Universidad ORT Uruguay
Facultad de Ingeniería

OBLIGATORIO 2
DISEÑO DE APLICACIONES 1

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Ignacio Quevedo (271557)

Gonzalo Camejo (256665)

Tutores: Diego Balbi - Gastón Mousqués

Grupo: M5A

Repositorio: https://github.com/IngSoft-DA1-2023-2/271557_256665

2023

Declaración de autoría

Nosotros, Ignacio Quevedo y Gonzalo Camejo, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano, por lo cual aseguramos que:

• La obra fue producida en su totalidad mientras realizamos el Obligatorio 2 de 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 como por ejemplo inteligencia artificial (Chat gpt);
• 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.

Gonzalo Camejo
16 de Noviembre del 2023

Ignacio Quevedo
16 de Noviembre del 2023

Abstract

El obligatorio a realizar está compuesto por 6 grandes secciones.
Estas secciones serian:

- User Interface (Interfaz de Usuario)
- Controller (Controlador)
- Business Logic (Lógica de Negocio)
- User Repository (Repositorio Del Usuario)
- Database (Base De Datos)
- Tests de cada sección

Palabras Clave

Blazor, C#, Unit Tests, Code Average Tests, Clase abstracta, Polimorfismo, Gitflow, Bootstrap, Controlador,Sql Context

Índice

Descripción general del trabajo

En este respectivo trabajo se optó por realizar un startup cuyo nicho de mercado son todas aquellas personas que busquen un gran servicio mediante el cual puedan manejar sus finanzas, esta aplicación es denominada como FinTrac.

FinTrac es un sistema totalmente responsive en la cual los diferentes usuarios pueden ir registrando todos sus movimientos económicos, ya sean tanto de ingresos como de egresos.

El sistema es capaz de registrar todo componente que contiene un movimiento de capital. Estos correspondientes son objeto de suma importancia y mostrárselos a los usuario genera una sensación de satisfacción y comodidad a la hora de su uso.

Estos correspondientes son:

Usuario, Categoría, Reporte, Cuenta, Tipo de cambio, Transacción y Objetivos de gastos.

El usuario solamente es capaz de registrar y visualizar sus propios movimientos, lo que implica que haya una total unidireccionalidad con el sistema, nos parece correcto y sensato lo mencionado dado que toda actividad económica de una persona es algo que carece de sentido de mostrar al público y esto solamente conlleva un riesgo a niveles de seguridad.

Git Flow

Para el desarrollo de la solución, se utilizaron los lineamientos de Git Flow, desde el comienzo se fue trabajando en ramas específicas cuyo objetivo era una implementación única del sistema. Nombradas así las ramas que cumplían esta determinación, se fueron implementando y con el paso del tiempo, al alcanzar su finalización, todos los integrantes del grupo aplicaban una serie de pasos con el fin de subir sus cambios a producción.

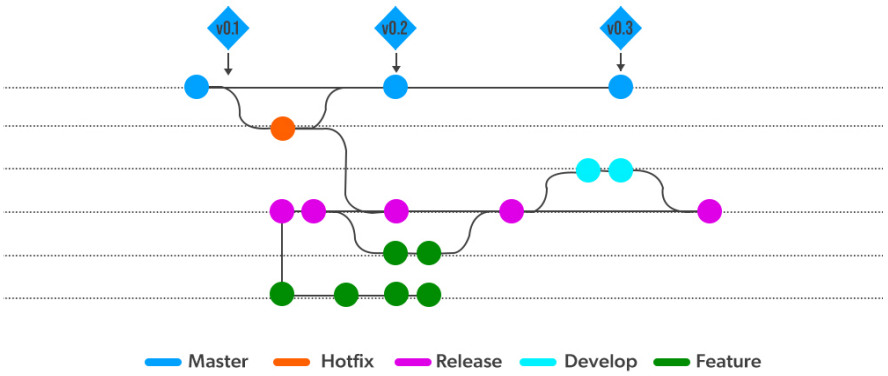
Con el fin de subir estas ramas a producción, las ramas salían del área local del programador y eran enviadas al repositorio remoto (residente en GitHub), una vez allí se realizaba el merge con la rama develop.

Hacemos noción de que para poder realizar el merge con la rama develop primero

tenían que realizar una **pull request** a un reviewer, el cual este mismo sería el otro integrante del equipo. Esto fue conllevado dado que al momento de subir ciertas características a la rama producción este merge debe ser verificado por otro integrante con el fin de evitar los errores de git flow, como podría ser un merge a main, por lo cual únicamente una feature sería mergeada con develop una vez que todos los integrantes del grupo estuvieran conformes con la decisión y reportará esto en un mensaje previo a dar “commit merge” lo cual significa que ellos hayan entendido los cambios de la misma.

Acompañando el trabajo con los lineamientos de Git Flow, se utilizó TDD y clean code para el desarrollo del código.

El uso de Git Flow resultó ser clave para no tener grandes conflictos al reunir distintas piezas de código, dado que facilitaba los merge hacia develop evitando inconsistencias entre los integrantes y generaba que toda la área de trabajo fuese mucho más entendible y limpia. Pudiendo atacar distintos frentes de batalla, de tal forma que los integrantes del equipo eran capaces de desarrollar distintas funcionalidades de forma independiente y asincrónica. Aumentando la velocidad del trabajo en forma exponencial.



Manejo de TDD

A su vez, desarrollar toda la parte lógica del sistema mediante el uso de TDD facilitó a gran escala la comprensión de las funcionalidades que eran de total aclamación a implementar, logrando por consiguiente poner objetivos claros para cada una de las clases y requerimientos a implementar.

Queremos dejar explícito que la técnica de TDD fue algo muy útil para nosotros, ya que mediante la misma pudimos realmente tener una sensación de seguridad al estar trabajando, ya que estamos validando todos aquellos detalles que antes hubiéramos saltado, ya que previo a esta materia estábamos engranadas a desarrollar lo importante primero y luego a ver todo aquel detalle minucioso.
Pero, al estar aplicando TDD, e ir de lo más pequeño a lo más grande, todos aquellos problemas que hubiéramos tenido al final del proyecto, que podemos decir con total virtud que hubieran generado un gran cambio en la lógica del negocio, no fueron de gran complicación, dado que al estar totalmente enfocados en ellos y partir de una lógica donde los mismos se encuentran desde los principios de lógica, facilitó en gran magnitud la implementación del sistema.

Para las ramas en las cuales se implementan nuevas funcionalidades se les denomina mediante el siguiente patrón de diseño: features/[nombre-funcionalidad]. En caso de que se detectará un error considerable luego del merge con develop, el equipo opta por crear ramas denominadas fix/[nombre-fix] y luego realizar el merge correspondiente a develop.

Para aquellos fix que no fuesen de una magnitud considerable, optamos por hacer los commits necesarios para arreglar el problema directamente en una feature relacionada, que luego sería mergeada a develop.

Esta decisión fue tomada por motivos de organización y entendimiento del repositorio remoto. Al principio cada integrante realiza una implementación correspondiente de la lógica del negocio en su completitud ya que nos parecía más

cómodo si cada uno trabajase en implementaciones que no tuvieran mucho en común, con el fin de evitar errores de mergeo a futuro.

En casi la totalidad del proyecto se mantuvo el uso recomendado por los docentes de RED, GREEN y REFACTOR, exceptuando algunos casos donde creímos más correcto usar Green/Refactor. Esto solo fue realizado en muy pocos casos que consideramos innecesario el uso de Red.

Reporte de problemas y bugs

Problema #1

Descripción: Cuando se intenta actualizar un campo de fecha con el teclado, el sistema lanza una excepción no controlada.

Motivo de no implementación: Falta de tiempo para buscar el bug específico.

Frecuencia: Siempre

Problema #2

Descripción: Las gráficas rara vez no cargan correctamente y se cae la página.

Motivo de no implementación: Falta de conocimientos sobre async functions.

Frecuencia: Muy rara vez, se ha observado pero desconocemos el origen del problema y no logramos trackear el error.

Problema #3

Descripción: Al dar click en refresh del navegador la página se buggea la página.

Motivo de no implementación: Falta de tiempo para buscar el bug específico.

Frecuencia: Siempre

Problema #4

Descripción: Si se crea un objetivo de gasto, cuya currency no es pesos uruguayos, al validar si el objetivo de gasto se cumple o no, falla, esto se debe a que nunca estamos haciendo la conversión del valor de la moneda a pesos uruguayos

Motivo de no implementación: Falta de tiempo

Frecuencia: Siempre

Diagrama de paquetes

Ver diagrama de paquetes en: **Anexo 1**

Con el fin de conseguir nuestro diagrama de paquetes, tuvimos que identificar los grupos en los cuales podíamos diferenciar las distintas clases. En un inicio, debimos llegar a un acuerdo de cuántos paquetes eran necesarios para el proyecto, ya que esto condiciona a futuro cómo se distribuye el trabajo en la aplicación y si no se toman buenas de decisiones puede resultar en una pérdida tremenda en cuanto a la mantenibilidad del código a futuro.

Concluimos en una aplicación separada en distintas capas ya que consideramos pertinente separar la Business Logic de lo que está relacionado con la User Interface ya que es la base de lo que debe estar separado dentro de la aplicación con el fin de asegurar dicha mantenibilidad futura, sino nuestro código sería desorganizado y mezclar lógica de negocio con interfaz de usuario resultaría en algo poco entendible para quienes vienen desde afuera a mirar nuestra App, siguiendo uno de los principios de clean code es que tomamos esta decisión, permitiendo así una mayor reusabilidad del código.

Está separación es muy útil, pero no es suficiente para reducir el acoplamiento, para esto añadimos las capas Data Managers y Controllers con el fin de que la UI nunca se comunique de forma directa con la Business Logic de forma directa, eliminando así el alto acoplamiento y por ende aumentando la cohesión, esto se debe a que cada clase realiza una única responsabilidad siguiendo uno de los principios SOLID el cual es el SRP Single Responsibility Principle.

Otro de los principios SOLID que nos permite utilizar esta separación por capas es la de la (DIP) Inversión de dependencias ya que este habla sobre la inyección de dependencias en la aplicación con el fin de depender de capas que sean abstracciones de otra y de esta manera no depender directamente de una capa. Encontramos muy útil el hecho de poder separar con interfaces nuestro controller ya

que con ellas podemos elegir que ve la persona encarga de trabajar con la UI sin necesidad de que personas que no estén relacionadas con cierta página de la interfaz puedan ver métodos genéricos que no le corresponden.

Un ejemplo claro de esta reusabilidad se ve reflejado en una iteración de negocio, si deseamos cambiar la interfaz por algún motivo, no deberíamos estar buscando entre lógica de negocio e interfaz para saber dónde debemos cambiar nuestro código, sino que debemos ir directamente y debe resultar sencillo para cualquier persona de una empresa cambiar el código relacionado a la interfaz o la Business Logic, debemos evitar dolores de cabeza y dejar el campo limpio para otros en orden de mejorar.

Diagrama de clases

Ver diagrama de clases en: **Anexo 2**

Capas generadas para la realización de la aplicación:

- BusinessLogic
- DataManagers
- Controllers
- Fintrac (UI)

Capa BusinessLogic

Esta capa existe desde la primera entrega del proyecto ya que fue la primera idea de separación con la UI, al implementar nuestra solución usando Code First, primero se generó esta lógica negocio que nos permite generar todas las clases necesarias que interactúan de fondo con sus respectivos métodos para poder ejecutar diversas operaciones que afectan a la aplicación.

Capa DataManagers

Esta capa es la encargada de contener lo referente a la base datos como ser todas las operaciones CRUD necesarias para interactuar con la base de datos y de esta manera poder trabajar con la misma. El contexto que refiere a la base de datos también se encuentra dentro de esta capa ya que el mismo es totalmente necesario para la interacción con la base de datos, tanto la real como la base de datos que se usará para las pruebas de los tests.

En nuestro caso utilizamos una clase llamada userRepository para el manejo de todas las operaciones relacionadas con la base de datos.

Las migraciones también se encuentran en una carpeta aislada dentro de esta capa ya que las mismas contienen el modelo que la base de datos debe seguir, conteniendo los reglamentos generados automáticamente por entity framework.

Capa Controllers

Esta capa cumple con la función de desacoplar la interfaz de la business logic y poder de esta manera brindarle a la businessLogic una liberación de responsabilidades, filtrando así lo que queremos que vea el encargado de manejar la UI por medio de un controlador de fachada el cual es separado en interfaces para que las diferentes páginas de la UI no conozcan todos los métodos que posee el controlador genérico

Persistencia y tablas

Entorno de desarrollo:

Nuestro BackUp se encuentra en: \FinTrac\DataManagers\Backups

Aplicaciones usadas para el desarrollo de la persistencia: Docker y Azure Data Studio

Versión de Docker: 4.25.0

Versión de Azure Data Studio: 1.47

Versión de Entity Framework: 7.0.13

Connection string utilizado en appsettings.json:

```
  "ConnectionStrings": {
    "DefaultConnection": "Server=127.0.0.1;Database=Blazor2;User
Id=sa;Password=Pass.2022;TrustServerCertificate=true;" }
```

Para el desarrollo de nuestra aplicación, todos los integrantes del proyecto utilizaron las aplicaciones mencionadas anteriormente con el fin de dirigir el enfoque de desarrollo hacia la persistencia de las entidades mediante la utilización de Entity Framework.

Aclaramos que ambos integrantes trabajamos con sistemas operativos distintos (MacOS Sonoma y Windows 11) por lo cual nuestra persistencia se encuentra 100% funcional en ambos sistemas operativos.

Para realizar el backup, nuestros pasos son diferentes a los de las soluciones en aulas ya que nuestro backup se encontrará en la máquina virtual accediendo a la carpeta.

Modelo de tablas:

Ver modelo de tablas: Anexo 3

El modelo de tablas fue generado automáticamente por Entity Framework al generar todas las relaciones existentes dentro de nuestras Entidades por medio de las relaciones que fueron creadas por nosotros dentro de la businessLogic. Con estas relaciones, Entity Framework pudo localizar las tablas necesarias a crear pudiendo de esta manera generar una migración con lo necesario para la creación de una base de datos.

Explicación de mecanismos generales y principales decisiones de diseño

En cuanto a los mecanismos generales del diseño de nuestra aplicación, optamos por tener una buena separación de capas dentro de la aplicación priorizando el conocimiento entre las diferentes clases es por esto que realizamos la separación en capas mencionada anteriormente. Decidimos dejar todas las funciones de la businessLogic en función del usuario ya que nos pareció que él es el encargado de realizar sus operaciones ya que le conciernen a él mismo.

Una principal decisión de diseño tomada fue la implementación de diferente Scopes en la pagina program para poder así visualizar los controladores en cada página generando así un ciclo de vida para los mismos y que solo fueran iniciados cuando sea necesario en cada página y no tener la necesidad de que los mismo quedarán consumiendo recursos de fondo sin necesidad alguna, por este motivo optamos por el uso de scopes y no de singletons en la interfaz de usuario.

Optamos por la implementación de DTOs para que nuestra UI no hable directamente con los objetos de nuestra businessLogic sino que solamente utilice un objeto de transferencia de datos que contiene en su totalidad datos limpios y planos, es decir solamente datos que deberán ser desplegados por la interfaz dependiendo de la page. Para estos DTOs se generó la necesidad de implementar los Mappers con el fin centralizar todo lo referido al pasaje de un objeto de businessLogic hacia otro que se encuentra en la UI.

Otra estrategia de diseño que aplicamos fue el controlador de fachada el cual funciona como una interfaz de ocultamiento hacia la UI y filtra los métodos que se verán desde las pages usando los IControladores con el fin de que no se conozcan todos los métodos del controlador general.

Consideramos necesario el borrado de una clase que previamente se encontraba en la UI y que se encargaba de dejar vivo un usuario durante toda la aplicación ya que consideramos que esto consume muchos recursos y genera tener una clase innecesaria en la UI. Esto se encontrara ahora reemplazado por el uso de cascading values que pasan un DTO de un usuario entre las diferentes pages con el fin de no perder dicho usuario durante el transcurso de la aplicación.

Decidimos implementar un helper para poder así englobar funciones que eran generales para todas las clases.

Ver diagramas de interacción: Anexo 4

Pruebas Unitarias

Ver detalle de pruebas unitarias: Anexo 5

En el anexo mencionado, se muestra un detalle de las pruebas realizadas y los tests pasados en estado Green al final de la entrega con todos los tests corriendo correctamente y sin problemas.

Las pruebas unitarias fueron realizadas durante el transcurso de la programación de la Business Logic, el controlador, los mappers y los DTOs. Consiguiendo un coverage de las pruebas unitarias de un 97%

Al inicio nuestro test coverage fue variando, pero nos tomamos el tiempo de debuggear y poder así ver cuales era los problemas de coverage que teníamos y los fuimos arreglando. Logrando así la coverage observada, que a pesar de no ser perfecta, se encuentra a 3 puntos por debajo del 100%, debido en su mayoría a que las excepciones lanzadas dejan una cierta línea del código sin cubrir y esto genera que el porcentaje de cobertura de cada test no sea de 100%.
Sin embargo, pudimos arreglar muchos de los tests que no funcionaban en la entrega anterior pudiendo subir así 3 puntos la cobertura de nuestros tests ya que pudimos encontrar errores relacionados a métodos que no estamos testeando y de esta manera pudimos subir nuestra cobertura.

Este motivo es el principal que nos implicó no poder alcanzar el 100% en las pruebas obtenidas. Dejamos en noción de que este tema fue discutido previamente con los docentes, quienes comentaron que el error es algo normal que sucede y se debe a una falla del programa. Y es algo que todos los equipos van a tener.

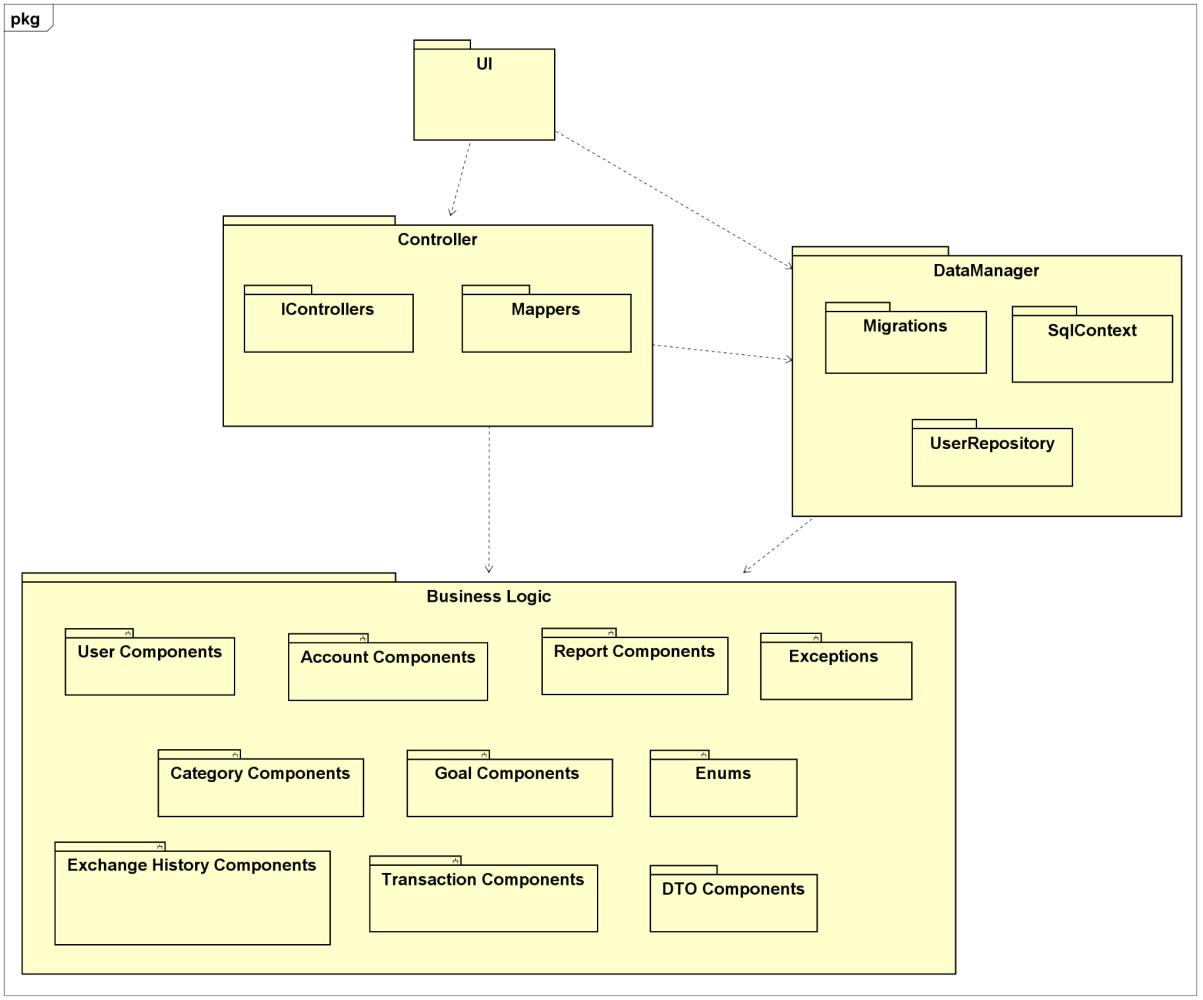
A su vez aclaramos que hay ciertos métodos implementados por nosotros que al estudiar la cobertura se puede observar que tienen menos de un 90% de coverage, esto se debe a que para ciertos métodos faltó evaluar los wrong cases y debido a esto determinados tests no evalúan determinadas líneas del código, lo que baja la cobertura del test.

Cada namespace donde se adjudican los conjuntos de tests supera por amplitud el 90% de coverage, por ende se puede denotar que los métodos evaluados cuya cobertura es menor de 90% se encuentran en pequeña proporción.

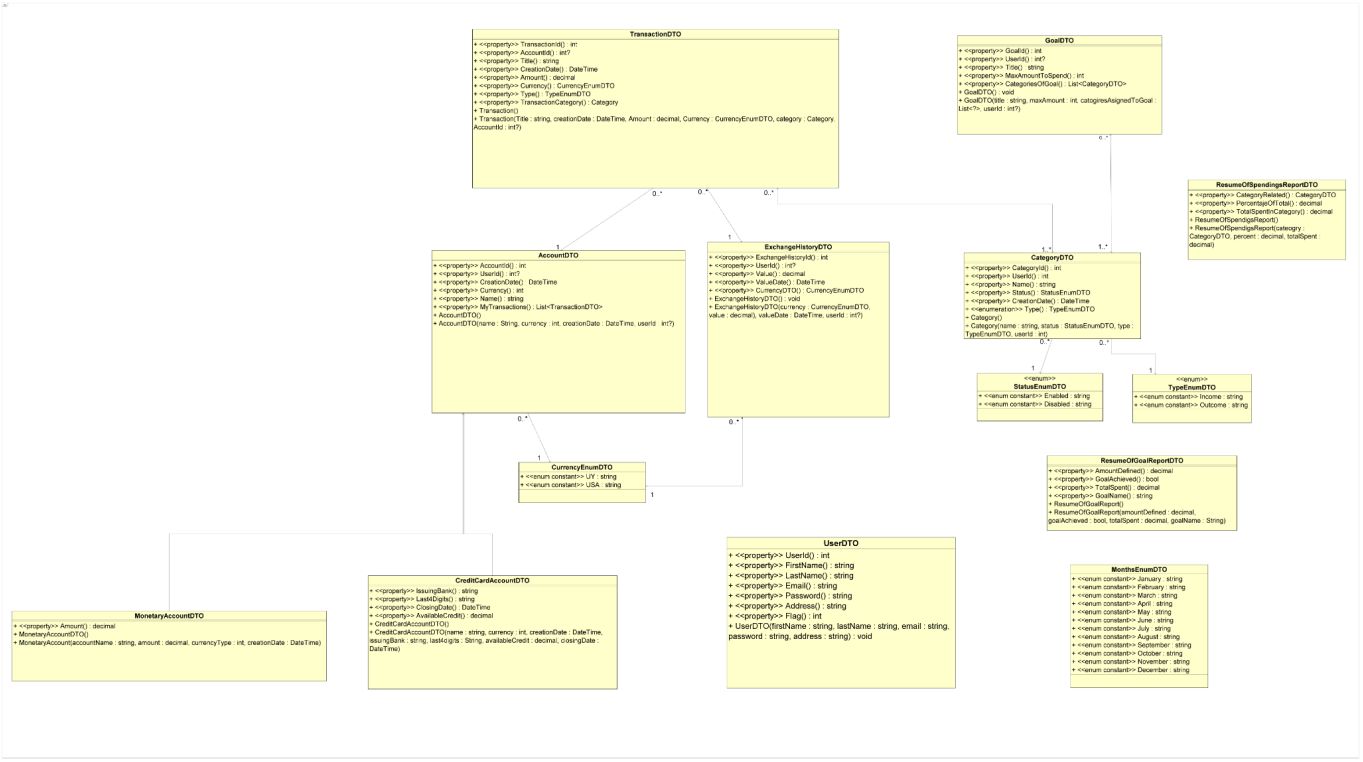
El cambio solicitado sobre agregar una moneda en Euros es un cambio que fue fácil de implementar ya que usamos un enum.

Anexos

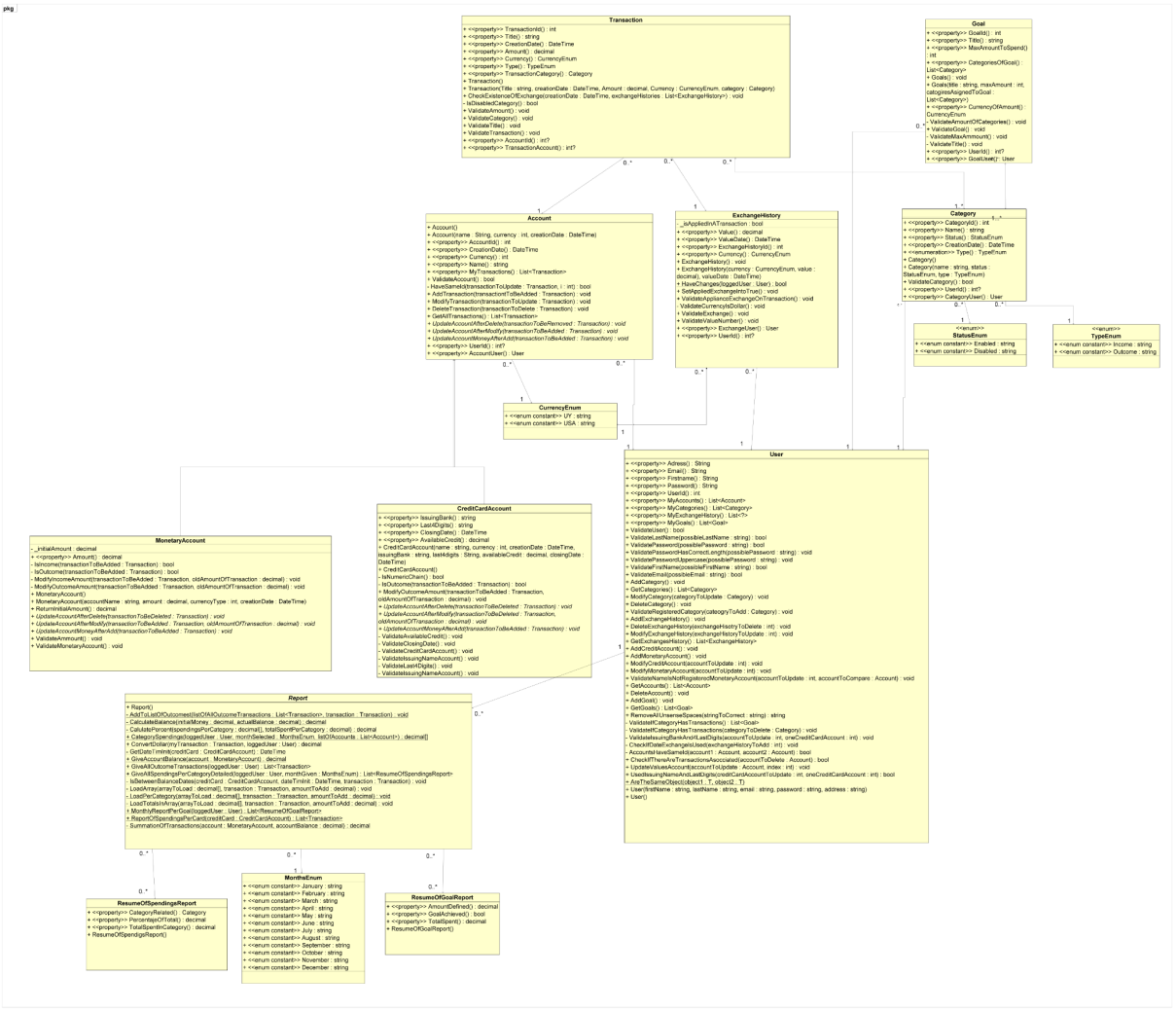
Anexo 1: Diagrama de paquetes

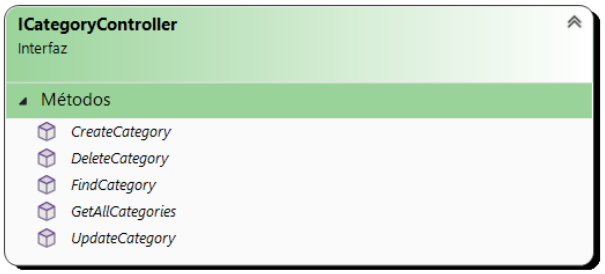


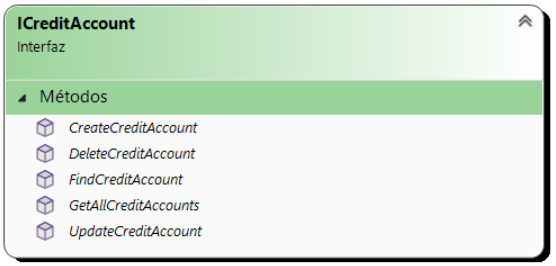
Anexo 2: Diagrama de clases

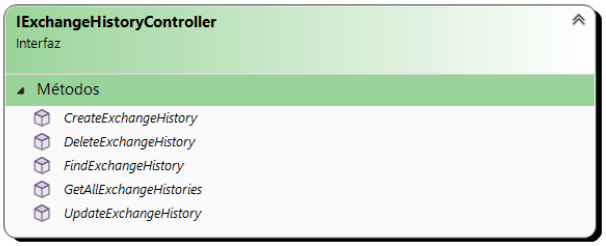


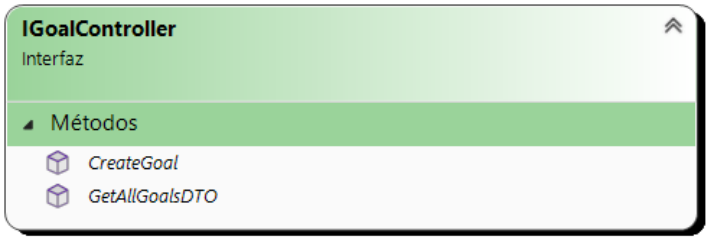

pkg
MapperUser
+ ToUserDTO(userGiven : User) : UserDTO
+ ToUser(userGiven : UserDTO) : User
- FormattUserProperties(userGiven : User) : void
MapperCategory
+ ToCategoryDTO(categoryGiven : Category) : CategoryDTO
+ ToCategory(categoryGiven : CategoryDTO) : Category
+ ToListCategory(categoryGiven : List<CategoryDTO>) : List<Category>
+ ToListCategoryDTO(categoryGiven : List<Category>) : List<CategoryDTO>
MapperAccount
+ ToListAccount(accounts : List<AccountDTO>) : List<Account>
+ ToListAccountDTO(accounts : List<Account>) : List<AccountDTO>
MapperExchangeHistory
+ ToExchangeHistory(myExchange : ExchangeHistoryDTO) : ExchangeHistory
+ ToExchangeHistoryDTO(myExchange : ExchangeHistory) : ExchangeHistoryDTO
+ ToListOfExchangeHistoryDTO(myExchange : List<ExchangeHistory>) : List<ExchangeHistoryDTO>
+ ToListOfExchangeHistory(myExchange : List<ExchangeHistoryDTO>) : List<ExchangeHistory>
MapperMovementInXDays
+ ToMovement(movementsInXDaysDTO : MovementsInXDaysDTO) : MovementsInXDays
+ ToMovementDTO(movementsInXDays : MovementsInXDays) : MovementsInXDaysDTO
MapperTransaction
+ ToTransactionDTO(trans : Transaction) : TrannsactionDTO
+ ToTransaction(trans : TransactionDTO) : Transaction
+ ToListOfTransactionsDTO(trans : List<Transaction>) : List<TransactionDTO>
+ ToListOfTransactions(trans : List<TransactionDTO>) : List<Transaction>
MapperMonetaryAccount
+ ToMonetaryAccountDTO(monet : MonetaryAccount) : MonetaryAccountDTO
+ ToMonetaryAccount(monet : MonetaryAccountDTO) : MonetaryAccount
+ ToListOfMonetaryAccountDTO(monet : List<MonetaryAccount>) : List<MonetaryAccount>
MapperCreditCardAccount
+ ToCreditAccountDTO(credit : CreditCardAccount) : CreditCardAccountDTO
+ ToCreditAccount(credit : CreditCardAccountDTO) : CreditCardAccount
+ ToListOfCreditAccountDTO(credit : List<CreditCardAccount>) : List<CreditCardAccountDTO>
MapperGoal
+ ToGoalDTO(goal : Goal, List<CategoryDTO> list : int) : GoalDTO
+ ToGoal(goal : Goal, List<Category> list : int) : Goal
+ ToListGoal(list : List<GoalDTO>) : List<Goal>
+ ToListGoaDTOl(list : List<Goal>) : List<GoalDTO>
MapperResumeOfGoal
+ ToResumeOfGoalDTO(resume : ResumeOfGoal) : ResumeOfGoalReportDTO
+ ToResumeOfGoal(resume : ResumeOfGoalDTO) : ResumeOfGoalReport
+ ToListResumeOfGoal(resume : List<ResumeOfGoalDTO>) : List<ResumeOfGoalReport>
+ ToListResumeOfGoalDTO(resume : List<ResumeOfGoal>) : List<ResumeOfGoalReportDTO>
MapperResumeCategory
+ ToResumeOfCategoryDTO(resume : ResumeOfCategory) : ResumeOfCategoryReportDTO
+ ToResumeOfCategory(resume : ResumeOfCategoryDTO) : ResumeOfCategoryReport
+ ToListResumeOfCategory(resume : List<ResumeOfCategoryDTO>) : List<ResumeOfCategoryReport>
+ ToListResumeOfCategoryDTO(resume : List<ResumeOfCategory>) : List<ResumeOfCategoryReportDTO>


pkg
Transaction
+ <<property>> TransactionId() : int
+ <<property>> Title() : string
+ <<property>> CreationDate() : DateTime
+ <<property>> Amount() : decimal
+ <<property>> Currency() : CurrencyEnum
+ <<property>> Type() : TypeEnum
+ <<property>> TransactionCategory() : Category
+ Transaction()
+ Transaction(Title : string, creationDate : DateTime, Amount : decimal, Currency : CurrencyEnum, category : Category)
+ CheckExistenceOfExchange(creationDate : DateTime, exchangeHistories : List<ExchangeHistory>) : void
- IsDisabledCategory() : bool
+ ValidateAmount() : void
+ ValidateCategory() : void
+ ValidateTitle() : void
+ ValidateTransaction() : void
+ <<property>> AccountId() : int?
+ <<property>> TransactionAccount() : int?
Goal
+ <<property>> GoalId() : int
+ <<property>> Title() : string
+ <<property>> MaxAmountToSpend() : int
+ <<property>> CategoriesOfGoal() : List<Category>
+ Goals() : void
+ Goals(title : string, maxAmount : int, categoriesAsignedToGoal : List<Category>)
+ <<property>> CurrencyOfAmount() : CurrencyEnum
- ValidateAmountOfCategories() : void
+ ValidateGoal() : void
- ValidateMaxAmmount() : void
- ValidateTitle() : void
+ <<property>> UserId() : int?
+ <<property>> GoalUser() : User
Account
+ Account()
+ Account(name : String, currency : int, creationDate : DateTime)
+ <<property>> AccountId() : int
+ <<property>> CreationDate() : DateTime
+ <<property>> Currency() : int
+ <<property>> Name() : string
+ <<property>> MyTransactions() : List<Transaction>
+ ValidateAccount() : bool
- HaveSameId(transactionToUpdate : Transaction, i : int) : bool
+ AddTransaction(transactionToBeAdded : Transaction) : void
+ ModifyTransaction(transactionToUpdate : Transaction) : void
+ DeleteTransaction(transactionToDelete : Transaction) : void
+ GetAllTransactions() : List<Transaction>
+ UpdateAccountAfterDelete(transactionToBeRemoved : Transaction) : void
+ UpdateAccountAfterModify(transactionToBeAdded : Transaction) : void
+ UpdateAccountMoneyAfterAdd(transactionToBeAdded : Transaction) : void
+ <<property>> UserId() : int?
+ <<property>> AccountUser() : User
ExchangeHistory
- _isAppliedInATransaction : bool
+ <<property>> Value() : decimal
+ <<property>> ValueDate() : DateTime
+ <<property>> ExchangeHistoryId() : int
+ <<property>> Currency() : CurrencyEnum
+ ExchangeHistory() : void
+ ExchangeHistory(currency : CurrencyEnum, value : decimal, valueDate : DateTime)
+ HaveChanges(loggedUser : User) : bool
+ SetAppliedExchangeIntoTrue() : void
+ ValidateAppliancesExchangeOnTransaction() : void
- ValidateCurrencyIsDollar() : void
+ ValidateExchange() : void
+ ValidateValueNumber() : void
+ <<property>> ExchangeUser() : User
+ <<property>> UserId() : int?
Category
+ <<property>> CategoryId() : int
+ <<property>> Name() : string
+ <<property>> Status() : StatusEnum
+ <<property>> CreationDate() : DateTime
+ <<enumeration>> Type() : TypeEnum
+ Category()
+ Category(name : string, status : StatusEnum, type : TypeEnum)
+ ValidateCategory() : bool
+ <<property>> UserId() : int?
+ <<property>> CategoryUser() : User
<<enum>>
StatusEnum
+ <<enum constant>> Enabled : string
+ <<enum constant>> Disabled : string
<<enum>>
TypeEnum
+ <<enum constant>> Income : string
+ <<enum constant>> Outcome : string
CurrencyEnum
+ <<enum constant>> UY : string
+ <<enum constant>> USA : string
User
+ <<property>> Adress() : String
+ <<property>> Email() : String
+ <<property>> Firstname() : String
+ <<property>> Password() : String
+ <<property>> UserId() : int
+ <<property>> MyAccounts() : List<Account>
+ <<property>> MyCategories() : List<Category>
+ <<property>> MyExchangeHistory() : List<?>
+ <<property>> MyGoals() : List<Goal>
+ ValidateUser() : bool
- ValidateLastName(possibleLastName : string) : bool
- ValidatePassword(possiblePassword : string) : bool
- ValidatePasswordHasCorrectLength(possiblePassword : string) : void
- ValidatePasswordUppercase(possiblePassword : string) : void
- ValidateFirstName(possibleFirstName : string) : bool
- ValidateEmail(possibleEmail : string) : bool
+ AddCategory() : void
+ GetCategories() : List<Category>
+ ModifyCategory(categoryToUpdate : Category) : void
+ DeleteCategory() : void
+ ValidateRegisteredCategory(categoryToAdd : Category) : void
+ AddExchangeHistory() : void
+ DeleteExchangeHistory(exchangeHisotryToDelete : int) : void
+ ModifyExchangeHistory(exchangeHistoryToUpdate : int) : void
+ GetExchangesHistory() : List<ExchangeHistory>
+ AddCreditAccount() : void
+ AddMonetaryAccount() : void
+ ModifyCreditAccount(accountToUpdate : int) : void
+ ModifyMonetaryAccount(accountToUpdate : int) : void
+ ValidateNameIsNotRegisteredMonetaryAccount(accountToUpdate : int, accountToCompare : Account) : void
+ GetAccounts() : List<Account>
+ DeleteAccount() : void
+ AddGoals() : void
+ GetGoals() : List<Goal>
+ RemoveAllUnsenseSpaces(stringToCorrect : string) : string
- ValidateNotCategoryHasTransactions() : List<Goal>
- ValidateNotCategoryHasTransactions(categoryToDelete : Category) : void
- ValidateIssuingBankAndLastDigits(accountToUpdate : int, oneCreditCardAccount : int) : void
- CheckIfDateExchangeIsUsed(exchangeHistoryToAdd : int) : void
- AccountsHaveSameId(account1 : Account, account2 : Account) : bool
+ CheckIfThereAreTransactionsAsociated(accountToDelete : Account) : bool
+ UpdateValuesAccount(accountToUpdate : Account, index : int) : void
+ UsedIssuingNameAndLastDigits(creditCardAccountToUpdate : int, oneCreditCardAccount : int) : bool
+ AreTheSameObject(object1_T, object2_T)
+ User(firstName : string, lastName : string, email : string, password : string, address : string)
+ User()
MonetaryAccount
- _initialAmount : decimal
+ <<property>> Amount() : decimal
- IsIncome(transactionToBeAdded : Transaction) : bool
- IsOutcome(transactionToBeAdded : Transaction) : bool
- ModifyIncomeAmount(transactionToBeAdded : Transaction, oldAmountOfTransaction : decimal) : void
- ModifyOutcomeAmount(transactionToBeAdded : Transaction, oldAmountOfTransaction : decimal) : void
+ MonetaryAccount()
+ MonetaryAccount(accountName : string, amount : decimal, currencyType : int, creationDate : DateTime)
+ ReturnInitialAmount() : decimal
+ UpdateAccountAfterDelete(transactionToBeDeleted : Transaction) : void
+ UpdateAccountAfterModify(transactionToBeAdded : Transaction, oldAmountOfTransaction : decimal) : void
+ UpdateAccountMoneyAfterAdd(transactionToBeAdded : Transaction) : void
+ ValidateAmmount() : void
+ ValidateMonetaryAccount() : void
CreditCardAccount
+ <<property>> IssuingBank() : string
+ <<property>> Last4Digits() : string
+ <<property>> ClosingDate() : DateTime
+ <<property>> AvailableCredit() : decimal
+ CreditCardAccount(name : string, currency : int, creationDate : DateTime, issuingBank : string, last4digits : String, availableCredit : decimal, closingDate : DateTime)
+ CreditCardAccount()
- IsNumericChain() : bool
- IsOutcome(transactionToBeAdded : Transaction) : bool
+ ModifyOutcomeAmount(transactionToBeAdded : Transaction, oldAmountOfTransaction : decimal) : void
+ UpdateAccountAfterDelete(transactionToBeDeleted : Transaction) : void
+ UpdateAccountAfterModify(transactionToBeDeleted : Transaction, oldAmountOfTransaction : decimal) : void
+ UpdateAccountMoneyAfterAdd(transactionToBeAdded : Transaction) : void
- ValidateAvailableCredit() : void
- ValidateClosingDate() : void
- ValidateCreditCardAccount() : void
- ValidateIssuingNameAccount() : void
- ValidateLastFourDigits() : void
- ValidateIssuingNameAccount() : void
Report
+ Report()
- AddToListOfOutcomesIfIsOfAllOutcomeTransactions : List<Transaction>, transaction : Transaction) : void
- CalculateBalance(initialMoney : decimal, actualBalance : decimal) : decimal
- CalculatePercentIspendingsPerCategory : decimal[], totalSpentPerCategory : decimal) : decimal
+ CategorySpendingsSoggedUser : User, monthSelected : MonthsEnum, listOfAccounts : List<Account>) : decimal[]
+ ConvertDollarImyTransaction : Transaction, loggedUser : User) : decimal
- GetDateTimeIncredit(Card : CreditCardAccount) : DateTime
+ GiveAccountBalance(account : MonetaryAccount) : decimal
+ GiveAllOutcomeTransactions(loggedUser : User) : List<Transaction>
+ GiveAllSpendingsPerCategoryVistaBed(loggedUser : User, monthGiven : MonthsEnum) : List<ResumeOfSpendingsReport>
- IsBetweenBalanceDates(credit/Card : CreditCardAccount, dateTimInit : DateTime, transaction : Transaction) : void
- LoadArray(arrayToLoad : decimal[], transaction : Transaction, amountToAdd : decimal) : void
- LoadPerCategory(arrayToLoad : decimal[], transaction : Transaction, amountToAdd : decimal) : void
- LoadTotalsInArray(arrayToLoad : decimal[], transaction : Transaction, amountToAdd : decimal) : void
+ MonthlyReportPerGoal(loggedUser : User) : List<ResumeOfGoalReport>
+ ReportOfSpendingsPerCardcreditCard : CreditCardAccount) : List<Transaction>
- SummationOfTransactions(account : MonetaryAccount, accountBalance : decimal) : decimal
ResumeOfSpendingsReport
+ <<property>> CategoryRealted() : Category
+ <<property>> PercentageOfTotal() : decimal
+ <<property>> TotalSpentInCategory() : decimal
+ ResumeOfSpendigsReport()
MonthsEnum
+ <<enum constant>> January : string
+ <<enum constant>> February : string
+ <<enum constant>> March : string
+ <<enum constant>> April : string
+ <<enum constant>> May : string
+ <<enum constant>> June : string
+ <<enum constant>> July : string
+ <<enum constant>> August : string
+ <<enum constant>> September : string
+ <<enum constant>> October : string
+ <<enum constant>> November : string
+ <<enum constant>> December : string
ResumeOfGoalReport
+ <<property>> AmountDefined() : decimal
+ <<property>> GoalAchieved() : bool
+ <<property>> TotalSpent() : decimal
+ ResumeOfGoalReport()

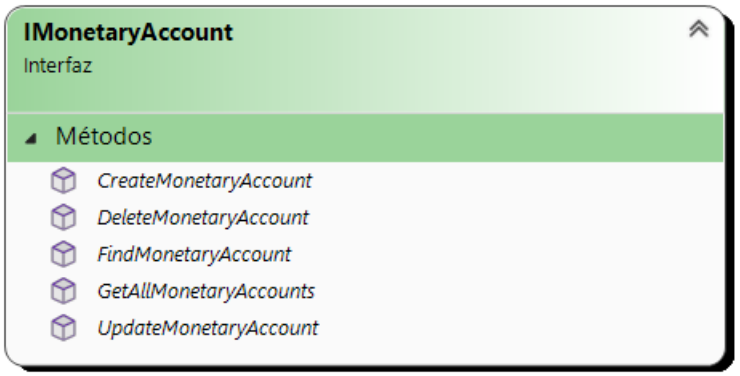

ICategoryController
Interfaz
Métodos
CreateCategory
DeleteCategory
FindCategory
GetAllCategories
UpdateCategory

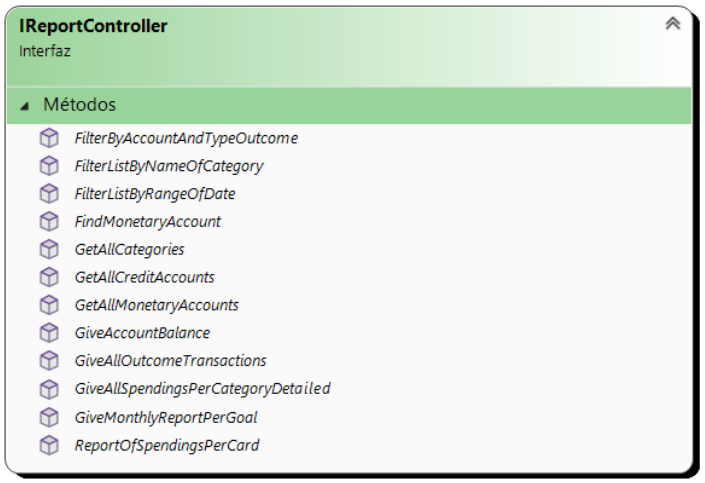

ICreditAccount
Interfaz
Métodos
CreateCreditAccount
DeleteCreditAccount
FindCreditAccount
GetAllCreditAccounts
UpdateCreditAccount

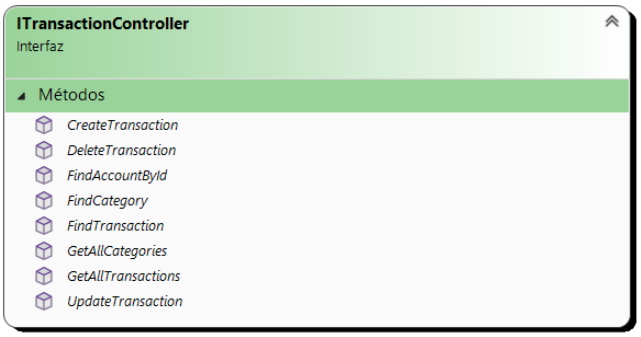

IExchangeHistoryController
Interfaz
Métodos
CreateExchangeHistory
DeleteExchangeHistory
FindExchangeHistory
GetAllExchangeHistories
UpdateExchangeHistory


IGoalController
Interfaz
Métodos
CreateGoal
GetAllGoalsDTO

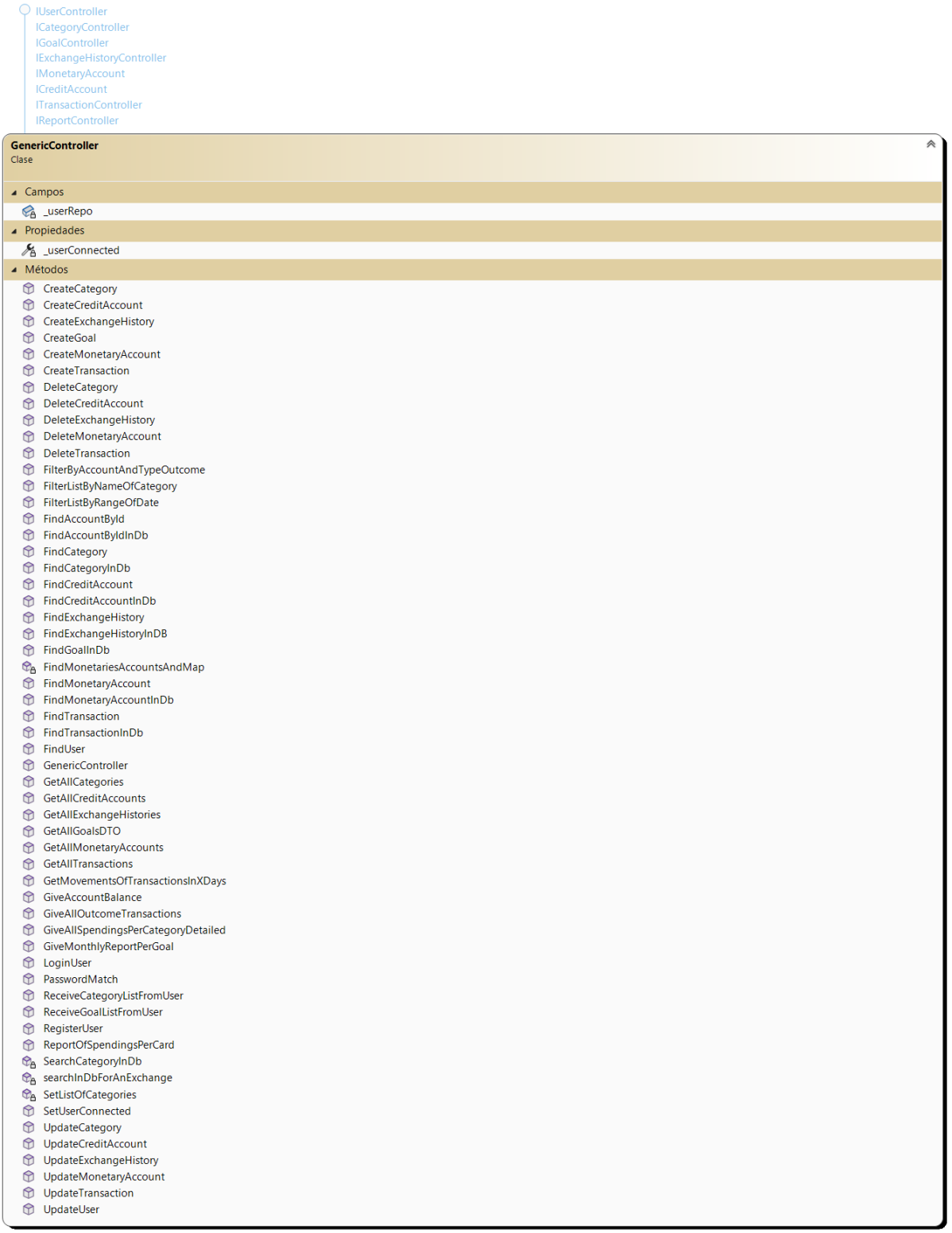
}


IMonetaryAccount
Interfaz
Métodos
CreateMonetaryAccount
DeleteMonetaryAccount
FindMonetaryAccount
GetAllMonetaryAccounts
UpdateMonetaryAccount

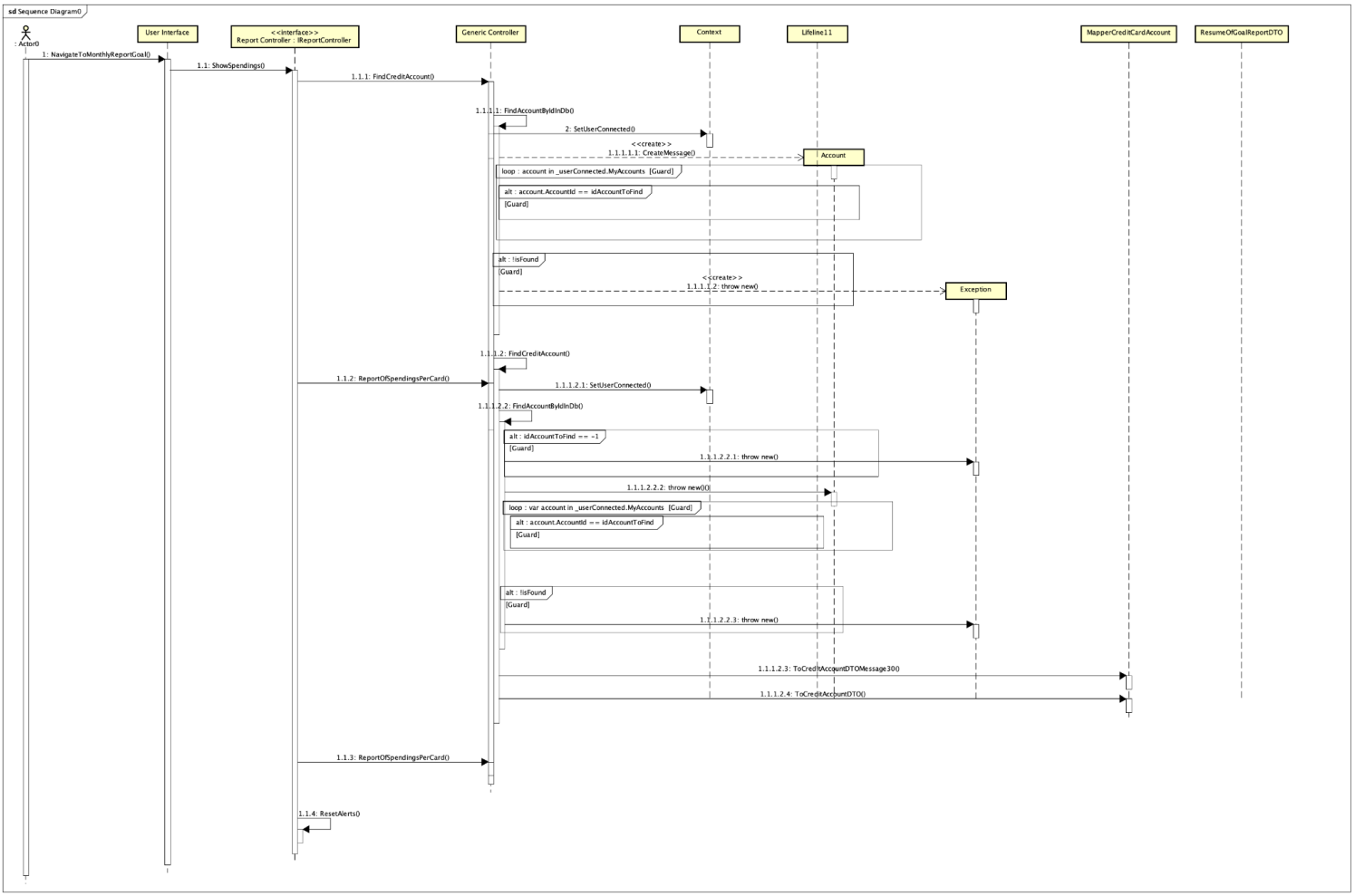

IReportController
Interfaz
Métodos
FilterByAccountAndTypeOutcome
FilterListByNameOfCategory
FilterListByRangeOfDate
FindMonetaryAccount
GetAllCategories
GetAllCreditAccounts
GetAllMonetaryAccounts
GiveAccountBalance
GiveAllOutcomeTransactions
GiveAllSpendingsPerCategoryDetailed
GiveMonthlyReportPerGoal
ReportOfSpendingsPerCard


ITransactionController
Interfaz
Métodos
CreateTransaction
DeleteTransaction
FindAccountById
FindCategory
FindTransaction
GetAllCategories
GetAllTransactions
UpdateTransaction


IUserController
Interfaz
Métodos
FindUser
LoginUser
PasswordMatch
RegisterUser
UpdateUser


IUserController
ICategoryController
IGoalController
IExchangeHistoryController
IMonetaryAccount
ICreditAccount
ITransactionController
IReportController
GenericController
Clase
Campos
_userRepo
Propiedades
_userConnected
Métodos
CreateCategory
CreateCreditAccount
CreateExchangeHistory
CreateGoal
CreateMonetaryAccount
CreateTransaction
DeleteCategory
DeleteCreditAccount
DeleteExchangeHistory
DeleteMonetaryAccount
DeleteTransaction
FilterByAccountAndTypeOutcome
FilterListByNameOfCategory
FilterListByRangeOfDate
FindAccountById
FindAccountByIdInDb
FindCategory
FindCategoryInDb
FindCreditAccount
FindCreditAccountInDb
FindExchangeHistory
FindExchangeHistoryInDB
FindGoalInDb
FindMonetariesAccountsAndMap
FindMonetaryAccount
FindMonetaryAccountInDb
FindTransaction
FindTransactionInDb
FindUser
GenericController
GetAllCategories
GetAllCreditAccounts
GetAllExchangeHistories
GetAllGoalsDTO
GetAllMonetaryAccounts
GetAllTransactions
GetMovementsOfTransactionsInXDays
GiveAccountBalance
GiveAllOutcomeTransactions
GiveAllSpendingsPerCategoryDetailed
GiveMonthlyReportPerGoal
LoginUser
PasswordMatch
ReceiveCategoryListFromUser
ReceiveGoalListFromUser
RegisterUser
ReportOfSpendingsPerCard
SearchCategoryInDb
searchInDbForAnExchange
SetListOfCategories
SetUserConnected
UpdateCategory
UpdateCreditAccount
UpdateExchangeHistory
UpdateMonetaryAccount
UpdateTransaction
UpdateUser

Anexo 3: Diagramas de interacción



Queremos dejar en claro que por falta de tiempo no nos dio para hacer los diagramas de secuencia. Pedimos disculpas pero si no era por tal falta de tiempo los mismos hubiesen sido realizados con exito. Hicimos el mas dificil como pedido el de reporte.

Anexo 4: Modelo de tablas

Tabla Usuario

	UserId	FirstName	LastName	Email	Password	Address
1	NULL	NULL	NULL	NULL	NULL	NULL

Tabla Categorías

	CategoryId	Name	CreationDate	Status	Type	UserId
1	NULL	NULL	NULL	NULL	NULL	NULL

Tabla Goals

	GoalId	Title	MaxAmountToSpe…	CurrencyOfAmou…	UserId
1	NULL	NULL	NULL	NULL	NULL

Tabla CategoryGoal

	CategoriesOfGoalCategoryId	CategoryGoalsGoalId
1	NULL	NULL

Tabla CreditCardAccounts

	AccountId	IssuingBank	Last4Digits	AvailableCredit	ClosingDate
1	NULL	NULL	NULL	NULL	NULL

Tabla ExchangeHistory

	ExchangeHistor…	Currency	Value	ValueDate	UserId
1	NULL	NULL	NULL	NULL	NULL

Tabla MonetaryAccount

	AccountId	Amount
1	NULL	NULL

Tabla Transaction

	TransactionId	Title	CreationDate	Amount	Currency	Type	CategoryId	AccountId
1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

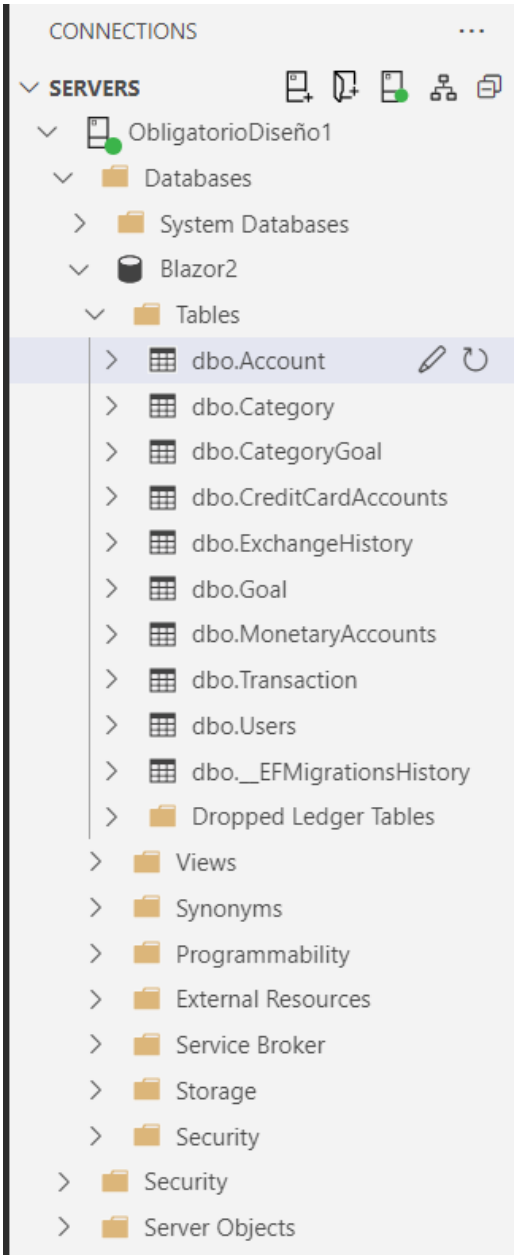
Tabla Cuenta

	AccountId	Name	Currency	CreationDate	UserId
1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

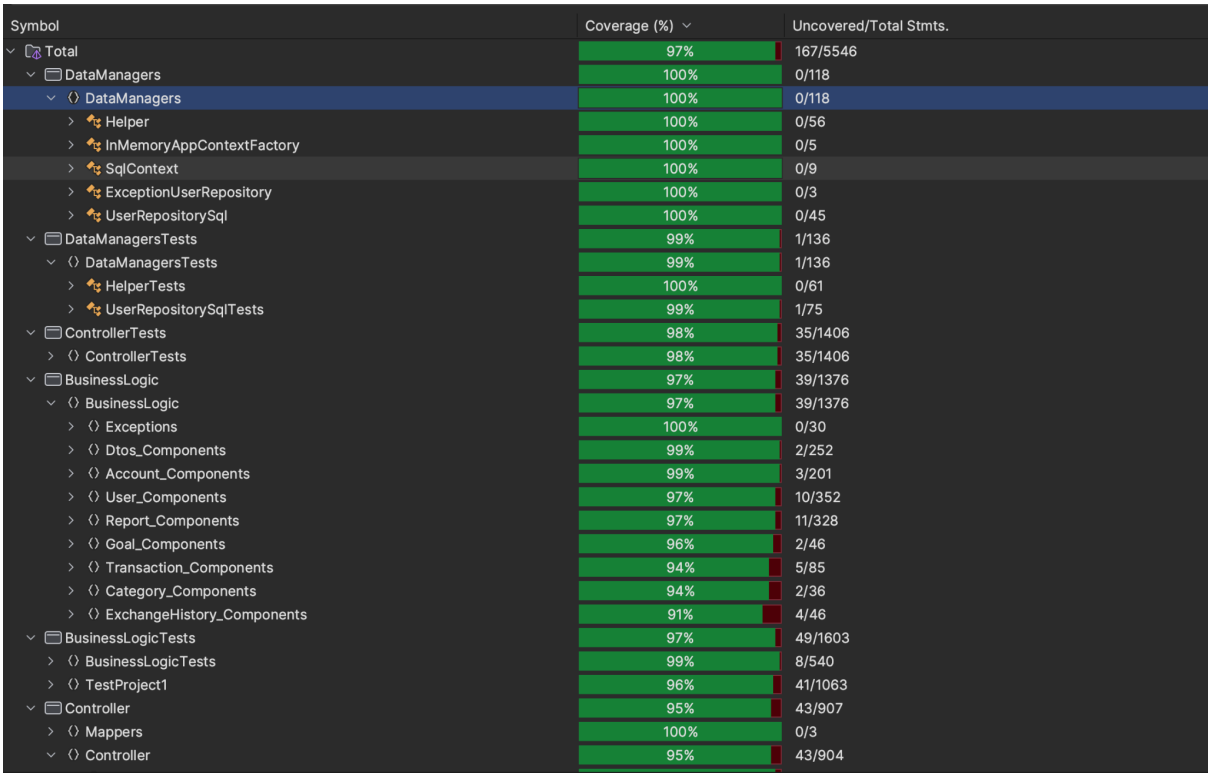
Tabla de migraciones

	MigrationId	ProductVersion
1	20231111020001_RefactorDb	7.0.13

Imagen general de la conexión



Anexo 5: Tests



Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	97%	167/5546
DataManagers	100%	0/118
DataManagers	100%	0/118
Helper	100%	0/56
InMemoryAppContextFactory	100%	0/5
SqlContext	100%	0/9
ExceptionUserRepository	100%	0/3
UserRepositorySql	100%	0/45
DataManagersTests	99%	1/136
DataManagersTests	99%	1/136
HelperTests	100%	0/61
UserRepositorySqlTests	99%	1/75
ControllerTests	98%	35/1406
ControllerTests	98%	35/1406
BusinessLogic	97%	39/1376
BusinessLogic	97%	39/1376
Exceptions	100%	0/30
Dtos_Components	99%	2/252
Account_Components	99%	3/201
User_Components	97%	10/352
Report_Components	97%	11/328
Goal_Components	96%	2/46
Transaction_Components	94%	5/85
Category_Components	94%	2/36
ExchangeHistory_Components	91%	4/46
BusinessLogicTests	97%	49/1603
BusinessLogicTests	99%	8/540
TestProject1	96%	41/1063
Controller	95%	43/907
Mappers	100%	0/3
Controller	95%	43/904