OOP Assignment 2

Group Report

Group UT1

Chad Simpson, Luke Ward, Shaun Webb

***Question 1 :***

**a)**

It is possible for the canWithdraw function to be inherited by the ChildAccount class, due to the fact that it is a child of the SavingsAccount class, which is a child of the BankAccount class.

One of the possible problems that would arise with using the canWithdraw function in the ChildAccount class is that it has a minimumBalance due to the fact that it is a child of the SavingsAccount. The BankAccount version of the canWithdraw function will not be taking into account the data member minimumBalance of the SavingsAccount classes, which then opens the possibility of withdrawing below the minimumAmount.

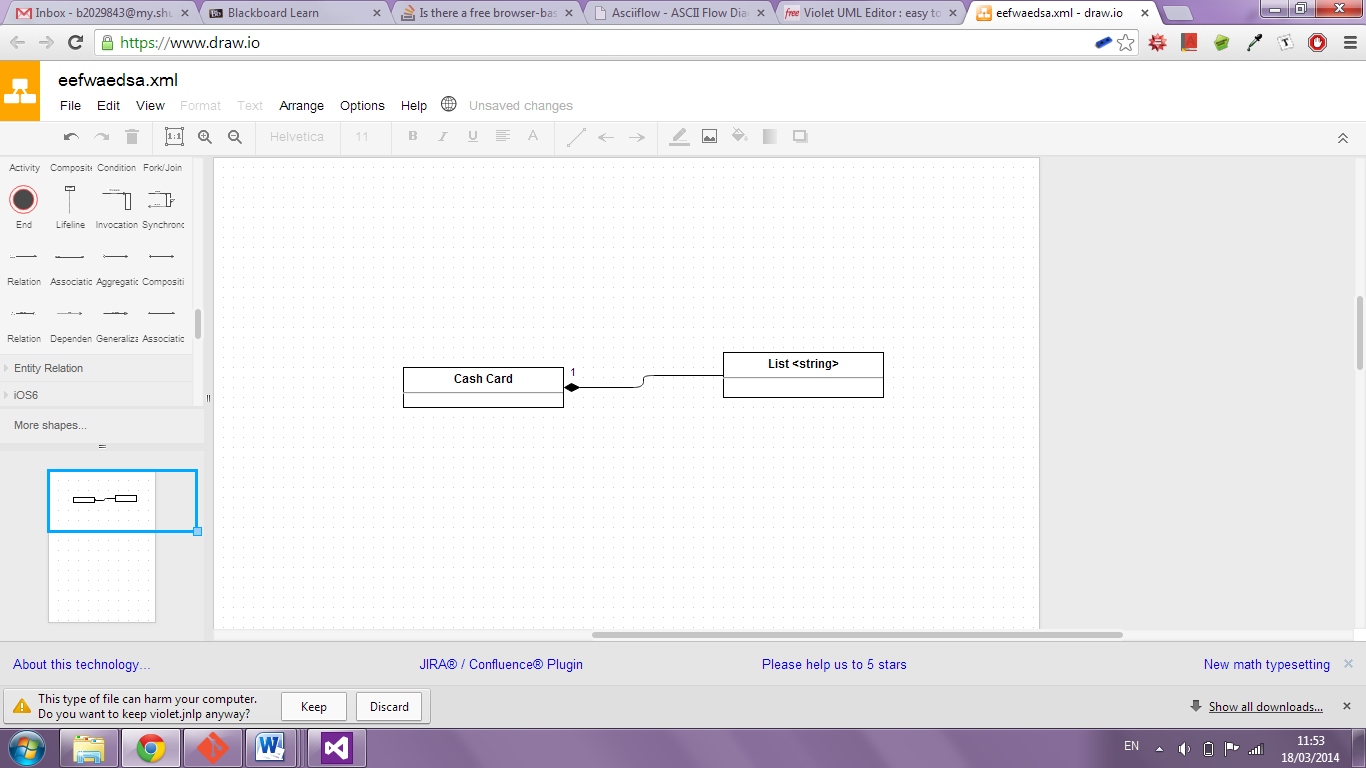
A fix for this issue would be for the ChildAccount to define its own version of the canWithdraw function, and to apply the “virtual” keyword on the BankAccount version.

**b)**

In this system the CashPoint class does not need to create a whole new instance of the BankAccount class, instead the CashPoint only needs indirect access to the BankAccount class via a pointer. This allows the CashPoint to access the already existing BankAccounts. An example of this would be if you had several different types of accounts, a pointer would be able to hold all of the BankAccount types, instead of having to recreate objects every time you wish to use a different type.

**c)**

The relationship between the CashCard class and the List<string> classes is composition. The C++ mechanism that is involved in the implementation of it is that the List<string> is declared inside of the CashCard class.



**d)**

The UserInterface class is not an abstract class; this is due to the lack of pure virtual functions within the class. Abstract classes are useful as base classes only, and must have at least one pure virtual function present.

The only situation in which it would be appropriate for UserInterface to be an abstract class is if the end product needed different GUIs for different platforms, such as mobile and PC. In this case, you would need UserInterface as a base abstract class which the other UI classes could use as a parent class.

For the purpose of this program however, the only use will be on the ATM platform exclusively, negating the need for having UserInterface as an abstract class.

**e)**

In this program the function Date::currentDate() is required to be accessed outside of its class. Declaring the function as static allows currentDate to be called without using a class member object. In this program this is useful as it allows the current date to be accessed without creating a new Date object.

**f)**

***Line 1***: Time t (t1 + Time(0, 0, 12));

Which functions are called?

* *Basic Constructor –* The line calls the basic constructor for the Time class, using the three integers supplied in the arguments.
* *Addition Operator – T*he first line is also using the addition operator to add two Time objects together (t1 and (Time(0, 0, 12)).
* *Copy Constructor –* The result of the addition is then used by copy constructor which is called by Time t.

Issues – None.

***Line 2***: Time t (Time(12) + t1);

Which functions are called?

* *Conversion Constructor* – First of all, the second line calls the conversion constructor in Time(12).
* *Addition Operator* – Similarly to the first line, it is also using the addition constructor to add the two Time objects together.
* *Copy Constructor* – The second line lastly calls the copy constructor on the two Time objects, creating Time t from the sum of their products.

Issues – None.

***Line 3***: Time t (t1 + 12);

Which functions are called?

* *Addition Operator –* The addition operator is called on the two arguments of the constructor for Time t. The reason why it is possible to add an integer to a Time object in this manner is because the copy constructor called will first convert the time in t1 to an integer via the inSeconds function, which then allows the second parameter to be added.
* *Copy Constructor –* Lastly, line 3 calls the copy constructor on the product of the two parameters.

Issues – None.

***Line 4***: Time t (12 + t1);

Which functions are called?

* *Addition Operator –* The addition operator is called on the two parameters of the constructor of Time t. In this case, it will not compile due to the fact that it is trying to add an integer to a Time object without first converting the Time object to an integer.
* *Copy Constructor –* Time t tries to call the copy constructor on the two parameters it has been given.

Issues – Due to the fact that the addition operator adds from left to right, when it tries to add the integer to the Time first rather than trying to add the Time to the integer, it turns out to be a type mismatch.

**g)**

The following method would not work correctly, this is due to the line olderTransactions().deleteGivenTransaction(tr). The function olderTransaction returns a constant TransactionList object as part of its method. As this is constant the function deleteGivenTransaction cannot be used to change the object.

Even if the previous line could work the function as a whole would still be unsuccessful, this is due to olderTransaction returning a copy transaction list. These changes are not then applied to the main transaction list.

**h)**

Having BankAccount::prepareFormattedAccountDetails as virtual would allow the function to be dynamically binded to different types at run time. In this case this would be useful for the different types of child bank accounts to output their details in the same format using the same parent function.

**i)**

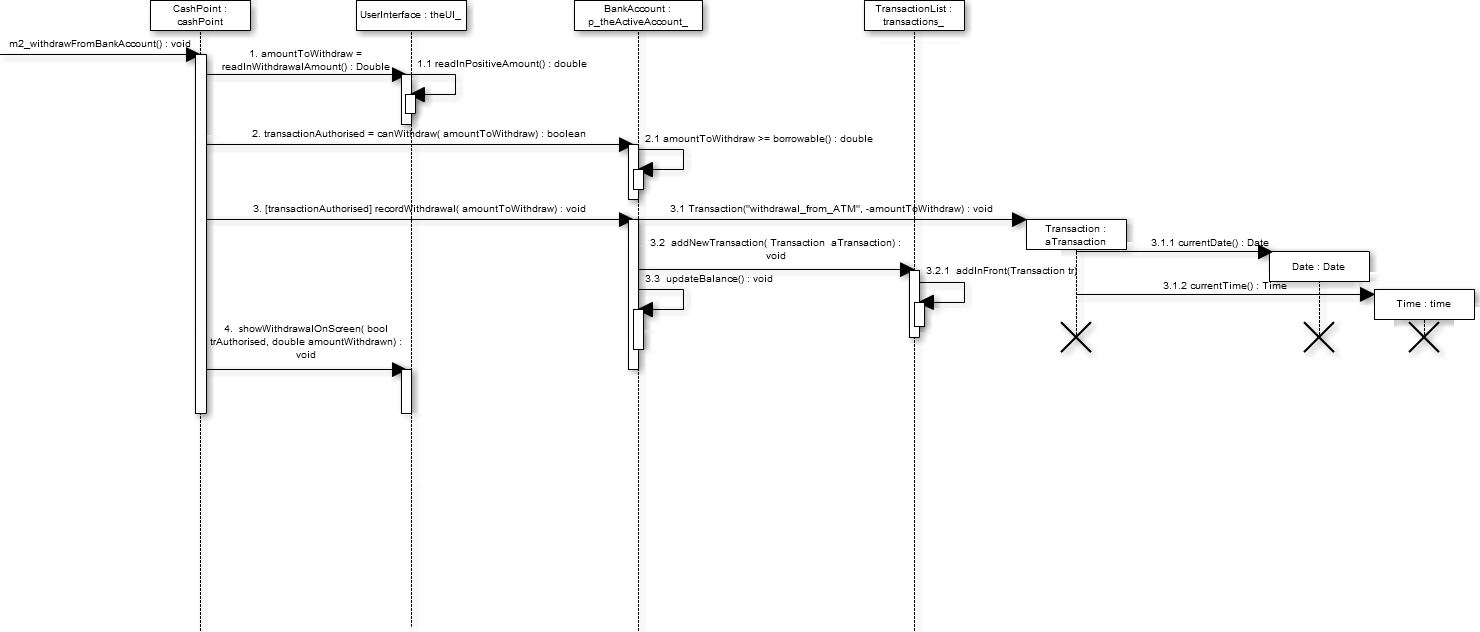
This expression would not work due to p\_theActiveAccount\_ being a pointer of type BankAccount while pointing to an object of the derived class CurrentAccount.

The getOverdraftLimit function is a method of the currentAccount class, due to the hierarchy the BankAccount pointer would be unable to access the function of the derived class.

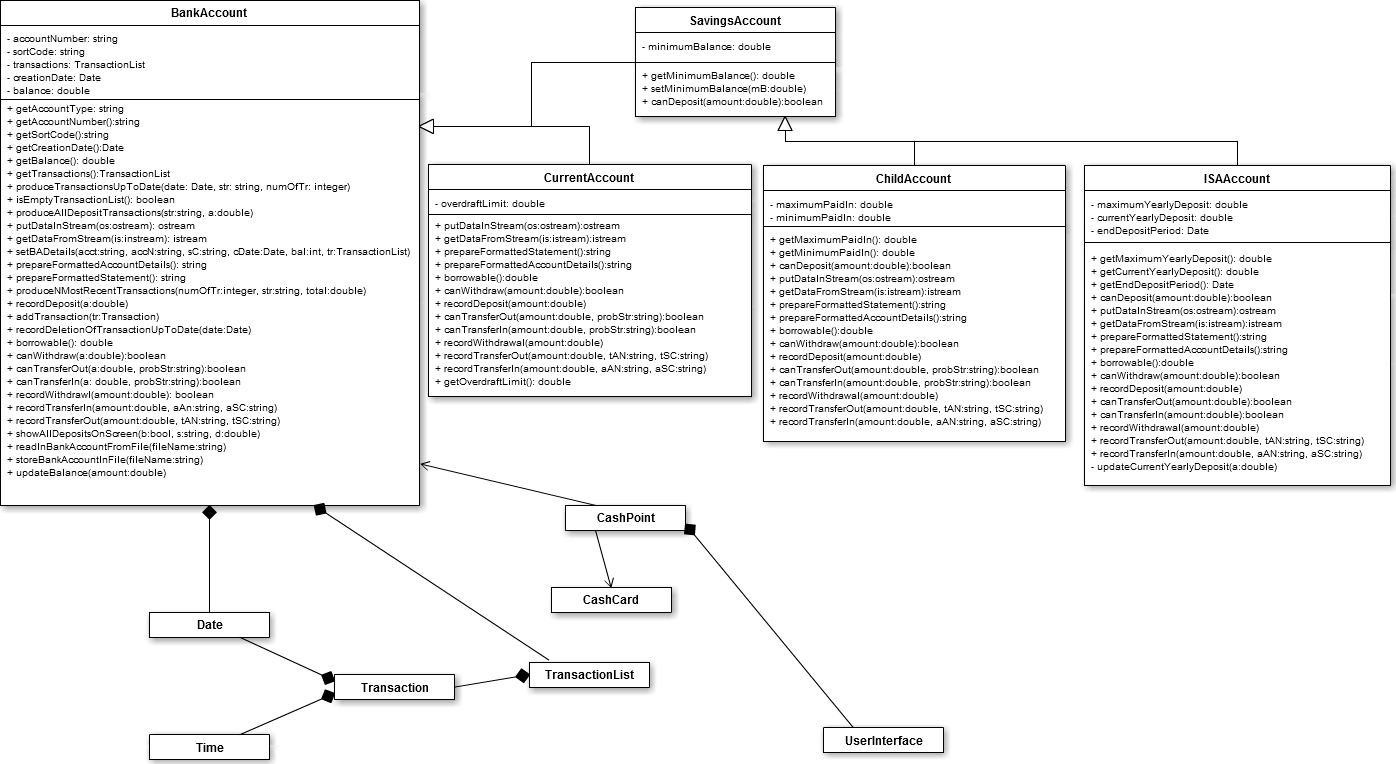
**j)**

This function could not be written as such, the line p\_theActiveAccount->balance\_ would not work as balance is a private data member of the class BankAccount. Instead of accessing the data member in this style the p\_theActiveAccount should use an accessor function such as getBalance().

**2 a)**



**2 b)**



**Appendix 3 Functions**

**Option 6:**

void CashPoint::m6\_showMiniStatement() const

{

bool noTransaction = p\_theActiveAccount\_->isEmptyTransactionList();

int numOfTr = 0;

double total = 0.0;

string str = "";

if (!noTransaction)

{

numOfTr = p\_theUI\_->readInNumberOfTransactions();

p\_theActiveAccount\_->produceNMostRecentTransactions(numOfTr, str, total);

}

str = p\_theActiveAccount\_->prepareFormattedAccountDetails() + "\n" + str;

p\_theUI\_->showMiniStatementOnScreen(numOfTr, str, total);

}

void const BankAccount::produceNMostRecentTransactions(int numOfTr, string& str, double& total) const

{

TransactionList trl = transactions\_.getMostRecentTransactions(numOfTr);

total = trl.getTotalTransactions();

str = trl.toFormattedString();

}

TransactionList TransactionList::getMostRecentTransactions(int numOfTr) const

{

list<Transaction> copy(listOfTransactions\_);

TransactionList temp;

for (int i = 0; i < numOfTr; ++i)

{

if (copy.size() > 0)

{

temp.addTransaction(copy.front());

copy.pop\_front();

}

}

return temp;

}

double TransactionList::getTotalTransactions() const

{

double total = 0.0;

TransactionList copy(\*this);

for(int i = 0 ; i < size() ; ++i)

{

total += copy.newestTransaction().getValue<double>();

copy.deleteFirstTransaction();

}

return total;

}

**Option 7: Several point of this section have been implemented differently to be able to implement templates.**

void CashPoint::m7\_searchForTransactions() const

{

bool noTransaction = p\_theActiveAccount\_->isEmptyTransactionList();

if(noTransaction)

p\_theUI\_->showNoTransactionsOnScreen();

else

searchTransactions();

}

void CashPoint::searchTransactions() const

{

double amount;

string title;

string date;

p\_theUI\_->showSearchMenu();

int opt = p\_theUI\_->readInSearchCommand();

switch(opt)

{

case 1:

amount = p\_theUI\_->readInSearchAmount<double>();

p\_theUI\_->showMatchingTransactionsOnScreen(m7a\_showTransactionsForAmount(amount));

break;

case 2:

title = p\_theUI\_->readInSearchAmount<string>();

p\_theUI\_->showMatchingTransactionsOnScreen(m7a\_showTransactionsForAmount(title));

break;

case 3:

Date date = p\_theUI\_->readInSearchAmount<Date>(p\_theActiveAccount\_->getCreationDate());

p\_theUI\_->showMatchingTransactionsOnScreen(m7a\_showTransactionsForAmount(date));

break;

}

}

template <typename T> string m7a\_showTransactionsForAmount(T amount) const

{

ostringstream os;

TransactionList results(p\_theActiveAccount\_->getTransactions().getTransactionsForAmount(amount));

if (!results.size() == 0)

os << "\n\n" << results.size() << " TRANSACTIONS FOUND\n" << results.toFormattedString(); //one per line

else

os << "\n\nNO TRANSACTION IN BANK ACCOUNT MATCH THE SEARCH CRITERION GIVEN!";

return (os.str());

}

template <typename T> TransactionList produceTransactionsForAmount(const T amount) const

{

TransactionList copy(\*this);

TransactionList temp;

for(int i = 0 ; i < (\*this).size() ; ++i)

{

if(copy.size() > 0)

{

if(copy.newestTransaction().getValue<T>() == amount)

{

temp.addTransaction(copy.newestTransaction());

}

copy.deleteFirstTransaction();

}

}

return temp;

}

TransactionList TransactionList::getTransactionsUpToDate(const Date& date, TransactionList temp)

{

if(temp.size() <= 0)

{

TransactionList dummyTrl;

return dummyTrl;

}

else

if(temp.newestTransaction().getValue<Date>() <= date)

return temp;

else

{

temp.deleteFirstTransaction();

TransactionList test(getTransactionsUpToDate(date, temp));

return test;

}

}

void UserInterface::showMatchingTransactionsOnScreen( const string& results) const {

cout << "\n\_\_\_\_\_\_\_\_ SEARCH RESULTS \_\_\_\_\_";

cout << results;

}

**Options 8:**

void CashPoint::m8\_clearTransactionsUpToDate() const

{

bool noTransaction = p\_theActiveAccount\_->isEmptyTransactionList();

bool deletionConfirmed;

Date d, cd;

int numOfTr = 0;

string str = "";

if(!noTransaction)

{

cd = p\_theActiveAccount\_->getCreationDate();

d = p\_theUI\_->readInValidDate(cd);

p\_theActiveAccount\_->produceTransactionsUpToDate(d, str, numOfTr);

}

if(numOfTr > 0)

p\_theUI\_->showTransactionsUpToDateOnScreen(noTransaction, d, numOfTr, str);

else

p\_theUI\_->showNoTransactionsUpToDateOnScreen(d);

if(!noTransaction && str != "")

deletionConfirmed = p\_theUI\_->readInConfirmDeletion();

if(!noTransaction && str != "" && deletionConfirmed)

p\_theActiveAccount\_->recordDeletionOfTransactionUpToDate(d);

if(!noTransaction && str != "")

p\_theUI\_->showDeletionOfTransactionUpToDateOnScreen(d, numOfTr, deletionConfirmed);

}

bool Date::isValidDate(const Date& creationDate) const

{

if((\*this <= currentDate()) && (\*this >= creationDate))

if(day\_ > 0)

if(day\_ < 28 && month\_ == 2)

return true;

else if(day\_ < 30 && (month\_ == 4 || month\_ == 6 || month\_ == 9 || month\_ == 11))

return true;

else if(day\_ < 31 && (month\_ == 1 || month\_ == 3 || month\_ == 5 || month\_ == 7 || month\_ == 10 || month\_ == 12))

return true;

return false;

}

TransactionList TransactionList::getTransactionsUpToDate(const Date& date, TransactionList temp)

{

if(temp.size() <= 0)

{

TransactionList dummyTrl;

return dummyTrl;

}

else

if(temp.newestTransaction().getValue<Date>() <= date)

return temp;

else

{

temp.deleteFirstTransaction();

TransactionList test(getTransactionsUpToDate(date, temp));

return test;

}

}

void TransactionList::deleteTransactionsUpToDate(const Date& date)

{

if(size() <= 0)

{

return;

}

else

{

if (newestTransaction().getValue<Date>() <= date)

{

\*this = olderTransactions();

this->deleteTransactionsUpToDate(date);

}

else

{

Transaction firstTr(newestTransaction());

this->deleteFirstTransaction();

this->deleteTransactionsUpToDate(date);

this->addNewTransaction(firstTr);

}

}

}

**Option 9:**

void CashPoint::m9\_transferCashToAnotherAccount()

{

string accNo = "";

string sortNo = "";

p\_theUI\_->showCardOnScreen(p\_theCashCard\_->toFormattedString());

string bankAccFileName = p\_theUI\_->readInAccountToBeProcessed(accNo, sortNo);

int validAccountCode = validateAccount(bankAccFileName);

p\_theUI\_->showValidateAccountOnScreen(validAccountCode, accNo, sortNo);

if(validAccountCode == 0)

{

p\_theTransferAccount\_ = activateBankAccount(bankAccFileName);

attemptTransfer(p\_theTransferAccount\_);

releaseBankAccount(p\_theTransferAccount\_, bankAccFileName);

}

}

int CashPoint::validateAccount( const string& bankAccountFileName) const {

//check that the account is valid

int validBankCode;

//The account is already open

if (p\_theActiveAccount\_ != nullptr)

if (p\_theActiveAccount\_->getFileName() == bankAccountFileName)

return ACCOUNT\_ALREADY\_OPEN;

if (!canOpenFile(bankAccountFileName))

//account does not exist

validBankCode = UNKNOWN\_ACCOUNT;

else

//unaccessible account (exist but not listed on card)

if (!p\_theCashCard\_->onCard(bankAccountFileName))

validBankCode = UNACCESSIBLE\_ACCOUNT;

else

//account valid (exists and accessible)

validBankCode = VALID\_ACCOUNT;

return validBankCode;

}

BankAccount\* CashPoint::activateBankAccount( const string& aBAFileName) {

//check the type of the account (already checked for validity)

int accType( checkAccountType( aBAFileName));

//effectively create the active bank account instance of the appropriate class

//& store the appropriate data read from the file

BankAccount\* p\_BA( nullptr);

switch( accType)

{

case CURRENTACCOUNT\_TYPE:

cout << "\n-------CURRENT-------\n";

p\_BA = new CurrentAccount;

p\_BA->readInBankAccountFromFile( aBAFileName);

return p\_BA;

break;

case CHILDACCOUNT\_TYPE:

cout << "\n-------CHILD-------\n";

p\_BA = new ChildAccount;

p\_BA->readInBankAccountFromFile( aBAFileName);

return p\_BA;

break;

case ISAACCOUNT\_TYPE:

cout << "\n-------ISA-------\n";

p\_BA = new ISAAccount;

p\_BA->readInBankAccountFromFile( aBAFileName);

return p\_BA;

break;

}

//use dynamic memory allocation: the bank account created will have to be released in releaseBankAccount

return p\_BA;

}

void CashPoint::attemptTransfer(BankAccount\* p\_theTransferAccount\_)

{

string trInProblemStr = "NO ERROR";

string trOutProblemStr = "NO ERROR";

double transferAmount = p\_theUI\_->readInTransferAmount();

bool trOutOK = p\_theActiveAccount\_->canTransferOut(transferAmount, trOutProblemStr);

bool trInOK = p\_theTransferAccount\_->canTransferIn(transferAmount, trInProblemStr);

if(trOutOK && trInOK)

recordTransfer(transferAmount, p\_theTransferAccount\_);

p\_theUI\_->showTransferOnScreen(trOutOK, trInOK, transferAmount, trInProblemStr, trOutProblemStr);

}

void CashPoint::recordTransfer(const double& transferAmount, BankAccount\* p\_theTransferAccount\_)

{

string tAN = p\_theTransferAccount\_->getAccountNumber();

string tSC = p\_theTransferAccount\_->getSortCode();

p\_theActiveAccount\_->recordTransferOut(transferAmount, tAN, tSC);

string aAN = p\_theActiveAccount\_->getAccountNumber();

string aSC = p\_theActiveAccount\_->getSortCode();

p\_theTransferAccount\_->recordTransferIn(transferAmount, aAN, aSC);

}

void CurrentAccount::recordTransferIn(const double& amount, const string& aAN, const string& aSC)

{

Transaction transferTransaction( "transfer\_from\_ACC\_" + aAN + "\_" + aSC, amount);

addTransaction(transferTransaction);

updateBalance(amount);

}

**Header Files**

**Cashpoint.h**

#ifndef CashPointH

#define CashPointH

//---------------------------------------------------------------------------

//CashPoint: class declaration

//---------------------------------------------------------------------------

#include "CashCard.h"

#include "BankAccount.h"

#include "CurrentAccount.h"

#include "ChildAccount.h"

#include "ISAAccount.h"

#include "UserInterface.h"

#include <fstream>

#include <string>

using namespace std;

class CashPoint {

public:

//constructors & destructor

CashPoint(); //default constructor

~CashPoint(); //destructor

void activateCashPoint();

private:

//data items

BankAccount\* p\_theActiveAccount\_;

BankAccount\* p\_theTransferAccount\_;

CashCard\* p\_theCashCard\_;

const UserInterface\* p\_theUI\_;

//support functions

void performCardCommand( int);

void performAccountProcessingCommand( int);

int validateCard( const string&) const;

int validateAccount( const string&) const;

void processOneCustomerRequests();

void processOneAccountRequests();

void attemptTransfer(BankAccount\* p\_theTransferAccount\_);

void recordTransfer(const double& transferAmount, BankAccount\* p\_theTransferAccount\_);

void performSubMenuCommand( int);

//commands

//option 1

void m1\_produceBalance() const;

//option 2

void m2\_withdrawFromBankAccount();

//option 3

void m3\_depositToBankAccount();

//option 4

void m4\_produceStatement() const;

//option 5

void m5\_showAllDepositTransactions() const;

//option 6

void m6\_showMiniStatement() const;

//option 7

void m7\_searchForTransactions() const;

//option 8

void m8\_clearTransactionsUpToDate() const;

//option 9

void m9\_transferCashToAnotherAccount();

//current account functions

void requestOverdraftLimit();

//savings account functions

void requestMinimumBalance();

//child account functions

void requestDepositConstraints();

//isa account functions

void requestIsaDetails();

//support file handling functions & creation of dynamic objects

template <typename T> string m7a\_showTransactionsForAmount(T amount) const

{

ostringstream os;

TransactionList results(p\_theActiveAccount\_->getTransactions().produceTransactionsForAmount(amount));

if (!results.size() == 0)

os << "\n\n" << results.size() << " TRANSACTIONS FOUND\n" << results.toFormattedString(); //one per line

else

os << "\n\nNO TRANSACTION IN BANK ACCOUNT MATCH THE SEARCH CRITERION GIVEN!";

return (os.str());

}

void searchTransactions() const;

bool canOpenFile( const string&) const;

int checkAccountType( const string&) const;

bool linkedCard( string cashCardFileName) const;

BankAccount\* activateBankAccount( const string&);

BankAccount\* releaseBankAccount( BankAccount\*, string);

void activateCashCard( const string&);

void releaseCashCard();

};

#endif

**UserInterface.h**

#ifndef UserInterfaceH

#define UserInterfaceH

//---------------------------------------------------------------------------

//UserInterface: class declaration

//---------------------------------------------------------------------------

#include "constants.h"

#include "Date.h"

#include <iomanip>

#include <iostream>

#include <string>

using namespace std;

class UserInterface {

public:

//constructors & destructor

static UserInterface\* getInstance(){

static UserInterface instance;

return &instance;

}

void showWelcomeScreen() const;

void showByeScreen() const;

int readInCardIdentificationCommand() const;

int readInAccountProcessingCommand(const string& accType) const;

void showErrorInvalidCommand() const;

void wait() const;

const string readInCardToBeProcessed( string& cardNum) const;

void showValidateCardOnScreen( int validCode, const string& cardNum) const;

void showCardOnScreen( const string& cardSt) const;

const string readInAccountToBeProcessed( string& aNum, string& sCod) const;

void showValidateAccountOnScreen( int valid, const string& aNum, const string& sCod) const;

double readInWithdrawalAmount() const;

double readInDepositAmount() const;

int readInNumberOfTransactions() const;

bool readInConfirmDeletion() const;

double readInTransferAmount() const;

void showProduceBalanceOnScreen( double bal) const;

void showDepositOnScreen( bool auth, double deposit) const;

void showWithdrawalOnScreen( bool auth, double withdrawal) const;

void showStatementOnScreen( const string&) const;

void showMiniStatementOnScreen(const int&, const string&, const double&) const;

void showAllDepositsOnScreen(bool noTransaction, string str, double total) const;

void showDeletionOfTransactionUpToDateOnScreen(const Date& date, const int& numOfTr, const bool& deletionConfirmed) const;

void showNoTransactionsUpToDateOnScreen(const Date& date) const;

void showTransactionsUpToDateOnScreen(const bool& noTransactions, const Date& d, const int& numOfTr, const string& str) const;

void showTransferOnScreen(const bool& trOutOK, const bool& trInOK, const double& transferAmount, const string& InprobStr, const string& OutProbStr) const;

void showOverdraftLimitOnScreen(const double& oD) const;

void showMinimumBalanceOnScreen(const double& mB) const;

void showDepositConstraintsOnScreen(const double& minPI, const double& maxPI) const;

void showIsaDetailsOnScreen(const double& maxYD, const double& currYD, const Date& eOD) const;

Date readInValidDate(const Date& date) const;

void showNoTransactionsOnScreen() const;

void showSearchMenu() const;

int readInSearchCommand() const;

template <typename T> T readInSearchAmount() const

{

T amount;

cout << "\n ENTER SEARCH AMOUNT: ";

cin >> amount;

return amount;

//return searchAmount;

}

template <typename T> Date readInSearchAmount(T cr) const

{

int day,month,year;

Date searchDate;

do{

cout << "\n ENTER VALID DATE: ";

cout << "\n ENTER SEARCH DAY: ";

cin >> day;

cout << "\n ENTER SEARCH MONTH: ";

cin >> month;

cout << "\n ENTER SEARCH YEAR: ";

cin >> year;

searchDate = Date(day, month, year);

} while (!searchDate.isValidDate(cr));

return searchDate;

}

void showMatchingTransactionsOnScreen( const string& results) const;

private:

//support functions

UserInterface(){}

UserInterface(const UserInterface &){}

UserInterface& operator=(const UserInterface &){}

void showCardIdentificationMenu() const;

void showAccountProcessingMenu(const string& accType) const;

int readInCommand() const;

double readInPositiveAmount() const;

};

#endif

**TransactionList.h**

#ifndef TransactionListH

#define TransactionListH

//---------------------------------------------------------------------------

//TransactionList: class declaration

//---------------------------------------------------------------------------

//#include "ListT.h"

#include "Transaction.h"

#include <cassert> // for assert()

#include <sstream>

#include <list>

class TransactionList {

public:

void addNewTransaction( const Transaction&);

const Transaction newestTransaction() const;

const TransactionList olderTransactions() const;

void deleteFirstTransaction();

void deleteGivenTransaction( const Transaction&);

int size() const;

TransactionList getAllDepositTransactions();

double getTotalTransactions() const;

TransactionList getMostRecentTransactions(int numOfTr) const;

TransactionList getTransactionsUpToDate(const Date& date, TransactionList temp);

void deleteTransactionsUpToDate(const Date& date);

void addTransaction(const Transaction tr);

template <typename T> TransactionList produceTransactionsForAmount(const T amount) const

{

TransactionList copy(\*this);

TransactionList temp;

for(int i = 0 ; i < (\*this).size() ; ++i)

{

if(copy.size() > 0)

{

if(copy.newestTransaction().getValue<T>() == amount)

{

temp.addTransaction(copy.newestTransaction());

}

copy.deleteFirstTransaction();

}

}

return temp;

}

const string toFormattedString() const; //return transactionlist as a (formatted) string

ostream& putDataInStream( ostream& os) const; //send TransactionList info into an output stream

istream& getDataFromStream( istream& is); //receive TransactionList info from an input stream

private:

list<Transaction> listOfTransactions\_; //list of transactions

};

//---------------------------------------------------------------------------

//non-member operator functions

//---------------------------------------------------------------------------

ostream& operator<<( ostream&, const TransactionList&); //insertion operator

istream& operator>>( istream& is, TransactionList& trl); //extraction operator

#endif

**Date.h**

#ifndef DateH

#define DateH

//---------------------------------------------------------------------------

//Date: class declaration

//---------------------------------------------------------------------------

#include <ctime> // for date functions

#include <iomanip>

#include <iostream>

#include <string>

#include <sstream>

using namespace std;

class Date {

public:

Date(); //default constructor

Date( int, int, int); //constructor

int getDay() const; //return a data member value, day\_

int getMonth() const; //return a data member value, month\_

int getYear() const; //return a data member value, year\_

void setDate( int, int, int); //set new values for date

static const Date currentDate() ; //return the current date

string toFormattedString() const ; //return date as formatted string ("DD/MM/YYYY")

bool isValidDate(const Date& cd) const;

ostream& putDataInStream( ostream& os) const; //send Date info into an output stream

istream& getDataFromStream( istream& is); //receive Date info from an input stream

bool operator==( const Date& d) const; //true if (\*this == d)

bool operator!=( const Date& d) const; //true if (\*this != d)

bool operator<( const Date& d) const; //true if (\*this < d) (strictly earlier)

bool operator<=(const Date& d) const;

bool operator>=(const Date& d) const;

bool operator>(const Date& d) const;

private:

int day\_;

int month\_;

int year\_;

};

//---------------------------------------------------------------------------

//non-member operator functions

//---------------------------------------------------------------------------

ostream& operator<<( ostream&, const Date&); //output operator

istream& operator>>( istream&, Date&); //input operator

#endif

**BankAccount.h**

#ifndef BankAccountH

#define BankAccountH

//---------------------------------------------------------------------------

//BankAccount: class declaration

//---------------------------------------------------------------------------

#include "TransactionList.h"

#include <fstream>

using namespace std;

class BankAccount {

public:

//constructors & destructor

BankAccount();

BankAccount( const string& typ, const string& acctNum, const string& sCode,

const Date& cD, double b,

const TransactionList& trList);

virtual ~BankAccount();

//getter (assessor) functions

const string getAccountType() const;

const string getAccountNumber() const;

const string getSortCode() const;

const Date getCreationDate() const;

double getBalance() const;

const TransactionList getTransactions() const;

const void produceTransactionsUpToDate(const Date& date, string& str, int& numOfTr);

bool isEmptyTransactionList() const;

void produceAllDepositTransactions(string&, double&);

//functions to put data into and get data from streams

virtual ostream& putDataInStream( ostream& os) const = 0;

virtual istream& getDataFromStream( istream& is) = 0;

void setBADetails(const string& accT, const string& accN, const string& sC, const Date& cDate, const int& bal, const TransactionList& tr);

//other operations

virtual const string prepareFormattedAccountDetails() const = 0;

virtual const string prepareFormattedStatement() const = 0;

virtual const void produceNMostRecentTransactions(int numOfTr, string& str, double& total) const;

virtual void recordDeposit( double amount);

void addTransaction(Transaction tr);

void recordDeletionOfTransactionUpToDate(const Date& date);

virtual double borrowable() const = 0;

virtual bool canWithdraw( double amount) const = 0;

virtual bool canTransferOut(double amount, string& probStr) const = 0;

virtual bool canTransferIn(double amount, string& probStr) const = 0;

virtual void recordWithdrawal( double amount) = 0;

virtual void recordTransferIn(const double& amount, const string& aAN, const string& aSC) = 0;

virtual void recordTransferOut(const double& amount, const string& tAN, const string& tSC) = 0;

void readInBankAccountFromFile( const string& fileName);

void storeBankAccountInFile( const string& fileName) const;

void updateBalance( double amount);

void setFileName(const string& fName);

string getFileName();

//templates

private:

//data items

string accountType\_;

string accountNumber\_;

string sortCode\_;

Date creationDate\_;

string fileName;

double balance\_;

TransactionList transactions\_;

};

//---------------------------------------------------------------------------

//non-member operator functions

//---------------------------------------------------------------------------

ostream& operator<<( ostream&, const BankAccount&); //output operator

istream& operator>>( istream&, BankAccount&); //input operator

#endif

**CurrentAccount.h**

#ifndef CurrentAccountH

#define CurrentAccountH

#include "BankAccount.h"

using namespace std;

class CurrentAccount: public BankAccount

{

public:

CurrentAccount();

~CurrentAccount();

double getOverdraftLimit() const;

ostream& putDataInStream( ostream& os) const;

istream& getDataFromStream( istream& is);

const string prepareFormattedStatement() const;

const string prepareFormattedAccountDetails() const;

double borrowable() const;

bool canWithdraw( double amount) const;

void recordDeposit( double amount);

bool canTransferOut(double amount, string& probStr) const;

bool canTransferIn(double amount, string& probStr) const;

void recordWithdrawal( double amount);

void recordTransferOut(const double& amount, const string& tAN, const string& tSC);

void recordTransferIn(const double& amount, const string& aAN, const string& aSC);

private:

double overdraftLimit;

};

#endif

**SavingsAccount.h**

#ifndef SavingsAccountH

#define SavingsAccountH

#include "BankAccount.h"

using namespace std;

class SavingsAccount: public BankAccount

{

public:

SavingsAccount();

virtual ~SavingsAccount();

double getMinimumBalance() const;

void setMinimumBalance(const double& mB);

virtual bool canDeposit(const double& amount) const = 0;

private:

double minimumBalance;

};

#endif

**ChildAccount.h**

#ifndef ChildAccountH

#define ChildAccountH

#include "SavingsAccount.h"

using namespace std;

class ChildAccount: public SavingsAccount

{

public:

ChildAccount();

~ChildAccount();

double getMinimumPaidIn() const;

double getMaximumPaidIn() const;

bool canDeposit(const double& amount) const;

ostream& putDataInStream( ostream& os) const;

istream& getDataFromStream( istream& is);

const string prepareFormattedStatement() const;

const string prepareFormattedAccountDetails() const;

double borrowable() const;

bool canWithdraw( double amount) const;

void recordDeposit( double amount);

bool canTransferOut(double amount, string& probStr) const;

bool canTransferIn(double amount, string& probStr) const;

void recordWithdrawal( double amount);

void recordTransferIn(const double& amount, const string& aAN, const string& aSC);

void recordTransferOut(const double& amount, const string& tAN, const string& tSC);

private:

double minimumPaidIn;

double maximumPaidIn;

};

#endif

**ISAAccount.h**

#ifndef ISSAccountH

#define ISSAccountH

#include "SavingsAccount.h"

using namespace std;

class ISAAccount: public SavingsAccount

{

public:

ISAAccount();

~ISAAccount();

double getMaximumYearlyDeposit() const;

double getCurrentYearlyDeposit() const;

Date getEndDepositPeriod() const;

ostream& putDataInStream( ostream& os) const;

istream& getDataFromStream( istream& is);

const string prepareFormattedStatement() const;

const string prepareFormattedAccountDetails() const;

void recordTransferOut(const double& amount, const string& tAN, const string& tSC);

void recordTransferIn(const double& amount, const string& aAN, const string& aSC);

bool canDeposit(const double& amount) const;

double borrowable() const;

bool canWithdraw( double amount) const;

void recordDeposit( double amount);

bool canTransferOut(double amount, string& probStr) const;

bool canTransferIn(double amount, string& probStr) const;

void recordWithdrawal( double amount);

private:

double maximumYearlyDeposit;

double currentYearlyDeposit;

Date endDepositPeriod;

void updateCurrentYearlyDeposit(const double& a);

};

#endif

**Advanced Design Code**

**Recursion:** TransactionList::getTransactionsUpToDate

TransactionList TransactionList::getTransactionsUpToDate(const Date& date, TransactionList temp)

{

if(temp.size() <= 0)

{

TransactionList dummyTrl;

return dummyTrl;

}

else

if(temp.newestTransaction().getValue<Date>() <= date)

return temp;

else

{

temp.deleteFirstTransaction();

TransactionList test(getTransactionsUpToDate(date, temp));

return test;

}

}

**Recursion:** TransactionList::deleteTransactionsUpToDate

void TransactionList::deleteTransactionsUpToDate(const Date& date)

{

if(size() <= 0)

{

return;

}

else

{

if (newestTransaction().getValue<Date>() <= date)

{

\*this = olderTransactions();

this->deleteTransactionsUpToDate(date);

}

else

{

Transaction firstTr(newestTransaction());

this->deleteFirstTransaction();

this->deleteTransactionsUpToDate(date);

this->addNewTransaction(firstTr);

}

}

}

**Options 7 Templates:**

template <typename T> TransactionList produceTransactionsForAmount(const T amount) const

{

TransactionList copy(\*this);

TransactionList temp;

for(int i = 0 ; i < (\*this).size() ; ++i)

{

if(copy.size() > 0)

{

if(copy.newestTransaction().getValue<T>() == amount)

{

temp.addTransaction(copy.newestTransaction());

}

copy.deleteFirstTransaction();

}

}

return temp;

}

template <typename T> string m7a\_showTransactionsForAmount(T amount) const

{

ostringstream os;

TransactionList results(p\_theActiveAccount\_->getTransactions().produceTransactionsForAmount(amount));

if (!results.size() == 0)

os << "\n\n" << results.size() << " TRANSACTIONS FOUND\n" << results.toFormattedString(); //one per line

else

os << "\n\nNO TRANSACTION IN BANK ACCOUNT MATCH THE SEARCH CRITERION GIVEN!";

return (os.str());

}

template <typename T> const T getValue() const

{

return date\_;

}

template <> const Time getValue<Time>() const

{

return time\_;

}

template <> const string getValue<string>() const

{

return title\_;

}

template <> const double getValue<double>() const

{

return amount\_;

}

template <typename T> T readInSearchAmount() const

{

T amount;

cout << "\n ENTER SEARCH AMOUNT: ";

cin >> amount;

return amount;

//return searchAmount;

}

template <typename T> Date readInSearchAmount(T cr) const

{

int day,month,year;

Date searchDate;

do{

cout << "\n ENTER VALID DATE: ";

cout << "\n ENTER SEARCH DAY: ";

cin >> day;

cout << "\n ENTER SEARCH MONTH: ";

cin >> month;

cout << "\n ENTER SEARCH YEAR: ";

cin >> year;

searchDate = Date(day, month, year);

} while (!searchDate.isValidDate(cr));

return searchDate;

}

**Abstract Classes**

class BankAccount {

public:

//constructors & destructor

BankAccount();

BankAccount( const string& typ, const string& acctNum, const string& sCode,

const Date& cD, double b,

const TransactionList& trList);

virtual ~BankAccount();

//getter (assessor) functions

const string getAccountType() const;

const string getAccountNumber() const;

const string getSortCode() const;

const Date getCreationDate() const;

double getBalance() const;

const TransactionList getTransactions() const;

const void produceTransactionsUpToDate(const Date& date, string& str, int& numOfTr);

bool isEmptyTransactionList() const;

void produceAllDepositTransactions(string&, double&);

//functions to put data into and get data from streams

virtual ostream& putDataInStream( ostream& os) const = 0;

virtual istream& getDataFromStream( istream& is) = 0;

void setBADetails(const string& accT, const string& accN, const string& sC, const Date& cDate, const int& bal, const TransactionList& tr);

//other operations

virtual const string prepareFormattedAccountDetails() const = 0;

virtual const string prepareFormattedStatement() const = 0;

virtual const void produceNMostRecentTransactions(int numOfTr, string& str, double& total) const;

virtual void recordDeposit( double amount);

void addTransaction(Transaction tr);

void recordDeletionOfTransactionUpToDate(const Date& date);

virtual double borrowable() const = 0;

virtual bool canWithdraw( double amount) const = 0;

virtual bool canTransferOut(double amount, string& probStr) const = 0;

virtual bool canTransferIn(double amount, string& probStr) const = 0;

virtual void recordWithdrawal( double amount) = 0;

virtual void recordTransferIn(const double& amount, const string& aAN, const string& aSC) = 0;

virtual void recordTransferOut(const double& amount, const string& tAN, const string& tSC) = 0;

class SavingsAccount: public BankAccount

{

public:

SavingsAccount();

virtual ~SavingsAccount();

double getMinimumBalance() const;

void setMinimumBalance(const double& mB);

virtual bool canDeposit(const double& amount) const = 0;

private:

double minimumBalance;

};

**STL Containers**

#include <list>

class TransactionList {

public:

**……**

private:

list<Transaction> listOfTransactions\_; //list of transactions

};

void TransactionList::addNewTransaction( const Transaction& tr) {

listOfTransactions\_.push\_front( tr);

}

const Transaction TransactionList::newestTransaction() const {

return (listOfTransactions\_.front());

}

const TransactionList TransactionList::olderTransactions() const{

TransactionList trlist( \*this);

trlist.deleteFirstTransaction();

return trlist;

}

void TransactionList::deleteFirstTransaction() {

listOfTransactions\_.pop\_front();

}

**Singleton Pattern**

static UserInterface\* getInstance(){

static UserInterface instance;

return &instance;

}

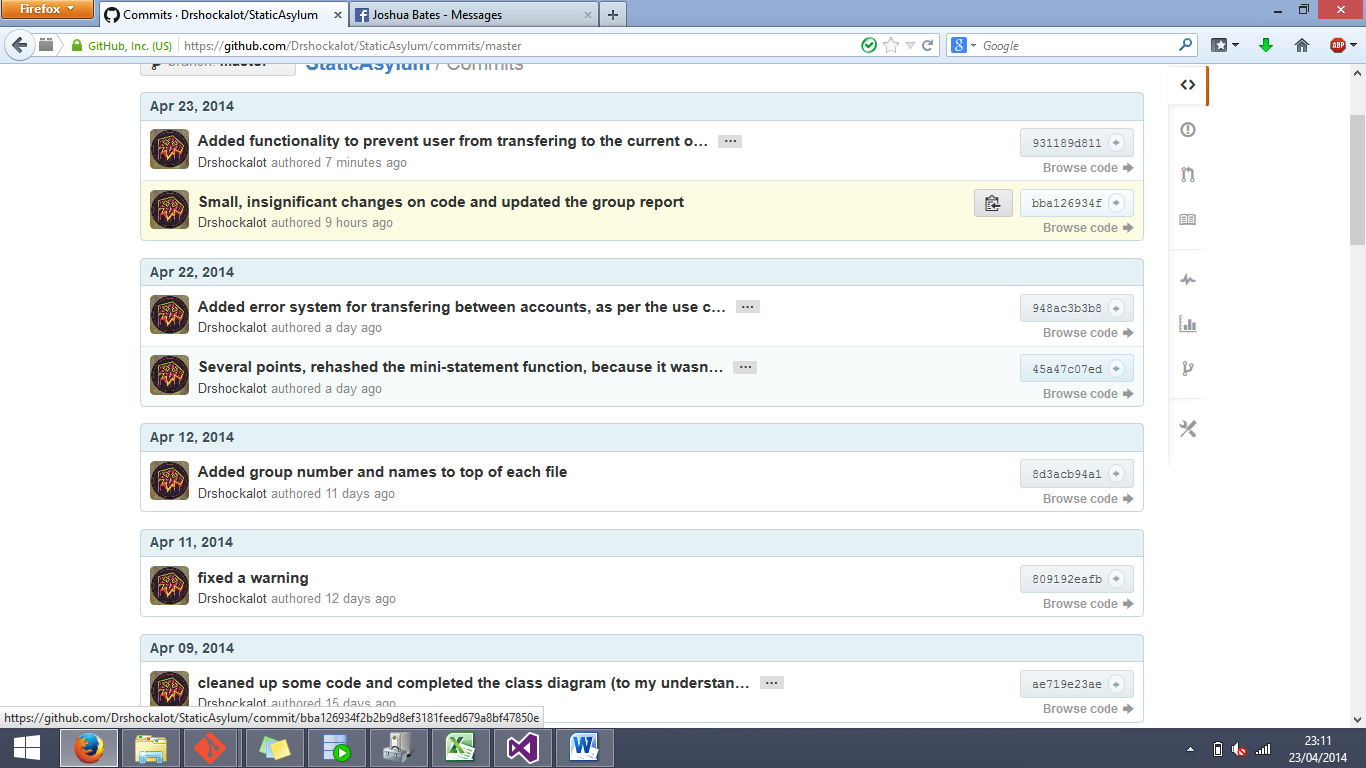
CashPoint::CashPoint()

: p\_theActiveAccount\_( nullptr), p\_theCashCard\_( nullptr), p\_theTransferAccount\_( nullptr), p\_theUI\_(UserInterface::getInstance())

{ }

**Version Control Log**

Legend:

* Drshockalot = Chad Simpson
* TehWebby = Shaun Webb
* Lukeeward = Luke Ward

