Design Document

Updated specification:

The app is for users who are seeking additional help with enhancing their wellness by allowing us to track their healthy lifestyle and workout routines. The users will be able to follow training regimens and keep a note of their everyday accomplishments. Furthermore, the app will come with numerous types of exercises including (but not limited to) weighted, and rep based exercises.

Description of major design decision:

For our major design decisions, since we are making an android app, we had to decide how we were going to organize our workflow between the android activities and our third layer of clean architecture. We started off with MVC (model view controller), but realized that it doesn't adhere to clean architecture because the models can communicate with the views in this pattern. So we ended up going with MVP (Model view presenter) so that there are clear boundaries between our layers. This meant that we had to have interfaces for each presenter which allowed them to communicate to the activities through dependency inversion. Here on the screen you can see an example of an interface we used which our Activity class implemented and our presenter class used to communicate to it. All of our presenters had an interface similar to this one.

Another big design decision we made was implementing Firebase as our database into our system. This meant that we needed gateway classes that dealt with the Firebase details so that they could be separate from the rest of our program. We had a Loadable and Saveable interface (which are shown to the left of our slide) which was implemented by any gateway class that was loading or saving from the database, and we had our presenters interact with the interfaces to deal with the gateway classes. This meant that we had clear boundaries between each of our layers to follow clean architecture, as well as the SOLID principles. On the right you can see an example of a method in one of our presenters, which deals with the gateway call. As you can see it's very simple and independent of implementation, which makes it really easy to change implementation or specific details.

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.

Open/Closed (O)

We heavily followed the Open/Closed principle wherever we could as we realized the importance of the app's need to be easily extendible. Since this is a fitness tracker app, the most important example is the way in which we have designed the concept of "sets" in our app. An exercise is differentiated by the type of sets it uses. We currently have two sets, RepSet and WeightSet, both of whom implement the Set interface. Let's say for instance we have a type of set other than Weighted or Rep, we can add a new class and implement Set. This new class will then be supported by the Exercise classes. We would only need to add a method or two per layer to have this new type of set fully functional. This allows us to **extend** our code **without modifying** any of the existing code. Other than this example, we emphasized inheritance in many instances, especially the entity classes to utilize this principle.

Liskov's Substitution (L)

Liskov's Substitution Principle is an important principle when it comes to abstraction, which then further enables our app to be easily extendable. We adhered to this principle and kept it in mind for the core components of our software. An example of this is the Set interface which enables our application to have multiple different types of classes which implement the set and those classes will work with our existing code due to the abstraction the Set interface provides. Similarly, we have ensured to use abstract implementations, such as List, as an abstraction for our arrays. Allowing us to switch to any concrete implementation of List very easily. Overall, we have used abstract implementations wherever possible such that objects are easo;y replaceable with their subtypes without affection program correctness.

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:

In phase 1 it was initially brought up that our packaging style we decided to use was by layer. The reasoning at the time was because this is ultimately a clean architecture software design course and sorting our classes by the clean architecture layers. By layer made 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. However further down into phase 2 certain things changed that made packaging by component more appealing. Packaging by component looks at putting packaging things how they group in the backend. Initially many of us would just pick something in the program to work on, meaning we were working on entire layers rather than just sections of uniform code. Although we got the work done, it proved to be a bit disorganized.

We decided to give tasks to specific people to focus on, looking at specific parts of the code like profiles, workouts, authentication, etc. This meant that our packaging by layer was inefficient, as someone working on the Profile part of our program had to go through multiple packages to get the code they needed. By component meant everyone had every class they were assigned to work in in the same folder, as now focusing on one specific part like profiles required certain entity, use case, presenter, and other classes that were now grouped together. Not to mention this meant we could start package protecting parts of our program, allowing us to not have to make some classes public as they were in the same package as other classes they needed. This makes our code a lot better since everything is more private and cannot be accessed where it is not needed. We tried to follow the component principles highlighted in the clean architecture text as much as possible as well. In the end, packaging by component was the best as it made sure everyone had the exact classes they were assigned to without having unnecessary classes packaged or not the correct classes packaged together. It also allowed us to have certain classes that were not interacting separately from each other so we could package protect the program.

Similarly the other packages like by feature were not considered for the same reasons in phase 1 where by feature looks to sort entire features or tasks your program can do from a front end perspective. For us that would be sorting all the classes that focused on profile and sorting all the classes that focused on workouts together since these are the two main features of our program from a user's perspective. 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 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. It's not the worst however it would mean unnecessary jumps between packages with stuff like use cases and entity classes as they would be farther from the user and would make coding a little less organized. By component proved to be the best as it now organized our code allowing people working specifically on large sections of connected code to have all the classes they exactly needed, without any extra classes they didn't need.

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

Factory Design Pattern

- We implemented the factory design pattern in a couple of places to instantiate the correct objects. This design pattern is important in maintaining the extendability of the Set and Exercise element of the app. For example, when adding a set to the exercise, we must ensure that is the correct information. For example, we don't want to send just the rep count to a weighted rep set, as it needs a weight and a rep count. The class SetFactory solves this issue by deciding the correct Set to create. We also adhered strongly to open closed principle in the SetFactory, so adding support for a new set is as simple as adding a new method. Another subtle use of this design pattern is in the ExerciseTemplate class, which instantiates the exercise object with the correct type of set.

Model View Presenter Pattern

- Through using the mvp 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.

Iterator Design Pattern

- We used the iterator design pattern to make the Routine class iterable. This is useful because a routine is a collection of workouts, so iterating over an individual routine means iterating through the workouts in that routine. We want to hide our code implementation as much as possible, so with the iterator design pattern we wouldn't need to call a method to get the workouts from the routine, we can use a for loop and iterate directly on a routine.

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.

Testing:

We used a Test Lab tool within Firebase called Robo Test, in which a program runs an emulator with our app and goes through all the possible options to test if there are any failures and if everything works well. There is also an option where you can provide credentials so that the program can login to the app as well, which is a good test to see if our authentication is working as intended. This was very useful to run every once in a while after making a big change to ensure nothing crashes.

On top of this, we added a large amount of unit tests to our entity and use case classes using JUnit. This was helpful because these were tests we could run after every change to our first or second layer to ensure everything was still working. They also help us test specific classes, unlike the Robo Test, which just tested the frontend and UI specific things.

Refactoring:
Phase 2 of our project required a lot of refactoring. Our main focus in phase 2 was satisfying our specification by completing our project, as well as refactoring everything we did in phase 1. The biggest things we changed were the way our Profile, Workout, and Exercise class worked.

Our Exercise and Workout class were heavily violating the SRP since they were responsible for two things. They were responsible for storing workout and exercise templates for the user, while at the same time, they acted as use cases to start and perform a workout. To solve this issue, we separated the classes into **ExerciseTemplate, WorkoutTemplate, PerformExercise, and PerformWorkout** (pull request #63). This new design adhered better to SRP and was more functional than our old design. Furthermore, the exercise class had a major flaw as it contained the fields for reps and sets. The best way to illustrate the problem with this design is with an example: Suppose you have an Exercise with the name X. X contains two fields, numReps and numSets. That is, after performing an exercise, you can set numReps to 10 and numSets to 3. This implies that two sets of ten reps were performed. But it is rarely the case that a user will perform ten sets of exactly 2 sets. We wanted to enable our user to perform a set of 10 reps, and then a set of 8 reps and so on. This led to the creation of the **Set** interface and it's implementers; **RepSet and WeightSet**. The performExercise would hold a list of

Set(s). Thus, enabling the user to have the functionality required above. Another added benefit of that is the strong adherence to extendability of the software. To add a different type of exercise, we only need to add another implementer of set.
We also needed to change how our Profile class was structured. It was responsible for too much in our phase 1 program and was violating a lot of rules in SOLID and clean architecture. We removed the use case classes that were stored in it since Profile is just an entity so it cannot store use cases, then we combined it with the User class since they were essentially responsible for the same things. We then removed the dependency on Profile from our fourth and third layer as much as we could. This made our program a lot better because if we need to change anything in the Profile class now, we won't have to adjust a lot of code in our third and fourth layer since they aren't heavily dependent on Profile. This adheres to the OCP and SRP a lot better now. Here is pull request #76, where we changed the Profile class and dependencies on it.

Finally, the last refactor was in the Profile class. Previously, our android app operated by passing around instances of Profile all over the app. Which required us to store workouts, exercises, and pretty much any object we needed access to inside of the profile. However, this heavily violated SRP and clean architecture. Furthermore, we did not want our domain layer to conform to the third and fourth layer. Rather, we want the fourth and third layer to conform to the lower two layers. The reason we had a design like that in the first place is because we were storing the profile object to the database; thus, we wanted everything inside the profile. However, we refactored our database and gateways to store specific objects. For example, we now store Routines, Workouts, PerformedWorkouts and Profile separately. This enabled us to remove the dependency on Profile on many of our classes. While simultaneously adhering to SRP. And lastly, it enabled our gateways to instantaneously load and save only the pieces of information that are needed, rather than storing and loading everything at once (pull request 74).

Use of Github features

Github was a pivotal tool in working together. We heavily utilized pull requests and code reviews. Our group set a rule that at least two other developers will review the pull request before it is to be merged. The careful reviews improved the quality of code that was merged on to the main branch. It also helped us prevent merge conflicts, although our communication played a large role in that as well. We also utilized the issues feature of Github by creating ongoing issues and solving them as needed. Another feature we utilized was Github Actions, a failed check from Github Actions meant that one of the unit tests failed. This led to immediate feedback. Lastly, we also utilized GitHub Projects for task management.

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.

Project Accessibility Report

Universal Design Principles:

The Principles of Universal Design are extremely important to adhere to for any program, be it an app, website, or even a newsletter. It is used to guide the design of the environment, products and communication. From all 7 principles, we successfully implemented the ones we felt are necessary for our app. We made sure that the design and composition is very easy to understand, used to its utmost extent and is visually pleasing. We have used the following principles that made sure we meet our users needs and wants:

Flexibility in Use: The flexibility in use design helps the user to have multiple individual preferences and abilities. We were able to implement this Universal Design in various ways. Firstly, we have made sure that the visual elements on the screen are positioned correctly, indicating good design. The size and shapes of our buttons and texts are easy to read and are provided with labels for each. We have tried to make the level of reading comprehension easy and standard, resulting in each box for a text input to have instructions and labels. One thing we can add on in the future is, making our app run on different platforms such as iOS.

Simple and Intuitive Use: This universal design was implemented in our app to make sure that regardless of the user's prior knowledge, our fitness app is easily used and understood. We have made the format very easy to understand and eliminated all the unnecessary pages which caused complexity. Just like other fitness apps, we have made sure that we stay consistent with the layouts and user expectations. Moreover, we also have pop up messages that showcase if the user enters a wrong password or email, so that it is easy for them to understand the problem.

Perceptible Information: We adhered to the Perceptible information in our program. We made sure that there is adequate contrast between all the essential information, such as the 'back' and 'save' button are in different colours to show differentiation to the user. This way, it is easy to give our users instructions and ability to understand what's going on at first look.

Low Physical Effort: We have used this universal design in our app because we wanted to make sure that our app is very efficient for the users, resulting in no extra work. We have already put default exercises for our users to choose from, so that they don't have to manually input anything unless they want to create a unique exercise of their own.

Size and Space for Approach and Use: This is an important universal design that we made sure to implement. We made sure that we are using detailed graphics for our users to see the important visual information. Moreover, we also used high contrast colours to make the UI look visually pleasing and easy to see when viewing from different angles/places. Moreover, we even made sure that we are not using the colours 'blue', 'green' and 'red' in consideration of colour blind people. This makes our app user friendly for everyone.

A Universal Design we did not need:

Tolerance for Error: This design principle is not appropriate to implement in our program because it does not include any possible hazards or accidental unintended actions

A Universal Design we hope to implement in the near future:

Equitable use: We were unable to implement the equitable use universal design but we will make sure to adhere to the features in the coming future. Our goal is to allow our users to customize the appearance of the app according to their own preference. Right now we just have dark mode, but we will be implementing a light mode as well so our users can customize according to their wish. Secondly, once we upload the app for our users in the future, we will have it free or price it for minimum value such that anyone can afford it and use it. Lastly, we would love to implement speaking into the microphone feature, which helps the user to input information with no effort.

Target Market:

First off, our program is an app, so the largest category this program is marketed towards would be people with android phones. However, this is a very large target audience and it can be made more specific. Our app is fitness, meant to not only help you with your workouts but make it more fun too by tracking your routines. So, a clear customer base would be people exercising. This isn't exclusive to just people going to gyms, but also people working out at home, running, or doing certain other physical activities. Not only that but the apps functions allow for long term use hopefully keeping this market getting these people to share the app with friends. The apps tracking abilities are not only usable by extreme athletes but also beginners that want some structure for their workouts.

Who is unlikely to use the app:

As explained before a broad category that won't really use our program is people who don’t have phones since it's an app. This could include people that are economically disadvantaged and can’t afford phones, or people who are disabled in a way that makes using phones incredibly difficult (like people who are fully paralyzed). However some people with disabilities like blindness may find difficulty using the app, yet, it's not impossible for them to use it with the use of third-party devices on the phone that may announce every action. Also, since this is a fitness app, physically disabled people who might not work out because of their disability (people who are paralyzed, or maybe missing limbs for example) may not see a point in using the app.

Contributions:

Souren: Implemented classes to fit with our specification and improve user experience like SettingsActivity, SettingsPresenter, SetupActivity, SetupPresenter. This involved implementing more MVP design patterns. Added large amounts of class javadoc to almost every file. Refactored code when signing up or on your profile to display limitations or mistakes in the users actions when inputting unusable information. Refactored code to allow saving and movement of information between any page that updated or changed a user's profile.

Souren's Significant pull requests:

Pull Request #17
Added one of the first uses of design patterns in our program after learning them, being MVC and Observer Observable. This would lead to many discussions on design patterns and would serve as a blueprint for all the implementations of MVP we now use in so many classes.
Pull Request #27

Another implementation of a very important design pattern being MVP for the Profile class. The profile class is almost a hub to a lot of our program so adding MVP fully overhauled it and made it follow clean architecture and solid, allowing other classes to interact with it correctly.

Sana:
Ever since Phase 1, my group and I have been nonstop working on this project and on every step we have tried to improve it in various ways. I have been working on the frontend of our project and implementing all the design and visual decisions. I have made the UI for all the pages, except 1-2 of the pages which were implemented by someone else. I have made sure that regardless of who made the page, I have edited and made it look appealing to the users. I have tried to implement as many Universal Design principles as I could, including choosing the proper colours (even made sure we aren't using colours such as red, green and blue for colour blind people). I have tried my best to make our app look very pleasing and I have designed it to satisfy the wants and needs of the users. I put one of my pull requests that showcases everything I have mentioned above. Made + added the logo, fixed UI for all pages, and changed the main page

Abdullah:
In phase 2, I decided to introduce effective project management tools into our project. This included very specific discord channels, weekly deadlines and progress updates, and the integration of Github Project. Furthermore, I refactored a major part of the app which involved the core classes, Workout and Exercise by making them more functional and splitting them into four classes, ExerciseTemplate, WorkoutTemplate, PerformExercise, PerformWorkout (pull request #63). The classes now conform better to SOLID and clean architecture. Furthermore, I built the Workout Tracker page which supports different types of exercises (pull request #71). And, lastly, I refactored the overall app design to remove the dependency on the Profile class; as the entire app used Profile as a central hub to get other objects (pull request 74). Note that this last refactor was a combined effort of Uthman and I.

Uthman:
In phase 2, I needed to refactor a lot of the code I made for authentication, since it was violating clean architecture and SOLID rules. Pull request #60 highlights the main refactoring I did, where I separated the login and signup into their own classes to follow SRP and OCP. I also refactored the Profile class and removed dependencies to it, which made our overall design a lot cleaner. This can be seen in pull request #76.

Munim:
In phase 2, there have been many changes and a huge amount of progress. One of the most vital aspects of a program is to test how well it works. I have been in charge of the testing, and I have created a few test cases to intensively check our program. I have also included javadoc where it was necessary for good design. I have tried to make sure that our program is bug free and works properly. Other than that, I have contributed in making the design decisions, giving ideas of how we can apply the SOLID principles etc. Lastly, I have also helped with the design document and made sure that it looks professional and easy to read.
Here is my pull request where i added all of our test cases: Add Tests

Victor:

In phase 2, I had worked on the design pattern that we implemented sections along with refactoring and the test cases. Java doc was included with the test cases and I communicated with the group about how we should implement certain aspects of the project.