BlueBudget

Universidade de Aveiro

Luís Moura, Maria Lavoura

Versão Final

BlueBudget

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Luís Moura, Maria Lavoura
(83808) luispedromoura@ua.pt, (84681) mjlavoura@ua.pt

07 Junho 2019

Resumo

Neste relatório descreve-se o Projecto Blue Budget realizado no âmbito da disciplina de Bases de Dados. O Projecto pretende simular o que seria a base de dados de uma aplicação de gestão de finanças pessoais, incluindo informação do utilizador, mas também informação reservada à gestão de backoffice da aplicação, como subscrições. Depois da Introdução sào apresentados, a Análise de Requisitos, Diagramas, SQL DDL associados à definição da estrutura da BD em SQL Server, SQL DML associadas a cada formulário gráfico, assim como Índices, Triggers, Stored Procedures e UDF utilizados.

Conteúdo

1 Introdução **1**

2 Análise de Requisitos **2**
2.1 Principais Entidades 2
2.1.1 Back-Office 2
2.1.2 Aplicação 2
2.2 Principais Queries 3

3 SQL DDL e DML **5**
3.1 Normalização 5
3.2 Índices 5
3.3 Stored Procedures (SP) e User Defined Functions (UDF) 6
6subsection.3.3.1
3.3.2 Stored Procedures 7
3.3.3 Triggers 10

4 Aplicação Windows Forms (C#) **14**
4.1 DB_IO - Conexão e Pedidos à Base de Dados 14
4.1.1 SqlCommand and Parameterizer 14
4.1.2 ExecuteNonQuery, ExecuteReader, ExecuteScalar 14
4.1.3 Insert, Delete, Update, SelectScalar, SelectReader 14
4.1.4 DBconnect, DBdisconnect 15
4.2 DB_API - Interface para Aplicações 15
4.3 Classes Windows Forms, Classes de Entidades - A aplicação 16

5 Conclusão **17**

Lista de Figuras

2.1 Diagrama de Relação de Entidades 3
2.2 Esquema Relacional 3

3.1 Índices da Relação Transaction 6
3.2 UDF Annual Statistics 6
3.3 Procedimento de Inserção de uma Categoria 7
3.4 Actualização do utilizador 8
3.5 Eliminação de uma conta 8
3.6 Procedimento de seleção de orçamento 9
3.7 Trigger de actualização de saldo de duas carteiras na inserção de uma transferência 11

4.1 Pilha de chamadas de funções para leitura na classe DB_IO 15
4.2 Exemplo de assinatura dos métodos da classe DB_API 16
4.3 Screenshots da aplicação em Windows Forms 16

Capítulo 1

Introdução

O objetivo do projeto *"BlueBudget"* é a criação de uma base de dados que representa a informação necessária para a gestão e utilização de uma aplicação de controlo de finanças pessoais. Também uma interface em Windows Forms com vertente de uso Back-Office para gestão de utilizadores e vertente de uso pessoal de cada utilizador. Na base de dados procurou-se representar o máximo de elementos possíveis, sendo eles **Utilizadores** (com ou sem **Subscrição**), Contas de Utilizador (1 ou mais por utilizador), cada uma com uma ou várias **Carteiras** às quais poderiam ser associadas **Transações**, **Transferências**, **Empréstimos**, **Objectivos**, **Ações**, **Categorias** e respectivos **Orçamentos**.

O projecto focou-se não só na criação das várias entidades mas principalmente no desafio de atualização dos valores das várias devido às interligações intrínsecas entre as mesmas.

Capítulo 2

Análise de Requisitos

2.1 Principais Entidades

2.1.1 Back-Office

- Users;
- Subscriptions;
- Money Accounts;

A Aplicação permite a distinção entre utilizadores preferenciais ou livres. Um utilizador pode ter várias Contas (por exemplo: "Conta Pessoal", "Conta Biscates", etc). Uma Conta pode ser gerida por mais do que um utilizador (por exemplo: conta familiar pai/filho, conta negócio gestor/empregados)

2.1.2 Aplicação

- Wallets;
- Categories;
- Budgets;
- Goals;
- Transactions;
- Stocks;
- Loans;

A cada Conta podem ser associados Empréstimos, compra e venda de Ações, e Objectivos. Também cada conta pode gere várias carteiras onde separa o total monetário (por exemplo: Dinheiro físico, conta corrente, conta poupança). Cada Conta tem associadas Categorias e Subcategorias que terão cada uma associados um ou mais Orçamentos. As Transações (Despesa, Rendimento ou Transferência) deverão pertencer a uma Categoria e afectam o saldo das Carteiras, das Contas e dos Orçamentos. Da mesma forma, os Empréstimos e Ações afectam o saldo das Contas e Carteiras, assim como o seu património (valor monetário somado ao valor de bens e dívidas).

*ver diagramas 2.1 e 2.2

Figura 2.1: Diagrama de Relação de Entidades

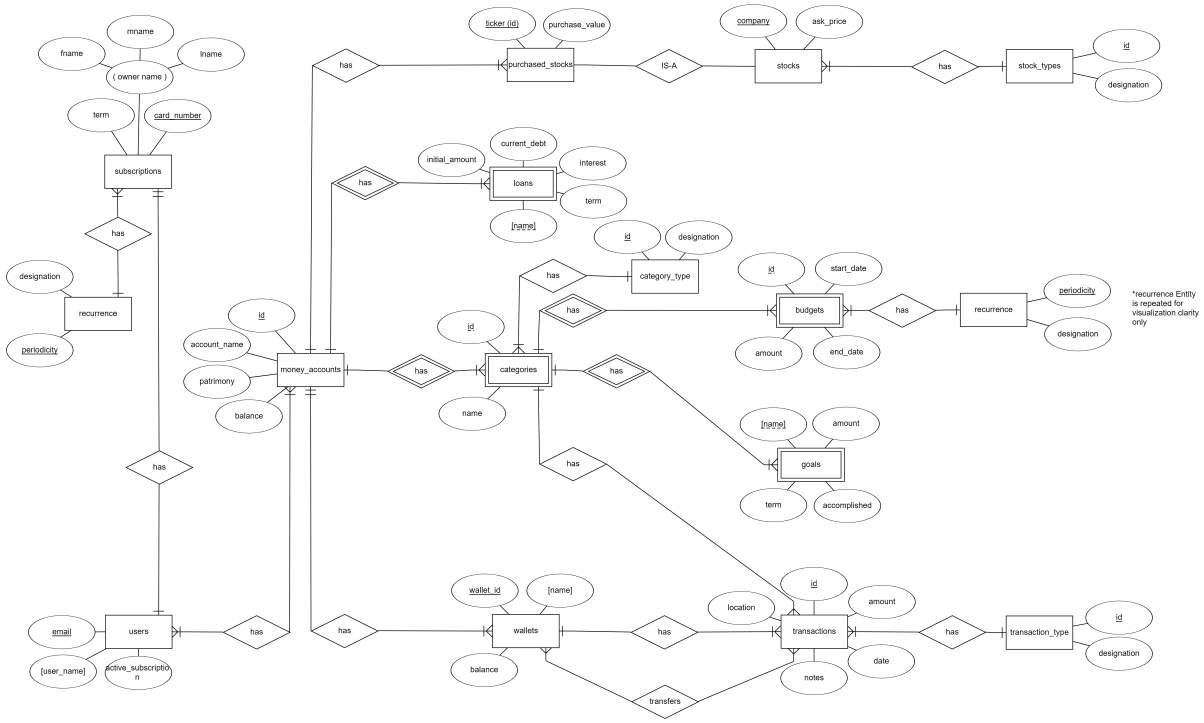
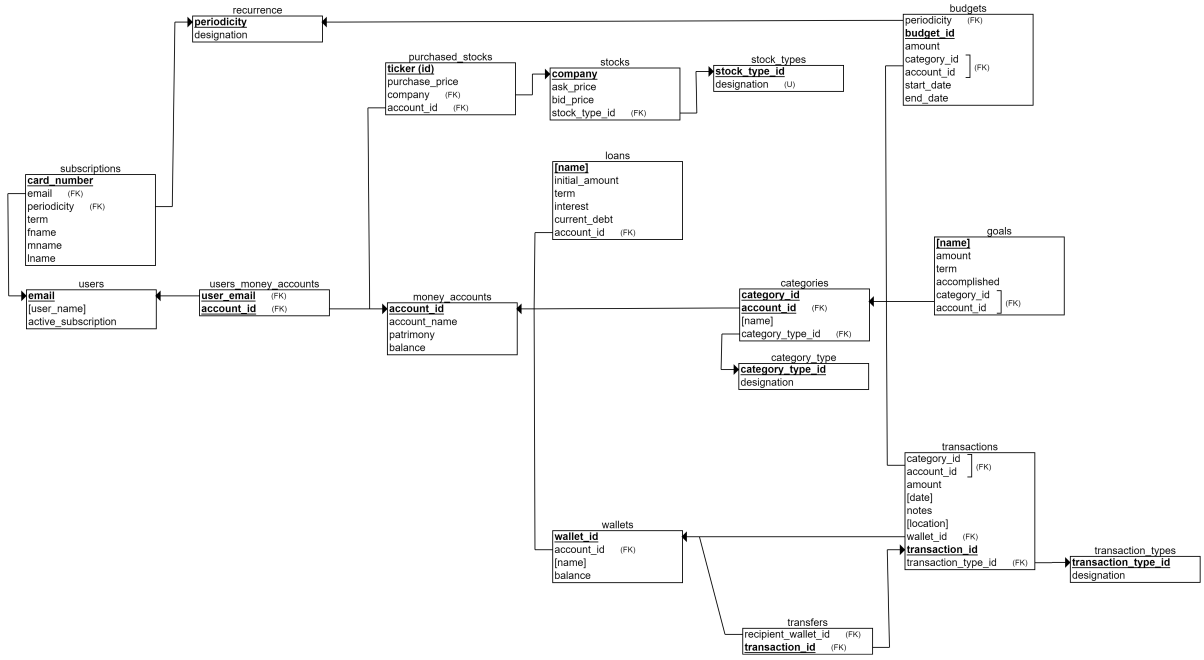


Figura 2.2: Esquema Relacional



2.2 Principais Queries

As maioria das Queries necessárias à aplicação são diretamente relacionadas com SELECT, INSERT, UPDATE e DELETE, comuns a todas as Entidades. A dificuldade é encontrada na própria base de dados, principalmente na Inserção e Eliminação de novos tuplos, uma vez que a maioria das entidades estão ligadas entre si, criando alterações em cascata necessárias para garantir integridade dos dados. De forma geral essas relações são:

- A atualização/eliminação de Utilizadores exige a verificação/actualização/eliminação das subscrições;
- A seleção de utilizadores faz a junção com a subscrição;
- A eliminação de uma Conta de Utilizador implica a eliminação de todos os valores que lhe são associados nas Relações de Budgets, Goals, Loans, Purchased Stocks, Transactions, Wallets, Users Money Accounts e Categories;
- As Categorias e Subcategorias são uma só entidade, distinguidas por uma gama de ID que têm de ser calculados;
- A inserção/eliminação de Empréstimos tem um efeito no saldo das Carteiras que se propaga par ao saldo da Conta de Utilizador, mas mantendo o património inalterado;
- A compra e venda (inserção/eliminação) de Ações tem também efeito no saldo das Carteira e Contas de Utilizador, mas altera o património ds últimas;
- A consulta do valor de Ações está simulado na própria base de dados, exigindo uma reatribuição de um novo valor a cada consulta.
- A inserção de Objectivos gera a inserção de uma nova Carteira, por forma a separar o valor financeiro colocado de parte para o mesmo;
- A inserção/eliminação de Transações afeta o saldo de uma ou mais Carteiras, assim como o saldo e património da Conta de Utilizador. Há ainda uma relação indireta com o Orçamento da Categoria à qual a Transação está afetada.

Capítulo 3

SQL DDL e DML

3.1 Normalização

O desenho das Relações criadas para este projeto seguiu a estratégia aprendida durante as aulas teóricas e treinada nas aulas práticas. Assim, todas as Relações respeitam as seguintes Formas:

- **Primeira Forma Normal - 1NF**
  - Todos os atributos são atómicos. Não existem atributos compostos.
  - Não existem relações dentro de relações
- **Segunda Forma Normal - 2NF**
  - Não existem dependências parciais. Nas Relações Loans, Categories e Goals, onde a chave primária é composta, todos os atributos dependem da totalidade da chave.
- **Terceira Forma Normal - 3NF**
  - Não existem dependências transitivas.
- **Formal Normal de Boyce-Codd - BCNF**
  - Em todas as Relações os atributos são depende apenas da chave primária, de toda a chave a nada mais.
- **Quarta e Quinta Formas Normais - 4NF/5NF**
  - O tipo de situação que necessidade uma destas normalizações é raro e difícil de detectar. Relações que respeitem a BCNF normalmente respeitam também a 4NF e 5NF. Devido à relativa simplicidade do projeto e após análise concluiu-se que as suas Relações respeitam também a 4NF e 5NF.

3.2 Índices

Nos acessos à base de dados pela aplicação de Windows Forms, para a pesquisa e seleção de tuplos, a mesma não é sempre feita usando a chave primária. Será ainda expectável que uma aplicação mais complexa exija mais velocidade de pesquisa usando outros atributos. Neste sentido, foram adicionados Índices a várias Relações. Por exemplo, na seleção de Transações é natural querer pesquisar todas as transações de uma determinada Conta ou de uma determinada Carteira. Deste modo foram adicionados Índices para estes atributos (Relações nde se aplicou: Transactions, Budgets, Goals). O mesmo foi aplicado em Relações em que além de um atributo identificador existe um atributo nome, que é também um atributo bastante usado para a pesquisa e seleção de tuplos (Relações onde se aplicou: Money Accounts, Users, Subscriptions, User Money Accounts, Purchased Stocks). (Figura 3.1)

Figura 3.1: Índices da Relação Transaction

```
CREATE TABLE Project.transactions
(
    transaction_id INT IDENTITY(1,1),
    amount MONEY NOT NULL,
    [date] DATE NOT NULL,
    notes VARCHAR(50),
    [location] VARCHAR(50),
    category_id INT,
    account_id INT,
    transaction_type_id INT,
    wallet_id INT,
    CONSTRAINT PK_TRANSACTIONS PRIMARY KEY (transaction_id),
    CONSTRAINT FK_TRANSACTIONS_CATEGORIES FOREIGN KEY (category_id, account_id) REFERENCES Project.categories(category_id, account_id)
        ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT FK_TRANSACTIONS_TRANSACTIONTYPES FOREIGN KEY (transaction_type_id) REFERENCES Project.transaction_types(transaction_type_id)
        ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT FK_TRANSACTIONS_WALLETS FOREIGN KEY (wallet_id) REFERENCES Project.wallets(wallet_id)
        ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE NONCLUSTERED INDEX IX_TRANSACTIONS_ACCOUNT_ID ON Project.transactions
(account_id) WITH (FILLFACTOR = 75, PAD_INDEX = ON);
CREATE NONCLUSTERED INDEX IX_TRANSACTIONS_CATEGORY_ID ON Project.transactions
(category_id) WITH (FILLFACTOR = 75, PAD_INDEX = ON);
CREATE NONCLUSTERED INDEX IX_TRANSACTIONS_WALLET_ID ON Project.transactions
(wallet_id) WITH (FILLFACTOR = 75, PAD_INDEX = ON);
```

3.3 Stored Procedures (SP) e User Defined Functions (UDF)

A interface da base de dados foi pensada de forma a apenas permitir o acesso aos dados apenas através dos Stored Procedures e User Defined Functions.

3.3.1 User Defined Functions[1]

- udf_annual_statistics (Figura 3.2)
  - Esta função realiza várias operações sobre a Relação Transactions e gera uma Relação completamente nova com as estatísticas de despesas, rendimentos e saldos mensais para um determinado ano.

Figura 3.2: UDF Annual Statistics

```
CREATE FUNCTION udf_annual_statistics (
    @account_id INT,
    @year INT
)
RETURNS @table TABLE (date_year INT, date_month INT, income MONEY, expenses MONEY, balance MONEY)
AS BEGIN

    DECLARE @year_str VARCHAR(4);
    SELECT @year_str=CAST(@year as varchar(10));

    INSERT @table(date_year, date_month, income, expenses, balance)
    SELECT @year, I.mes, inc, expe, (inc-expe)
    FROM (
        (SELECT MONTH([date]) AS mes, SUM(amount) AS inc
        FROM Project.transactions
        WHERE
            transaction_type_id=1 AND YEAR([date])=@year_str AND account_id=@account_id
        GROUP BY MONTH([date])
        ) AS I

        JOIN

        (SELECT MONTH([date]) AS mes, SUM(amount) AS expe
        FROM Project.transactions
        WHERE
            transaction_type_id=-1 AND YEAR([date])=@year_str AND account_id=@account_id
        GROUP BY MONTH([date])
        ) AS E

        ON I.mes=E.mes
    );
    RETURN;
END
GO
```

[1] A preferência foi usar os Stored Procedures apenas por questões de modularidade do lado da aplicação em C#. Ainda assim foi usada uma UDF para uma operação mais complexa de modo a demonstrar o conhecimento adquirido durante as aulas.

3.3.2 Stored Procedures

- **Inserção** - Várias Entidades têm de ser inseridas pelo utilizador da aplicação. Várias inserções necessitam verificações adicionais e geram alterações em mais do que uma Relação. Abaixo indicam-se as alterações feitas além da Relação mencionada no próprio nome do Procedimento: (Figura 3.3.2)

Figura 3.3: Procedimento de Inserção de uma Categoria

```
CREATE PROC pr_insert_category (
    @category_type_id INT,
    @account_id INT,
    @name VARCHAR(20)
) AS
BEGIN

    BEGIN TRANSACTION
    SAVE TRANSACTION insert_category_savepoint

    BEGIN TRY
            -- calculate new category id
            DECLARE @category_id INT;
            SET @category_id = (
                SELECT MAX(category_id)
                FROM Project.categories
                WHERE account_id=@account_id
            );
            SET @category_id = ((@category_id%100)+1)*100;

            INSERT INTO Project.categories (category_id, category_type_id, account_id, [name])
            VALUES (@category_id, @category_type_id, @account_id, @name);
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION insert_category_savepoint
    END CATCH
END
GO
```

  - **pr_insert_user**
    - Insere uma Subscrição se existir.
  - **pr_insert_money_account**
    - Verifica a existência de uma Money Account com o mesmo nome para o mesmo User
    - Insere na Relação User Money Accounts a associação entre utilizador e nova conta
  - **pr_money_account_add_user**
  - **pr_insert_wallet**
    - Verifica a existência de uma Wallet com o mesmo nome para a mesma Money Account
  - **pr_insert_category**
    - Calcula o novo ID para a categoria selecionando o valor máximo de Categories
  - **pr_insert_subcategory**
    - Calcula o novo ID para a subcategoria selecionando valores de Categories
  - **pr_insert_loan**
  - **pr_insert_budget**
  - **pr_insert_goal**
  - **pr_insert_transaction**
    - Insere na Relação Transfers se a transação for uma transferência
  - **pr_insert_purchased_stock**
    - Executa pr_add_balance_to_wallet

- **Atualização** - Os procedimentos de atualização foram pensados por forma a serem o mais genéricos possíveis, e assim permitirem a atualização de um ou mais atributos à escolha em simultâneo. Por ser uma operação de menor valor para a aplicação em Windows Forms, foram apenas criados os procedimentos essenciais que permitiam demonstrar o conhecimento adquirido. Várias atualizações foram usadas para criar *Triggers* Abaixo indicam-se as alterações feitas além da Relação mencionada no próprio nome do Procedimento: (Figura 3.4)

Figura 3.4: Actualização do utilizador

```
-- verify if user is already subscribed
DECLARE @subscribed BIT;
SET @subscribed = (SELECT active_subscription FROM Project.users WHERE email=@email);

-- update users table
UPDATE Project.users...;

-- update subscription table
IF @subscribed=1 AND (@active_subscription=1 OR @active_subscription IS NULL)
BEGIN

    UPDATE Project.subscriptions
    SET
        card_number=ISNULL(@card_number, card_number),
        term=ISNULL(@term, term),
        fname=ISNULL(@fname, fname),
        mname=ISNULL(@mname, mname),
        lname=ISNULL(@lname, lname),
        periodicity=ISNULL(@periodicity, periodicity)
    WHERE email=@email;
END

IF @subscribed=1 AND @active_subscription=0...

IF @subscribed=0 AND @active_subscription=1...
```

  - **pr_update_user**
    - Insere/atualiza/elimina a sua subscrição.
  - **pr_update_stocks_values**
    - Serve apenas como uma falsa API de cotações em bolsa, e atualiza dos preços de todas as ações para um valor aleatório.

- **Eliminação** - Os procedimentos de eliminação são na sua maioria simples e eliminam apenas um tuplo da Relação pretendida. Alguns, no entanto, geram eliminações "em cascata"noutras Relações. Foram apenas criados os procedimentos essenciais ao funcionamento da aplicação Windows Forms. Abaixo indicam-se as alterações feitas além da Relação mencionada no próprio nome do Procedimento: (Figura 3.5)

Figura 3.5: Eliminação de uma conta

```
CREATE PROC pr_delete_money_account (
    @account_id INT
) AS
BEGIN

    BEGIN TRANSACTION
    SAVE TRANSACTION delete_money_account_savepoint
    BEGIN TRY

            DELETE FROM Project.budgets...;

            DELETE FROM Project.goals...;

            DELETE FROM Project.loans...;

            DELETE FROM Project.purchased_stocks...;

            DELETE FROM Project.transactions...;

            DELETE FROM Project.wallets...;

            DELETE FROM Project.users_money_accounts...;

            DELETE FROM Project.categories...;

            DELETE FROM Project.money_accounts...;
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION delete_money_account_savepoint
    END CATCH
END
GO
```

  - **pr_delete_subscription**
  - **pr_delete_user**

    - Executa pr_delete_subscription se existir subscrição
  - **pr_delete_money_account**
    - Elimina da Relação Budgets se existirem orçamentos associados
    - Elimina da Relação Goals se existirem objetivos associados
    - Elimina da Relação Loans se existirem empréstimos associados
    - Elimina da Relação Purchased Stocks se existirem activos associados
    - Elimina da Relação Transactions se existirem transações associadas
    - Elimina da Relação Wallets se existirem carteiras associadas
    - Elimina da Relação Categories se existirem categorias associadas
    - Elimina da Relação Users Money Accounts os utilizadores associados
  - **pr_money_account_remove_user**
  - **pr_delete_category**
  - **pr_delete_transaction**
  - **pr_delete_purchased_stocks**
    - Executa pr_add_balance_to_wallet

- **Seleção** - Os procedimentos de seleção, uma vez que são os mais usados, foram feito por forma a serem genéricos (figura 3.6), assim é possível filtrar da base de dados vários tuplos de uma Entidade utilizando vários argumentos. Abaixo indicam-se as alterações feitas além da Relação mencionada no próprio nome do Procedimento: (Figura 3.6)

Figura 3.6: Procedimento de seleção de orçamento

```
CREATE PROC pr_select_budgets (
    @account_id INT = NULL,
    @category_id INT = NULL,
    @budget_id INT = NULL,
    @amount MONEY = NULL,
    @periodicity INT = NULL,
    @start_date DATE = NULL,
    @end_date DATE = NULL
) AS
BEGIN

    SELECT *
    FROM Project.budgets
    WHERE
            (account_id=@account_id OR @account_id IS NULL)
        AND (category_id=@category_id OR @category_id IS NULL)
        AND (budget_id=@budget_id OR @budget_id IS NULL)
        AND (amount=@amount OR @amount IS NULL)
        AND (periodicity=@periodicity OR @periodicity IS NULL)
        AND ([start_date]=@start_date OR @start_date IS NULL)
        AND (end_date=@end_date OR @end_date IS NULL)
END
GO
```

  - **pr_select_users**
    - Junta um ou mais utilizadores à sua subscrição
  - **pr_select_recurrences**
  - **pr_select_recurrence_id**
  - **pr_select_money_accounts**
    - Permite a seleção de várias Contas filtrando por valores mínimo e máximo de saldo e/ou património
  - **pr_select_user_money_accounts**
    - Seleciona todas as contas de um dado utilizador, juntando a sua informação a cada uma
  - **pr_select_wallets**
  - **pr_select_categories**
  - **pr_select_category_types**

        * Seleciona tipos de categorias usando a sua designação ou o seu ID. Se nenhum for fornecido, devolve todos os tipos disponíveis.
    - **pr_select_loans**
    - **pr_select_budgets**
    - **pr_select_budgets_by_category_id**
    - **pr_select_goals**
    - **pr_select_transactions**
        * Se for dado um ID de uma categorias devolve todas as transações dessa categoria e de todas as suas subcategorias. Se for dado o ID de uma subcategoria devolve apenas transações dessa subcategoria.
    - **pr_select_transaction_types**
        * Seleciona tipos de transações usando a sua designação ou o seu ID. Se nenhum for fornecido, devolve todos os tipos disponíveis.
    - **pr_select_purchased_stocks**
        * Junta a informação das ações compradas com a a informação das ações do mercado
    - **pr_select_stocks**
        * Executa a atualização de preços das ações do mercado (pr_update_stocks_values). Junta a informação das ações com o seu tipo.

- **Outros** - Mais alguns procedimentos foram necessários tanto para fornecer funcionalidades adicionais à aplicação como para modularizar algumas operações e evitar repetição de código na própria base de dados:
    - **pr_exists_user**
    - **pr_exists_money_account**
    - **pr_recalculate_patrimony**
        * Usada para somar os saldos e bens de uma conta. Normalmente chamada dentro de Triggers.
    - **pr_add_balance_to_wallet**
        * Usada para atualizar o saldo de uma carteira ao inserir/eliminar transações, empréstimos e ações.
    - **pr_exists_loan**
    - **pr_loan_payment**

3.3.3 Triggers

Devido à forte ligação entre as várias Entidades ao nível, são necessários várias verificações e atualizações de valores na introdução ou eliminação de tuplos. Deste modo, vários Triggers foram necessários: (Figura 3.7)

Figura 3.7: Trigger de actualização de saldo de duas carteiras na inserção de uma transferência

```
CREATE TRIGGER tr_update_wallet_balance_on_transfer_insert
ON Project.transfers
AFTER INSERT
AS
BEGIN

    DECLARE @transaction_id INT;
    DECLARE @recipient_wallet_id INT;
    SELECT
        @transaction_id=transaction_id,
        @recipient_wallet_id=recipient_wallet_id
    FROM INSERTED;

    DECLARE @value MONEY;
    SELECT @value=amount
    FROM Project.transactions
    WHERE transaction_id=@transaction_id;

    UPDATE Project.wallets
    SET balance = balance + COALESCE(@value, 0)
    WHERE wallet_id=@recipient_wallet_id;

    DECLARE @from_wallet_id INT;
    SELECT
        @from_wallet_id=wallet_id,
        @value=amount
    FROM Project.transactions
    WHERE transaction_id=@transaction_id;

    UPDATE Project.wallets
    SET balance = balance - COALESCE(@value, 0)
    WHERE wallet_id=@from_wallet_id;
END
GO
```

- **tr_insert_base_wallets_on_new_money_account_insert**
  - A aplicação exige que algumas carteiras estejam disponíveis para o utilizador logo no arranque. Assim, com a criação de cada conta são-lhe imediatamente associadas várias carteiras
- **tr_insert_base_categories_on_new_money_account_insert**
  - No arranque da aplicação são fornecidas algumas categorias de despesas gerais para permitir ao utilizador o seu uso imediato. Assim, com a criação de cada conta são-lhe imediatamente associadas várias categorias
- **tr_insert_new_wallet_on_goal_insert**
  - Quando um utilizador define um objetivo de poupança, necessita ter uma forma de distinguir o dinheiro que põe de parte. Assim, ao criar um novo objetivo é inserida na sua conta uma nova carteira com o mesmo nome do objetivo
- **tr_update_goal_completion_with_wallet_transfer**
  - Quando o utilizador transfere uma quantia de dinheiro para a sua carteira associada a um objetivo, é necessário atualizar o valor de completude do mesmo
- **tr_update_account_patrimony_on_wallet_balance_update**
  - Sempre que uma carteira de uma dada conta atualiza o seu saldo, o património dessa mesma conta tem de ser atualizado
- **tr_update_wallet_balance_on_loans_insert**
  - Sempre que um empréstimo é contraído numa dada conta, o salda da sua carteira principal tem de ser atualizado
- **tr_update_account_balance_on_wallet_balance_update**
  - Sempre que o saldo de uma carteira é actualizado, o saldo da conta a qual está associada essa carteira tem de ser atualizado

- **tr_update_wallet_balance_on_transaction_insert**
  - Sempre que uma transação é feita, o saldo da carteira usada tem de ser atualizado
- **tr_update_wallet_balance_on_transaction_delete**
  - Sempre que uma transação apagada, o saldo da carteira usada tem de ser atualizado
- **tr_update_wallet_balance_on_transfer_insert**
  - Sempre que uma transferencia é feita, o saldo das carteiras usadas tem de ser atualizado
- **tr_insert_new_transaction_on_loan_payment_update**
  - Sempre que o pagamento de um empréstimo é feito, uma transação tem de ser criada por forma a manter a integridade do saldo da conta.

Fluxo de execução de Triggers

- pr_insert_money_account
  - tr_insert_base_wallets_on_new_money_account_insert
  - tr_insert_base_categories_on_new_money_account_insert
- pr_insert_transaction
  - tr_update_goal_completion_with_wallet_transfer (*apenas se a transação for uma transferência)
  - tr_update_wallet_balance_on_transaction_insert
    - tr_update_account_balance_on_wallet_balance_update
    - tr_update_account_patrimony_on_wallet_balance_update
      - pr_recalculate_patrimony
  - tr_update_wallet_balance_on_transfer_insert
    - tr_update_account_balance_on_wallet_balance_update
    - tr_update_account_patrimony_on_wallet_balance_update
      - pr_recalculate_patrimony
- pr_delete_transaction
  - tr_update_wallet_balance_on_transaction_delete
    - tr_update_account_balance_on_wallet_balance_update
    - tr_update_account_patrimony_on_wallet_balance_update
      - pr_recalculate_patrimony
- pr_insert_goal
  - tr_insert_new_wallet_on_goal_insert
- pr_insert_loan
  - tr_update_wallet_balance_on_loans_insert
    - pr_add_balance_to_wallet
      - tr_update_account_balance_on_wallet_balance_update
      - tr_update_account_patrimony_on_wallet_balance_update
      - pr_recalculate_patrimony
- pr_loan_payment
  - tr_insert_new_transaction_on_loan_payment_update
    - pr_insert_transaction

            - *... fluxo descrito acima ...*

- pr_insert_purchased_stock
    - pr_add_balance_to_wallet
        - *... fluxo descrito acima ...*

Capítulo 4

Aplicação Windows Forms (C#)

A aplicação em Windows Forms foi desenha com o objetivo de demonstrar as capacidades de interação com a base de dados e os mecanismos que a mesma usa para manter a integridade de todos os valores. Não representa uma aplicação a ser usada em mundo real. A aplicação inicia com a possibilidade de demonstrar ser usada por dois tipos de utilizador. O BackOffice onde é possível fazer a gestão de utilizadores e as suas subscrições, e a App que demonstra as possibilidades de utilização por um utilizador final.

Relativamente à implementação em C# da aplicação foram projetados 3 blocos principais com o intuito de modularizar o código e garantir que nenhuma interface tivesse de aceder ou conhecer implementação de estrturas que não lhe dissessem respeito. Os 3 blocos são:

- Várias Classes Windows Forms e de Entidades
- DB_API
- DB_IO

4.1 DB_IO - Conexão e Pedidos à Base de Dados

Nesta classe foram incluídos todos os métodos de conexão e interação com a base de dados. A sua implementação foi pensada por forma a fornecer uma interface o mais genérica possível para os métodos de execução de *queries* mas garantindo segurança a nível da injeção de código SQL. Esta interface permite a um utilizador externo fornecer apenas o nome do Procedimento que pretende usar e os parâmetros necessários (num dicionário) e chamar os métodos Insert, Delete, Update, SelectScalar e SelectReader sem ter de conhecer a implementação interna necessária para interagir ou conectar com a Base de Dados. (Figura 4.1)

As seguintes subsecções apresentam a forma como esta modularidade foi conseguida

4.1.1 SqlCommand and Parameterizer

Para garantir segurança contra injeção de código em SQL é sempre usada a classe *SqlCommand*. Esta permite a atribuição verificada de parâmetros. É sobre as instâncias desta clase que são aplicados os métodos de execução de *queries*.

4.1.2 ExecuteNonQuery, ExecuteReader, ExecuteScalar

Estes são os métodos que permitem a interação com a Base de Dados. Uma vez que se utilizaram comandos e não SQL dinâmico, foram criados os métodos ExecuteProcNonQuery, ExecuteProcReader e ExecuteProcScalar que encapsulam em si a criação da instância SqlCommand, a sua parametrização e a sua execução.

4.1.3 Insert, Delete, Update, SelectScalar, SelectReader

Por fim, estes são os métodos públicos fornecidos por esta interface, juntamente com um enumerado que contém todos os nomes dos procedimentos e funções fornecidas pela Base de Dados. Assim, em suma,

um cliente desta interface apenas terá de saber que Procedimento/Função quer executar e passar os parametros correspondentes.

4.1.4 DBconnect, DBdisconnect

Para a comunicação efetiva com a BAse de Dados é necessário estabelecer (e terminar) conexão a cada pedido. A conexão é sempre estabelecida nos métodos de criação e execução de comandos (e.g. Execute-ProcNonQuery) e é sempre terminada após a execução dos comandos independentemente do seu sucesso ou resultado.

Figura 4.1: Pilha de chamadas de funções para leitura na classe DB_IO

```
public static int Insert(System.Enum proc, IDictionary<System.Enum, Object> attrValue)
{
    return ExecProcNonQuery(proc.ToString(), attrValue);
}

private static int ExecProcNonQuery(String procName, IDictionary<System.Enum, Object> attrValue)
{
    SqlConnection cnx = DBconnect();

    // step 1: create stored procedure command (cmd)
    SqlCommand cmd = new SqlCommand(procName, cnx)
    {
        CommandType = CommandType.StoredProcedure
    };

    // step 2: parameterize cmd
    CmdParameterizer(cmd, attrValue);

    // step 3: execute
    return ExecuteNonQuery(cnx, cmd);
}

private static void CmdParameterizer(SqlCommand cmd, IDictionary<System.Enum, Object> attrValue)
{
    foreach (KeyValuePair<System.Enum, Object> entry in attrValue)
    {
        cmd.Parameters.AddWithValue("@" + entry.Key, entry.Value);
    }
}

private static int ExecuteNonQuery(SqlConnection cnx, SqlCommand cmd)
{
    int rows = 0;
    try
    {
        rows = cmd.ExecuteNonQuery();
    }
    catch (SqlException ex)
    {
        ErrorMessenger.Exception(ex);
    }
    finally
    {
        DBdisconnect(cnx);
    }
    return rows;
}
```

4.2 DB_API - Interface para Aplicações

Para a criação de um aplicação é natural que se procure uma interface com métodos específicos e os mais atómicos possíveis. Disponibilizar na própria Base de Dados métodos para qualquer pedido possível seria irrealista. E uma interface de leitura genérica não é o mais adequado. Assim, implementou-se a Interface DB_API que fornece uma camada extra de encapsulamento e métodos especializados para o uso dos desenvolvedores da aplicação. Estes não necessitam agora de ter qualquer conhecimento sobre a Base de Dados, sua implementação e seus métodos de acesso, pois estes são resolvidos por esta classe. Esta classe recebe então argumentos num formato tradicional de C# e pede à DB_IO para fazer uma conexão à Base de Dados indicando-lhe o o Procedimento/Função que pretende usar e os parâmetros necessários. (Figura 4.2)

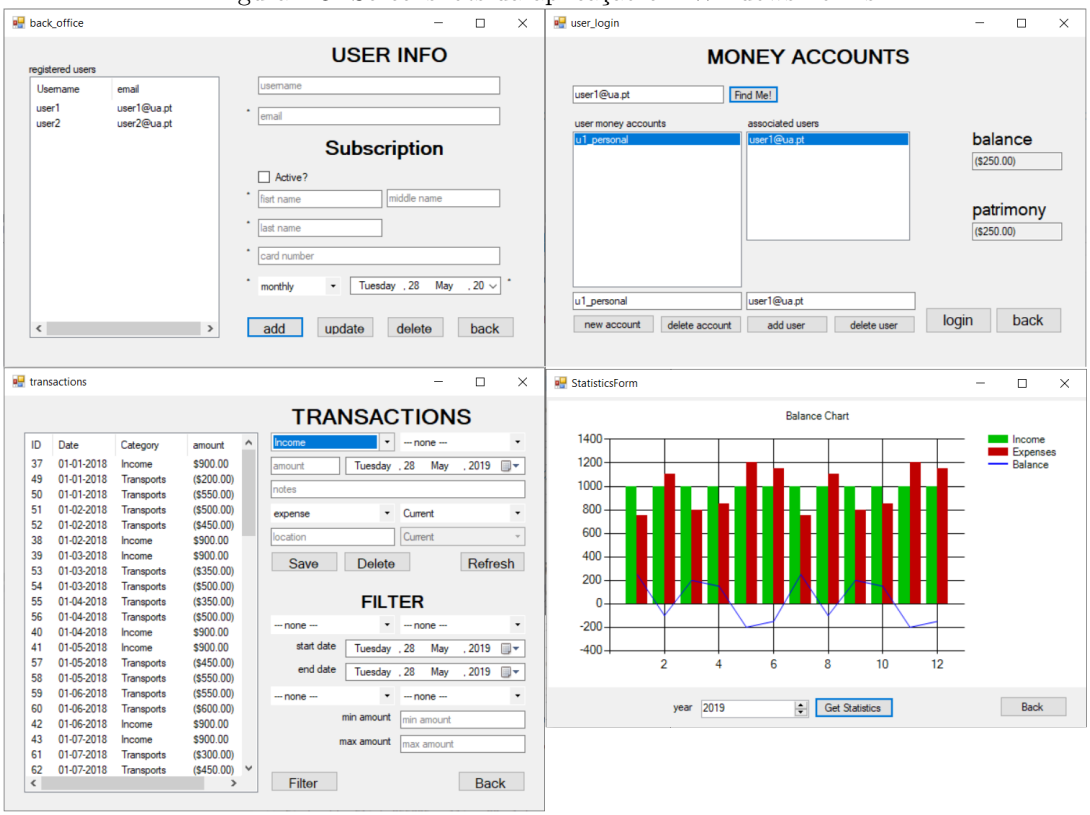
Figura 4.2: Exemplo de assinatura dos métodos da classe DB_API

```
public static bool ExistsUser(string email)...

public static void InsertUser(string username, string email, string fname, string mname, string lname,
    string cardNo, int periodicity, DateTime term, bool active)...

public static void UpdateUser(string username, string email, string fname = null, string mname = null,
    string lname = null, string cardNo = null, int periodicity = 1, DateTime term = new DateTime(), bool? active = null)...

public static void DeleteUser(string email)...

public static DataTableReader SelectUserByEmail(string email)...

public static DataTableReader SelectAllUsers()...
```

4.3 Classes Windows Forms, Classes de Entidades - A aplicação

Finalmente, a cada Windows Form tem associada a sua própria classe para interação com os elementos visuais. Para as várias entidades foram criadas classes correspondentes em C# para as situações em que a sua leitura repetida era desnecessária e fosse mais útil guarda a informação lida da Base de Dados em instâncias de Objectos (a cada momento são guardadas apenas as informações mínimas essenciais ao funcionamento da aplicação). (Figura 4.3)

Figura 4.3: Screenshots da aplicação em Windows Forms



Capítulo 5

Conclusão

O maior desafio deste projecto, resultou da complexidade de interligação entre Entidades o que exigiu uma maior atenção na criação de Triggers e nas chamadas encadeadas de Procedimentos. Além disso, o desenvolvimento em C# de uma aplicação Windows Forms também teve os seus momentos de aprendizagem e pesquisa que acresceram esforço ao projeto. A criação de um sistema que permitisse realizar todas as operações possíveis para uma aplicação de Controlo de Finanças Pessoais com todas as Entidades desenvolvidas era no mínimo ambiciosa para as condicionantes do projeto. No a tentativa de sintetizar o essencial da aplicação, garantindo uma boa usabilidade mostrou-se uma ferramenta de aprendizagem valiosa.

Contribuições dos autores

Percentagem de contribuição por autor:

- Luís Moura: 50%
- Maria Lavoura: 50%