Design Document

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. We did also use a little bit of MVC to implement the observer observable design pattern.

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** (obviously unless its fixing a bug)**.** 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 have many classes signifying our profound implementation of this principle, however there is

a noticeable inconsistency. Our fitapp files could use a lot more classes instead of having huge, packed ones, being more prone to have more reasons to change and thus more responsibilities. The followManager class concerns both following and followers, which we can consider breaking down to avoid any violations of this principle. We can address this to attain neater single classes and a more organized and uniform environment of our program.

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. However, it is clear that replacing the objects of Exercise with its subclass' would likely change the behaviour of the code. This is because the Exercise class itself has **more behaviours** in the forms of methods than it's child classes (similar to the square-rectangle example explored in class). This violates Liskov's substitution and a way to address this violation is to rework this model into interfaces.

Interface Segregation(I)

Since we don't have that many interfaces and the ones we do are used for distinct components. We haven't really focused on this principle for our project. We created the appropriate interfaces for certain classes to implement. There is only a **one-to-one relationship** between interfaces and classes in our project. For example:

This shows a brief execution of interface segregation. Two different interfaces, where one involves a username and the other doesn't for its respective subclass to use in order to work. However, this principle hasn't been of much concern throughout this project. Whenever we do get the chance of making more interfaces, we will ensure to enforce this principle.

Dependency Inversion (D)

Again, not many interfaces were used in our project but wherever we had the chance, we ensured to follow this principle. One example can be the following:

There is an arraylist of Exercise objects, communicating through Exercise as an abstraction, so we could switch among which subclass(es) the arraylist is referring to. Clearly the dependencies are coupled and communicate through a mutual abstraction point. This is a brief example of our

adherence to this principle as we haven't had many interfaces involved. A more prominent example is having this principle addressed in our database implementation. For instance, we have a ReadWriter interface that LoginUseCase and SignUpUseCase **depend** on. This principle was vital when constructing our database as the database itself is a large part so decoupling this aspect allows more freedom of changes without compromising the entire thing.

Overall, we've had many counts of inheritance involved (superclass and subclass) in our OO design to explore these SOLID principles and as we have dived deeper there were a few anomalies we could address. We haven't had an adequate amount of interfaces which questions our overall compliance with certain SOLID principles like interface segregation or dependency inversion. We will try to add more interfaces as it's generally even needed to avoid violation of Liskov's substitution and just to overall emphasize our understanding and enforcement of these principles.

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. By 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 pattern. 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.

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 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.

Summary of what each group member did:

Abdullah: Implementing the MVP Architecture, ViewRoutines, AddRoutine, ViewRoutine, AddWorkout, ViewWorkout, AddExericse (Backend)

Souren: Implemented MVP, implemented MVC, ProfilePresenter, LoginPresenter, Profile Activity, ModelProfile, User, Profile, Profiles, main_profile.xml, create_account.xml, activity_main.xml, Observer design pattern, Design doc (packaging strategy question)

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