System Architecture and Design

Advisor: Prof Olga Baysal, TA.Mehardeep Bhalla

Course: COMP3004 Winter 2020

Team Name:

Just Five

App Name:

Daily+

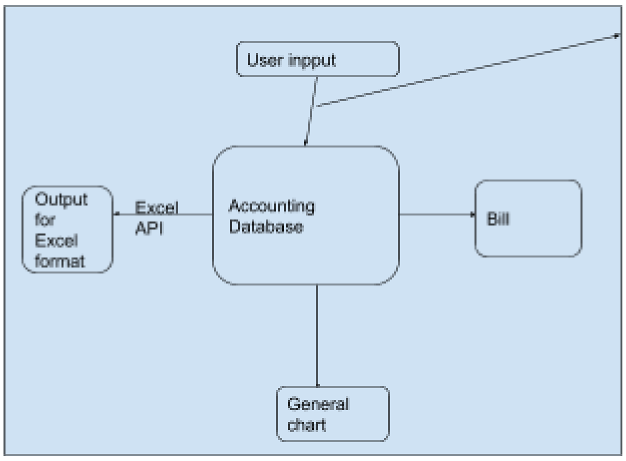
Team Member:

Jie Ren	101087545
Zhenqing Lang	101087276
Payton Pei	101065299
Haiyang Ni	101067092

Abstract:

This semester, our team is developing financial accounting software, because for students, developing an accounting habit can help them to rationally plan their assets in the future. This is also the habit that our members want to cultivate, so we took this opportunity to develop accounting software that can run on an Android system.

For our mobile software, its design styles are a more inclined Database-centric architecture style and Component-based software engineering style. This database-centric architecture or data-centric architecture as a data center is usually related to the software architecture where the database plays a key role. In our software, the database is associated with the amount, category and date of the items entered by the user. And the user can directly interact with the database through simple interaction, for example, long-press an already added item, the user can choose to delete, or can click the plus sign to add a new item. For our functions such as bill display, increase or decrease data, excel form generation, pie chart generation, these functions are based on data construction and use SQL statements to call database data. These important functions are inseparable from our database. The reason why our team uses the database as the basis for communication is that it uses the transaction processing and indexing provided by the database management system to simplify the design to achieve high reliability, performance, and capacity.



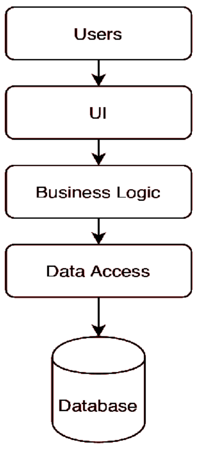


Figure 1.1 (Database-centric Architecture)

Figure 1.2 (Detail for user input process)

The other architecture style which is a component-based software engineering style, the main reason we use this design approach is that this style is easy to develop, system maintenance and development, reusable and independent. For example, in our code structure, the database is our main receiving and transmitting component, and multiple components want instructions from the database unidirectionally, such as chart and bill. This allows one component to be used in multiple components, which means that when software maintenance is required, maintenance costs can be reduced, and changes can be easily implemented and updated without affecting the rest of the system. In our development, when the database is established, our team can develop independently and flexibly, because this is all developed in parallel, which can improve the productivity of our software development.

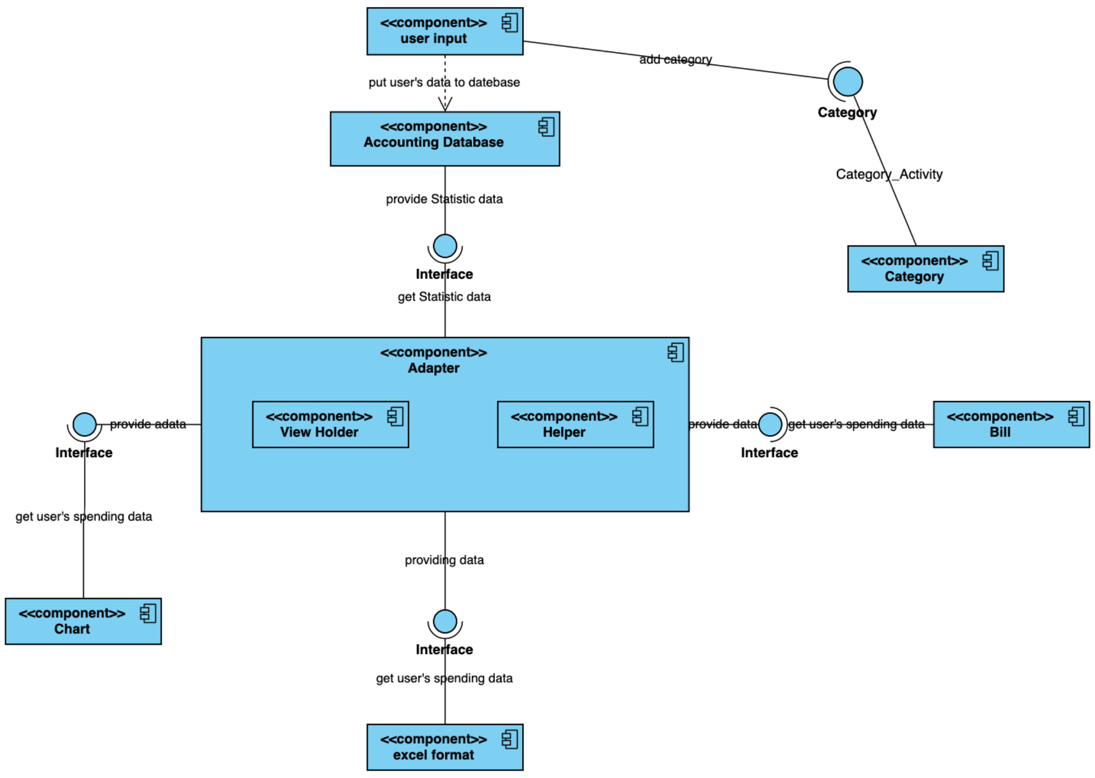


Figure 2.2 (component-based software engineering diagram)

This diagram shows the relationship between different components. As can be seen from the figure, each component is independent, and for some components are one-way requirements such as chart and bill, so this is easier to change and update implementation without affecting the rest of the system.

Overview of the functionality that Daily+ aims to provide:

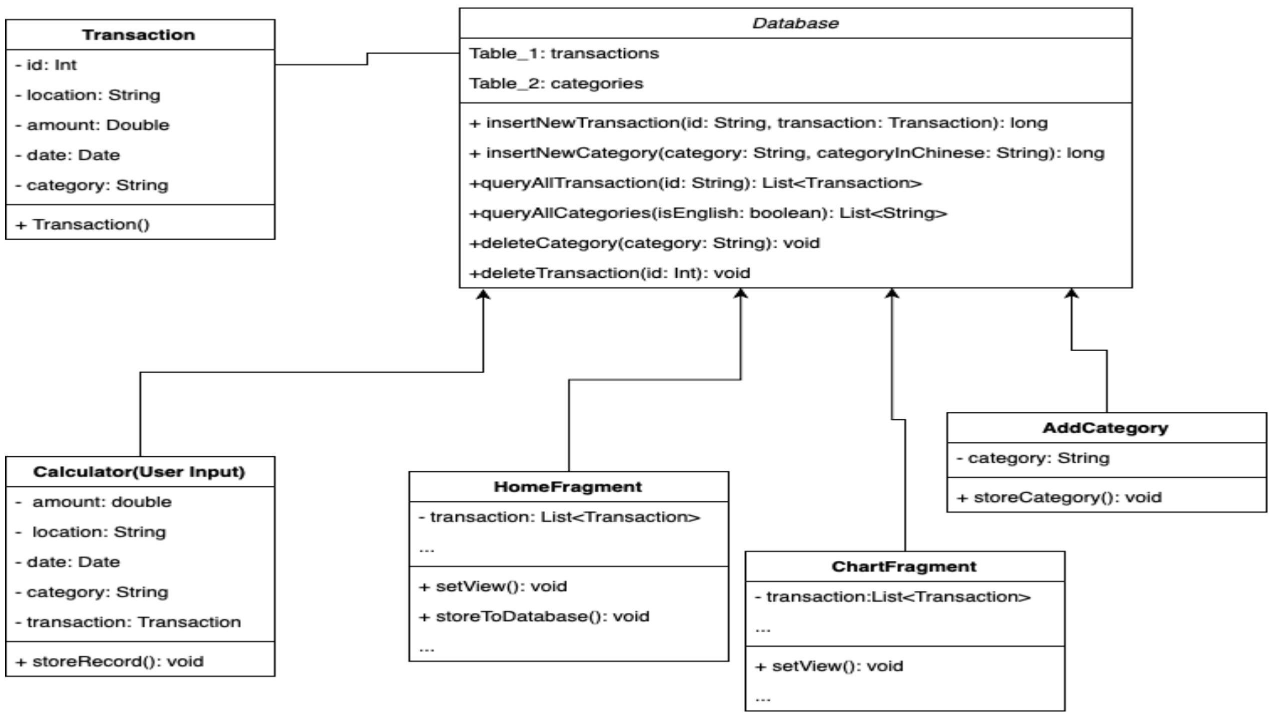
- Enter the spending/income information
- Delete expense records
- Calculator
- Data visualization
- Chart generator
- Login function
- In-App user feedback

Non-functionals, quality, and attributes Daily+ wants to exhibit:

- Decent User Interface
- Password login
- Language settings
- Space requirement and software performance

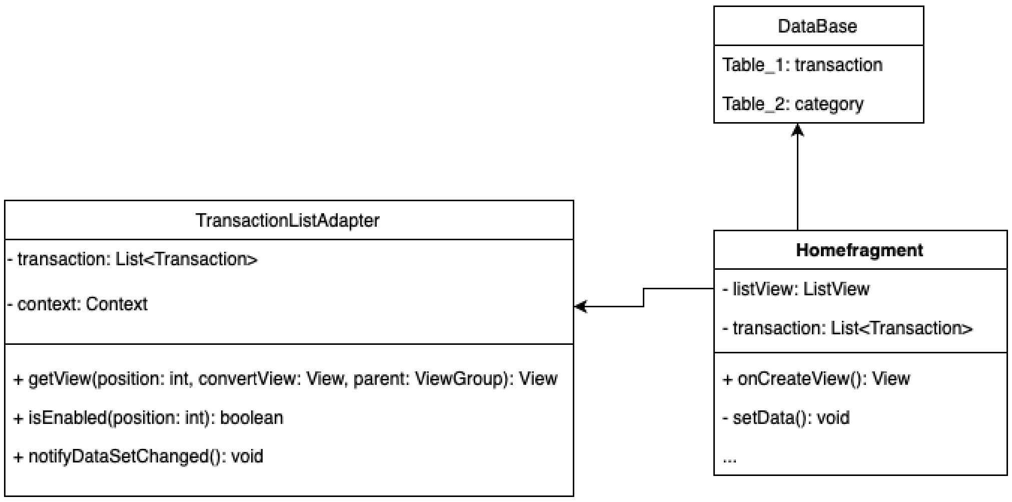
Design Pattern:

One of the design patterns we are using is **Singleton**, which is a creational pattern. The intent of Singleton patterns is to ensure a class has only one instance, and provide a global point of access to that instance. It is the reason that we choose this pattern to design our database. Our goal is to create a single database object, and it can be accessed by other objects. Since we have multiple fragments that need to access the database, this design pattern can prevent us from having multiple copies of the database.


Transaction
- id: Int
- location: String
- amount: Double
- date: Date
- category: String
+ Transaction()
Database
Table_1: transactions
Table_2: categories
+ insertNewTransaction(id: String, transaction: Transaction): long
+ insertNewCategory(category: String, categoryInChinese: String): long
+queryAllTransaction(id: String): List<Transaction>
+queryAllCategories(isEnglish: boolean): List<String>
+deleteCategory(category: String): void
+deleteTransaction(id: Int): void
Calculator(User Input)
- amount: double
- location: String
- date: Date
- category: String
- transaction: Transaction
+ storeRecord(): void
HomeFragment
- transaction: List<Transaction>
...
+ setView(): void
+ storeToDatabase(): void
...
AddCategory
- category: String
+ storeCategory(): void
ChartFragment
- transaction:List<Transaction>
...
+ setView(): void
...

For instance, the input fragment first receives a spending record that the user entered, then it is stored into the database. Then other fragments (i.e. home, chart) can access that information through that single database, and it makes sure that the data is consistent. The strength of singleton pattern is that we only need to initialize the database once; the weakness is every component can access the database and sometimes it can be hard to sync the data.

The secondary design pattern thas the application used is the **Adapter** design pattern, which is to transform the interface of a class into another interface clients expect. Since we need to convert data in the database into different formats of data visualization


DataBase
Table_1: transaction
Table_2: category
TransactionListAdapter
- transaction: List<Transaction>
- context: Context
+ getView(position: int, convertView: View, parent: ViewGroup): View
+ isEnabled(position: int): boolean
+ notifyDataSetChanged(): void
Homefragment
- listView: ListView
- transaction: List<Transaction>
+ onCreateView(): View
- setData(): void
...

For example, we want to show a list of spending information in the home fragment. Once a user opens the home fragment, then home fragment requests the adapter to create a list of records using the data inside the transaction list. The pro of the adapter pattern is that one adapter only responsible for one purpose.

System Design:

To build our mobile app more rationalization and easy to maintain or other programmers implement new features based on the current version appropriately. We design our system carefully.

Database Design:

Our income and expense data are stored in a Relational Database, and most of the functions are based on bills, which means closely related to the database. Under this situation, we apply the **Singleton** design pattern in our mobile program. Which means database as our global class and only one instance. Therefore, it provides a global point of access to it. Designing a good format database is our first difficult to overcome. We divide information into subject-based tables to reduce redundant data. The database satisfies Boyce-Codd Normal Forum. We designed 2 tables(image 1), one is called “Transactions” and another is called “Categories”. In the “Transaction” table, we set the ID as the primary key to access data like amount, location and date. Besides, we also have a “category” column which is a foreign key connected with the “Categories” table. We have 2 columns in the “Categories” table, one is the “English” column which is the primary key and the other is the “Chinese” attribute. Our potential user is an international student, after designing the "Categories" table, users can add custom tags and developers can organize code nite and save memory space. Also, we can just add more columns to support more languages. This will make the work of other developers easier. After the database design part is done, We need to solve the interaction between data and user action. Database-centric architecture(Diagram 1.2) is the most suitable solution for our program. The last problem we met is how we show the data stored in the database to our user. We use the Adapter design pattern as our solution. We retrieve data from the database and put in transcationListAdpater that extends ArrayAdapter to operate the data. Finally, users can access data efficiently and beautifully.

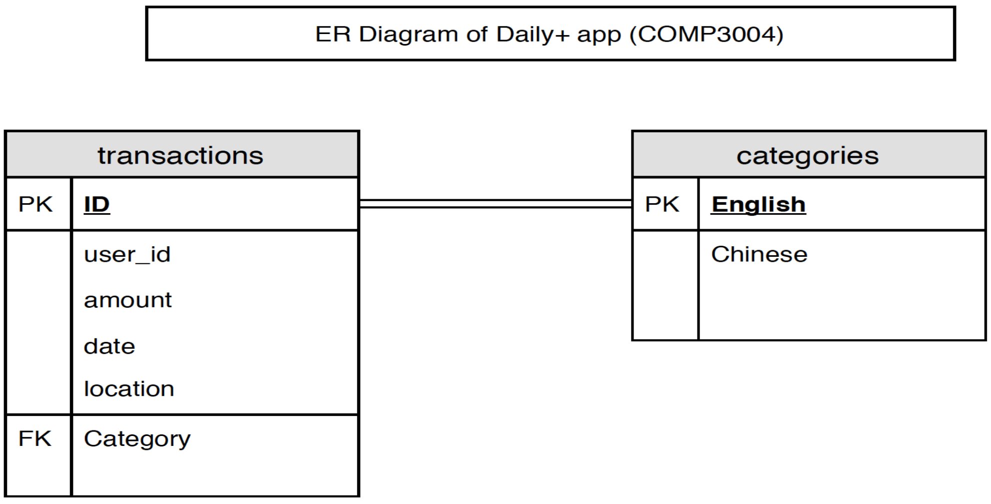


Figure (2.3)

Application Programming Interface Interaction:

Facebook Login API:

Client Side:

When a user chooses to login our Daily+ using his Facebook account. When the user clicks the login button, a login request will be sent to the Facebook server. And this request wants to get First name, Last name, and image according to the corresponding Facebook account

Facebook Server:

When the Facebook server gets the request that is sent by the user, it will begin to look for the user id. When it finds the email in the database, it will start to check if the password the user typed in is matching the password that is stored in the Facebook server. Once the input password matches the password in the server. The server will send back the information that the user demanded.

The information that Daily+ demanded contains 4 strings. 1st string is the first name, 2nd string is the last name, 3rd string is user email and 4th string is the profile icon URL

Client side:

When Daily+ gets the information we requested, we will begin to change default information with the user's actual information on the Facebook server. First name, last name and email will be changed automatically. However, the profile icon takes another step. The client only gets the image URL; he needs to parse the URL to an image by using Facebook's parsing tool.

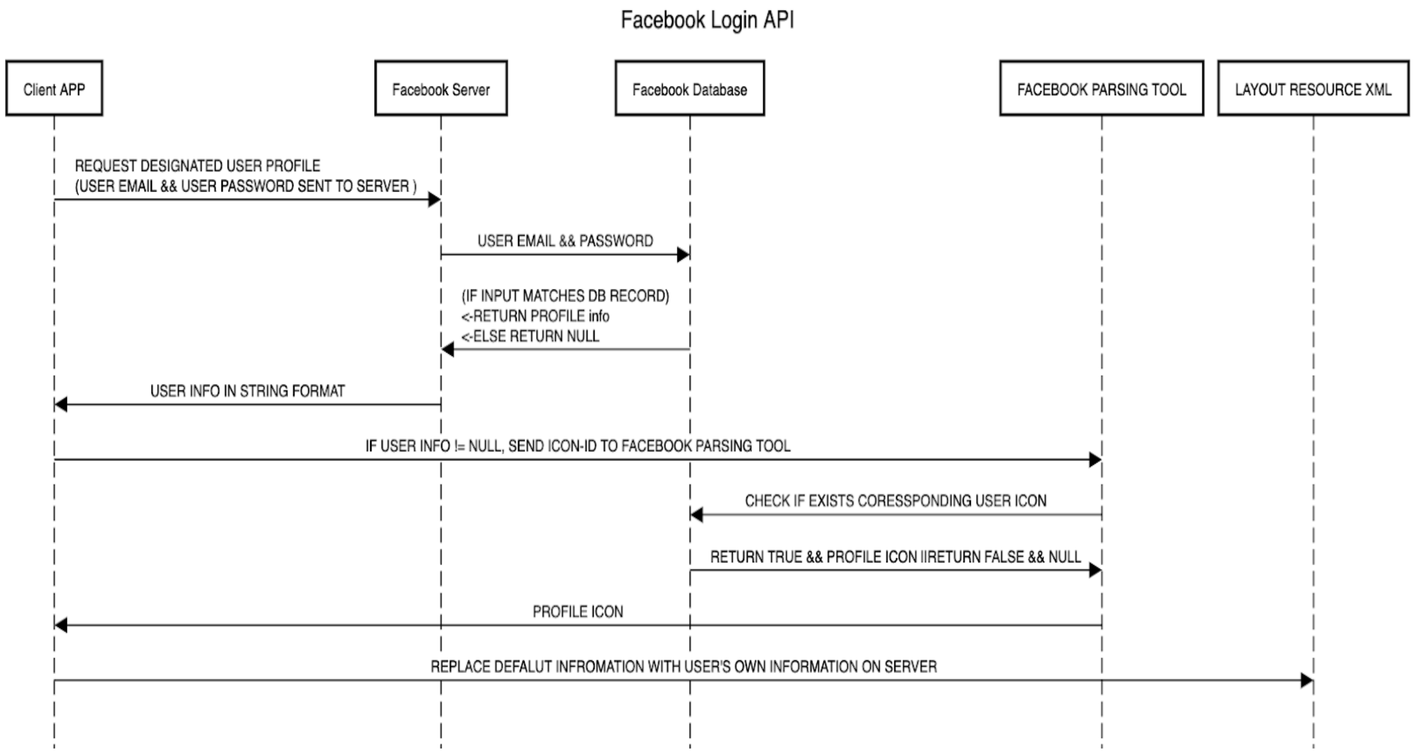

Facebook Login API
Client APP
Facebook Server
Facebook Database
FACEBOOK PARSING TOOL
LAYOUT RESOURCE XML
REQUEST DESIGNATED USER PROFILE
(USER EMAIL && USER PASSWORD SENT TO SERVER)
USER EMAIL && PASSWORD
(IF INPUT MATCHES DB RECORD)
<-RETURN PROFILE info
<-ELSE RETURN NULL
USER INFO IN STRING FORMAT
IF USER INFO != NULL, SEND ICON-ID TO FACEBOOK PARSING TOOL
CHECK IF EXISTS CORESSPONDING USER ICON
RETURN TRUE && PROFILE ICON ||RETURN FALSE && NULL
PROFILE ICON
REPLACE DEFALUT INFROMATION WITH USER'S OWN INFORMATION ON SERVER

Figure 3

Circular Image View for Android API:

Client-side: Create a normal image that where the client gets from Facebook servers. It will create a Circular image button. (*Circular Image is stored in Maven ()*)

Maven Software's server: It will process the square image and render the image into a circular image. And send the circular image back to the XML resource file.

Our Approach to minimize coupling and accommodates changing requirements :

We tried to Useless class inheritance and use interfaces to hide implementation details. In addition to the introduction of java object-oriented programming interfaces to support polymorphism, hiding implementation details is one of the purposes. We want to explain a bit more about **Class** and **Interface.** They have different levels of abstraction since **Class** is an abstraction of the objects, but Interface is an abstraction of the action of the objects. Implementing interfaces will save more space and memory. For example, in our program, we have an action listener that monitors all the click actions that happen on the screen. If we did not implement the View.OnClickListener interface, we will have to define every action of each button which will lead to redundant lines of code, space, and memory. When we implement the interface would make great progress towards our program. We no longer need to implement action; we can just pass in the button itself. When the user clicks the button, the switch case will help Click Event know which button the user is clicking and do the corresponding action after clicking. This makes the program more reusable.

We also tried to use more local variables rather than global variables. This will also make a huge impact on the design of the program. Local variables only keep the variable available within the scope of the local class, and this makes it easier to modify the program when you want to upgrade or change the structure of the program. This will make the program more flexible.

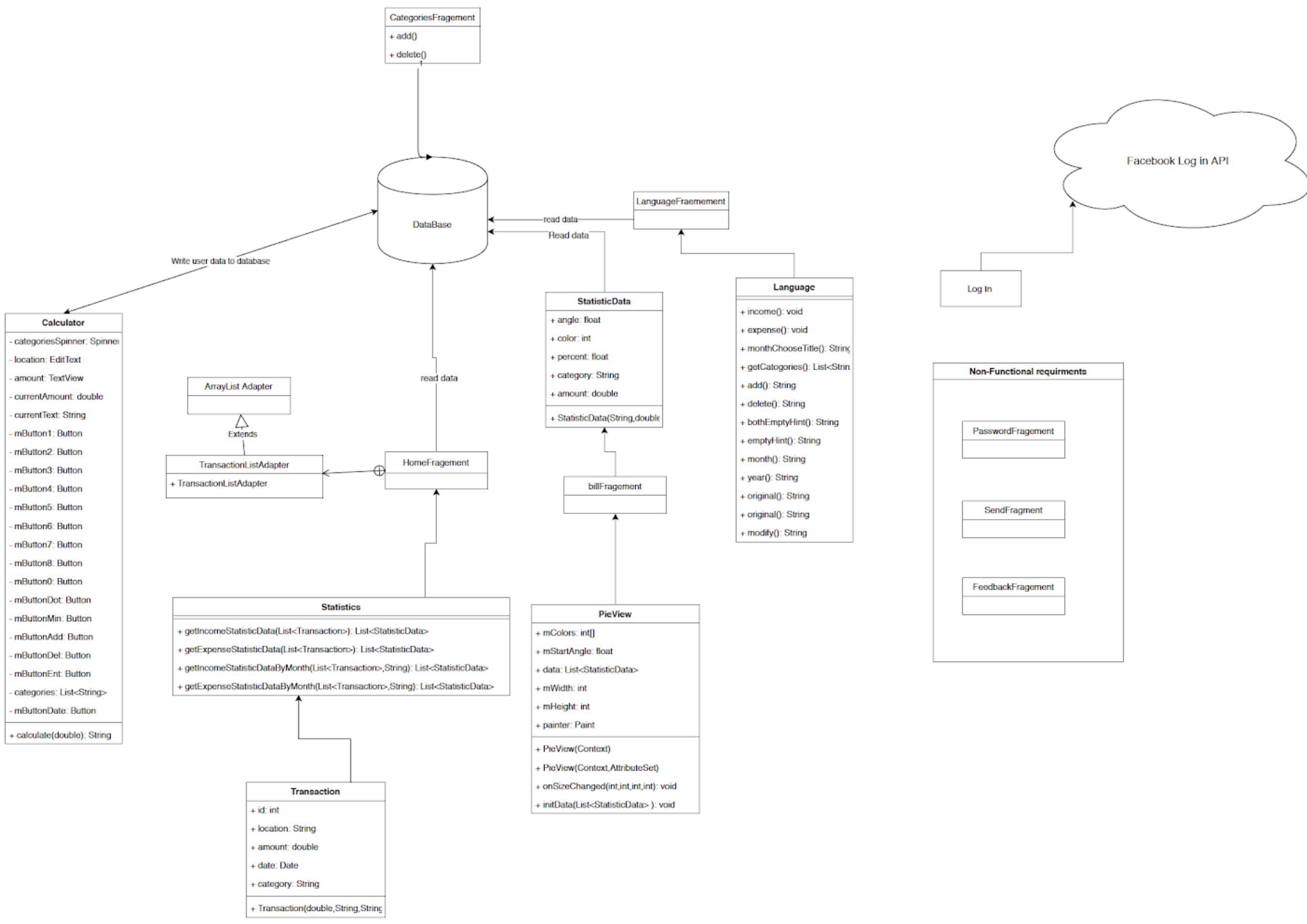
We also use more private keywords throughout the program during our design. Public keywords grant access to the class object by anyone. Everyone who wants to access the data member or member function can just access them directly without any other steps. This will make the program unsafe. This could lead to data leakage, but if we encapsulate all the data members and privatize all the class functions. If private data wants to be modified in the class, it is required to use the setter function provided by the class. If private data wants to be retrieved, it is required to use the getter function.

The model-View-Control design pattern is also widely used in our program. We completely separate our view and our program logic. The layout of our application is designed using Android Studio User Interface Tool and it is stored in an XML resource file. And all the logics of our program are operated in Java File. This separation is really useful in our app because once we found some problems in Java File, we could use the Junit test to test what goes wrong and try to figure out a solution. We do not need to display all potential error messages on the screen. Once our program logic is correct, we could display correct data and operate the functions on the screen. If there is something wrong with the user interface, wecan keep modifying XML until making it right without influencing the program logic and the database.

Meanwhile, we decided to not hardcode the logic of our program. This will make development easier in the future. Developers do not require hardcoding if there are updates on the application. For instance, there is a category class in the program and it is used to add categories to the database. We have built-in **Transportation, Food** and **Income** for our application**.** And these three items have their icon. When the user is entering his/her spending record he can choose the corresponding category. If users’ intended item’s category is not on the list, the user may also add his customized category to our database. This avoids adding all potential item’s category

Based on the facts we just stated, we felt like we tried our best to minimize coupling and accommodate changing requirements. Our application is also developer-friendly for future development. Developers do not need to hard code all the code to keep the application updated. Developers can understand the code by reading comments and continue to evolve the application based on our program

UML Diagram：



UML DESCRIPTION:

Here is our UML diagram about the whole program, some non-functional class shown in the right side of the picture. The cylinder is our database, and this diagram can show the relationship between each class. That is similar to our diagram of above architecture style.

PROJECT DISTRIBUTION:

Database Architect:

Name: Haiyang Ni

Role: Design database schema, add and remove information from database. Responsible for anything that is related to the database.

Design Pattern Designer:

Name: Zhenqing Lang

Role: Design the what kind of design pattern the application should use. This role is in charge of the general structure of the application.

User Interface Designer:

Name: Jie Ren

Role: Design the polish the layout. This role is in charge of anything that is related to User Interface

Junior Software Developer:

Name: Payton Pei

Role: Coming ideas and giving advice on the functionality of the application and implementing it. This role is typically in charge of advice and implement components of functions in the app

Date: MARCH 15 2020