OOP – TASK 1c – SEMESTER 2 GROUP ASSIGNMENT

OUTLINE BRIEF

You are required to demonstrate your understanding of some of the principles of the OO approach and your competence in applying the C++ programming techniques presented in Semester 2 to implement a design documented in UML.
The questions for this assignment refer to the Case Study #2 (ATM) used in some of your lectures and tutorials on UML and C++.
This document (**OOP_Task1c_BRIEF.docx**) contains the list of additional requirements for the system together with relevant UML documentation. You are also provided with a partial C++ implementation of the system that is available from Bb (project and source files in **OOP_Task1c_SKELETON.zip**).

Study the C++ program provided and the related UML documentation and:

- answer questions about them (**Question 1**)
- provide some additional UML documentation (**Question 2**)
- apply fundamental OO principles and techniques presented this semester to your C++ implementation and extend the C++ skeleton program so that it offers additional functionality (**Question 3**)
- implement a hierarchy of classes allowing for true polymorphic behaviour (**Question 4**)
- use more advanced design and programming techniques including templates, abstract classes, use of STL containers and singleton pattern (**Question 5**).

You need to produce a **report** with answers to the questions and extracts of your solutions and to participate to a **walkthrough** (i.e. demonstration of your C++ program).
This is a **group assignment** that counts for **25%** of the mark for the OOP module.

OBJECTIVES

- Demonstrate your understanding of basic elements of UML
- Apply some essential OO facilities offered by C++ and demonstrate programming skills required to implement a small OO software system
- Relate design documentation in UML with C++ implementation of a small OO software system
- Work effectively as a member of a project team on the development and maintenance of a small OO software system
- Produce a clear and comprehensive report on work produced.

GROUPS CONSTITUTION

See details on **Appendix 6**.

DEADLINES

ELECTRONIC SUBMISSION [CODE AND REPORT]: 17 April 2018, 3pm
WALKTHROUGH: Week commencing 16 April 2018 *(time tba and booking via wiki)*

***NOTE: Marks can be deducted** if your report presentation and/or structure of electronic files provided do not follow the instructions or if you do not attend the walkthroughs.*

DELIVERABLES AND WALKTHROUGH

You will be assessed on:

1. A Word document for your group **report** containing your answers to Question 1, the UML diagrams for Question 2, C++ code extracts for Questions 3, 4 and 5 plus a version control log.

Answer questions about code provided: You must justify your answers to get the marks on this question.

UML diagrams: You must adopt the diagram conventions presented in class or specified in this document. However, if necessary, you can use English to complement your explanations so long as they are clear and unambiguous. You can produce the diagrams by hand or use tools such as Visio, Rational Rose or Together to help you.

C++ code: You must include the code for the required header files and specified function definitions.

Version Control Log: This report, automatically generated by the tool you have used to support the version control for the group work, lists the changes made by the various members of your team. It will be used to evidence individual contributions to the group work. So when make sure that the log information documents actual members' contributions.

Report: This report is to be presented on A4 paper typed, stapled (easily accessible for adding comments), paginated and identified with your assignment group number (e.g. SE1, CS4G3) and names of group members clearly indicated.

2. Your **application**.

Application: Your C++ program MUST work in MS Visual Studio C++ 2017 and run on the University network.
Each file should include comments at the top to label your group clearly - assignment group number (e.g. SE1, CS4G3) and names of everyone in your group.

3. A **walkthrough** for (i.e. demonstration of) your C++ program.

Walkthrough: It contributes to the final mark for this assignment and involves you demonstrating your program to your tutor, describing what it does and how it works. You will have the chance to point out interesting or innovative aspects of your work. You must be able to answer questions about your design and your code and made changes to it on request to modify some of the program behaviour.

Hand in your **'Group Contribution'** form. See **Appendix 6** for details.

4. Electronic submission of your work in **ONE .zip** file (called **OOP_Task1c_*yourGroupNumber*.zip**) holding:
   - **OOP_Task1c_Report_*yourGroupNumber*.doc**: a Word document for your report (Questions 1 and 2, C++ code extracts for Questions 3, 4 and 5 and version control log).
   - **C++ Visual Studio project and associated files** for your C++ application (Questions 3, 4 and 5).

See **Appendix 5** for details related to 'Hand-In Report, Walkthrough and Marking Scheme' and 'Groups Constitution' sections for details.

SYSTEM REQUIREMENTS

CURRENT SYSTEM

The (simple) **ATM** system given to you in the skeleton program for this assignment (in **OOP_Task1c_SKELETON.zip**) implement a small program that allows a customer to request some operations on their bank account(s) from a virtual ATM.

- It includes some basic functionality, e.g., the customer can get the balance on the account, deposit or withdraw some money from the account and see a statement showing details of the account including the list of transactions.
- A customer can have multiple accounts. A single customer's card grants access to all of that customer's accounts.
- The customer interacts with the ATM machine via a set of menus. To start with the customer enters his/her card, indicates which account is to be activated and, when this is successful, selects from a menu of options a command to be applied to the active account. This interaction between the customer and the system is described in **Fig. 1a** shown in **Appendix 1**.

Classes such as **Card**, **BankAccount**, **Customer**, **Date**, **Time**, **TransactionList** and **Transaction** have been identified in the initial phases of the system design and implemented in the system.

- A **BankAccount** is identified by an **accountNumber**. It also contains information such as its **type**, **creationDate** and the current **balance** on the account. Each **BankAccount** has a link to an object of class **TransactionList**. A **BankAccount** can be queried or updated using operations in its public interface (such as **recordDeposit**, **getBalance**, or **closeAccount**).
- The **Card** is associated to a (single) **Customer** and gives access to all of that customer's **BankAccount**(s), each identified by its **accountNumber.**
- A **TransactionList** acts as the record of all the transactions that have taken place on a **BankAccount**. A **TransactionList** can be queried or updated using operations in its public interface (such as **addTransaction** or **getTotalTransactions** or **deleteTransactionUpToDate**).
- Each **Transaction** records a **date**, a **time**, a **title** (e.g. the reference and location of the ATM where money was withdrawn, or the number of a cheque) and an **amount** (positive for deposits, negative for withdrawals).
  A **Transaction** can be queried or manipulated using operations in its public interface (such as **getAmount** or **updateAmount**).
- **Date** and **Time** are used to record and manipulate date and time values and are used by several other classes in the system.

All data about the bank accounts are stored in text files provided in the **data** folder included in project **.zip** file.
There is no facility in this part of the system to create new accounts or definitively erase existing accounts. All accounts used already have an associated text file containing the relevant valid data for that account. You can use the files provided or create additional text files yourself to test the various functionalities offered by your application. However, when the program is running you must check whether the account that the user want to access actually exists.

> ***NOTE:*** *During the final walkthrough, the test data files provided (holding the information given in* ***Appendix 4****) will be used to verify that your application is working correctly.*

You need to modify and extend this program so that it handles different types of bank accounts and it includes additional functionality described below.

NEW TYPES OF BANK ACCOUNTS

The system should handle accounts of various types (i.e. current accounts, savings accounts and child accounts). The structure and relationships between these various types of accounts is represented in the (incomplete) **Class Diagram** given in **Fig. 2** shown in **Appendix 2**.

A bank account represents the general abstract structure common to all types of bank accounts. It only has the following attributes:

- **accountNumber**: a string representing the account number used to uniquely identify the account,
- **transactions**: the list (i.e., history) of all the bank transactions (currently) recorded for that account,
- **creationDate**: the date at which the account was first opened (i.e., created),
- **balance**: a double value representing the amount of money currently in the account – typically a positive value.

A current account is a specialised concrete type of bank account that allows overdrafts. The balance of such an account can either be positive to indicate how much money is available or negative to show how much has currently been overdrawn. It has the following new attribute:

- **overdraftLimit**: a (positive or zero) double number set when the account is first opened that represents the maximum amount of money the user can owe the bank at any time.

A savings account is another specialised abstract type of bank account that offers an interest on the balance in the account *(not handled in this part of the system)* and can specify the amount of money that must remain on the account. It has the following new attribute:

- **minimumBalance**: a (positive or zero) double number that represents how much money must be left in the account at any time.

A child account is a specialised concrete type of savings account that is only available to save money for children. No money can be taken out of the account until it is closed when the child becomes an adult *(not handled in this part of the system)*. It also restricts the amount of money that can be added to the account per transaction. It has the following new attributes:

- **minimumPaidIn** and **maximumPaidIn**: two (positive or zero) double numbers set when the account is first opened that represent restrictions on the amount of money that can be paid in for any individual transaction (i.e., deposit or transfer in of funds).

An ISA account is a specialised concrete type of savings account that earns tax fee interest *(not calculated in this part of the system)* but has restriction on the maximum amount of money that can be paid in overall on the account in any one tax year. The current yearly deposit considers all the money that was paid in to the account (it does not matter that some of that money may have been taken out in the meantime). It must never exceed the maximum yearly deposit allowed on the account. After the tax year has ended, money can no longer be added to the account, but the money can stay there incurring gross interest *(not calculated in this part of the system).* It has the following three new attributes:

- **maximumYearlyDeposit**: a (positive or zero) double number set when the account is first opened that represents the maximum accumulated amount of money that can be paid in into the account (through deposit transactions or transfers of funds) during the valid deposit period.
- **currentYearlyDeposit**: a (positive or zero) double number that represents the current amount of accumulated payment into the account (through deposit or transfers of funds) during the valid deposit period.
- **endDepositPeriod**: a date that is set automatically when the account is first created (e.g., 1 year after the account's creation date). No money can be paid in (deposited or transferred) into the account after that date.

ADDED FUNCTIONALITY

1. Recording bank account details

Specific information about each bank account is stored in a text file. The structure of such a text file will differ depending on the nature of the bank account (e.g., all accounts will have account number, creation date, balance and transaction list, but current accounts will also have overdraft limit and child accounts minimum balance, minimum deposit, etc.) They should match the structure of the sample files stored in the **data** folder of the project provided.

2. Showing mini-statement

A new command (**Option 6 on "Account" menu**) is to be offered by the system to show a mini statement for the account. It reads in the number of most recent transactions required by the user and displays the details of these transactions, if any, on the bank account together with the cumulated total for all these transactions. It also shows the account number, balance and available funds (i.e., how much could be withdrawn from the account). If there are fewer transactions than requested, it shows all the recorded transactions.
The Use Case in **Fig. 2** shown in **Appendix 1** describes what happens when a customer opts for the 'Show Mini Statement' option. The (incomplete) **Sequence Diagram in Fig. 3a** shown in **Appendix 3** represents some of the interactions that that take place to implement this option.

3. Searching for transactions matching a given criterion

A new command (**Option 7 on "Account" menu**) is to be offered by the system to search of transactions matching a given criterion (date, title or amount) provided by the user. The system displays a sub-menu for the various searches available, reads in the value to be searched for and displays the details of all the matching transactions, if any, together with the number of such transactions.
The Use Case in **Fig. 1c** shown in **Appendix 1** describes what happens when a customer opts for the 'Search Transactions' option. The **Sequence Diagram in Fig. 3b** shown in **Appendix 3** represents some of the interactions that take place to implement a customer's search for a given amount. The other searches should be implemented in a similar way. The search per title should search for either a full match or a substring of the transactions' title.

4. Clearing all transactions up to date

A new command (**Option 8 on "Account" menu**) is to be offered by the system to clear the transaction history. It reads in a valid date, displays the details of all the transactions, if any, on the bank account up to and including that given date together with the number of such transactions. The user must then confirm that these transactions are to be removed from the transaction history before they are deleted from the account. The system displays a message to confirm that they have been deleted. No other part of the account is to be modified. To be valid the date must be between the account creation date and the current date – both included.
The Use Case in **Fig. 1d** shown in **Appendix 1** describes what happens when a customer opts for the 'Clear Transactions Up To Date' option. The **Sequence Diagram in Fig. 3c** shown in **Appendix 3** represents some of the interactions that take place to implement this option.

5. Showing all available funds

A new command (**Option 9 on "Account" menu**) is to be offered by the system to show available funds across all accounts accessible with that card (i.e., cumulated amount that the user could withdraw now if they had to). For each account listed on the current card, the system displays information about the account (account number, balance and available funds (i.e., how much could be withdrawn from the account). It also indicates the total of all the available funds for all accounts on the card.
The Use Case in **Fig. 1e** shown in **Appendix 1** describes what happens when a customer opts for the 'Showing available funds' option. The **Sequence Diagram in Fig. 3d** shown in **Appendix 3** represents some of the interactions that take place to implement this option.

6. Transfer of funds between accounts

A new command (**Option 2 on "Card" menu**) is to be offered by the system to transfer money between bank accounts associated to the same card. When a requested transfer is authorised, both accounts should record the transfer, with the titles of these transactions indicating whether the money is transferred in or out of the account and giving the number of the other account.
The Use Case in **Fig. 1f** shown in **Appendix 1** describes what happens when a customer decides to transfer money from the account currently being processed to another of the accounts accessible with the active card. The (incomplete) **Sequence Diagrams in Fig. 3d1** and **3d2** shown in **Appendix 3** represents some of the interactions that take place to implement this option.

USER INTERFACE

The **UserInterface** class is responsible for ALL interactions with the user (and perform some very basic data entry validation, e.g., checking that a value is positive). No other classes should expect input or provide output directly: all input and output are to be delegated to the **UserInterface** class.
This class is to be as loosely coupled to the other classes in the system as possible. In a fuller development of this system this would also contribute to providing a secure and efficient interaction with various types of user interfaces *(outside the scope of this system)*. In other words, except for instances of the **Time** and **Date**, the **UserInterface** class only uses C++ predefined data types such as strings, integer, character, floating point or Boolean values.

TRUE POLYMORPHIC BEHAVIOUR

Whenever appropriate, the system must consider, at run time, the nature of the (active or transfer) bank accounts to provide the necessary information, perform required validations and execute the customer's request where possible or show appropriate error messages.

ABSTRACTION

Where possible and appropriate use templates, abstract classes to provide abstraction and implement a robust and extendable system.

QUESTIONS

1. Study the UML documentation provided and the (partial) C++ code given carefully and briefly answer each of the following questions.

> ***NOTE:*** *You must justify your answers to get full marks on these questions. Marks will be awarded for correctness, completeness and overall clarity of the answers.*

(20 marks)

a) According to the UML Class Diagram in **Fig. 2** is it possible for the **canWithdraw** operation to be inherited and used directly in the **ChildAccount** class? Describe what problems may arise **when implementing this design** and what can be done to solve these.

b) The C++ **`ATM`** class includes a **`BankAccount`** pointer as one of its data members. Explain why, in this case study, using a **`BankAccount`** instance instead of a pointer to implement this relationship, would not be appropriate. Give specific examples **from your final solution** to illustrate your answer.

c) In the C++ implementation given, what is the nature of relationship between the **`Card`** and **`List<string>`** classes, how should it represented in UML and what C++ mechanisms are involved in its C++ implementation?

d) Is the **`UserInterface`** class an abstract class? How do you know? If not, should it be?

e) Why is the **`Date::currentDate()`** function declared as **`static`**? How does this mechanism work?

f) Assuming that **`t1`** is a valid **`Time`** instance, indicate which functions are called in each of the following lines of code. Will they work with the **`Time`** class given? Explain the issues, if any, and describe what changes are needed in the **`Time`** class for each of these statements to be valid, equivalent and works as expected (i.e., create the same instance **`t`**).

```
Time t(t1 + Time(0, 0, 12));          //line 1
Time t(Time(12) + t1);                //line 2
Time t(t1 + 12);                      //line 3
Time t(12 + t1);                      //line 4
```

g) The purpose of the **`TransactionList::deleteGivenTransaction`** function is to delete the first occurrence, if there is one, of a given transaction from a non-empty transaction list. Is the following version correct? If not, describe all the reasons why it isn't and show how it should be amended to work correctly.

```
void TransactionList::deleteGivenTransaction(const Transaction& tr)
{
   assert(size() != 0);
   if (newestTransaction() == tr)
      *this = olderTransactions();
   else
   {
      Transaction firstTr(newestTransaction());
      olderTransactions().deleteGivenTransaction(tr);
      this->addNewTransaction(firstTr);
   }
}
```

h) What differences would it make to declare a method such as **`BankAccount::prepareFormattedAccountDetails`** as **`virtual`**? Explain why you might want to do this.

i) Would the expression **`p_theActiveAccount_->getOverdraftLimit()`** be valid if the pointer **`p_theActiveAccount_`** were to currently points to a **`CurrentAccount`** instance? If not, explain what could be done, if anything, to solve this problem.

j) Could the function **`ATM::m_acct1_produceBalance`** have equally been rewritten as follow? If not, give all the reasons why not.

```
void ATM::m_acct1_produceBalance() const {
   theUI_.showProduceBalanceOnScreen(p_theActiveAccount_->balance_);
}
```

2. Produce some UML diagrams for the application:

a) The C++ skeleton program given to you offers a command that allows the customer to withdraw money from a bank account.
Study the C++ code given carefully and **draw the Sequence Diagram** for this command.
It should start with the **m_acct2_withdrawFromBankAccount** message sent to the object '**theATM**' and represent the interactions that take place in the system when the withdrawal transaction requested can be authorised.

(5 marks)

NOTE: *Do not represent the details of what happens in the* ***List*** *class's operations and do not show more than 3 levels of nested interactions.*

Marks will be awarded for:

- overall clarity of the diagram
- showing all the objects involved in the interaction (class' identifiers and instance' names),
- showing construction and deletion of any temporary objects.
- showing all the messages involved in the interaction,
- correctly labelling the messages and showing their direction,
- showing both the type of the parameters and return value of any message.
- indicating the numbering (using the nested numbering system presented in class) and / or nesting of the messages activation boxes,
- placing guards where needed.

b) **Complete the class diagram** given in **Fig. 2**, for the system you have developed. It should show all the classes and class templates (if any) and their relationships. The classes in the **BankAccount** class hierarchy should show their attributes and methods. The other classes in the system should simply be represented by their icon (i.e., a box with only their name, no other details needed).

(5 marks)

Marks will be awarded for:

- showing ALL the classes' boxes and identifiers,
- showing all attributes and methods of the classes in the **BankAccount** class hierarchy but only the icons of the other classes in the system,
- (when appropriate) indicating both the type of the parameters and return value of the methods as well as the type of the attributes,
- (when appropriate) indicating the level of access of any class member (**+** for public and **-** for private members).
- showing ALL the relationships between classes using appropriate links (e.g., diamonds arrows for aggregation relationships, clear head arrow for specialisation relationships) and labelling these links when appropriate

NOTE: *If a software tool is used to create the diagrams, and the tool follows a different convention for showing access levels, a key should be provided with the diagram.*

3. **Modify the C++ program given**, to add the following four new commands**:**

   a) Add **Option 6** on the "Account" menu to implement the "Show Mini Statement" Use Case so that it works with any instance of the **`BankAccount`** class.
   Your implementation should match the Use Case in **Fig. 1b** and the (partial) Sequence Diagram in **Fig. 3a**.

   (4 marks)

   b) Add **Option 7** on the "Account" menu to implement the "Search for Transactions" Use Case so that it works with any instance of the **`BankAccount`** class.
   Your implementation should match the Use Case in **Fig. 1c**, and the (partial) Sequence Diagram given in **Fig. 3b1** and **3b2.**

   (8 marks)

   c) Add **Option 8** on the "Account" menu to implement the "Clear All Transactions up to Date" Use Case so that it works with any instance of the **`BankAccount`** class.
   Your implementation should match the Use Case in **Fig. 1d** and the (partial) Sequence Diagram in **Fig. 3c**.

   (8 marks)

   d) Add **Option 9** on the "Account" menu to implement the "Transfer Funds to Another Account" Use Case so that it works with any instances of the **`BankAccount`** class used either as the origin or as the destination of the transfer operation.
   Your implementation should match the Use Case in **Fig. 1f**, the Class Diagram in **Fig. 2** and the (partial) Sequence Diagrams given in **Fig. 3d1** and **Fig. 3d2**.

   (10 marks)

   e) Add **Option 1** on the "Card" menu to implement the "Show Available Funds" Use Case so that it works with any instance of the **`BankAccount`** class.
   Your implementation should match the Use Case in **Fig. 1e** and the (partial) Sequence Diagram in **Fig. 3e**.

   (5 marks)

Marks will be awarded for:

- **ensuring that your implementation matches the (partial) UML design given** (e.g., develop new public member functions in the **`TransactionList`** class such as **`getMostRecentTransactions`** and **`getTotalTransactions`** to match corresponding messages in **Fig 3a**. and **`getTransactionsForAmount`** to match the corresponding message in **Fig 3b**, etc.
- **producing a clear OO design** (e.g., reuse of existing methods, appropriate level of access to member functions, avoiding unnecessary dependencies between classes, appropriate allocation of responsibility to classes and appropriate implementation of relationships between classes)
- **making use of relevant additional C++ OO facilities** (e.g., use of **`const`** class functions, **`const`** reference parameters, **`const`** return values, **`static`** functions, use of template classes and member functions)
- **using recursion** for some of the new **`TransactionList`** functions
- **producing a robust system** by generally enforcing preconditions with **`assert`** statement where appropriate (e.g., checking the validity of user input) and checking that possible problematic situations are dealt with appropriately (e.g., checking that an account exists, is accessible with the given card before loading the data into the system, checking that there are some transactions on the account before asking for a date or number of required transactions)
- **producing readable code** (e.g., consistency of style, comments, meaningful identifiers, program layout).

NOTE: *To help you with this question look at the approach used to develop the "Show All Deposit Transactions" command in the "Task 1c Preparation" tutorial.*

4. **Develop a hierarchy of `BankAccount` classes** and **modify the C++ program given**, so that the user can work with bank accounts of different types and the program options work taking into account the nature of the (active) bank account(s) used for their operation:
   Your implementation should match the description of the various accounts given in p4 of this document and the related UML diagrams. i.e., the UML Class Diagram in **Fig. 2** and the relevant Use Cases and Sequence Diagrams.

   (20 marks)

   a) Develop a hierarchy of **`BankAccount`** classes that allows for abstraction, encapsulation and true polymorphic behaviour.

   b) Update (if necessary) the options on the account processing menu and test them to check that that they work properly with any instance of the **`BankAccount`** class hierarchy. In particular you should ensure that **Options 9** on the "Account" menu and **Options 2** on the "Card" menu work properly with any bank accounts of the **`BankAccount`** class hierarchy.

NOTE: *Some of the commands should not be affected by the types of the account used and even work in the same way whether you have or not implemented the* ***BankAccount*** *hierarchy.*

Marks will be awarded for (as in question 3 plus):

- **ensuring that your implementation matches your design**,
- **providing encapsulation** whenever appropriate for the implementation of the various bank accounts — each class in the **`BankAccount`** hierarchy may have **`private`** data (and function) members that are not directly accessible from outside that class,
- **using public inheritance** (i.e., public derivation) for the implementation of the various bank accounts — the **`BankAccount`** and **`SavingsAccount`** classes should be abstract using appropriate pure **`virtual`** functions, the other accounts should be concrete classes.
- **developing a truly polymorphic system** (where the system does not need to know in advance the type of type of the account selected for general processing or transfer). You could identify the type of an account from a marker —first digit— in the file name (e.g., account_101.txt for a current account, account_201.txt for a child account) or by checking the first field in the account file (e.g., string **`"CURRENT"`** for a current account or **`"CHILD"`** for a child account). The appropriate functions in the **`BankAccount`** hierarchy should be made **`virtual`** and pointers to **`BankAccount`** class should be used to provide access the various instances of this **`BankAccount`** hierarchy used in the application.
- **making use of relevant C++ other OO facilities** incl., placing code for each class in separate modules, providing appropriate initialisation lists in constructors, using **`const`** functions and data whenever possible, overriding functions when appropriate, making use of operator overloading where appropriate (e.g., **`operator>>`** and **`operator<<`** for each type of bank accounts), using a function template for **`operator>>`** and **`operator<<`** so that they can be used to insert in stream or extract from stream data of any type of object used in the program.
- **producing a clear OO design of classes** (e.g., providing natural class public interface, developing a highly cohesive (i.e., consistent, clear and limited responsibility for each class) and loosely coupled system (i.e. minimal amount of coupling between classes), following the design guidelines given in lectures for the implementation of the relationships between classes in the system.
- **producing a robust system** by checking that possible problematic situations are dealt with appropriately (e.g., preventing attempts to transfer money from an account to itself, checking for each transfer that money can be both withdrawn from the active account and credited to the transfer account).

5. Implement relevant part of the code using **more advanced design and programming techniques**:

   a) **Recursion**: Provide a recursive version for two of the functions used in Question 3c:
   `TransactionList::getTransactionsUpToDate`
   `TransactionList::deleteTransactionsUpToDate`

   b) **Templates**: Use templates in your implementation of the solution to Question 3b.

   c) **Abstract classes**: Implement the **`BankAccount`** hierarchy for Question 4 using abstract classes appropriately.

   d) Use of **STL containers**: Modify the **`TransactionList`** class so that it uses the **`list`** STL container to store the **`Transaction`** instances (instead of a **`List<Transaction>`**).

   e) **Singleton pattern**: Modify the implementation of the **`UserInterface`** class to ensure that the **`theUI_`** instance is unique in the program.

(15 marks)

HAND-IN REPORT, WALKTHROUGH AND MARKING SCHEME

ELECTRONIC SUBMISSION (REPORT AND CODE)

You are asked to submit your work electronically on Bb in a **.zip** file (called **OOP_Task1c_*yourGroupNumber*.zip**). Only one member per group should upload this file in required format and before the deadline.

This **.zip** file should hold:

- **OOP_Task1c_Report_*yourGroupNumber*.docx**: Word document holding a **complete report** with clear presentation, labelling each page with group member's names, tutorial group and page number. It MUST include
  - Your answers to Question 1
  - Your UML diagrams for Question 2
  - The code for the declarations and definitions of the functions corresponding to messages highlighted in yellow in **Appendix 3** and listed below for Question 3
  - Extracts of code related to your implementation of more advanced design and programming techniques for Question 5.
  - A version control log (or equivalent).

- Your **C++ program** (with implementation of solutions to Questions 3, 4 and 5)**.** This should include the **.vcxproj**, **.sln**, **.cpp** and **.h** files and the **data** folder that holds the **.txt** files storing the card and account data used for testing. Nothing else!
  Put as a comment on each **.h** and **.cpp** files the names of the students in the group.

WALKTHROUGH

The **walkthrough** will be organised during the tutorials that follow the hand-in date of your report. It will contribute to the final mark for this assignment.
You will be expected to:

- Have a program running with MS Visual Studio 2017 on one of the PCs in the University labs.
- Use the given test data text file to explore and test the various aspects of your program.
- Answer any questions about code
- Make simple modifications to your program on demand
- Be aware of your program limitations (i.e., don't let ME crash it!).

NOTE: *Marks can be deducted for poor report and lack of participation to group work and/or walkthroughs to the maximum of:*
- *Report and electronic submission: **-5%***
- *Walkthrough: **-5%***

Typically, all members will receive the same mark for the group work, however, when appropriate, individual marks may be will be adjusted to reflect member's contribution to the group work.

MARKING SCHEME

See details on **Appendix 5**.

APPENDIX 1: USE CASES DESCRIPTIONS

"Top level" Use Case

1. The system shows the "Welcome" menu to allow user to enter their card details or quit the application.
2. The customer selects **Option 1** on the top "Welcome" menu to activate a card.
3. The system prompts for the card number.
4. The customer enters a bank card (i.e., here type in the card number).
5. The system presents the "Card" menu that displays a menu of operations available on that card.
6. The customer selects **Option 1** on the "Card" menu to manage an individual account.
7. The system displays the list of bank accounts associated with that card (i.e., account numbers identifying each of the bank accounts on the list).
8. The customer indicates which of the bank accounts accessible with that card is to be processed (i.e. appearing on the bank account list).
9. The system presents the "Account" menu that displays a menu of operations available on that account (e.g., "Back to card menu", "Display balance", "Withdraw from account", "Deposit into account", etc.").
10. The customer indicates which of these options is requested.
11. The system acts on that request, i.e. the request is executed, and the system goes back to Step 9.

Alternate courses:

at 2. If the customer selects **Option 0** on the top "Welcome" menu (to quit the application) the program ends.

at 5. If the card is invalid, the system displays an appropriate message (e.g., "INVALID CARD") and goes back to Step 1.

at 6a. If the customer selects **Option 0** on "Card" menu the system goes back to Step 1.

at 6b. If the option selected is not on the menu or not implemented yet the system displays an appropriate message (e.g., "INVALID COMMAND") and goes back to Step 1.

at 9a. If the account selected is not on the card list the system displays an appropriate message (e.g., "ACCOUNT NOT AVAILABLE WITH CURRENT CARD") and goes back to Step 1.

at 9b. If the account selected does not exist (i.e., no corresponding file) the system displays an appropriate message (e.g., "ACCOUNT DOES NOT EXIST") and goes back to Step 1.

at 9c. If the type of the account selected is not recognised (i.e., not matching an existing type of account) the system displays an appropriate message (e.g., "ACCOUNT TYPE NOT RECOGNISED") and goes back to Step 1.

at 11a. If the option selected is not on the menu or not implemented yet, the system displays an appropriate message (e.g., "INVALID COMMAND") and goes back to Step 5.

at 11b If the request is "Quit" the system goes to Step 5.

Fig. 1a: Use Case describing top level interaction between the customer and the system.

"Showing Mini Statement" Use Case (Option 6 on "Account" menu)

1. The system shows the options on "Account" menu for processing the active account.
2. The customer chooses **Option 6** to see a mini statement of the account.
3. The system asks for the number of required transactions.
4. The customer enters the number.
5. The system displays the current date and time (e.g., "RECENT TRANSACTIONS REQUESTED AT 12:03:45 ON 20/03/2018").
6. The system displays information about the account (i.e., account number, balance and available funds (i.e., how much could be withdrawn from the account).

may be the same as the balance, but not always)

7. The system displays the details of the required number of the most recent transactions in the bank account together with their cumulated amount.

may be £0.00, positive or negative depending on the transactions.

8. The system indicates that the operation has been successful and goes back to Step 1.

Alternate courses:

at 3. If there are no transaction in the bank account the system displays an appropriate message (e.g., "NO TRANSACTIONS IN BANK ACCOUNT") and carries on to Step 8.

at 4. If the number of transaction requested is not valid (i.e., negative or zero), the system asks for another one until a valid number is read in and carries on to Step 5.

at 7. If there are less than the required number of transactions in the bank account, the system displays all the transactions there is together with their cumulated amount (may be 0) and carries on to Step 8.

Fig. 1b: AMENDED Use Case describing the 'Show Mini Statement' option

"Search For Transactions" Use Case (Option 7 on "Account" menu)

1. The system shows the options on "Account" menu for processing the active account.
2. The customer chooses **Option 7** to search for given transaction on the account.
3. The system shows a sub-menu with the options available for searching for a transaction (a) per amount, (b) per title, (c) per date or (d) exit without searching.
4. The customer chooses how to search for the transaction on the account.
5. The customer enters the required search criterion (i.e., given (a) amount, (b) title or subtitle or (c) date).
6. The system displays the details of the transactions found on the bank account that match that search criterion.
7. The system indicates that the operation has been successful, displays an appropriate message giving the number of transactions found (e.g., "THERE ARE 3 TRANSACTIONS IN BANK ACCOUNT MATCHING THAT SEARCH CRITERION") and goes back to Step 1.

Alternate courses:

at 3. If there are no transactions in the bank account the system displays an appropriate message (e.g., "NO TRANSACTIONS IN BANK ACCOUNT") and goes back to Step 1.

at 4. If the user chooses to exit the search menu the system goes back to Step 1.

at 5a. If the search criterion entered is not valid (e.g., invalid date, empty string for title), the system asks for another one until a valid search criterion is read in and carries on to Step 3.

at 5b. If there is no transaction matching the search criterion given the system displays an appropriate message (e.g., "NO TRANSACTION IN BANK ACCOUNT MATCH THE SEARCH CRITERION GIVEN") and goes back to Step 1.

Fig. 1c: Use Case describing the "Search for Transactions" option

"Clear All Transactions Up To Date" Use Case (Option 8 on "Account" menu)

1. The system shows the options on "Account" menu for processing the active account.
2. The customer chooses **Option 8** to clear the list of transactions on the account up to a given date.
3. The system asks for a date.
4. The customer enters the date.
5. The system displays the number and details of all the transactions, if any, on the bank account up to and including that given date.
6. The system asks for confirmation of the clearing operation.
7. The user confirms that the shown transactions are to be deleted.
8. The system updates the list of transactions accordingly.
9. The system indicates that the operation has been successful, displays an appropriate message giving the number of transactions that have been deleted (e.g., "THE 23 TRANSACTIONS IN BANK ACCOUNT UP TO DATE 01/01/2018 HAVE BEEN DELETED") and goes back to Step 1.

Alternate courses:

at 3. If there are no transactions in the bank account the system displays an appropriate message (e.g., "NO TRANSACTIONS IN BANK ACCOUNT") and goes back to Step 1.

at 5a. If the date entered is not valid (i.e. impossible date – ignoring leap years, date either before the creation date of the bank account or after the current date), the system displays an appropriate error message and asks for another one and goes back to Step 3.

at 5b. If there is no transaction for or before that date the system displays an appropriate message (e.g., "NO TRANSACTION IN BANK ACCOUNT UP TO DATE 01/01/2018") and goes back to Step 1.

at 7. The user cancels the operation, the system indicates that the operation has been cancelled and goes back to Step 1.

Fig. 1d: Use Case describing the "Clear All Transactions up to Date" option (option 8)

"Transfer Funds To Another Account" Use Case (Option 9 on "Account" menu)

1. The system shows the options on "Account" menu for processing the active account.
2. The customer chooses **Option 9** to transfer some money out of this account to another of the accounts available with that card.
3. The system identifies the bank account(s) associated with that card (i.e. appearing on its list of bank accounts) and lists their account numbers.
4. The customer indicates which one of the bank account(s) accessible with that card is to receive the transferred money.
5. The system confirms that this selection was authorised by displaying an appropriate message (e.g., "THE ACCOUNT (NUMBER: 201) IS NOW OPEN!")
6. The system prompts the customer for in a valid amount (positive) of money to be transferred.
7. The customer types in the amount of money that is to be transferred.
8. The system indicates that the transfer can be granted.
9. The system records the appropriate debit (out) and credit (in) transactions on the selected accounts.
   Each of these two transactions must indicate in their titles the account number of the accounts involved in the transfer (e.g., when money is transferred from (active) account no "102" to (transfer) account no "201", the titles of the transactions should be "transfer_out_to_acct_201" in the active account and "transfer_in_from_acct_102" in the transfer account). They also record the time of the transfer and the amount of money that has been transferred (i.e. withdrawn from account 201 and credited to account 101).
10. The system indicates that the transfer has been successful and goes back to Step 1.

Alternate courses:

at 5a. If the account selected does not exist (no corresponding file) the system displays an appropriate error message (e.g., "THE ACCOUNT (NUMBER: 107) DOES NOT EXIST!") and goes back to Step 1.

at 5b. If the account selected exists but is not accessible with the current card the system displays an appropriate error message (e.g., "THE ACCOUNT (NUMBER: 103) IS NOT ACCESSIBLE WITH THIS CARD!") and goes back to Step 1.

at 5c. If the customer attempts transfer within the same, already active, account the system displays an appropriate error message (e.g., "THE ACCOUNT (NUMBER: 101) IS ALREADY OPEN!") and goes back to Step 1.

at 8a. If the customer enters an invalid amount (negative or zero) the system displays an appropriate message (e.g., "AMOUNT SHOULD BE A POSITIVE NUMBER!"), asks for another amount and goes back to Step 6.

at 8b. If the current state and characteristics of the accounts involved prevent the transfer of the amount requested (e.g., not enough money, even considering possible authorised overdraft, not respecting minimum balance specified, minimum or maximum transaction limits (if any)) the system displays an appropriate message (indicating clearly the nature of the problem) and goes back to Step 1.

Fig. 1f: Use Case describing funds transfer between accessible accounts

<u>**"Show Available Funds" Use Case (Option 2 on ”Card” menu)**</u>

1. The system shows the options available on ”Card” menu for processing the accounts on the active card.
2. The customer chooses **Option 2** to display the available funds on all accounts for that card.
3. The system identifies the bank account(s) associated with that card (i.e. appearing on its list of bank accounts).
4. The system calculates the amount of money available (i.e., that could be withdrawn if needed) on each account and their cumulated sum.
5. For each account on the current card, the system displays the account details (account number, balance and associated amount available (i.e., that could be withdrawn) as well as the total of these sums that the user could take out at this point.
6. The system goes back to Step 1.

Alternate courses:

at 2a. If there are no account listed on the current card the system displays an appropriate error message (e.g., "NO ACCOUNT ACCESSIBLE WITH THIS CARD!") and goes back to Step 1.

Fig. 1e: Use Case describing display of available funds

APPENDIX 2: CLASS DIAGRAM

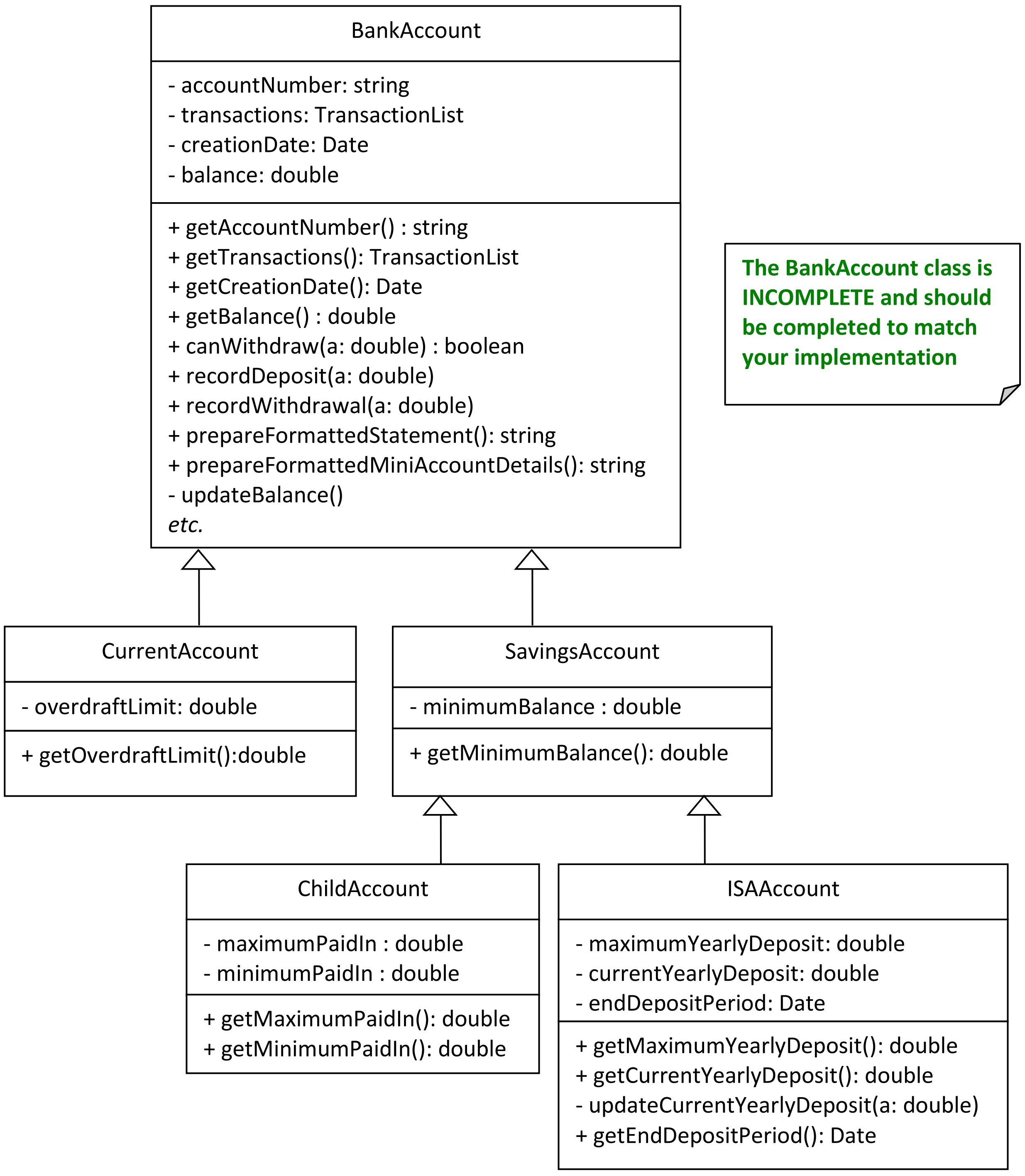

BankAccount
- accountNumber: string
- transactions: TransactionList
- creationDate: Date
- balance: double
+ getAccountNumber() : string
+ getTransactions(): TransactionList
+ getCreationDate(): Date
+ getBalance() : double
+ canWithdraw(a: double) : boolean
+ recordDeposit(a: double)
+ recordWithdrawal(a: double)
+ prepareFormattedStatement(): string
+ prepareFormattedMiniAccountDetails(): string
- updateBalance()
etc.
The BankAccount class is INCOMPLETE and should be completed to match your implementation
CurrentAccount
- overdraftLimit: double
+ getOverdraftLimit():double
SavingsAccount
- minimumBalance : double
+ getMinimumBalance(): double
ChildAccount
- maximumPaidIn : double
- minimumPaidIn : double
+ getMaximumPaidIn(): double
+ getMinimumPaidIn(): double
ISAAccount
- maximumYearlyDeposit: double
- currentYearlyDeposit: double
- endDepositPeriod: Date
+ getMaximumYearlyDeposit(): double
+ getCurrentYearlyDeposit(): double
- updateCurrentYearlyDeposit(a: double)
+ getEndDepositPeriod(): Date

Fig. 2: (Incomplete) Class Diagram showing the bank accounts hierarchy.

APPENDIX 3: SEQUENCE DIAGRAMS

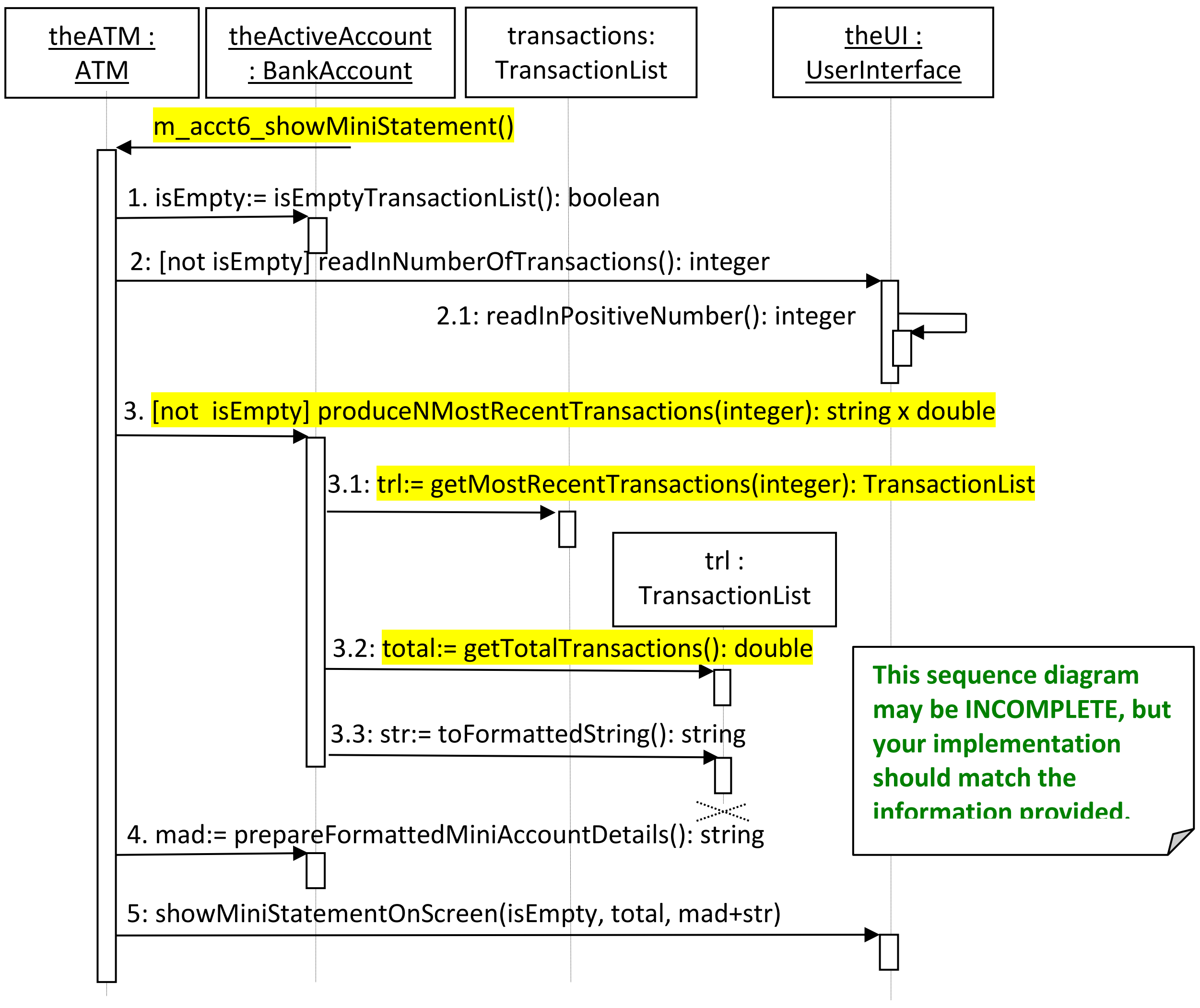

theATM : ATM
theActiveAccount : BankAccount
transactions: TransactionList
theUI : UserInterface
m_acct6_showMiniStatement()
1. isEmpty:= isEmptyTransactionList(): boolean
2: [not isEmpty] readInNumberOfTransactions(): integer
2.1: readInPositiveNumber(): integer
3. [not isEmpty] produceNMostRecentTransactions(integer): string x double
3.1: trl:= getMostRecentTransactions(integer): TransactionList
trl : TransactionList
3.2: total:= getTotalTransactions(): double
3.3: str:= toFormattedString(): string
4. mad:= prepareFormattedMiniAccountDetails(): string
5: showMiniStatementOnScreen(isEmpty, total, mad+str)
This sequence diagram may be INCOMPLETE, but your implementation should match the information provided.

Fig. 3a: (<u>Partial</u>) Sequence Diagram showing a mini statement

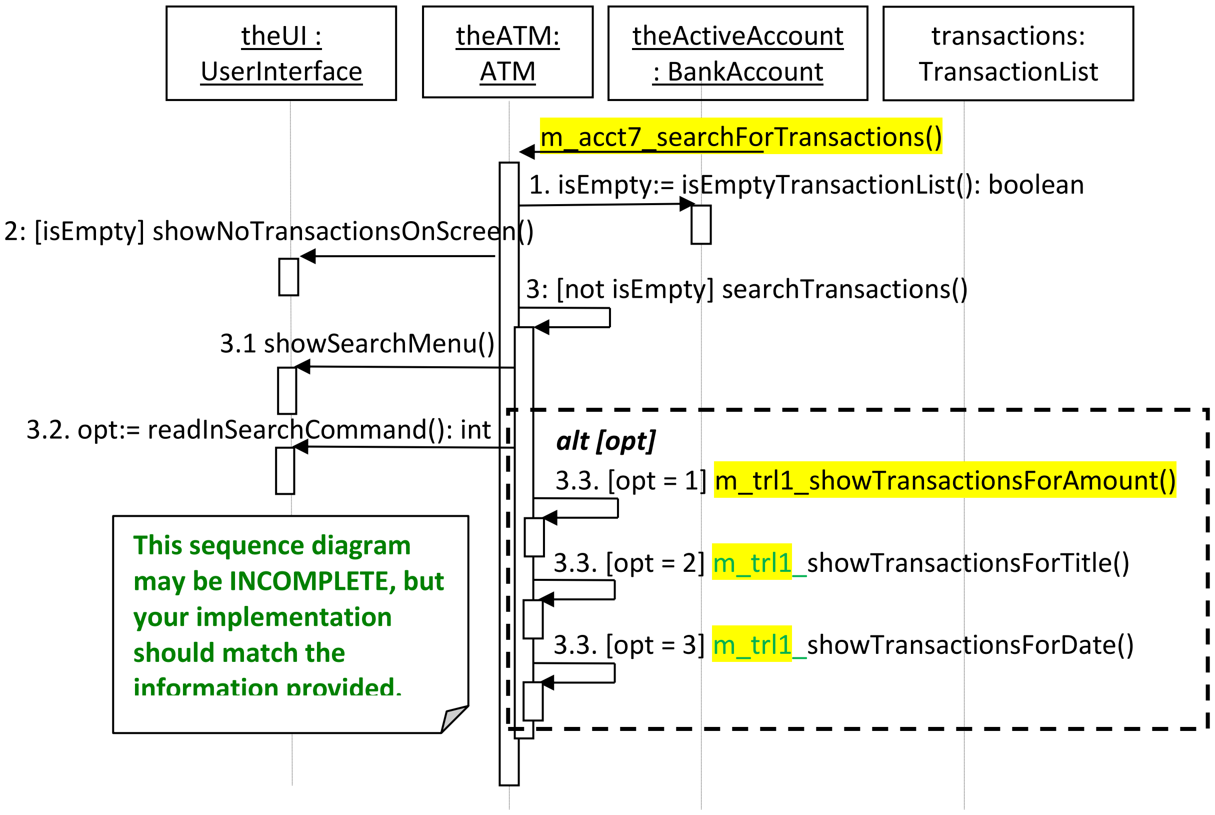

theUI : UserInterface
theATM: ATM
theActiveAccount : BankAccount
transactions: TransactionList
m_acct7_searchForTransactions()
1. isEmpty:= isEmptyTransactionList(): boolean
2: [isEmpty] showNoTransactionsOnScreen()
3: [not isEmpty] searchTransactions()
3.1 showSearchMenu()
3.2. opt:= readInSearchCommand(): int
alt [opt]
3.3. [opt = 1] m_trl1_showTransactionsForAmount()
3.3. [opt = 2] m_trl1_showTransactionsForTitle()
3.3. [opt = 3] m_trl1_showTransactionsForDate()
This sequence diagram may be INCOMPLETE, but your implementation should match the information provided.

Fig. 3b1: (<u>Partial</u>) Sequence Diagram showing transactions search for a given criterion (option 7)

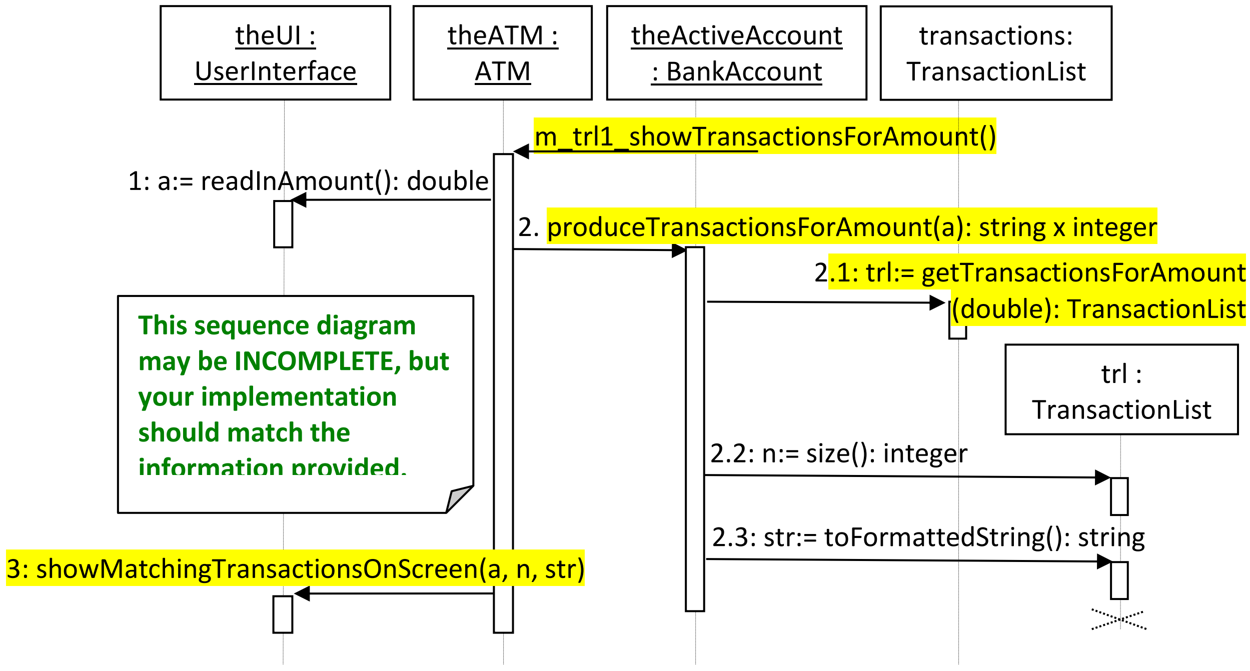

theUI : UserInterface
theATM : ATM
theActiveAccount : BankAccount
transactions: TransactionList
m_trl1_showTransactionsForAmount()
1: a:= readInAmount(): double
2. produceTransactionsForAmount(a): string x integer
2.1: trl:= getTransactionsForAmount (double): TransactionList
trl : TransactionList
2.2: n:= size(): integer
2.3: str:= toFormattedString(): string
3: showMatchingTransactionsOnScreen(a, n, str)
This sequence diagram may be INCOMPLETE, but your implementation should match the information provided.

Fig. 3b2: (<u>Partial</u>) Sequence Diagram showing details of transactions for a given amount

NOTE: The validity of the search date needs to be checked as in option 8.

NOTE: If you use templates to implement some of the functionality in option 7 you may need to rename some of the messages above (e.g., function **readInAmount** could become function template **readInSearchCriterion**, function **produceTransactionsForAmount** could be function template **produceTransactionsForSearchCriterion** and function **getTransactionsForAmount** could be function template **getTransactionsForSearchCriterion**).

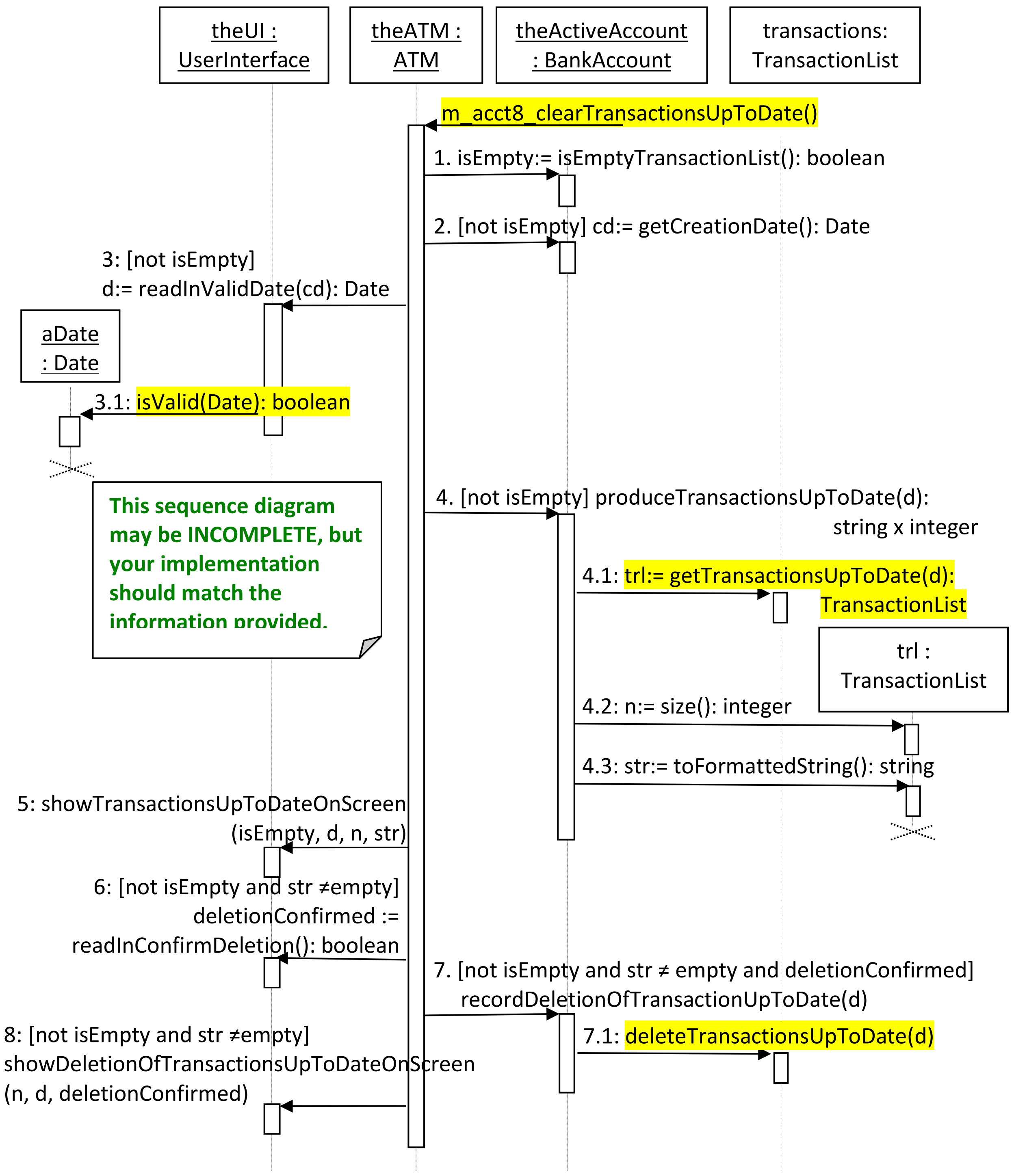
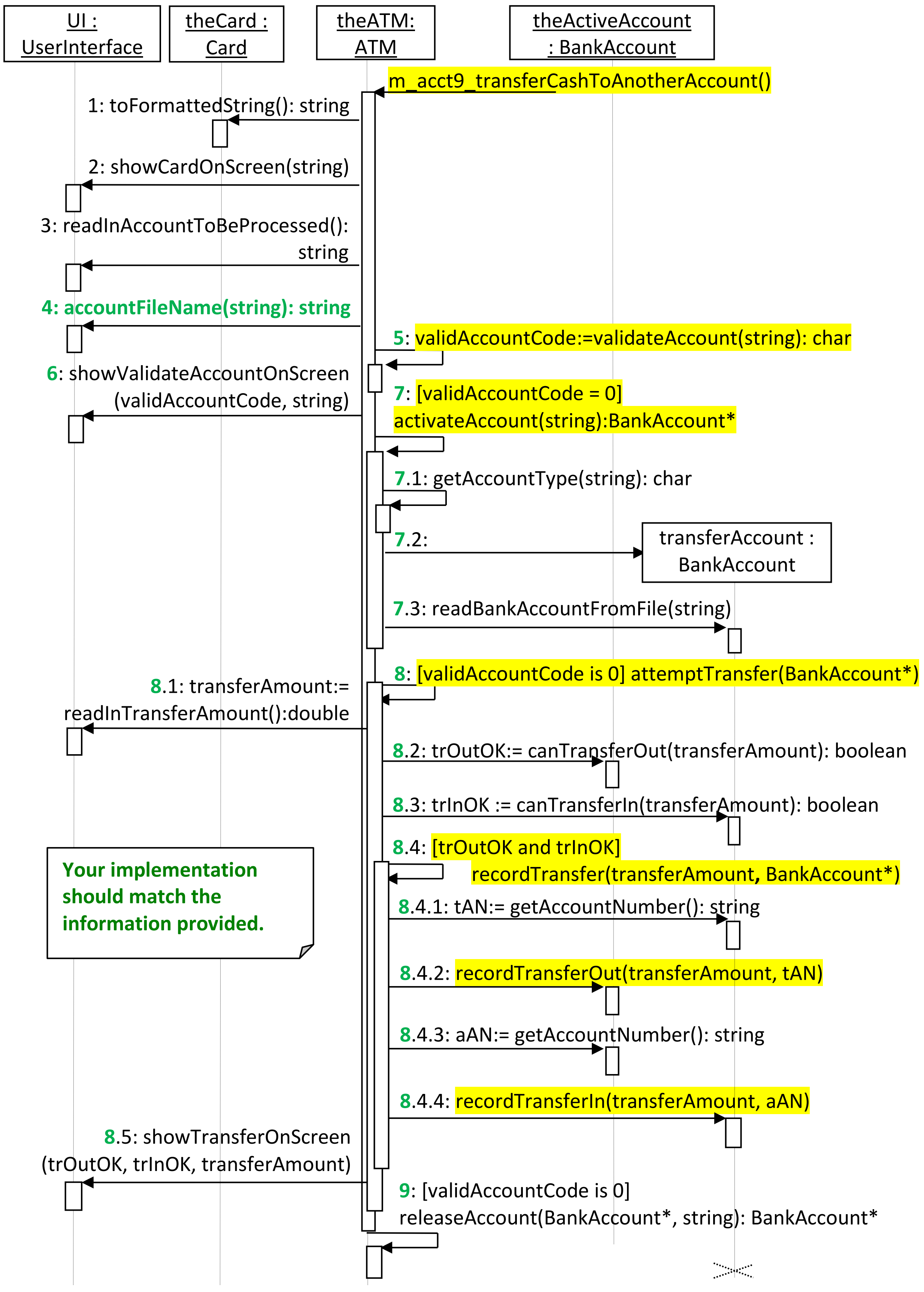

theUI : UserInterface
theATM : ATM
theActiveAccount : BankAccount
transactions: TransactionList
m_acct8_clearTransactionsUpToDate()
1. isEmpty:= isEmptyTransactionList(): boolean
2. [not isEmpty] cd:= getCreationDate(): Date
3: [not isEmpty] d:= readInValidDate(cd): Date
aDate : Date
3.1: isValid(Date): boolean
This sequence diagram may be INCOMPLETE, but your implementation should match the information provided.
4. [not isEmpty] produceTransactionsUpToDate(d): string x integer
4.1: trl:= getTransactionsUpToDate(d): TransactionList
trl : TransactionList
4.2: n:= size(): integer
4.3: str:= toFormattedString(): string
5: showTransactionsUpToDateOnScreen (isEmpty, d, n, str)
6: [not isEmpty and str ≠empty] deletionConfirmed := readInConfirmDeletion(): boolean
7. [not isEmpty and str ≠ empty and deletionConfirmed] recordDeletionOfTransactionUpToDate(d)
7.1: deleteTransactionsUpToDate(d)
8: [not isEmpty and str ≠empty] showDeletionOfTransactionsUpToDateOnScreen (n, d, deletionConfirmed)

Fig. 3c: (Partial) Sequence Diagram showing clearing of transactions up to a given date

NOTE: The **isValid** operation of the **Date** class returns true if the date it is applied to is valid (i.e., it has 31 days in Jan, 28 in Feb (not taking into account leap years), 31 in March, etc. and if it is a date between the date currently passed as a parameter and the current date). This will be used to check that the date is after the creation date of the account and before the current date (included).



UI : UserInterface
theCard : Card
theATM: ATM
theActiveAccount : BankAccount
m_acct9_transferCashToAnotherAccount()
1: toFormattedString(): string
2: showCardOnScreen(string)
3: readInAccountToBeProcessed(): string
4: accountFileName(string): string
5: validAccountCode:=validateAccount(string): char
6: showValidateAccountOnScreen (validAccountCode, string)
7: [validAccountCode = 0] activateAccount(string):BankAccount*
7.1: getAccountType(string): char
7.2:
transferAccount : BankAccount
7.3: readBankAccountFromFile(string)
8: [validAccountCode is 0] attemptTransfer(BankAccount*)
8.1: transferAmount:= readInTransferAmount():double
8.2: trOutOK:= canTransferOut(transferAmount): boolean
8.3: trInOK := canTransferIn(transferAmount): boolean
8.4: [trOutOK and trInOK] recordTransfer(transferAmount, BankAccount*)
Your implementation should match the information provided.
8.4.1: tAN:= getAccountNumber(): string
8.4.2: recordTransferOut(transferAmount, tAN)
8.4.3: aAN:= getAccountNumber(): string
8.4.4: recordTransferIn(transferAmount, aAN)
8.5: showTransferOnScreen (trOutOK, trInOK, transferAmount)
9: [validAccountCode is 0] releaseAccount(BankAccount*, string): BankAccount*

Fig. 3d1: (Partial) Sequence Diagram showing transfer of funds between two bank accounts

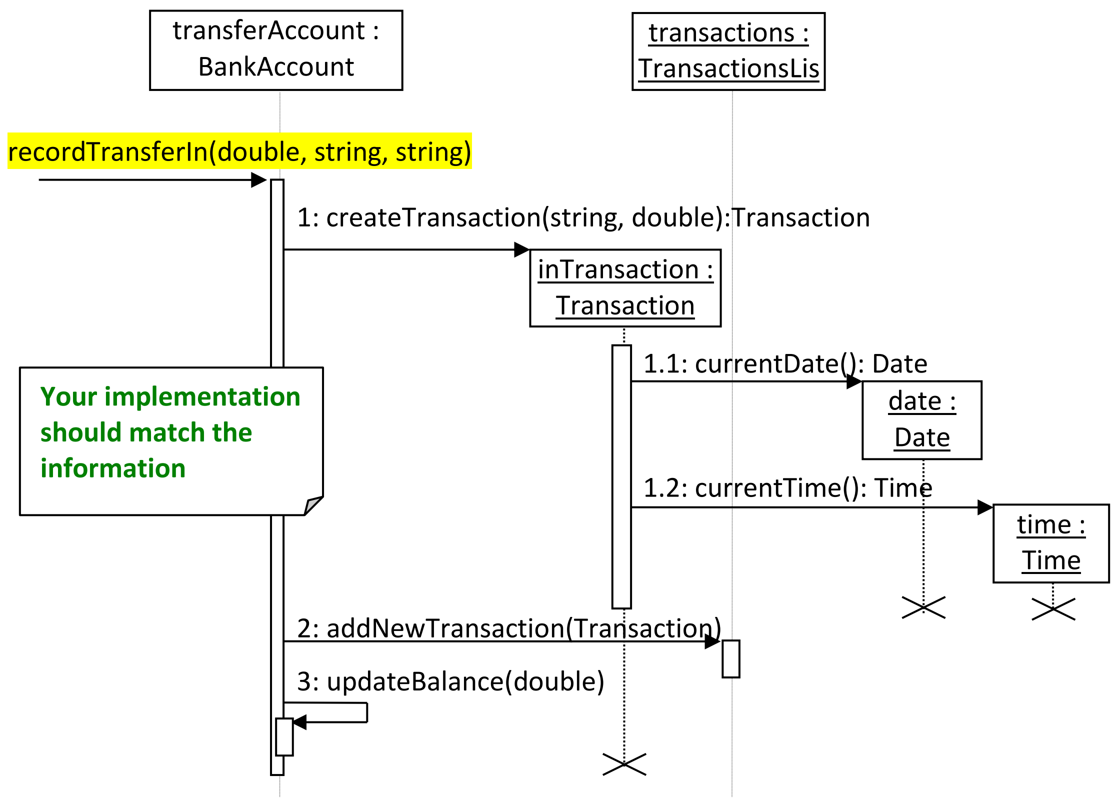

transferAccount : BankAccount
transactions : TransactionsLis
recordTransferIn(double, string, string)
1: createTransaction(string, double):Transaction
inTransaction : Transaction
1.1: currentDate(): Date
date : Date
Your implementation should match the information
1.2: currentTime(): Time
time : Time
2: addNewTransaction(Transaction)
3: updateBalance(double)

Fig. 3d2: (Partial) Sequence Diagram showing record of authorised funds transfer

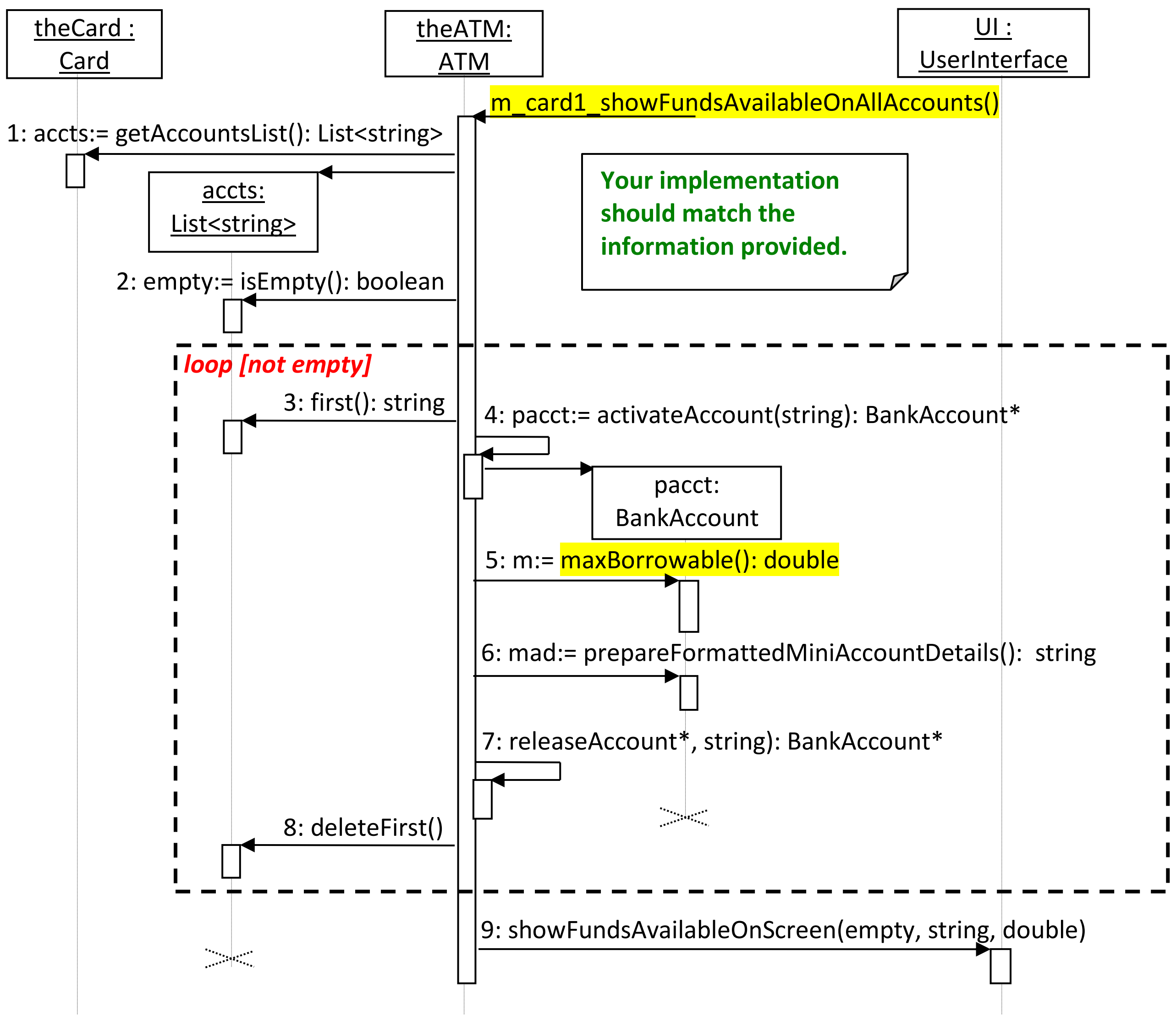

theCard : Card
theATM: ATM
UI : UserInterface
m_card1_showFundsAvailableOnAllAccounts()
1: accts:= getAccountsList(): List<string>
accts: List<string>
Your implementation should match the information provided.
2: empty:= isEmpty(): boolean
loop [not empty]
3: first(): string
4: pacct:= activateAccount(string): BankAccount*
pacct: BankAccount
5: m:= maxBorrowable(): double
6: mad:= prepareFormattedMiniAccountDetails(): string
7: releaseAccount*, string): BankAccount*
8: deleteFirst()
9: showFundsAvailableOnScreen(empty, string, double)

Fig. 3e: (Partial) Sequence Diagram showing the available funds on all accounts

APPENDIX 4: TEST DATA

Here is the test data that will be used when demo'ing your programs. You should create appropriate text files for the following cards and bank accounts and include the files for these cards and accounts with your electronic submission.

For Card

File name	card_01.txt	card_02.txt	card_03.txt	card_04.txt
Card Number	01	02	03	04
Accounts listed with that card	001 002 003	001 101 102 201 202 301 401	101 102 301 302 401 402	

For BankAccount *

File name	account_001.txt	account_002.txt	account_003.txt
Account type**	BANK	BANK	BANK
Account number	001	002	003
Creation date	1/1/2017	10/12/2017	10/12/2017
Balance	100	200	1000
List of transactions	No transactions	At least 3 transactions	At least 3 transactions (all withdrawals)

For CurrentAccount

File name	account_101.txt	account_102.txt	account_103.txt
Account type**	CURRENT	CURRENT	CURRENT
Account number	101	102	103
Creation date	1/1/2017	10/12/2017	10/12/2017
Balance	100	200	1000
Overdraft limit	50	100	50
List of transactions	No transactions	At least 3 transactions	At least 3 transactions (all withdrawals)

For Saving Account *

File name	account_201.txt	account_202.txt
Account type* *	SAVINGS	SAVINGS
Account number	201	202
Creation date	1/1/2017	10/12/2017
Balance	100	200
Minimum balance	50	100
List of transactions	No transactions	At least 3 transactions

For ChildAccount

File name	account_301.txt	account_302.txt
Account type**	CHILD	CHILD
Account number	301	302
Creation date	1/1/2017	10/12/2017
Balance	100	200
Minimum balance	50	100
Minimum paid in	50	100
Maximum paid in	100	2000
List of transactions	No transactions	At least 3 transactions

For ISAAccount

File name	account_401.txt	account_402.txt
Account type**	ISA	ISA
Account number	401	402
Creation date	1/1/2017	10/12/2017
Balance	100	200
Minimum balance	50	100
Maximum yearly deposit	500	1000
Current yearly deposit	450	900
End deposit period**	1/1/2018	10/12/2018
List of transactions	No transactions	At least 3 transactions

NOTE: *If you have made the* ***BankAccount*** *and* ***SavingsAccount*** *classes abstract – as requested, then of course you won't use the test files for these classes. They are just there in case your group has not progressed that far in the development of a solution for question 4.*

*****NOTE:*** *The data in the "Account type" and "End deposit period" fields is not stored in files but calculated when the account is created.*

APPENDIX 5: MARKING SCHEME

Question		Marks
1	**Answer to questions about UML and C++ code provided**	**20%**
2	**UML diagrams**	
2a	Sequence Diagram	**5%**
2b	Class Diagram	**5%**
3	**C++** Include the code for the following header files **ATM.h, UserInterface.h, TransactionList.h, Date.h** and the following member functions definitions:	
3a	`ATM::am6_showMiniStatement`	**4%**
	`BankAccount::produceNMostRecentTransactions`	
	`TransactionList::getMostRecentTransactions`	
	`TransactionList::getTotalTransactions`	
3b	`ATM::m_acct7_searchTransactions`	**8%**
	`ATM::m_trl1_showTransactionsForAmount`	*templates*
	`BankAccount::produceTransactionsForAmount`	
	`TransactionList::getTransactionsForAmount`	
	`UserInterface::showMatchingTransactionsOnScreen`	
3c	`ATM::m_acct8_clearTransactionsUpToDate`	**8%**
	`Date::isValid`	*recursion*
	`TransactionList::getTransactionsUpToDate`	
	`TransactionList::deleteTransactionsUpToDate`	
3d	`ATM::m_acct9_transferCashToAnotherAccount`	**10%**
	`ATM::validateAccount`	
	`ATM::activateAccount`	
	`ATM::attemptTransfer`	
	`ATM::recordTransfer`	
	`BankAccount::recordTransferIn` and `BankAccount::recordTransferOut`	
3e	`ATM::m_card2_showFundsAvailableOnAllAccounts()`	**5%**
	`maxBorrowable():double` (from `CurrentAccount` class)	
4	**Inheritance in C++**	**20%**
4a	Include the code for the following header files: **BankAccount.h**, **CurrentAccount.h**, **SavingsAccount.h**, **ChildAccount.h**, **ISAAccount.h**	*abstract classes*
4b	**options 2** to **8** work properly with any **`BankAccount`** instance.	
4c	**options 9** and **10** works properly with any **`BankAccount`** instance.	
5	**More advanced design and programming techniques**	**15%**

5a	**Recursion**
	`TransactionList::getTransactionsUpToDate`
	`TransactionList::deleteTransactionsUpToDate`	

5b	**Templates** for some functions implementing option 7 in question 3b	
5c	Abstract classes for `BankAccount` and `SavingsAccount` classes	
5d	STL containers (to implement the `TransactionList` class)	
5e	Singleton pattern (for `UserInterface` class)	

APPENDIX 6: GROUP CONTRIBUTION FORM

This work is to be undertaken as a group. The groups will be self-created groups.

- There must be 3 (possibly 2 or 4) students per group. It may, exceptionally, be possible to work on your own if (a) you have special reasons for this and (b) this has been agreed this *IN ADVANCE* with your tutor. Outside this exceptional arrangement, individual solutions will be marked but cannot be awarded more than a pass mark (i.e., 40%).
- Any change in a group composition MUST be approved by all the group members and the teaching team.
- Typically, the same mark will be allocated to all members of the group. However, if you feel - as a group - that you can agree on a fairer ratio, the marks will be allocated pro-rata, taking into account the information you provide in this form to reflect different level of group contribution. This will be discussed and agreed at the final walkthrough.

This evaluation form is to be **signed by ALL members** and **handed in at the final walkthrough**.
To do this evaluation of each group member contribution:

1. Provide a short 'work description' (150 words max) highlighting what each of you has actually done. It should list the key parts of the reports to which each of you contributed or were responsible for.
2. Identify the group member(s) who you ALL agree has contributed the most and give that member (or these members) a value of 10. If you have all contributed equally, put a value of 10 for everyone. Anyone not listed on this form will get a mark of 0.
3. Evaluate on a scale of 0 to 10 the other member's individual contribution to the group work.

Your tutor will moderate this evaluation (using evidence from group meeting participation, walkthrough contribution and, where appropriate, version control logs).
The group mark will then be scaled accordingly (e.g., anyone with a 10 will get the full mark, someone with, let's say, an 8 with get 80% of the mark).

Group Number:		**Individual Contribution**	
Member name (printed) and signature	**Brief Description of work done**	**Group's evaluation**	**Tutor's evaluation**

***NOTE:** Please note that you are ultimately all responsible for the work produced and should ensure that you understand all aspects of this work as part of your learning for this module.*
Your individual knowledge and understanding of the work done will also be assessed in the Examination.