UFMG

TRABALHO PRÁTICO 1

SISTEMA DE CONTROLE BANCÁRIO

Grupo: Arthur de Lapertosa Lisboa - 2018115469
Lara Souza Ramos - 2017050070
Rafaela Milagres Moreira - 2018013640
Thayan Castro de Lima - 2017050096
Pedro Emanuel Lemos da Silveira - 2018013828

Disciplina: Programação Orientada a Objetos
Professor: Cristiano Leite de Castro

Introdução

Este documento visa orientar e facilitar a análise do primeiro trabalho prático da disciplina Programação Orientada ao Objeto que consiste em programar um Sistema de Controle Bancário. Para tanto, foram criadas cinco classes de objetos: Classe Cliente, Classe Movimentação, Classe Conta, Classe Banco e Classe Interface. Cada uma dessas classes têm seus atributos privados e seus métodos especificados no desenvolvimento deste documento, onde encontram-se instruções de como compilar e executar o código e como foi feita a implementação deste.

Como compilar e executar

Para a realização do trabalho, utilizamos a ferramenta GitKraken que permite a conexão com a conta do github e o compartilhamento do código entre os integrantes do grupo, a fim de que seja possível fazer alterações no código sem que se prejudique o código original. Como IDE, o grupo utilizou a ferramenta Eclipse Photon e Visual Studio e para compilar o programa utilizamos o compilador g++ e o sistema operacional utilizado para o projeto foi o Linux.

Implementação do código

Header file

Banco.h

```
class Banco {
private:
  std::string nomeBanco;
  std::list<Cliente> clientes;
  std::vector<Conta> contas;
```

Na classe banco, foram utilizados como atributos privados, o nome do banco, a lista de clientes e o vetor de contas.

```
public:
  //Cria um banco com o nome "nomeB"
  Banco();
  Banco(std::string nomeB);

  //Adiciona um nome cliente
  void adicionaCliente(Cliente& novo);

  //Cria uma nova conta retorna true se criado e false se não criado
  bool criarConta(std::string cpf);

  //Checa se o cliente tem alguma conta
  bool possuiConta(std::string cpf);

  //Deleta um cliente PRECONDI��O: Cliente n�o possui conta
  bool deletaCliente(std::string cpf);

  //Printa a lista de clientes
  void printClientes();

  //Printa a lista de contas
  void printContas();

  //Exclui uma conta
  void deleteConta(int numConta);

  //Faz um deposito em uma conta
  void depositoConta(int numConta, double valor);

  //Faz um saque em uma conta
  bool saqueConta(int numConta, double valor);

  //Faz uma transferência entre contas
  bool transferencia(int numContaOrigem, int numContaDestino, double valor);

  void cobrarTarifa();

  void cobrarCPMF();

  //Retorna o saldo de uma conta
  double obterSaldo (int numConta);
```

```
//Retorna o extrato do último mês de uma conta
std::vector<Movimentacao> obterExtratoMesAtual(int numConta);

//Retorna o extrato de uma conta desde a data inicial
std::vector<Movimentacao> obterExtrato(std::string dataIni, int numConta);

//Retorna o extrato de uma conta no período especificado
std::vector<Movimentacao> obterExtrato(std::string dataIni, std::string dataFim, int numConta);

//Obtem a lista de clientes
std::list<Cliente> clientesLista();

//Obtem a lista de contas
std::vector<Conta> contasLista();

//Escreve os dados no arquivo
void writeFile();

//Lê os dados no arquivo
void readFile();

//Write the member variables to stream objects
friend std::ostream& operator << (std::ostream& out, const Banco& obj);

//Read data from stream object and fill it in member variables
friend std::istream& operator >> (std::istream& in, Banco& obj);
```

Na parte pública foram colocados os métodos que fazem movimentações e mudanças nas contas e alterações na lista de clientes. Utilizamos também o atributo friend que declara que a função fora da classe é amiga e, com isso, podemos acessar os atributos privados da classe. Esse atributo também foi inserido nas classes Movimentação, Conta e Cliente.

Movimentacao.h

```
class Movimentacao {
private:
  time_t dataMov;
  std::string descricao;
  char debitoCredito;
  double valor;

public:
  Movimentacao();
  Movimentacao(std::string descricao, char debitoCredito, double valor);
  Movimentacao(time_t dataMov, std::string descricao, char debitoCredito, double valor);
  virtual ~Movimentacao();
  //Movimentacao(const Movimentacao &other);

  time_t getDataMov() const;
  char getDebitoCredito() const;
  const std::string& getDescricao() const;
  double getValor() const;

  //Write the member variables to stream objects
  friend std::ostream& operator << (std::ostream& out, const Movimentacao& obj);

  //Read data from stream object and fill it in member variables
  friend std::istream& operator >> (std::istream& in, Movimentacao& obj);
```

Em movimentação, utilizamos os atributos de data, débito ou crédito, descrição e valor como privados e utilizamos o tipo aritmético time_t para a data. Além disso, foram usados getters para obter os valores dos atributos.

Cliente.h

```
class Cliente {
private:
  std::string nomeCliente;
  std::string cpf_cnpj;
  std::string endereco;
  std::string fone;
public:
  //constructors
  Cliente(std::string nomeCliente, std::string cpf_cnpj, std::string endereco, std::string fone);
  Cliente();
  virtual ~Cliente();
  //Cliente(const Cliente &other);

  //getters and setters
  const std::string& getCpfCnpj() const;
  void setCpfCnpj(const std::string &cpfCnpj);
  const std::string& getEndereco() const;
  void setEndereco(const std::string &endereco);
  const std::string& getFone() const;
  void setFone(const std::string &fone);
  const std::string& getNomeCliente() const;
  void setNomeCliente(const std::string &nomeCliente);

  //Write the member variables to stream objects
  friend std::ostream& operator << (std::ostream& out, const Cliente& obj);

  //Read data from stream object and fill it in member variables
  friend std::istream& operator >> (std::istream& in, Cliente& obj);
```

Na classe cliente, utilizamos como atributo privado o nome do cliente, cpf ou cnpj, endereço e telefone. Em público, utilizamos métodos getters e setters para atribuir e obter os valores dos atributos.

Conta.h

```
class Conta {
private:
  int numConta;
  double saldo;
  Cliente *cliente;
  std::vector<Movimentacao> movimentacoes;
  static int proximoNumConta;
public:
  //constructors
  Conta();

  Conta(Cliente *clienteNovo);

  virtual ~Conta();
  //Conta(const Conta &other);

  //getters and setters
  const Cliente& getCliente() const;
  const std::vector<Movimentacao> getMovimentacoes();
  int getNumConta() const;
  //static int getProximoNumConta();
  double getSaldo() const;

  //Seta o ponteiro pro cliente
  void setCliente(Cliente *newCliente);

  //methods
  bool debitar(double valor, std::string descricao);
  void creditar(double valor, std::string descricao);
  vector<Movimentacao> obterExtrato(std::string dataIni, std::string dataFim);
  vector<Movimentacao> obterExtrato(std::string dataIni);
  vector<Movimentacao> obterExtratoMesAtual();

  //Write the member variables to stream objects
  friend std::ostream& operator << (std::ostream& out, const Conta& obj);

  //Read data from stream object and fill it in member variables
  friend std::istream& operator >> (std::istream& in, Conta& obj);
};
```

Na classe conta, utilizamos como atributos privados o número da conta, saldo, um objeto da classe Cliente um vetor da classe Movimentação e um inteiro estático do próximo número da conta. Este inteiro estático significa que para todo os objetos desta classe o valor deste inteiro será exatamente o mesmo para todos os objetos. Utilizamos getters para retornar um cliente, um vetor de movimentações,

um número de conta e o saldo da conta. Foram, ainda, criados métodos públicos para debitar, creditar e obter extratos das contas, além de uma função para setar o ponteiro para o cliente.

Interface.h

```
class Interface {
private:
  Banco banco;
public:
  Interface(Banco &banco);
  Banco getBanco();
  void menu();
  void printarContas();
  void printarClientes();
  void printarExtrato(vector<Movimentacao> extrato);
  void cadastrarCliente();
  void criarConta();
  void excluirCliente();
  void excluirConta();
  void depositar();
  void sacar();
  void transferir();
  void cobrarTarifa();
  void cobrarCPMF();
  void obterSaldo();
  void obterExtrato();
  virtual ~Interface();
};
```

Na classe interface, foi incluído um objeto banco da classe Banco como privado e na parte pública foi incluído a função menu e as funções que serão as opções do menu. Note que o programa só salva em arquivo, quando é digitada a opção 14 do menu. Essa opção sai do programa e salva todos os dados do banco em arquivo. Quando o programa é iniciado e entra no menu, ele lê automaticamente do arquivo “banco.txt” todos os dados salvos anteriormente.

Construtores

Em todos as classes, foram criados construtores dos objetos correspondentes.

- Banco.cpp:
  - Construtor:
    O

```
Banco::Banco() {

}

Banco::Banco(std::string nomeB) {
        nomeBanco = nomeB;
}
```

  - O Construtor de Cópia e o Destrutor utilizados foram os defaults.

- Cliente.cpp:
  - Construtor:

```
Cliente::Cliente(std::string nomeCliente, std::string cpf_cnpj, std::string endereco, std::string fone) {
        this->nomeCliente = nomeCliente;
        this->cpf_cnpj = cpf_cnpj;
        this->endereco = endereco;
        this->fone = fone;
}

Cliente::Cliente() {

}
```

  - O Construtor de Cópia e o Destrutor utilizados foram os defaults.

- Conta.cpp:
  - Construtor:

```
Conta::Conta() {
        this->saldo = 0.0;
        this->numConta = Conta::proximoNumConta;
        Conta::proximoNumConta++;
        //this->cliente = nullptr;
}

Conta::Conta(Cliente * clienteNovo) {
        this->saldo = 0.0;
        this->cliente = clienteNovo;
        this->numConta = Conta::proximoNumConta;
        Conta::proximoNumConta++;


}
```

    - Construtor de cópia e o destrutor utilizados foram os defaults.

- Interface.cpp:
    - Construtor:

```
Interface::Interface(Banco &banco) {
        this->banco = banco;
}
```

    - Construtor de cópia e o destrutor utilizados foram os defaults.

- Movimentacao.cpp
    - Construtor:

```
Movimentacao::Movimentacao()
{
        descricao = "";
        debitoCredito = 'n';
        valor = 0;
}

Movimentacao::Movimentacao(std::string descricao, char debitoCredito, double valor) {
        this->descricao = descricao;
        this->debitoCredito = debitoCredito;
        this->valor = valor;
        this->dataMov = time(0);

}

Movimentacao::Movimentacao(time_t dataMov, std::string descricao, char debitoCredito, double valor) {
        this->descricao = descricao;
        this->debitoCredito = debitoCredito;
        this->valor = valor;
        this->dataMov = dataMov;

}
```

- O Construtor de Cópia e o Destrutor utilizados foram os defaults.

Sobrecarga de operadores

Durante a criação do código, dois operadores foram sobrecarregados, a fim de que seja possível imprimir e copiar valores no arquivo da classe Banco e em todas as outras classes. Deste modo é possível imprimir e ler no arquivo todas as informações e atributos de todos os objetos que estão na classe "Banco", incluindo a si mesmo. Esses operadores são “>>” e “<<".

```cpp
std::ostream& operator<<(std::ostream& out, const Banco& obj)
{
	// Primeiro escreve o nome do banco
	out << obj.nomeBanco << "\n";

	//Começa escreve a lista de clientes
	out << obj.clientes.size() << "\n";
	for (auto i : obj.clientes) {
		out << i << "\n";
	}

	//Escreve o vetor de contas
	out << obj.contas.size() << "\n";
	for (long unsigned int i = 0; i < obj.contas.size(); i++) {
		out << obj.contas[i] << "\n";
	}
	std::cout << std::endl;

	return out;
}


std::istream& operator>>(std::istream& in, Banco& obj)
{
	//Primeiro pega o nome do banco
	in >> obj.nomeBanco;

	//Pega os clientes
	int numClientes;
	in >> numClientes;
	for (int i = 0; i < numClientes; i++) {
		Cliente* novo = new Cliente;
		in >> *novo;
		obj.clientes.push_back(*novo);
		delete novo;
	}

	//Pega as contas
	int numContas;
	in >> numContas;
	for (int i = 0; i < numContas; i++) {
		Conta* nova = new Conta;
		in >> *nova;
		std::string cpf;
		cpf = nova->getCliente().getCpfCnpj();
```

```
            //Acha o endereço do cliente associado para colocar na conta corretamente
            for (auto& i : obj.clientes) {
                    if (i.getCpfCnpj() == cpf) {
                            Cliente* novo = &i;
                            nova->setCliente(novo);
                    }
            }

            obj.contas.push_back(*nova);
            delete nova;
    }

}
```

Getters e Setters básicos

Utilizamos getters nas classes Cliente, Movimentação e Conta com o objetivo de pegar as informações dos objetos das classes que estão privados de forma segura e os setters para atribuir valores a esses objetos. É possível ver na classe Cliente abaixo a utilização dos getters e dos setters:

```
const std::string& Cliente::getCpfCnpj() const {
  return cpf_cnpj;
}

void Cliente::setCpfCnpj(const std::string &cpfCnpj) {
  cpf_cnpj = cpfCnpj;
}

const std::string& Cliente::getEndereco() const {
  return endereco;
}

void Cliente::setEndereco(const std::string &endereco) {
  this->endereco = endereco;
}
```

Nela, utilizamos esses métodos para atribuir valor e pegar esse valor do atributo endereço e cpfCnpj.

Tratamento de exceções

```
std::string stropcao = "0";
int opcao = 0;

while (opcao != 1 && opcao != 2 && opcao != 3 && opcao != 4 && opcao != 5 && opcao != 6 && opcao != 7
    && opcao != 8 && opcao != 9 && opcao != 10 && opcao != 11 && opcao != 12 && opcao != 13 && opcao != 14){
  std::cout << "Escolha uma opção válida: ";

  getline(cin, stropcao);
  try {
    opcao = stoi(stropcao);
  } catch (...) {
    opcao = 0;
  }

}
```

Para o tratamento de exceções, utilizamos na classe Interface as instruções “try” e “catch”. Como é possível ver acima, recebemos a entrada de quem está utilizando o programa e caso ela não corresponda à alguma opção do Menu, o catch irá obter a exceção.

Main

```
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
#include <ctime>
#include "Interface.h"
#include "Banco.h"
#include "Cliente.h"
#include "Movimentacao.h"
#include "Conta.h"

using namespace std;

int main(int argc, char* argv[]){

  Banco *b = new Banco("Inter");
  Interface *i = new Interface(*b);

  i->menu();


  return 0;
}
```

No main, foi incluído todas as bibliotecas utilizadas no trabalho e posteriormente criamos um novo objeto da classe banco e interface. Depois chamamos a função menu da classe interface para poder abrir o menu e conseguir selecionar opções do programa.

Conclusão

Concluímos que esse trabalho prático se mostrou importante para aplicar conceitos de programação, como a passagem de parâmetro por referência e praticar o padrão de desenvolvimento de código orientado a objetos. A interação entre classes foi um obstáculo enfrentado durante o desenvolvimento do projeto. Por fim, acreditamos que o trabalho nos possibilitou um aprendizado significativo em programação orientada ao objeto.