INDIVIDUAL ASSIGNMENT

FOR

Object Oriented Methods with UML (CX006-3-3-OMU)

Group 1 - Banking System

By

Adrien Poupa

TP040869

INTAKE: UCFEFREI1603

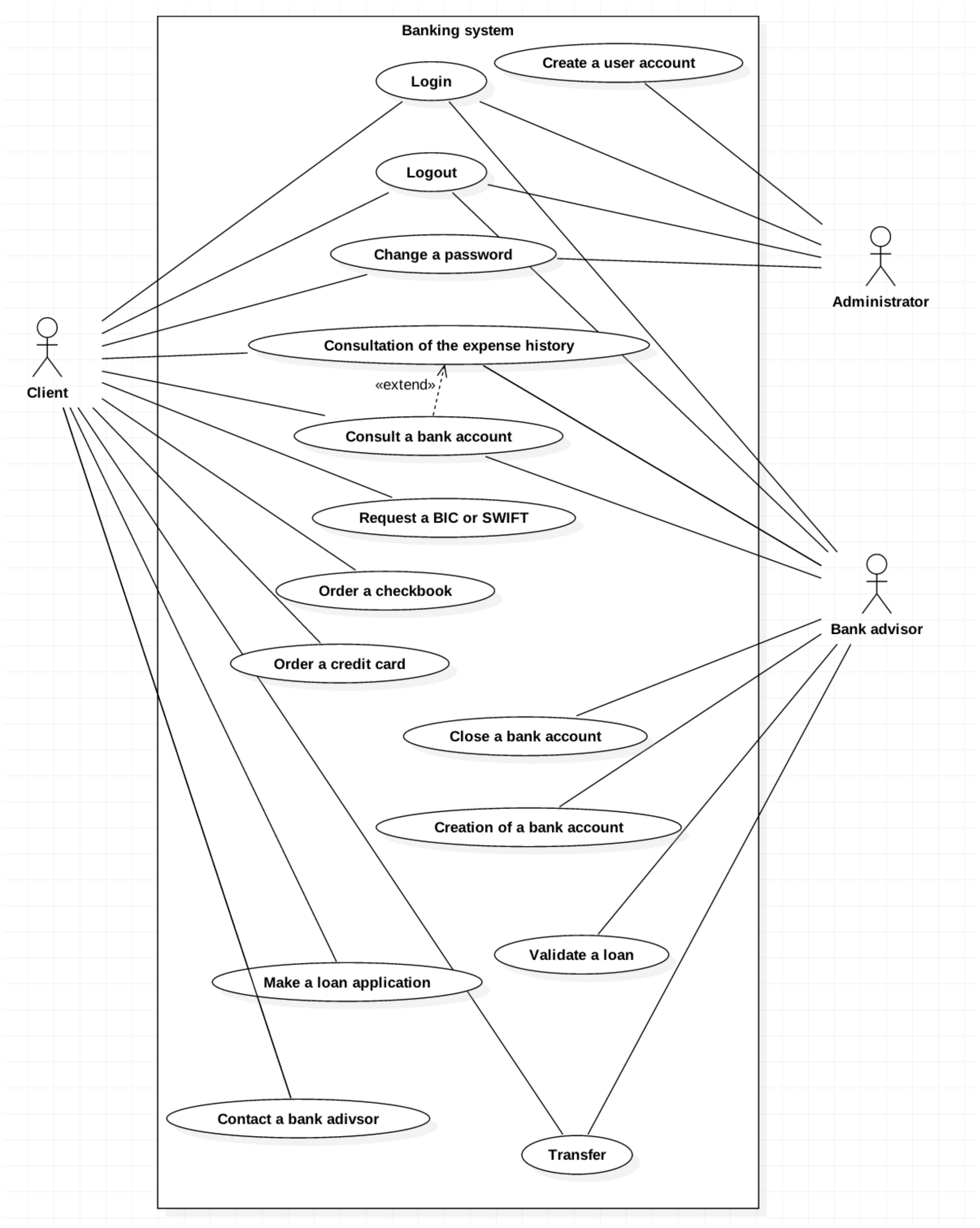
DUE DATE: 25th May 2016

ONLINE REPOSITORY: https://github.com/AdrienPoupa/banking-system

NAME OF LECTURER: MRS BAZILA BANU

Table of contents

Use case diagram



Use cases

Use case 1	Consultation of the expense history
Pre-condition	User is logged in
Post-condition	Display the expense history
Main success scenario	1 – The user enters expense 2 – The user knows his expense history
Alternate flow	1 – User wants to display an account that does not exist: display “This account does not exist” 2 – User does not have the rights to consult his account: display “You can’t access this account”

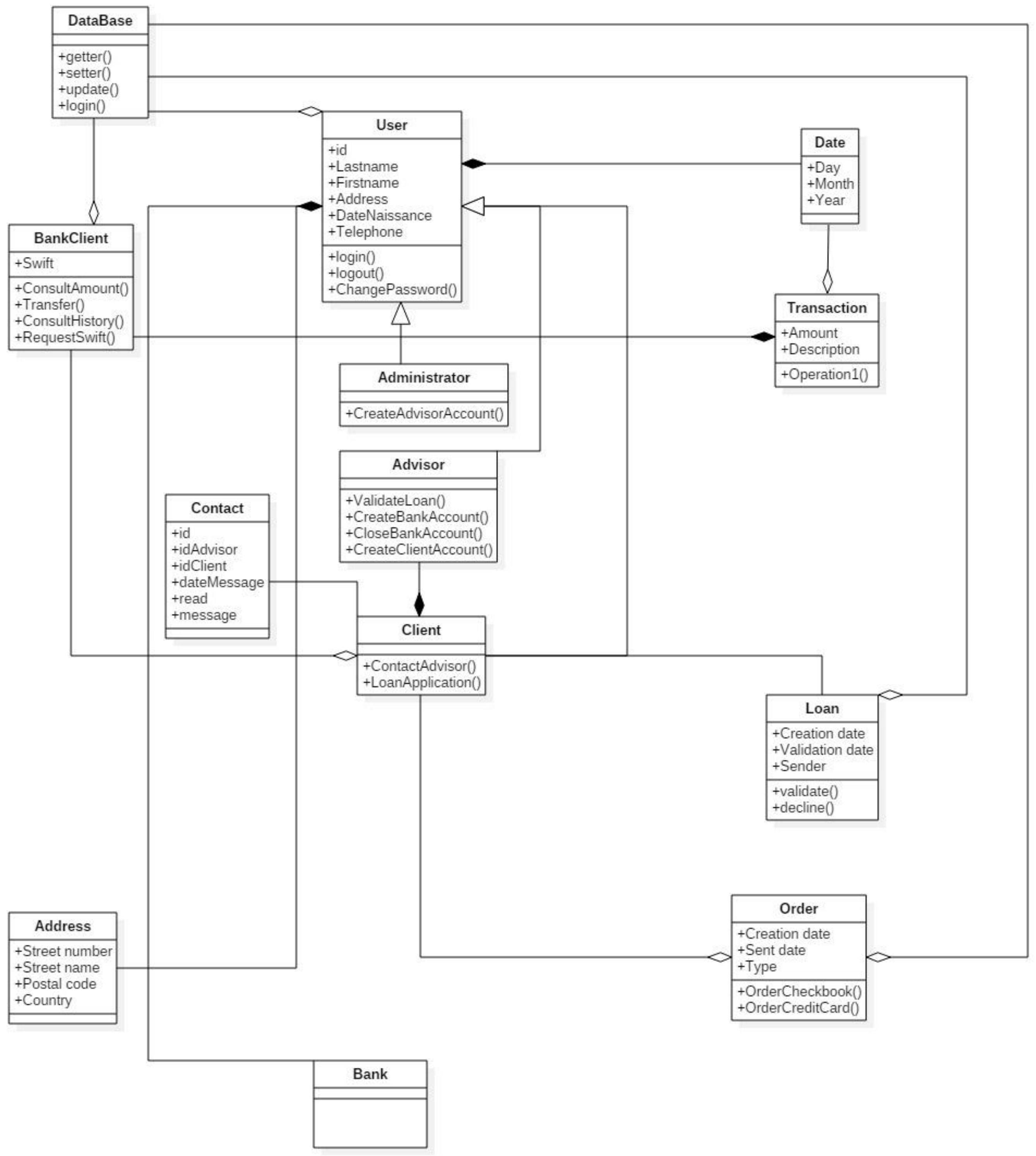
Use case 2	Order a checkbook
Pre-condition	User is logged in
Post-condition	Inform the user his checkbook has been ordered, display: “Your checkbook has been ordered”
Main success scenario	1 – The user enters checkbook 2 – Send the order
Alternate flow	User already has a checkbook order pending, display: “You already have a checkbook order pending”

Use case 3	Order a credit card
Pre-condition	User is logged in
Post-condition	1 – The user enters credit card 2 – Send the order
Main success scenario	Send the credit card, display “Your credit card has been sent”
Alternate flow	User already has a credit card

Use case 4	Change password
Pre-condition	User is logged in
Post-condition	Inform the user his password has been changed
Main success scenario	1 – The user enters password 2 – The user is asked to enter his new password twice 3 – Change user’s password, display “Your password has been changed”
Alternate flow	1 – Password is invalid (too short) 2 – Password is invalid (same as username)

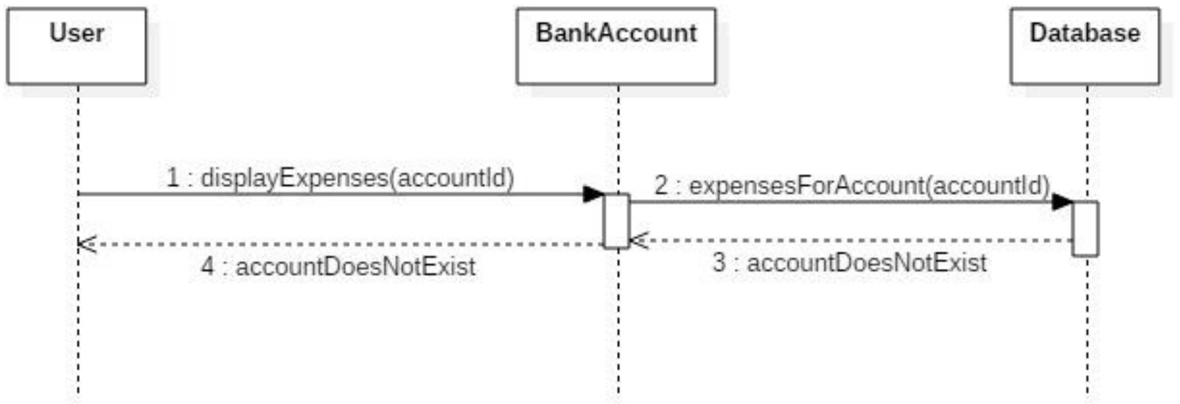
Use case 5	Request a SWIFT/BIC
Pre-condition	User is logged in
Post-condition	1 – The user enters SWIFT 2 – Inform the user of his SWIFT and BIC credentials
Main success scenario	Display the SWIFT and BIC number

Class diagram

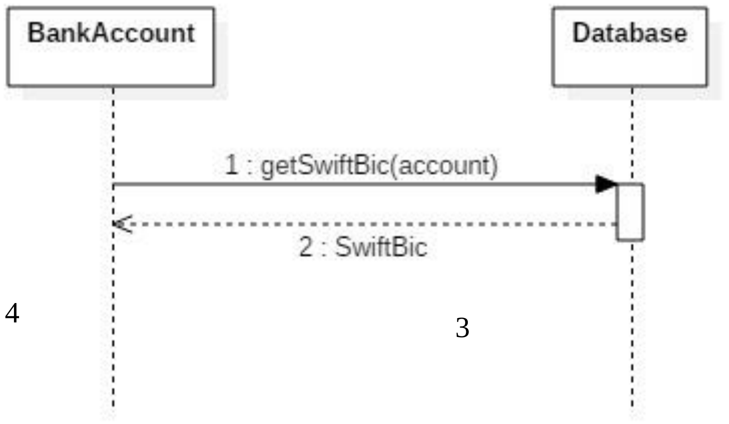


Sequence diagrams

Consultation of the expense history



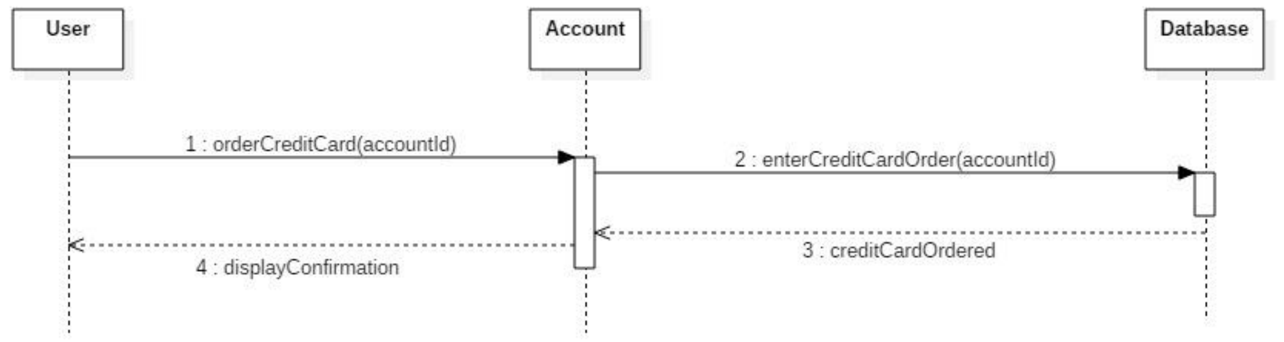
Alternate flow 1



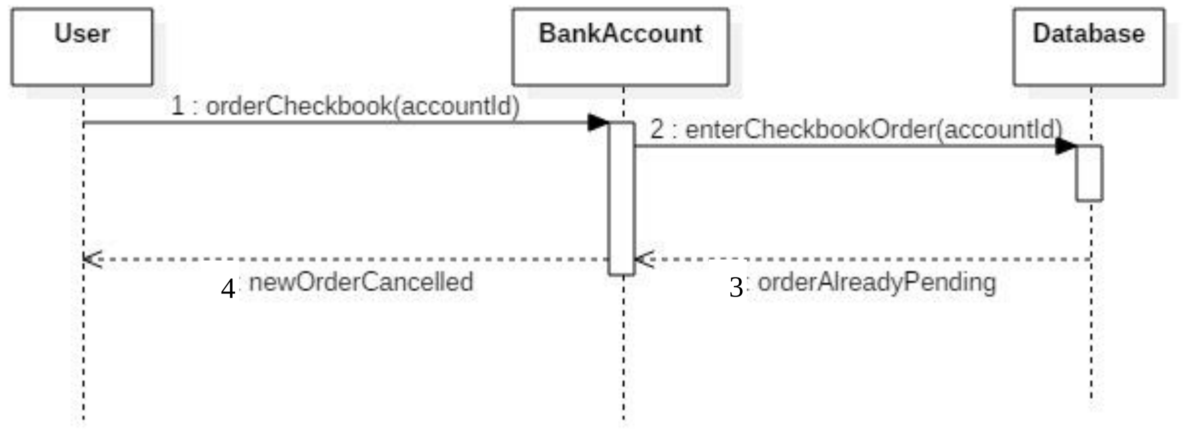
4

3

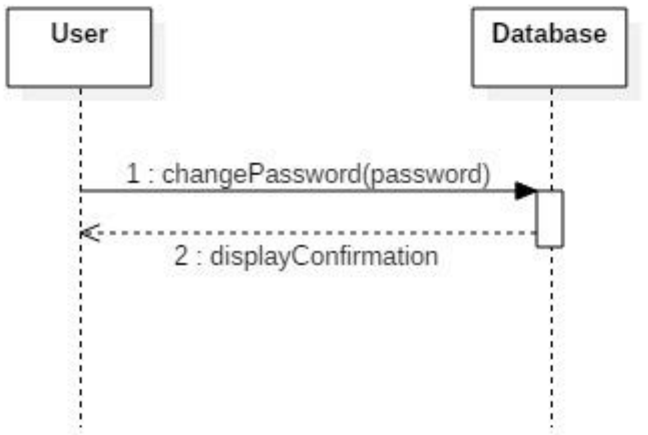
Order a credit card


User
Account
Database
1 : orderCreditCard(accountId)
2 : enterCreditCardOrder(accountId)
3 : creditCardOrdered
4 : displayConfirmation

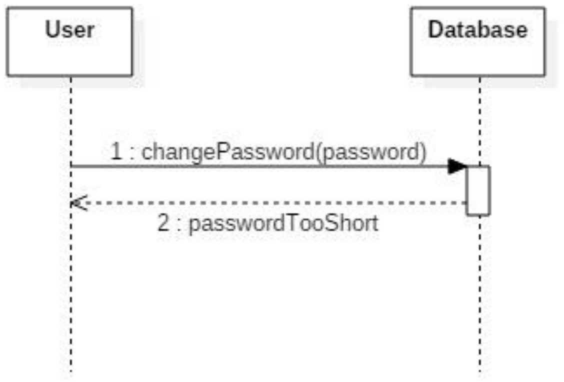
Order a checkbook


User
BankAccount
Database
1 : orderCheckbook(accountId)
2 : enterCheckbookOrder(accountId)
3 orderAlreadyPending
4 newOrderCancelled

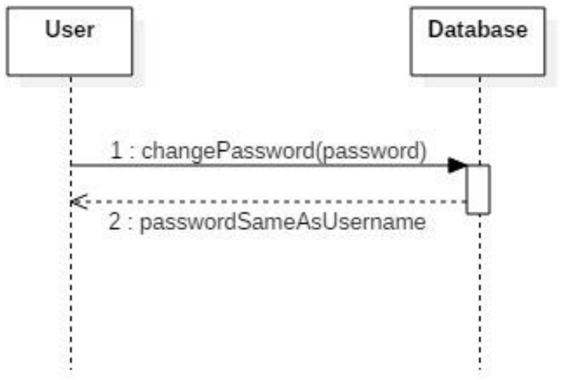
Change password



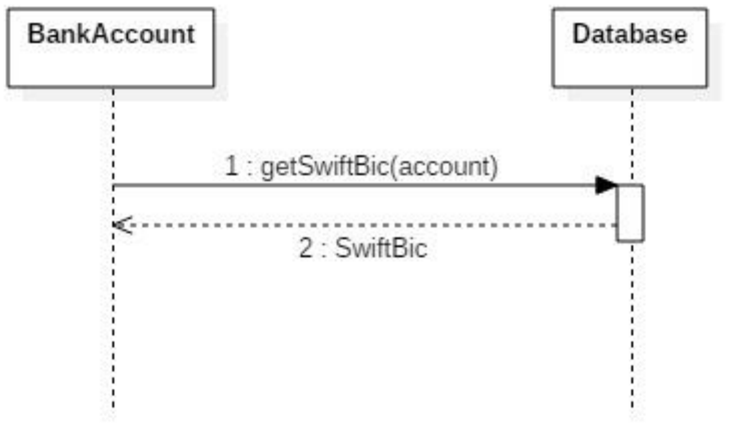
Alternate flow 1



Alternate flow 2

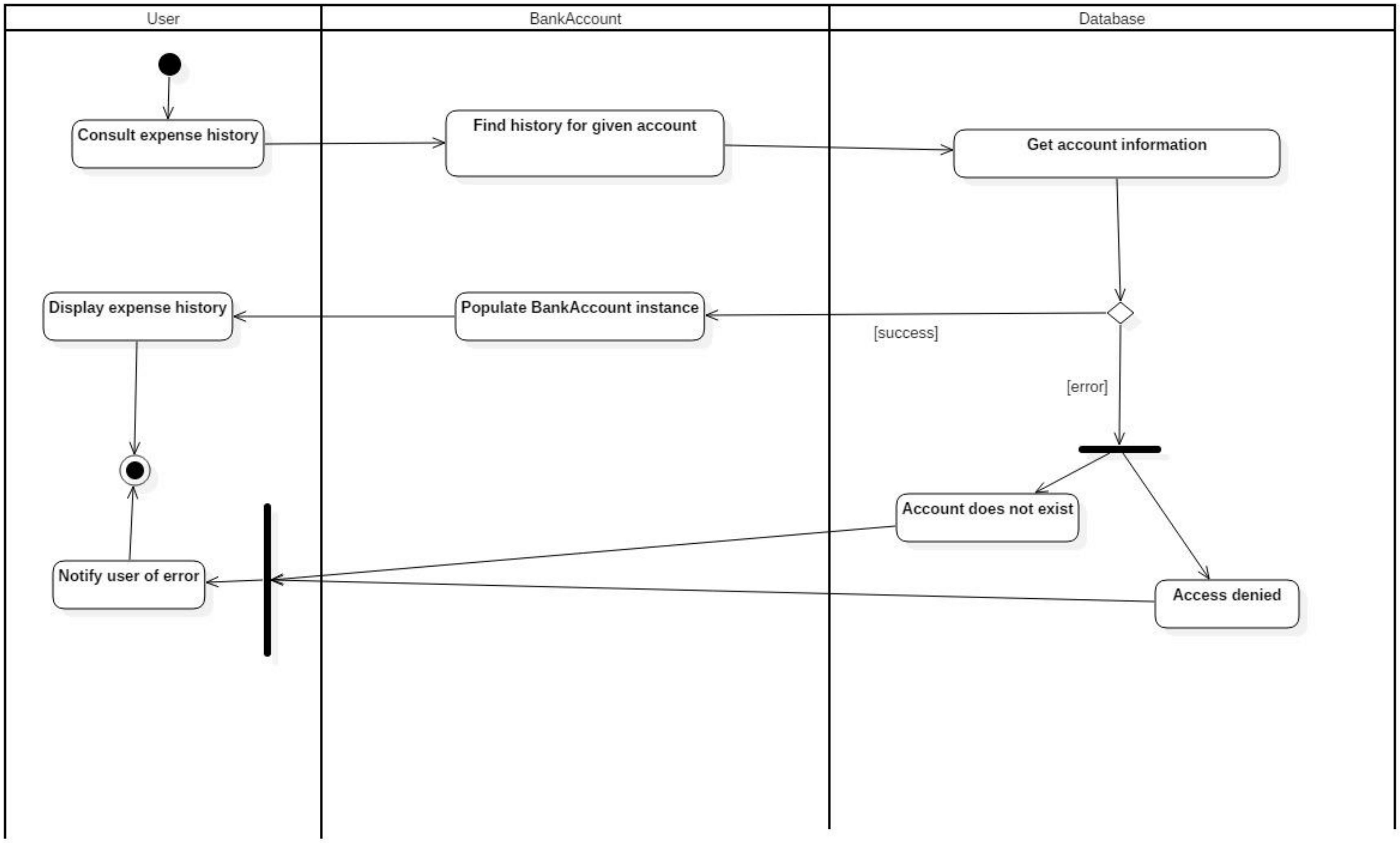


Request a SWIFT/BIC

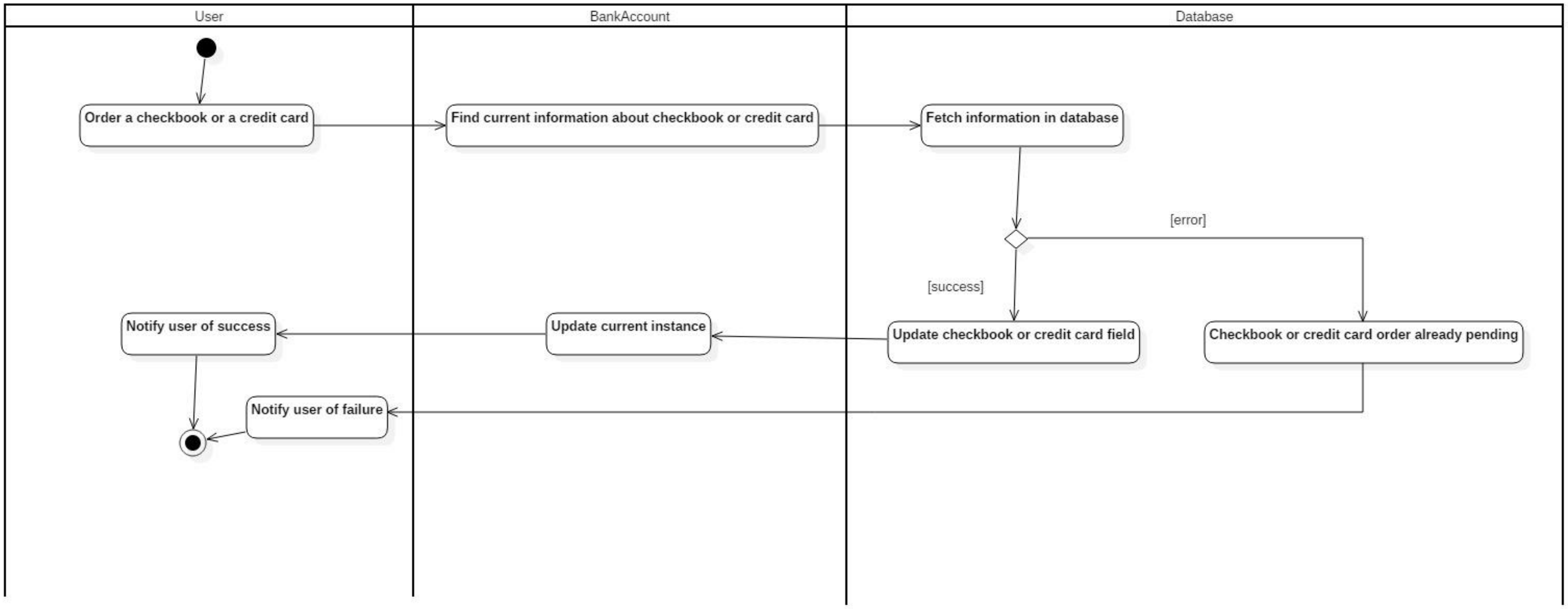


Activity diagrams

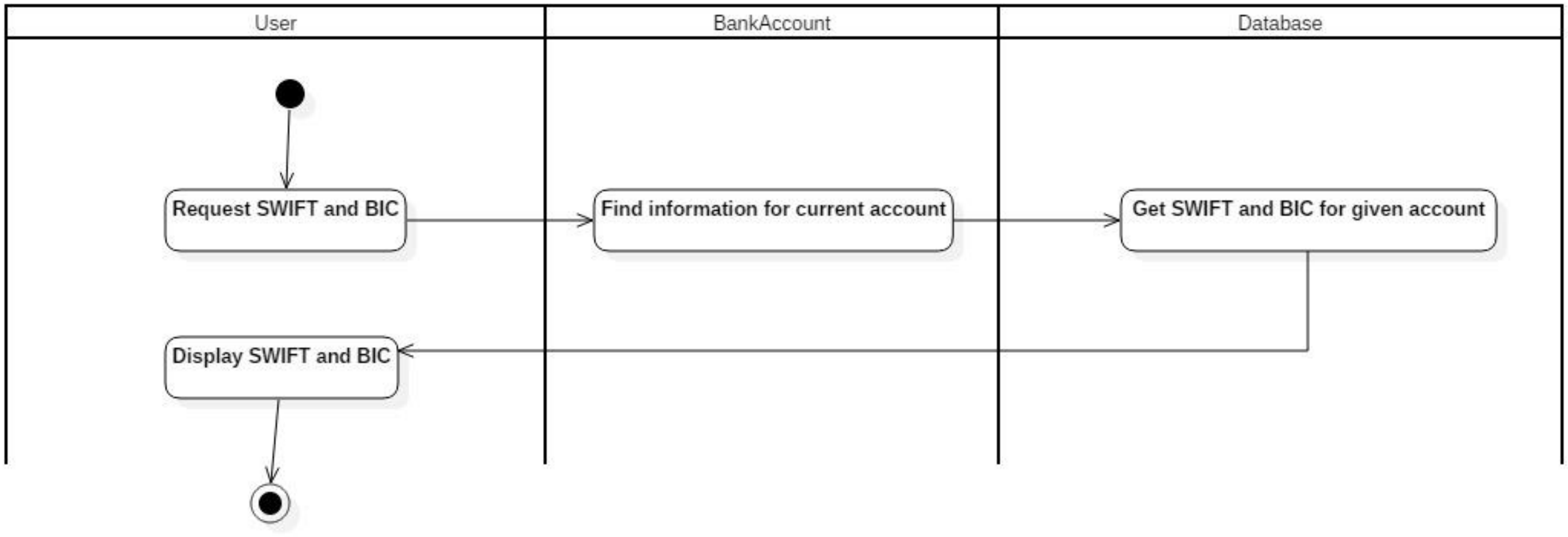
Consultation of the expense history



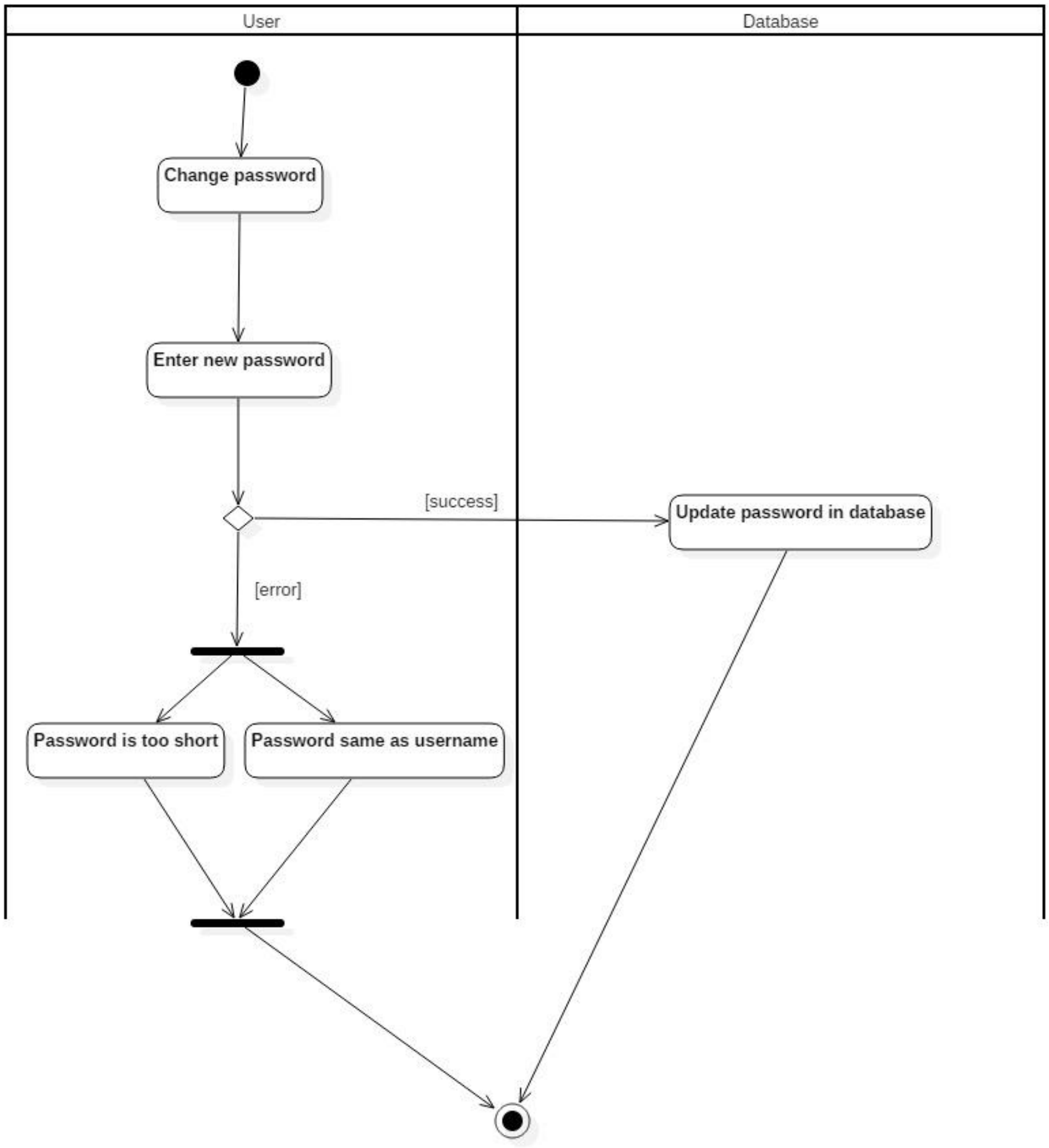
Order a checkbook or a credit card


User
BankAccount
Database
Order a checkbook or a credit card
Find current information about checkbook or credit card
Fetch information in database
[error]
[success]
Notify user of success
Update current instance
Update checkbook or credit card field
Checkbook or credit card order already pending
Notify user of failure

Request SWIFT/BIC


User
BankAccount
Database
Request SWIFT and BIC
Find information for current account
Get SWIFT and BIC for given account
Display SWIFT and BIC

Change password



Implementation

Consultation of the expense history

The user can consult his expenses from the main menu. He selects the correct option (number 5) and a query is run in order to get the IDs of the accounts he owns. For every account, using a loop, a new instance of each account is created and the appropriate function is called (see appendix).

This function, called ConsultHistory, consists of another query to the transactions table in order to get all the expenses (transactions IDs) for the given account. Once all the transactions IDs are entered into a set of ints, a new loop is run for each transaction, allowing it to be displayed properly for the user.

Order a checkbook or a credit card

To order a checkbook or a credit card, one has to select the proper option in the menu (numbers 6 and 7). Once it is done, the list if his accounts is displayed using the getBankAccounts function from the Client class.

As for the previous use case, it uses a query to select the accounts owned by the user from its ID and a loop to show each of them. Then, one can enter the ID of the bank account he or she wants to order a checkbook or a credit card to. If the ID is invalid (ie: he does not own the account or the account does not exist) an error message is displayed and he has to enter a proper ID. This is done comparing the ID he entered to the IDs fetched from the database and stored in a set of ints. When the ID is correct, the function returns a reference to the account the user has chosen.

Then, a new Order object is instantiated. Using setters, the current user (client) and the bank account he has chosen are linked to it. Finally, the OrderCheckbook (or OrderCreditCard) function is called from this new object and does the following: it queries the database to make sure an order is not already pending for the given user; all the attributes for the order are correctly filled (date of the day, type of the order, etc) and a new notification to the advisor informing him of the order is entered.

Finally, the user is prompted a message telling him his order has been entered into the database.

Change password

To change his password, the user has to be logged in. He has to select the "Change password" option in the menu and is prompted to enter its new password. Once the password is entered in the console, the setPassword function from the User class is called. If the password entered by the user is shorter than 8 characters, an error message is displayed. If the password entered by the user is similar to his name or surname, another error message is displayed as well.

If the password is valid, it is hashed using the SHA256 class. The resulting hash is then inserted into the database, and the user can login using his new password right away.

Request a SWIFT/BIC

This feature is accessible through the "Consult bank account" option in the main menu. When the user selects this option, a list of his bank accounts is displayed, including obviously the SWIFT and BIC information. In order to display them, the getBankAccounts function from the Client class is called, querying the database to get the account(s) of a specific user.

```
-- Account's list of Poupa Adrien --

 ID |      SWIFT      |    BIC

----|-------------------|--------------

6     frAQBc8Wsa1         xVPfvJcr
```

After a short display of the accounts is displayed, the user is offered to see the details of one account in particular by entering its ID. If the ID is invalid (ie: the given account does not exist) an error message is displayed and he has to enter a proper ID. This is done comparing the ID he entered to the IDs fetched from the database and stored in a set of ints. When the ID is correct, it displays further information about the account selected such as the amount of money available.

Appraisal

Database

In order to store the information for multiple sessions, we have chosen to use a database, SQLite. It is simple and uses a flat file in order to store the data. I have decided to use the BaseModel class to query the database; it is a class that Timothée Barbot and I have written in December for a common C++ project at Efrei, hence it is not plagiarism if we use the same codebase. It is written on top of SQLiteCPP, a library used as a mapper for the official C library provided by SQLite. Same goes for the Date and Address class which were written at this moment.

This class allows to get an entity by its ID, to run a SELECT query using WHERE clauses or not, to save an entity using INSERT INTO queries and remove an element with DELETE. The arguments taken by these functions are the ID request, or more often the fields to SELECT or the WHERE conditions.

The select method returns a map<int, map<string, string>>. A set of result looks like:

map<0, map<id, 1>>

map<1, map<id, 2>>

The save function inserts or update a result depending on the ID of the instance: 0 in case of insertion, the actual ID if it needs to be updated. It takes a map<string, vector<string>> as argument, like multiple lines of {"id", {to_string(_id), "int"}}. The first element is the name of the field, the second its value and the last one its type, int or string. Since all results must be passed as a string, the integers have to be converted using to_string.

To create an instance from an ID, a SELECT query is made with the function getById, which returns a map<string, string>. Then, in each class, the constructor taking only an ID can use the map to create the instance, while taking care of casting results if needed, since all of them are strings when they extracted from the database using stoi (string to integer) function.

map<string, string> data = BaseModel::getById(_dbTable, id);

_isAdmin = stoi(data["isadmin"]);

Finally, the remove function is the simplest since it only needs the table and the ID to be deleted.

Hashing

Having created websites before, I know how important it is to encrypt users' password into database and not to store them in plain text in the unlikely event of a hacking. In order to do so, I used a class I found to encrypt a string using the SHA256 algorithm. As a result, a simple word results in the following hash:

9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08

Notifications

While this was not planned at the beginning, I felt like I should add a feature to display notifications to the user or the advisor connecting. Otherwise, how could he know quickly who contacted him or her and what the message was? How could he know if his or her loan application was approved? To do so, I created a new class named Notification and a new table to store the notifications. Each notification only has a small number of information: an ID, the message, who it is addressed to and if it has already been read. When a user connects, a notification may pop up like this:

```
################################
#                              #
#  --- New Notifications ---   #
#                              #
################################
```

1. Your loan inquiry for RM5000 has been approved

After the notification has been displayed once, it is marked as read in the database.

Encapsulation

All the attributes are either protected or private, hence a good encapsulation. Of course, accessors (setters and getters) are available to modify and consult those attributes easily.

Overloading

The large majorities of objects have their cin and cout methods overloaded, allowing an elegant syntactic sugar when one has to create or display such an object by doing

`cin >> objectNameToCreate;` or `cout << objectNameToDisplay;`

Singleton Pattern

The Singleton pattern is used in order to launch the application in a convenient and reliable way. Indeed, this prevents the SQlite file to be corrupted with concurrent access.

User interaction class: Bank

The Bank class is used to interact between the user and the classes not accessible to him directly. It handles the menus, the selections and the login/logout part.

Users and inheritance

Last but not least, the users and their **inheritance**. The User class is **abstract** since it has a virtual function. It is not instantiable, and has three children: Administrator, Advisor and Client. This is very convenient because it allows to do operations such as get an ID for a User entity without knowing if the actual object is an Administrator, an Advisor or a Client

Criticism

We did not include polymorphism as we thought we would, mainly because we did not anticipate that our functions would be tight with each other. Clearly, we lacked time and efficiency to include polymorphism properly without making it pointless.

Another point that we could have improved is the menu number: right now, numbers are not following each other, not providing a smooth user experience.

Finally, we should have added an option to cancel any operation, but we realized it would have been a nice feature too late in the development.

Appendix

Instructions of installation

The banking system is easy to install. All you need to do is install a C++ compiler that is compatible with C++11 and make sure the database file (bank.db3) is located at the root of the project and is accessible for reading and writing to the user launching the program (more precisely the compiler).

The default passwords are “123” or “test”.

Of course, you can run the executable file for an easier installation. Make sure that the database file is located at the same level than the executable file and double click banking-system.exe. You are good to go.

Code

BaseModel.cpp, used to query the database

```
/**
 * SQlite mapper
 * Adrien Poupa & Timothée Barbot
 * December 2015
 */

#include "BaseModel.h"

using namespace std;

map<string, string> BaseModel::getById(const string& table, const int& id)
{

    map<string, string> data = map<string, string>();

    try
    {
        SQLite::Database db("bank.db3");

        SQLite::Statement query(db, "SELECT * FROM " + table + " WHERE 
id=?");
        query.bind(1, id);

        int resultCount = 0;
        while (query.executeStep())
        {
            resultCount++;
            for(int i = 0; i < query.getColumnCount(); i ++)
            {
                data.insert({query.getColumnName(i), 
query.getColumn(i).getText()});
            }
        }

        return data;

    } catch (exception& e) {
        cout << "exception: " << e.what() << endl;
        return data;
    }
}

map<int, map<string, string>> BaseModel::select(const string& table, const 
string& fields, const string& where)
{

    map<int, map<string, string>> data = map<int, map<string, string>>();

    try
    {
        SQLite::Database db("bank.db3");
        SQLite::Statement query(db, "SELECT " + fields + " FROM " + table +
(where.length() != 0 ? " WHERE " + where : ""));

        int resultCount = 0;
        while (query.executeStep())
        {
            resultCount++;
            for(int i = 0; i < query.getColumnCount(); i ++)
            {
                data[resultCount].insert({query.getColumnName(i),
query.getColumn(i).getText()});
            }
        }

        return data;

    } catch (exception& e) {
        cout << "exception: " << e.what() << endl;
        return data;
    }
}

int BaseModel::save(const string& table, map<string, vector<string>> data)
{

    /*
     data : {
        {"_id", {"15", "int"}},
        {"attr1", {"jean", "string"}},
        ...
     }
     */

    // Update
    if (stoi(data["id"][0]) != 0) {
        try
        {
            SQLite::Database db("bank.db3",
SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE);

            string queryString = "UPDATE " + table + " SET ";

            for(auto const& elem : data)
            {
                if (elem.first != "id")
                {
                    queryString += elem.first.c_str();
                    queryString += "=?, ";
                }
            }

            queryString = queryString.substr(0, queryString.size() - 2);

            queryString += "WHERE id=" + data["id"][0] + ";";

            // queryString : UPDATE table SET attr1=?, attr2=?, attr3=? WHERE
id=?
```

```
            SQLite::Statement query(db, queryString);

            int n = 1;
            for(auto const& elem : data)
            {
                if (elem.first != "id")
                {
                    if (elem.second[1] == "string")
                    {
                        query.bind(n, elem.second[0]);
                    }
                    else if (elem.second[1] == "int")
                    {
                        query.bind(n, stoi(elem.second[0]));
                    }
                    n++;
                }
            }

            // queryString: UPDATE table SET attr1=val1, attr2=val2,
attr3=val3 WHERE id=_idval

            query.exec();

            return true;
        }
        catch (exception& e)
        {
            cout << "SQLite exception: " << e.what() << endl;
            return false;
        }
    }

    // Insert
    else
    {
        try
        {
            SQLite::Database db("bank.db3",
SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE);

            string queryString = "INSERT OR IGNORE INTO " + table + "(";

            for(auto const& elem: data)
            {
                if (elem.first != "id")
                {
                    queryString += elem.first + ", ";
                }
            }

            queryString = queryString.substr(0, queryString.size() - 2);
            queryString += ") ";

            queryString += "VALUES( ";
            for(auto const& elem: data)
            {
                if (elem.first != "id")
                {
                    queryString += "?, ";
                }
            }

            queryString = queryString.substr(0, queryString.size()-2);

            queryString += ");";

            // Insert query
            SQLite::Statement query(db, queryString);

            int n = 1;
            for(auto const& elem : data)
            {
                if (elem.first != "id")
                {
                    if (elem.second[1] == "string")
                    {
                        query.bind(n, elem.second[0]);
                    }
                    else if (elem.second[1] == "int")
                    {
                        query.bind(n, stoi(elem.second[0]));
                    }
                    n++;
                }
            }

            query.exec();

            // Update current ID
            int tmp = db.execAndGet("SELECT last_insert_rowid();");
            return tmp;
        }
        catch (exception& e)
        {
            cout << "SQLite exception: " << e.what() << endl;
            return false;
        }
    }
}

int BaseModel::getCount(const string& table, string filter){

    if (filter != ""){
        filter = " WHERE " + filter;
    }
    try
    {
        SQLite::Database db("bank.db3");

        SQLite::Statement query(db, "SELECT count(*) FROM " + table +
filter);
```

```
        while (query.executeStep())
        {
            return query.getColumn(0);
        }


    } catch (exception& e) {
        cout << "exception: " << e.what() << endl;
    }
    return 0;
}

bool BaseModel::remove(const string& table, const int& id)
{

    // We cannot delete a non-existing field
    if (id == 0)
    {
        return false;
    }

    try
    {
        SQLite::Database db("bank.db3",
SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE);
        // Delete query
        SQLite::Statement   query(db, "DELETE FROM " + table + " WHERE
id=?");
        query.bind(1, (int) id);
        query.exec();
    }
    catch (exception& e)
    {
        cout << "SQLite exception: " << e.what() << endl;
        return false;
    }
    return true;
}
```

Consultation of the expense history

Bank.cpp, call to the appropriate function

```
// Expense history
case 5: {
    Client client = Client(_currentUser->getId());
    set<int> accounts = client.getAccountsIds();

    for(auto i : accounts) {
        BankAccount* account = new BankAccount(i);
        account->ConsultHistory();
    }

    break;
}
```

Functions used to display the expense history, Client.cpp, BankAccount.cpp and Transaction.cpp

```
set<int> Client::getAccountsIds() {
    map<int, map<string, string>> bankAccount =
BaseModel::select("bank_account", "id", "id_user = " + to_string(_id));

    int totalAccount = (int)bankAccount.size();

    set<int> AccountIds = set<int>();

    for (int i = 1; i != totalAccount + 1; i++)
    {
        AccountIds.insert(stoi(bankAccount[i]["id"]));
    }

    return AccountIds;
}


BankAccount::BankAccount(const int id) // Get a User from an ID provided by
DB
{
    map<string, string> data = BaseModel::getById(_dbTable, id);

    if (!data.empty()) {
        _id = (unsigned) id;
        _swift = data["SWIFT"];
        _bic = data["BIC"];
        _balance = stoi(data["balance"]);
        _idUser = (unsigned) stoi(data["id_user"]);
    }
    else {
        throw invalid_argument("The id of the bank account does not exist.");
    }
}
```

```
void BankAccount::ConsultHistory() {
    cout << "Expenses for account: " << getId() << endl;
    set<int> expenses = getExpenses();
    for (auto j : expenses) {
        Transaction* transaction = new Transaction((unsigned) j);
        cout << *transaction << endl;
    }
}

Transaction::Transaction(const unsigned int id) {
    map<string, string> data = BaseModel::getById(_dbTable, id);

    if (!data.empty())
    {
        _id = id;
        _account = BankAccount(stoi(data["account"]));
        _date = Date(data["date"]);
        _amount = stoi(data["amount"]);
        _description = data["description"];
    }
    else
    {
        throw invalid_argument("Please enter a valid ID");
    }
}
```

Request SWIFT/BIC

Function used to display the SWIFT/BIC, Client.cpp

```
BankAccount* Client::getBankAccounts() {
    map<int, map<string, string>> bankAccount =
BaseModel::select("bank_account", "id, swift, BIC, id_user, balance",
                                                             "id_user =
" + to_string(getId()));

    int totalAccount = (int)bankAccount.size();

    int idToOpen;
    set<int> AccountIds = set<int>();
    bool correctId = false;

    do{
        cout << "--------------------------------------------------------" <<
endl;
        cout << " -- Account's list of "<< getLastName() << " " <<
getFirstName() <<" --" << endl;
        cout << " ID |       SWIFT        |     BIC     " << endl;
        cout << "----|--------------------|---------------" << endl;
        for (int i = 1; i != totalAccount + 1; i++)
        {
                cout << bankAccount[i]["id"] << "     " <<
bankAccount[i]["SWIFT"] << "          " <<
                bankAccount[i]["BIC"] << endl;
                AccountIds.insert(stoi(bankAccount[i]["id"]));
        }

        cout << endl << "Bank account's id : " << endl;
        cin >> idToOpen;

        if(cin.fail())
        {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
        else
        {
            correctId = AccountIds.find(idToOpen) != AccountIds.end();
        }

        if (!correctId)
        {
            cout << "Unknown ID ..." << endl;
        }

    } while(!correctId);
    BankAccount *bc = new BankAccount(idToOpen);
    cout << *bc << endl;
    return bc;
}
```

Order a checkbook

Call from Bank.cpp

```
// Order a checkbook
case 6: {
    Client client = Client(_currentUser->getId());

    cout << "Available accounts:" << endl;
    Client client2 = Client(_currentUser->getId());

    BankAccount* toOpen = client2.getBankAccounts();

    Order* order = new Order();
    order->setClient(client);
    order->setAccount(*toOpen);
    order->OrderCheckbook();

    cout << "Checkbook ordered" << endl;

    break;
}
```

OrderCheckbook function from Order.cpp

```
void Order::OrderCheckbook() {

    map<int, map<string, string>> orders = BaseModel::select("orders", "id",
                                                              "user_id = " +
to_string(_user.getId()) + " AND sent = '0001-01-01'");

    int totalOrders = (int)orders.size();

    if (totalOrders > 0) {
        cout << "You already have an order pending" << endl;
    }
    else {
        Date* today = new Date();
        Date* sent = new Date(-1, -1, -1);
        this->setCreation(*today);
        this->setSent(*sent);
        this->setType(0);
        this->save();

        // Notify the advisor
        string notificationMessage = "New checkbook order from "
                                     + _user.getLastName() + " " +
_user.getFirstName();
        Notification* notification = new Notification(notificationMessage,
(unsigned) _user.getAdvisor());
        notification->save();
    }
}
```

To order a credit card, the code schema is almost the same.

Change password

Call from Bank.cpp

```
case 3:
    char newPassword[256];
    cout << "New password: " << endl;
    cin.ignore(1, '\n');
    cin.getline(newPassword, '\n');
    _currentUser->setPassword(newPassword);
    _currentUser->save();
    break;
```

setPassword function from User.cpp

```
void User::setPassword(const string password)
{
    if (password.length() < 8) {
        cout << "Password too short (must be 8 characters at least)" << endl;
    }
    else if (password == _firstName || password == _lastName) {
        cout << "Password same as username" << endl;
    }
    else {
        _password = sha256(password);
    }
}
```

Notification system

Notification.h

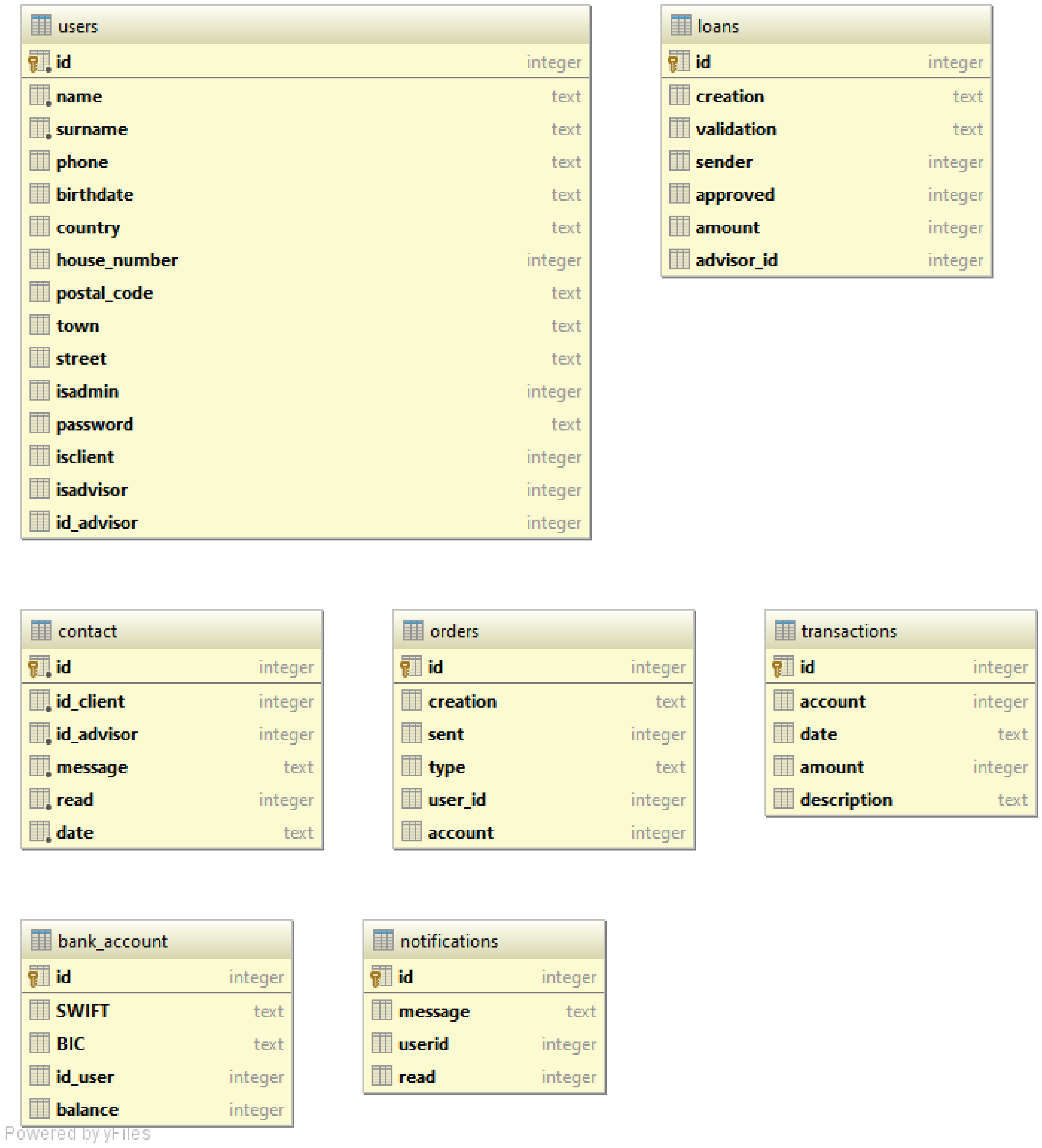
```
class Notification {
protected:
    unsigned int _id;
    std::string _message;
    unsigned int _userId;
    bool _read;

    static std::string _dbTable;
public:
    Notification();
    Notification(unsigned int id);
    Notification(std::string message, unsigned int userId);

    unsigned int getId();
    void setId(unsigned int id);

    std::string getMessage();
    void setMessage(std::string message);

    unsigned int getUserId();
    void setUserId(unsigned int userId);

    bool getRead();
    void setRead(bool read);

    bool save();

    bool remove();

    friend std::ostream& operator<< (std::ostream& stream, const
Notification& notification);
    friend std::istream& operator>> (std::istream& stream, Notification&
notification);
};
```

Notification.cpp

```
#include "Notification.h"
#include "BaseModel.h"

using namespace std;

string Notification::_dbTable = "notifications";

Notification::Notification() {
    _id = 0;
    _read = false;
}

Notification::Notification(unsigned int id) {
    map<string, string> data = BaseModel::getById(_dbTable, id);

    if (!data.empty())
    {
        _id = id;
        _message = data["message"];
        _userId = (unsigned) stoi(data["userid"]);
        _read = (bool) stoi(data["read"]);
    }
    else
    {
        throw invalid_argument("Please enter a valid ID");
    }
}

Notification::Notification(string message, unsigned int userId) :
_message(message), _userId(userId) {
    _id = 0;
}

unsigned int Notification::getId() {
    return _id;
}

void Notification::setId(unsigned int id) {
    _id = id;
}

std::string Notification::getMessage() {
    return _message;
}

void Notification::setMessage(std::string message) {
    _message = message;
}

unsigned int Notification::getUserId() {
    return _userId;
}
```

```
void Notification::setUserId(unsigned int userId) {
    _userId = userId;
}

bool Notification::getRead() {
    return _read;
}

void Notification::setRead(bool read) {
    _read = read;
}

bool Notification::save() {
    int res = BaseModel::save(_dbTable, {
            {"id", {to_string(_id), "int"}},
            {"message", {_message, "string"}},
            {"userid", {to_string(_userId), "int"}},
            {"read", {to_string((int) _read), "int"}},
    });

    if (_id == 0)
    {
        _id = res["id"];
    }

    return (bool) res;
}

bool Notification::remove() {
    return BaseModel::remove(_dbTable, _id);
}

std::ostream& operator<< (std::ostream& stream, const Notification&
notification) {
    stream << notification._message << endl;

    return stream;
}

std::istream& operator>> (std::istream& stream, Notification& notification) {
    cout << "Message:" << endl;
    stream.ignore(1, '\n');
    getline(stream, notification._message, '\n');

    return stream;
}
```

Database diagram



users	
id	integer
name	text
surname	text
phone	text
birthdate	text
country	text
house_number	integer
postal_code	text
town	text
street	text
isadmin	integer
password	text
isclient	integer
isadvisor	integer
id_advisor	integer

loans	
id	integer
creation	text
validation	text
sender	integer
approved	integer
amount	integer
advisor_id	integer

contact	
id	integer
id_client	integer
id_advisor	integer
message	text
read	integer
date	text

orders	
id	integer
creation	text
sent	integer
type	text
user_id	integer
account	integer

transactions	
id	integer
account	integer
date	text
amount	integer
description	text

bank_account	
id	integer
SWIFT	text
BIC	text
id_user	integer
balance	integer

notifications	
id	integer
message	text
userid	integer
read	integer

References

C++ sha256 function :: zedwood.com. 2016. *C++ sha256 function :: zedwood.com*. [ONLINE] Available at: http://www.zedwood.com/article/cpp-sha256-function. [Accessed 24 May 2016].

GitHub. 2016. *GitHub - AdrienPoupa/mediatheque-cpp: Projet Efrei C++*. [ONLINE] Available at: https://github.com/AdrienPoupa/mediatheque-cpp. [Accessed 24 May 2016].

GitHub. 2016. *GitHub - SRombauts/SQLiteCpp: SQLiteC++ (SQLiteCpp) is a smart and easy to use C++ SQLite3 wrapper.*. [ONLINE] Available at: https://github.com/SRombauts/SQLiteCpp. [Accessed 24 May 2016].

GitHub. 2016. *GitHub - meganz/mingw-std-threads: Standard threads implementation currently still missing on MinGW GCC on Windows*. [ONLINE] Available at: https://github.com/meganz/mingw-std-threads. [Accessed 24 May 2016].

Marking Scheme

Student Name (ID): ___________________________________ Total: _________ / 100%

	Fail	Marginal Fail	Pass	Credit	Distinction
Design (40%)	**0-15**	**16-19**	**20-25**	**26-29**	**30-40**
	• Minimal understanding of program design • Poor illustration of program design using UML • Major / obvious errors / omissions in UML diagrams • Incomplete design missing major diagrams	• Design solution covers less than half of the basic requirements of the system • Minimal to non-application of object oriented concepts in the design	• Some understanding of the program design • Design solution covers most of the basic requirements of the system • Some errors / omissions in UML diagrams • Completed all the major UML diagrams (Use Case, Activity and Class) with missing some diagrams • Appropriate and basic object oriented concepts applied in the design.	• Good understanding of the program design • Design solution covers all the basic requirements of the system • Minor errors / omissions in UML diagrams • No missing UML diagrams • Appropriate and advanced object oriented concepts applied in the design.	• Excellent understanding of the program design • Detailed design solution covers all the requirements of the system • No missing UML diagrams with hardly any errors / omissions • Appropriate and advanced object oriented concepts applied in the design. • One suitable design patterns applied
Working Prototype (30%)	**0-11**	**12-14**	**15-19**	**20-22**	**23-30**
	• Poor solution with minimal to non relations with the scenario • Unresolved compilation errors or able to execute with major errors • The coding solution lacks proper structure	• Little or no mapping between design and solution • Implemented less than half of the basic requirements identified in the design • Implemented suitable data structures/algorithms in the solution with more errors	• Appropriate solution with relations to the scenario • Prototype – able to compile and execute with some errors • Implemented all of the basic requirements identified in the design (use case) • Implemented most of the object oriented concepts stated in the design with good mapping between design and solution. • Implemented suitable data structures/algorithms in the solution with few errors	• Good solution with relations to the scenario • Prototype – able to compile and execute with minor errors • Implemented most of the requirements identified in the design • Implemented most of the object oriented concepts stated in the design with Good mapping between design and solution • Implemented suitable data structures/algorithms in the solution with no errors.	• Excellent solution with relations to the scenario • Prototype – able to compile and execute with no error • Implemented all the requirements identified in the design • Implemented all of the object oriented concepts stated in the design • Excellent mapping between design and solution • Design pattern(s_ stated in the design has been applied. • Implemented suitable data structures/algorithms in the solution without errors and design fault.

	Fail	Marginal Fail	Pass	Credit	Distinction
	0 – 11	**12-14**	**15 - 19**	**20-22**	**23-30**
Critical Evaluation and Documentation (30%)	• Incomplete documentation submitted • No appraisal of the solution • No referencing • Bad structure and presentation	• Brief justification on the application of object oriented concepts • Missing major components within the document • No or illogical assumptions made	• Completed all the major components within the document • Basic structure and presentation • Did some referencing, which adhere to Harvard Name Referencing but with some errors • Acceptable justification done on object oriented concepts applied. • Brief appraisal done on the solution • Brief assumptions made	• Good structure, presentation and standards • No missing components within the document • Adhered to Harvard Name Referencing standards but with minor errors / omissions • Good justification done on object oriented concepts applied. • Acceptable appraisal done on the solution • Acceptable justification of the design pattern applied • Suitable assumptions made	• Professional standards with excellent structure and presentation • A complete documentation submitted • Adhered to Harvard Name Referencing standards with no errors / omissions • Detailed justification done on object oriented concepts applied. • Detailed appraisal done on the solution • Detailed justification on the choice pattern applied in the solution. • Good assumptions made

Remarks:

____

____

____