System design document for Olympus (android)

A Training Tracking App

Victor Hui, Philip Lindström Rabia, Patrik Olsson, Valdemar Vålvik, Oscar Wallin

2021-10-23
version 3

Contents

1 Introduction

The purpose of this document is to introduce an overview of the architecture and design patterns used for the development of the app.

1.1 Definitions, acronyms

- **PB/PR** - Personal best/Personal record
- **Reps** - repetitions
- **Sets** - is the multiplier of repetitions separated by rest ex. 3 sets of 12 repetitions
- **1-rep max** - what is the heaviest someone can lift only 1 time
- **Olympus** - the name of the application

2 System Architecture

Because of using the Android Framework, the architecture is built around the **MVVM**-pattern and our application consists of four parts:

- The **Model**, containing the business logic for the application. To keep dependencies low between the viewmodels and the model, a facade-class is used.

- The **View**, handling the display and user interface - a presentation.

- The **ViewModel**, handling commands and sending/receiving updates.

- The offline **Database**, used for storing the various data used in the application. In this prototype, the program will only be able to store temporary objects. This means that an object will be stored in the database during runtime, but will be removed when the application is terminated. Some model objects will be hardcoded inside the mock-database for display purposes.

2.1 Application Flow

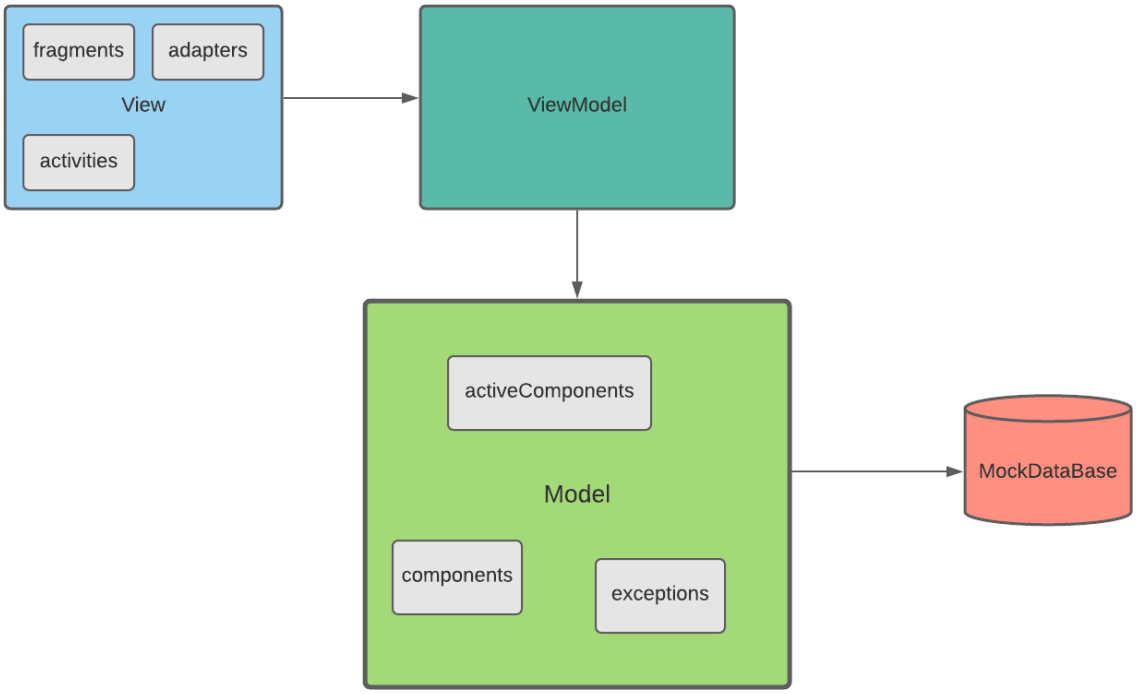
When the app starts, an instance of the database will be created with hardcoded plans, workouts, exercises and completed workouts. As mentioned above, the database will only be able to save the hardcoded trainingComponents between runs and not any newly created ones during runtime. All the components inside a plan (workouts and exercises) are created separately and later put together. This enables the user to close down the application but not lose newly created plans or workouts when they do not, at that point contain anything more than a name. This does not work with the current database because of the earlier mentioned limitations.

Because the application is built using Android and its framework, the starting point of the application is given from the chosen activity in the AndroidManifest, which in our case is the MainActivity which directs the user to the home screen with its UI elements. When completely exiting the application (terminating the process), the user loses the data they have provided to create components during runtime because of the database limitations described earlier. If the app instead is exited by switching to another one or returning to the Android home screen, the database will still be running and later on display the user-created data if they choose to enter the application again.

3 System Design

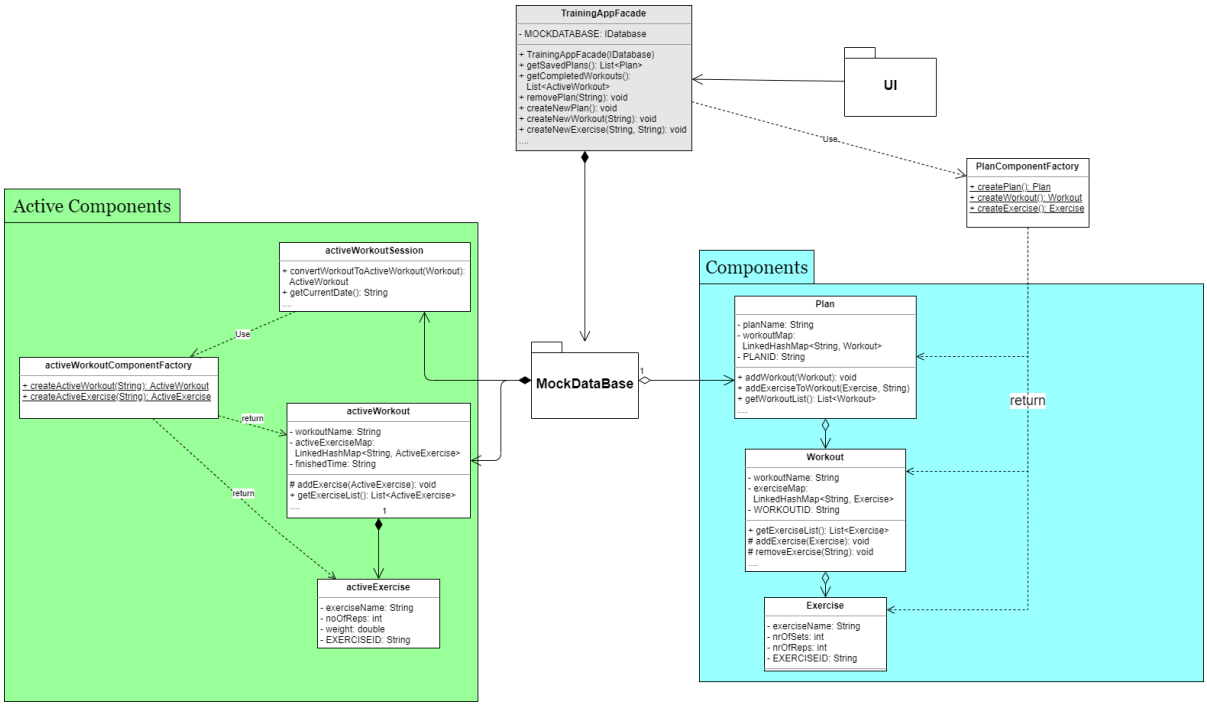
In this section, all the packages and classes will be displayed and explained through UML-diagrams. Furthermore the design patterns used will be discussed.

3.1 Top Level Packages



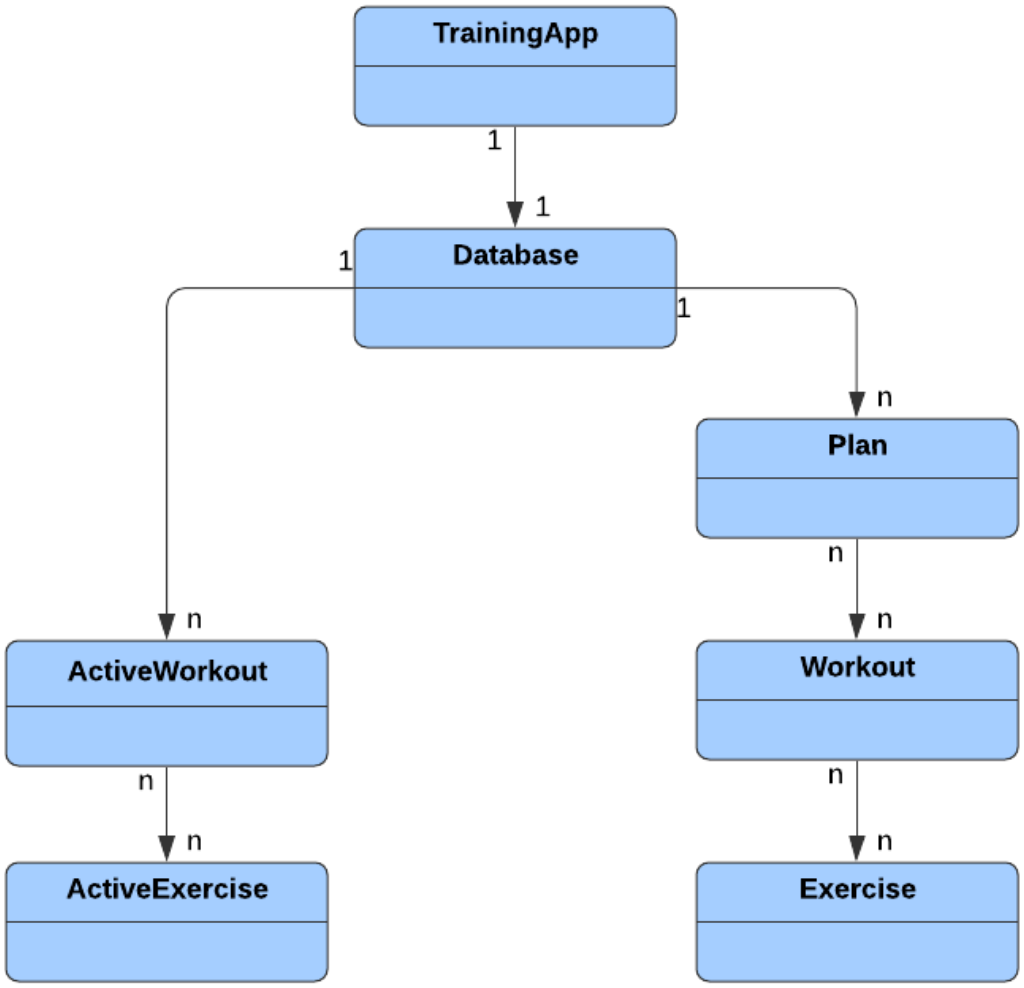
The View communicates with the ViewModel. The ViewModel communicates with the Model through a facade class inside the Model-package. The Facade class in turn communicates with the Database and retrieves or stores necessary information.

3.2 Model package



Our model relies heavily on delegation to carry out the commands from the UI. This makes the model less coupled since only classes that need to communicate with each other have dependencies. For example the MockDatabase only needs to be dependent on Plan in order to change an Exercise since it delegates the task to Plan which in turn delegates the task to Workout.

3.2.1 Domain model vs Design model



Each part of the domain model is represented in our design model as a class. The relationship between the classes are also kept the same between both models. Towards the end of the development the two models ended up closely related and the relations between our classes also remained about the same.

3.2.2 Facade

The facade class TrainingAppFacade acts as the single entry point to the model for the ViewModels. All changes to the model have to go through the facade which means that there is less coupling between the ViewModels and the classes inside the model package. When the UI

part of the application wants to make changes to the model, a ViewModel communicates with the facade. The facade in turn communicates with the database and a factory-class to handle potential commands from the UI. Our design philosophy is that every single change to the model has to go through TrainingAppFacade.

3.2.3 Defensive copies

In order for the views to know what they should display they are sent copies of objects in the database through the model. By sending copies we can prevent accidental changes in these copies from affecting the objects in the database. It also forces the ViewModel to go through the TrainingAppFacade when it wants to make changes to the model. Each copy has a String-Id that corresponds with the actual object, thus updates to the actual object can be done through sending this String-Id along with the data that needs to be updated to the model.

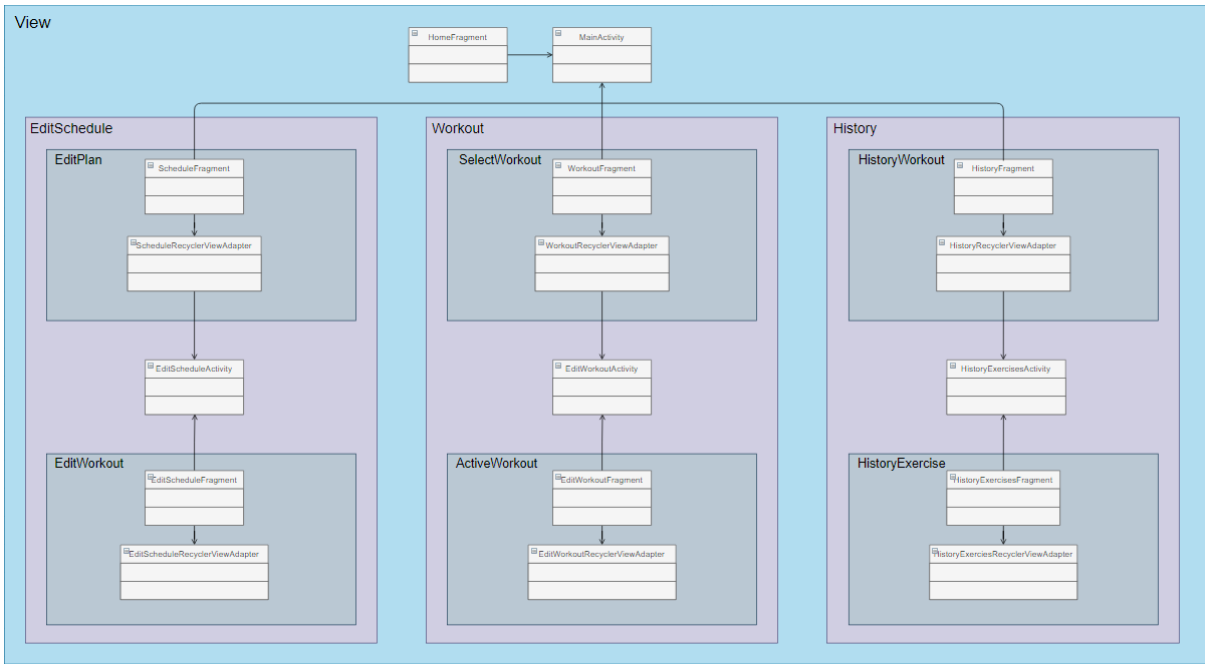
3.2.4 Database

To store both pre-made objects and user-created objects during runtime, an offline MockDatabase is used. The database manages, modifies and updates the various data stored and lends itself to making the data more secure and simplifies the communication on a package level. A TrainingAppFacade gets access to the Database through dependency injection, this way we can easily switch out the database for a different one.

3.2.5 Exceptions thrown in model

Exceptions thrown in the model are forwarded to the ViewModel. In this application the only exceptions that are thrown are NullPointerExceptions, e.g when trying to get objects from a HashMap with an invalid key. We use Objects.requireNonNull to throw this exception so that a message is passed along with it. Currently the ViewModel doesn't catch these exceptions, but that is something that could be implemented in the future. However one can argue that by not catching the exception the app is allowed to crash. This is reasonable considering that the nullpointerexception should never be able to happen and would indicate serious problems in the code.

3.3 View Package



The view package contains a FragmentView for each page of the program, a few activities and an adapter for each recyclerView. A fragment is used to initiate all the needed UI-components (i.e. text or buttons) on the associated layout. The information of positions and design of each element is set in the corresponding XML file.

3.3.1 Activities

The MainActivity contains all of the pages including a navigation bar and is the starting point for the application when the user enters it. It is possible to make an app using only a MainActivity. However, to make a hierarchical system, additional activities are suitable. For example, the page “Schedule” displays all of the user’s plans. Editing a plan’s workout brings the user to the EditScheduleFragment page. This was done using an additional activity as the page should not be found on the navigation bar.

3.3.2 Listeners

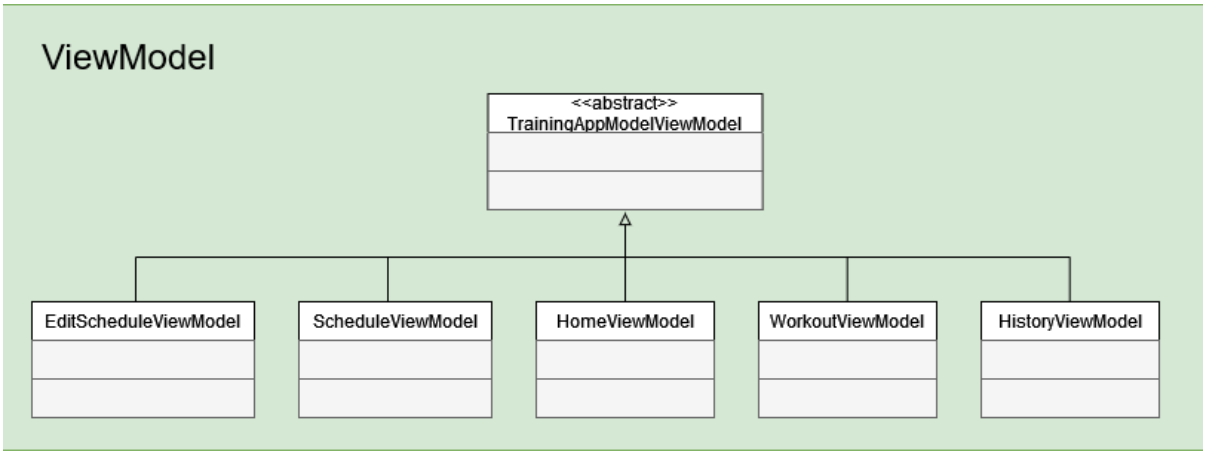
Changes to the state happen when the user either clicks on a button or edits a textfield. This is done by the usage of listeners that exist in the android library. Buttons use onClick listeners

and each textfield has an onFocusChanged listener. These listeners are set when the button or textfield is initiated.

3.3.3 Miscellaneous

As the model method getSavedPlans(); returns a copy of the database list, a system had to be implemented to make sure the spinner(dropdown) worked properly. When a new plan is created, the new plan should be moved to the top of the spinner which also means it is on the first position in a list. This could not be done without major changes to the model as the database is a linkedhashmap. Instead the UI gets its own list to display, that does not have the same order as the database. This allows the spinner to make the new plan instantly selected. The UI is synced up with the database again when the user leaves the page that contains the spinner.

3.4 ViewModel Package



The ViewModel acts as a bridge between our view and the model. Every fragment has a corresponding ViewModel. For instance, WorkoutFragment.java has an instance of WorkoutViewModel.java. In order to change the state of the program, methods in the ViewModel are called upon from the button and textfield listeners.

Every page that can change the state of the program has to have an instance of EditScheduleViewModel. This ViewModel has the needed methods to update and get data from the model. Every method calls upon the training app facade to interact with the model. In order for multiple pages to share information, a singleton instance is implemented in the EditScheduleViewModel. For instance, when the user selects a plan in the spinner, or clicks on a workout, that information is set in the singleton instance. The next page then gets the selected workout or plan. These objects are not actual references to the model but an

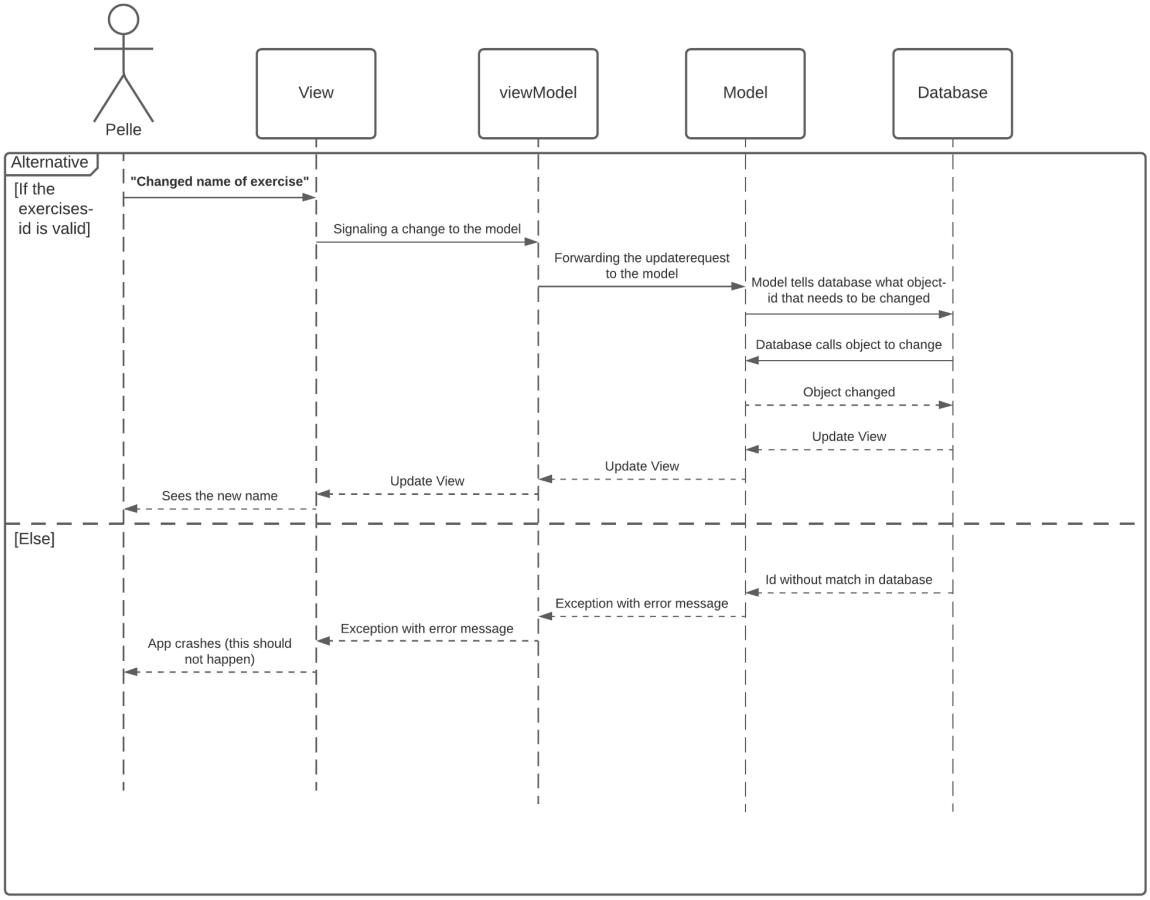
important part of the UI to sync up each page. Selected plan and workout contains the same id as the model which makes it easy to find a specific plan, workout or exercise.

3.4.1 TrainingAppViewModel

In order to solve the problem that all ViewModel classes need to go through the same instance of the Facade class in order to access the same database, we have created a common abstract class that all ViewModels extend from. This abstract class in turn extends the ViewModel class and contains a method for getting the same instance of TrainingAppFacade every time it is called.

3.4.2 Updating ViewModel

Below is an example sequence diagram of how a ViewModel is updated.



3.5 Design Patterns

3.5.1 Adapter
An adapter acts as a bridge between the data and the view. Various adapters are needed in order to set certain UI-elements' data and behaviour. This means a spinner (dropdown menu) can contain object references instead of a string. This allows us to access objects directly instead of having to find the associated object by comparing strings. Using adapters is a common practice in Android development.

3.5.2 Factory
ActiveWorkoutComponentFactory and PlanComponentFactory uses the Factory pattern to encapsulate the logic behind creating the objects. Some of the constructors lack a corresponding factory method. The purpose of these are only to make it easier to create the mock-objects in the mock database and would not exist otherwise.

3.5.3 Facade
As previously mentioned, we use TrainingAppFacade to act as the single entry point to the model for the viewmodels,

3.5.4 Singleton
To make sure that only one instance of certain viewModels are running the Singleton pattern was implemented, returning the current instance if it is running or creating a new one if the opposite is true.

3.5.5 Observer (Listener)
The listener pattern is used for buttons, windows for editing texts and popup messages in order for code to run when certain events occur (onFocusChange, onFocus, onClick). No implementations of custom Listener interfaces were needed in the application since our application only has to use them during button presses or when changing values in text windows, functionality that the base Android tools can provide.

3.5.6 Composite
Our main design for Plan, Workout and Exercise as well as ActiveWorkout and ActiveExercise uses the composite pattern, composing the objects into a tree structure and the application is then able to handle these structures like individual objects.

4 Persistent Data Management

In our case, we use a MockDatabase to store the various data in our application which means that when you terminate the application, the current data saved during the session will not be available the next time you start the application. If we would want the data to be available throughout sessions, an online database would have to be created and be running.

5 Quality

5.1 Testing

For model testing, the project has used the JUnit framework and all tests are present in the test package. The bulk of the tests are located in TrainingAppFacadeTest to operate the tests in a similar way to how the application runs.

5.1.2 Circle Ci

To automate tests and more easily identify and correct errors in the code from various pushes, the continuous integration platform CircleCi was used as it provides instant feedback for the project.

5.2 Known issues/improvements

- The use of LinkedHashMap could be improved by implementing ways to make a deep copy of the LinkedHashMap instead of cloning a new Map that points to the same references.
- Some parts of the application do not return to the previous view when using the back button part of the general Android interface, instead returning the user to the starting point of the application (Home).
- If the MockDatabase were located in the modelfolder instead of a seperate folder, several update/set-methods could be set protected in the Plan-class.
- The GUI used hard-coded dimensions which can result in the fact that some phones do not see the whole UI for the application.
- Implementing Interface for Plan and ActiveWorkout to further increase abstraction.
- Implement a common abstract class for Exercise and ActiveExercise, as they share a significant amount of methods. The need could also be seen for Workout and ActiveWorkout.
- When ending an active workout to add it to the history, the application replaces the current activeWorkout object with null, which may lead to further problems and is a sub-optimal solution.
- The exceptions that are forwarded to the ViewModel are not caught which means that they will crash the application. We would like to implement code for catching the exception and displaying an error message to the user.

5.3 Analytical Tools

5.3.1 STAN

Not able to generate STAN at this point. May be added later.

6 References

6.1 Tools

- **Circle Ci**
  - Used for continuous integration, alternative to travis
  - https://circleci.com/

- **Figma**
  - Prototyping tool
  - https://www.figma.com/

- **Google DrawIO**
  - Diagram software used for UML, domain model and class-uml
  - https://app.diagrams.net/

- **Android studio** (IDE)
  - Integrated development environment for Google's Android operating system
  - https://developer.android.com/studio

- **JUnit**
  - Unit testing framework for the Java programming language.
  - https://junit.org/junit5/

- **Gradle**
  - Build tool used with Android Studio
  - https://gradle.org/

- **Lucidchart**
  - A software for creating UML diagrams
  - https://www.lucidchart.com/pages/

Requirements and Analysis Document for Olympus

A Training Tracking App

Victor Hui, Philip Lindström Rabia, Patrik Olsson, Valdemar Vålvik, Oscar Wallin

2021-10-23
version 2

1 Introduction

Moving past the time of the Corona pandemic and lockdowns, people start to get back to the gym. Therefore the demand for fast and agile training apps increases. The app, provided by AEY investment group aims to give both experienced and inexperienced gym attendees. The tool to help with planning and tracking of workouts and progression. Through a sleek UI and easy functionality the aim is to, through the app, motivate the user to a more active lifestyle.

1.1 Definition, acronyms, and abbreviations

- PB/PR “Personal best/Personal record”
- reps “repetitions”
- sets “is the multiplier of repetitions separated by rest ex. 3 sets of 12 repetitions”
- 1-rep max “what is the heaviest someone can lift only 1 time”

2 Requirements

2.1 User Stories

S1: Workout plan

Implemented

- Yes

Description

As a user, I want to make a workout plan that contains x amount of workouts.

Functional

- There is a way to add/remove a workout plan
- There is a way to name the workout plan.
- There is a way to edit(add/remove workouts) the workout plan
- There is a way to view previously saved workout plans

Non-functional

- The user goes through a wizard-like process to create the workouts.
- The user can delete workouts by swiping left and then pressing a minus sign
- The user can add a workout by pressing a button.

S2: Workouts

Implemented

- Yes

Description

As a user, I want to be able to create workouts that contain exercises with sets and reps.

Functional

- There is a way to add/remove a workout
- There is a way to name the workout
- There is a way to edit(add/remove exercises) the workout
- There is a way to view previously saved workouts.

Non-functional

- I can delete exercises by swiping left then pressing the minus sign
- I can add an exercise by pressing a button

S3: Exercises

Implemented

- Yes

Description

As a user, I want to be able to create exercises. When creating an exercise I want it to be connected to an exerciseId. All exercises of the same type(e.g Bench Press, Deadlift) should have the same exerciseId.

Functional

- There is a way to add/remove exercises.
- There is a way to name the exercise.
- There is a way to add an exerciseId for identifying purposes.
- There is a way to edit(add/remove sets) the exercise.
- There is a way to view previously saved exercises.
- The exerciseId should be saved automatically in the database on creation.

Non-functional

- The user cannot have two exercises of the same name.

S4: Current Workout

Implemented

- Yes

Description

As a user, I want to create a current workout session where I can input data (weight, rep, sets) for each exercise.

Functional

- The user can start a current workout by choosing a workout plan and then a workout from that plan.
- I can change information about the exercise such as sets, reps and weight during my workout.
- I can add exercises that weren't originally in the workout.
- Time that has passed since workout has started should be displayed(in minutes/seconds, and maybe hours/minutes/seconds eventually)
- After finishing a workout, the workout should be saved automatically.

Non-functional

S5: Navigation Bar

Implemented

- Yes

Description

As a user, I want to be able to easily change between the different parts of the application.

Functional

- There is a way to switch between the different parts of the program.
- The bar for changing to different parts is located at the bottom of the application.
- There is a way to see where you currently are by looking at the navigation bar.

Non-functional

- You should be able to see the text and animation of the nav-bar at all times.

S6: Workout History

Implemented

- Yes

Description

As a user, I want to see a history of my previously performed workouts.

Functional

- There is a way to see previously performed workouts
- Each workout is accompanied by the date it was performed

Non-functional

- Workout history can be accessed at the history tab

S6: Saving PB's

Implemented

- No

Description

As a user, I want to save my PB's and write down the date, weight and reps as well as what exercise.

Functional

- There is a way to see previous PB's
- There is a way to add and update personal PB's
- After finishing a workout the app will check automatically if a new PB is achieved for any exercise.

Non-functional

- The relevant PB-data can be accessed at the history tab

S7: PB graph

Implemented

- No

Description

As a user, I want to see the progress of my PB's through a graph.

Functional

- Use the PB-data from an exercise and display it as a graph.

Non-functional

- The relevant PB-data can be accessed at the history tab

S8: Calculating 1-rep max

Implemented

- No

Description

As a user, I want to be able to calculate my 1-rep max.

Functional

- Use a calculator and the following algorithm (insert here) to easily show the 1-rep max?
- A part of our PB section?

Non-functional

- The user has 2 input boxes to write nr of reps and weight. The calculator shows the corresponding 1-rep max as output.

S9: About Us page

Implemented

- No

Description

As developers, we want to be seen in suits on our ‘About Us’ page.

Functional

- There is a way to find and view an ‘About Us’ page
- There is some relevant information on this page

Non-functional

S10: Workout presets

Implemented

- No

Description

As a user, I want to be able to copy certain workouts both from a set of standard workouts or other famous workout presets

Functional

- Have a (database) with standard and other known workout presets
- Ability to swap between workout presets

Non-functional

- These could be available at the same place as the user created workouts/exercises

S11: Setting options

Implemented

- No

Description

As a user, I want to be able to change different settings.

Functional

- There is a way to change the color theme
- There is a way to change measurement units
- There is a way to change the app based on acceptability problems, like color blindness

Non-functional

- The settings will be available at the settings tab. A usage of checkboxes or some other input box will be used for the different setting options.

S12:Startpage

Implemented

- No

Description

As a user, I want to get some fun and inspiring information from the apps startpage

Functional

- There is a graph of some interesting performance from past workouts
- There are some inspirational quotes/texts that are shown

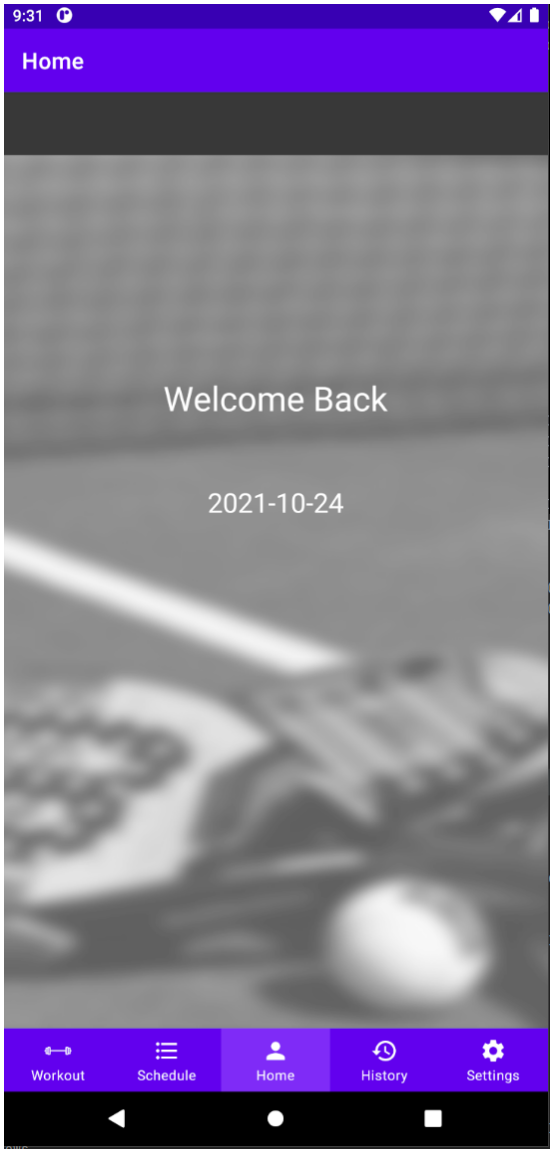
Non-functional

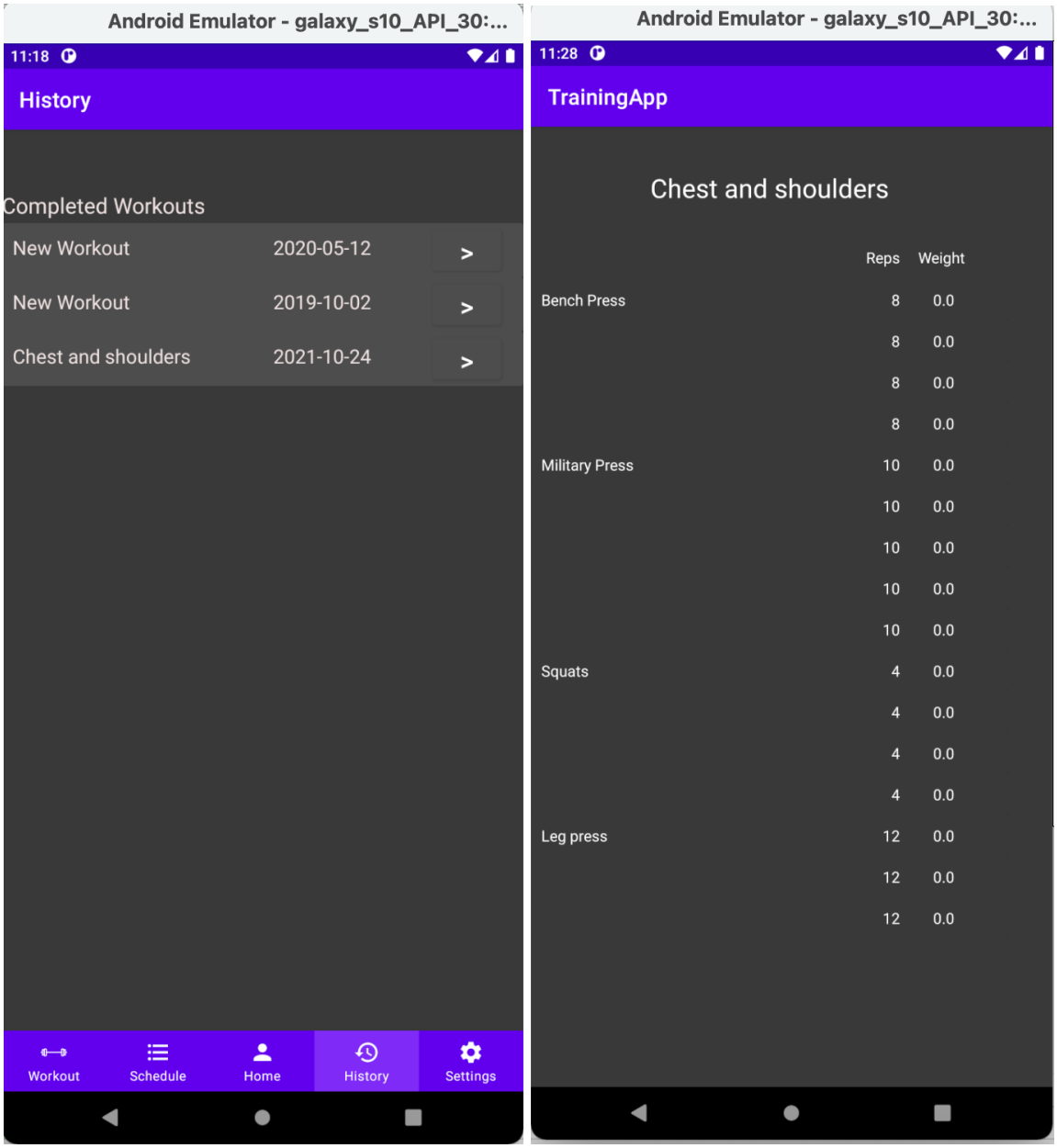
2.2 Definition of Done

- All the acceptance criterias are met for the user stories
- All the co-authors to the user stories commits are given
- The user story-specific code is merged to the main branch
- The code should be executable
- Appropriate code should be documented

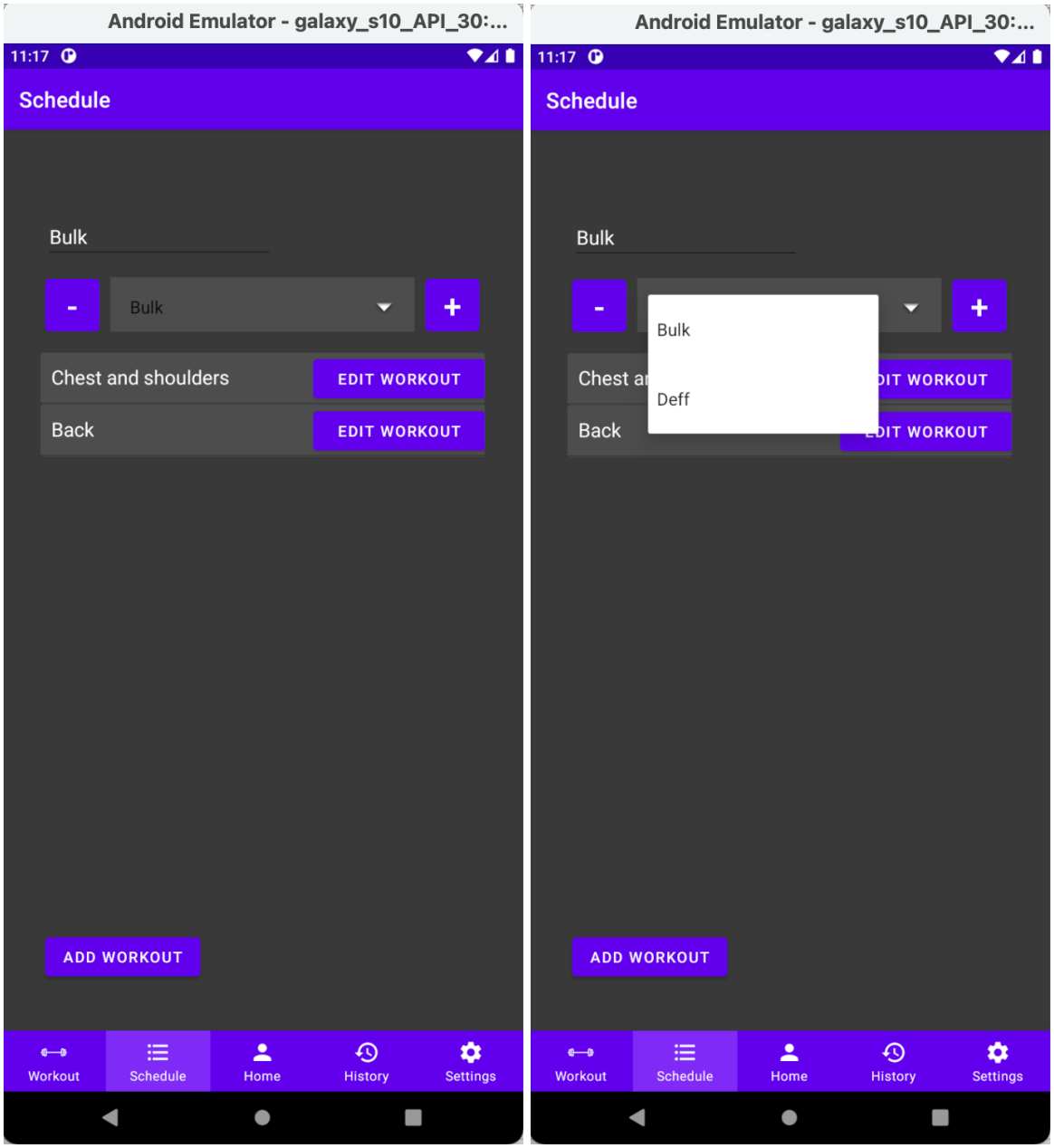
2.3 User interface

When starting the app the user is shown a home page which displays today's date and welcome back.

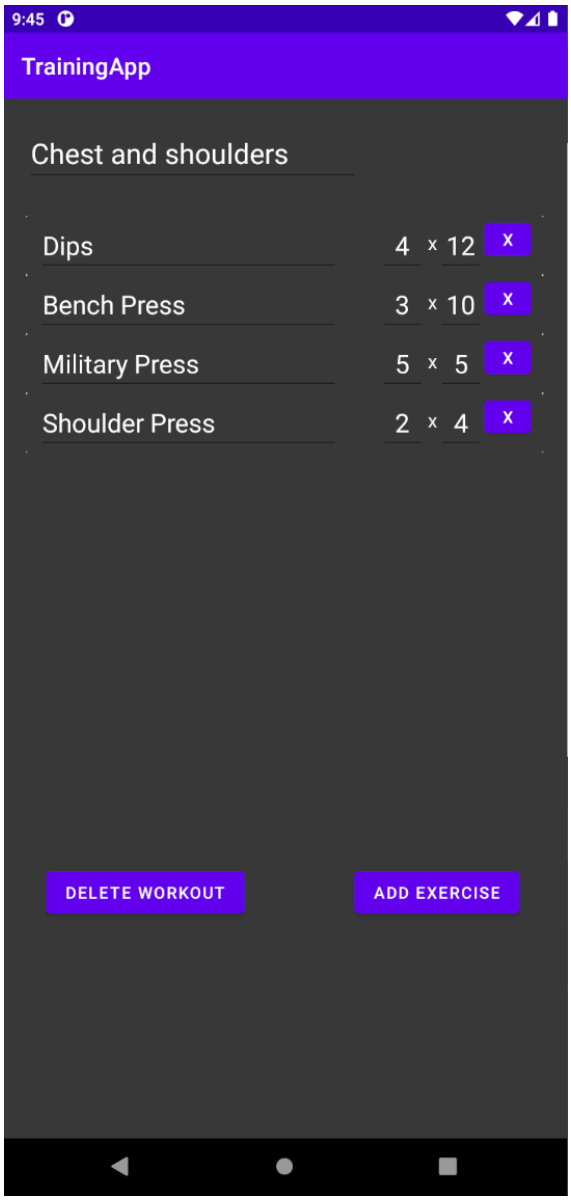




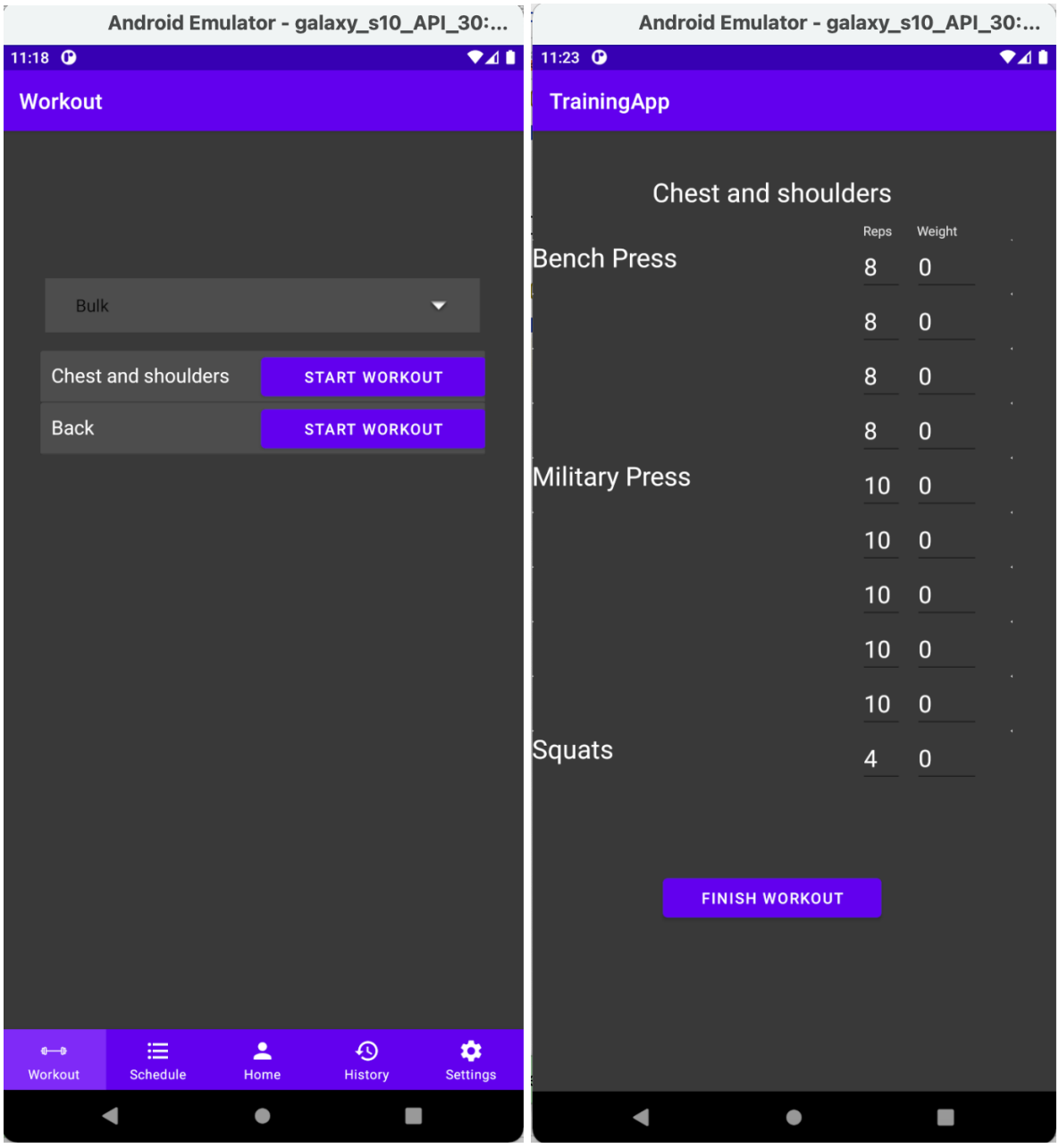
These two pictures show the view in the “History”-tab. The user can alternate between training history (left), which shows the workout performed on a specific date and exercises done in a certain workout (right), which shows the exercises reps, sets and weights.



This is the “Schedule”-tab. Here the user can create a new schedule or enter previously saved ones. The user can select a schedule by using the dropdown menu (right). The “-” and “+” buttons remove the selected schedule. A popup message pops up and asks the user if they are sure they want to remove a trainingschedule. Add workout adds a workout to the selected schedule. The user can click on the edit workout button to edit it.

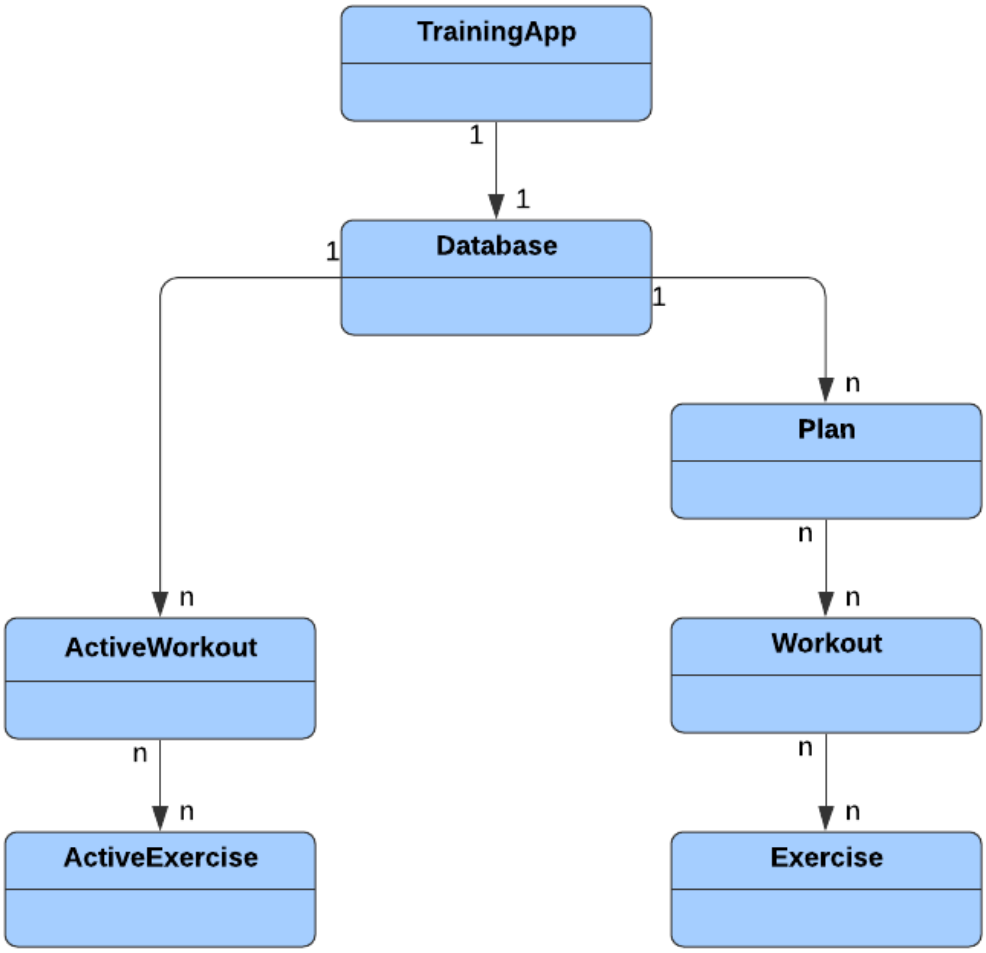


This is the page the user sees when clicking the edit workout button. The user can change the name of the workout, the name of each exercise and the amount of reps and sets of each exercise. They can also remove an exercise by click on the “x” button. “Add exercise” adds a new exercise and delete workout deletes the whole workout. Once again, a popup asks the user if they are sure.



This is the “Workout”-tab. At first the user is presented by a list of available workouts based on the preplanned plan from the schedule (left). After selecting a workout, all the exercises are displayed in that workout (right). In this stage the user can edit weights and reps depending on how the actual workout went. Finished workouts are saved in the history tab.

3 Domain model



3.1 Class responsibilities

TrainingApp
TrainingApp is supposed to act as a datahandler that connects all the pieces of the model and makes it easy to pass it along to the different viewmodels.

Database
The Database stores all information related to workouts, such as workout plans and previously finished workouts. The class contains methods for editing the workout plans.

Plan

A Plan acts as a container for workouts. It is also responsible for editing the workouts.

Workout
A Workout acts as a container for Exercises. It is also responsible for editing the exercises.

Exercise
An Exercise contains information about an exercise such as sets and reps.

ActiveWorkout
An ActiveWorkout is a Workout that the user has selected and wants to perform. It is a copy of an already existing workout but with the added attribute of date(when the ActiveWorkout was finished). Like workout it also has a container but for ActiveExercises instead of Exercises and contains methods for editing said ActiveExercise.

ActiveExercise
An ActiveExercise is a copy of an Exercise with the added attribute of weight.

4 References

- **Circle Ci**
  - Used for continuous integration, alternative to travis
  - https://circleci.com/
- **Figma**
  - Prototyping tool
  - https://www.figma.com/
- **Google DrawIO**
  - Diagram software used for UML, domain model
  - https://app.diagrams.net/
- **Android studio** (IDE)
  - Integrated development environment for Google's Android operating system
  - https://developer.android.com/studio
- **JUnit**
  - Unit testing framework for the Java programming language.
  - https://junit.org/junit5/
- **Gradle**
  - Build tool used with Android Studio
  - https://gradle.org/

Peer review of Group 17E
by AEY

Names:
Class and method names follow the camel case notation which is good practice. The names are well thought out. It's easy to intuitively understand the purpose of each class/method which in turn makes it easier to follow along in the code.

Coding Style: The coding style is consistent. Same indentation and indentation of curly brackets used throughout the whole codebase.

Documentation: Some documentation missing for certain classes/interfaces. Whether or not this is a problem depends on if the group's Definition of Done specifies documentation to be written for every class as a requirement. It can be seen that most of the documentation is done for the public methods in the model-package classes, which is good since at the very least the model needs to be well documented.

Testing: According to Junit there is 100% coverage of the model classes so all of the classes have been tested. Not all methods in these classes have been tested, however most of the untested methods are getters and setters, and we think it's fine leaving them untested.

Structure: Good structure on package level, follows the MVC model. Model is also further divided into subpackages based on their responsibility. This shows that the codebase follows the Single Responsibility Principle. It also shows good separation of concerns which avoids unnecessary dependencies. The codebase follows the principle of SMART models, THIN Controllers, and DUMB Views. Their controller uses handlers to change data in the model and update views. Their views do not contain any logic and only change the graphical interface.

Performance/Security:
The program is lightweight and easy to run. We did not find any performance issues after trying the program out. Although the program behaved a bit strange during the sign up screen when the user entered the wrong format for the textfields. Instead of pointing out all the fields that were entered wrongly, the program only highlighted one of them.

A bit inconsistent in which methods are made package private. The setter methods in User are made package private, which is good since that ensures that no other part of the code can access these methods which increases the security of the class. Meanwhile the setter methods in Listing are set to public instead of package private, and since they aren't used anywhere yet it's hard to know what the correct choice is. If the other parts of the code do not need access to the setter methods it's better to make them package private.

In one case a getter(getUserAdress in User) returns a direct reference to the object, consider implementing defensive copying in order to avoid alias problems and increase security. This can

also be done for the getters that return attributes, since these are made private in the code but can be directly referenced through the getters. In some instances the attributes are made final which is great since it removes the worry that they will get changed later.

Abstraction: Good use of abstraction in view with an interface that all scenes implement, and controller where there is also an interface. Although the use of this interface doesn't seem to have been implemented yet as the interface itself is empty. Could increase abstraction by adding interfaces to the model.

Reusability/Maintenance: Multiple scenes in the view package reuse the code in the abstract class AbstractHyroScene through inheritance, which means it would be easy to create new scenes with similar functionality. And as mentioned before all scenes implement a common interface so if you needed to add or exchange a scene it wouldn't be difficult to do so. Could make the code more modular and thus easier to maintain through adding interfaces, for example if an interface for ListingHandler was added you would be able to easily switch between different Listing modules.

Design patterns:
Factory is used in HyroSceneHandlar to create the different scenes used in the application. The application itself is structured with MVC pattern in mind as is correctly set up on package levels, and the different parts of the application follows SMART Model, DUMB View, THIN Controller. Singleton can be seen in UserHandler, specifically in getInstance where it checks if the handler is view or not, creating a new one if true and returning the active one if it is already running.

Things to improve or think about:
Making the program more immutable by using defensive copying. This would make the program more secure by preventing difficult to find bugs. This should not affect performance that much considering how small the program is. Maybe you could expand the use of interfaces to increase abstraction. Currently there is one for the controllers and one for the views but none in the model-package.

```java
public void switchTo(String newSceneName) {
    hyroScene newScene = scenes.get(newSceneName.toLowerCase());
    switchTo(newScene);
}

public boolean switchTo(hyroScene newScene) {
    if(newScene == null) {
        return false;
    }
    switchScenes(newScene);
    return true;
}
```

Is it necessary for switchTo to be two separate methods? The true/false logic is never used when returned from the second switchTo. If the goal is to make sure that newScene is not null, maybe throw an exception if newScene == null.