Universidad ORT Uruguay
Facultad de Ingeniería

OBLIGATORIO 1
DISEÑO DE APLICACIONES 1

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

Gonzalo Camejo (256665)

Ignacio Quevedo (271557)

Tutorores: Diego Balbi - Gastón Mousqués

Grupo: M5A

Repositorio: https://github.com/IngSoft-DA1-2023-2/271557_256665
Link al video: https://youtu.be/SVpJGAmeJBk
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 que podemos asegurar de que:
• La obra fue producida en su totalidad mientras realizábamos el Obligatorio 1 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

Ignacio Quevedo
12 de Octubre del 2022

12 de Octubre del 2023

Abstract

El obligatorio a realizar está compuesto por 4 grandes secciones.
Mediante las secciones

Palabras Clave

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

Índice

Descripción general del trabajo realizado y del sistema.
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, este trabajo es denominado 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. Para ello 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: 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 su respectivo implicite solamente conllevaría a un riesgo a niveles de seguridad.

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. La features, 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.
Para ello, las ramas salían del área local del programador y eran enviadas al repositorio remoto (residente en GitHub). Y una vez allí se realizaba el mergo con la rama develop. Hacemos noción de que para poder realizar el merge con la rama develop primero se tenia que realizar una pull request a un reviewer, el cual este mismo sería el otro integrante del equipo. Esto fue conllevado dado que subir ciertas características a la rama de producción no es algo que se debía de tomar a la ligera, por lo cual únicamente una feature seria mergeada con develop una vez que todos los integrantes del grupo estuvieran conformes con la decision y hayan entendido los objetivos 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 claves 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 trabaja 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.

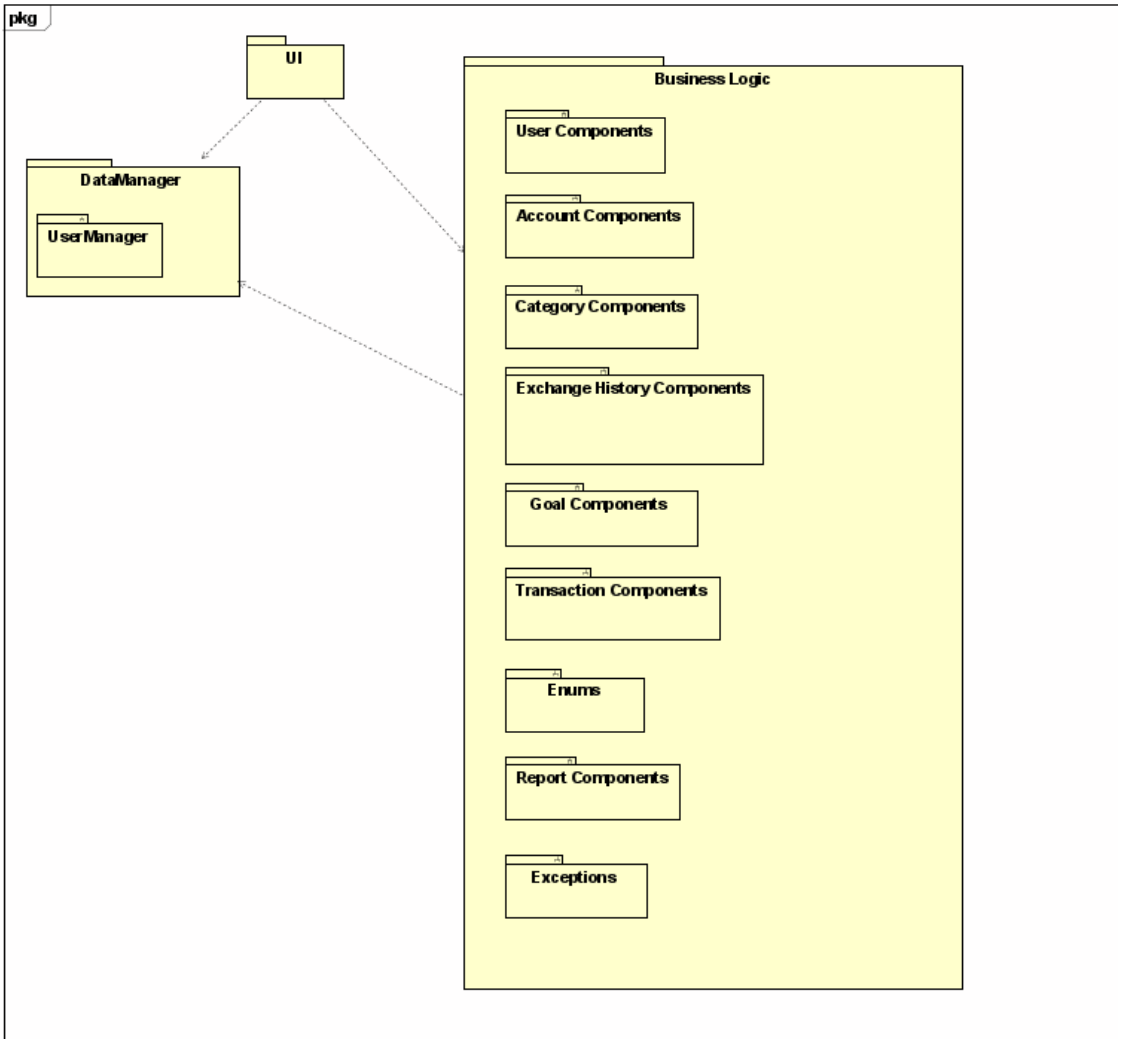
A su vez, desarrollar toda la parte lógica del sistema mediante el uso de TDD facilitó a gran escala la comprensión de las funcionalidades que eran de total aclamación a implementar, logrando por consiguiente poner objetivos claros para cada una de las clases y requerimientos a implementar. Queremos dejar explicito 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 nos los hubiéramos salteado, ya que antes de 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 refactoreo 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, facilito en una gran magnitud la implementación del sistema.

Para las ramas en las cuales se implementan nuevas funcionalidades se les fue denomina siguiente el patrón de diseño a mostrar, 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 decision fue tomada por el tema de la organización y entendimiento del repositorio remoto. Al principio cada integrante realizaba 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. Esto con el fin de evitar errores de mergeo en un porvenir.

Diagrama de paquetes:

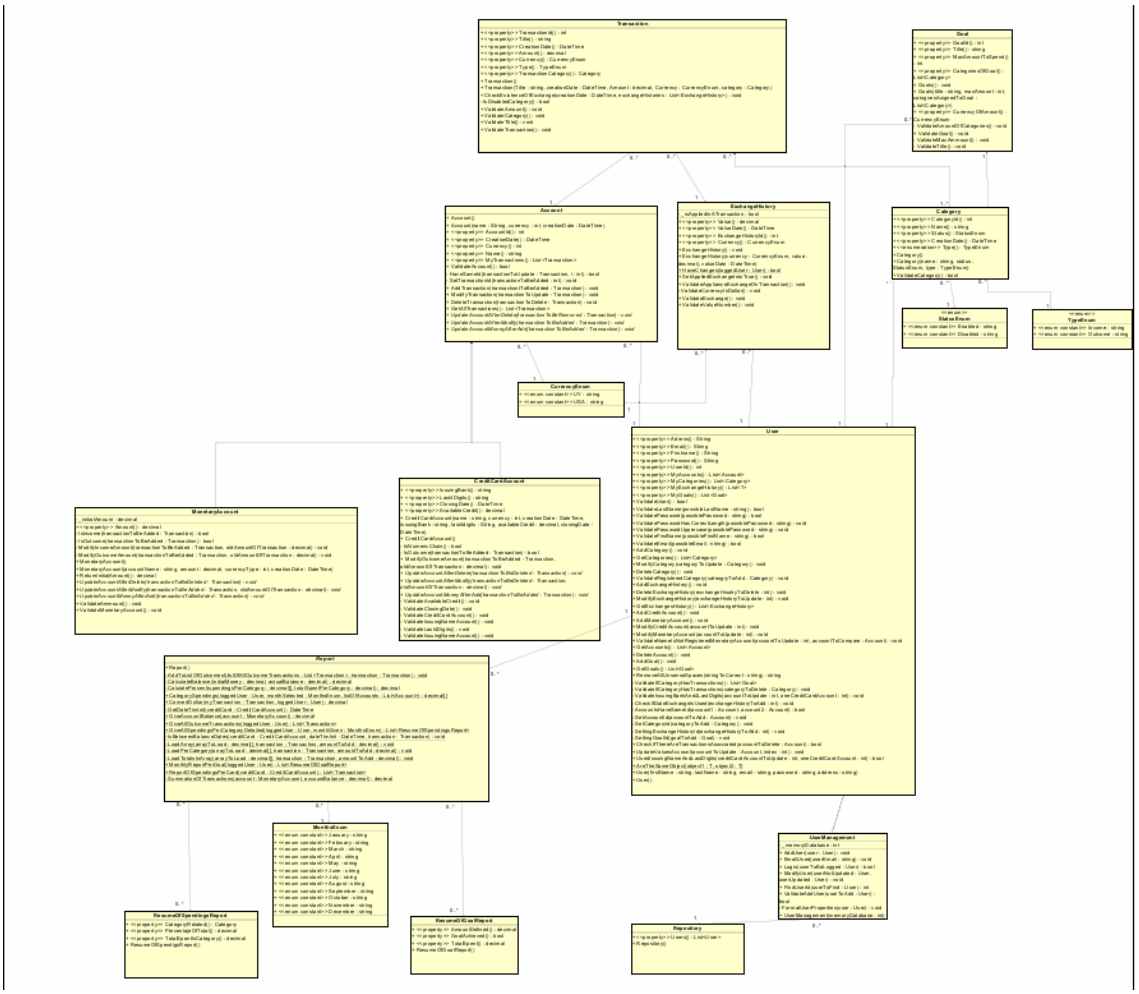


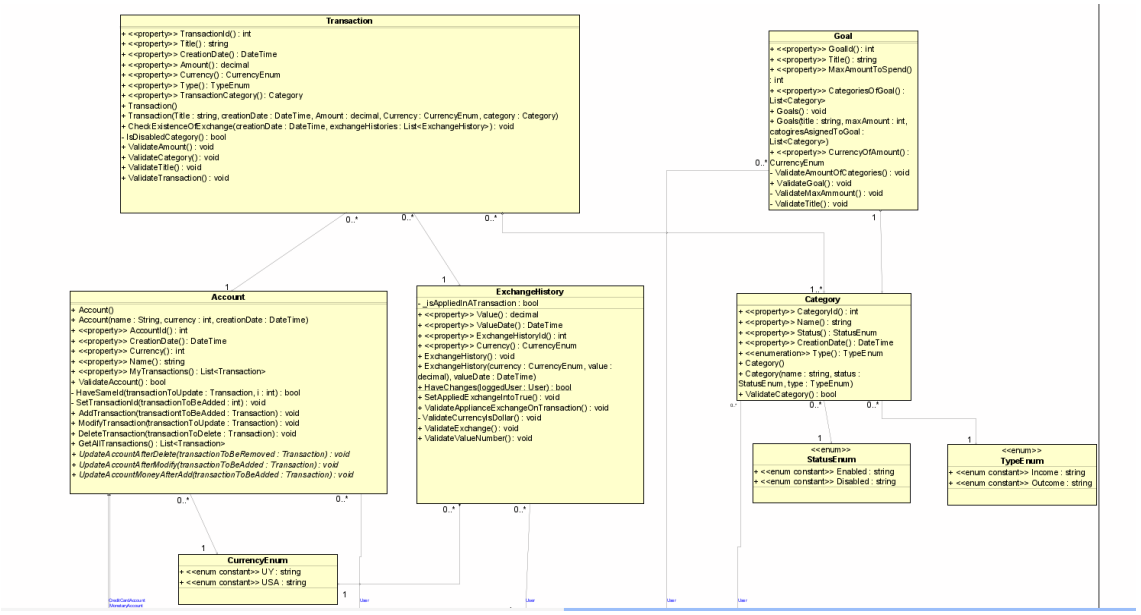
Con el fin de conseguir nuestro diagrama de paquetes, tuvimos que identificar los grupos en las cuales podíamos diferenciar las distintas clases. En un inicio, debimos llegar a un acuerdo de cuantos paquetes eran necesarios para el proyecto, ya que esto condiciona a futuro como se distribuye el trabajo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

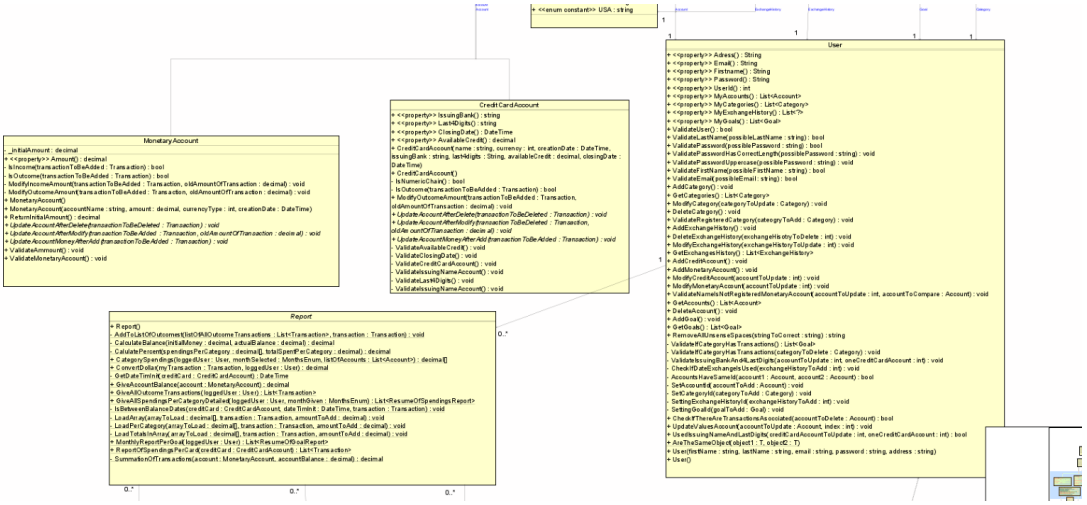
Concluimos en una aplicación separada en distintas capas ya que consideramos pertinente separar la Business Logic de lo que está relacionado con la User Interface ya que es la base de lo que debe estar separado dentro de la aplicación con el fin de asegurar dicha mantenibilidad futura, sino nuestro código 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.

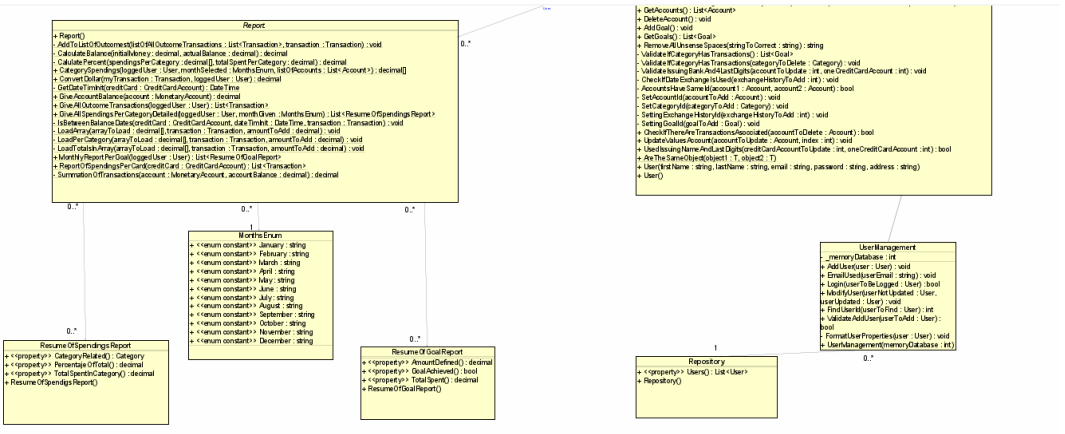
Un ejemplo claro de esta reusabilidad se ve reflejado en una iteración de negocio, si deseamos cambiar la interfaz por algún motivo, no deberíamos estar buscando entre lógica de negocio e interfaz para saber dónde debemos cambiar nuestro

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.




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
Goal
+ <<property>> GoalId() : int
+ <<property>> Title() : string
+ <<property>> MaxAmountToSpend() : int
+ <<property>> CategoriesOfGoal() : List<Category>
+ Goals() : void
+ Goals(title : string, maxAmount : int, catogoriesAsignedToGoal : List<Category>)
+ <<property>> CurrencyOfAmount() : CurrencyEnum
- ValidateAmountOfCategories() : void
+ ValidateGoal() : void
- ValidateMaxAmmount() : void
- ValidateTitle() : void
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
- SetTransactionId(transactionToBeAdded : int) : void
+ 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
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
- ValidateAppliancaExchangeOnTransaction() : void
- ValidateCurrencyIsDollar() : void
+ ValidateExchange() : void
+ ValidateValueNumber() : void
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
<<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

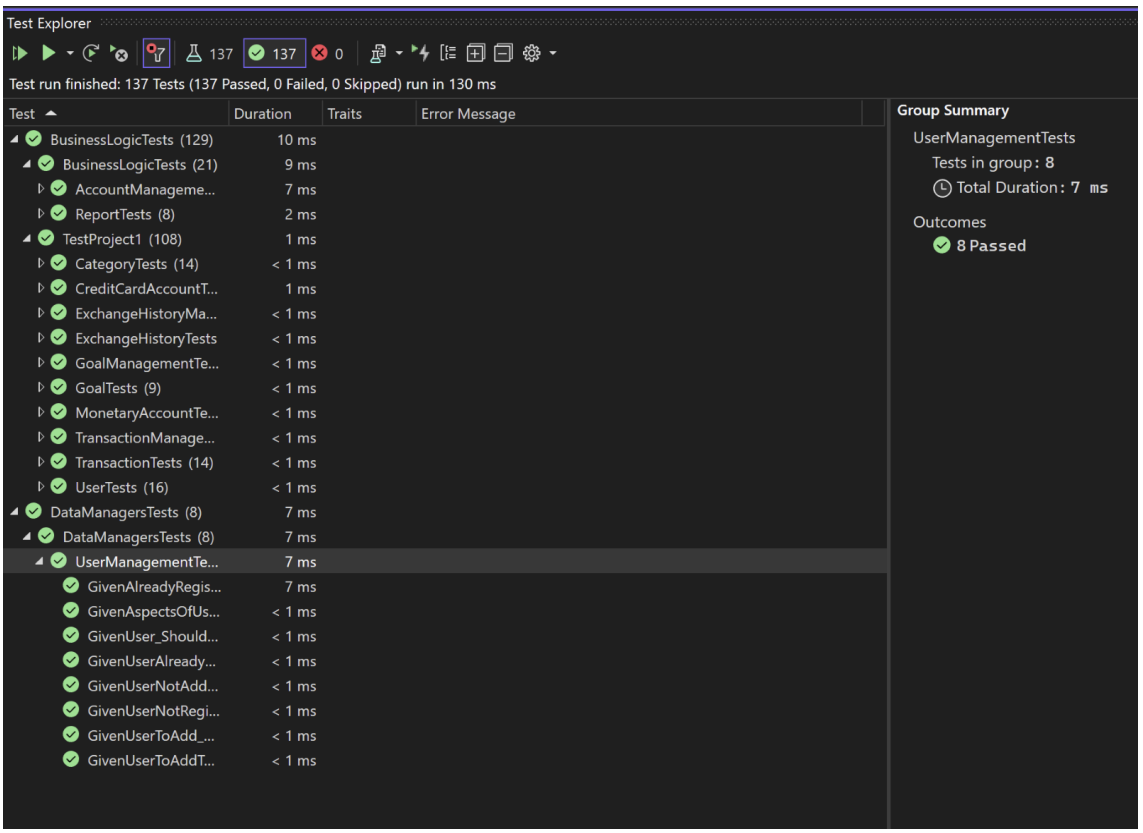

+ <<enum constant>> USA : string
User
+ <<property>> Adress() : String
+ <<property>> Email() : String
+ <<property>> Firstname() : String
+ <<property>> Password() : String
+ <<property>> UserId() : int
+ <<property>> MyAccounts() : List<Account>
+ <<property>> MyCategories() : List<Category>
+ <<property>> 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(exchangeHistoryToDelete : int) : void
+ ModifyExchangeHistory(exchangeHistoryToUpdate : int) : void
+ GetExchangeHistory() : 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
+ AddGoal() : void
+ GetGoals() : List<Goal>
+ RemoveAllUnnecessarySpaces(stringToCorrect : string) : string
- ValidateIfCategoryHasTransactions() : List<Goal>
- ValidateIfCategoryHasTransactions(categoryToDelete : Category) : void
- ValidateIssuingBankAndLastDigits(accountToUpdate : int, oneCreditCardAccount : int) : void
- CheckIfDateExchangeIsUsed(exchangeHistoryToAdd : int) : void
- AccountsHaveSameId(account1 : Account, account2 : Account) : bool
- SetAccountId(accountToAdd : Account) : void
- SetCategoryId(categoryToAdd : Category) : void
- SettingExchangeHistoryId(exchangeHistoryToAdd : int) : void
- SettingGoalId(goalToAdd : Goal) : void
+ CheckIfThereAreTransactionsAssociated(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)
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>> LastNDigits() : string
+ <<property>> ClosingDate() : DateTime
+ <<property>> AvailableCredit() : decimal
+ CreditCardAccount(name : string, currency : int, creationDate : DateTime, issuingBank : string, lastNDigits : String, availableCredit : decimal, closingDate : DateTime)
+ CreditCardAccount()
- IsNumericChain() : bool
- IsOutcome(transactionToBeAdded : Transaction) : bool
+ ModifyOutcomeAmount(transactionToBeAdded : Transaction, oldAmountOfTransaction : decimal) : void
+ UpdateAccountAfterDelete(transactionToBeDeleted : Transaction) : void
+ UpdateAccountAfterModify(transaction ToBeDeleted : Transaction, oldAmountOfTransaction : decimal) : void
+ UpdateAccountMoneyAfterAdd(transaction ToBeAdded : Transaction) : void
- ValidateAvailableCredit() : void
- ValidateClosingDate() : void
- ValidateCreditCardAccount() : void
- ValidateIssuingNameAccount() : void
- ValidateLastNDigits() : void
- ValidateIssuingNameAccount() : void
Report
+ Report()
- AddToListOfOutcomes(listOfAllOutcomeTransactions : List<Transaction>, transaction : Transaction) : void
- CalculateBalance(initialMoney : decimal, actualBalance : decimal) : decimal
- CalculatePercentSpendingsPerCategory : decimal[], totalSpentPerCategory : decimal) : decimal
+ CategorySpendings(loggedUser : User, monthSelected : MonthsEnum, listOfAccounts : List<Account>) : decimal[]
+ ConvertDollar(myTransaction : Transaction, loggedUser : User) : decimal
- GetDateTimInMs(creditCard : CreditCardAccount) : DateTime
+ GiveAccountBalance(account : MonetaryAccount) : decimal
+ GiveAllOutcomeTransactions(loggedUser : User) : List<Transaction>
+ GiveAllSpendingsPerCategoryDetailed(loggedUser : User, monthGiven : MonthsEnum) : List<ResumeOfSpendingsReport>
- IsBetweenBalanceDates(creditCard : CreditCardAccount, dateTimInt : 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>
+ ReportOfSpendingsPerCard(creditCard : CreditCardAccount) : List<Transaction>
- SummationOfTransactions(account : MonetaryAccount, accountBalance : decimal) : decimal


Report
+ Report()
- AddToListOfOutcomes(listOfAllOutcomeTransactions : List<Transaction>, transaction : Transaction) : void
- CalculateBalance(initialMoney : decimal, actualBalance : decimal) : decimal
- CalculatePercentSpendingsPerCategory : decimal[], totalSpentPerCategory : decimal) : decimal
+ CategorySpendings(loggedUser : User, monthSelected : MonthsEnum, listOfAccounts : List<Account>) : decimal[]
+ ConvertDollar(myTransaction : Transaction, loggedUser : User) : decimal
- GetDateTimInMs(creditCard : CreditCardAccount) : DateTime
+ GiveAccountBalance(account : MonetaryAccount) : decimal
+ GiveAllOutcomeTransactions(loggedUser : User) : List<Transaction>
+ GiveAllSpendingsPerCategoryDetailed(loggedUser : User, monthGiven : MonthsEnum) : List<ResumeOfSpendingsReport>
- IsBetweenBalanceDates(creditCard : CreditCardAccount, dateTimInt : 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>
+ ReportOfSpendingsPerCard(creditCard : CreditCardAccount) : List<Transaction>
- SummationOfTransactions(account : MonetaryAccount, accountBalance : decimal) : decimal
+ GetAccounts() : List<Account>
+ DeleteAccount() : void
+ AddGoal() : void
+ GetGoals() : List<Goal>
+ RemoveAllUnnecessarySpaces(stringToCorrect : string) : string
- ValidateIfCategoryHasTransactions() : List<Goal>
- ValidateIfCategoryHasTransactions(categoryToDelete : Category) : void
- ValidateIssuingBankAndLastDigits(accountToUpdate : int, oneCreditCardAccount : int) : void
- CheckIfDateExchangeIsUsed(exchangeHistoryToAdd : int) : void
- AccountsHaveSameId(account1 : Account, account2 : Account) : bool
- SetAccountId(accountToAdd : Account) : void
- SetCategoryId(categoryToAdd : Category) : void
- SettingExchangeHistoryId(exchangeHistoryToAdd : int) : void
- SettingGoalId(goalToAdd : Goal) : void
+ CheckIfThereAreTransactionsAssociated(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()
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
UserManagement
- _memoryDatabase : int
+ Add User(user : User) : void
+ EmailUsed(userEmail : string) : void
+ Login(userToBeLogged : User) : bool
+ Modify(UserUserNotUpdated : User, userUpdated : User) : void
+ FindUserId(userToFind : User) : int
+ Validate Add User(userToAdd : User) : bool
- FormatUserProperties(user : User) : void
+ UserManagement(memoryDatabase : int)
Repository
+ <<property>> Users() : List<User>
+ Repository()
ResumeOfSpendingsReport
+ <<property>> CategoryRelated() : Category
+ <<property>> Percentage Of(total) : decimal
+ <<property>> TotalSpentInCategory() : decimal
+ ResumeOfSpendingsReport()
ResumeOfGoalReport
+ <<property>> AmountDefined() : decimal
+ <<property>> GoalAchieved() : bool
+ <<property>> TotalSpent() : decimal
+ ResumeOfGoalReport()

Para el diagrama de clases, lo que hicimos fue buscar una separación de las clases clara, con el fin de dividir las responsabilidades de cada clase y poder así, asignar distintas responsabilidades a las mismas. Para de esta manera lograr un diseño más mantenible del codigo.
Decidimos delegar responsabilidades al usuario debido a que nos pareció la mejor opción viable ya que lo hicimos pensando que en un futuro usaríamos blazor y el pasaje entre pages por medio del cascading value, por esto decidimos delegar tantas responsabilidades al usuario y que sea el mismo el que se encarga de manejar todas las listas que le conciernen.
Luego nos pareció una buena idea representar la lista de todos los usuarios separada ya que la misma representa una especie de base de datos encargada de validar el loggeo de los usuarios, es por esto que la misma presenta una lista de usuarios únicamente
En un inicio realmente nuestro UML no era claro, pero luego con el avance del proyecto fue tomando distintos caminos y mediante muchas iteraciones del mismo llegamos al UML que se encuentra adjuntado, Nos pareció importante señalar todas las relaciones que se dan entre las clases ya que las mismas nos ayudaron durante el proyecto a saber que debíamos hacer en cada clase y nos presentó una idea inicial y general para poder darle un punto de partida a nuestro proyecto.

Pruebas Unitarias:

A continuación, 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, consiguiendo un coverage de las pruebas unitarias de un 94.67%

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 uno logrando así la coverage mostrada, la cual se encuentra 6 puntos por debajo del 100% debido a que las exception dejan una linea sin cubrir cada vez que se ejecutan y es por este motivo que no podemos alcanzar el 100% este tema fue discutido ya con los docentes quienes comentaron este error y es una falla que todos deberíamos tener por un tema de funcionamiento de visual studio code y las pruebas unitarias.