The University of Western Ontario, Faculty of Science
Department of Computer Science

Computer Science 3307, Object Oriented Design and Analysis
Quiz 1

Quiz: OO Concepts, Analysis and Models

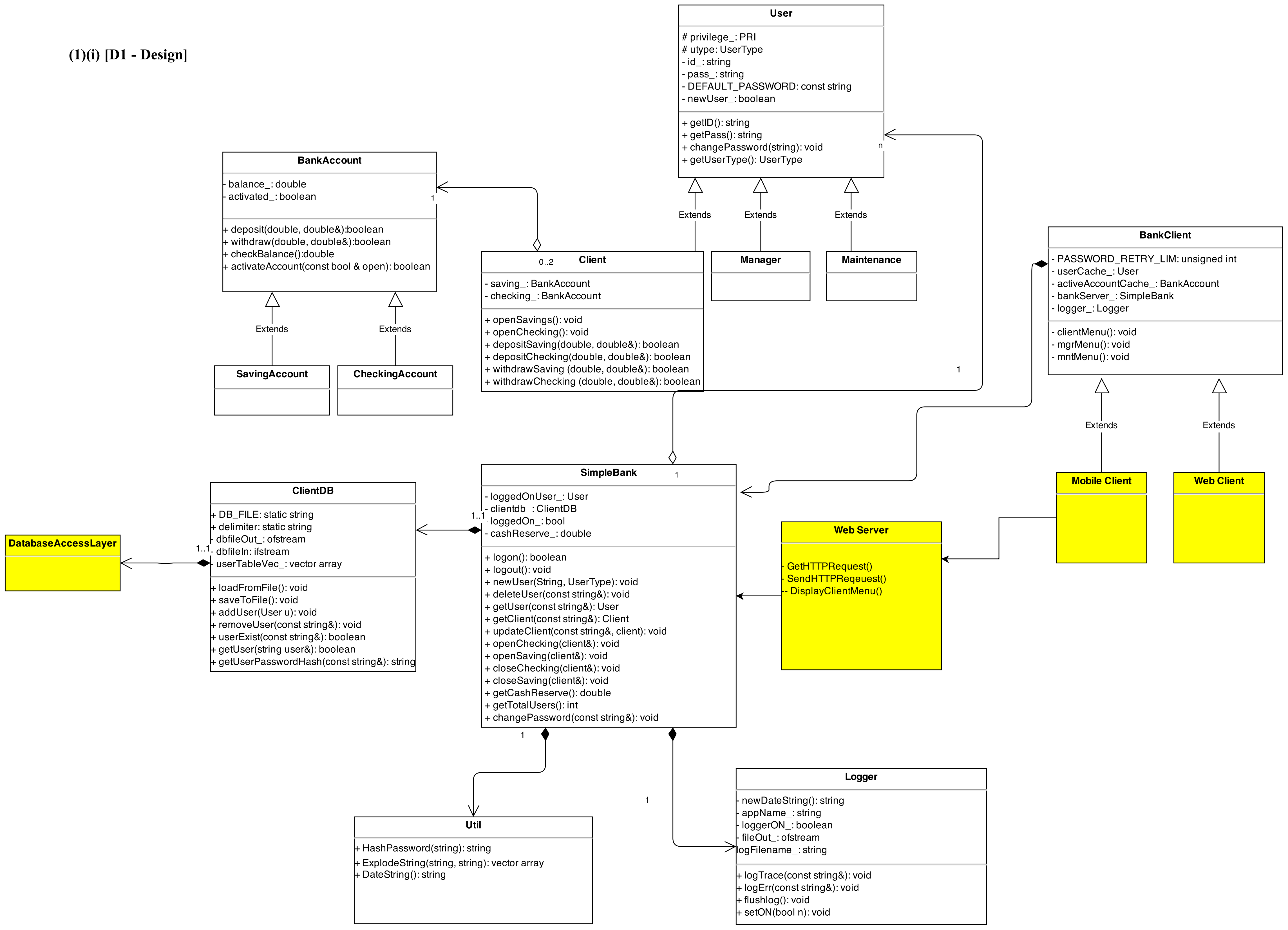
Group 15

Zhao Lin
zlin45@uwo.ca

Ziqi Rui
zrui@uwo.ca

October 15, 2014

(1)(i) [D1 - Design]


User
# privilege_: PRI
# utype: UserType
- id_: string
- pass_: string
- DEFAULT_PASSWORD: const string
- newUser_: boolean
+ getID(): string
+ getPass(): string
+ changePassword(string): void
+ getUserType(): UserType
n
Extends
Extends
Extends
BankAccount
- balance_: double
- activated_: boolean
+ deposit(double, double&):boolean
+ withdraw(double, double&):boolean
+ checkBalance():double
+ activateAccount(const bool & open): boolean
1
0..2
Client
- saving_: BankAccount
- checking_: BankAccount
+ openSavings(): void
+ openChecking(): void
+ depositSaving(double, double&): boolean
+ depositChecking(double, double&): boolean
+ withdrawSaving (double, double&): boolean
+ withdrawChecking (double, double&): boolean
Manager
Maintenance
BankClient
- PASSWORD_RETRY_LIM: unsigned int
- userCache_: User
- activeAccountCache_: BankAccount
- bankServer_: SimpleBank
- logger_: Logger
- clientMenu(): void
- mgrMenu(): void
- mntMenu(): void
Extends
Extends
SavingAccount
CheckingAccount
1
Extends
Extends
Mobile Client
Web Client
SimpleBank
1
- loggedOnUser_: User
- clientdb_: ClientDB
- loggedOn_: bool
- cashReserve_: double
+ logon(): boolean
+ logout(): void
+ newUser(String, UserType): void
+ deleteUser(const string&): void
+ getUser(const string&): User
+ getClient(const string&): Client
+ updateClient(const string&, client): void
+ openChecking(client&): void
+ openSaving(client&): void
+ closeChecking(client&): void
+ closeSaving(client&): void
+ getCashReserve(): double
+ getTotalUsers(): int
+ changePassword(const string&): void
ClientDB
+ DB_FILE: static string
+ delimiter: static string
- dbfileOut_: ofstream
- dbfileIn: ifstream
- userTableVec_: vector array
+ loadFromFile(): void
+ saveToFile(): void
+ addUser(User u): void
+ removeUser(const string&): void
+ userExist(const string&): boolean
+ getUser(string user&): boolean
+ getUserPasswordHash(const string&): string
1..1
1..1
DatabaseAccessLayer
Web Server
- GetHTTPRequest()
- SendHTTPReqeuest()
-- DisplayClientMenu()
1
1
Util
+ HashPassword(string): string
+ ExplodeString(string, string): vector array
+ DateString(): string
Logger
- newDateString(): string
- appName_: string
- loggerON_: boolean
- fileOut_: ofstream
- logFilename_: string
+ logTrace(const string&): void
+ logErr(const string&): void
+ flushlog(): void
+ setON(bool n): void

1 Program Design

(1) (ii) **[D2 – Class reference in code]**

Class Name	**Interface Location**	**Implementation Location**
BankAccount	BankAccount.h line 12	BankAccount.cpp 15
SavingAccount	BankAccount.h line 33	BankAccount.cpp line 72
CheckingAccount	BankAccount.h line 42	BankAccount.cpp line 82
User	User.h line 18	User.cpp line 14
Client	User.h line 63	User.cpp line 26
Manager	User.h line 101	User.cpp line 168
Maintenance	User.h line 109	User.cpp line 154
ClientDB	ClientDB.h	ClientDB.cpp
SimpleBank	SimpleBank.h	SimpleBank.cpp
Logger	Logger.h	Logger.cpp
Utils	Utils.h	Utils.cpp
main	-	main.cpp

(1) (iii a) **[D3 – operations within classes]**

The following is a list of methods for the above Classes

BankAccount
- BankAccount
- deposit
- withdraw
- checkBalance
- activateAccount
- isAccountActivated

SavingAccount
- SavingAccount

CheckingAccount
- CheckingAccount

User
- User
- init
- getID
- getPass
- changePassword
- getUserType
- isNewUser

Client

- Client
- openSavings
- openChecking
- closeSavings
- closeChecking
- isSavingOpened
- isCheckingOpened
- isAccountActivated
- getSavingBalance
- getCheckingBalance
- withdrawSavings
- withdrawCheckings
- depositSavings
- depositCheckings

Manager

- Manager

Maintenance

- Maintenance

ClientDB

- ClientDB
- addUser
- updateUser
- updateClient
- removeUser
- userExists
- getUserPasswordHash
- formatedExport
- saveToFile
- loadFromFile
- makeUser
- find
- getTotalUsers

SimpleBank

- SimpleBank
- logon
- logout
- newUser
- deleteUser
- userExists
- updateClient
- getUser

- openSavings
- closeSavings
- openChecking
- closeChecking
- exportUser
- save()
- getCashReserve
- changePassword

Logger

- Logger
- logTrace
- logError
- flushLog
- setON
- setAppName
- isON

Utils

- HashPassword
- ExplodeString
- DateString

(1) (iii b) **[D4 – operation reference in code]**

Class	**Method**	**Location**
BankAccount	BankAccount	BankAccount.cpp line 15
	deposit	BankAccount.cpp line 46
	withdraw	BankAccount.cpp line 32
	checkBalance	BankAccount.cpp line 61
	activateAccount	BankAccount.cpp line 66
	isAccountActivated	BankAccount.h line 26
Utils	HashPassword	Utils.cpp line 9
	ExplodeString	Utils.cpp line 17
	DateSTring	Utils.cpp line 36

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

The set of classes that have an important relationship is:

{User, Client, Manager, Maintenance, BankAccount, SavingsAccount, CheckingAccount, SimpleBank}

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

The above set of classes have a great importance because it represents a layer of abstraction, namely the banking server layer. In this set we define the relative primitives of the server domain, that is the types of users, types of bank accounts, and the public interface the bank server allows for client modules to access. We didn't include sets such as {BankClient} because {BankClient} represents a type of client code-- a console client-- that can interact with the server. The server and the console client code is designed to be loosely coupled, that is, we can easily implement a different kind of client, such as the web server/web client, in replacement of the console client without modifications to the server. This design choice has positive implications with stakeholders such as software engineers and customers. For example, new customers may want to have more convenience accessing the bank with mobile and web applications. The software designer and engineers can easily implement new clients to interact with the same server.

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

There is an "is-a" relationship among BankAccount, SavingsAccount, and CheckingAccount. The relationship is chosen because both SavingsAccount and CheckingAccount *is a* BankAccount, and thus we can inherit common properties of BankAccount to its subclasses.

The second relationship among the above classes is the "part-of" relationship, that is, SimpleBank aggregates all other classes in the set, the types of banks and users. This relationship represents the server module, it encapsulates unnecessary deals away from others. For example, a customer logged in from the console client is only concerned that he/she can deposit or withdraw funds, and not what the internal representation of the types of users or bank accounts there are.

2 Adequacy of Design

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

The following table approximately maps the major problem statements in assignment 1 to our OO-design classes.

ASN 1 Problem Statement	**OO Element in Design**	**Design Description**
The bank has an unspecified number of customers.	Client, ClientDB	ClientDB stores the collection of an unspecified number of Client objects; it also supports persistent data storage via files I/O
Manager can open/close client accounts; see bank details;	Manager, SimpleBank	SimpleBank provides appropriate function calls only to those user type of Manager
- Open/Close an account; close restricted to zero balance accounts - Deposit and withdraw money from an account	SimpleBank, SavingAccount, CheckingAccount	BankAccount types provide an interface for open/close accounts or deposit/withdraw
Transfer a sum from one account to another	BankClient	The Client application talks to the bank server to transfer funds
Obtain account balances	SimpleBank	SimpleBank has an interface to obtain client & account information
Maintenance personnel turning on/off trace logging	Logger, Maintenance	Logger has an interface to append string logs to file
Warn on checking withdraw less than $1000, levied $2 if continued.	BankClient	The Client application does the withdraw and $2 fine

(2)(ii) **[D9 – Assessment of Design Adequacy]**

A few specifications in assignment 1 were not directly mapped with our design, or not adequately designed. Primarily, we have not paid enough scrutiny to the security of our application as a whole. The areas that would need improvements are: user authentication, user authorization, data security, and better separation of concern among client and server modules.

a) The problem at hand was manageably small, thus we had chosen to implement secure user login with functional decomposition (e.g. hashing user passwords on the fly from user inputs) and user authorization with if-else statements (i.e. only show the client menus if a client is logged in). The password hash is stored as a string in the User object. This does not provide flexibility in working with the hash or add security features. Furthermore, if the amount of user types were to increase in the system, our way of checking for authorization with if-else statements would become increasingly unmanageable and be more prone to errors and security faults. Although our current method works for the small set of types of users to some degree, it would not scale or provide adequate security for a larger, or realistic banking system.

b) Our persistent data storage is implemented by using files storing data in a csv (comma-separated values) format; this is extremely unsecure. In the event of a system compromise, an attacker can modify the plain text values to increase account balances. A form of data storage with better authentication would be required.

c) The console client (BankClient) that we've developed is tightly coupled with the server module (SimpleBank). Currently, to achieve the specifications of assignment 1, both the client and server module is strictly required; and to change either one of the modules would require additional design and development efforts. We say this because we've noticed that the client has implementations of core banking features when it should have been an abstraction provided by the server. As an example: our implementation of fund transfers does not separate concerns of client module to that of a server. The client is implementing fund transfer by calling withdrawing and depositing API calls from the server in its own code, rather than calling a single *transfer* call to the server module. The obvious problem with this is that, if we need to implement another client, such as a web client, we would have to again implement the transfer feature by depositing and withdrawing, rather than having a transfer call provided by the server. This has implications in security where a client application can be comprised to make illegal fund transfers.

(2)(iii) **[D10aA – Improved Design]**

With respect to the above limitations in design, the following would solve or improve the inadequacies:

a) i) To improve better management and security of our user password hashes, we would create a new class **PasswordHash** that wraps around the plain string object. If effect, we can add additional methods such as check parity, serialize (for cross domain transfers), or rehash (for added security), among others.

ii) To address the issue of checking for user authorization, instead of using if-else statements to check the user type, we would improve it by implementing bit masking checking. For example, each user types would have different hex bits representing increasing priorities, e.g. 0x0001 for Customers and 0x1000 for Managers. Then, specific features would grant access by bit masking its unique number with the users'. This way, we eliminate if-else statements and have a single Authorization module.

b) To improve persistent data storage, we would change the design to use a commercially reliable relational database, such as Microsoft's SQL Server, or Oracle Database. Using this change, the interface of ClientDB would have to change as well, eliminating methods such as "loadFromFile" or "addUser" and add database related calls such as "OpenConnection" or "QueryAddUser".

c) The last issue mentioned from 2(ii) is the separation of concern of client and server modules, in particular the transfer method. As a design change, we would add a method with the signature:

```
void SimpleBank::transfer(BankAccount a, BankAccount b)
```

to the SimpleBank (server) Class, rather than implementing transfer with withdraw and deposit in the client code.

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

In addendum to the table provided in 2(i), these are the improved changes to design corresponding to the problem description.

Problem statement	**Addition of Classes**
Each user of the system has an "id" for secure login	PasswordHash
Manager has managerial powers; client can only see client menus	Authorization
Persistent data storage	ClientDB //interface change to use a database
Transfer funds from one account to another	SimpleBank //interface change to add transfer method

3 OO Design Principles

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

We have designed three basic layers of abstraction in conformation with “separation of concerns”. The first is the user data layer, the second is the bank layer, and the last is a client layer. All three layers were designed with high cohesion in mind, i.e. their respective modules perform exactly what they are intended to, and nothing more. For example, our data layer is implemented in the class ClientDB; its role is to only read and store user data sets to and from file to satisfy data persistence. The bank layer, represented by the class SimpleBank class does not care about how user data is stored; it simply calls the interface methods from the data layer. The bank layer is only concerned about its responsibilities such as managing users, managing bank accounts, and providing API calls to the next layer, the client. The client module was implemented as a console application; in effect, its implementations deal with only reading user input and displaying console text menus. The client’s interaction with the lower bank layer is strictly through the bank’s API, and nothing to do with bank specific implementations. By having these layers of abstraction in the banking system, the modules exhibit separation of concerns.

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

Our banking system exhibits strengths of OO design in both intra and inter-component relationships.

Intra-component relationships:

Our user and account representation within the bank layer manifests an “is-a” hierarchy relationship. That is, instances of a SavingAccount or a CheckingAccount are also an instance of BankAccount (by single inheritance), and similarly, instances of a Client, a Manager, or a Maintenance user are also an instance of the base class User. Strengths of this design include dynamic typing, or the ability to code to an interface. By coding to an interface, we mean the use of polymorphism in our objects. As an example, inside our client module BankClient, the currently logged in user is cached as an instance variable of type User. While the logged in user can be of any type (Client, Manager, or Maintenance), the BankClient is only concerned that it’s a User; and that it only acts upon the User interface, such as getting the user ID, or changing the user password – which is common to all users.

Inter-component relationships:

The aforementioned layers of abstraction, in particular, the bank layer and the user data layer, displays an aggregation, or "part-of" relationship among the modules. The SimpleBank implementation encapsulates an instance of ClientDB as a property. The ClientDB provides an interface to read and save client data. This way, the bank can pass processed actions, such as withdraw or deposit, into the ClientDB and preserve that change. Strengths of this relationship include encapsulation. The calls to saving or opening client data is private to SimpleBank, i.e. clients communicating with the bank has no information on how user data is stored or retrieved. As an added benefit, this design improves abstraction for the bank and hides its complexity away from client modules. As mentioned in (2), one of the inadequacies was the lack of security in our data storage implementation. Given opportunity, and a relational database is incorporated with the banking layer, the client modules will need little to no change implementation because the interface of the bank will remain consistent. This benefits code management and potentially cost in redesign of more modules.

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

The two inter and intra-component relationships (hierarchy and aggregation) listed above in 3(iii) align with design principles such as abstraction and encapsulation. In addition to the listed design relationships, we've also paid attention to concepts such as separating the implementation from the interface. In C++, this is done by strictly separating the interface, or declarations, inside .h files, while the implementation, or definitions, inside .cpp files. Again, the reason for this choice is to provide clients of a particular code only the interface, or protocol, that module will guarantee to implement, and not worry about how it's done exactly.

There is one imbalance within the system that we feel did not align with the principle of modularity. In particular, the client module (BankClient) is implemented in a single ~800 lines .cpp file. All user tasks, input mechanisms, and bank interactions are clumped together with no consideration in modularity. If we were to continue to maintain this console client code by adding more and more methods to satisfy requirements, the code would become more and more unmanageable.

(3)(iv) **[D14 – Rational for principles]**

The reason for the design choices we've made is to provide our project with greater manageability and reduce complexity when translating the problem domain into working OO code. Without using these design rules and patterns of OO-design, some inherent limitations would arise:

Dynamic Typing:

Our saving's account and checking's account is implemented as a single inheritance hierarchy to the base Class bank account. Without defining this "is-a" relationship among the bank accounts, both accounts would be defined as a concrete C++ class. This separation among the bank accounts would require implementation of common methods (e.g. withdraw, deposit) in both Classes. However, as the program evolves, if the core implementation of a method needs to be modified, both classes would need to be changed concurrently. This brings unwanted, but necessary, scrutiny to code management among the bank account classes, increasing risk to errors.

Aside from repeating code when modifying implementations, defining this bank account relationship also allows further specialization of bank accounts, for example, an addition of a TFSA (tax-free savings account), we just need to worry about its differences from a savings account and just implement the differences.

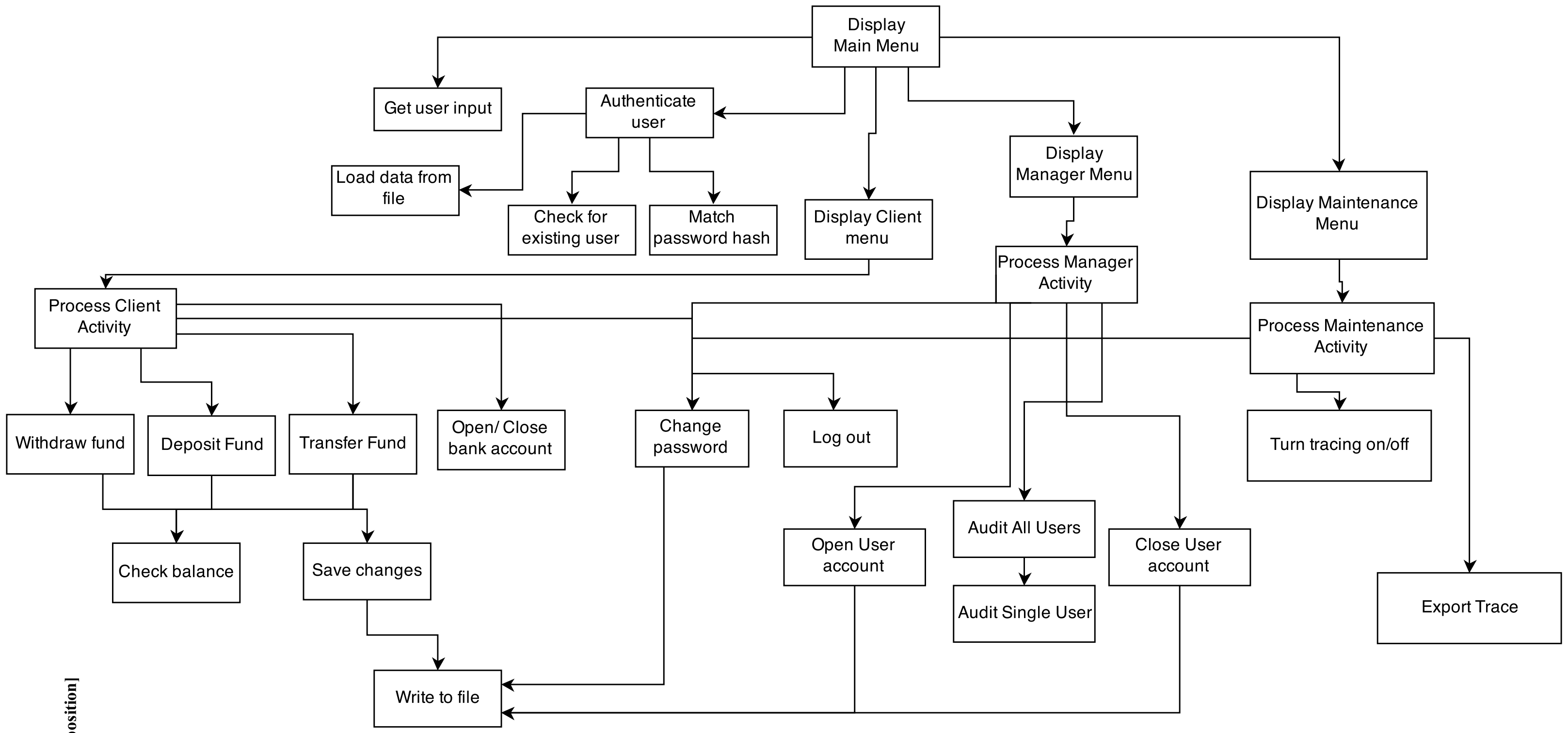
Abstraction

Although the problem domain from assignment one was relatively small, the implementation can get inherently complex. When working with a specific module, it is more effortless to only worry about its own problem domain rather than others as well. For example, the bank layer needs to save user data sets, it's effortless if there's a save method ready to use, rather than having to implement its own. We achieve this by implementing layers of abstraction discussed earlier. When each layer is only concerned with its responsibilities, the inherent complexity becomes smaller and more manageable.

Encapsulation

Classes can send messages to objects of other classes only through the specified public interface. The private fields of a class are encapsulated away from the client code. This enforcement is important in preserving the internal state of an object -- serving as a form of security. If we did not enforce this restriction, i.e. allow every object property to have a public access modifier, objects can be modified and rendered meaningless. For example, in our BankAccount class, the internal balance is private. The only legal way to modify the balance is through the public withdraw and deposit methods. By following through with the public methods, the internal state remains legal and consistent. On the other hand, if the balance property was public, it can be modified directly to an amount without "depositing money".

4 Structured Design



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

(4)(ii) **[D16 – Functional vs. OOP]**

With a functional decomposition of the assignment, we see that it can also solve the original problem. In comparison with the OOP design, both designs can implement design patterns such as polymorphism. In OOP we can achieve this by defining a hierarchy, and during run time assign that base class to a valid child class (e.g. Bank account, savings account, and checking account). In functional decomposition, we can do this by having a main function and checking an argument for the type of bank account at run time, and then subsequently branch off to specific functions.

Data is highly coupled with its methods in OOP, while in functional it's the opposite. In OOP we have concrete objects that represents real objects of the problem domain, and with running operations on these objects, their states change appropriately. In functional, we don't keep states of an object, but rather compute outputs based on a set of arguments. This aspect of functional programming will cause no side effects that may arise in OOP.

Lastly, it seems OOP decomposition has greater flexibility in reducing the inherent complexity of the problem domain. We can implement hierarchies and aggregate data types together to develop layers of abstraction.

5 System Dynamics

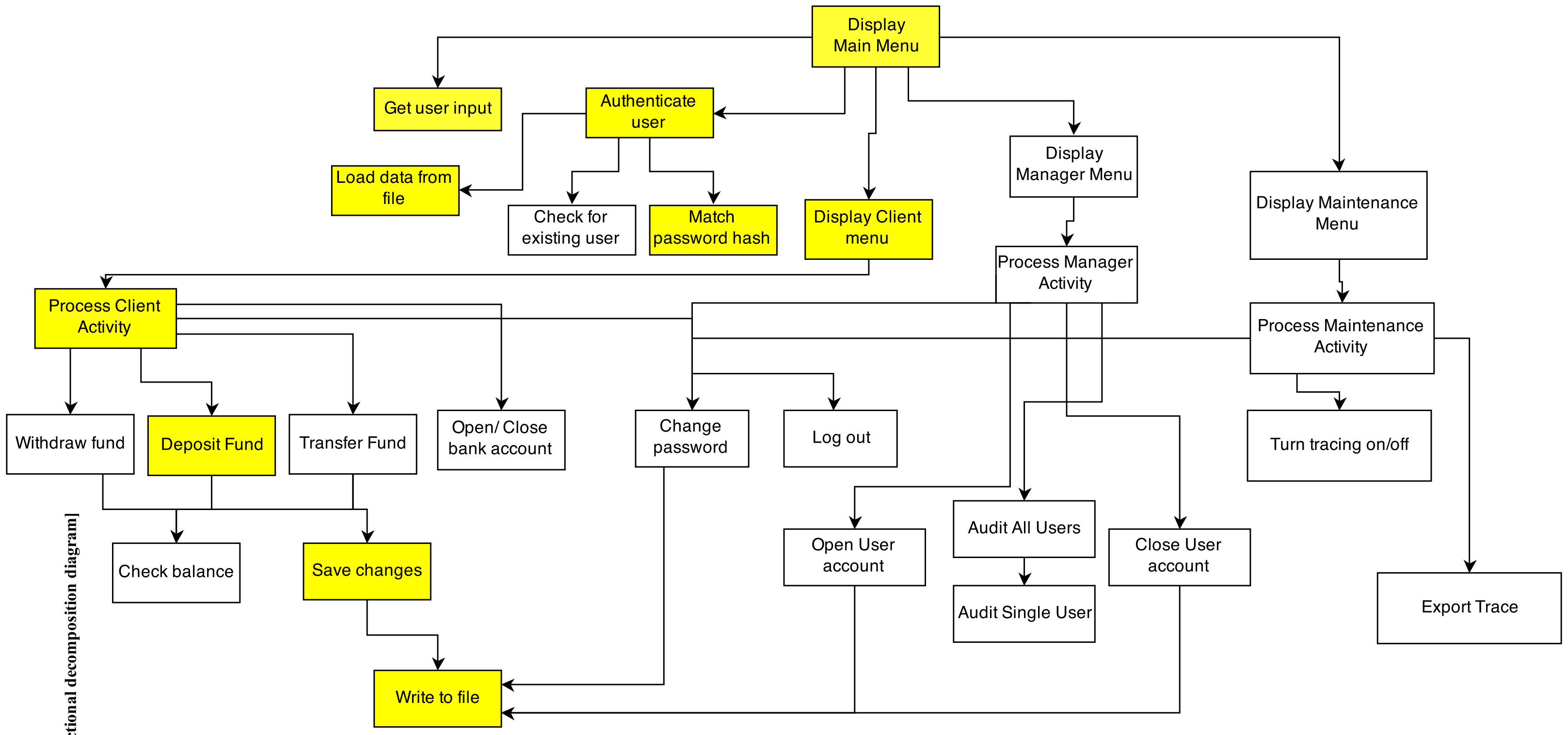
(5)(i) **[D17 – Scenario – user and system aspects]**

Scenario: A customer deposits $500 to his checking's account.

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

The following shows the approximate data flow among functions. Functions in [], and data in ().

[Display Main Menu]→[Authenticate User] (Username, password)→ [Match password hash]→[Display Client Menu](user id)→[Process Client Activity]→(user id, account type, total amout)→[Deposit Fund](user id, new blance)→[Save Changes]



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

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

```
Tue Oct 14 00:20:15 2014|SYSTEM|          Starting SimpleBank system v0.0.0
Tue Oct 14 00:20:17 2014|SYSTEM|          Attempting to login as user customer000
Tue Oct 14 00:20:39 2014|SYSTEM|          Successfully authenticated customer000
Tue Oct 14 00:20:43 2014|customer000|     Access Checking Account
Tue Oct 14 00:20:59 2014|customer000|     Open new Checking Account
Tue Oct 14 00:21:00 2014|customer000|     Successfully deposited $500 to Checking
Tue Oct 14 00:21:03 2014|SYSTEM|          Saving bank state to file
Tue Oct 14 00:21:13 2014|SYSTEM|          Logging out user: customer000
```

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

In comparison between the highlighted functional decomposition path and the actual program trace, both paths for the execution appear to be nearly identical. This is possibly due to the fact that the program of assignment 1 logs trace lines on the functional level, or at the start of each major interaction between the user and the system. The functional decomposition, in parallel with the tracing, decomposes the problem at each of the major functional levels, and in effect, having identical tracings with the program.

6 Domain and elements of OO design

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

- ATM customers
- Mobile customers
- Web portal customers
- Internal employees
- Central bank server
- Local bank managers
- Communication components
- Bank receptionists
- System IT specialists
- Web server components
- Database components
- Distributed & parallel systems
- Investment specialists
- Investment data mining
- Data visualization
- System security

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

The categories of objects that are directly mapped in our design are: ATM users, Local bank managers, system IT specialists (maintenance), database storage, and the central bank server.

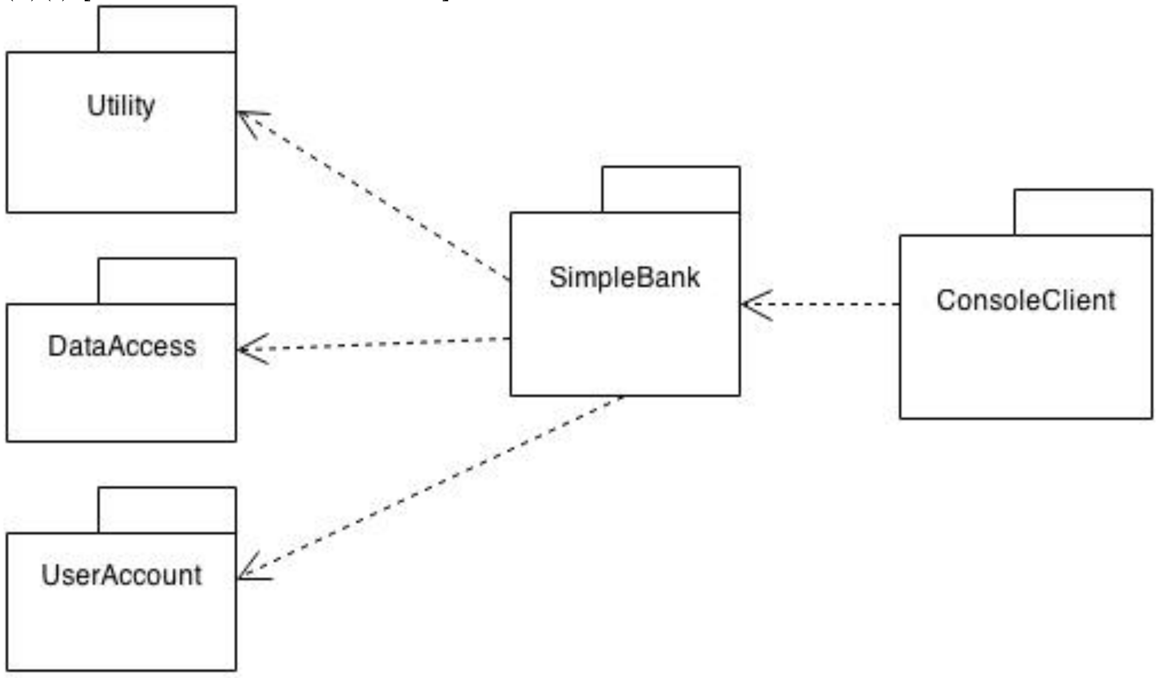
(6)(iii a) **[D23 – System enhancement --analysis]**

New Objects	**Old objects**	**Inter-relationships**
WebUser, MobileUser	User	inherits from User
WebServer	SimpleBank	Communicates requests to SimpleBank main server
DatabaseAccessLayer	ClientDB	DatabaseAccessLayer becomes “part-of” the ClientDB to user a relational database instead of file

(6)(iii b) **[D24 – System enhancement --design]**
The modified design is highlighted in yellow in (1)(i) UML Class diagram.

7 Models

(7)(i) **[D25 – Package diagram]**



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

The criteria used to create the package abstractions above are mainly separation of concerns and concerns for high cohesion. The three main layers we've mentioned (client, server, and data access) can be distinctively identified in a top down dependency view. The client communicates strictly with the server to modify the data layer, and never directly with the data. The server includes other packages such as Utils for miscellaneous functions unrelated to banking such as time keeping, and UserAccount for user permission, and bank account types.

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

A difference between the high-level design in the package diagram compared to the low-level design in the UML diagram is the level of abstraction each provides. The package diagram gives a big overview of the system as a whole and little detail to how each package is implemented, or represented. In contract, the UML diagram shows distinctively each Class involved in the problem domain, as well as available methods and data fields. Both designs offer a view of the system, and depending on the situation, one may be more appropriate than the other. For example, a program manager may want to see the overview of the system as a whole using the package diagram, while a software engineer may need to see the UML for implementation details.

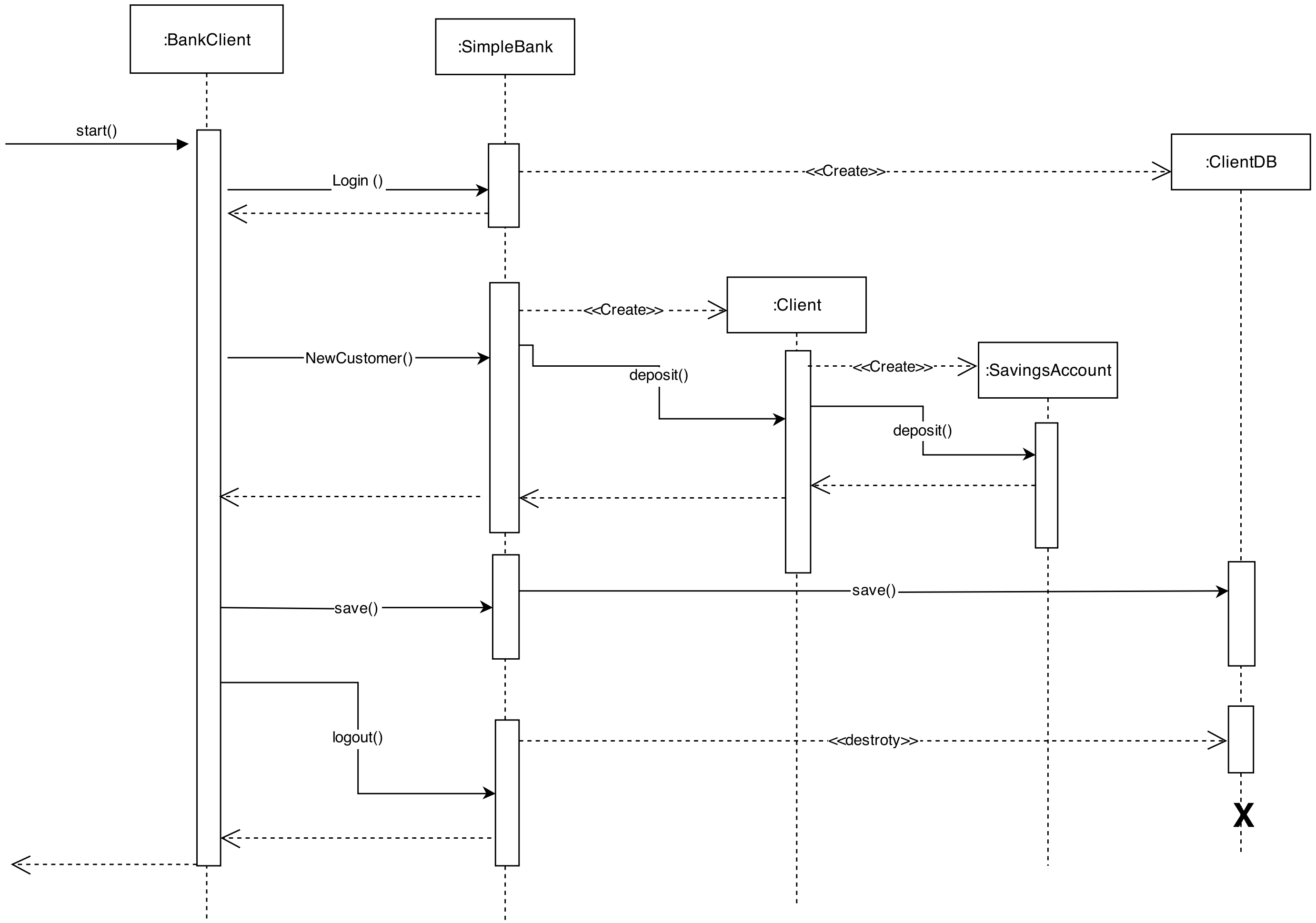
(7)(iv) **[D28 – Scenario cutting across several packages]**

The following scenario cuts across all the packages:

- A local manager logs in with the console client to the bank server [ConsoleClient]
- The manager creates a new customer on the server [SimpleBank][UserAccount]
- The manager deposits $100 credit to the customer's savings account as a signing bonus [UserAccount]
- The system saves the changes to file [Data Access]
- The system logs the above transactions into a trace with the current date and time [Utility]

(7)(v) **[D29 – Sequence Diagram]**

(7)(v) [D29 - Sequence Diagram]



8 Lesson Learnt

(8) **[D30 – Lessons learnt]**

There are three major lessons learned during this quiz:

1. A better understanding of OO concepts. We weren't aware of the many strengths and powers available to an object oriented programming language. There are many abstraction mechanisms that we've previously used, but not officially formalized, such as encapsulation, is-a vs part-of hierarchies, and separation of concerns.
2. It's hard to design a system that is optimal the first time. After understanding more OO concepts, we've noticed that our initial design of assignment one had many inadequacies. Designing an acceptable complex system is not a trivial task.
3. There are many ways to model the system such as UML class diagrams, package diagrams, and sequence diagrams.