Western University | Faculty of Science

Department of Computer Science

London, Ontario, Canada

CS3307: Object Oriented Analysis and Design

Nazim H. Madhavji

System Enhancement Project: SimpleBank

Group No. 8

Zhao Lin, [zlin45@uwo.ca](mailto:zlin45@uwo.ca)

Ziqi Rui, zrui@uwo.ca

Robert Goldfarb, rgoldfar@uwo.ca

Peter Antoniou, pantoni@uwo.ca

Sam Taghavi-Zadeh, [staghav@uwo.ca](mailto:staghav@uwo.ca)

December 3, 2014

D1 Enhanced design

Please see the attached UML diagram above.

D2 C++ source code

* All source code is located in SimpleBank/src
* To build on a Unix based system, type make in the same directory
* Run bank system with ./SimpleBank.exe
* Run vendor system with ./vendor.exe
* make clean to remove trace files and objects files
* make reset to reset database files

D3 Explanation of correspondence between code and design

Our bank system’s old data persistence interface could no longer accommodate the new vendor system, and so an additional one, DBDelegate, is added. The old interface implemented by ClientDB was limited in that it parsed a csv (comma-seprated filed) as plain text. Each row on the file directly related to a user along with its account balances. To incorporate additional information from the vendor system, that simple model does not work. For example, adding a transaction record of a purchase would be difficult to encode for each user in the same file and would have a lot of coding overhead. A relational database would be necessary.

DBDelegate aims to delegate the old ClientDB requests to a new database interface. Instead of using file parsing, we use the self-embedded C++ relational database library sqlite. We chose sqlite for its expressive power for simple sql queries such as create tables, select, and update. It is self-embedding in a sense that we don’t need to have a separate server database process, and is built into the application. DBDelegate creates three main tables: users, accounts, and transactions. users hold all use info such as user id, user password hash and some user options. accounts hold all the bank account information, such as account ids, balances, and owner ids. Finally, transactions hold every purchase a user approves at the vendor. DBDelegate provides an interface to both the bank system and the vendor system. In effect, our systems use a shared database, the bank can see vendor’s purchase log, and the vendor can see user’s credit information.

The new vendor system is implemented in the following set of classes: Vendor, VendorClient, and v\_main. The vendor system is a separate executable instance and is instantiated by v\_main. VendorClient, like BankClient, is a front-end client module that accesses the Vendor module. More specifically, in our case, VendorClient is a console client. We’ve again considered separation of concerns in this design. If in the future we were to implement a new vendor client in a different medium, such as a web client (e.g. web checkout), we can simply adapt the Vendor interface into that new client. This design aims for low coupling between the server and client code, while having high cohesion within each module itself.

Vendor has a very simple interface. It has a static vector containing a list of available inventory for purchase. Realistically, this would be modeled inside vendor’s own database, but for simplicity, it is a static vector of roughly 30 items. Then vendor has a reference to the DBDelegate interface. The interface provides simple methods to get and store information to the shared database, such as authenticating users, checking user credit validity, or recording purchases. And finally, VendorClient implements a simple method that randomly generates a “special” item for sale by randomly selecting an item from its inventory for a random price.

Inside the bank system, we’ve extended the BankAccount hierarchy and have added the class CreditAccount as a derived class. A credit account shares many properties with the BankAccount, such as balances, withdraw (make payments), and deposit (making purchases). Again, due to design issues from assignment 1, there would be a lot of code restructuring to integrate the credit and vendor system to the old system. As a workaround to satisfy the requirements, we’ve sought to user functional techniques instead of object oriented ones. For example, to update a user’s account balance, instead of calling User.CreditAcccount.Depsoit, we call the method:

void UpdateAccountBalanceDel(int uid, int atype, double newBalance);

inside DBDelegate. In a sense, we’re calling more functions and passing object properties instead of calling methods on objects.

Another relationship among the new vendor code and the old bank system code is the amount of code reuse in the vendor system. Many methods and types can be used again in the vendor, some include: console client code, checking user input, displaying menu, logging module, database module, user type module, and the utilities module. Knowing the reusability of so many components in the bank system, future enchantment would include better organization of these modules, and have separate public header interfaces for them.

The last aspect from our new design is the new CONFIG.h file. This header file contains predefined system constants and configurations. It serves two purposes: one, users can define common settings such as default usernames and passwords; and two, we’ve added constants across the system to eliminate magic numbers. This is very useful if we need to change commonly used constants in the future in just one place instead of at all the locations.

D4 Operational Evidence

Scenario 1: Purchase made at the vendor (valid credit card). Bank notified of transaction.

This scenario involves the following steps:

1. A customer with a valid credit card logs into the vendor
2. The customer approves the random item for purchase
3. A transaction record is saved in the database, and the bank is therefore notified

trace.txt

Sat Nov 29 23:43:02 2014|VENDOR| Authenticating user...

Sat Nov 29 23:43:04 2014|VENDOR| Attempting to login as joey

Sat Nov 29 23:43:04 2014|VENDOR| Login failed.

Sat Nov 29 23:43:04 2014|VENDOR| Login failed for user joeyAttempt: 0

Sat Nov 29 23:43:04 2014|VENDOR| Successfully logged in as joey

Sat Nov 29 23:43:07 2014|VENDOR| Attempting to login as joey

Sat Nov 29 23:43:07 2014|VENDOR| Successfully logged in as joey

Sat Nov 29 23:43:07 2014|VENDOR| Credit card approved for user

Sat Nov 29 23:43:08 2014|VENDOR| Starting console vendor client

Sat Nov 29 23:43:09 2014|VENDOR| Displaying special item

Sat Nov 29 23:43:09 2014|VENDOR| Purchase for CHICKEN approved: Notifying bank system database.

Sat Nov 29 23:43:10 2014|VENDOR| Logging out vendor

Terminal Console

**[Welcome to G8'S GROCER self checkout!]**

**YOUR EVERYDAY FOODS | CHECKOUT**

**Sat Nov 29 23:43:00 2014**

**<Please select an option to continue>**

**----------menu----------**

**1 Login**

**2 Quit**

**>**1

**User ID:** joey

**PIN:**password

**Incorrect username or password.**

**User ID:** joey

**PIN:**pass

**[Welcome joey]**

**Sat Nov 29 23:43:07 2014**

**You have a valid credit card from SimpleBank!**

**1 Buy today's special!**

**2 Logout**

**>**1

**[Special Item for sale!]**

**Item Name: CHICKEN**

**Price: $92**

**Continue with purchase? [y\n]**

**>**y

**[Welcome joey]**

**Sat Nov 29 23:43:09 2014**

**You have a valid credit card from SimpleBank!**

**1 Buy today's special!**

**2 Logout**

**>**2

Database: transactions table

tid customer\_id amount description date hidden

---------- ----------- ---------- ----------- ------------------------ ----------

…

3 3 92.000000 CHICKEN Sat Nov 29 23:43:09 2014 0

Here, transaction 3 has the cutomer\_id (the user) that made the above purchase. The bank is able to look at this table.

Scenario 2: Purchase made at the vendor (frozen credit card). Customer notified.

This scenario involves the following steps:

1. Customer with a frozen credit card makes attempts to login to the vendor
2. The customer’s credit card is rejected and is notified

Database: accounts table

aid owner\_id balance type activated

---------- ---------- ---------- ---------- ----------

1 3 0 1 1

2 3 196.0 2 0

The user’s credit account (type 2) is currently frozen (i.e. not activated in the db).

trace.txt

Sun Nov 30 01:26:54 2014|VENDOR| Starting console vendor client

Sun Nov 30 01:26:36 2014|VENDOR| Authenticating user...

Sun Nov 30 01:26:37 2014|VENDOR| Attempting to login as joey

Sun Nov 30 01:26:37 2014|VENDOR| Successfully logged in as joey

Sun Nov 30 01:26:37 2014|VENDOR| Your Credit Card is not valid or is frozen!

Please contact your bank for assistance.

Terminal Console

**[Welcome to G8'S GROCER self checkout!]**

**YOUR EVERYDAY FOODS | CHECKOUT**

**Sun Nov 30 01:26:34 2014**

**<Please select an option to continue>**

**----------menu----------**

**1 Login**

**2 Quit**

**>**1

**User ID:** joey

**PIN:**pass

**[Welcome joey]**

**Sun Nov 30 01:26:37 2014**

**Your Credit Card is not valid or is frozen!**

**Please contact your bank for assistance.**

Scenario 3: Customer views current month’s purchases at ATM

This scenario involves the following steps:

1. Customer logs in to the vendor system
2. Customer approves multiple purchases
3. Customer logs into the bank system
4. Customer views purchases made at the vendor

trace.txt

Sun Nov 30 01:35:24 2014|VENDOR| Authenticating user...

Sun Nov 30 01:35:26 2014|VENDOR| Attempting to login as joey

Sun Nov 30 01:35:26 2014|VENDOR| Successfully logged in as joey

Sun Nov 30 01:35:26 2014|VENDOR| Credit card approved for user

Sun Nov 30 01:35:46 2014|VENDOR| Starting console vendor client

Sun Nov 30 01:35:29 2014|VENDOR| Purchase for TURKEY approved: Notifying bank system database.

Sun Nov 30 01:35:30 2014|VENDOR| Purchase for TOMATO SAUCE approved: Notifying bank system database.

Sun Nov 30 01:35:32 2014|VENDOR| Purchase for BEEF approved: Notifying bank system database.

Sun Nov 30 01:35:33 2014|VENDOR| Purchase for PEANUTS approved: Notifying bank system database.

Sun Nov 30 01:35:35 2014|VENDOR| Purchase for CHICKEN BROTH approved: Notifying bank system database.

Sun Nov 30 01:35:37 2014|VENDOR| Purchase for SAUSAGE approved: Notifying bank system database.

Sun Nov 30 01:35:38 2014|VENDOR| Purchase for POPCORN approved: Notifying bank system database.

Sun Nov 30 01:35:39 2014|VENDOR| Purchase for CHILI approved: Notifying bank system database.

Sun Nov 30 01:35:41 2014|VENDOR| Purchase for CHICKEN approved: Notifying bank system database.

Sun Nov 30 01:35:43 2014|VENDOR| Purchase for RAISONS approved: Notifying bank system database.

Sun Nov 30 01:35:44 2014|VENDOR| Purchase for TEA approved: Notifying bank system database.

Sun Nov 30 01:35:50 2014|SB\_Console Client| Starting Console Client

Sun Nov 30 01:35:55 2014|SB\_Console Client| Attempting to login as joey

Sun Nov 30 01:35:55 2014|SB\_Console Client| Succesfully logged in as joey

Sun Nov 30 01:35:59 2014|SB\_Console Client| Viewing credit purchase history and report

Sun Nov 30 01:39:13 2014|SB\_Console Client| Logging out user: joey

Terminal Console

**Welcome joey!**

**Your role: Client**

**Sun Nov 30 01:35:55 2014**

**Delegate db id: 3**

**----------menu----------**

**1 Access Savings Account**

**2 Access Checking Account**

**3 Transfer funds to\from account**

**4 Change Password**

**5 View Credit Report**

**6 Logout**

**>**5

**[Monthly Credit Purchase History]**

**---------------------------------------------------------**

**Date Item Price**

**Sat Nov 29 23:37:17 2014 SAUSAGE $100.00**

**Sat Nov 29 23:41:45 2014 JUICE $4.00**

**Sat Nov 29 23:43:09 2014 CHICKEN $92.00**

**Sun Nov 30 01:35:29 2014 TURKEY $65.00**

**Sun Nov 30 01:35:30 2014 TOMATO SAUCE $86.00**

**Sun Nov 30 01:35:32 2014 BEEF $93.00**

**Sun Nov 30 01:35:33 2014 PEANUTS $7.00**

**Sun Nov 30 01:35:35 2014 CHICKEN BROTH $21.00**

**Sun Nov 30 01:35:37 2014 SAUSAGE $28.00**

**Sun Nov 30 01:35:38 2014 POPCORN $35.00**

**Sun Nov 30 01:35:39 2014 CHILI $49.00**

**Sun Nov 30 01:35:41 2014 CHICKEN $56.00**

**Sun Nov 30 01:35:43 2014 RAISONS $70.00**

**Sun Nov 30 01:35:44 2014 TEA $77.00**

**---------------------------------------------------------**

**Total: $783.00**

**[CREDIT SUMMARY]**

**---------------------------------------------------------**

**Checking Balance $1034.00**

**Unpaid Credit Balance $587.00**

**Credit to Checking ratio %56**

**Enter a key to continue...**d

Scenario 4: End-of-Month trigger by a manager

This scenario involves the following steps:

1. A customer (joey) has enough checking balance to pay his credit purchases
2. A customer (robbie) does not have enough checking balance to pay his credit purchases
3. Transaction history for those who successfully pay in full will be hidden
4. Both customers have the default setting of paying in full
5. A manager logs in to the bank system to trigger the end of month

trace.txt

Sun Nov 30 02:04:27 2014|SB\_Console Client| Starting Console Client

Sun Nov 30 02:04:30 2014|SB\_Console Client| Attempting to login as 3307

Sun Nov 30 02:04:30 2014|SB\_Console Client| Succesfully logged in as 3307

Sun Nov 30 02:04:33 2014|SB\_Console Client| [Processing End of Month Credit Payments...]

Sun Nov 30 02:04:42 2014|SB\_Console Client| Logging out user: 3307

Sun Nov 30 02:04:33 2014|BANK SERVER| Triggering end of month event credit processing

Sun Nov 30 02:04:33 2014|BANK SERVER| Processing user: joey, credit balance: 204.000000

Sun Nov 30 02:04:33 2014|BANK SERVER| Processing user: bobbie, credit balance: 139.000000

Sun Nov 30 02:04:38 2014|BANK SERVER| Failed to process user: bobbie due to insufficient checking funds

FailedCreditPayments.txt

Sun Nov 30 02:04:38 2014|SYS| User: 4 bobbie failed to pay credit balance of $139.000000

Sun Nov 30 02:04:38 2014|SYS| Freezing credit account for: bobbie

D5 Design inspection with analysis and findings

We have conducted analysis on group 9’s code. The following is our findings.

**Structural correspondence between Design and Code:**

Are all the classes and interrelationships programmed in the application explicitly represented in the class diagram of the system?

Yes No Partly (Can be improved)

 Comment on your analysis: The class diagram appropriately represents the implemented system.

Comment on your findings: The class diagram appropriately represents the implemented system.

**Functionality:**

Do all the programmed classes perform their intended operations as per the requirements?

Yes No Partly (Can be improved)

Comment on your analysis: All the functions tested (mentioned in the findings below) accurately perform in accordance with the requirements.

Comment on your findings: The following functions were tested and accurately performed their purpose:

Customer Functions

o Withdraw

 o Deposit

 o Transfer

Maintenance Person Functions

o Trace function (associated ‘Wraps.cpp’)

Manager Functions

o Totals

o Create user

o Close account

o Edit

o Account view

**Cohesion:**

Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion: the functionalities embedded in a class, accessed through its methods, have much in common, e.g., access common data)

Yes No Partly (Can be increased)

Comment on your analysis:

The function of the customer class is the manipulation of account values. The functions of the customer class could be more properly implemented based on this purpose using just one ‘MANIPULATION’ method. There is no need for separate values for withdrawals, deposits, and transfers – these are all ultimately doing the same thing! One manipulation function is all that is needed (dealing with negative numbers for withdrawal, positive numbers for deposit, and a combination for both.

In the manager class, gathering values from the accounts is done completely separately in the total function and account view functions. This could be made more cohesive by gathering data in the same way (i.e. with the same function). The total function could just do the identical method that the account function does but repeated and totalled.

Comment on your findings:

• Customer Functions

o Withdraw – deals with a unique ‘withdrawAmount’ variable

o Deposit – unique ‘depositAmount’ variable

 o Transfer – unique ‘transferAmount’ variable

• Maintenance Person Functions

o Trace function (associated ‘Wraps.cpp’)

• Manager Functions

o Totals – Gathering values completely independent from gathering value in account view

o Create user

o Close account

o Edit o Account view

**Coupling**:

Do the programmed classes have excessive inter-dependency? (High Coupling: In this case a class shares a common variable with another, or relies on, or controls the execution of, another class.)

Yes No Partly (Can be reduced)

Comment on your analysis: There is very low coupling HOWEVER this is made a significantly easy accomplishment by the very low separation of concerns. The manager class is coupled more than it needs to be as the variables and functions are all named/designed to refer directly to accounts and users. This manager function could be much more reusable if it was implemented as a ‘controlling class’ that would work on any other [attribute bearing] class (such as a user or an account).

Comment on your findings: There are no shared variables or inter-control.

**Separation of concerns:**

Is the scoped problem decomposed into separate concerns where each concern is encapsulated in a construct such as a class with well-defined interface and cohesive functions with minimal of connections with other concerns?

Yes No Partly (Can be improved)

Comment on your analysis: Because a banking system can have 1) different types of accounts (savings/checking...) and 2) different types of users (consumer/small business/large business...) it is important to separate each of these concerns into separate, loosely coupled classes.

Additionally, the accounts (checking and savings) are not only un-capsulated, but are also only represented by a single value. These accounts should clearly be represented by a separate class to allow for proper scalability both in terms of adding more attributes to the accounts and in terms of eventually adding other types of accounts.

Comment on your findings: The accounts and users are defined together in one class! This is not a proper separation of concerns and results in a program that is not scalable.

**Do the classes contain proper access specifications (e.g.: public and private methods)?** Yes No Partly (Can be improved)

Comment on your analysis: A more effective encapsulation and separation of concerns could be implemented by declaring certain functions as private and isolating them to their respective class.

The real issue here is that the functions are not separated enough to allow for the proper balance between private and public functions. As mentioned earlier, manipulating the amount in an account can be done by one [private!] function (for deposit, withdraw, and transfer).

Comment on your findings: It appears that NO classes are privatized.

**Reusability:**

Are the programmed classes reusable in other applications or situations? Yes, most of the classes No, none of the classes Partly, some of the classes Don’t know

Comment on your analysis: Both the user and account functionality are completely not reusable. They are coupled into the same class and this therefore hinders both their ability to be reused in another system as a user and account class respectively. The manager class could be made more reusable (as mentioned above) if it were designed as a more generic ‘controlling class’.

Comment on your findings: The functions in the user class and the implementation of the account system are coupled into the same class.

**Simplicity:**

Are the functionalities carried out by the classes easily identifiable and understandable? Yes No Partly (Can be improved)

Comment on your analysis: The naming and organization of the classes, functions, and variables properly reflected their functionalities. There was no confusion here.

Comment on your findings: No discrepancies between the setup and the functionalities.

Do the complicated portions of the code have /\*comments\*/ for ease of understanding? Yes No Partly (Can be improved)

Comment on your analysis: The amount of comments was definitely sufficient to understand the code. Comment on your findings: Comments on top of all functions and variable declarations. Also comments on top of various other complicated processes.

**Maintainability**:

Does the application provide scope for easy enhancement or updates? (i.e., enhancement in the code does not require too many changes in the original code (see, for example, requirements of the “enhancement” project))

Yes No Partly (Can be improved) Don’t know

Comment on your analysis: As mentioned in the section of ‘separation of concerns’, this program is not particularly well separated. This means that changes to one aspect of the program might have affects on others. For example, the User and Account functions are completely coupled into the same class. Thus, changing something in the User functionality would most definitely have an effect on the account functionality. This would make updates significantly more tedious than if the program was more modular.

Comment on your findings: Refer to findings for separation of concerns.

**Efficiency:**

Does the design introduce inefficiency in code (e.g., causes too many nested loops or delays in concurrent processing)?

Yes No Partly (Can be improved) Don’t know

Comment on your analysis: If the accounts were separated into their own class, then the ‘total’ function in the manager class could loop through account objects to get the total. Right now it loops through all the users and checks both checking AND savings accounts. Some of these users might have no money in either accounts, or money in just one of their accounts. This makes looping through all users and both accounts for each inefficient. Other than that, the system appears efficient.

Comment on your findings: The ‘total’ function in the manager class loops through all users.

**Depth of inheritance:**

Do the inheritance relationships between the ancestor/descendent classes go too deep in the hierarchy? (The deeper a class in the hierarchy, the greater the number of methods it will probably inherit from its ancestors, making it harder to predict its behaviour).

Yes No Partly (Can be improved) Comment on your analysis: User hierarchy is only a 1 level hierarchy.

Comment on your findings: Only hierarchy exists is with users.

Children:

Does a parent class have too many children classes? (This could possible suggest an abstraction problem.)

Yes No Partly (Can be improved)

Comment on your analysis: There is appropriate hierarchical structure (no excessive children).

Comment on your findings: There is a user class that has 3 children (customer, manager, maintenance).

D6 CCCC Software Metrics Report

D7 Spreadsheet of historical OO measures

D8 Quality trend across releases R1 and R2

D9 Modified design including design pattern

D10 Explanation of design pattern

Factory Method Pattern: The factory method design pattern is a creational pattern that can be used in the banking system application really well. As opposed to most design patterns opting to use new classes, with the Factory method only a new operation is required. In a modified version of the design and banking patterns we can use the factory method pattern by using a template method to create instances of a class.

An account manager would make use of the template class and be the factory in the situation. The account manager template class would be responsible for creating accounts for clients. The vendor system would act like the account manager and have control over creating and managing specific accounts. The account manager class would contain an interface for creating objects while subclasses like the vendor system would control what is instantiated at the time (i.e. new accounts, transactions etc)