# ECE 651: Foundations of Software Engineering

**Final Report**

**Group 3**

# FreeX

**Ke Chen 20456026 (kchen)**

**Zhechen Du 20340480 (z6du)**

**Shangru Li 20637274 (s488li)**

**Hongming Wang 20667749 (h539wang)**

**Zhao Zhang 20689699 (z664zhan)**

# Table of Contents

*List of Figures*



*List of Tables*

# 1. Introduction

With the rapid development of globalization, the interactions among different countries have been increasing every year, this is especially true in terms of volume and frequency. The flexibility and liquidity of the currency are always a great concern among central banks, businesses, and even individuals. [1] However, secure mobile platform is very hard to find. Thus, the idea of this project is to develop a mobile application with user friendly platform which can support individual currency exchange with reasonable exchange rate. To achieve this, the architecture design plays an important role of system.

# 2. Architecture

## 2.1. Comprehensive Architecture

The comprehensive architecture design is based on supporting functions. Form the user stand point, the application function can be divided to two parts. The first part is done in the front-end, which are simple functionality that don't require the access of backend server. The second part are more complex functions that need to connect with web server and database. Isolating these functions would decrease coupling of the system and reduce the burden on the server, which would embody efficiency and complexity as non-functional properties(NFP).

### 2.1.1. Functions achieved in the front-end

Functions such as real-time exchange calculator and financial breaking news do not require connection to the back-end server. Web API can be used to achieving these functionalities.

### 2.1.2. Functions require connection to server and database

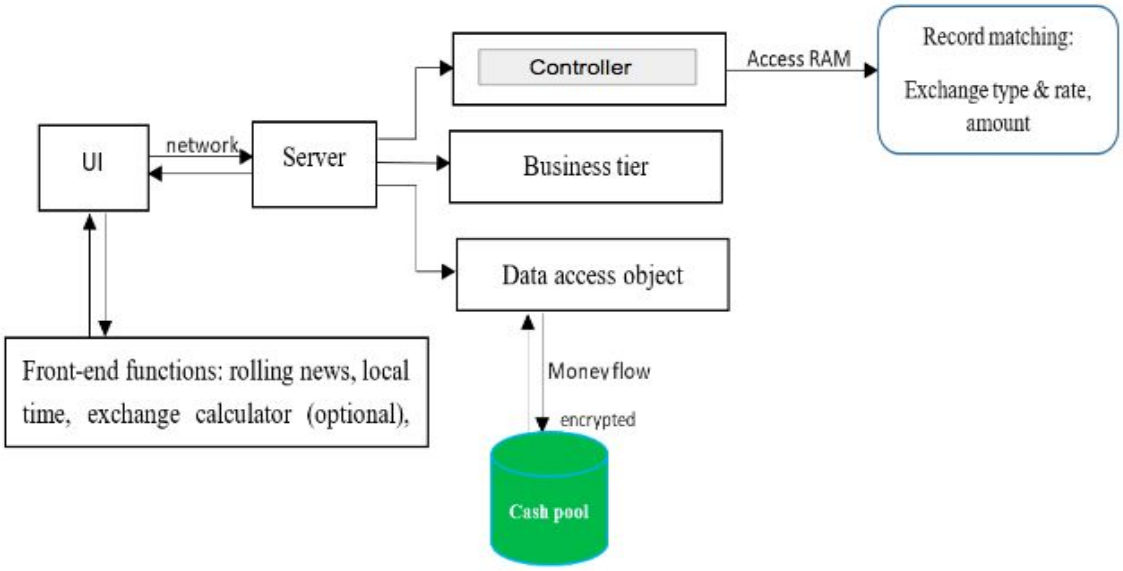


*Figure 1. Comprehensive Architecture Diagram*

In general, the system uses MVC architecture for the overall design. [2] The system design contains three layers which are client side layer, connector layer and backend layer. Be specific, the backend layer can be divided as business tier (algorithm database) and database (abstract common database). Speaking of the functions which require to connect web sever and database includes login, signup, fetching transaction history and balance, checking user information and trading. In specific, the trading part contains deposit, withdraw, buy in desired currencies urgently and sell holding currencies with ideal exchange rate.

Two of the functions shown above, buy and sell, need to connect with the business tier which is the self-designed algorithm to match clients' desired currencies. Rest of functions, such as login, signup, deposit, withdraw, checking balance and fetching user information are directly communicate between front-end and database through server. Controlling some functions isolate from currency exchange algorithm database is to comply efficiency NFP.

Since those functions mentioned above require connecting with backend, data transmission plays an important role for this system. Thus, the data transmission is suppose to be efficiently passing through three layers and security store in database. To comply with security NFP, the system contains two times encryption by using MD5 methods in data transmission process: from UI layer to connector layer and from connector layer to database layer. [1] There is a rule based checking point to make sure only necessary information are passed to the client. Thus, the system would ensure that the data transmission process is secure. The system uses sessions to identify different

clients. User ID and transaction history are the primary keys during the transmission process.

## 2.2. Server Architecture

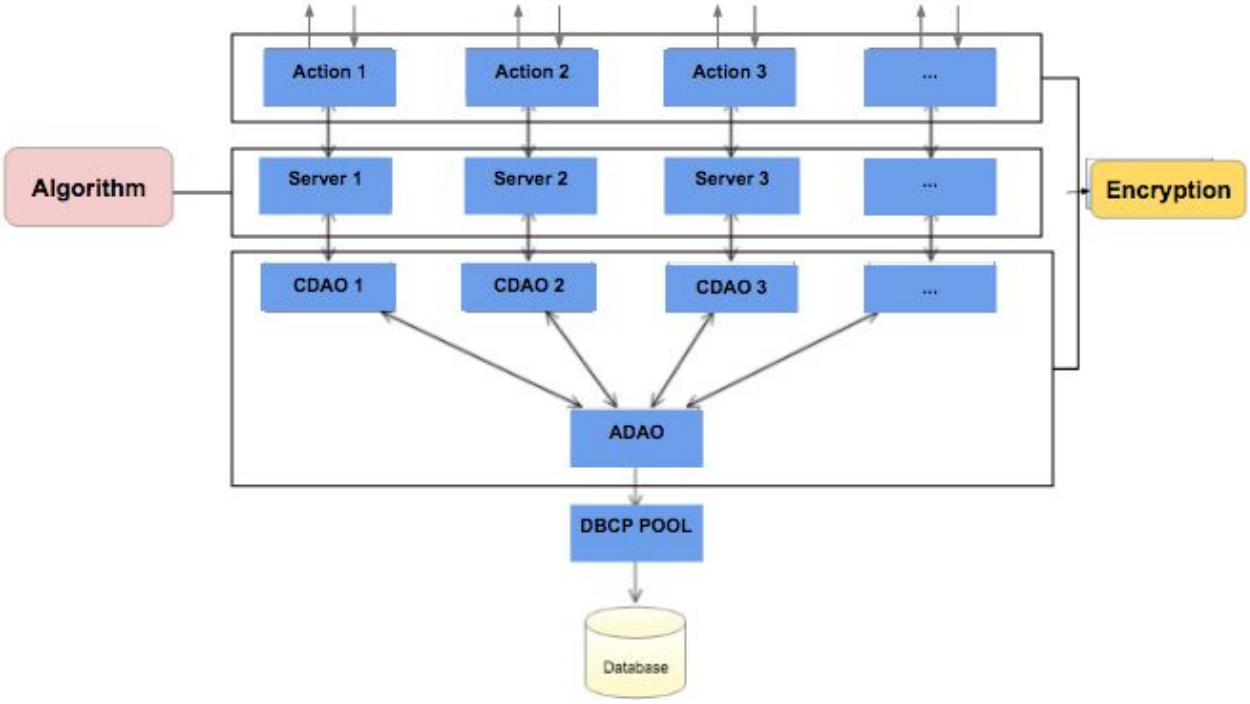


*Figure 2. Server Architecture Diagram*

On server end, the entire system was divided into three layers, namely, they are connector layer, service layer and data access object layer. On each layer, the architecture was designed to satisfy security, scalability and dependability.

The connector layer is responsible for the request dispatching and connection to the clients. The Sturts2 framework was used for better encapsulation of HTTP methods and easily communication with both client and service layer. For higher security, MD5 encryption was used on this layer for both encrypting the information and sending to client through network and decrypting the information which is form the clients. In addition, the Struts annotation would map different request to different action methods to increase scalability. [1] All methods on this connector layer were designed to use user classes as the parameters. That means connector layer only needs to take care of the communication with clients and the dispatching based on different requests and reture result, and it connects to the service layer by dependency. This could make the entire layer more independent with less coupling and more cohesion.

The service layer contains the business logics and the core algorithm which is used to match transaction requests from users. Basically, the connector layer would obtain user requested and encapsulate the information as objects. It then invokes a certain service

layer method to pass the parameters to the service layer. The trade matching activities and matching operations would execute on this layer. If the trade request does not match in the algorithm, the service layer would return the request to the connector layer; otherwise, it would invoke a certain data access object method based on the generic type to pass the data to the next layer. Single responsibility principle would be demonstrated on this layer.

The data access layer is responsible for the data operation. On this layer, the factory design pattern was used to decrease coupling and increase scalability of the system. Basically, the abstract common data access object would act as a factory for generating the specific type of the subclass to operate the database. If there is a table in the database extended in the future or extending a new table, it is easy to extend the data access object layer to satisfy the requirement by adding a new subclass of common abstract data access object. For the security, the MD5 encryption was also used before the data goes into the database. [2]

## 2.3. Exchange Algorithm Architecture:

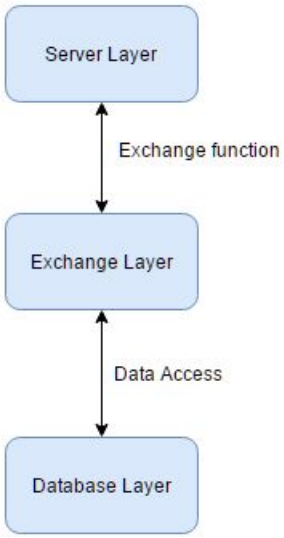


*Figure 3. Exchange Algorithm Architecture*

The exchange algorithm can be split up into two layers. The exchange layer, and the database layer. The exchange layer retrieves seller and buyer info from the server. It will then pass the required information into the specific database access function. After the

database returns the required info, the exchange layer will then calculate the necessary info and pass the required field back to the server.

The database layer contains three different databases. By decoupling exchange functionalities into separate databases, it improves efficiency and scalability of the database. Furthermore, it enables concurrent access to different databases. The database itself is designed with efficiency and scalability in mind. Further details will be explained in the design part of this report.

## 2.4. Android Client Side Architecture

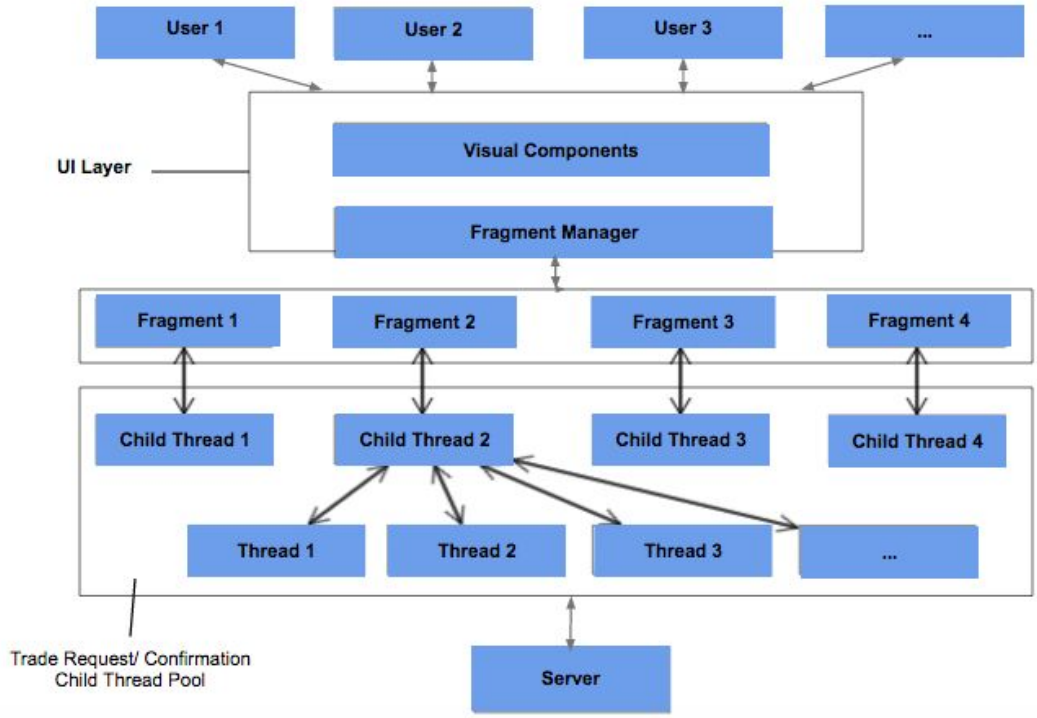


*Figure 4. Client Architecture Diagram*

On the client side, there are three layers between users and the server side. When interactive events are triggered by users, the inputs go through each layer respectively with minimum delay. The details and analysis of architecture are given as follows.

In the UI layer, there is a fragment container which takes up the most of the screen size. This general-purpose container is responsible for displaying the content of every fragments. We manage to build a trade UI layout that supports four types of transaction. Most of the control widgets are reused for all different types of trade. By passing a type identifier, the system identifies the specific type and then enable the components accordingly in the UI.

The reason why we use fragments instead of activities as the function supporter is because fragments are more lightweight with less computational load. Therefore, all functions in the client side are divided into four fragments. All these fragments are in the charge of a fragment manager. Because of this, the user can switch among these fragments to use different functions without redundant setup and waiting. Every fragment is parallel with each other and keep a weak coupling regarding data. Therefore, newly developed features can be attached to any fragment easily. It can also duplicate itself without modification to the original feature.

To get better responsive properties, we use a multi-thread programming technique named AsyncTask in the data exchange layer. [4] We typically encapsulate time-consuming operations like sending HTTP request to the server as well as receiving responses from child thread pool. When the communication is achieved, UI layout update operation is invoked in the main thread.

# 3. Design

## 3.1. Server Design

The server contains three layers for the different classes, and each layer has the same function classes and provides the interface for the upper layer to invoke.

The lowest layer is Data Access Object(DAO). On this layer, there is a common abstract data access object class(BaseDAO) which is responsible for creating the specific type of the data access object class. For the specific type of the entity data access object, they are in the DAO package. UserDao is responsible for the operations of User table; UserHistoryDao is responsible for the operations of UserHistory table; BalanceDao is responsible for the operations of Balance table; TransactionHistoryDao is responsible for the operations of TransactionHistory table. This layer exposes interfaces to the service layer.

The service layer is upper layer of the Data Access Object layer which contains the transaction details and algorithm. UserService provides methods for the User entity related service details. UserHistoryService provides methods for the UserHistory entity related service details. BalanceService provides methods for the Balance entity related service details. TransactionHistoryService provides methods for the TransactionHistory entity related service details. This layer exposes the interfaces to the connector layer.

The connector layer is responsible for getting requests and dispatching requests. We use this layer to connect the service layer and clients. UserAction dispatches user related requests and implements methods in service layer. TransactionHistoryAction dispatches trading request and implements methods in service layer. UserHistoryAction dispatches requests related to users' history and implements UserHistory methods. BalanceAction dispatches balance related requests and implements BalanceService methods.

For the design of this system, the factory design pattern is used in the data access object layer for a better scalability of new object database operations. The service and DAO layer are designed to use interfaces for the future extension. All the data passing through each layer is encapsulated as objects for the better.

Based on this specific system with its functionalities, select this design model is the most manipulated and easy to understand for beginner. Since this system requires data interaction and concurrency, it is necessary to design layers to lower the coupling and extend scalability to the whole system. On the other hand, using abstract common database accessing object obeys open and closed design principle; also, it supports the extension of more functionalities in the future, which complies evolvability NFP. Moreover, using controller is easy for whole system to identify different clients with their different demands. Compare to other alternative designs, this design applied is the easy to

handover which obeys Liskov Substitution principle. [5] The external interface is straight forward and clear.

For the coupling and cohesion, the design of this system is used MVC architecture, and different layer is responsible for different obligation. For example, the connector layer which is Action layer in the source code is used for connect to the clients and invoke the service layer. All the data would be encapsulated before passing to the next layer. Spring IoC would be used to manage creating new objects after the initialization of the entire system. Therefore, each class would have dependency relationship with its invoking layer, and it could decrease the coupling of each layer. Single responsibility principle would be used in this design for each layer and class. For example, each DAO class is responsible for a certain database operations of a object. If it needs to be extended for new demands, it could just add new methods to satisfy the new requirement.

For the future extension, this design supports to add new functions and modify the existing functions. If the new function requires new entity which does not exist in the database, a new object can be added in entity package as well as its attributes. Then creating classes in action layer, service layer and DAO layer for the purposes of connection to the client, handling transaction details and persisting the data to the database. However, if new function does not need a new entity, new methods can be added in each layer or extends the existing classes.

There are some shortages needed to be improved in this design. First, the Spring IoC control would be changed to automatic assembling to decreasing the code interaction. [6] For example, automatic assembling does not need to write set methods to inject the instances. Second, some transaction related code should be moved to service layer to separate the connection layer and service layer. In addition, it could make connection layer more lightweight, and increase the cohesion of the system.

## 3.2. Android Client Side Design

The client side mainly contains three activities. Due to long setup time of the application, we created an activity called WelcomeActivity to handle tasks like loading data and network deployment. We register it as the launch activity by adding filter in AndroidManifest.xml. This activity displays a splash screen and execute the operations for initializing the application. We use this design to maintain normal start up of the application without affecting the user's experience. The second activity is FullScreenActivity. It defines all the logics before login. The user can trigger different buttons to pop up dialogs of login and signup. The last but the most important activity is MainActivity. MainActivity is a functionality module with a fragment manager. the fragment manager is used to manage the fragment transactions. All functions in the activity are divided into four fragments within it.

Data exchange between activities and fragments is an important design concern. For instance, once the user is logged in, the system should keep an entity for storing the information of that user globally. In this project, we use the interface to reduce the coupling among multiple classes. Specifically speaking, to pass an instance of User class, MainActivity implements the interface called UserPass, and the fragments that need in this instance directly holds a UserPass field as a listener. Then by casting MainActivity into Userpass, and assigning it to the UserPass listener that fragments hold, fragments can safely visit the public fields that MainActivity holds.

For the UI to improve usability and support compatibility, several UI classes are inherited from the superclass. In TitlePopup class, we use Adapter pattern to customize the output format of the list view. Using this design method helps to improve the transparency of the code, other developers in our team can easily understand the principles of the software, thus make the code more reusable.

Most of the fragments and activities contains many widgets that need to be programmed to achieve certain tasks. Rather than adding listeners to them individually, we assign a single listener to all of them, and identify them with their resource ID in the trigger function. This kind of design appears several times in the project. Since doing this, the scalability is improved, more widgets can be added with a switch-case branch including new widget's id. This eliminates modification of the software structure

## 3.3. Exchange Algorithm Design

The exchange algorithm is designed with emphases on efficiency and scalability. On the exchange layer, if a trader wants to buy a certain currency, the exchange algorithm will query for the best possible exchange rate in the trade database. If one seller could not match buyer trade amount, multiple seller will be added together to match the buyer trade amount. The trade information and seller information will then be pushed into a stack and outputted to the server, the trade information will then be passed to the buyer. If the buyer decides not to buy, the seller info will be pushed back into the database. Currently, the precision of the exchange system is set at four digits after decimal place.

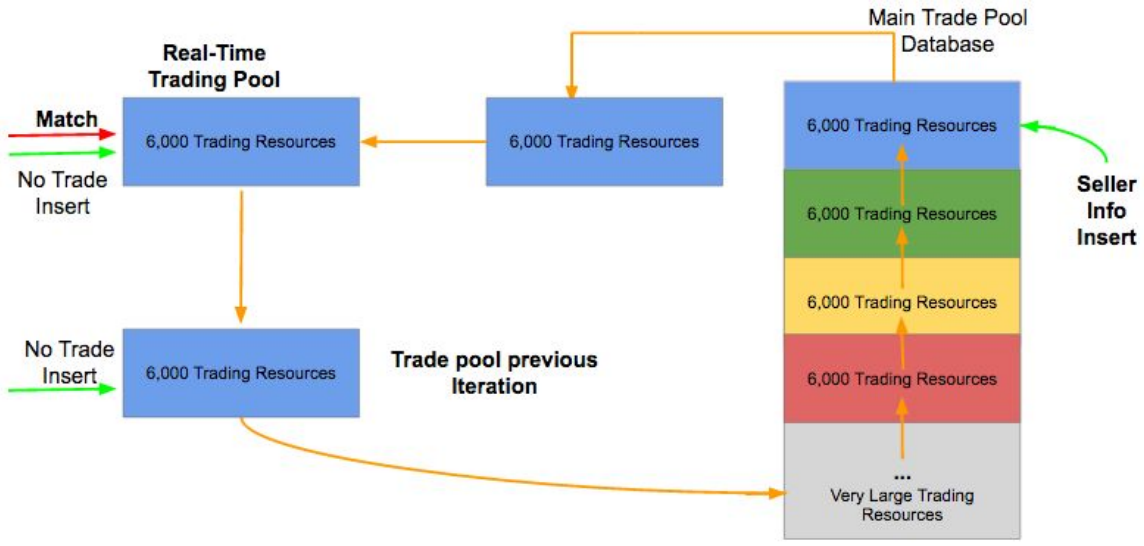


*Figure 5. RAM Database Diagram*

In total, the exchange algorithm contains three separate RAM databases using H2. [7] The trade database, the sort database, and the main database. As illustrated in Figure, the database block will flow to the next state after a set amount of time. Currently, each trade database contains 6 different trade scenarios with 1000 trade resources in each scenario. The trade resources are sorted by exchange rate. To quickly quarry the database, a separated table is created in each database to store each currency's top available seller row position. One of the advantage for the design is scalability. New currencies and trade scenarios can easily append to the end of the database. Because the compactness of the database, adding 100 new currencies will only expand the database to 4 million rows, which is significantly smaller than the server RAM space.

The trade database contains two different tables, the first table is used to match buyer with seller, the seller position is retrieved from the currency position table. if there is a match, the seller information will be popped out the database. If the buyer decides not to trade, then the seller info will be pushed back into the trade pool. If the database changed to the next state, which means the trader is no longer in the real-time trade pool, then the seller info will be pushed into the second table, the previous iteration trade pool. The second database is the main database, it is used to most of the buyer info, it can be dynamically expanded when there are more data entered the database. The seller information will also be inserted into this database. The sort database is used to sort the exchange rate from the lowest to the highest. The sorted database will then insert into the real-time trade pool in the next iteration.

*Table 1. RAM Database Structure*

| Id | UserID | | Currencyinout | Amount | Rate | Time |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | | 12 | 1000 | 5 | 2 |
| | …. | | | | | |
| 1001 | 2 | | 21 | 1000 | 0.2 | 5 |
| | …. | | | | | |
| | …. | | | | | |

## **3.4.** Class Diagram

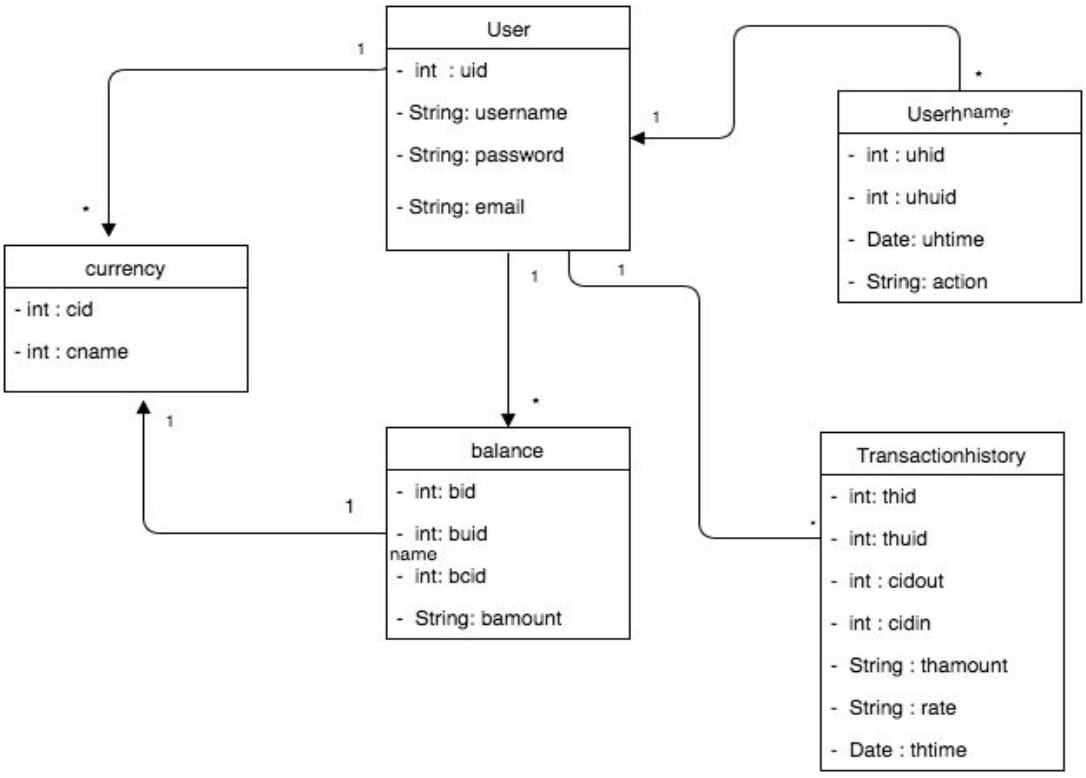


*Figure 6. Class Diagram*

## 3.5. Sequential Diagram

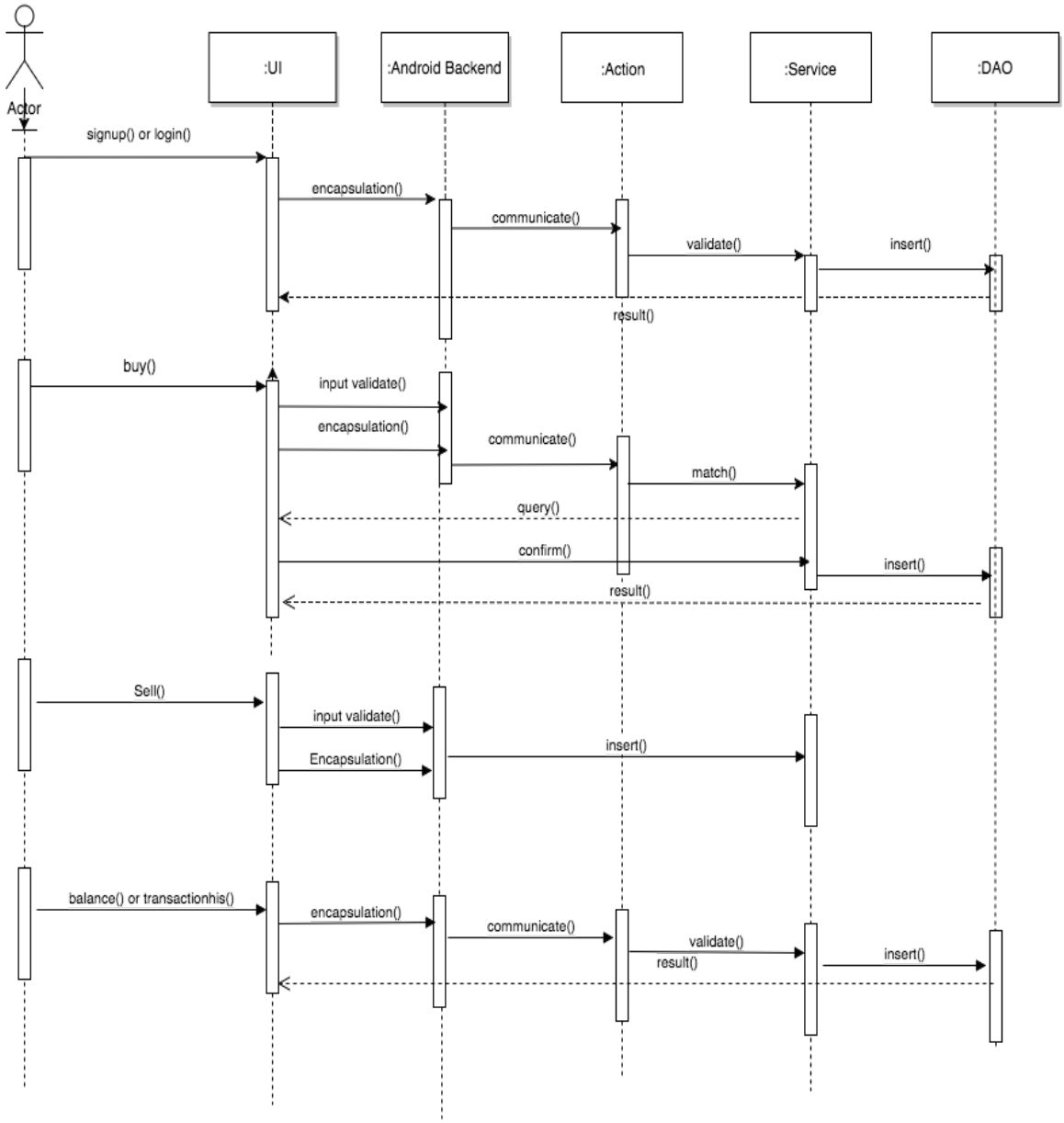


*Figure 7. System Sequential Diagram*

# 4. Group Member Contribution Summary

Hongming Wang:
Design and implement the client of FreeX in Android platform. Test and debug the system. Coordinate the data interaction between the client and the communication layer of back end. Also, design an algorithm for encryption in front end as well. In addition, analyze several real-world situations and then provide a solution to the face-to-face transaction model.

Shangru Li:
Basically, I am responsible for the design and implementation of the server architecture and the implementation of the data access object layer. In addition, I am responsible for the design of the objects and the relationship between them in database. Implementation of the server architecture with the Struct Spring and Hibernate framework with Maven. Implementation of the data access object layer with factory design pattern.

Zhao Zhang:
I am responsible for the connector layer. In this layer, I designed the application logic and support the android clients. In this layer, we receive all the requests and then works with the server layer to prepare any data needed by the android clients and generate the final response. Also in this layer, we perform the business operations that modifies the state of the data model.

Zhechen Du
I am responsible for the design of exchange algorithm and RAM database. The main functionality for this layer is buy and sell. but major work is to improve algorithm's efficient and scalability. Things like database sorting and iteration change are essential to the success of this project. Because of all this, the entire algorithm and database is written without using any open source code or module.

Ke Chen:
Design the client side logic with hongming and implement some of UI layout. I am also responsible for generating and scanning of QR code for face to face trade. Besides, I am responsible for functionalities achieved on UI side such as real time exchange rate calculator. Also, generating all type of data and using MD5 encryption in R for filling the database. I also conduct the overall planning and projections as a "connector".

# Source Code

Source code of the client available at: https://github.com/chesterboy01/FreeX.
Source code of the back end available at: https://github.com/Ruins7/FreeX_Server.

# References


[1] A. Jim and C. Gregory, "Liquidity Risk at Banks: Trends and Lessons Learned," December 2008. [Online]. Available: http://www.bankofcanada.ca/wp-content/uploads/2012/01/fsr-1208-armstrong.pdf. [Accessed 1 December 2016].

[2] "Basic MVC Architecture," Tutorials Point, 2016. [Online]. Available: https://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm. [Accessed 1 December 2016].

[3] "MD5," Tech Target, September 2005. [Online]. Available: http://searchsecurity.techtarget.com/definition/MD5. [Accessed 20 September 2016].

[4] "struts," Apache, 2016. [Online]. Available: https://struts.apache.org/. [Accessed 20 September 2016].

[5] "AsyncTask," Google, 2016. [Online]. Available: https://developer.android.com/reference/android/os/AsyncTask.html. [Accessed 1 Decemeber 2016].

[6] "Liskov's Substitution Principle(LSP)," Oodesign, [Online]. Available: http://www.oodesign.com/liskov-s-substitution-principle.html. [Accessed 1 December 2016].

[7] "spring IoC," Spring, [Online]. Available: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html. [Accessed 1 December 2016].

[8] "H2 Database," H2, 2016. [Online]. Available: http://www.h2database.com/html/main.html. [Accessed 2 Decemeber 2016].