CS3307 – Object Oriented Analysis and Design

Quiz: OO Concepts, Analysis and Models

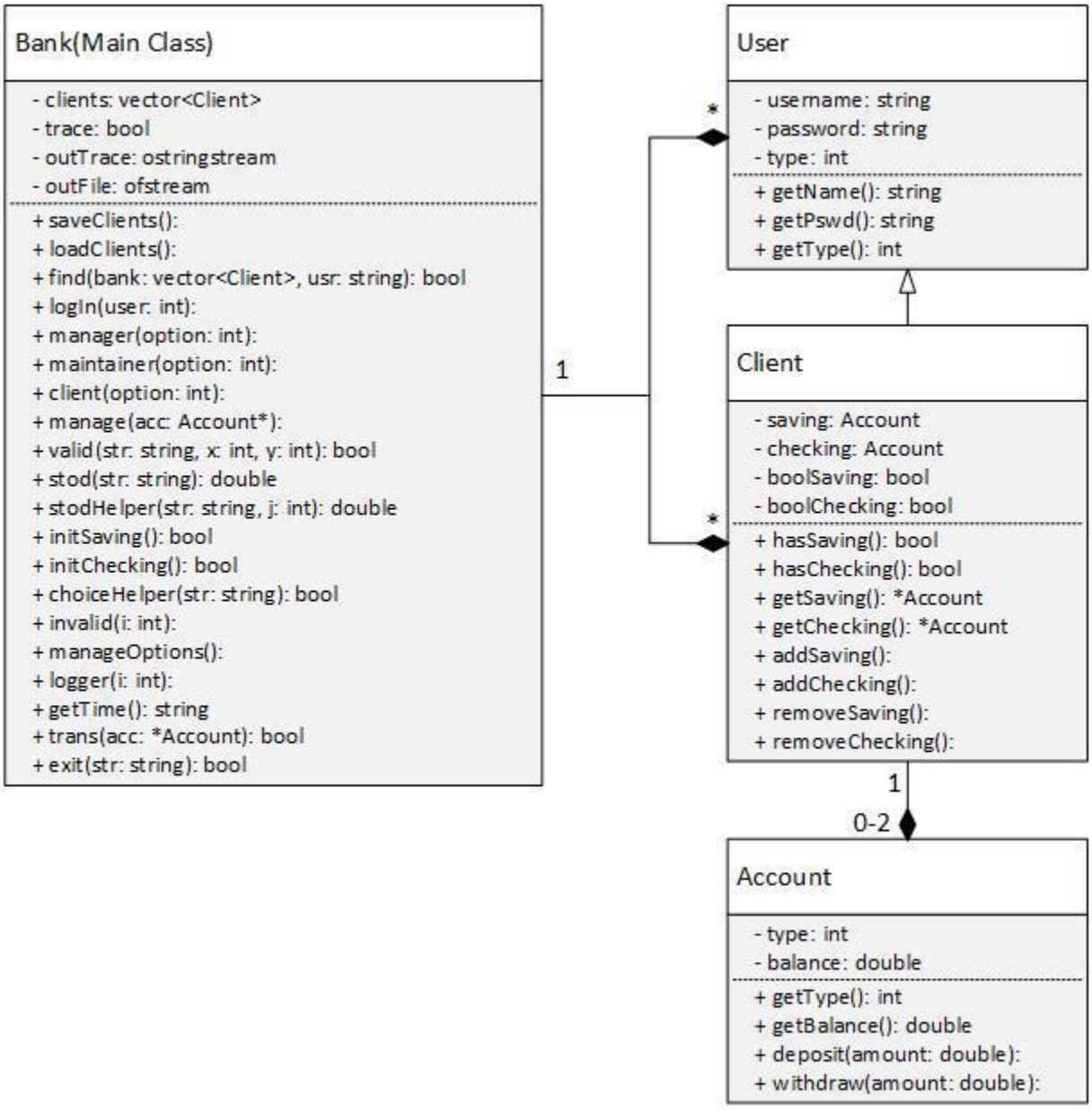
Banking System Project

Group 1:

Valmir Verbani, 250695712, vverbani@uwo.ca
Zaid Albirawi, 256626065, zalbiraw@uwo.ca

Due Date: October, 15, 2014

1. i. [D1 -- design]


Bank(Main Class)
- clients: vector<Client>
- trace: bool
- outTrace: ostringstream
- outFile: ofstream
+ saveClients():
+ loadClients():
+ find(bank: vector<Client>, usr: string): bool
+ logIn(user: int):
+ manager(option: int):
+ maintainer(option: int):
+ client(option: int):
+ manage(acc: Account*):
+ valid(str: string, x: int, y: int): bool
+ stod(str: string): double
+ stodHelper(str: string, j: int): double
+ initSaving(): bool
+ initChecking(): bool
+ choiceHelper(str: string): bool
+ invalid(i: int):
+ manageOptions():
+ logger(i: int):
+ getTime(): string
+ trans(acc: *Account): bool
+ exit(str: string): bool
1
*
User
- username: string
- password: string
- type: int
+ getName(): string
+ getPswd(): string
+ getType(): int
Client
- saving: Account
- checking: Account
- boolSaving: bool
- boolChecking: bool
+ hasSaving(): bool
+ hasChecking(): bool
+ getSaving(): *Account
+ getChecking(): *Account
+ addSaving():
+ addChecking():
+ removeSaving():
+ removeChecking():
*
1
0-2
Account
- type: int
- balance: double
+ getType(): int
+ getBalance(): double
+ deposit(amount: double):
+ withdraw(amount: double):

1. ii. [D2 -- class reference in code]

Class Name	Reference in Code
Bank(Main Class) (implementation)	**Filename:** “3307Assignment1.cpp” - **Line**: 0
User (interface)	**Filename:** “User.h” - **Line**: 13 - 28
Client (interface)	**Filename:** “Client.h” - **Line**: 14 - 82
Account (interface)	**Filename:** “Account.h” - **Line**: 16 - 52

1. iii. a. [D3 -- operations within classes]

Refer to the program diagram in section **1. i. D1 -- Design.**

1. iii. b. [D4 -- operations reference in code]

Class Name	Function	Reference in Code
User	**getName**	**Filename:** “User.h” - **Line**: 19
User	**getPswd**	**Filename:** “User.h” - **Line**: 20
User	**getType**	**Filename:** “User.h” - **Line**: 21
Account	**getType**	**Filename:** “Account.h” - **Line**: 25
Account	**getBalance**	**Filename:** “Account.h” - **Line**: 30
Account	**deposit**	**Filename:** “Account.h” - **Line**: 35
Account	**withdraw**	**Filename:** “Account.h” - **Line**: 41 - 46

1. iv. a. [D5 -- classes with important relationships]

User, Client, and Account.

1. iv. b. [D6 -- justification of importance]

The relationship between the User, Client, and Account is very important to the program because it forms the core of the program. Those classes work hand in hand to store and configure the bank’s user’s and clients’ information.

1. iv. c. [D7 -- relationship: what and why]

There is a couple of relationships between those three classes. Firstly, there is an inheritance relationship between the classes User, and Client. This relationship exists because the Users and Clients have the “username” and “password” fields in common. Every user needs login credentials but not every user needs bank accounts, and that is what separates them. The User class will only hold the user’s login information, while the client class will inherit information and top it off with the account information. This relationship satisfies the domain requirements by allowing each user to have their own login credentials. Furthermore, there exists a composition relationship between the Client and the Account classes. This is a composition relationship because if the Client object does not exists then the Account object will not either. This works for the benefit of the program because it allows the Client class to hold the bank account information for each specific client. This relationship is vital to the program’s requirements because it allows the client to perform numerous actions and gives them freedom on what they wish to do with their bank accounts.

2. i [D8 -- correspondence between problem statement and design elements]

Problem	OO Element Design
Open/Close Account	User
Deposit/Withdraw/Transfer/Obtain Balance	Account
Chequing/Saving	Client relationship with Account
Manager/Maintenance/Client	User
Save/Load Clients	Function in Bank
Logger	Object in Bank

2. ii [D9 -- assessment of design adequacy]

Problem	Assessment
Open/Close Account	Having a User class allows us to give the Manager role the freedom to open and close Client account as they wish, as long as these Client account have distinguishable usernames.
Deposit/Withdraw/Transfer/Obtain Balance	This does not directly reflect on the use of OO design, but having an account class will make the job easier when managing clients' accounts information.
Chequing/Saving	Having an Account class, allows us to offer the user a chequing account, a saving account, or both, and keep both of their information separate which decrease the level of complexity when having to deal with account information for each user.
Manager/Maintenance/Client	Having a User class allows us to specify the type of account and based on that we can handle that users restrictions and permissions. Also, decreases the complexity of the program.
Save/Load Clients	The main class(Bank) handles the list of clients, that contains all the client objects. It loads the list from a text file on the systems start up, and saves it when the system is exited.
Logger	The system is designed so if the logger switch is on, the system will keep a record of all the operation performed on the system.

2. iii [D10b -- justification of (near) perfect design]

Considering the problems faced with, the OO design is near perfect. This is because, first and foremost the design fulfills all the program's design specifications. The problem is divided into multiple part, where the User class handles the user information, the Client class handles the clients' information (Accounts), the Account class, handles the accounts information and performs all the required operations from within and, finally, the Bank class makes use of all of these classes together to create a system that fulfills all requirements.

3. i. [D11 -- Separation of Concerns]

The initial idea for the banking software system was to decompose it into parts that overlap in functionality as little as possible. In order to fully achieve this, it was essential to create different classes that are able to interact with each other. In our project, these different classes are ‘Account.h’, ‘Client.h’ and ‘User.h’. Each class has its own attributes and functions which reduces the dependency between modules through interfaces. With these interfaces, efficiency is increased by reducing coupling while at the same time increasing cohesion. Taking the Client and Account classes into consideration. The Account class does not, in anyway, depend on the Client class. The separation of concern is apparent in this relationship because each class can be fully functional without the other. Each of those classes handles different type of problems, the Account class handles all account related problems, while the client class handles all the client related problems. Furthermore, while the Client and Account classes take care of the user information type of problems, the Bank class focus more on the user interface side of the program. The Bank class handles all the requests in the program, calls the appropriate functions, contained in the Client and Account classes, and displays the answer to the user. Also, the Bank class handles all the error detections in the program through some of its methods, like invalid(), manageOptions(), etc.

ii. [D12 -- Strength of inter- vs intra-component relationships]

The intra-component relationships are stronger than the inter-component relationships in this program. For example, let us take the Client, and Account classes into consideration. These classes contain multiple methods that handle the Client accounts’ information and Account’s operations. The Client class contains methods such as, hasSavings, getSavings, addSavings, etc. which do not depend on any other components in the program. While the Account class contains methods like, deposit, withdraw, getBalance, which are also stand-alone methods that do not depend on other components or methods. On the other hand, the presence of inter-component relationships in the program is embodied in the inheritance relationship between the Client class and the User class, which share attributes such as username, password, and account type. To conclude, the reason behind the superiority of the intra-component relationships over the inter-component relationships in the program is because the classes that exists are specialized class that handles a specific problems.

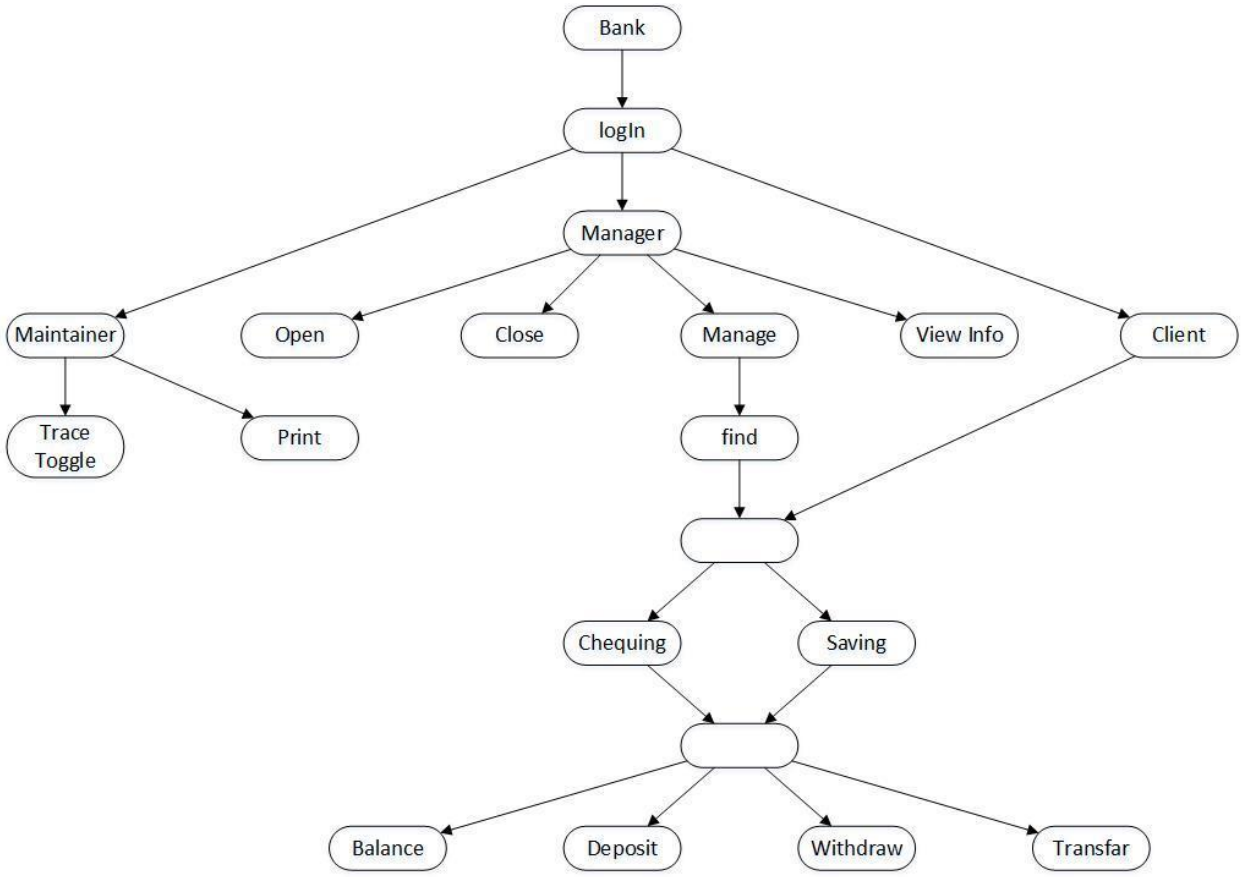
iii.[D13 - Principles of software design]

The relative balance between inter- and intra-component relationships in our design aligns to certain principles of software design. The reason why we had targeted the principles of software to design was to reduce fragility; to be able to make changes without have a drastic impact on the system itself, immobility; to be able to reuse in another application. The first design principle involved was the open/closed principle. We have classes, such as our ‘client’ class that allows us to create a client entity where it we are allowed its behavior to be modified without changing the source code in the process. Another design principle used is the single responsibility design, this is useful because each method and each class have its own role. This is key for changes where a behavior is modified because it is easy to track our newly developed errors.

iv.[D14 - Rationale for principles]

These particular design principles are set out this way for multiple reasons. One of the main reasons being consistency. Consistency not only makes the code easier to read and understand but it also makes it easier to make changes to specific sections without having big consequences. The second reason being functionality. Each class and each method in that class has a unique function. These classes and methods are ideal because when a certain object is needed, an object that meets that criteria is created. Similar with methods, each method has its own responsibility that made it reliable when called upon.

4. i. [D15 -- Functional decomposition]



ii.[D16- Functional vs. OO]

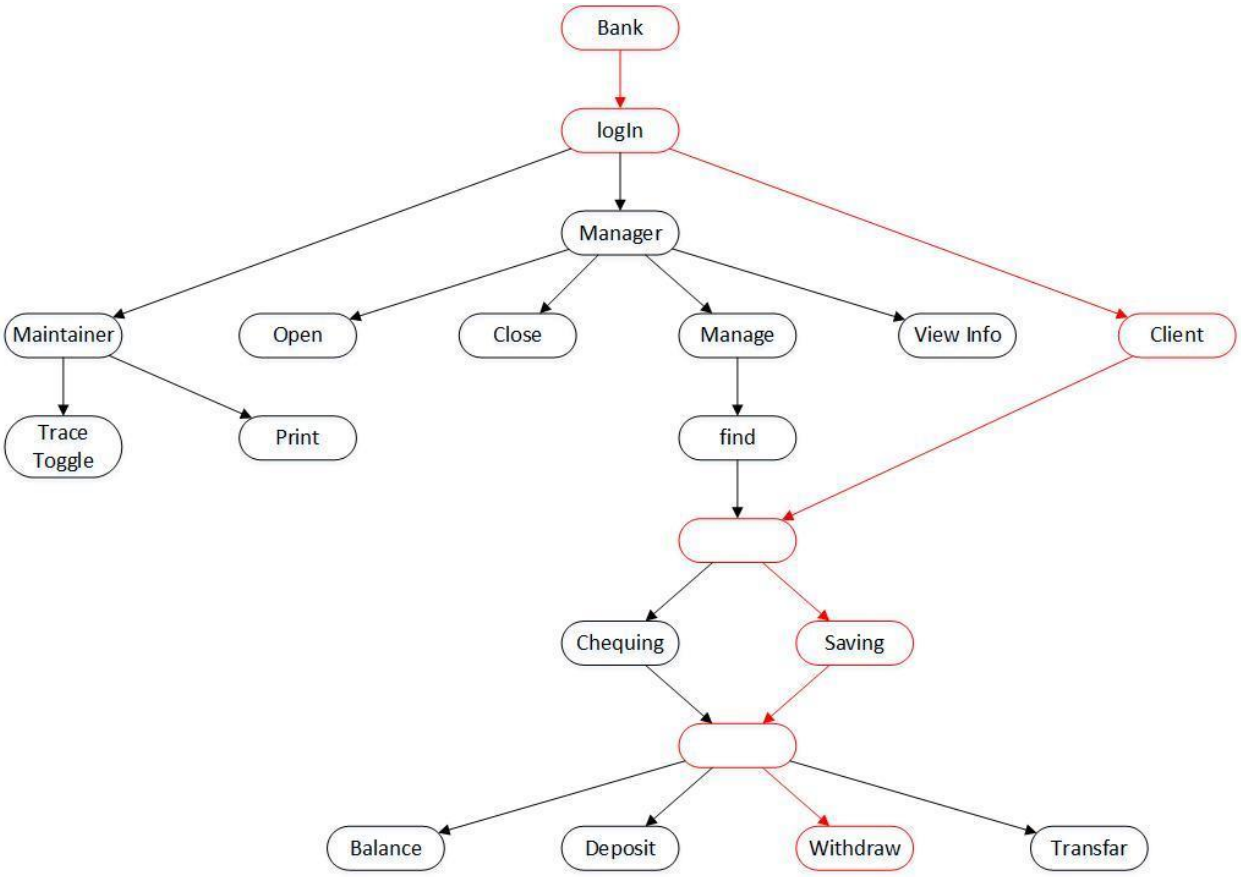
Both of these approaches are used for managing software complexity. In the functional decomposition the system is broken down into smaller component function which is later also broken down to even smaller components. OO on the other hand is an approach that deals with objects where these objects include certain data that have certain set of operations that alter the objects data. In the OO our program is organized through collections of objects. In our system, these objects include account, manager, client, and maintainer. Each of these objects have their own data and their own characteristics. In the account file we have an account object that recognizes whether a client, manager, or a maintenance person has opened the account. With these we have also created addition operations that only apply to the account that is created. For example we have created a function for it that returns the type of account, who has logged in. This object is redefined with the user who logs in, as a client you can deposit or withdraw. This allows us to have certain control over an account depending on who has logged into that account. An account has an object, client, where this client is allowed to view or change savings or chequing accounts and is able to transfer money from one account to another. In OO an entity client is created with certain options: view, and/or change account info. The OO allows us to save the entity's banking information with its data then later retrieve the same data,

5. i. [D17 -- Scenario--user and system aspects]

Money withdrawal:

- **System** >> greet user
- **System** >> ask user of the username
- **User** >> provide username
- **System** << fetch username
- **System** >> ask user of the password
- **User** >> provide password
- **System** << fetch password
- **System** <> find the User object that is associated with the username
- **System** <> validate password
- **System** >> display client options, which bank account the user would like to manage
- **User** >> enter option
- **System** << fetch choice
- **System** <> validate choice
- **System** >> display account management options
- **User** >> enter option(withdraw)
- **System** << fetch choice
- **System** <> validate choice
- **System** >> ask user for the amount they would like to withdraw
- **System** << fetch choice
- **System** <> validate choice
- **System** >> return the status of the process, and return to the previous menu

5. ii. [D18 -- Execution path on functional decomposition diagram]



5. iii. [D19 -- Execution trace from the program]

Username: zalbiraw
Password: password
Time logged: Wed Oct 15 23:53:48 2014

Managing the Saving account.
Withdrawn, $1000

5. v. [D20 -- Comparison of execution traces]

The execution trace turns out to be extremely similar to the generated trace from the functional decomposition diagram path. Firstly, the trace starts by printing out the login information, which matches the login state in the functional decomposition diagram. Secondly, the trace prints out the massage, “Managing the Saving account”, suggesting that it entered the managing mode for the savings account, which is the third node on the path in the diagram. Finally, the trace prints out that the withdrawal operation was performed, and that corresponds to the last state in the diagram path.

6. i. [D21 -- Categories of objects in the banking domain]

Categories:

- ATM
- Web App
- Mobile App
- Bank Computer Software

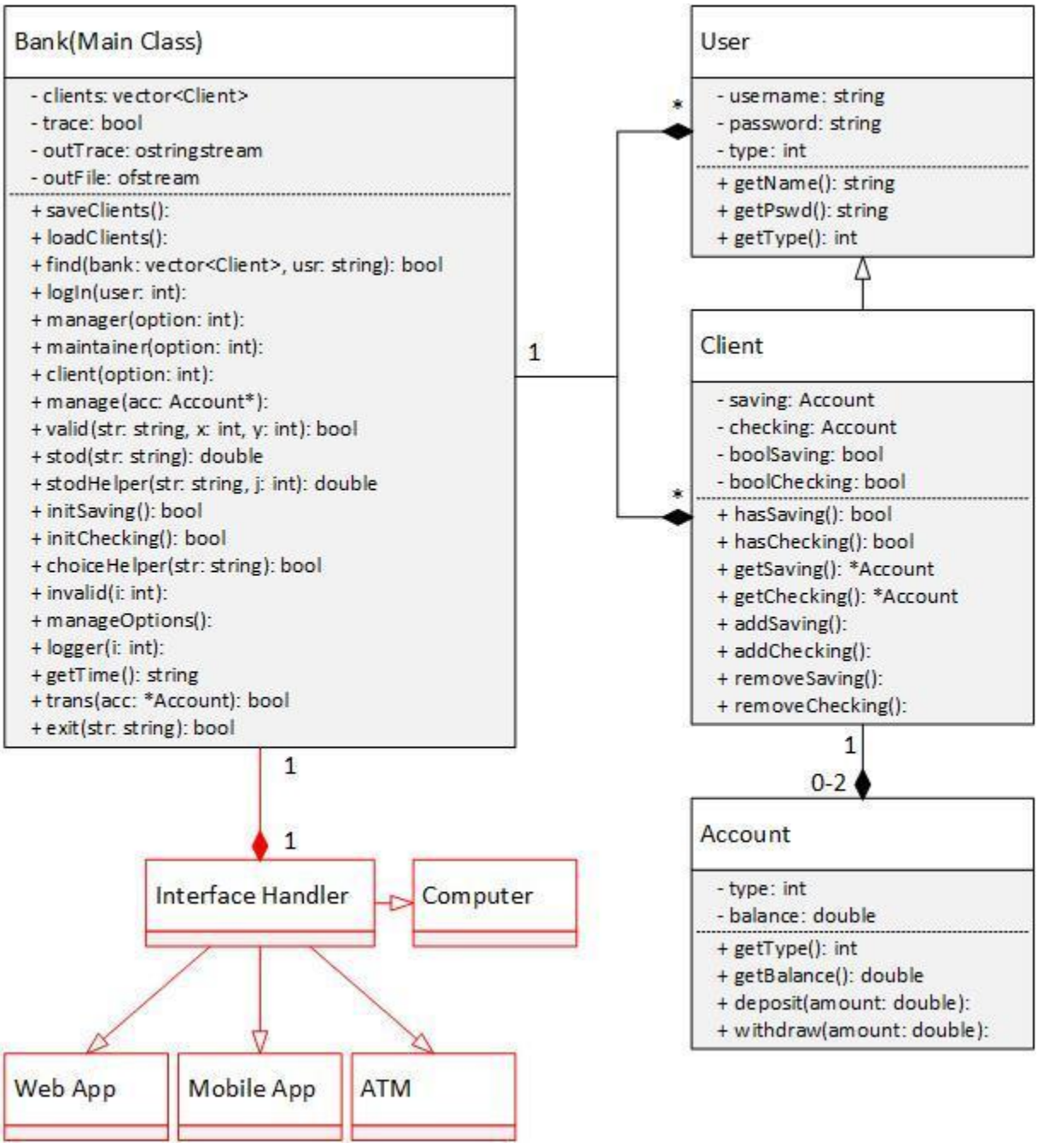
6. ii. [D22 -- Representation of object types]

Bank Computer Software is the only category that is directly represented in the diagram. That is because the Bank class acts a terminal interface for the program.

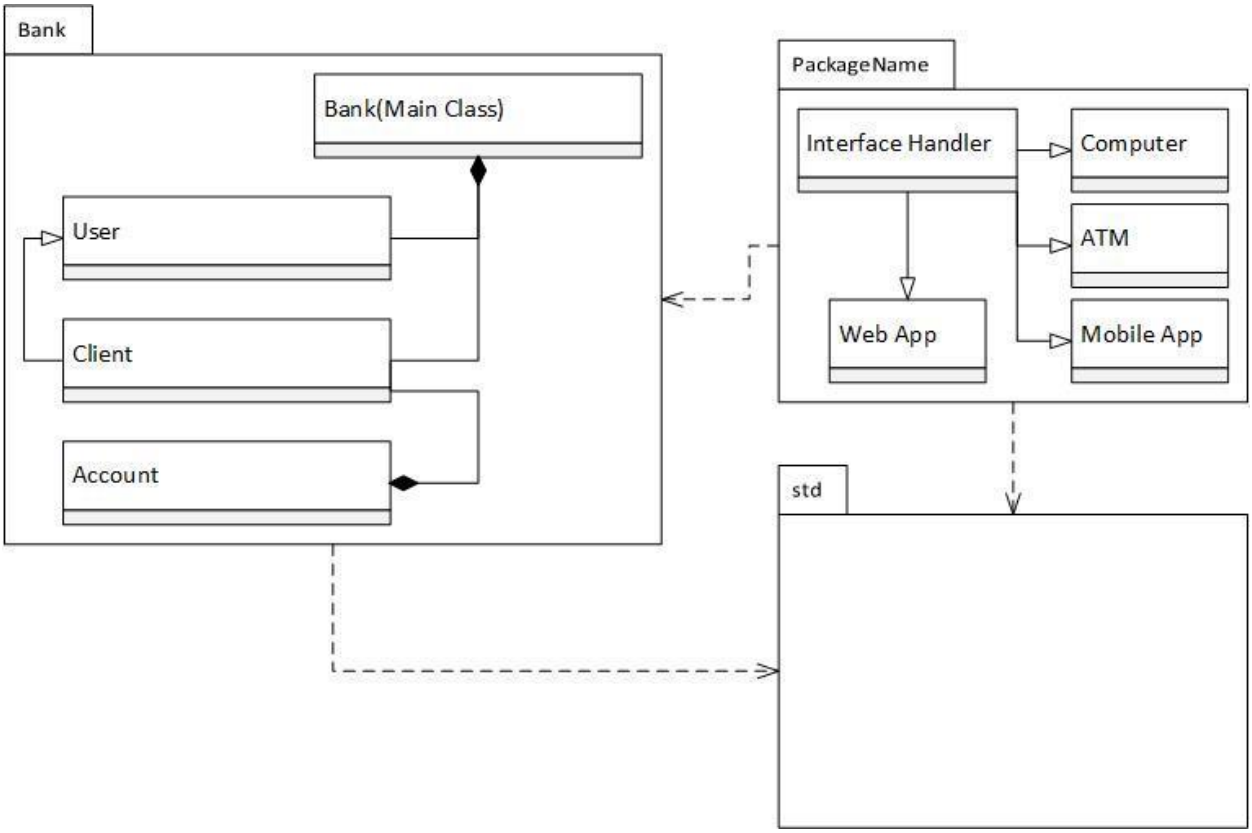
6. iii. a. [D23 -- System enhancement -- analysis]

New Object	Old Object
ATM Interface	Bank
Web App Interface	Bank
Mobile App Interface	Bank
Bank Computer Software Interface	Bank
Interface Handler	Bank

6. iii. b. [D24-System enhancement -- design]



7.i. [D25 - Package Diagram]



7.ii [D26 - Criteria for Packaging and Justification]

The criteria that was used for the packaging was the idea that the purpose for the package is for organizing model elements and diagrams into groups. This provides encapsulation of grouped elements that are semantically related. These elements can also be contained in other packages. These packages represent the different layers of our software system. The main criteria used was packaging elements by relation.

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

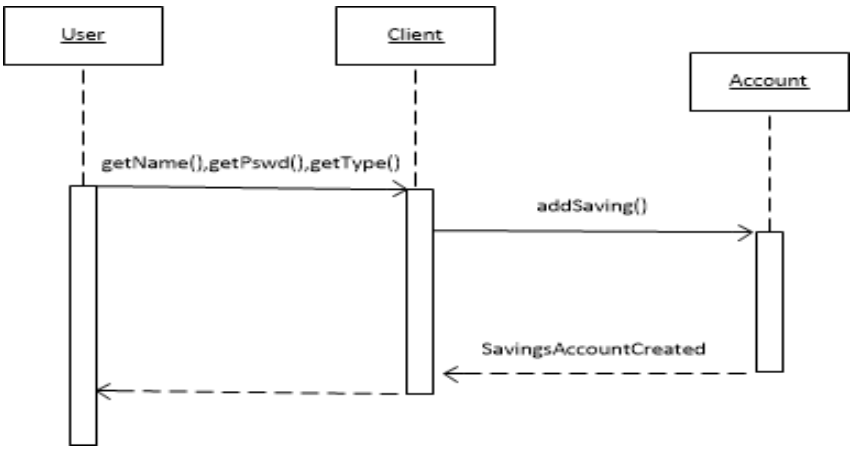
There is contrast between high-level design and low- level design packages for example, high-level design provides an overview of our entire system while the low-level design will look at each element of our design in detail. In the high-level design for our design, we cover the system architecture and database design. This describes the relation between the modules and the functions of the system, how it flows. While in the low-level design we would look into the elements in detail, in this case this would be for each class that we would look at their certain functions and the relations that are shared between each of these classes and these functions.

7.iv [D28- Scenario cutting across several packages]

Creating a user account with a savings account and then accessing its contents from main:

- The main class cuts into the username package
- This is where the main gets access to the user
- User receives a username, password, type
- Multiple types of users can be created
- After initializing the user is able to create an account through the client package
- Main class accesses the client package once user is created
- This is where the account package is accessed by client
- Now the user is able to access the contents of the client and account package
- While the main class has access of all these accounts through its own contents that have connections to the other classes

7.v [D29- Sequence Diagram]



8. [D30 - Lessons Learned]

The lessons learned from this quiz were:

- That an algorithmic decomposition was a diagram. Understanding an algorithmic decomposition of a system design sounds like a written ‘algorithmic form’ of relating the architect of design and its components.
- The second thing learned was the difference between high-level design and low-design.
- The difference between the inter-component vs. intra-component and how important they affect the program. The stronger the intra-component relationship gets, the higher separation of concern in the program.
- Always separate the program into smaller components because as the program grows the improvements that you’ll need to make to the program are will easily found when you specify a component for every problem, such as database, interface, login verifications, etc.