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.

1 a) Yes, the canWidthdraw function can be used on the ChildAccount class because ChildAccount inherits this function and the function is public. When implementing this design, there are problems. For example, in the base class canWidthdraw function, it compares the amount to withdraw is less than the balance. In the ChildAccount class, this will be implemented differently since this will need to take in account of the balance AND the minimum paid in and max paid in. So, the minimum amount of money must be deposited into the ChildAccount and cannot deposit more than the maximumPaidIn.

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.

1 b) make sense to use pointer instead of instance because if there is a BankAccount instance, copying another subclass to a base class means object slicing occurs. Some of the information from the subclasses of BankAccount is lost in the bank account pointer. However, using a pointer means the BankAccount pointer can point to a subclass and this means down casting can occur so information from the subclass is recovered and can be accessed.

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

1 c) The relationship between Card and a List<String> class is composition. This should be represented in UML with a black filled diamond going to the Card class. In order to implement this, the whole class (Card) has an instance of the part class (List of type string) as one of its data members. The part instance is created in the constructor of the whole instance and destroyed when the whole instance goes out of scope.

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

1 d) The **UserInterface** is not an abstract class because there is no pure virtual function that exists in the class. For example, there is no function that is marked as virtual and there is no “= 0” at the end of the function declaration. This should not be an abstract class because there is no class that inherits from user interface so it wouldn’t make any sense to make the class abstract.

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

1 e) The reason why this is declared as static is because every time a new transaction happens, there should be no need to create a Date instance just to call this function and to store the date. Therefore, a better way is to make this static. This works by calling the function through the class name and using the scope resolution operator (::).

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**

1 f)

Line 1 - Runs the constructor with h, m, s as parameter for the Time (0, 0, 12). The overloaded operator + is run which adds the seconds from the first time to the second time. Then conversion because the Time(long) has been returned from the overloaded + operator function. Then it leaves the operator + function **but then runs the destructor** **for** **Time (0, 0, 12)**

Line 2 – runs the conversion constructor to convert Time (12) into a time object. Then runs the overloaded + operator function which adds 12 and the seconds of t1. This returns a Time(long) so the conversion constructor is run again. The destructor is run to destroy Time (12).

These will work as long as the conversion constructor and the overloaded operator + function exists.

Line 3 – runs the conversion constructor to convert 12 into a **temporary** Time object. This will now call the operator + function. This will call the conversion constructor again to add seconds from both time objects. Then the destructor is run to remove the temporary Time object. The call to operator + function should not happen or this could be unexpected. In order to fix this issue, make the conversion constructor explicit so that the user has to create a temporary time object with the value of 12 to run the conversion constructor compared to just explicitly typing in 12 as an integer.

Line 4 – This will not work as there is no overloaded + function that takes 12 as the first parameter and a time object for the second. To fix this, create a function outside the class that overloads the + operator. The parameter for this function would be an int and a const Time& object. Then inside the function definition, return Time(timeobject.getSeconds() + int).

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);**

**}**

**}**

1 g)

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

1 h) this allows the subclasses to redefine this method but also if there is a bank account that points to a subclass at runtime, the compiler will wait to see what method it should call (the method that the pointer points to) compared to calling the method that is from the pointer’s type. The subclasses may have different ways of implementing the method for example, the current account will contain an overdraft limit and the saving accounts contains a minimum balance this will be a part of the account details so each of them will be different to the bank account class and this is the reason to use virtual on the method.

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

1 I) No because the pointer has only got access to the method of the pointer type (i.e. the bank account methods) and not the subclass methods. In order to fix this, downcast the base pointer to a derived class pointer using dynamic cast which allows the access of attributes and functionality only for the derived class. Then the method can be accessed using the derived class pointer.

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\_);**

**}**

1 j) No because balance\_ is a private member variable in the bank account class and so this cannot be accessed through the active account pointer. Also, there is no check to see if the pointer is null and so this may cause unexpected behaviour in the program.