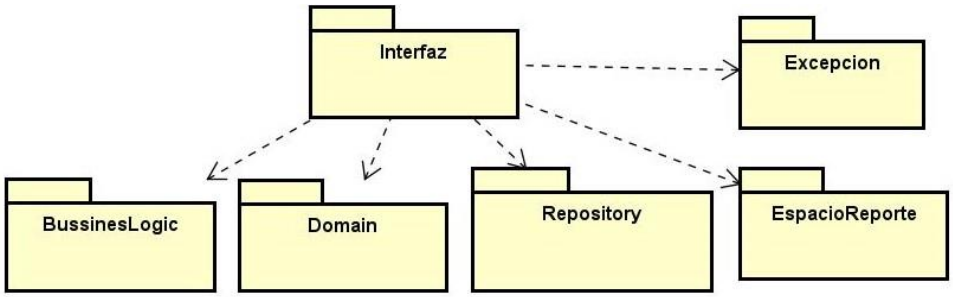


Imagen Diagrama 1.1

En el Diagrama 1.1 Mostramos la interacción de dependencias exclusiva del paquete Interfaz con el resto de los paquetes de la aplicación.

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.

Diagramas de clases

Diagrama de clases de paquete Domain

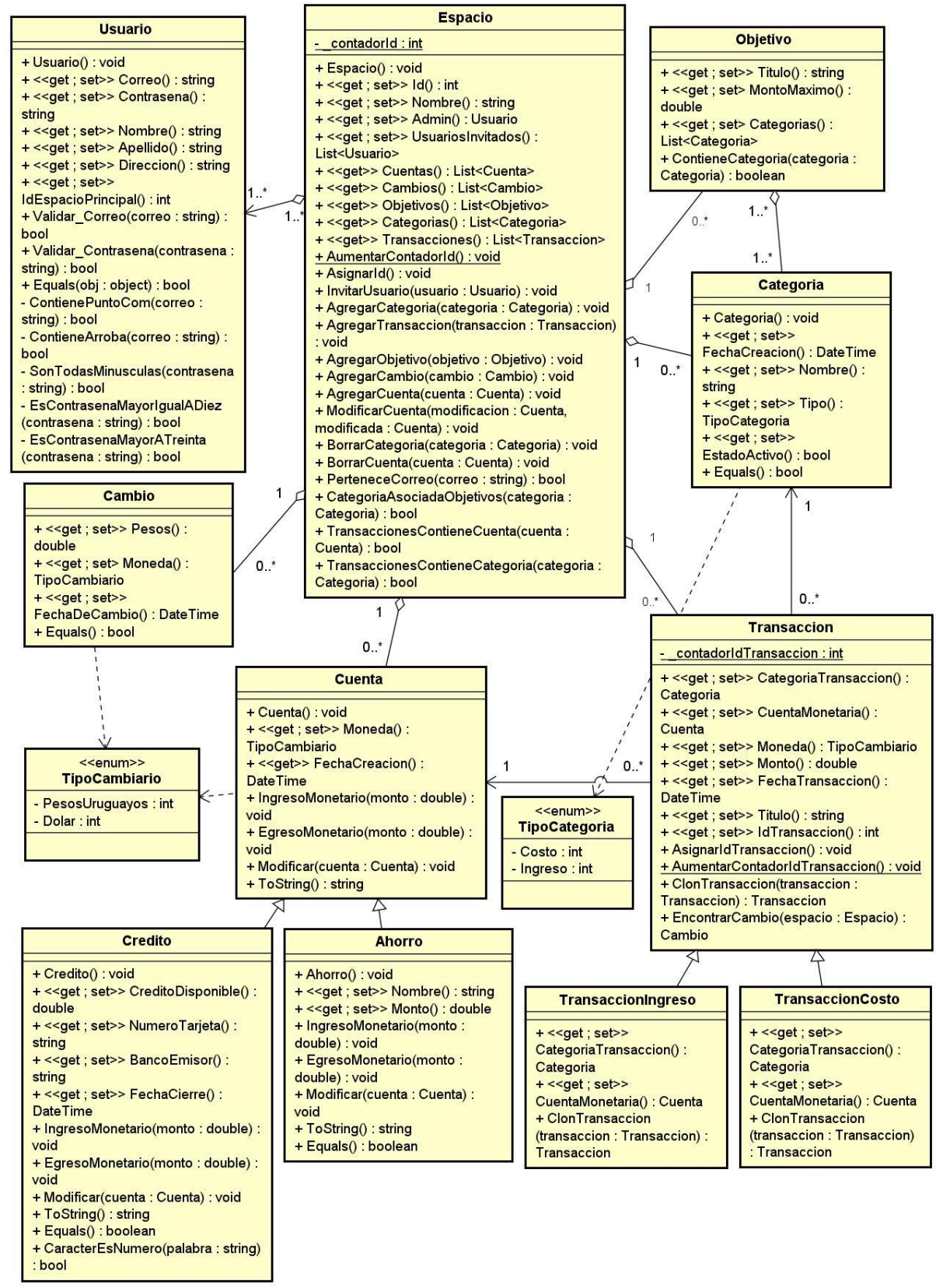


Imagen Diagrama 2.0

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

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 (por ahora) para determinada fecha, el tipo de moneda es un enumerado (TipoCambiario) para mejorar la mantenibilidad por si a futuro se implementan más monedas.

Clase Cuenta aquí utilizamos el polimorfismo para reutilizar código, el beneficio que nos brinda es que un espacio tiene una lista de Cuentas (ya sea Credito o Ahorro), si no aplicáramos herencia habría reutilización de código, lo que no favorece la mantenibilidad ni calidad de código. Con esto se heredan los atributos y métodos (públicos) definidos en cuenta,

Clase Transacción aquí también utilizamos la herencia, hicimos override a las properties “CategoriaTransaccion” para que en el set podamos definir que, si es TransaccionIngreso se setean categorías de ingreso, por el contrario, para TransaccionCosto se setean solo categorías de costo, esto lo manejamos con una excepción de DomainEspacioException. Luego hicimos un override en la property "CuentaMonetaria" (lanza DomainEspacioException si la cuenta tiene moneda diferente a la de la transaccion), puede parecer reutilización de código, pero es una solución que se encontró al hecho de que al clonar una transacción se podía cambiar el tipo de moneda, lo que lanzaba excepción, esto lo hicimos para definir un atributo privado y en el método ClonarTransaccion se lo pasamos directo sin pasar por la property.

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 DomainEspacioException para que se lancen si estos son vacíos.

Espacio, esta clase presenta gran acoplamiento ya que es la que contiene toda la lista de todos los objetos mencionados anteriormente que necesitan un espacio para existir. Pensamos reducir este acoplamiento subdividiéndola en más clases, pero encontramos que esto dificultaría el manejo del código y además no nos parecía necesario ya que vemos bastante cohesión de atributos en ella. En los métodos manejamos DomainEspacioException al no cumplir con algo requerido para su espacio por ejemplo repetir nombres o borrar objetos utilizados por otras listas de dicho espacio, también definimos una variable estática idTransaccion ya que no había atributos únicos para identificar cada instancia.

Diagrama de clases de paquete EspacioReporte

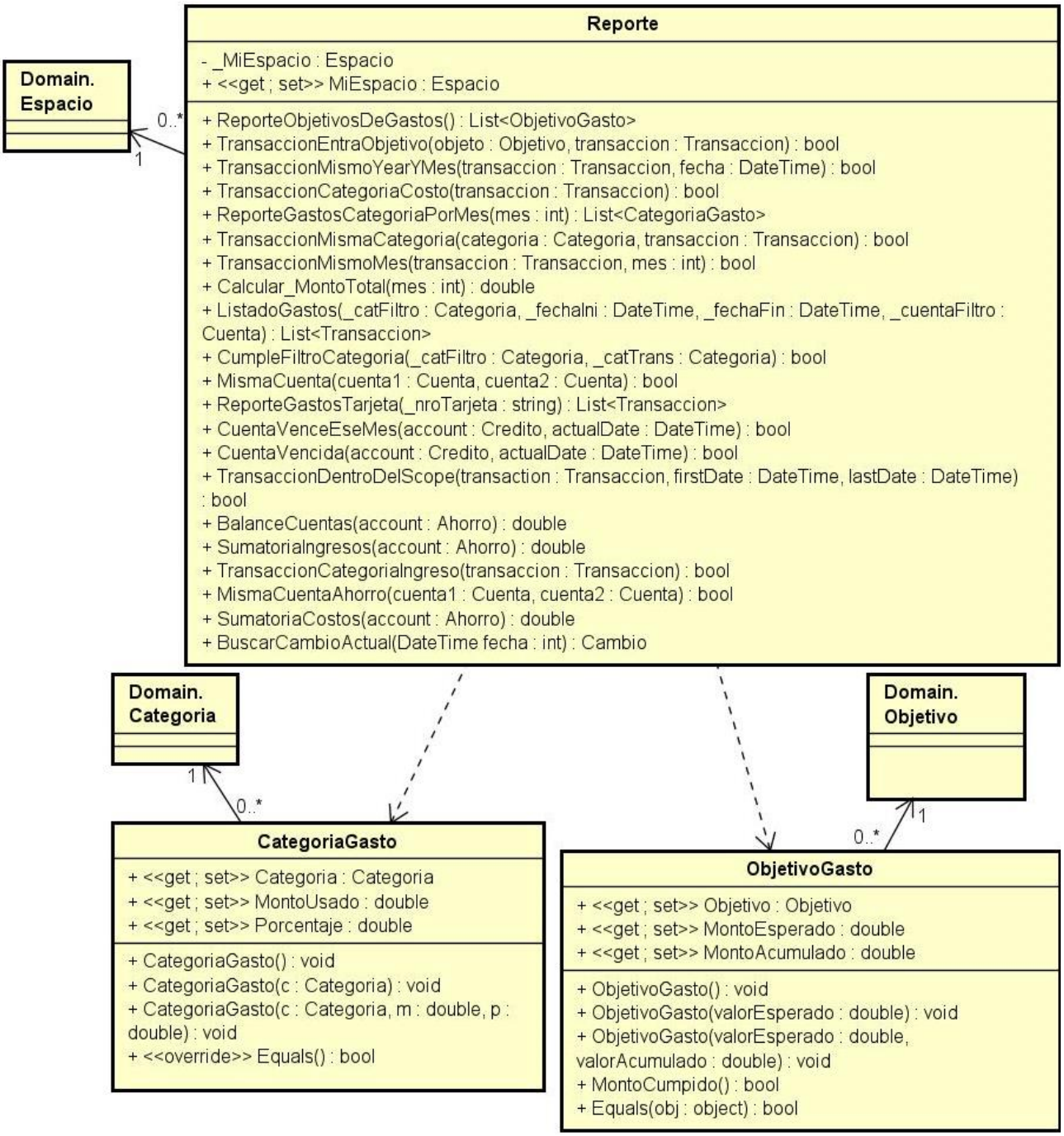


Imagen Diagrama 2.1

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 Domain 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 Categoria 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 Repository

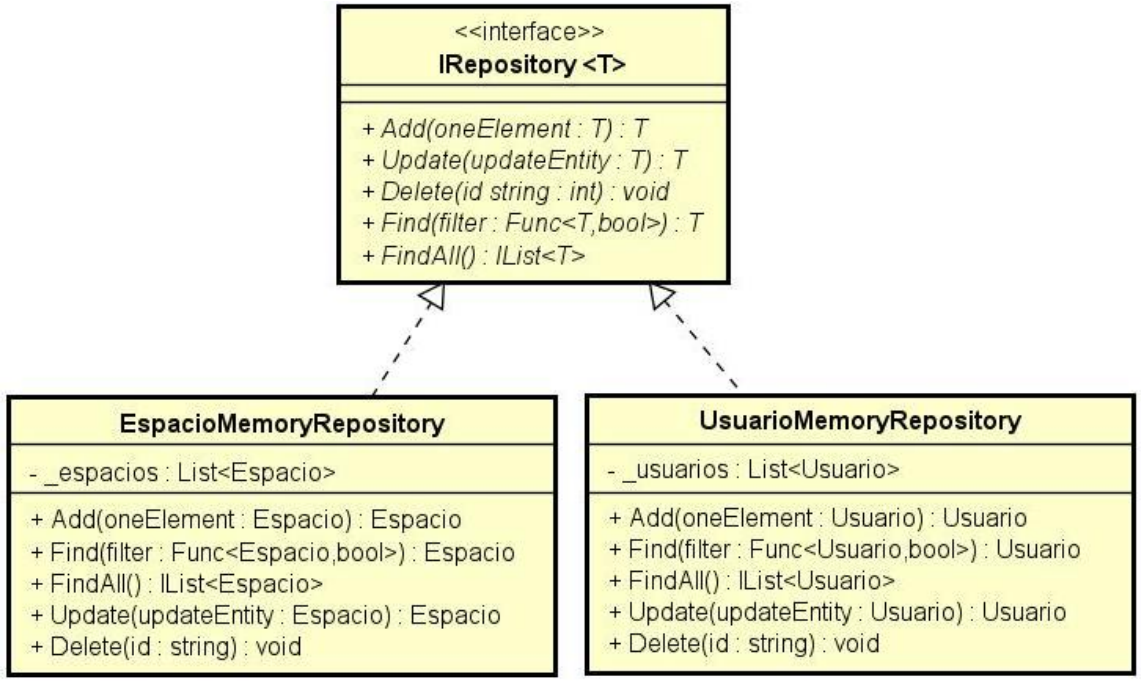


Imagen Diagrama 2.2

Basándonos en el patrón de diseño repository como se puede observar en el 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) en la base de datos en la futura implementación. Definimos dos entidades UsuarioMemoryRepository que almacena los objetos de usuario en una lista en memoria y realiza las operaciones correspondientes en esa estructura. Lo mismo para EspacioMemoryRepository pero con Espacio. Escogimos estas entidades para la memoria 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.

Diagrama de clases de paquete BusinessLogic

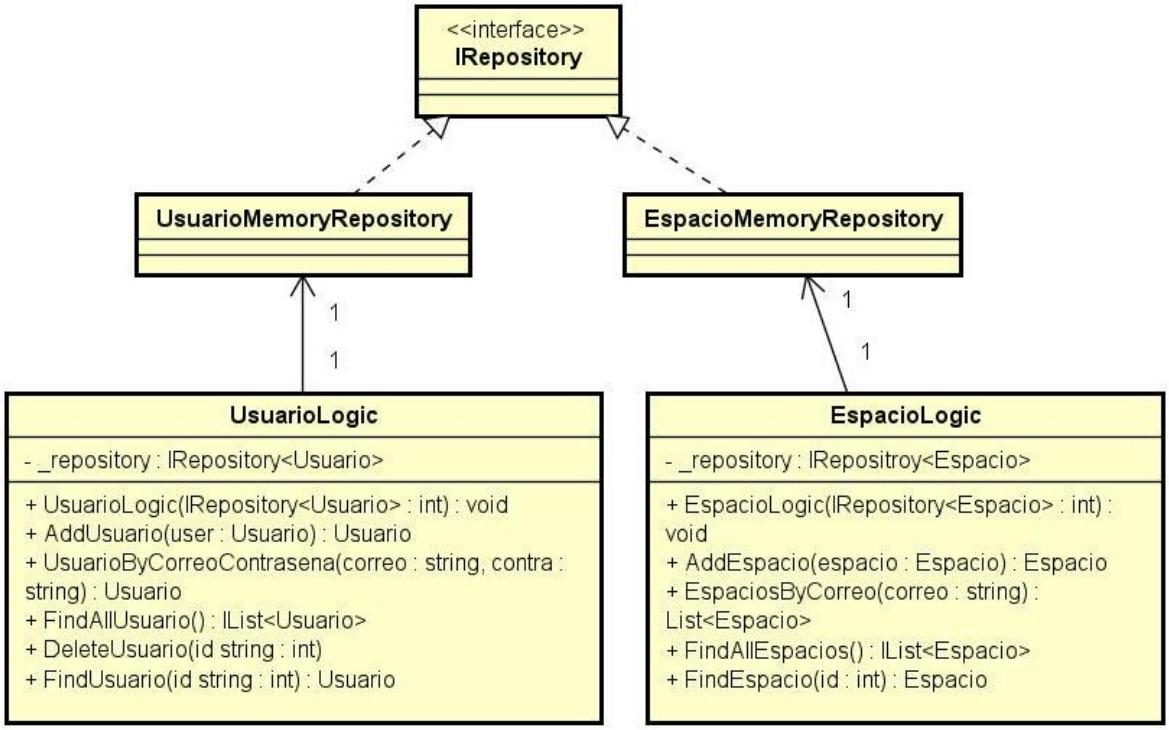


Imagen Diagrama 2.3

Este diagrama corresponde a el namespace de BusinessLogic, 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 BusinessLogicEspacioException y BusinessLogicUsuarioException para capturar las excepciones específicas de la lógica de usuario y espacio. Estas clases heredan de Exception.

Diagrama de clases de paquete Excepcion

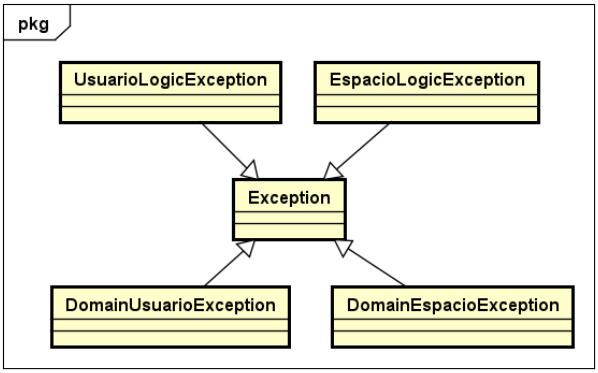


Imagen Diagrama 2.4

Como se muestra en el diagrama 2.4 creamos cuatro excepciones que heredan de Exception, su función es capturar excepciones, definidas por nosotros, cuando no se cumple una regla de negocio en el caso de UsuarioLogicException y EspacioLogicException, ademas estas excepciones nos ayudan a entender en donde se encuentra el problema en caso de una excepcion. Lo mismo para DomainUsuarioException y DomainEspacioException pero a nivel de dominio, para la excepción de usuario se lanzarían al crear un usuario con el formato incorrecto, y para espacio como en él se encuentran todas las clases excepto usuario, nos pareció oportuno simplificar el manejo agregándole DomainEspacioException no solo a Espacio, sino también a Cuenta, Transaccion, Objetivo, Cambio, Categoria. De dichas excepciones utilizamos el método Message() y constructor del padre, no definimos métodos personalizados ya que nuestra aplicación es a pequeña escala.

Diagrama de clases de Interfaz

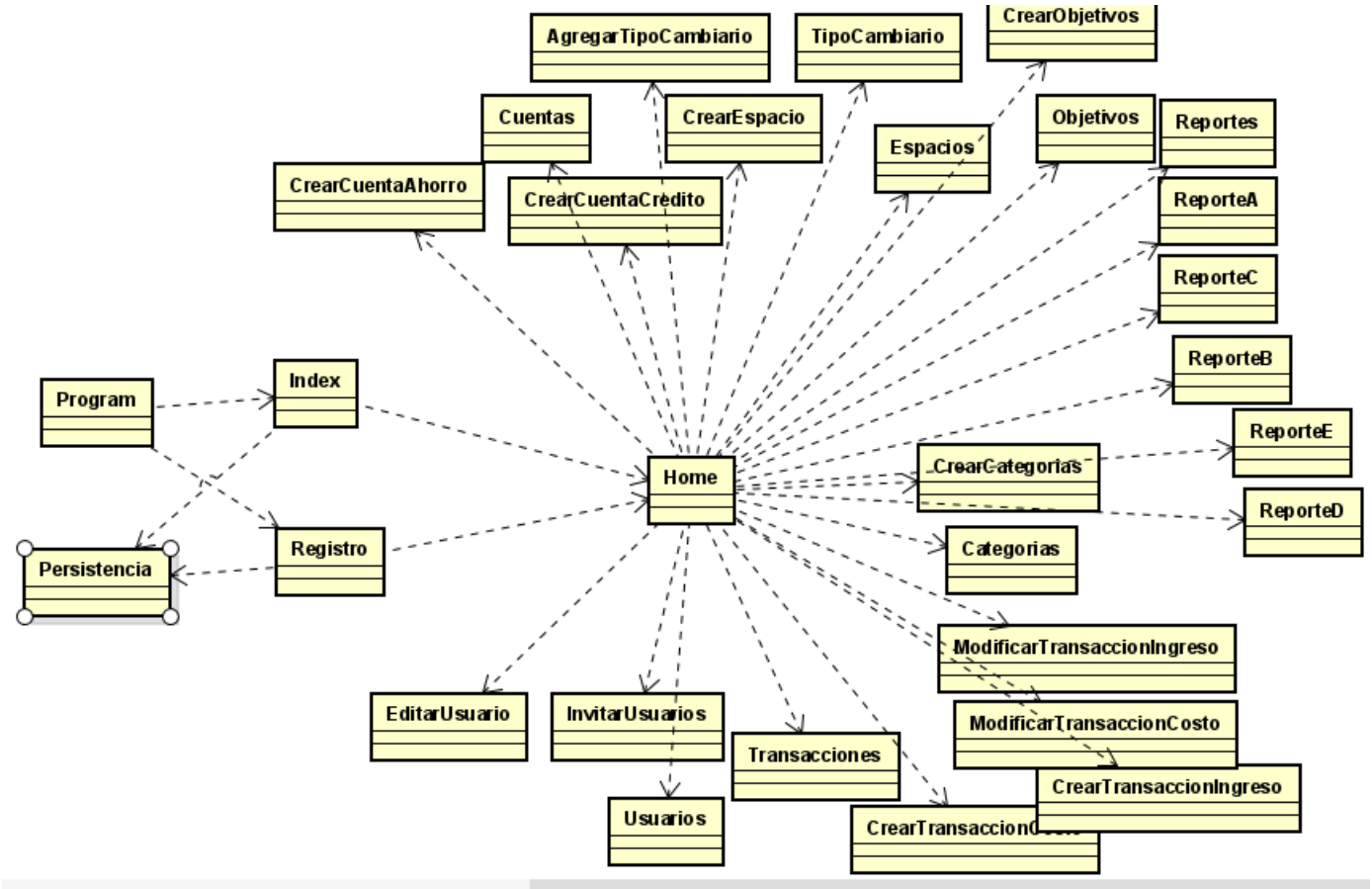


Imagen Diagrama 2.5

En este diagrama mostramos como interactúan las clases Program y Persistencia con las páginas de la interfaz.

Primero la clase Program inicia la página Index o en caso de no estar registrado se hace uso de la página Registro. Luego al ingresar, se pasa a la página Home y se guarda la información del usuario en la clase Persistencia. Una vez en Home se puede acceder a las demás páginas.

Implementación de la interfaz

Para implementar todas las capas de nuestra aplicación en una interfaz web utilizamos blazor server, en el configuramos en la clase Program.cs los siguientes servicios de Singleton:

```
// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<IRepository<Usuario>, UsuarioMemoryRepository>();
builder.Services.AddSingleton<UsuarioLogic>();
builder.Services.AddSingleton<IRepository<Espacio>, EspacioMemoryRepository>();
builder.Services.AddSingleton<EspacioLogic>();
builder.Services.AddSingleton<Persistencia>();
```

Estas llamadas a Services.AddSingleton() están registrando tipos concretos en el contenedor de inyección de dependencias para que estén disponibles en toda la aplicación al utilizarlo se crea una única instancia del servicio durante todo el tiempo de ejecución en memoria en nuestro caso ya que no hay persistencia de datos de una base de datos.

También cabe destacar que utilizamos una clase llamada Persistencia.cs que se encuentra en la carpeta interfaz la cual contiene como atributo un Id int (que utilizamos como identificador del espacio) y un string (correo que identifica a un usuario), de esta también creamos un singleton el cual inyectamos en toda la aplicación.

```
@inject NavigationManager NavigationManager;
@using BussinesLogic
@using Domain
@using EspacioReporte
@using Excepcion
@inject UsuarioLogic UsuarioLogic;
@inject EspacioLogic EspacioLogic;
@inject Persistencia Persistencia;
```

Inyectamos:

NavigationManager para poder utilizar el redireccionamiento de páginas en los eventos, UsuarioLogic que es una instancia por tiempo de ejecución que se sirve de el servicio UsuarioMemoryRepository, lo mismo para EspacioLogic con EspacioMemoryRepository. La interfaz tambien interactua con todas las clases de BussinesLogic Domain EspacioReporte y Excepcion por lo que añadimos un using.

Detalle de cobertura

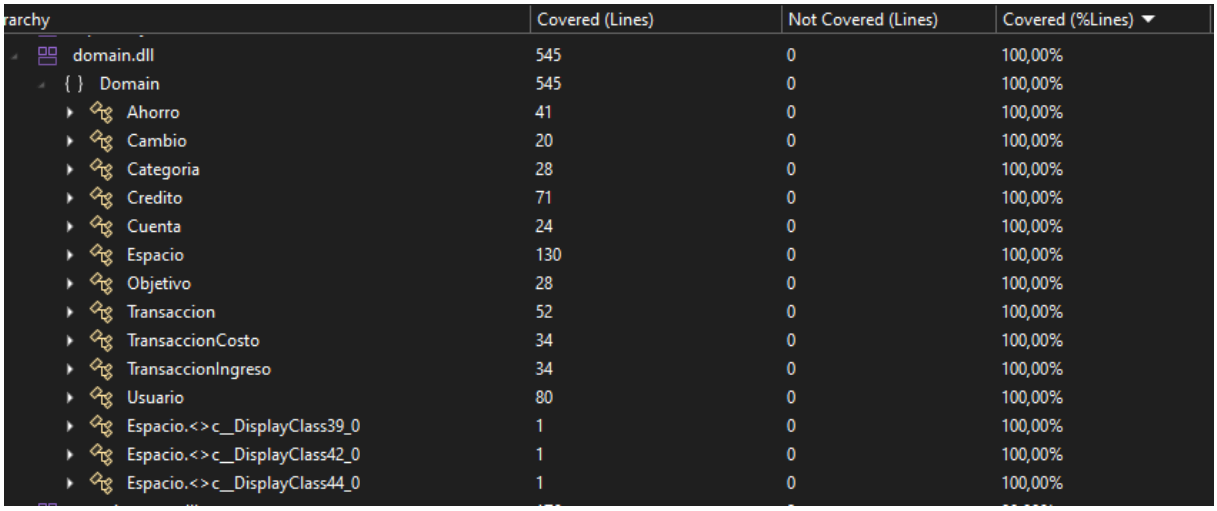
Como se aprecian en la imagen, todos los paquetes Domain, EspacioReporte, Repository, Bussineslogic y

Excepcion fueron testeados aplicando TDD y logrando una cobertura de 100%.

Hierarchy	Covered (Lines)	Not Covered (Lines)	Covered (%Lines)
betin_LAPTOP-CT3721DR_2023-10-12.14_42_09.coverage	4628	96	97,70%
espacioreporte.dll	323	0	100,00%
excepcion.dll	12	0	100,00%
bussineslogic.dll	67	0	100,00%
repository.dll	57	0	100,00%
domain.dll	545	0	100,00%

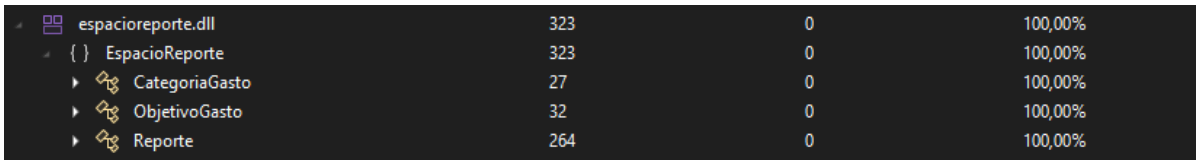
Este porcentaje lo tuvimos muy en cuenta a lo largo de todo el proyecto, asegurándonos de que con los test cubriéramos todos los posibles escenarios de cada funcionalidad.

Cobertura detallada de Domain:



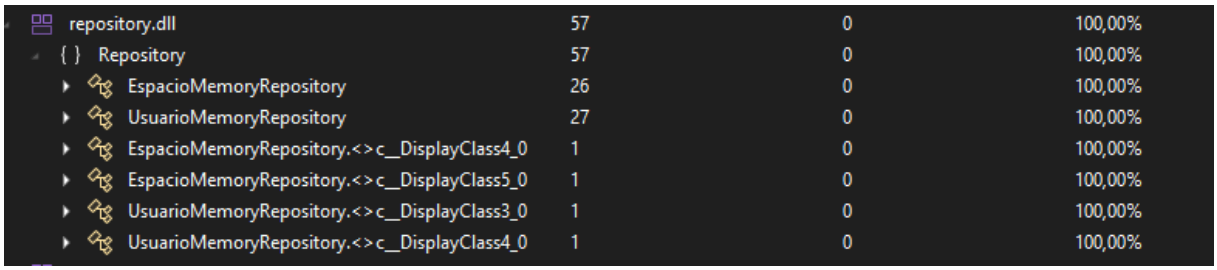
rarchy	Covered (Lines)	Not Covered (Lines)	Covered (%Lines)
domain.dll	545	0	100,00%
{ } Domain	545	0	100,00%
Ahorro	41	0	100,00%
Cambio	20	0	100,00%
Categoria	28	0	100,00%
Credito	71	0	100,00%
Cuenta	24	0	100,00%
Espacio	130	0	100,00%
Objetivo	28	0	100,00%
Transaccion	52	0	100,00%
TransaccionCosto	34	0	100,00%
TransaccionIngreso	34	0	100,00%
Usuario	80	0	100,00%
Espacio.<>c__DisplayClass39_0	1	0	100,00%
Espacio.<>c__DisplayClass42_0	1	0	100,00%
Espacio.<>c__DisplayClass44_0	1	0	100,00%

Cobertura detallada EspacioReporte:



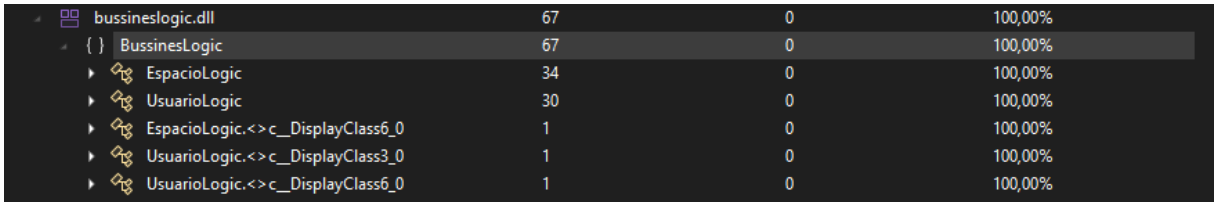
espacioreporte.dll	323	0	100,00%
{ } EspacioReporte	323	0	100,00%
CategoriaGasto	27	0	100,00%
ObjetivoGasto	32	0	100,00%
Reporte	264	0	100,00%

Cobertura detallada de Repository:



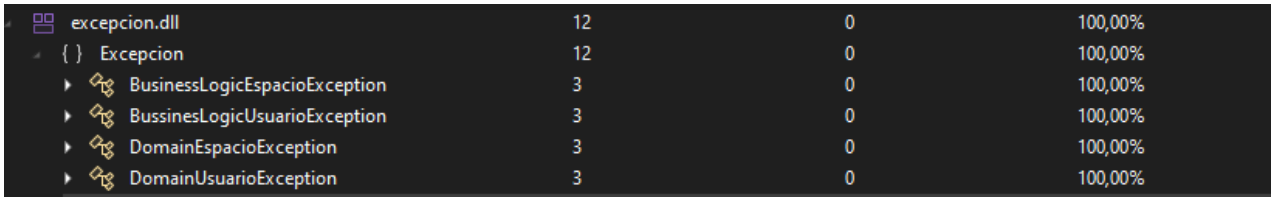
repository.dll	57	0	100,00%
{ } Repository	57	0	100,00%
EspacioMemoryRepository	26	0	100,00%
UsuarioMemoryRepository	27	0	100,00%
EspacioMemoryRepository.<>c__DisplayClass4_0	1	0	100,00%
EspacioMemoryRepository.<>c__DisplayClass5_0	1	0	100,00%
UsuarioMemoryRepository.<>c__DisplayClass3_0	1	0	100,00%
UsuarioMemoryRepository.<>c__DisplayClass4_0	1	0	100,00%

Cobertura detallada de Bussineslogic:



bussineslogic.dll	67	0	100,00%
{ } BussinesLogic	67	0	100,00%
EspacioLogic	34	0	100,00%
UsuarioLogic	30	0	100,00%
EspacioLogic.<>c__DisplayClass6_0	1	0	100,00%
UsuarioLogic.<>c__DisplayClass3_0	1	0	100,00%
UsuarioLogic.<>c__DisplayClass6_0	1	0	100,00%

Cobertura detallada de Excepcion:



excepcion.dll	12	0	100,00%
{ } Excepcion	12	0	100,00%
BusinessLogicEspacioException	3	0	100,00%
BussinesLogicUsuarioException	3	0	100,00%
DomainEspacioException	3	0	100,00%
DomainUsuarioException	3	0	100,00%

Anexo:

Evidencia de pruebas funcionales:

URL a videos en YouTube:

Parte 1 https://youtu.be/oOqG0jKQLPM

Parte 2 https://youtu.be/dHYM93eWMoI

Parte 3 https://youtu.be/A01g106ANjU