UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

INSEGNAMENTO DI BASI DI DATI E SISTEMI INFORMATIVI I

ANNO ACCADEMICO 2023/2024

SavingMoneyUnina

Autori:

Luigi CESARO	N86004727	luigi.cesaro@studenti.unina.it
Federico DE NICOLA	N86004507	federico.denicola@studenti.unina.it

Docente:

Prof. Mara SANGIOVANNI

Indice

Descrizione del progetto.....3
Filosofia di progetto.....3
Progettazione.....4
Class Diagram UML.....4
Diagramma ER.....5
Ristrutturazione.....6
Class Diagram ristrutturato.....6
Approccio alla ristrutturazione.....6
Dizionari.....7
Dizionario delle classi.....7
Dizionario delle associazioni.....8
Dizionario dei vincoli.....9
Progettazione Logica.....10
Tabelle.....10
Traduzione delle associazioni.....10
Family - User.....10
User - BankAccount.....10
BankAccount - Card.....10
Card - Transaction.....10
Portfolio - Transaction.....10
Schema Fisico.....12
Tabelle.....12
Family.....12
User.....12
BankAccount.....12
Card.....13
Category.....13
Portfolio.....13
Transaction.....14
Triggers e funzioni.....14
typeCard.....14
typeTransaction.....14
checkPortfolioName.....15
checkBalanceCard.....15
checkValidAmount.....15
checkPlafondDebitCard.....16
expireDateWhenTransaction.....16
checkTransactionFamily.....16

Descrizione del progetto

SavingMoneyUnina è un sistema completo per il monitoraggio delle finanze personali o familiari. Consente di gestire più carte di credito o debito, sia proprie che di altri membri della famiglia, tenendo traccia delle transazioni in entrata e in uscita.
Grazie alla sua funzionalità di suddivisione delle transazioni in diversi portafogli, gli utenti possono categorizzare le spese in modo efficace, ad esempio distinguendo tra svago, spese mediche, stipendio e bollette.
Il sistema offre la flessibilità di sincronizzare automaticamente le transazioni effettuate con una carta specifica con un gruppo predefinito oppure di inserire manualmente le transazioni. Questo strumento aiuta gli utenti a mantenere il controllo delle loro finanze, facilitando la gestione e l'analisi dei flussi di denaro in modo organizzato e intuitivo.

NOTA: nel caso ci fossero problemi riguardanti la visione dei vari diagrammi, è possibile visionarli accedendo alla repository Github del progetto.

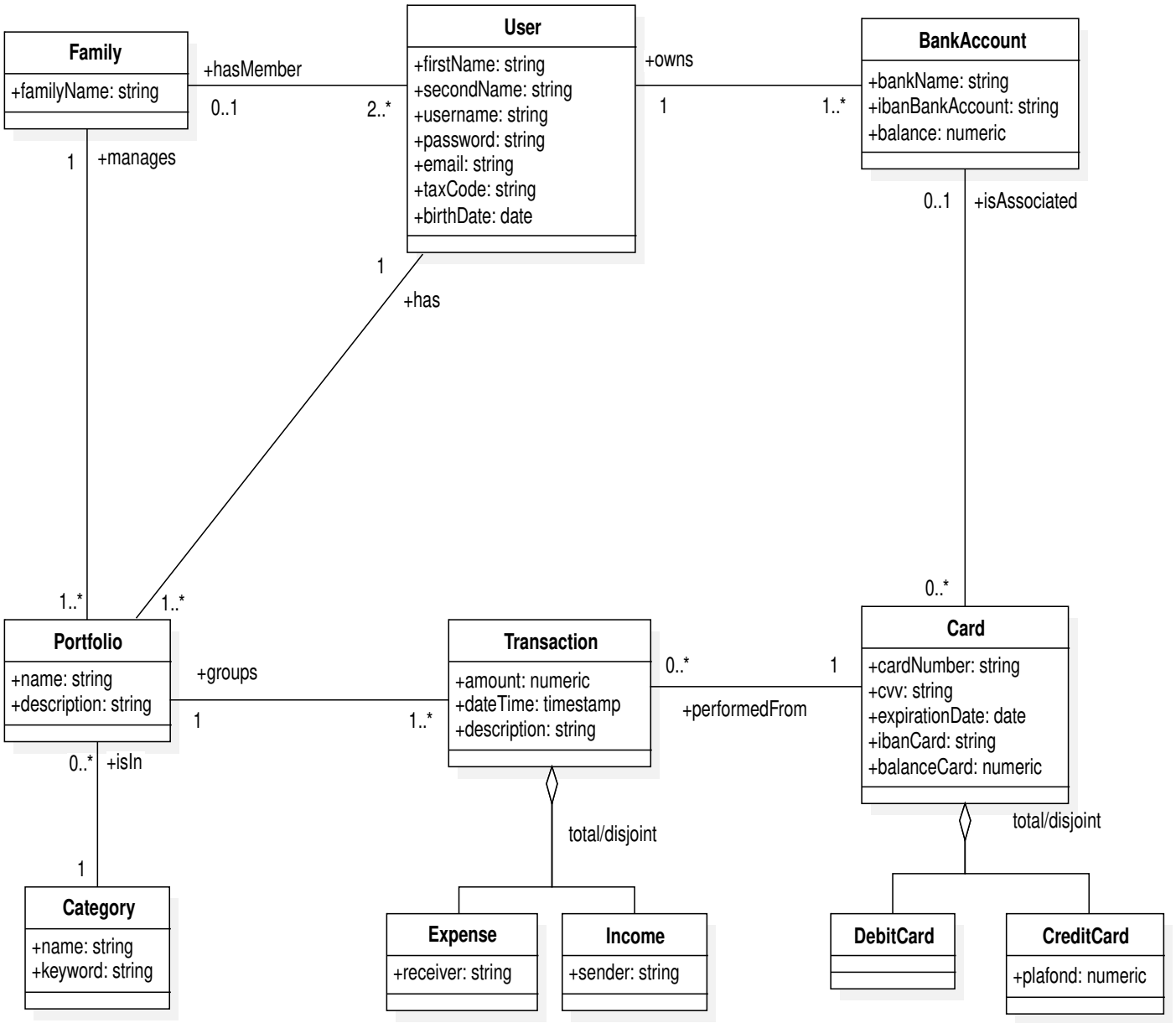
https://github.com/n00w4/SavingMoneyUNINA

Filosofia di progetto

L'utente (***User***) può possedere uno o più conti bancari (***BankAccount***), al quale potranno essere associate da zero o più carte (***Card***). Le carte potranno essere fondamentalmente di debito (***DebitCard***) o di credito (***CreditCard***).
Le transazioni (***Transaction***) possono essere effettuate da una carta e l'utente potrà effettuare zero o più transazioni che potranno essere raggruppate in appositi portafogli (***Portfolio***).
Il portafoglio può contenere più transazioni e saranno categorizzati attraverso le ***Category***.
Una transazione si divide a sua volta in transazione d'uscita (***Expense***) e transazione d'entrata (***Income***).
Più utenti possono far parte di una famiglia (***Family***).

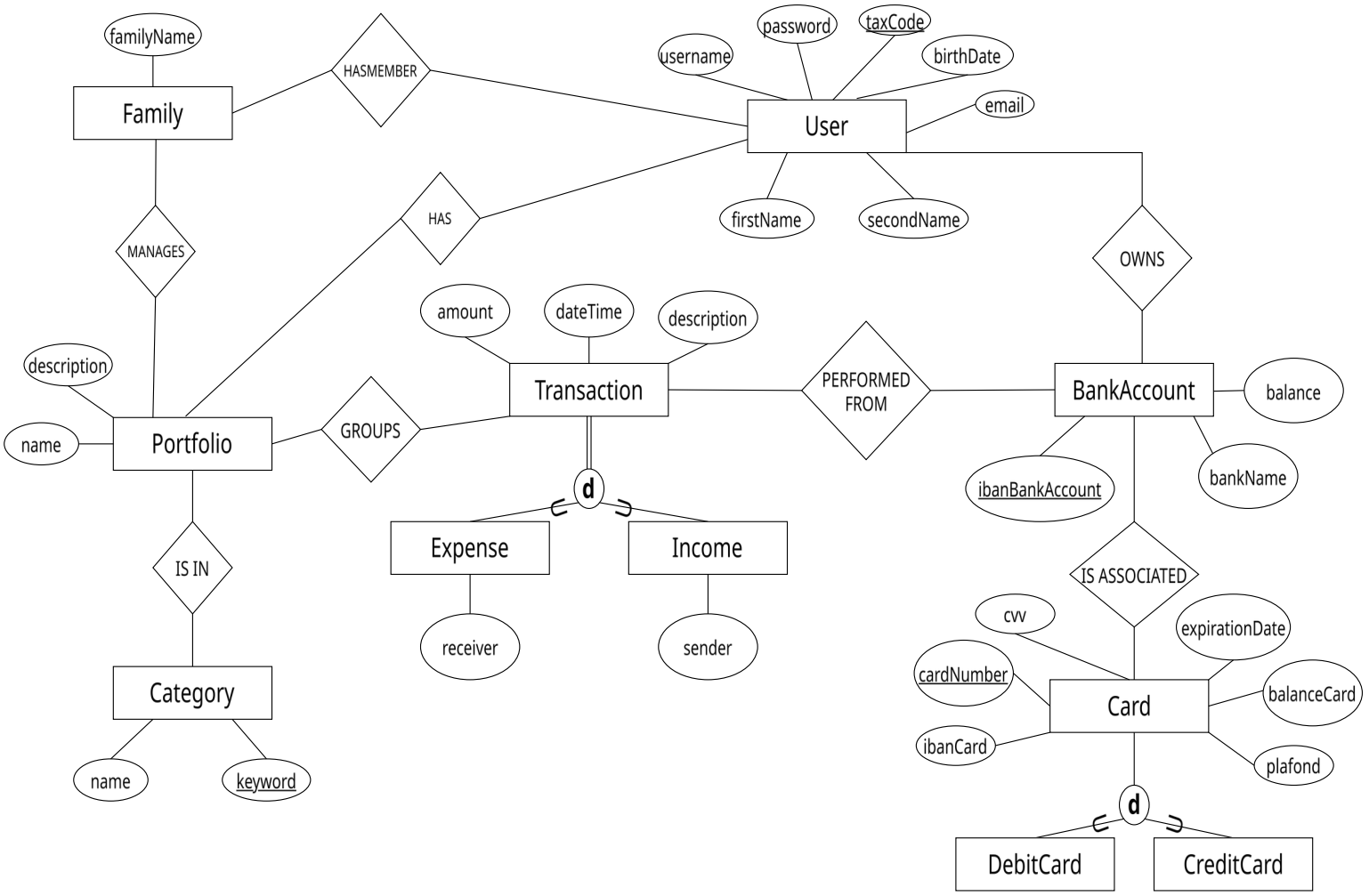
Progettazione

Class Diagram UML



L'UML è stato strutturato secondo la logica spiegata precedentemente.

Diagramma ER

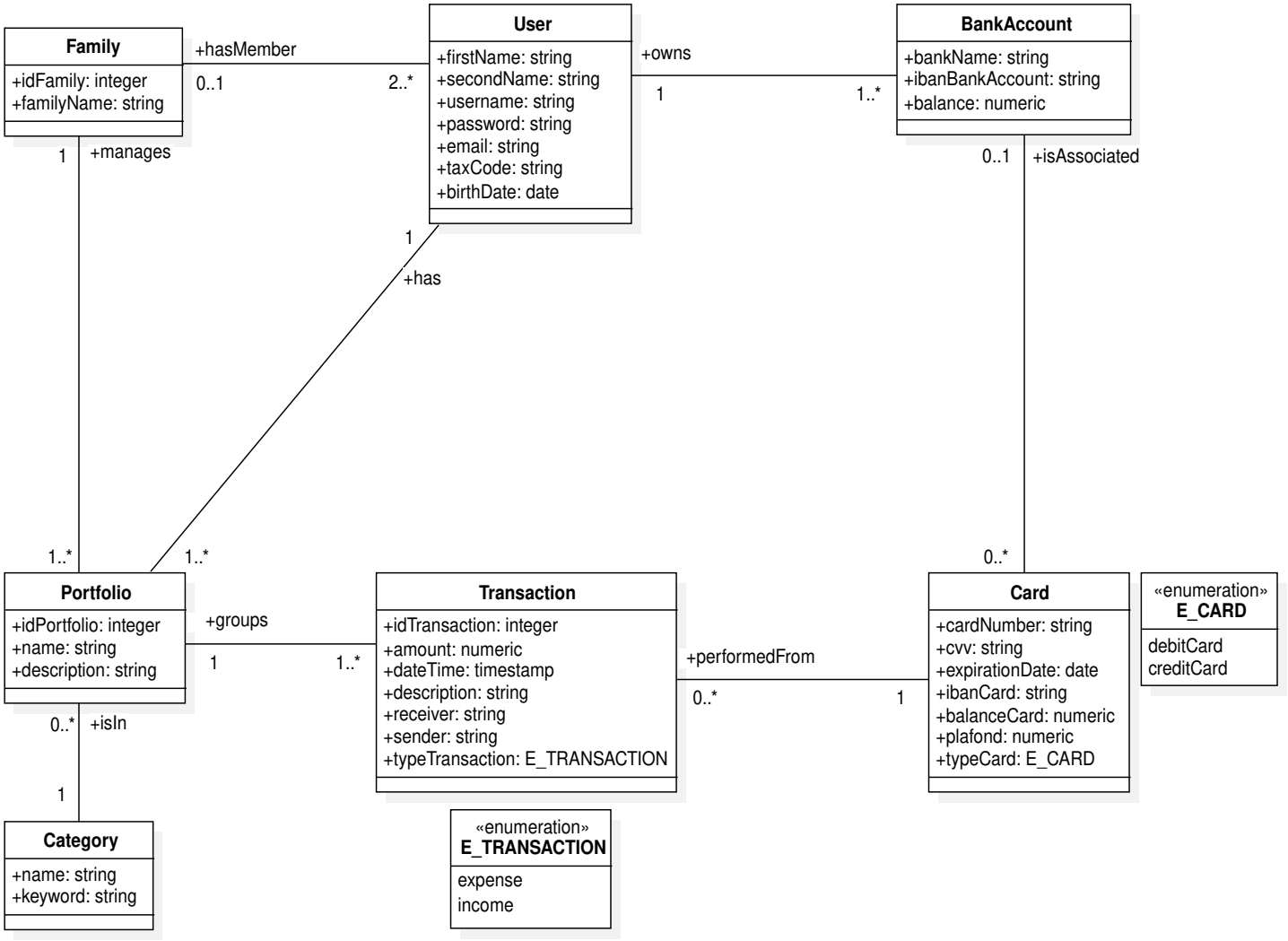


Come si può notare dal diagramma ER, le relazioni principali sono:

- per User → PERFORMS – OWNS – HAS : l'utente può eseguire transazioni, possedere conti bancari ed avere portafogli;
- per Portfolio → GROUPS: il portafoglio raggruppa le transazioni in entrata e in uscita;
- per BankAccount → IS ASSOCIATED: un conto bancario è associato a nessuna o più carte;
- per Card → PERFORMED FROM: una o più transazioni vengono effettuate da una carta;
- per Category → IS IN: nessuno o più portafogli possono essere contenuti in una categoria;
- per Family → HAS MEMBER: nessuna o una famiglia può avere 2 o più membri al suo interno.

Ristrutturazione

Class Diagram ristrutturato



Approccio alla ristrutturazione

L'UML è stato ristrutturato scegliendo di integrare enumerazioni con le classi Transaction e Card: è stato scelto questo approccio in quanto evita di creare sottoclassi che potrebbero risultare quasi inutili e fini a loro stesse.

L'enumerazione in Transaction con typeTransaction permette di poter identificare il tipo di transazione in un modo molto semplice. Lo stesso vale per Card con typeCard, che permette di identificare la sua carta di credito o di debito.

Dizionari

Dizionario delle classi

Classe	*Descrizione*	*Attributi*
User	Classe che rappresenta un utente.	**firstName**(*string*): Il nome dell'utente. **secondName**(*string*): Il cognome dell'utente. **username**(*string*): Username dell'utente. **password**(*string*): Password dell'utente. **email**(*string*): Email dell'utente. **taxCode**(*string*): Codice fiscale dell'utente. **birthDate**(*date*): Data di nascita dell'utente.
Family	Classe che rappresenta una famiglia.	**idFamily**(*integer*): Identifica una famiglia. **familyName**(*string*): Il nome dela famiglia
Transaction	Classe che rappresenta una transazione	**idTransaction**(*integer*): Identifica una transazione. **amount**(*numeric*): Importo della transazione. **dateTime**(*timestamp*): Data e ora della transazione. **description**(*string*): Descrizione della transazione. **receiver**(*string*): Destinatario della transazione. **source**(*string*): Fonte della transazione. **typeTransaction**(*E_TRANSACTION*): Tipo di transazione.
BankAccount	Classe che rappresenta un conto bancario.	**bankName**(*string*): Nome dell'istituto bancario. **ibanBankAccount**(*string*): IBAN del conto bancario. **balance**(*string*): Saldo del conto bancario.
Card	Classe che rappresenta una carta di credito o di debito.	**cardNumber**(*string*): Numero della carta di credito o di debito. **cvv**(*string*): CVV della carta di credito o di debito. **expirationDate**(*date*): Data di scadenza della carta di credito o di debito. **ibanCard**(*string*): IBAN della carta di credito o di debito. **balanceCard**(*numeric*): Saldo della carta di credito o di debito. **plafond**(*numeric*): Saldo spendibile della carta di credito o debito. **typeCard**(*E_CARD*): Tipo di carta di credito o di debito.
Portfolio	Classe che rappresenta un portafoglio.	**idPortfolio**(*integer*): Identifica un portafoglio. **name**(*string*): Nome del portafoglio. **description**(*string*): Descrizione del portafoglio.
Category	Classe che rappresenta una categoria.	**name**(*string*): Nome della categoria. **keyword**(*string*): Parola chiave associata alla categoria.

Dizionario delle associazioni

Associazione	Descrizione	Classi coinvolte
owns	Esprime la proprietà dell'utente di possedere un conto bancario.	**User [1]**: L'utente possiede 1 o più conti bancari. **PaymentMethod [1..*]**: Uno o più conti bancari sono in possesso dell'utente
groups	Esprime la proprietà del portafoglio di poter raggruppare le transazioni	**Portfolio [1]**: Un portafoglio può raggruppare più transazioni. **Transaction [1..*]**: Una o più transazioni possono essere raggruppate in un portafoglio
isAssociated	Esprime l'associazione di una carta con il suo conto bancario.	**BankAccount [1..*]**: Un conto bancario può essere associato con 0 o più carte (credito o debito). **Card [0..*]**: 0 o più carte (credito o debito) sono associate ad un conto bancario.
performedFrom	Esprime l'associazione tra la carta ed una o più transazioni effettuate.	**Card [1]**: Una carta (credito o debito) può essere il mezzo per eseguire una o più transazioni. **Transaction [0..*]**: Zero o più transazioni possono essere eseguite solo da una carta (credito o debito).
has	Esprime l'associazione tra il portafoglio e l'utente,	**Portfolio [1..*]**: Uno o più portafogli può essere di un utente. **User[1]**: Un utente può avere uno o più portafogli.
manages	Esprime la proprietà di una famiglia di poter gestire uno o più portafogli.	**Family [1]**: Una famiglia può avere uno o più portafogli. **Portfolio [1..*]**: Uno o più portafoglii può essere gestito da una famiglia.
hasMember	Esprime l'associazione tra una famiglia e gli utenti che la compongono.	**Family[1]**: Una famiglia può avere 2 o più utenti. **User[2..*]**: 2 o più utenti possono essere componenti di una famiglia.
isIn	Esprime l'associazione tra una categoria e i portafogli associati.	**Portfolio[0..*]**: Zero o più portafogli possono essere legati ad una categoria. **Category[1]**: Una categoria può essere legata a nessun o a più portafogli.

Dizionario dei vincoli

Vincolo	*Tipo*	*Descrizione*
checkValidPasswd	Di dominio	Il valore dell'attributo **password** della classe ***User*** deve contenere almeno 8 caratteri, avere almeno una lettera maiuscola, una minuscola, un numero e un carattere speciale.
checkValidUsername	Di dominio	L'**username** di un ***User*** è unico e non può essere NULL.
checkValidIBAN	Di dominio	Gli attributi **ibanCard** e **ibanBankAccount** di ***Card*** e ***BankAccount*** devono essere una stringa alfanumerica di 27 caratteri unica.
checkValidEmail	Di dominio	L'attributo **email** della classe ***User*** deve contenere una combinazione non nulla di lettere, numeri e simboli, seguiti da una chiocciola ("@"), altre lettere/simboli, un punto ("."), e finire con almeno due lettere.
checkValidTaxCode	Di dominio	L'attributo **taxCode** della classe ***User*** è unico e composto da 16 caratteri alfanumerici, secondo quest'ordine: 6 lettere, 2 cifre, 1 lettera, 2 cifre, 4 caratteri e 1 carattere.
checkPortfolioName	Intrarelazionale	Il nome (**name)** di un ***Portfolio*** è univoco solo per lo stesso utente e non può essere NULL.
checkValidCardNumber	Di dominio	L'attributo **cardNumber** di ***Card*** deve essere una stringa numerica di 16 caratteri.
checkCVV	Di dominio	L'attributo **cvv** di ***Card*** deve essere una stringa numerica di 3 caratteri.
checkBalanceCard	Intrarelazionale	Se **balanceCard** di ***Card*** è uguale a 0, il suo nuovo valore sarà quello di **plafond**. Quest'ultimo verrà posto di conseguenza a 0.
checkValidAmount	Interrelazionale	L'**amount** di una ***Transaction*** non può essere maggiore del **balance** della ***Card*** che ha effettuato la transazione.
typeCard	Di n-pla	Se l'attributo **plafond** di ***Card*** viene posto inizialmente a NULL, la carta sarà una **debitCard**.
checkPlafondDebitCard	Intrarelazionale	Se l'attributo **plafond** di ***Card*** è NULL, sarà automaticamente uguale al **balance** del ***BankAccount*** associato.
typeTransaction	Di n-pla	Se l'attributo **receiver** di ***Transaction*** è NULL, la transazione sarà di tipo **income**. Contrariamente sarà di tipo **expense** se l'attributo **source** sarà NULL.
isUserFamily	Interrelazionale	Se l'attributo **idFamily** in ***User*** è NULL, l'utente sarà singolo (non associato ad una famiglia).
checkBirthDateUser	Di dominio	In ***User*** l'attributo **birthDate** deve essere necessariamente antecedente alla data odierna.
expireDateWhenTransaction	Interrelazionale	Quando viene inserita una ***Transaction***, se effettuata con ***Card***, deve essere valida al momento della transazione.
checkValidExpirationDate	Di dominio	In ***Card*** l'attributo **expirationDate** deve essere necessariamente antecedente alla data odierna.
checkValidDateTime	Di dominio	In ***Transaction*** l'attributo **dateTime** deve essere necessariamente antecedente al timestamp di quel momento.
checkTransactionFamily	Interrelazionale	Una transazione di un ***Portfolio*** familiare può essere eseguita solo da una carta di un componente di quella stessa famiglia.
checkKeyword	Di dominio	Una **keyword** deve essere una parola di massimo 50 caratteri senza spazi e scritta in minuscolo.

Progettazione Logica

Tabelle

Legenda:

- in **grassetto** i nomi delle tabelle;
- gli attributi sottolineati sono primary keys;
- gli attributi sottolineati due volte sono foreign keys.

Family	idFamily, familyName
User	firstName, secondName, username, password, email, taxCode, birthDate, idFamily
BankAccount	bankName, balance, ibanBankAccount, taxCode
Transaction	idTransaction, amount, dateTime, description, receiver, sender, typeTransaction, cardNumber, idPortfolio
Card	cardNumber, cvv, expirationDate, ibanCard, balanceCard, plafond, typeCard, ibanBankAccount
Portfolio	idPortfolio, name, description, taxCode, idFamily, keyword
Category	name, keyword

Traduzione delle associazioni

Family - User

- 0..1 Family has member 2..* Users
- **Foreign Key**: **idFamily** in ***User***

User - BankAccount

- 1 Users owns 1..* BankAccount
- **Foreign Key**: **taxCode** in ***BankAccount***

BankAccount - Card

- 0..1 BankAccount is Associated with 0..* Cards
- **Foreign Key**: **ibanBankAccount** in ***Card***

Card - Transaction

- 0..* Transactions are performed from 1 Card
- **Foreign Key**: **cardNumber** in ***Transaction***

Portfolio - Transaction

- 1 Portfolio groups 0..* Transactions
- **Foreign Key**: **idPortfolio** in ***Transaction***

User – Portfolio

- 1 User has 1..* Portfolio
- **Foreign Key**: **idUser** in ***Portfolio***

Family – Portfolio

- 1 Family manages 1..* Portfolio
- **Foreign Key**: **idFamily** in ***Portfolio***

Portfolio – Category

- 0..* Portfolio is in 1 Category
- **Foreign Key**: **keyword** in ***Portfolio***

Schema Fisico

Per la progettazione fisica, ci si è avvalsi di pgAdmin 4 per gestire ed amministrare il database in PostgreSQL, con sviluppo e testing avvenuto sia tra GNU/Linux (Luigi Cesaro) che Windows (Federico De Nicola).

Tabelle

Family

```sql
-- Table Family
CREATE TABLE smu.Family (
    idFamily INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    familyName VARCHAR(50) NOT NULL
);
```

User

```sql
-- Table User
CREATE TABLE smu.User (
    firstName VARCHAR(50) NOT NULL,
    secondName VARCHAR(50) NOT NULL,
    username VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    taxCode CHAR(16) PRIMARY KEY,
    birthDate DATE NOT NULL,
    idFamily INTEGER REFERENCES smu.Family(idFamily),

    CONSTRAINT checkBirthDateUser CHECK ((birthDate < CURRENT_DATE)),
    CONSTRAINT checkValidEmail CHECK (email ~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'),
    CONSTRAINT checkValidTaxCode CHECK (taxCode ~ '^[A-Z]{6}[0-9]{2}[A-EHLMPR-T][0-9]{2}[A-Z][0-9]{3}[A-Z]$'),
    CONSTRAINT checkValidPasswd CHECK (password ~ '^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$')
);
```

BankAccount

```sql
-- Table BankAccount
CREATE TABLE smu.BankAccount (
    bankName VARCHAR(100) NOT NULL,
    balance NUMERIC(12,2) NOT NULL DEFAULT 0,
    ibanBankAccount CHAR(27) PRIMARY KEY,
    taxCode CHAR(16) REFERENCES smu.User(taxCode),

    CONSTRAINT checkValidIBAN CHECK (ibanBankAccount ~ '[A-Z]{2}[0-9]{2}[A-Z]{1}[0-9]{5}[0-9]{5}[0-9A-Z]{5}')
);
```

Card

```sql
-- Tipo E_CARD per gestire la differenza tra carte di credito e carte di debito
DROP TYPE IF EXISTS E_CARD CASCADE;
CREATE TYPE E_CARD AS ENUM ('creditCard', 'debitCard');

-- Table Card
CREATE TABLE smu.Card (
    cardNumber CHAR(16) PRIMARY KEY,
    cvv CHAR(3) NOT NULL,
    expirationDate DATE NOT NULL,
    ibanCard CHAR(27) NOT NULL,
    balanceCard NUMERIC(12,2) NOT NULL DEFAULT 0,
    plafond NUMERIC(12,2),
    typeCard E_CARD NOT NULL,
    ibanBankAccount CHAR(27) REFERENCES smu.BankAccount(ibanBankAccount),

    CONSTRAINT checkCVV CHECK (cvv ~ '^[0-9]{3}$'),
    CONSTRAINT checkValidExpirationDate CHECK ((expirationDate > CURRENT_DATE)),
    CONSTRAINT checkValidIBANCard CHECK (ibanCard ~ '[A-Z]{2}[0-9]{2}[A-Z]{1}[0-9]{5}[0-9]{5}[0-9A-Z]{5}')
);
```

Category

```sql
-- Table Category
CREATE TABLE smu.Category (
    name VARCHAR(255) NOT NULL,
    keyword VARCHAR(50) PRIMARY KEY,

    CONSTRAINT check_keyword CHECK (keyword ~ '^[a-z]{1,50}$')
);
```

Portfolio

```sql
-- Table Portfolio
CREATE TABLE smu.Portfolio (
    idPortfolio INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name VARCHAR(50) NOT NULL,
    description VARCHAR(255),
    taxCode CHAR(16) REFERENCES smu.User(taxCode),
    idFamily INTEGER REFERENCES smu.Family(idFamily),
    keyword VARCHAR(50) REFERENCES smu.Category(keyword)
);
```

Transaction

```sql
-- Tipo E_TRANSACTION per gestire la differenza tra entrata ed uscita
DROP TYPE IF EXISTS E_TRANSACTION CASCADE;
CREATE TYPE E_TRANSACTION AS ENUM ('expense', 'income');

-- Table Transaction
CREATE TABLE smu.Transaction (
    idTransaction INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    amount NUMERIC(12,2) NOT NULL,
    description VARCHAR(255) NOT NULL,
    dateTime timestamp NOT NULL,
    receiver VARCHAR(255),
    sender VARCHAR(255),
    typeTransaction E_TRANSACTION NOT NULL,
    cardNumber CHAR(16) REFERENCES smu.Card(cardNumber),
    idPortfolio INTEGER REFERENCES smu.Portfolio(idPortfolio),

    CONSTRAINT checkValidDateTime CHECK ((dateTime < CURRENT_TIMESTAMP))
);
```

Triggers e funzioni

typeCard

```sql
-- 1) typeCard: prima di inserire una nuova carta, controllo se il plafond è NULL per una debitCard e se non è NULL per una creditCard.

CREATE OR REPLACE FUNCTION smu.typeCardTrigger() RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.typeCard = 'debitCard' AND NEW.plafond IS NOT NULL THEN
        RAISE EXCEPTION 'ERROR: Plafond must be NULL for debitCard';
    END IF;
    IF NEW.typeCard = 'creditCard' AND NEW.plafond IS NULL THEN
        RAISE EXCEPTION 'ERROR: Plafond must NOT be NULL for creditCard';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkTypeCardTrigger
BEFORE INSERT ON smu.Card
FOR EACH ROW EXECUTE FUNCTION smu.typeCardTrigger();
```

typeTransaction

```sql
-- 2) typeTransaction: prima di inserire una nuova transazione, controllo se il tipo di transazione è income (receiver = NULL) o expense (sender = NULL).

CREATE OR REPLACE FUNCTION smu.typeTransactionTrigger() RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.typeTransaction = 'income' AND NEW.receiver IS NOT NULL THEN
        RAISE EXCEPTION 'ERROR: Receiver must be NULL for income';
    END IF;
    IF NEW.typeTransaction = 'expense' AND NEW.sender IS NOT NULL THEN
        RAISE EXCEPTION 'ERROR: Sender must be NULL for expense';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkTypeTransactionTrigger
BEFORE INSERT ON smu.Transaction
FOR EACH ROW EXECUTE FUNCTION smu.typeTransactionTrigger();
```

checkPortfolioName

```
-- 3) checkPortfolioName: prima di inserire un nuovo portfolio, controllo se il nome del portfolio è univoco per l'utente.
CREATE OR REPLACE FUNCTION checkPortfolioName()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM smu.Portfolio
        WHERE name = NEW.name
        AND taxCode = NEW.taxCode
    ) THEN
        RAISE EXCEPTION 'Il nome del portfolio "%" esiste già per questo utente', NEW.name;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER checkPortfolioNameTrigger
BEFORE INSERT OR UPDATE
ON smu.Portfolio
FOR EACH ROW
EXECUTE FUNCTION checkPortfolioName();
```

checkBalanceCard

```
--4)checkBalanceCard: Se balanceCard di Card è uguale a 0, il suo valore sarà quello di plafond. Quest'ultimo verrà posto di conseguenza a 0.

CREATE OR REPLACE FUNCTION smu.checkBalanceCard() RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.balanceCard = 0 AND NEW.plafond IS NOT NULL THEN
        NEW.balanceCard = NEW.plafond;
    NEW.plafond = 0;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkInsertBalanceCard
BEFORE INSERT ON smu.Card
FOR EACH ROW EXECUTE FUNCTION smu.checkBalanceCard();
```

checkValidAmount

```
--5)checkValidAmount: L'amount di una Transaction non può essere maggiore del balance della Card che ha effettuato la transazione.

CREATE OR REPLACE FUNCTION smu.checkValidAmount() RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.amount > NEW.balance THEN
        RAISE EXCEPTION 'ERROR: Transaction amount cannot be greater than the card balance';
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkTransactionAmount
BEFORE INSERT ON smu.Transaction
FOR EACH ROW EXECUTE FUNCTION smu.checkValidAmount();
```

checkPlafondDebitCard

```
--6)checkPlafondDebitCard: Se l'attributo plafond di Card è NULL, la transazione sarà automaticamnete uguale a balance del BankAccount associato.


CREATE OR REPLACE FUNCTION smu.checkPlafondDebitCard() RETURNS TRIGGER AS
$$
DECLARE
    bankAccountBalance NUMERIC;
BEGIN
    IF NEW.plafond IS NULL THEN
        SELECT balance INTO bankAccountBalance
        FROM smu.BankAccount
        WHERE iban = NEW.ibanBankAccount;

    NEW.amount = bankAccountBalance;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkAndSetTransactionAmount
BEFORE INSERT ON smu.Transaction
FOR EACH ROW EXECUTE FUNCTION smu.checkPlafondDebitCard();
```

expireDateWhenTransaction

```
--7)expireDateWhenTransaction: Quando viene inserita una Transaction, se effettuata con Card, deve essere valida al momento della transazione.

CREATE OR REPLACE FUNCTION smu.expireDateWhenTransaction() RETURNS TRIGGER AS
$$
DECLARE
    cardExpiration DATE;
    currentDate DATE = CURRENT_DATE;
BEGIN

    SELECT expitationDate INTO cardExpiration
    From smu.Card
    WHERE cardNumber = NEW.cardNumber;

    IF cardExpiration < currentDate THEN
        RAISE EXCEPTION 'ERROR: The card used for the transaction is expired';
    END IF;

END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER expireDate
BEFORE INSERT ON smu.Transaction
FOR EACH ROW EXECUTE FUNCTION smu.expireDateWhenTransaction();
```

checkTransactionFamily

```
--8)checkTransactionFamily: Una transazione di un Portfolio familiare può essere eseguita solo da una carta di un componente di quella stessa famiglia.

CREATE OR REPLACE FUNCTION smu.checkTransactionFamily() RETURNS TRIGGER AS
$$
DECLARE
    portfolioFamilyId INTEGER;
        cardFamilyId INTEGER;
BEGIN

    SELECT p.idFamily INTO portfolioFamilyId
    FROM smu.Portfolio p
    WHERE p.idPortfolio = NEW.idPortfolio;

    SELECT idFamily INTO portfolioFamilyId
    From smu.Card c
    JOIN smu.BankAccount b ON b.ibanBankAccount = c.ibanBankAccount
    JOIN smu.User u ON u.taxCode = b.taxCode
    WHERE c.cardNumber = NEW.cardNumber;

    IF portfolioFamilyId <> cardFamilyId THEN
        RAISE EXCEPTION 'ERROR: The card used for the transaction must belong to a family member of the same family as the portfolio';
    END IF;

    RETURN NEW;


END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER transactionFamily
BEFORE INSERT ON smu.Transaction
FOR EACH ROW EXECUTE FUNCTION smu.checkTransactionFamily();
```