Object Oriented Programming

Project - Bank

Preliminary Project

Author: Kacper Wojakowski

kwojakow

293064

TABLE OF CONTENTS

1. PROJECT OVERVIEW

1.1 Introduction

The aim of the project is to develop an application representing a database of operating banks. The program consists of 10 classes in total, all of them briefly described in *Part 1.2* and further explained in *Chapter 3*.

1.2 Classes overview

The program consists of 7 main classes:

- **Address:** Simple class storing basic address information, used by classes *branch* and *person*
- **Bank:** A class describing a bank institution, consisting of many *branches*
- **Branch:** A nested class of *bank,* describing a branch of a *bank*, consisting of *accounts, employees* and *clients*
- **Person:** A class describing a person, from which *client* and *employee* inherit
- **Client:** A class describing a client of a *bank*, storing a list of *accounts*
- **Employee:** A class describing an employee of a *bank*, storing a list of *positions*
- **Account:** A class storing a balance and recent transactions

It also uses a template container **List**, which is implemented as a double linked list.

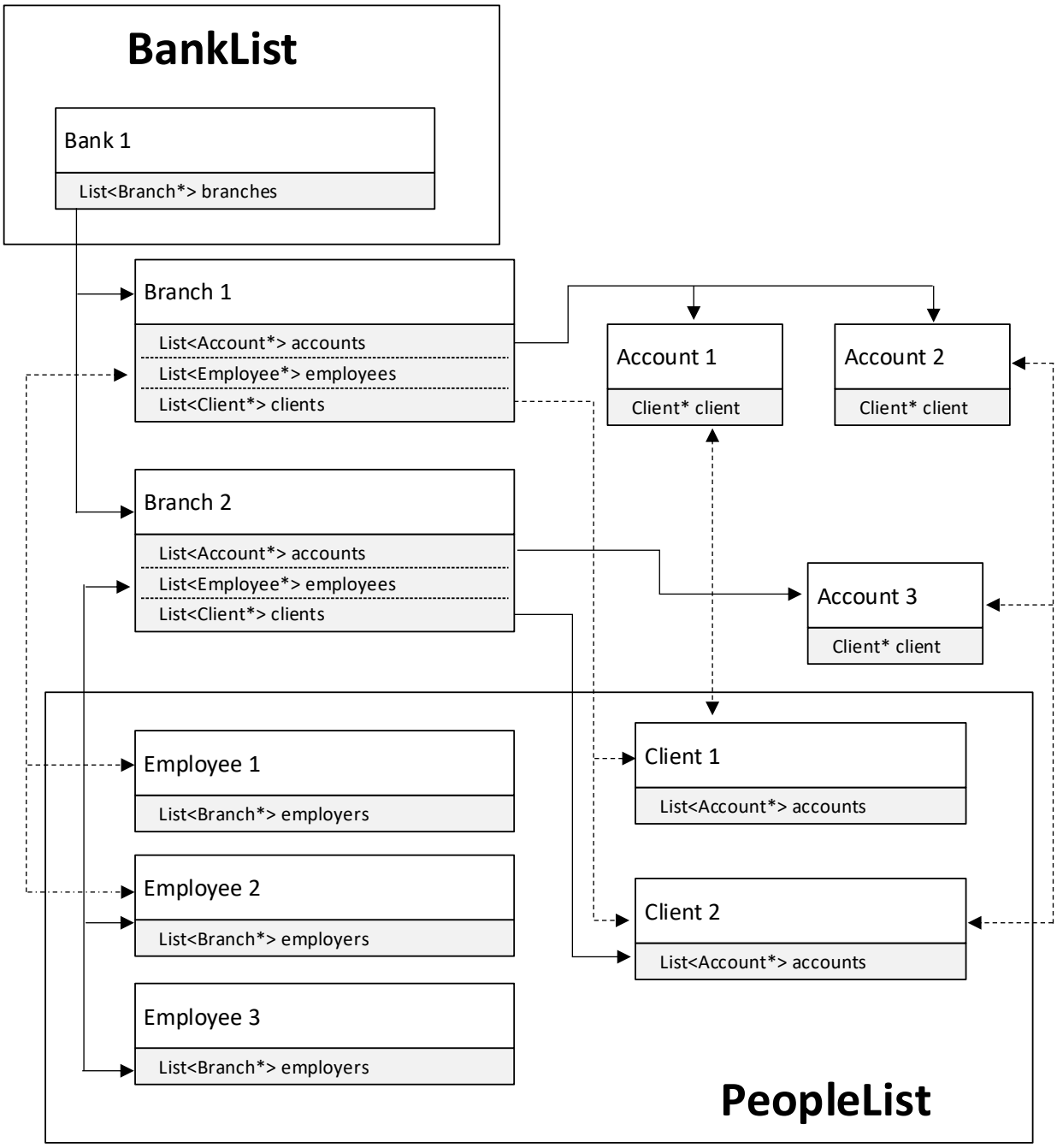
The program also consists of two “main containers”: **BankList** and **PeopleList**, which create, store and operate on *Banks* and *People*, respectively.

For proper use of the program, most of the classes should be created using other classes:

- *Clients* and *Employees* are created using *PeopleList*.
- *Banks* are created using *BankList*
- *Branches* are created using *Banks*
- *Accounts* are created using *Branches*
- Objects of class *Person* cannot be created by other classes
- *Address* can be created on its own, and passed to other classes


Account
Class
Fields
accountNumber
balance
client
transactions
Methods
~Account
Account (+ 1 overload)
deposit
getBalance
getNumber
getOwner
printTransactions
withdraw
Nested Types
Transaction
Struct
Fields
title
value
BankList
Class
Fields
banks
Methods
~BankList
addBank
BankList (+ 1 overload)
getBank
nameExists
operator=
removeBank
Bank
Class
Fields
branches
name
Methods
~Bank
Bank (+ 1 overload)
getBranch
getBranches
getName
idExists
newBranch (+ 1 overload)
removeBranch
Nested Types
Branch
Class
Fields
accounts
address
clients
employees
id
Methods
~Branch
Branch (+ 2 overloads)
closeAccount
fire
getAccounts
getAddress
getClients
getEmployees
getID
hire
newAccount
List<T>
Template Class
Fields
head
length
tail
Methods
~List
back
begin
clear
empty
end
erase
front
insert
List (+ 1 overload)
operator!=
operator[]
operator=
operator==
pop_back
pop_front
push_back
push_front
size
Nested Types
Employee
Class
Person
Fields
positions
totalEarnings
totalHours
Methods
~Employee
Employee (+ 2 overloads)
getEarnings
getEmployers
getHours
getPositions
newJob
quit
Nested Types
Position
Struct
Fields
employer
hours
wage
public
Person
Class
Fields
address
id
name
surname
Methods
~Person
fullName
getAddress
getID
getName
getSurname
Person (+ 3 overloads)
setName
Address
Class
Fields
buildingNo
city
flatNo
street
zipCode
Methods
~Address
Address (+ 1 overload)
getBuildingNumber
getCity
getFlatNumber
getStreet
getZipCode
set
toString
zipCorrect
public
Client
Class
Person
Fields
accounts
totalBalance
Methods
~Client
Client (+ 2 overloads)
closeAccount
getAccounts
getBalance
newAccount
update
PeopleList
Class
Fields
people
Methods
~PeopleList
addClient (+ 2 overloads)
addEmployee (+ 2 overloads)
getPerson
idExists
operator=
PeopleList (+ 1 overload)
removePerson

1.3 Memory map



1.4 Limitations

Due to the implementation of the program there exist some limitations to its usage:

- One person cannot be both an employee and a client of a bank, as they are two separate classes
- Classes *Branch,* and *Service* cannot change their 'owners' (*Branch* and *Person*), nor can they be initialized without one

There also exists some limitations by design of the program. (which are defined as preprocessor defines and can be changed)

2. TESTING

2.1 Testing approach

To test the program, "test classes" will be implemented. Inside the test classes all of the methods will be tested with correct parameters, as well as extreme cases. Each class will compare the results to those expected, and count any errors that may occur. In the final version, the program should throw an exception whenever any method or class is used incorrectly.

2.2 Examples

Here are some examples presenting the tests for various classes and methods. Not every test is listed, as other tests are analogical to those presented.

Action	Expected outcome
Creating a new *Bank* in a *BankList* with proper parameters	Normal usage of the program, the *Bank* is created
Creating a *Bank* with an already existing name	Exception thrown, such action is forbidden
Trying to delete a nonexistent account	Exception thrown, cannot delete an account which does not exist
Deleting an account from a *Branch* and trying to access it from a *Client*	The account should be gone from the list in *Client*, without leaving a null pointer, so an exception is thrown, because the account doesn't exist
Checking for memory leaks	All of the memory should be properly freed.
Trying to create both a *Client* and *Employee* with the same ID	Exception thrown, the program should check for uniqueness of ID
Creating an object (such as Address) with improper parameters – eg. a zip code not following the format	Exception thrown, the program should check for the correctness of inputted data

3. CLASS DETAILS

The members and methods of classes are an early draft and are a subject to change during development

3.1 Account.....

Description	
Class describing an account	
Members	

`struct Transaction`
`{`
`  double value;`
`  std::string title;`
`};`
	A simple structure holding information about last transactions

`List<Transaction> transactions`	A list of last transactions, limited to 10 by default
`double balance`	The balance of the account
`std::string accountNumber`	The number of the account
`Client* client;`	The owner of the account
Methods	

`Account()`
`Account( […] )`
`~Account()`
	Constructors and destructor

`double getBalance()`
`void printTransactions( […] )`
`Client* getOwner()`
	Access to members

`void deposit( […] )` `void withdraw( […] )`	Changing the value of balance

3.2 Address

Description	
Address is a simple class storing a couple of strings that denote an address	
Members	
`std::string street`	String denoting a street name
`std::string city`	String denoting a city name
`std::string zipCode`	String denoting a Zip Code
`std::string buildingNo`	String denoting a building number
`std::string flatNo`	String denoting a flat number
Methods	

`Address()`
`Address( […] )`
`~Address()`
	Constructors: default and taking all arguments, with flatNo being optional. Default destructor

`void set( […] )`	Sets the whole address, similarly to the constructor
`std::string toString()`	Returns the whole address as one single string

`std::string getStreet()`
`std::string getCity()`
`std::string getZipCode()`
`std::string getBuildingNumber()`
`std::string getFlatNumber()`
	Returning values of every member separately

3.3 Bank

Description	
Class describing a bank intuition	
Members	
`class Branch`	A nested class describing a branch, further explained in 2.2.1
`std::string name`	The name of the bank
`List<Branch> branches`	A list of all the branches of the bank
Methods	

`Bank()`
`Bank( […] )`
`~Bank()`
	Default constructor and a constructor with arguments. Default destructor

`std::string getName()` `const List<Branch>& getBranches()`	Getting the elements – constant reference to list of branches means we can see the branches but not modify them

`void newBranch( […] )`
`void removeBranch( […] )`
`Branch& getBranch( […] )`
	Creating, removing and modifying branches

3.3.1 Branch

Description	
A nested class of *Bank* describing a single branch	
Members	
`const std::string id`	An ID number identifying the bank branch
`Address address;`	The address of the branch *(see 2.1)*
`List<Account*> accounts`	List of all the services the branch provides *(see 2.4)*
`List<Employee*> employees`	List of all the employees of the branch *(see 2.3.2)*
`List<Client*> clients`	List of all the clients of the branch *(see 2.3.1)*
Methods	
`Branch( […] )` `~Branch()`	Constructors set the values of id and address. Destructor removes all services and fires all employees

`Address& getAddress()`
`std::string getID()`
`const List<Account*> getAccounts()`
`const List<Employee*> getEmployees()`
`const List<Client*> getClients()`
	Access to the members

`void newAccount( […] )`
`void closeAccount( […] )`
`void hire( […] )`
`void fire( […] )`
	Adding/Removing services and employees

3.4 Person

Description	
Basic class describing a person, from which *Employee* and *Client* inherit	
Members	
`std::string name`	Name of the person
`std::string surname`	Surname of the person
`std::string id`	ID of the person (PESEL)
`Address address`	Address of the person *(see 2.1)*
Methods	

`Person()`
`Person( […] )`
`~Person()`
	Default constructor and constructor with arguments.
	Default destructor.

`Address& getAddress()`
`std::string getName`
`std::string getSurname()`
`std::string getID()`
	Access to members

`std::string setName( […] )`	Changing of the name and surname (ID shouldn't change and Address can be changed through getAddress())
`std::string fullName()`	Returns a string consisting of both name and surname

3.4.1 Client

Description	
A class inheriting from *Person*, describing a client of a bank	
Members	
`List<Account*> accounts`	A list of all Services *(see 2.4)* of the client
`double totalBalance`	The total amount of balance on the accounts,
Methods	
`Client();` `~Client();`	Default constructor. Destructor closes all services.
`void newAccount( […] )` `void closeAccount( […] )`	Opening and closing of services
`const List<Service*> getServices()` `double getBalance()`	Access to members
`void update()`	Updating the value of totals

3.4.2 Employee

Description	
A class inheriting from *Person*, describing an employee of a bank	
Members	
`struct Position { Bank::Branch* employer; double wage; int hours; };`	Simple structure describing the positions of the employee
`List<Position> positions`	The list of posisions
`double totalEarnings`	Summarized earnings of the employee
`int totalHours`	Total weekly hours worked by the employee
Methods	
`Employee();` `~Employee();`	Default constructor. Destructor quits all jobs.

`const List<Bank::Branch*> getEmployers()`
`const List<Position>& getPositions()`
`int getHours()`
`double getEarnings()`
	Access to members

`void newJob( […] )` `void quit(…])`	Adding/removing a job