Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones 1
Obligatorio 2

Juan Pablo Poittevin(169766)
Joselen Cecilia(233552)

Entregado como requisito de la materia Diseño de aplicaciones 1

26 de noviembre de 2020

Declaraciones de autoría

Nosotros, Juan Pablo Poittevin y Joselen Cecilia, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

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

Resumen

Para la materia Diseño de Aplicaciones 1, en el segundo obligatorio se plantea la elaboración de un sistema que permita a los usuarios registrar y guardar sus transacciones pudiendo elegir entre diferentes tipos de monedas, y generar reporte de estas pudiendo exportar el reporte de gastos. El objetivo de este proyecto es aplicar los conocimientos adquiridos en clase.

Índice general

1. Descripción General

Los expertos en finanzas personales recomiendan el registro de gastos, y aseguran que hacerlo ayuda a identificar patrones de consumo, reconocer categorías en las que se gasta más, establecer objetivos de ahorro; y en general, lograr una mejor planificación financiera.

Este trabajo se enmarca en un proyecto para implementar una aplicación que permita a los usuarios registrar sus transacciones y obtener reportes. En esta primera etapa, el objetivo es la construcción de una prueba de concepto que modele las principales funcionalidades en forma sencilla, y con una interfaz de usuario básica que permita simular el uso de la aplicación.

Las características funcionales que interesan para esta primera versión son:

1 - La aplicación debe mantener un registro de categorías de gastos.

2 - Se debe permitir agregar transacciones individuales con una categoría y monto.

3 - Se debe poder establecer un presupuesto para cada mes, en el que cada categoría tiene un monto estimado de gasto.

4 - Se debe mostrar un reporte por mes de todos los gastos realizados.

5 - Se debe ver un reporte por mes, en el que se muestra para cada categoría el monto estimado y el real.

Además se agregaron nuevas funcionalidades:

6-Soporte multimoneda, se permite el registro de nuevas monedas en el sistema.

7-Se permite al registrar gastos seleccionar el tipo de moneda para este.

8-En el reporte de gasto se muestra el símbolo de la moneda en la que se encuentra el gasto.

9-Se permite exportar el reporte de gasto consultado, a un archivo TXT o CSV.
10-Se permite ver el listado de los objetos que se encuentran registrados en la base de datos.

Al ingresar a la aplicación el usuario se encuentra con una lista de botones cada uno referido a las distintas funciones del programa, registrar categoría, registrar gasto, registrar o editar presupuesto, registrar moneda, reporte de gastos, reporte de presupuestos, editar categoría y editar gastos, listado de objetos del sistema.

La lógica de negocio del programa fue desarrollada utilizando TDD (desarrollo guiado por pruebas) y siguiendo las buenas prácticas propuestas por Clean Code, el cual habla de como escribir código limpio, además hicimos uso de los patrones de diseño GRASP y los principios SOLID.

La persistencia de datos se realizo utilizando Entity Framework, para poder trabajar con la base de datos creamos objetos Dto que se guardan en esta, cada uno de estos se corresponde con cada objeto del dominio. Estos objetos se mapean de Dto(Base de datos) a dominio y viceversa

Para el versionado trabajamos con Git, que es una herramienta utilizada para el control de versiones del software. Esto nos permite tener versiones antiguas estables en caso de necesitarlas.

Para especificar la estructura interna del repositorio nos basamos en la arquitectura GitFlow, donde existe una rama master, de la cual luego se desprende una rama develop.

De la rama develop se crean las ramas "FEATURE BRANCHES", cada una de estas se corresponderá con una clase o una característica del proyecto.
Una vez teniendo la nueva característica o clase estable, tenemos que hacer merge con la rama develop. Además creamos features branches para el refactoring donde hicimos todos los cambios necesarios para mejorar la calidad del proyecto. Al final cuando se termine el proyecto se hara una release branch desde la ultima versión de develop hacia la rama master, haciendo merge con esta ultima.

2. Descripción general y justificación de diseño

2.1. Diagrama de paquetes

En nuestro obligatorio tenemos cuatros paquetes BusinessLogic, InterfazLogic, DataAccess y Test. Dentro de estos paquetes agrupamos las clases en namespaces para facilitar el manejo de estas, es decir poder encontrarlas mas rápido y organizarlas, las clases de cada paquete ser especificadas junto con las relaciones entre ellas en el diagrama de clases en el siguiente punto.

Dentro del paquete BusinessLogic, se encuentran las clases de manejo de lógica y objetos, en este paquete tenemos cuatro namespaces, domain, controller, exceptions y repository en las cuales se encuentran las clases que detallaremos mas adelante.

Dentro del paquete InterfaceLogic se encuentran las clases que manejan la interfaz, dentro de este paquete tenemos tres namespaces add, edit y report en las cuales se encuentran las clases que detallaremos mas adelante.

Dentro del paquete DataAccess se encuentran las clases que manejan la base de datos, dentro de este paquete tenemos cuatro namespaces DBObjects, Managers, MAppers y Migrations en las cuales se encuentran las clases que detallaremos mas adelante.

Dentro del paquete Test se encuentran las clases con los test, correspondientes a cada clase del paquete businessLogic, este paquete se encuentra dividido en dos grandes namespaces businesslogicTest y dataAccesTest

En el siguiente diagrama vemos la estructura explicada anteriormente:

[**Anexo 1**]

El paquete DataAcces referencia al paquete BusinessLogic.

El paquete InterfaceLogic referencia al paquete DataAccess y BusinessLogic.

Y por ultimo el paquete test referencia a businessLogic y a interfaceLogic.

Mas adelante explicaremos como se conectan entre si la lógica con la interfaz y mediante que clases o hace.

2.2. Diagrama de clases

En estos diagramas veremos que clases componen cada paquete y como se relacionan estas clases entre si, mas adelante explicaremos que función cumple cada una de estas.

-Diagrama del paquete BusinessLogic

[**Anexo 2**]

-Diagrama del paquete DataAccess

[**Anexo 3**]

Mostramos además de las clases de este paquete, las clases IRepository y ManagerRepository que son del paquete BusinessLogic para mostrar como se relacionan.

-Diagrama del paquete InterfazLogic

[**Anexo 4**]

Mostramos además de las clases de este paquete, la clase ManagerRepository y los controlladores que son del paquete BusinessLogic para mostrar como se relacionan.

A su vez las clases de estos paquetes interactúan entre ellas, como mostramos en los diagramas de interacción. Y como vamos a explicar específicamente para cada clase en la sección de asignación de responsabilidades.

2.3. Diagrama de interacción

En el primer diagrama vemos como interactúa el usuario con las ventanas y como pasan estas la informacion a los controladores para luego guardar la data en la base de datos.

Para esto usamos el ejemplo de registrar un expense, asumiendo que los datos son todos correctos. Este diagrama es muy similar al de registrar el resto de los objetos de dominio, elegimos uno para ilustrar el funcionamiento.

[**Anexo 5**]

En este segundo diagrama vemos como maneja los errores el sistema, para esto

usamos como ejemplo ilustrativo registrar una categoría donde el nombre o las keywords pueden tener errores.

Como vemos en la imagen, cuando se encuentra un error en alguno de los atributos, al setearlos para crear el objeto, se lanza una excepción correspondiente al error.
Estas excepciones son capturadas luego en la interfaz mediante un catch(excepción), donde se setea un mensajes para mostrar al usuario cual es el error, y así poder darle a este un feedback de lo que sucede en el sistema.
Este es un ejemplo especifico de manejo de errores pero el procedimiento es para todos igual, se lanza la excepción en la lógica y se captura en la interfaz mostrando un mensaje.

[**Anexo 6**]

Por ultimo en este diagrama vemos una visión general de como interactúan las clases entre si, desde las clases del paquete interface, por las clases de businessLogic hasta las clases del DataAccess.

Desde el user control correspondiente a la funcionalidad se llama al método que se quiere usar mediante el controller del objeto quien luego se encarga de llamar a los métodos de las demás clases , para ya sea guardar un objeto, actualizarlo, encontrarlo o lo que se desee hacer.

[**Anexo 7**]

2.4. Estructura de base de datos

Dentro de la base de datos se encuentran distintas tablas, correspondientes a los objetos del sistema, en el siguiente diagrama se muestra estas tablas y que columnas contienen además se muestra la PK y FK de cada una.
[**Anexo 8**]

2.5. Suposiciones de diseño

Para realizar el diseño del obligatorio nos basamos en la letra del mismo y en las preguntas que se fueron realizando en el foro, además hicimos algunos supuestos:

-Se puede crear un expense con los mismos description, amount, creationDate y category ya existentes en otro expense. Esto surgió de la necesidad de que un cliente pude ir dos veces en el mismo día al mismo lugar y querer registrar el mismo gasto.

-Al ingresar una nueva category se verifica que no se ingrese dos veces la misma "key word", en dicha category, ya que no tendría sentido tener dos veces la misma palabra clave en la misma category.

-Se crea una clase budgetCategory, la misma nos facilita el control del presupuesto para cada una de las categorías.

-No se permite que el nombre de una categoría sea solo números, ya que a nuestro parecer no tendría mucho sentido.

-Decidimos utilizar en la lógica un enum month para representar la variable month de las distintas clases ya que es en estas donde se trabaja con dicha variable, y en las ventanas un string ya que se obtiene así de los campos de estas.

-En las ventanas decidimos separar el registrar gasto del eliminar y editar gastos ya que creemos que facilita el uso y entendimiento de la funcionalidad al usuario.

-También decidimos separar el registrar categoría del editar categoría ya que creemos que facilita el uso y entendimiento de la funcionalidad al usuario.

-En cambio registrar presupuesto y editar presupuesto los dejamos en la misma ventana ya que el editar, simplemente agrega un botón de editar presupuesto por categoría, donde el usuario puede cambiar el monto de este, por lo que creíamos innecesario tener que separarlas.

-Se puede eliminar una moneda solo si esta no se encuentra en uso, es decir si no hay un gasto que tenga su monto en esta.

-Se encuentra registrada como moneda por defecto en el sistema el peso uruguayo, esta se crea automáticamente y no se puede editar.

-Decidimos separar el registrar moneda del editar moneda ya que creemos que facilita el uso y entendimiento de la funcionalidad al usuario.

-Hicimos clases mappers para todos los objetos con tres métodos que mapean los objetos del dominio a la base de datos, y viceversa, y hacen el update de los objetos si estos existen ya en la base de datos. A excepción del objeto budgetCategories que no tiene una clase mapper ya que esta contenido dentro de la clase Budget, y queremos mantenerlo oculto al resto del sistema.

- Para la carga de datos a la base de datos dejamos en false el lazy loading, porque generaba problemas con los mappers. Usamos eager loading.

- Asumimos que para el correcto funcionamiento del sistema, quien lo vaya a utilizar deberá tener correctamente configurado el connectionString en el archivo App.config, el ejcutable se encuentra en la raiz del repositorio.

-En los métodos mappers tenemos que validar que los objetos devueltos por el firstOrDefault no sean nulos, es por esto que hacemos if objeto is null, ya que este método es ajeno a nosotros decidimos validar esa condición en lugar de tirar una

excepción como recomienda clean code,de lo contrario puede generar errores.

Decidimos usar objetos DTO correspondientes a cada objeto del dominio para persistir en la base de datos donde indicamos cual es el ID de cada uno y la relación entre ellos y de esta forma no modificar las clases del dominio.

Se decidieron crear objetos intermedios para los reportes de gastos y presupuesto.
Al crear dichos objetos encapsulamos la información en estos, haciendo fácil el manejo y paso de esta.

-Para los test creamos una base de datos distinta a la del dominio, la misma solo se utiliza en la ejecución de estos.
La configuración de esta basa de datos se encuentra en el .ªpp.config"del proyecto test.

2.6. Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas.

2.6.1. Diseño actual

Para la implementación de este sistema creamos cuatro proyectos; dos Class Library .NET Framework a los cuales llamamos BusinessLogic y DataAccess, un MSTest .NET Framework al cual llamamos Test y por ultimo un WindowsForms .NET Framework al cual llamamos InterfaceLogic

BusinessLogic

Dentro de BusinessLogic creamos tres carpetas para agrupar las clases, y asi facilitar la busqueda de estas.

El namespace Controller contiene las clases para comunicar la interfaz de usuario con la lógica de negocio, estas clases son handlers para las distintas funcionalidades, uno que se encarga del registrar y editar categoría (CategoryController), otro que se encarga de registrar, dar reporte y editar gasto (ExpenseController), uno que se encarga de registrar, editar y listar las currencies (CurrencyController) y por ultimo uno que se encarga de registrar, dar reporte y editar presupuesto (BudgetController).
La ventaja de mantener separada la lógica de la interfaz es que si en un futuro se decide usar esta lógica para otro tipo de aplicación, por ejemplo una aplicación web, se puede adaptar fácilmente.

El namespace Domain contiene las clases correspondientes a los principales objetos del sistema Expense, Budget, Category, BudgetCategory. Tambien se encuentra la clase EnumMonths que contiene el enum Month el cual lista los meses del año, las clases que permiten exportar el reporte de expense IExportReport, ExpenseReportCSV, ExpenseReportTXT y las clases que permiten generar el reporte de expense y budget GenerationExpenseReport, GenerationBudgetReport, BudgetReportLine, BudgetReportLine.

El namespace Repository, que contiene la Interface IRepository generic y el managerRepository para invertir la dependencia y que el dataAcess dependa de businessLogic, de esta forma el businessLogic marca al dataAccess que funcionalidades debe implementar.

El namespace Expeptions que contiene las clases con las excepciones creadas.

DataAcess

En el paquete DataAcces creamos cuatro carpetas que agrupan las clases para facilitar la búsqueda de estas.

La carpeta DBOBjects contiene las clases de los objetos de tipo Dto, respectivo a cada objeto del dominio, estos objetos se persisten en la base de datos, además se indica en cada uno la información especifica para almacenarlos en la base de datos, como el ID y las relaciones.

La carpeta Managers contiene las clases DataBaseManagerRepository , ManageMemoryRepository los mismo centralizan la creación de los IRepository, que actualmente pueden ser de base de dato o memoria.

La carpeta Mappers contiene las clases que se encargan de mappear los objetos del dominio a objetos de la base de datos y viceversa, y los métodos update que se encargan de actualizar los objetos si ya están registrados en la base de datos.
Hay una de estas clases mappers por cada objeto del dominio, a excepción de como mencionamos anteriormente BudgetCategory.

La carpeta Migrations donde se almacenan las migraciones de los datos a la base de datos.

Además de estas clases mencionadas tenemos las clases, ContextDB que contiene los DbSet de cada uno de los objetos dto, es decir contiene las tablas de la base de datos y DataBaseRepository que implementa los métodos de IRepository en base de datos y MemoryRepository que implementa los métodos de IRepository en memoria.
El IRepository facilito el adaptar el proyecto que antes se almacenaba en memoria a que se almacenar en una base de datos, ya que junto con las clases mappers solo debimos implementar los seis métodos de la interface lo que demuestra lo fácil qué seria poder agregar mas tipos de almacenamiento.
Creemos que los métodos mappers al tratarse de un repository generic centralizan

la mayor dificultad del proyecto, pero decidimos arriesgarnos ya que la idea de que nuestro sistema se pudiera adaptar tan fácilmente a otra forma de almacenamiento nos llamo mucho.

Test

Dentro de Test tenemos las clase Tests correspondientes a cada clase del paquete BusinessLogic que tiene lógica, en las cuales fuimos creando los test para generar la lógica del sistema, para los test se cuentan con una base de datos propia la cual puede borrar e insertar datos cuando sea necesario para probar distintos escenarios.

InterfaceLogic

Dentro de InterfazLogic tenemos una clase UserControl por cada funcionalidad del sistema agrupadas en carpetas según si agregan, editan o reportan. También tenemos una clase Formulario que es el Menú principal, el cual contiene los botones a todas las funcionalidades,y dos formularios uno para editar las key words y otro para editara el budget category

La interfaz del proyecto se comunica con la lógica del sistema quien se encarga de almacenar y manejar los datos, es por esto que cada user control tiene su correspondiente controller el cual se encuentra en la lógica del sistema.

2.6.2. Aplicación de patrones GRASP y principios y patrones de diseño

Las interfaces creadas en el sistema IRepository, IManageRepository, Imapper y IExportExpenseReport se pensaron cumpliendo con el **principio de Segregacion de la Interfaz**, es decir que las clases que implementan estas no esten obligados a implementar métodos que no usa, para esto se intento que estas clases fueran muy especificas en cuanto a que metodos poner como parte del contrato.

El uso de una interfaz para el repositorio Irepository nos permite minimizar, como vimos en el transito del obligatorio 1 al obligatorio 2, el impacto del cambio en la forma de almacenamiento, ya que para implementar una nueva forma de almacenamiento basta con crear una nueva implementación de esta interfaz y cambiar las instancias de esta, lo que sugiere el **patrón GRASP variaciones protegidas**, implementar interfaces alrededor de las clases que es posible cambien en un futuro. Lo mismo sucede con el poder exportar los gastos en archivos csv y txt, para esta funcionalidad se creo la clase IExportExpenseReport ya que en un futuro se podrían agregar nuevos tipos de archivos, y para esto solo haría falta agregar una nueva clase que implemente esta interfaz, la cual en este momento es implementada por dos clases ExportReportTXT y ExportReportCSV correspondientes a los tipos solicitados en el obligatorio.

Además implementamos el **patrón strategy** para getReport junto con un factory (**patrón GRASP Fabricación pura**), entre el context y el strategy para simplificar el problema de qué strategy usar(ExportReportTXT o ExportReportCSV).

Lo mencionado anteriormente también favorece el patrón bajo acoplamiento, ya que con las interfaces favorecemos la baja dependencia y el bajo impacto del cambio, si se agregan como dijimos antes nuevas formas de almacenamiento o nuevas formas de exportar gastos el código sigue funcionando y no tenemos que cambiar las clases existentes si no que debemos agregar nuevas implementaciones de las interfaces para los nuevos tipos, haciendo con esto que el código sea extensible a nuevos cambios.

El usar estas interfaces no lleva a cumplir con el **principio open/closed**, ya que la firma de las interfaces no deberían poder tocarse porque al cambiar esto deberíamos cambiar todas las implementaciones, es decir que la clase es cerrada a la modificación, pero permite como mencionamos agregar nuevos tipo, es decir es abierta a la extensión.
Esto cumple también con el **patrón GRASP polimorfismo** permite variar los tipos sin usar lógica condicional

EL paquete businessLogic no depende del paquete dataAccess sino que ambos dependen de abstracciones, lo que cumple con el con el **principio de inversión de dependencias**

Con la separación de la interfaz de la lógica mediante los controladores,handlers, logramos que el repositorio se encargue únicamente del almacenamiento de datos, lo que cumple con el **principio Single responsability**, además cumple con el **patrón controlador** ya que resuelve el problema de manejar los eventos de entrada y salida del sistema.
El resto de las clases cumplen también con el **principio de single responsability**, lo que hace que también se cumpla el **patrón de alta cohesión**, ya que estas agrupan funciones que tienen relación entre ellas,que tienen las mismas responsabilidades.

En las clases Category y Budget se aplica el **patron creador** con las keyWords y budgetCategories.

2.6.3. Asignación de responsabilidades

Junto con la descripción de cada clase se indica que patrones y principios se aplicaron, estos se explicaron mas detalladamente en la sección anterior.

Category
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Crear los objetos de tipo Category.
-Manejar los datos de la categoría, nombre y palabras claves.
-Pasar a string la categoría.
-Ver si dos categorías son iguales.

Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, patrón creador, principio de responsabilidad única.

Expense
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:

-Crear los objetos de tipo Expense.
-Manejar los datos de los gastos, monto del gasto, fecha en la que se creo el gasto, descripción del gasto y categoría de este.
-Ver si dos gastos son iguales.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única.

Budget
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Crear los objetos de tipo Budget.
-Manejar los datos del presupuesto, monto total, mes y -año de cuando se crea, y una lista de -BudgetCategories.
-Pasar a string el presupuesto.
Ver si dos presupuestos son iguales.

Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, patrón creador, principio de responsabilidad única.

BudgetCategory
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Crear los objetos de tipo BudgetCategory.
-Manejar los datos del presupuesto, monto y categoría.
-Pasar a string el budgetCategory.
-Ver si dos budgetCategory son iguales.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única.

KeyWord
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Manejar el valor de tipo string de la palabra clave.
-Ver si dos keyWord son iguales y si una keyWord se encuentra ya en una lista de strings.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única

Currency
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Crear los objetos de tipo Currency.
-Manejar los datos de currency, Name, Symbol y Quotation.
-Pasar a string Currency.

-Ver si dos Currency son iguales.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única.

BudgetController Esta clase se encuentra dentro del paquete BusinessLogic:
-Es la fachada, el handler, entre Budget y la interfaz, es decir mediante esta clase se comunica la lógica del budget con las ventanas de funcionalidades de este objeto.
Patrones y principios aplicados y relevantes a esta clase Alta cohesión, patrón creador, bajo acoplamiento, patrón controlador, patrón indirección, principio de responsabilidad única.

CategoryController Esta clase se encuentra dentro del paquete BusinessLogic

-Es la fachada, el handler, entre category y la interfaz, es decir mediante esta clase se comunica la lógica de la category con las ventanas de funcionalidades de este objeto.
Patrones y principios aplicados y relevantes a esta clase Alta cohesión, patrón creador, bajo acoplamiento, patrón controlador, patrón indirección, principio de responsabilidad única.

ExpenseController Esta clase se encuentra dentro del paquete BusinessLogic:
-Es la fachada, el handler, entre expense y la interfaz, es decir mediante esta clase se comunica la lógica del expense con las ventanas de funcionalidades de este objeto.
Patrones y principios aplicados y relevantes a esta clase Alta cohesión, patrón creador, bajo acoplamiento, patrón controlador, patrón indirección, principio de responsabilidad única.

CurrencyController Esta clase se encuentra dentro del paquete BusinessLogic:
-Es la fachada, el handler, entre Currency y la interfaz, es decir mediante esta clase se comunica la lógica de la currency con las ventanas de funcionalidades de este objeto.
Patrones y principios aplicados y relevantes a esta clase Alta cohesión, patrón creador, bajo acoplamiento, patrón controlador, patrón indirección, principio de responsabilidad única.

IRepository
Esta clase se encuentra dentro del paquete BusinessLogic y se encarga de:
-Mostrar la firma, de los métodos básico para manejar un Repository, sin importar de que tipo sea el almacenamiento.
-Es el contrato de este,es una clase interfaz generic. **Patrones y principios aplicados y relevantes a esta clase** Alta cohesión, bajo acoplamiento, patrón variaciones protegidas, patrón polimorfismo, principio de responsabilidad única , principio Open/Closed, principio de segregación de la interfaz, principio de inversión de la responsabilidad.

ManagerRepository
Esta clase abstracta se encuentra dentro del paquete BusinessLogic y se encarga de:

-Instanciar los distintos IRepository que utiliza el sistema, explicitando los Yrepository que se deben implementar.
Patrones y principios aplicados y relevantes a esta clase bajo acoplamiento, patrón variaciones protegidas, patrón creador, principio de responsabilidad única , principio Open/Closed, principio de inversión de la responsabilidad.

BudgetCategoryDto
Esta clase se encuentra dentro del paquete DataAcess y se encarga de: -Implementar las funcionalidades utilizadas para persistir la información de budgetCategroy en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión, patrón bajo acoplamiento, principio de responsabilidad única

BudgetDto Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar las funcionalidades utilizadas para persistir la información de budget en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión, patrón bajo acoplamiento, principio de responsabilidad única

CategoryDto
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar las funcionalidades utilizadas para persistir la información de category en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión, patrón bajo acoplamiento, principio de responsabilidad única

CurrencyDto
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar las funcionalidades utilizadas para persistir la información de currency en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión, patrón bajo acoplamiento, principio de responsabilidad única

ExpenseDto
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar las funcionalidades utilizadas para persistir la información de expense en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión, patrón bajo acoplamiento, principio de responsabilidad única

KeyWordsDto
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar las funcionalidades utilizadas para persistir la información de keyWord en la base de datos.
Patrones y principios aplicados y relevantes a esta clase: Alta cohesión,

patrón bajo acoplamiento, principio de responsabilidad única

DataBaseManagerRepository
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Instanciar los objetos de tipo Irepository con objetos DataBaseRepository, esta clase hereda de ManagerRepository.
Patrones y principios aplicados y relevantes a esta clase: principio de responsabilidad única, patrón alta cohesión, patrón bajo acoplamiento

ManagerMemoryRepository
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Instanciar los objetos de tipo Irepository con objetos MemoryRepository, esta clase hereda de Manage Repository, la creamos para controlar que la dependencia de paquetes fuera de bajo nivel a alto.
Patrones y principios aplicados y relevantes a esta clase:principio de responsabilidad única, patrón alta cohesión, patrón bajo acoplamiento

IMapper
Estas clases se encuentran dentro del paquete DataAcess y se encargan de:
-Especificar la firma de los métodos para Mappear los objetos del dominio a objetos de la base de datos y viceversa, y hacer el update de estos.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión, patrón bajo acoplamiento, patrón polimorfismo, patrón variaciones protegidas, principio Open/Closed, principio de responsabilidad única, principio de segregación de la interfaz.

Mappers
Estas clases se encuentran dentro del paquete DataAcess y se encargan de:
-Implementar la interfaz IMapper que tiene los métodos para Mappear(transformar de dominio a dto) los objetos del dominio a objetos de la base de datos y viceversa, y hacer el update de estos objetos.
-Esto se hace teniendo en cada claseMapper tres métodos un método que convierte los objetos de tipo dominio a objetos DTO, un método que convierte los objetos de tipo DTO a objetos de dominio, junto con un método Update, que se fija si el objeto ya existe en la base de datos loa actualiza y de lo contrario lo crea.
Patrones y principios aplicados y relevantes a esta clase: patrón bajo acoplamiento, patrón alta cohesión, patrón polimorfismo, principio de responsabilidad única.

MemoryRepository
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implmentar la clase Irepository, por lo tanto al igual que esta es una clase generic, implementa esta interfaz para el almacenamiento en memoria.
-Esta clase también la utilizamos como un fake de la base de datos en los unit test.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única, patrón polimorfismo, patrón bajo bajo aco-

plamiento, principio de segregación de la interfaz.

DataBaseRepository
Esta clase se encuentra dentro del paquete DataAcess y se encarga de:
-Implementar la clase Irepository, por lo tanto al igual que esta es una clase generic, se implementa para el almacenamiento en base de dato, esta clase usa dos tipos genéricos donde uno de ellos representa los objetos de dominio y el otro los objetos de la base de datos, los cuales son mappeados con los métodos mappers que mencionaremos previamente.
Patrones y principios aplicados y relevantes a esta clase: patrón alta cohesión, principio de responsabilidad única, patrón polimorfismo, patrón bajo bajo acoplamiento, principio de segregación de la interfaz.

ContextDB
Esta clase se encuentra dentro del paquete DataAcess: -Esta clase hereda de la clase DbContext la cual permite consultar, crear, editar y eliminar registros en una base de datos. En esta clase se encuentran almacenados en DbSets los objetos de tipo Dto respectivos a los objetos del dominio.
Patrones y principios aplicados y relevantes a esta clase: principio de responsabilidad única, patrón bajo acoplamiento y patrón alta cohesión.

IExportExpenseReport
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Especificar la firma del método exportReport, que luego sera implementado por los distintos tipos de reporte.
Tener esta interfaz junto con la clase factory de la cual hablaremos luego nos permite que el exportar reporte de gastos sea extensible a nuevos tipos de archivos que no sean TXT o CSV, sin necesidad de tocar el código existente, ya que simplemente deberíamos crear una nueva clase exportType que implemente esta interface para el nuevo tipo.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón polimorfismo, patrón variaciones protegidas, principio Open/Closed, principio de responsabilidad única, principio de segregación de la interfaz, patrón strategy.

ExportReportTXT
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Implementar la clase IExportExpenseReport, para el tipo de archivo .txt, es decir implementa el método exportReport de dicha interfaz guardando el reporte en un archivo .txt con el formato indicado.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón polimorfismo, principio de responsabilidad única.

ExportReportCSV
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:

-Implementar la clase IExportExpenseReport, para el tipo de archivo .csv, es decir implementa el método exportReport de dicha interfaz guardando el reporte en un archivo .csv con el formato indicado.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón polimorfismo,principio de responsabilidad única.

FactoryExportReport
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Esta clase se encarga de decidir quien instancia a la interfaz IExportReport mediante un string que viene del user control ExpenseReport que indica el tipo ("txt.º çsv"). Es una clase artificial, no nace de la letra del problema por eso la llamamos factory.
Patrones y principios aplicados y relevantes a esta clase: Patrón fabricación pura,patron bajo acoplamiento, patrón polimorfismo, patrón variaciones protegidas,principio de responsabilidad única.

ExpenseReport

Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Crear el objeto para el reporte de expense, para esto contiene una lista de la lineas del reporte y el monto total para luego usarlo en la interfaz y así evitar hacer cálculos en esta.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón controlador, principio de responsabilidad única.

ExpenseReportLine
Esta clase se encuentra dentro del paqueteBussinesLogic y se encarga de:
-Esta clase tiene como atributos las columnas necesarias para el reporte de gastos, estos atributos se setean en el método getExpenseReport de expense controller, y como mencionamos anteriormente este objeto se encuentra almacenado en una lista en la clase ExpenseReport
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón controlador, principio de responsabilidad única.

BudgetReport
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Crear el objeto para el reporte de budget, para esto contiene una lista de la lineas del reporte y el monto total, la diferencia y el moto planificados valores que se necesitan mostrar en el reporte, para luego usarlo en la interfaz y así evitar hacer cálculos en esta.
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón controlador, principio de responsabilidad única.

BudgetReportLine
Esta clase se encuentra dentro del paquete BussinesLogic y se encarga de:
-Esta clase tiene como atributos las columnas necesarias para el reporte de presupuestos, estos atributos se setean en el método getBudgetReport de budgetController, y como mencionamos anteriormente este objeto se encuentra almacenado en una lista en la clase BudgetReport
Patrones y principios aplicados y relevantes a esta clase: Patrón alta cohesión,patrón bajo acoplamiento, patrón controlador, principio de responsabilidad única.

2.6.4. Mejoras de diseño

Como mejoras para la solución del obligatorio nos quedaron dos pendientes por falta de tiempo :

1-Poder hacer de los mappers que al día de hoy tenemos específicos para cada objeto, un mapper genérico, haciendo que reciban una clase origen y una clase destino.

2.Implementar los update en memoria que al día de hoy están implementados solo en base de datos, que como no es parte de este obligatorio y como no nos dio el tiempo no quedaron implementados, se agregan suplantando el add seguido de un delete juntando estas operaciones en una.

2.7. Pruebas

2.8. Cobertura de pruebas unitarias

Durante las pruebas unitarias, como fue mencionado previamente, se utilizo la metodologıa de TDD.

Lo que nos llevo a poder alcanzar un porcentaje de cobertura del 98 por ciento de la lógica de negocio.
Un porcentaje de prueba que creemos fue superior al que hubiéramos tenido de no utilizar la metodología de TDD.

Desarrollar el código de esta forma nos permitió ver claramente los casos de cada función, donde fallaba, que casos limites validar, y que variantes tener en cuenta.

[**Anexo 9**]

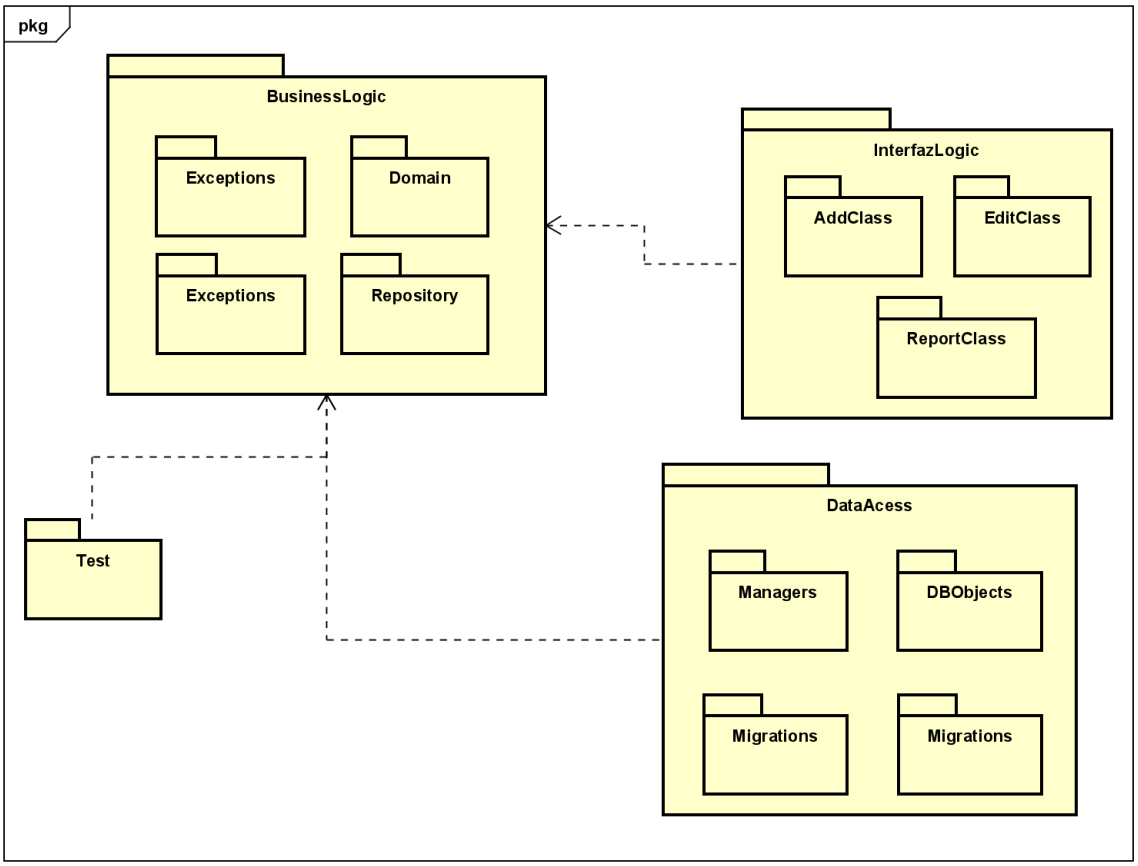
2.9. Pruebas de integración

Además de los unit test se hicieron test de integración con la base de datos. Los mismos se encuentran dentro del namespace DataAccess.Test

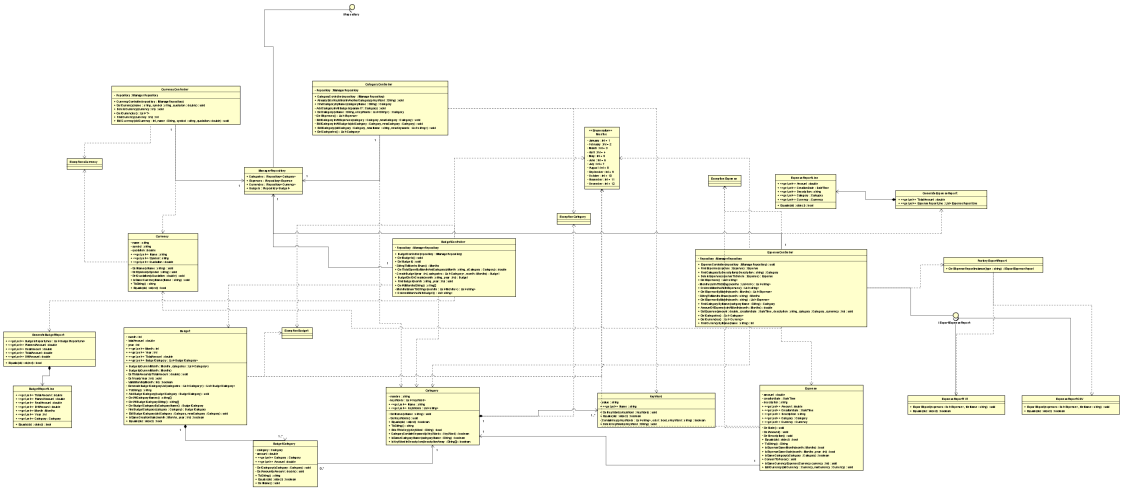
3. Anexo

Se adjuntan los archivos en la carpeta del proyecto porque al ser muy grande el diagrama no se distinguen bien los métodos

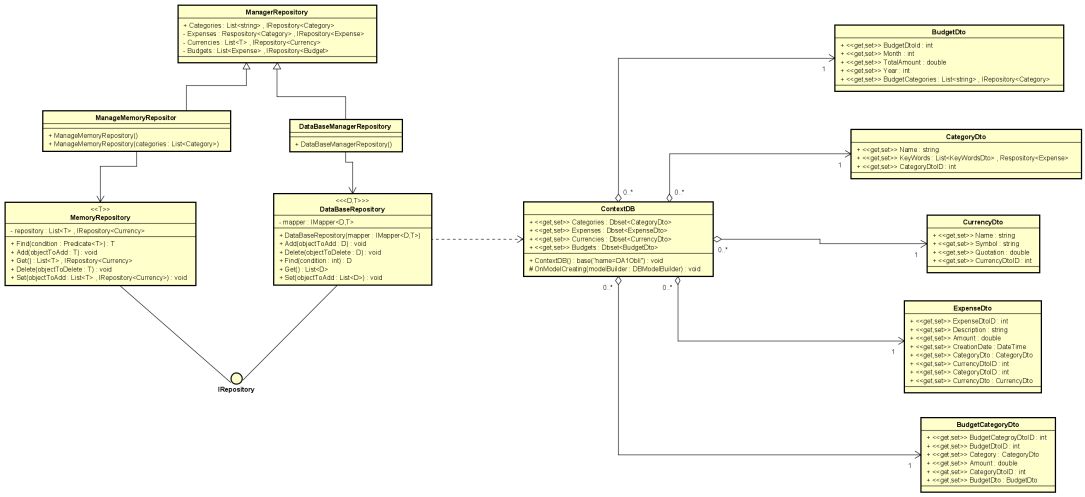
[Anexo 1] Diagrama de paquetes



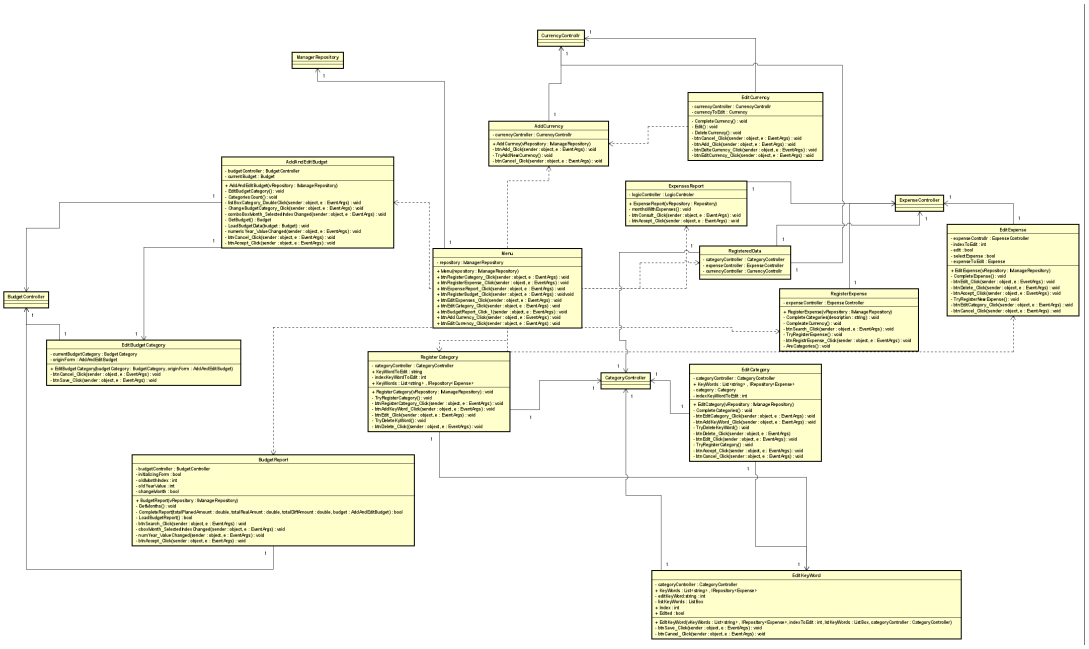
[Anexo 2] Diagrama de clases



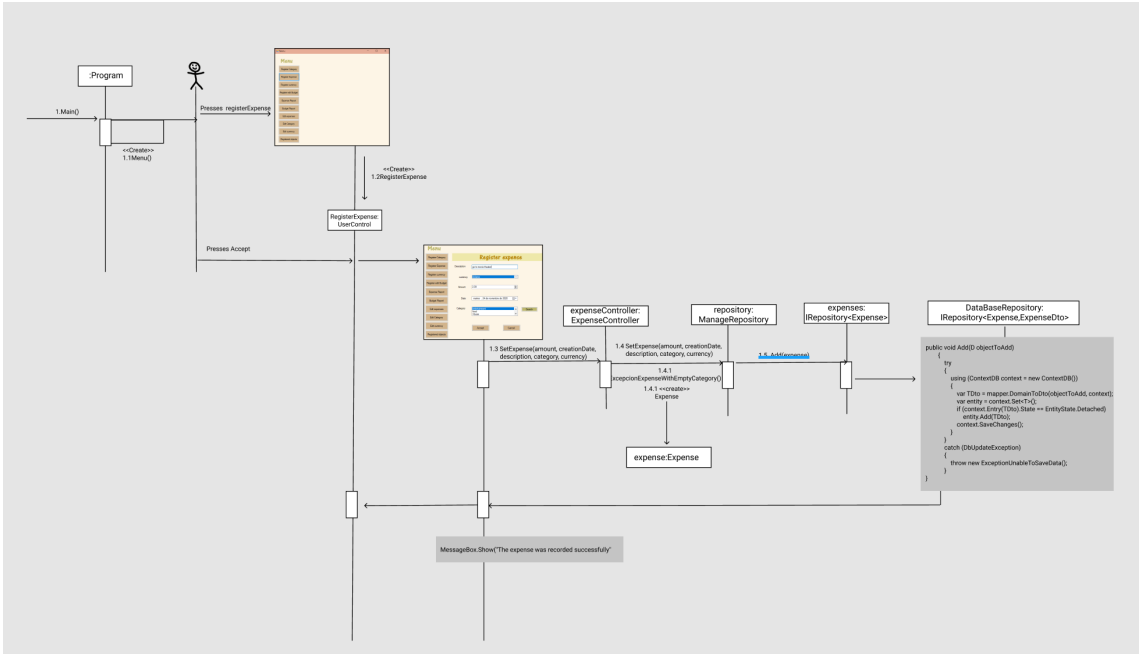
[Anexo 3] Diagrama de clases



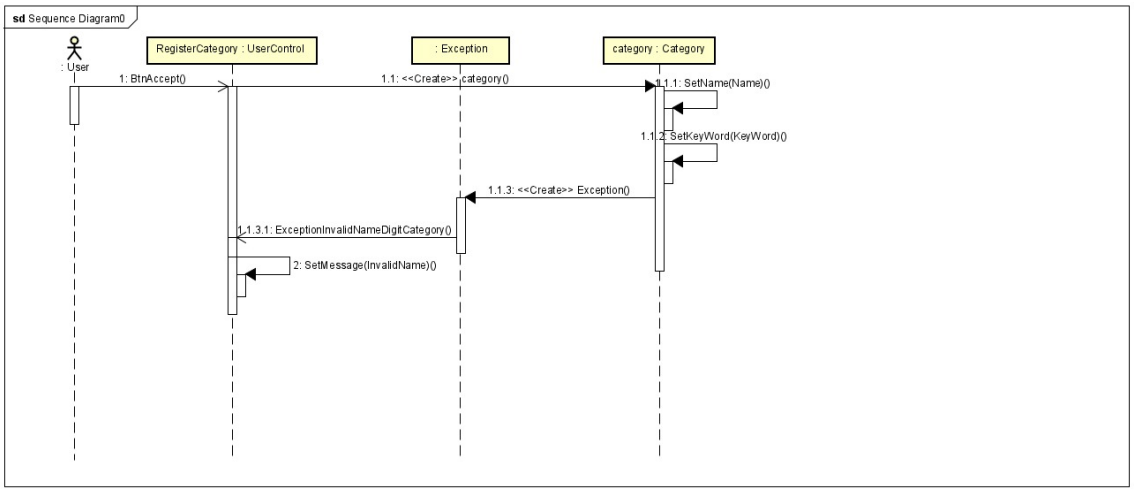
[Anexo 4] Diagrama de clases



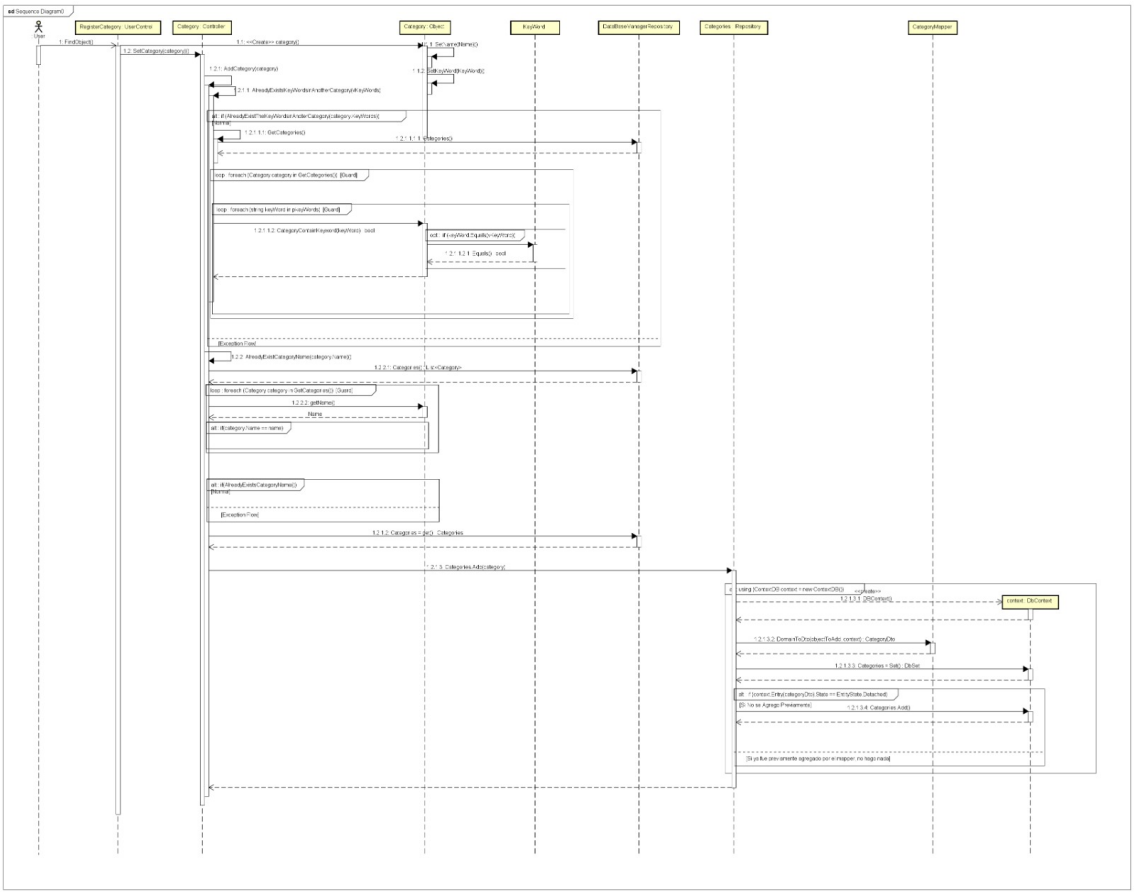
[Anexo 5] Diagramas de interacción



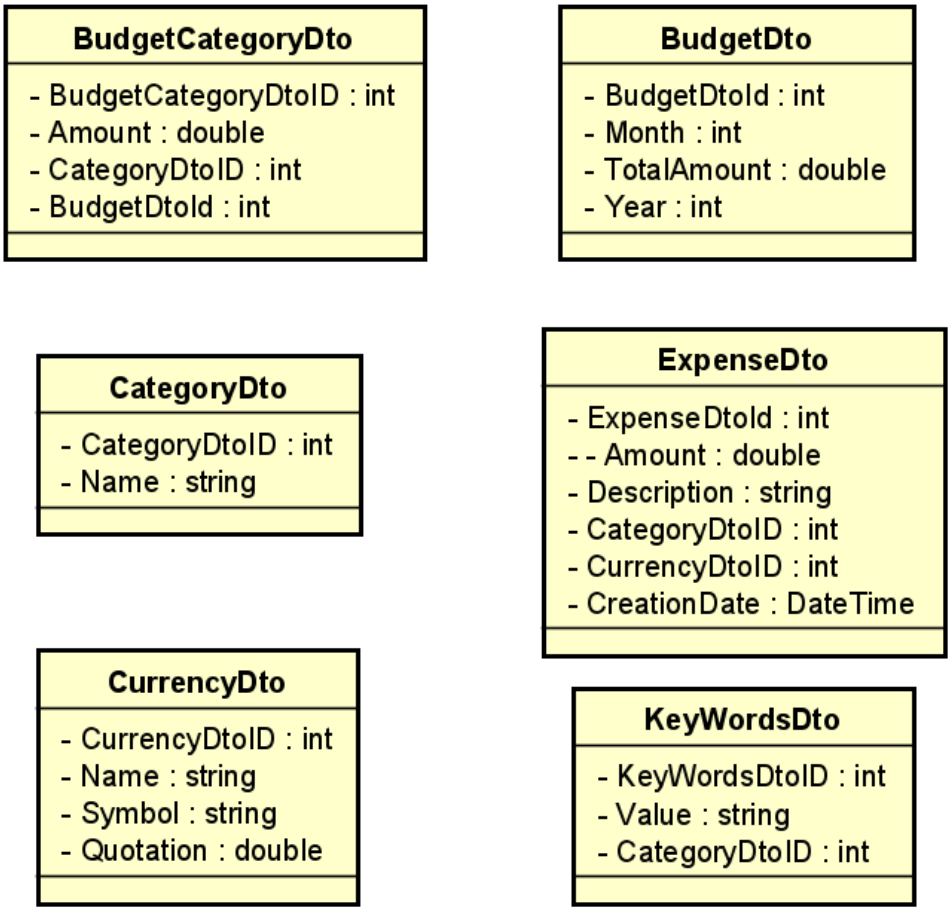
[Anexo 6] Diagramas de interacción



[Anexo 7] Diagramas de interacción



[Anexo 8] Diagrama base de datos



BudgetCategoryDto(BudgetCategoryDtoId,Amount CategoryDtoID, BudgetDtoId)

CategoryDtoID FK CategoryDTO
BudgetDtoId FK BudgetDto

BudgetDto(BudgetDtoId, Month, TotalAmount, Year)

CategoryDto(CategoryDtoID,Name)

ExpenseDto(ExpenseDtoID,Amount, CreationDate, CategoryDtoID, CurrencyDtoID)

CurrencyDtoID FK CurrencyDto
CategroyDtoID FK CategoryDto

CurrencyDto(CurrencyDtoID,Name,Symbol,Quotation)

KeyWordsDto(KeyWordDtoID,Value,CategroyDtoID)

CategroyDtoID FK CategoryDto

[Anexo 5] Cobertura de las pruebas unitarias

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Choche_LAPTOP-N82URT53 2...	69	3,05 %	2197	96,95 %
businesslogic.dll	0	0,00 %	722	100,00 %
BusinessLogic	0	0,00 %	722	100,00 %
Budget	0	0,00 %	130	100,00 %
Budget.<>c	0	0,00 %	6	100,00 %
BudgetCategory	0	0,00 %	24	100,00 %
Category	0	0,00 %	46	100,00 %
Category.<>c	0	0,00 %	2	100,00 %
Expense	0	0,00 %	45	100,00 %
LogicController	0	0,00 %	360	100,00 %
Repository	0	0,00 %	109	100,00 %