Design Document

Updated specification:

As per the feedback given in phase 0, we have limited our scope to just to the primary features of a fitness tracker while excluding the social media element. We had not anticipated the complexity that comes from integrating Android, thus, we believe this is the right decision.

Description of major design decision:

Designing has been a huge aspect to consider while making our app, and we have made some major decisions when it comes to proper style and implementing the correct patterns. Through research and planning, we decided to go with the MVP – Model, view, and presenter – which is an architectural pattern and a derivation of the MVC – model, view and controller -. We used MVP for building our user interfaces and for packaging, as it was important for our app to have a good architecture. Moreover, we used MVP pattern to adhere to clean architecture and separating the views, models and using a presenter to communicate between them.

Apart from that, another major design decision we took was implementing the database by a Gateway class called ProfileReadWriter which implements a ReadWriter interface so that the LoginUseCase and SignUpUseCase can depend on it due to dependency inversion principle. Moreover, we also implemented LoginPresenter and SignUpPresenter to pass the user input from the view to the use case, which then passed to the gateway and went into the database. Through the five solid principles, dependency inversion principle seemed to be the most ideal for implementing the database because we wanted to decouple our system, such that we can change individual pieces instead of the whole thing.

How project is consistent with SOLID design :

Let's discuss this by each SOLID principle in order:

Single Responsibility (S)

Looking at our organized project directory, it is easy to identify that we have well abided by this principle. Each and every class has a **single responsibility**, signified by the fact that it has only **one reason to change.** We can take a look at all our entity classes, where we have multiple classes concerning exercises. These classes are distinct by their unique role, and justifies this principle as we avoided putting all those exercises in a single class. We can look at our Presenter package, where every presenter has only one reason to change, and that is, if we require additional functionalities in the corresponding view. The only class that may be violating this principle is the FollowManager. We can further break it down into FollowingManager and FollowerManager.

Open/Closed (O)

The **inheritance** involved in many instances, especially the entity classes, strongly implies our utilization of this principle. The Exercise classes for instance, have many subclasses to prevent modification of existing code while allowing freedom in extension. Let's say for instance we have a new type of exercise other than Timed or Rep, we can add a new subclass and inherit any methods from the superclass. This allows us to **extend** our code **without modifying** any of the existing code. We would be willing to keep pursuing this principle when adding new features.

Liskov's Substitution (L)

We have many instances of inheritance involved like the exercise classes, and we have kept our compliance with the open/closed principle throughout. Since we only have to **add subclasses to add behaviours**, this does justify our adherence to this principle. A great example of this is with our Exercise family of classes. Where we have an abstract class, Exercise, and it's children, WeightedExercise, TimedExercise and so on. In this case, we can store all of these subchildren into one Array of Exercise. When processing the elements of the array, we can work with all of the types together due to our adherence to Liskov's substitution.

Interface Segregation(I)

We have adhered to the Interface Segregation Principle. The usage of interfaces increased as we have implemented MVP and has made it important to use interfaces so that we can adhere to dependency inversion. In our Presenters, we have defined an interface. This interface, unique to each presenter, is implemented by the presenter's corresponding view (Activity). We made sure to only define the necessary methods inside the interface. The necessity is determined if the method is actually used in the corresponding view. Thus, adhering to the Interface Segregation Principle.

Dependency Inversion (D)

Dependency Inversion is a crucial element of MVP in order to adhere to the clean architecture. For MVP to be functional, we needed a reference to the Activity/view in our presenter. But this violates clean architecture, as the presenter is in the third layer and the view is in the fourth layer. So we use Dependency Inversion. We let every view implement an interface defined in the presenter. And we store a reference to that interface in our Presenter. This way, it is clear that the presenter is storing an interface and calling its methods instead of directly calling the view and its methods.

Description of which packaging strategies we considered, which we used and why:

The packaging style we went with was packaging by layer, where you package every class by its place in clean architecture. The main reason for this was because this is ultimately a clean architecture software design course and sorting our classes by the clean architecture layers provides many benefits that outweigh the other packaging styles we discussed. By layer makes it not only easy for TA's to understand the clean architecture and layers of our code but also helps us in confirming we are following clean architecture and no class is doing a job it's not supposed to. Not only that but a lot of times people were working on one layer of clean architecture at a time, so not needing to jump between lots of packages to code was helpful. Not to mention that this was a lot better than the other packaging methods discussed. A big contender was package by feature which looks to sort entire features or tasks your program can do together. For us that would be sorting all the classes that focused on profile and sorting all the classes that focused on workouts together. Our program is very focused and doesn't have many small features but two very large ones, so organizing that way would've been pretty useless to just have two very large folders. Inside/Outside was discussed also as it groups together classes by order of how close they are to a user, so controllers would be grouped, then backend stuff, then the database related stuff. This works well and doesnt have too many issues, however a lot of us were working on layers of clean architecture at a time, so some packages would include a lot of extra classes people just aren't working on. My component was not really considered as just shoving all the front and back end into their own folders would be very messy for our backend which had lots of classes that were not directly working with each other.

Summary of design patterns we implemented (or plan to implement):

Mvc

- We used the model view controller pattern for the UI layer of the app. Simply put, it's used to view the information from the model, which will get updated from the controllers that we've implemented. The reason this was used was because it allowed us to simultaneously work on the project as a whole. Mainly, we used the mvc design pattern for the ProfileController and profile activities because it allows us to easily implement observer and observable design patterns. An example of the use of this pattern can be seen with the follow counter, this is a counter that gets updated each time that it changes because it checks to see if a profile follows another one and updates the view from that.

Mvp

- Through using the mvc pattern we went along with the mvp pattern as its based on the same concepts. Additionally, this pattern allows us to dictate how to structure the view, which is where it's use comes into play. Specifically, for the user interface and for packaging mvp was used and it helped us separate our programs functionalities in a simple way that we were able to understand and agree on. For example, with profilepresenter and profileactivity, the user goes to view and searches for a profile, profilepresenter will look at the model and communicate and update profileactivity to show the user what they are searching.

Facade design pattern

- We implemented the facade design pattern with the exercise class as it seamlessly collects data and is able to present the workout information easily. Additionally, there are no dependencies in regards to the exercise subclasses and allows the API to be easy to use, enabling flexible changes with all the functionalities that comes with the exercise class. This is done with a single class (exercise), which handles all the complexity of its subclasses.

Open questions:

For phase 1, our group has gotten together and worked towards the functionality of our app and implementing the design patterns. We have managed to get the basics working, and have tried to showcase all the design principles. However, it has been quite tricky and we have all struggled towards the making of the app (mostly the databases and packaging) and hence have a few questions that we would like to be covered.

1) For our app, we are using MCP so that we have a (model, view and presenter) folders. However, the model folder also contained all the entities, use cases and the gateways which did not look good so we decided to make subfolders for the same. While doing that, we wondered if we should have just made 5 folders, Entity, Use Case, Gateway, Presenter, and View which would have looked neater, but then we wouldn't have been able to see the MVP structure (model, view and presenter). So, our question is that **is it better to have less outer folders and more subfolders, or just have a bunch of folders and split classes within those?**

What has worked well so far:

In the last report, we identified that we needed work on UI and make sure to abide by clean architecture. Mainly, we now have a frontend and backend component. In the UI aspect, we also added image files to use for our interface. At that time we made the foundation of our program with the necessary classes and we have further expanded in code design by following the MVC and MVP patterns. We have sections for the model, view and presenters and we also sectioned our files by clean architecture layers like entity classes and use cases.

Other certain design patterns have been addressed as well, like observer and builder. We fulfilled everything that was highlighted in the previous progress report in terms of design overall. We have an xml-fueled UI environment to start off with, which we will be expanding more on soon. We also introduced files for our app to store overall profile data (fitappfiles). It is clear that the visual foundation of our app has been covered, now we need to make arrangements for a proper database(as of now we have set up a Gateway class and Firebase) and constantly update the UI when necessary. We will brainstorm what else we can do to execute these aspects for the next phase.

All in all, we dove deeper into the foundation of our program design and addressed it in a more detailed manner than before as desired.

Test Cases:

We added a large amount of test cases to pretty much all of our backend code. This was done to test to see if everything was working from phase 0 and make sure any issues we had with any front end code was because of the front end code

Refactoring:

Github was a pivotal tool in working together. We made full use of pull requests and discussed with at least one other group member before bushing the pull request to main to avoid any unwanted errors in the main code. It also helped to see if there were any merge conflicts however our communication prevented a lot of this. We made use of bringing up issues in github with that tab and resolving them among group members. Issues such as the name of certain classes led to us refactoring and submitting new refactored code to github resolving the issue. bcb77b63c59c2427402bc35e35fac9ebe36d1856 This specific git hash refers to a pull request in which lots of refactored code was implemented. It refers to refracting a lot of our UI to fit with clean architecture and solid principles by the

MVP design pattern as explained above. Other design patterns were added with slight refactoring to code such as observer observable with the MVC design pattern.

Functionality:

We demoed most of the functionality of our code, showing the general idea and giving a very basic run down showing how one might use the fitness side of the app to list out different workouts, but also the social media aspect with profiles and follows. It functions well without bugs but needs to interact more with databases allowing the search function to pull users from the database and store followers to the database. We have been perfectly following our specification, but have decided to scale back a little of the social media aspects like posting posts as the TA rightfully pointed out would be too much extra work. Beyond all these load states have been created with a database allowing to store a user's profile to a database and pull it later if they want to log in.

Abdullah: Implementing the MVP Architecture, ViewRoutines, AddRoutine, ViewRoutine, AddWorkout, ViewWorkout, AddExericse (Backend)
Souren: Implemented MVP, implemented MVC, ProfilePresenter, LoginPresenter, Profile Activity, MainActivity, SignUpActivity, SignUpPresenter, ModelProfile, User, Profile, Profiles, main_profile.xml, create_account.xml, activity_main.xml, Observer design pattern, Design doc (packaging strategy question, functionality, refactoring, test cases)
Uthman: Implemented Firebase database and put it into clean architecture by creating Presenter, Use Case, and Gateway classes. Refactored some code.
Sana: Ui for the following pages: activity_add_exercile.xml, activity_create_workout.xml, activity_main.xml (edited and fixed), and activity_track_workout.xml. Made test cases for the following: ExerciseTest, RepExerciseTest, TimedExerciseTest, UserTest, and WeightedRepExerciseTest. Design doc (open questions, and major design decisions)
Munim: Test cases for ProfileTest, RoutineTest, WeeklyScheduleTest, WorkoutTest, WorkoutTrackerTest, builder design pattern, design document (SOLID question), progress report(what has worked well with design), and documentation
Victor: Made Repexercise, timedexercise, weightedrepexercise, design document (patterns implemented), storing data

Plans for next time:

For the next phase, we are planning on just completing what we have started and making everything work perfectly as intended. To do so, everyone will continue to work on what they have been working on since phase 1.