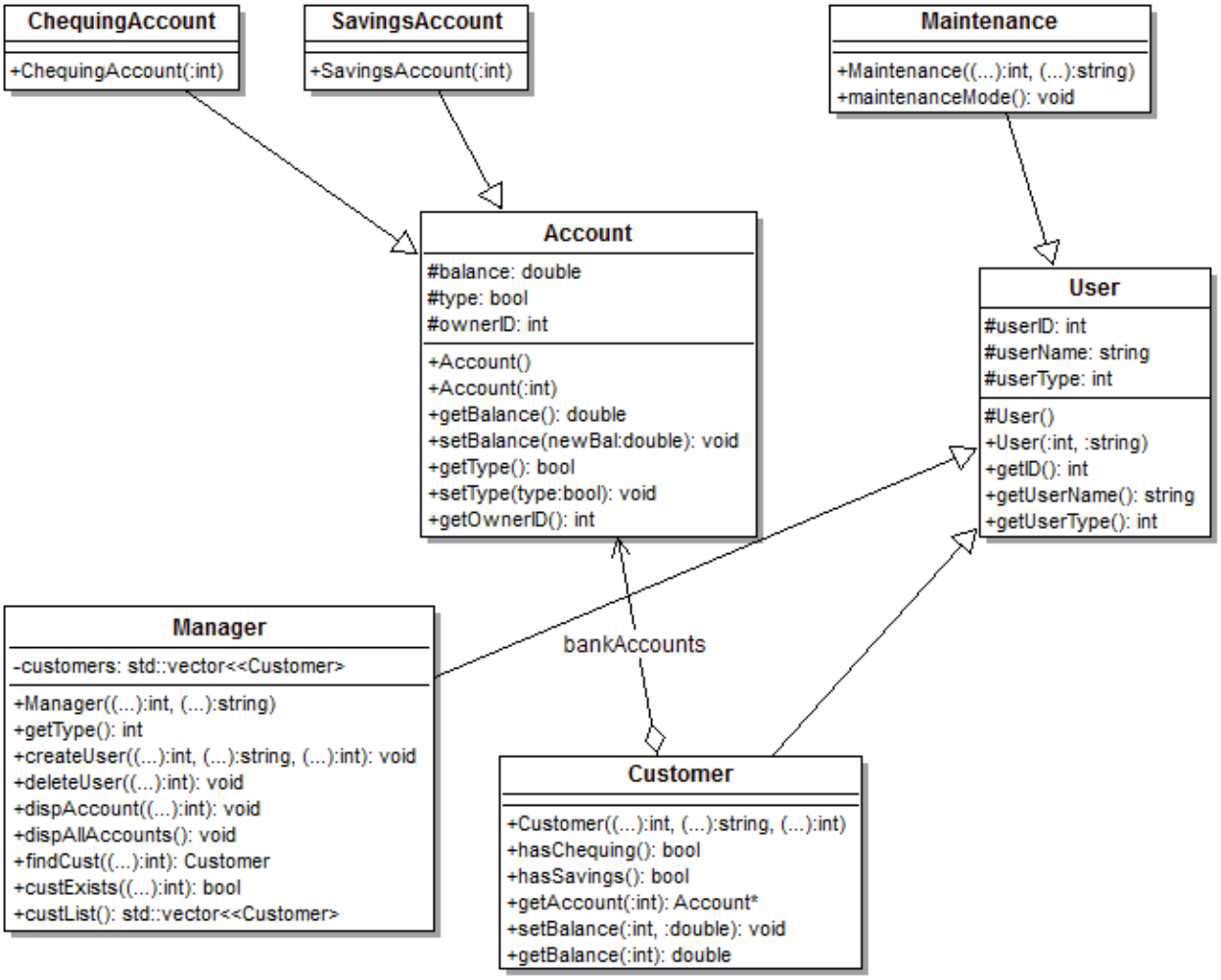
QUIZ: OO CONCEPTS, ANALYSIS AND MODELS
CS 3307 – OBJECT ORIENTED ANALYSIS AND DESIGN

GROUP 12
ALEX MACLEAN, amacle45@uwo.ca
ERNIE KWAN, ekwan6@uwo.ca

WEDNESDAY, OCTOBER 15th, 2014

(1) (i) [D1 –Design]



(1) (ii) [D2 – Class reference in code] and (1) (iv) [D4 – Operations within classes]

Class	Class Reference File	Class Reference Line	Operations
Account	Account.h	#4	getBalance(): account.cpp, line 10 setBalance(double): account.cpp, line 14
ChequingAccount	Account.h	#20	
SavingsAccount	Account.h	#25	

Class	Class Reference File	Class Reference Line	Operations
User	User.h	#11	
Maintenance	User.h	#60	
Manager	User.h	#44	createUser(int, string, int): user.cpp, line 138 dispAccount(int): user.cpp, line 177
Customer	User.h	#25	

(1) (iii) [D3 – Operations within classes]

Account class:
- Account()
- Account(int)
- getBalance()
- setBalance(double)
- getType()
- setType(bool)
- getOwnerID()

ChequingAccount class:
- ChequingAccount(int)

SavingsAccount class:
- SavingsAccount(int)

User class:
- User()
- User(int, string)
- getID()
- getUserName()
- getUserType()

Customer class:
- Customer(int, string, int)
- hasChequing()

- hasSavings()
- getAccount(int)
- setBalance(int)
- getBalance(int)

Manager class:
- Manager(int, string)
- getType()
- createUser(int, string, int)
- deleteUser(int)
- dispAccount(int)
- dispAllAccounts()
- findCust(int)
- custExists(int)
- custList()

Maintenance class:
- Maintenance(int, string)
- maintenanceMode()

(1) (iv) [D5 – Classes with important relationships]

The Customer, ChequingAccount, and SavingsAccount classes definitely have important relationships.

(1) (iv) [D6 – Justification of importance]

This set of classes is important because they are the main focus of the bank system. The whole reason for this program to exist is so a customer can access her bank accounts and perform transactions with them. Their relationships need to be working properly or else our software will not be of any use. A bank with a good system in place will have happy customers because of the enjoyable experience they have with the program. This is a more important relationship than say, the Manager and Maintenance classes. Those are two classes that don't have very much to do with each other at all.

(1) (iv) [D7 – Relationships: what and why]

Well, a Customer must be able to withdraw and deposit from either a ChequingAccount or a SavingsAccount. A Customer must also be able to transfer funds back and forth between the accounts if they both exist. If a customer does not have one of the accounts she has the ability to open it. Also, if an account has a balance of $0 she can close it.

(2) (i) [D8 – Correspondence between problem statement and design elements]

Problem Statement	OO Design Element
Each user of the system, regardless of their role type, has an "id" for secure login. After logging into the system, they will be bounded by their role as to what they can do with this system.	Knowing that every user of the system will need a unique id led us to create a user class that has a name, unique id and user type property. We de-coupled each role type by extending the User class and adding unique functionality to each subclass. For example, the Maintenance user has no accounts so we opted to only add a maintenanceMode method. Whereas the Customer subclass requires the ability to deposit and withdraw money and the Manager subclass requires the ability to open and close accounts. By separating these unique functionalities into their own subclasses we increase cohesion by ensuring relevant properties and functions are grouped together.
There is a bank manager who has managerial powers to open and close an account and see the critical details of a particular, or all, customers in a formatted display.	In order to provide the Manager with the power they need to perform their role the Manager subclass was provided with a vector of all the Customers. This created a 'one-to-many' scenario, with the assumption that there was only one manager and many customers. Functionally this allowed the Manager to search through all the Customers objects included adding or deleting.

Problem Statement	OO Design Element
The bank has an unspecified number of customers. For demonstration purposes, you can have a manageable number of customers. The term "client" is a synonym for customer. Each customer has either or both of a savings (S) account and a chequing (C) account.	A customer in our program "is-a" User, but to give the customer the ability to retrieve information we added additional functions to the class. This decoupled the Customer class from the Manager and Maintenance and give us a more cohesive and flexible framework to build upon.
A customer has power to use only their account within the permissible operations. Note that a customer can use multiple operations in one session of system use.	Our form of a secure login came in the form of a unique ID number. While we didn't implement a login/password system we did ensure that once 'logged in' a user can only perform operations within their own account. In lines **183-184** of main.cpp we check if the user exists and if so we use the manager object to load the specified customer object into 'userAccount'. From there on we only manipulate the loaded customer object. We did however overlook the implementation of the customer functions. By copying the customer into a new customer object and manipulating it we never make changes to the object within the vector. By not copying the object back to the vector, all the edits are lost once the user logs out. We should have made the design decision to move the Customer vector to another class to ensure more cohesion within the Manager class.

(2) (ii) [D9 – Assessment of design adequacy]

While the project specifications asked for a System class that would display error messages we opted instead to print the error messages when needed. This was done due to time constraints, but we understand the flexibility offered by a separate System class. We also think adding a BankBranch container class would better fulfill the requirements laid out in Assignment 1.

(2) (iii) [D10aA – Improved design]

As a revision to our current program we would add two classes, a System class and a BankBranch class. In addition we would change the way a Customer accesses and edits their accounts.

Adding a System class to catch errors would give us much more flexibility when printing out messages. Rather than having *main* 'cout' a message at every error we could have called a System class with the message to print. This would make our code more readable and provide us with a standard protocol to print out errors.

We also believe a separate BankBranch class that contained all the users of the system would be an improvement. Our current program is designed for one Manager and one Maintenance worker while a BankBranch class would allow us to contain all the users, including Managers, Customers, and Maintainers. It would also allow us to add future employees, such as; tellers, financial advisors, or security workers, under the BankBranch class and sets us up for a ProvincialBank superclass and higher. Decoupling the Customer class from the Manager would also help us with the oversight mentioned in **D8**.

(2) (iii) [D10aB – Table of improved design correspondence]

Problem Statement	Revised OO Design Element
There is a Manager, Maintenance worker, and an unspecified number of Customers. The manager has managerial power over the customers, and the customers have power over their own accounts.	With a BankBranch class we can decouple the Customer vector from the Manager. Then contain and manage all the users using the new class. Having all the users in one class increases cohesion and gives managerial powers to the Manager without coupling the Customer class to the Manager class. Having this container class increases extendibility within the system. We can structure our bank for multiple maintenance workers, and managers, and even create a superclass for regional, or nationwide banks.
The system should give warning messages for insufficient funds, nonexistent account, withdrawal from a savings account with less than $1000…etc	A System class that catches and outputs errors decreases coupling within our code. Instead of littering our code with cout statements we can call a function with a message within the System class. This also increases extensibility for future warning messages and provides a consistent method of outputting errors.

(3) (i) [D11 – Separation of concerns]

In main.cpp it will do nothing but ask for input and call the appropriate methods. The business logic of checking whether accounts and users exist or not will be included in the user.cpp file. This will give each class a more defined responsibility than including some business logic in main.cpp and some in user.cpp. If we decide to add more methods later on it will be easier to follow the pattern set by the existing methods than having to see what main.cpp is checking for.

(3) (ii) [D12 – Strength of inter- vs. intra-component relationships]

The strength of inter-components such as the ChequingAccount class and the Account class is quite strong. This is because the Account class is the super class of ChequingAccount. It inherits the attributes and methods of Account and thus increases the coupling between the classes since the functionality of ChequingAccount is dependant on how Account was designed. Other classes such as SavingsAccount and Maintenance are quite separated though. One does not have an influence over the other. This limits the coupling between the classes and keeps their concerns separate. Intracomponent relationships within classes are as cohesive as possible. Every class contains the necessary attributes and methods for it to perform the required actions in our program. The user classes do rely on the account classes though because the account classes keep track of the balances.

(3) (iii) [D13 – Principles of software design]

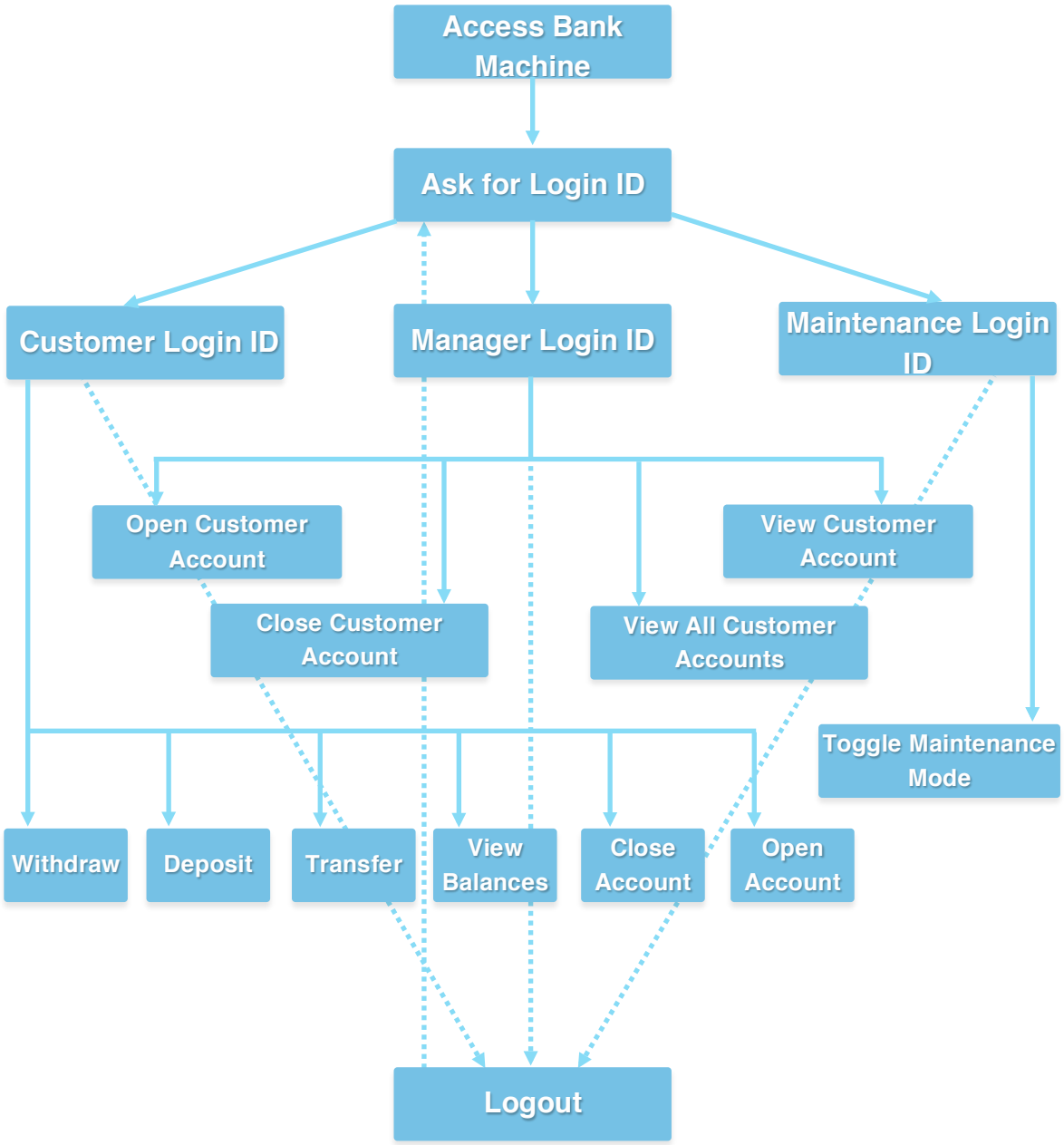
Yes, our design is in line with the principles of coupling and cohesion. Coupling between classes is minimal except for the cases where inheritance is used. Functional components of each class are kept inside of the proper classes whenever possible though. Cohesion is also used well since the attributes of one object are not spread across different classes. The functions contained within each class directly impact that class, as they should.

(3) (iv) [D14 – Rationale for principle]

Coupling is not desired because it complicates a system by making modules harder to understand, change, or correct since it is highly interrelated with other modules. When altering classes that are highly coupled it is likely that changes will need to be made in both classes, which doubles the work required compared to modifying a class that is not coupled with other ones.

Cohesion measures the degree of connectivity among the elements of a single module. Elements of a class or module should all work together to provide some well-bounded behaviour. It is not desirable for a mix of unrelated elements to be clumped together though. The object being modelled by a class should represent the object well but pertain to nothing more than the object in question. Other objects should belong to their own class or module.

(4) (i) [D15 – Functional decomposition]



(4) (ii) [D16 – Functional vs. OO]

This is a quite a different way of laying out our program. It really just focuses on the requirements and does not give any information about the inter-relationships between classes. Since we wrote our program with an OOP language this diagram does not full represent our design. If we used a functional programming language it would be more appropriate, but the UML diagram is much more descriptive. A UML diagram shows the location of methods by including them in each class container while this leaves their

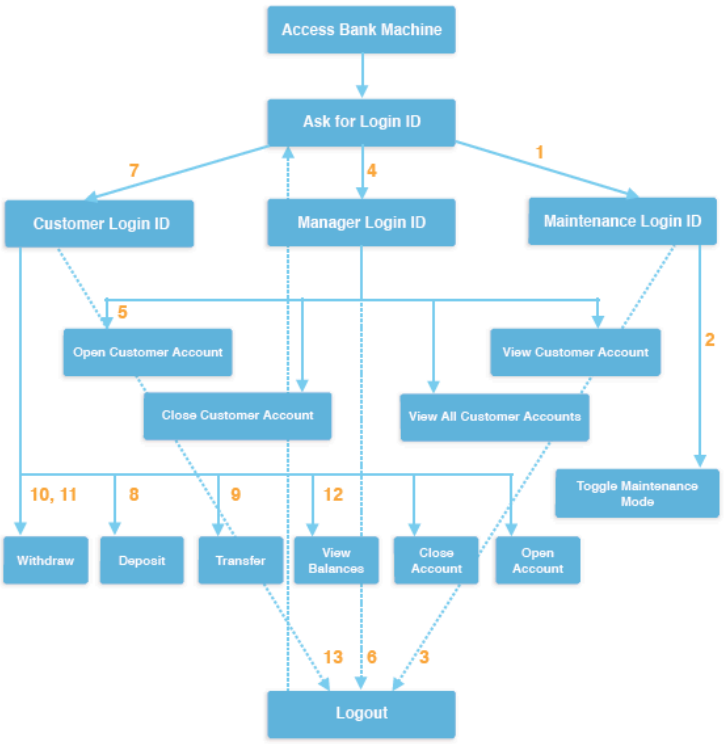
location ambiguous. The main functions of the program are included in a Functional Decomposition but more information is very helpful.

(5) (i) [D17 – Scenario – user and system aspects]

Steps	User Actions	Internal Operations
1	User logs in with Maintenance ID (2345678)	• Checks if ID matches maintenance ID • loginID == main.getID()
2	Turns on execution trace	• Call execution trace function to print timestamp to file. • Set bool to 1 for on
3	User logs in with Manager ID (123456)	• Checks if ID matches manager ID • loginID == man.getID()
4	Create a new Customer ID (98765)	• Checks if customer already exists, if not store new ID
5	Enter the new Customer's name (Ernie)	• Store new customer name
6	Enter type of account Customer would like (Both, 3)	• Store account type variable (Chequing, Savings or Both) • Create new Customer object with the three variables, store in vector
7	Logout	• goto start
8	Enter new Customer ID (98765)	• Load new Customer object using findCust(loginID)
9	Set a new checking balance (2000)	• userAccount.setBalance(0, newBalance)

Steps	User Actions	Internal Operations
10	Transfer money from Checking to Savings (1000)	• Check if there are enough funds to transfer • Set new balances for each account
11	Withdraw money from Savings (2000)	• Check if there are enough funds to withdraw • There is not; print ‘insufficientFund warning’ • Try again
12	Withdraw money from Savings (500)	• Check if there are enough funds to withdraw • There is, set new balance • Withdraw money
13	Display checking balance	• Check if account exists • If it does, display current balance • getBalance()
14	Logout	• goto start

(5) (ii) [D18 – Execution path on functional decomposition diagram]



Execution Path Number	Information Flow
1	Login ID □ Maintenance LoginID
2	-
3	-
4	Login ID □ Manager LoginID
5	new Customer ID, name, Account Type □ Open Customer Account
6	-

Execution Path Number	Information Flow
7	Login ID □ Customer ID
8	(Amount to Deposit) □ Deposit function
9	(Amount to Transfer) □ Transfer function
10	(Amount to Withdraw) □ Withdraw function
11	(Amount to Withdraw) □ Withdraw function
12	(Account Type) □ View Account
13	-

(5) (iii) [D19 – Execution trace from the program]

Execution trace on | 234567 | Tue Oct 14 00:06:37 2014
custExists | 123456 | Tue Oct 14 00:06:51 2014
setCheckBalance | 98765 | Tue Oct 14 00:07:00 2014
transferCheckToSave | 98765 | Tue Oct 14 00:07:07 2014
savingsWithdrawal | 98765 | Tue Oct 14 00:07:14 2014
insufficientFunds | 98765 | Tue Oct 14 00:07:14 2014
savingsWithdrawal | 98765 | Tue Oct 14 00:07:16 2014
displayChecking | 98765 | Tue Oct 14 00:07:20 2014

(5) (iv) [D20 – Comparison of execution traces]

The execution trace that was generated when we ran the user sequence does not match exactly with the execution path on the functional decomposition diagram due to the way we implemented our execution trace. Our execution trace does not explicitly say when a user has logged in or out because we display the user ID next to the function called; so a change in user is implied when the user ID changes. We also trace when the system prints

a message, something that we don't account for in our functional decomposition because we lacked the System class (touched up in **D10aA**).

So the difference between our execution trace and functional decomposition diagram can be fully accounted for by the lack of explicit login print outs and the lack of warning messages in the diagram.

(6) (i) [D21 – Categories of objects in the banking domain]

- Bank teller
- ATM user
- Bank manager
- ATM maintenance person
- Security guard
- ATM
- Vault
- Credit card
- Debit card
- Chequing account
- Savings account
- Line of credit

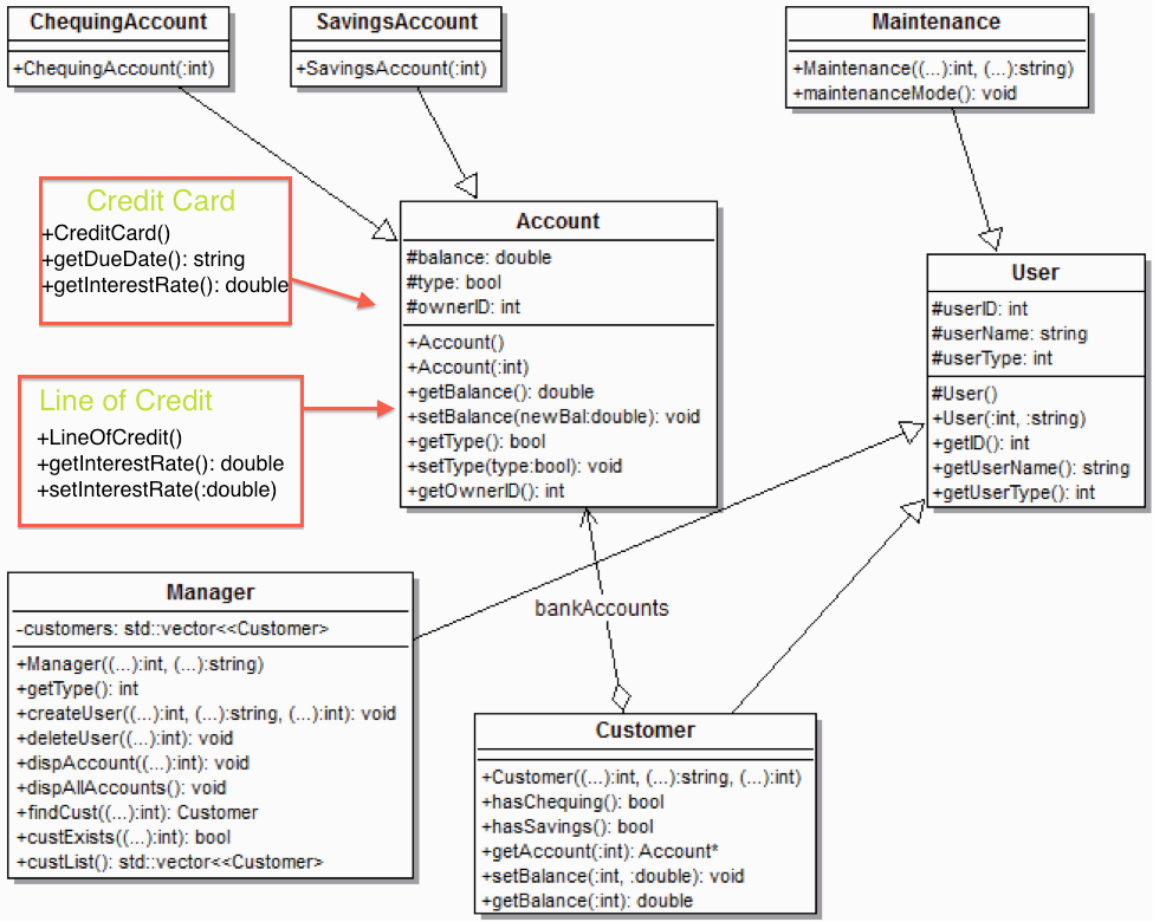
(6) (ii) [D22 – Representation of object types]

- ATM user
- Bank manager
- ATM Maintenance person
- Chequing account
- Savings account

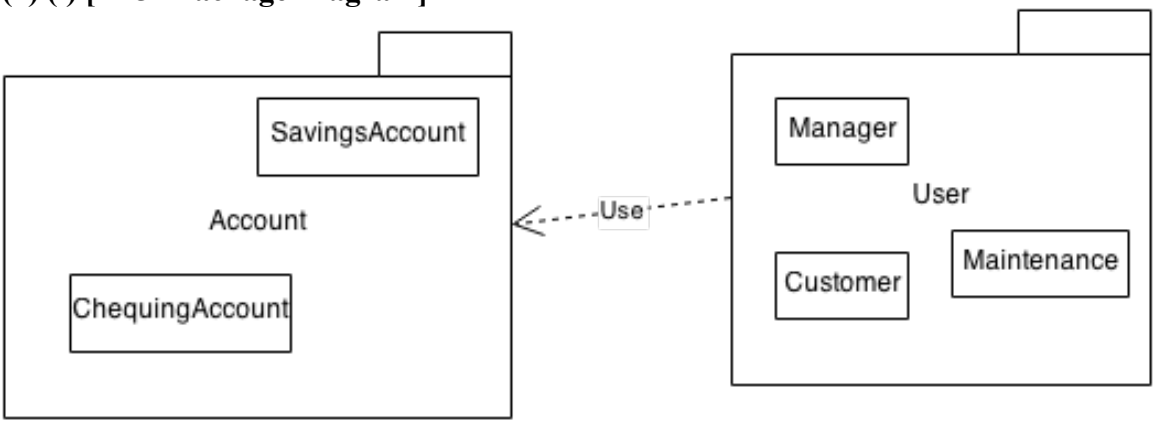
(6) (iii) [D23 – System enhancement – analysis]

New Objects	Old Objects	Inter-relationships
Credit Card	Customer, Manager	Customer would have another account he/she could interact with. Manager could open/close it.
Line of Credit	Customer, Manager	Another account customer would have to interact with. Manager could open/close it and set interest rates.

(6) (iii) [D24 – System enhancement – design]



(7) (i) [D25 –Package Diagram]



(7) (ii) [D26 –Criteria for packaging and justification]

Since SavingsAccount and ChequingAccount are both accounts I put them in an Account package. And since the Manager, Customer, and Maintenance classes are all subclasses of User I put them in a user package. In general, User classes use Account classes.

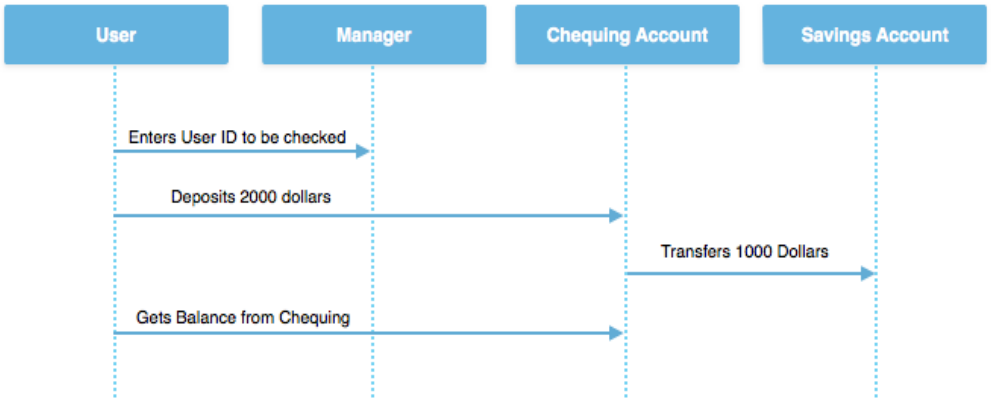
(7) (iii) [D27 –High-level design vs. Low-level design]

The package diagram is much more general than the class diagram. It groups classes together by common characteristics. The class diagram includes more details about the methods and attributes for each class. The package diagram just helps to get a very high-level overview of the different groups of the program.

(7) (iv) [D28 – Scenario cutting across several packages]

- User logs in with Maintenance ID
- User turns execution trace on
- User logs out
- User logs in with Manager ID
- User creates new Customer User with both account types
- User logs out
- User logs in with new Customer ID
- User deposits 2000 into Chequing Account
- User transfers 1000 into Savings Account
- User withdraws 2000 from Savings Account
- System gives insufficient funds warning
- User withdraws 500 from Savings Account
- User views balance of Chequing Account
- User logs out

(7) (v) [D29 – Sequence Diagram]



(8) [D30 – Lessons Learnt]

1. Decrease the coupling between classes more. For example we could have put the list of customers in a class other than the Manager class, like a container class called BankBranch that would exist even if different managers access the system.
2. It would have been helpful to spend more time on the design phase.
3. There are many ways to develop a program given a set of requirements.