COMP3004 Delivery 3

Team Name: Health Balancer

App Name: Health-Balancer

Team Member:

Jianghao Li	101072091	Peng Li	101047123
Chenxi Wang	101075058	Yiming He	101090748

Architectural Styles

1. **Object-Oriented**

Health Balancer is built based on JAVA programming language. The most based architectural style for this application is object-oriented architectural style. Because our application is built for the Android system, JAVA is one of the main languages for Android development. Therefore, when we pick JAVA to build Health Balancer, the object-oriented is one of our architectural styles.

In our project, we have to encapsulate multiple java classes for each activity, they are able to cooperate with each other or independently implement the system functionalities. And also, the objects must be needed in the object-oriented style, objects have to be instantiated first and then other classes are able to call the objects' methods to use. In Health Balancer, we instantiate an object called HelperNewEvent. There are two constructors in this class. First one is for instantiating Food Type Events, it stores all information about food type records for users. Inside of this constructor, it includes food event type (breakfast, lunch,dinner), long number for date, main food, the calories of main food, drink and the calories of drink. Second one if for instantiating Work Out Type Events, it stores all information about work out type records for users. Inside of this constructor, it includes type of events, long number for date, work out name and work out calories. And another object of Health Balancer is User object. This object is for sign up and login funtionalties. It stores uid and email address. The figure1 is showing the class diagram of Health Balancer.

functional goals:

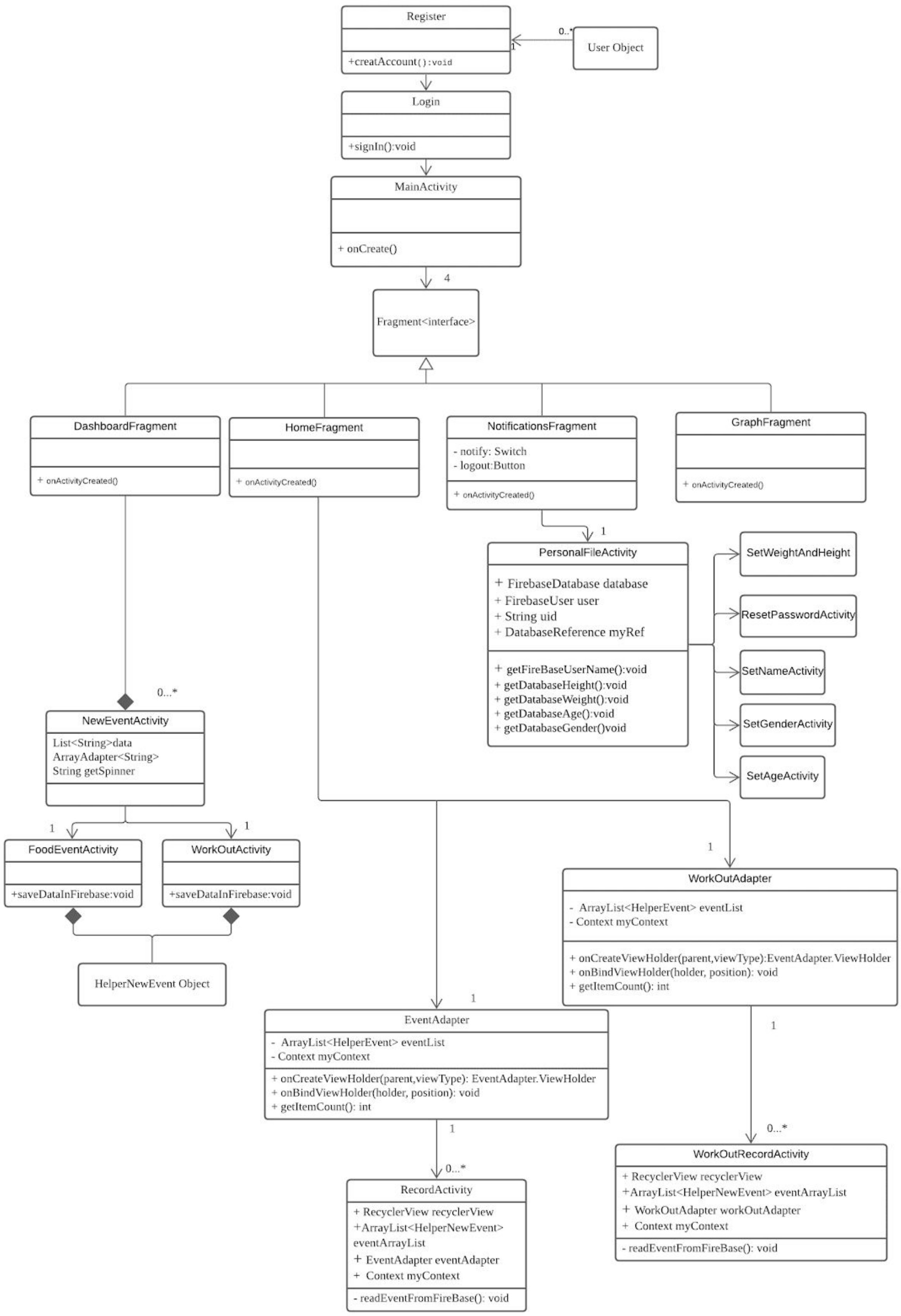
HelperNewEvent Object is used to create events, and save information to the database by calling the constructor inside of HelperNewEvent Object.

User Object is used to support sign up functionality of Health Balancer.

Non-functional goals:

Object-oriented architectural style supports non-functional properties that we mentioned on D1 are Security and Implementation. The data must be displayed only if users logged in. The source code of our program is based on JAVA and our program is launched on the Android system.

Figure1 class diagram



Client-Server Model

Client and server would be one of the best choices for our app, since our main actions have been done by the user which our clients. All those kinds of actions should be stored into a specific location and displayed to our clients, thus the server is necessary to play this role.

Client side

The role of the client is either login or create a new account on our Firebase server.

if the account is successful login or create, the client can view their existing data in the app and they can also create a new event or modify those existing events. The app should also provide a location for clients to modify their personal information, such as change the password available after they logged in.

Server side

If the client clicks login, the role of the server is to verify the account exists or not and check the password reach as the information from the Firebase database. If the client clicks sign up, the role of the server is to verify if there exists a duplicate account into our server, the account should not be created. In addition, the password should also be checked.

Server should respond to the clients that their account is either successfully created or logged in. If success, the server should display the existing information to the app, such as graphs, events or personal information. Except that, the server also has to listen to the action done by the client. Such as creating, updating or deleting. It should update that information into our database and display it on the app.

To specify the non-functional properties, client-server would be familiar for users, since it is easy to use as a normal app. Login part provides security for users and they can view their data information only if they are logged in. In addition, firebase is built-in in Android studio, therefore the loading time would not be too long.

Firebase Login and database API

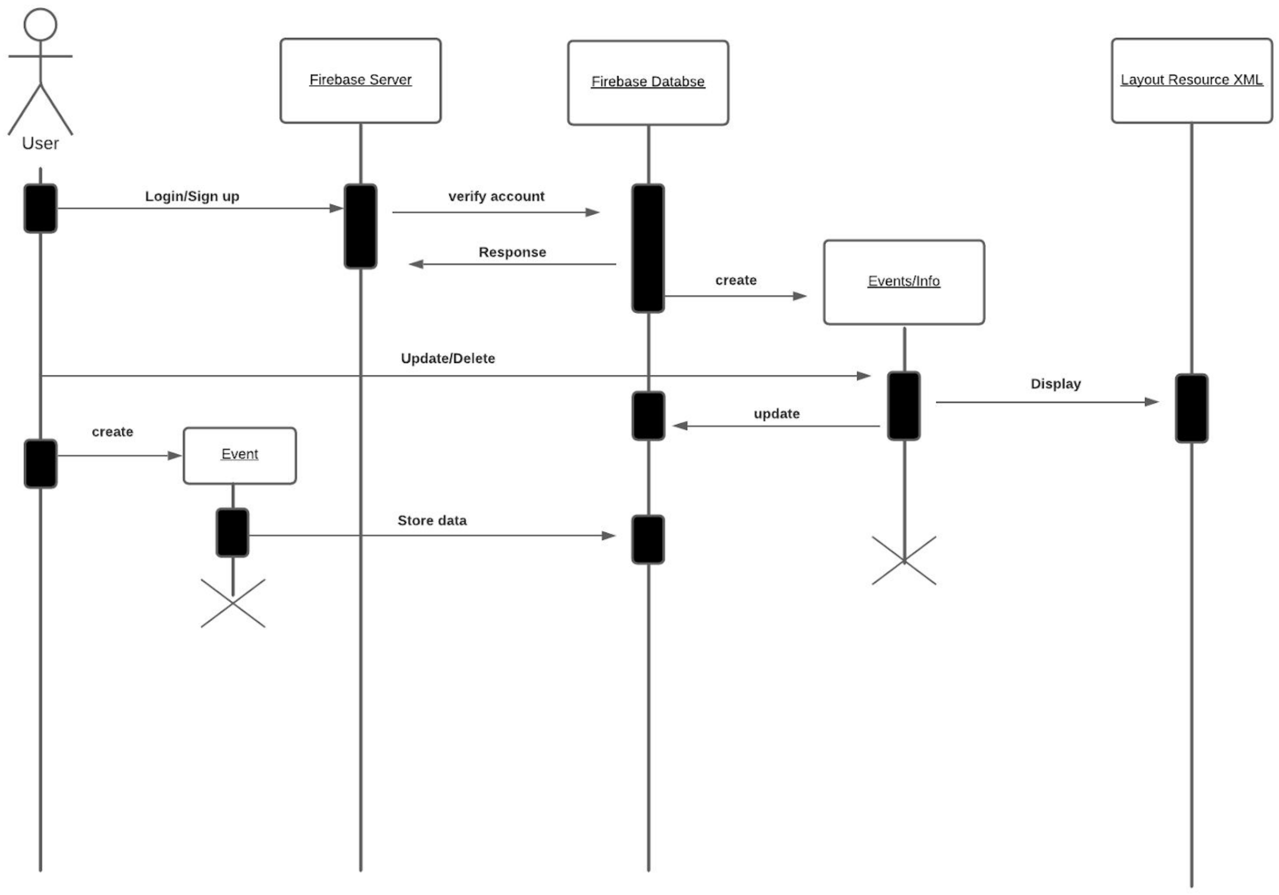


Figure 2

Design Pattern

First one design pattern of Health Balancer is the Adapter design pattern. This type of design pattern is used to show records to users. We designed two Adapters, one is called EventAdapter. This one is used for adapting food type events to RecyclerView. And the second one is called WorkOutAdapter, this one is used for adapting work out type events to RecyclerView.

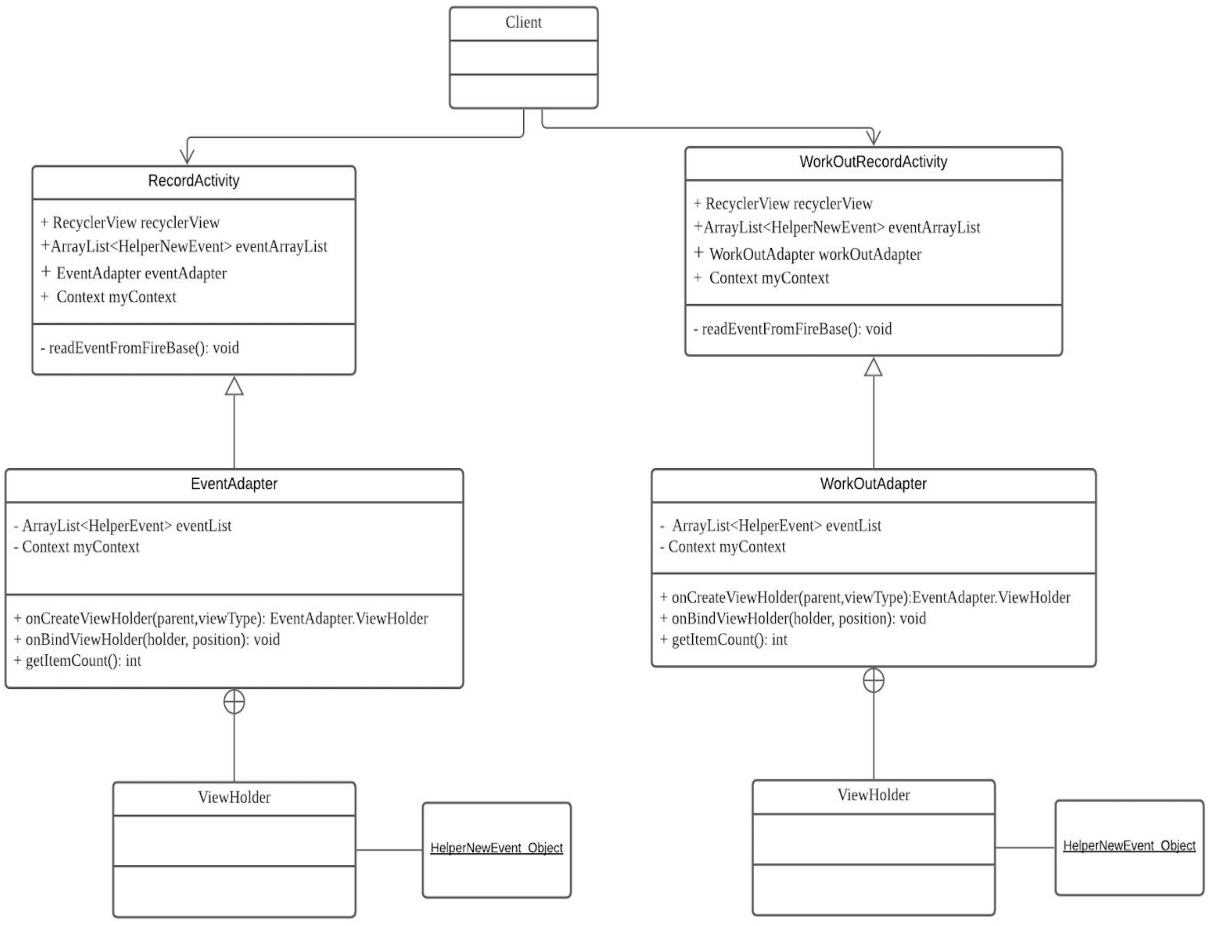
Each Adapter including a card view, the card view is an interface for saving record’s information from HelperNewEvent Object. And each Adapter extends RecyclerView.Adapter<...> class, and also each adapter has an inner class called ViewHolder extend RecyclerView.ViewHolder class. Therefore, the Adapter class and its inner class can override the methods of their parent class to bind the data to the card view to show. And those data are getted from an ArrayList that is saving HelperNewEvent Objects.

And then, in the RecordActivity or WorkOutRecordActivity class, we have to create a RecyclerView and set its adapter to be that we have created. By using the Adapter design pattern, we can convert the cardView interface into RecyclerView. The Figure 3 is showing the structure of the Adapter design pattern in our program.

The adapter design pattern is used for letting those classes that cannot work together because of incompatible interfaces can work together. In our Adapters, we bind the data from HelperEvent Object to card view, and set Adapter in RecyclerView in another class to show records. The reason that we have to use the adapter design pattern is that cardview with data can not be set in RecyclerView directly. We have to use the adapter as a brigade to connect cardview and RecyclerView.

Our Adapter design pattern is Single Responsibility Principle, it can make data conversion code separated from the main business logic of the program. If changing requirements is required, for example, the program has to show other types of records, we have to write another adapter to support that new type. Because in our Adapter design pattern, each adapter only converts a single user interface.

Figure 3



Client
RecordActivity
+ RecyclerView recyclerView
+ArrayList<HelperNewEvent> eventArrayList
+ EventAdapter eventAdapter
+ Context myContext
- readEventFromFireBase(): void
WorkOutRecordActivity
+ RecyclerView recyclerView
+ArrayList<HelperNewEvent> eventArrayList
+ WorkOutAdapter workOutAdapter
+ Context myContext
- readEventFromFireBase(): void
EventAdapter
- ArrayList<HelperEvent> eventList
- Context myContext
+ onCreateViewHolder(parent,viewType): EventAdapter.ViewHolder
+ onBindViewHolder(holder, position): void
+ getItemCount(): int
WorkOutAdapter
- ArrayList<HelperEvent> eventList
- Context myContext
+ onCreateViewHolder(parent,viewType):EventAdapter.ViewHolder
+ onBindViewHolder(holder, position): void
+ getItemCount(): int
ViewHolder
HelperNewEvent_Object
ViewHolder
HelperNewEvent_Object

Composite Pattern

We used a composite pattern to treat a group of similar objects as a single object. Composite patterns combine objects according to a tree structure to represent parts as well as the overall hierarchy.

In the pattern, all of the events would be our tree and each event would be our leaf. Thus, we combine all the events as a single object and those events save as a list type in the object. The Composite pattern can load the data from the database and create each event by call event class to separate them into different groups.

In other words, our project composite pattern would create 4 different single objects to store events, those are breakfast, lunch, dinner and workout. First of all, it would read the database. Secondly, it would check whether the class is created or not, if it does not have that class then create a new one, if there is an existing class then add it into the list of that class. After all the events saved into those 4 single objects, it would print the details in the terminal.

The reason that we used this design is that high-level module calls are simple, every time that we add a new event is free, we do not have to consider adding to a specific location. However, when using a composite pattern, the declarations of its leaves and branches are implementation classes. Since our data is read from the database, every time when we open the app, the HelperNewEvent class would be created for the system and when we close the app the class will be destroyed. We don't have to consider the issue of the inversion principle. Expect that, using a list of events to store the in the HelperNewEvent is decoupling the class.

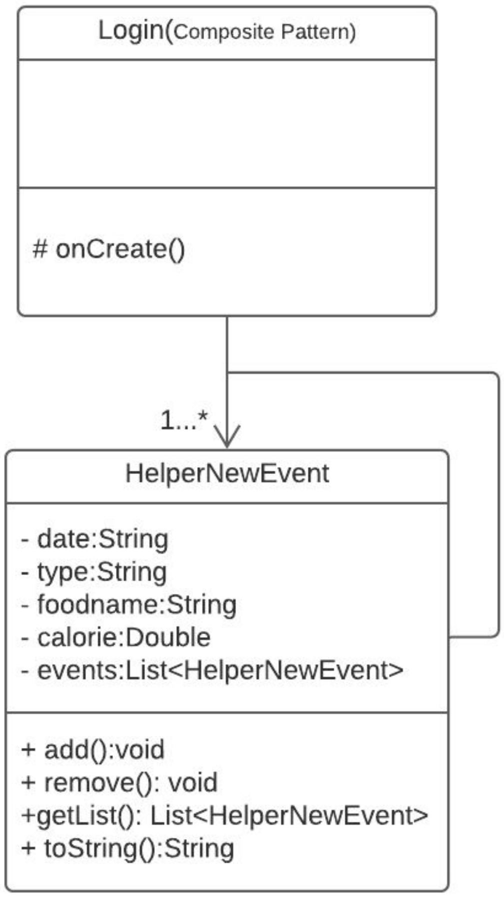

Login(Composite Pattern)
# onCreate()
1...*
HelperNewEvent
- date:String
- type:String
- foodname:String
- calorie:Double
- events:List<HelperNewEvent>
+ add():void
+ remove(): void
+getList(): List<HelperNewEvent>
+ toString():String

Figure 4

Mapping team members to architectural-level

contribution

Peng Li：Client-Server style design and Composite pattern, helping teammates design the UI layout and build the database. Implement daily notification for the app. Design the graph part by using the library of achartengine.

Jianghao Li:

Document part: object-oriented architectural style and Adapter design pattern.

Coding part: 1) Sign up and Login functionality. 2) Show records to users in Home 3) Personal file functionality 4) Participate in database building and create events functionality

Yiming He: Participate in document writing,object-oriented architectural style. Focus on testing and maintenance (Debug and fixing). Helping teammates design the UI layout.

ChengXi Wang: Participate in Composite Patterns. Participate in designing the UI layout and testing and maintenance.