System design document for Gym Companion

Augustas Eidikis, Daniel Olsson, Marcus Svensson, Erik Bock, Alexander Bergsten

2018-10-05
version 0.4

1 Introduction

This document details the structure of the components of the Gym Companion application, as well as it describes the implementation of different functions and features. Gym Companion is an application suited for helping both advanced and beginner users to keep track of their progress in the gym. The application offers a way to use both existing and customised routines and scheduling them through a calendar. Users can view their progress both on-the-go and after the workout through an intuitive statistics page, including an interactive graph and lifetime stats.

1.1 Definitions, acronyms, and abbreviations

- **Routine**: a workout routine, training routine
- **Exercise**: a part of a routine
- **Schedule**: a lists of intended routines and times
- **GUI**: Graphical User Interface (what the user sees and interacts with)
- **MVVM**: Model-View-ViewModel is a design pattern that helps us separate the logic from the user interface.
- **Model**: Also known as the domain model. Contains all logic
- **View**: A class that is connected to GUI and gets fed information to display
- **ViewModel**: A class that converts data into something that can be shown on a View

- **User Story**: A user story is a short description in common language of what a user wants to achieve.
- **Run-time**: Used to describe the state of the app when it's running
- **Subclass**: The classes under a specified class in a hierarchy
- **Super-class**: The classes above a specified class in a hierarchy
- **JSON**: JavaScript Object Notation, a data format for structuring objects and their properties.

2 System architecture

The application runs on a single local mobile device. It is divided into packages, each containing related classes and files. Gym Companion uses the MVVM pattern, where the ViewModel (our Controller/Presenter) handles communication between the View and the Model. These three application elements are split into three different packages with the respective responsibility area. This separates the logic from the user interface.

Unit tests for each package are located in the 'src/test/java/se.chalmers.group22.gy mcompanion' folder. LocalDatabaseTest and ParserTest need an Application context in order to run and access local files, therefore they are located in the 'src/androidTest /java/se.chalmers.group22.gymcompanion' folder.

The project consists of one single component: the application. The application's subsystem is divided into the following components:

2.1 GymCompanion

This package contains the main class of the application, **GymCompanion**. **GymCompanion** is the class that is used to represent the model for the viewmodels. There is only one instance of **GymCompanion** that is shared between all viewmodels.

2.2 Schedule

This package contains a schedule model. The schedule model is responsible for maintaining the user's scheduled routines.

2.2.1 Diagrams

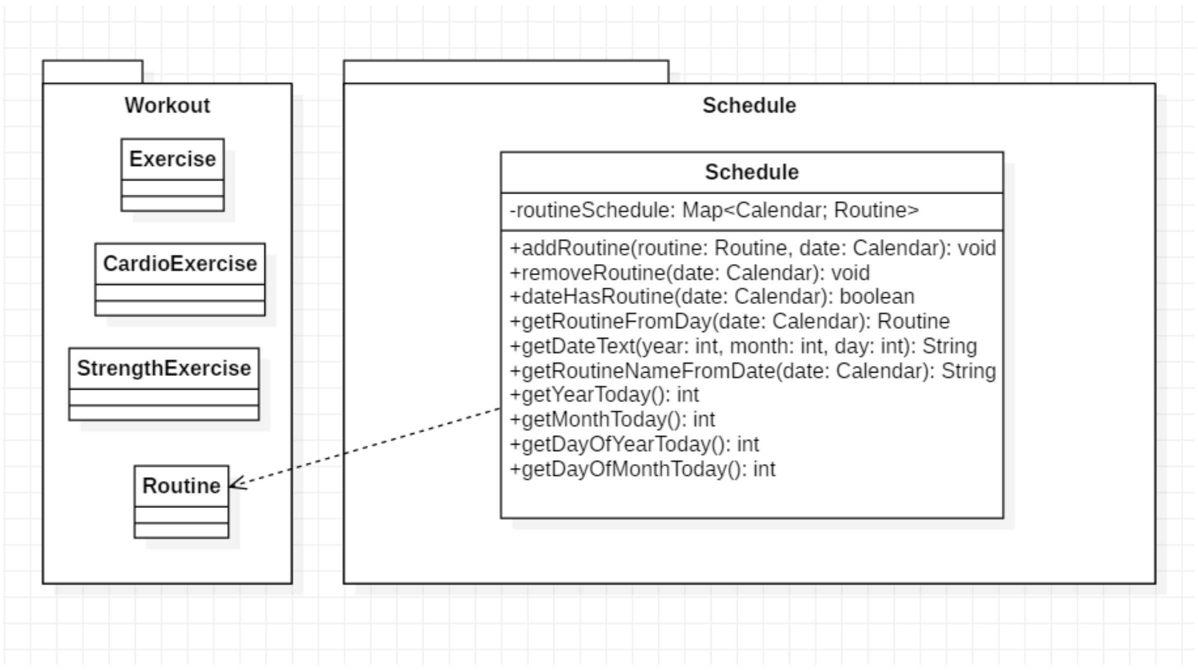
Workout
Exercise
CardioExercise
StrengthExercise
Routine
Schedule
Schedule
-routineSchedule: Map<Calendar; Routine>
+addRoutine(routine: Routine, date: Calendar): void
+removeRoutine(date: Calendar): void
+dateHasRoutine(date: Calendar): boolean
+getRoutineFromDay(date: Calendar): Routine
+getDateText(year: int, month: int, day: int): String
+getRoutineNameFromDate(date: Calendar): String
+getYearToday(): int
+getMonthToday(): int
+getDayOfYearToday(): int
+getDayOfMonthToday(): int

Figure 1: Schedule Package UML

2.3 Statistics

This package contains a statistics model. It's responsibility is to calculate statistics and prepare the data for plotting graphs. It takes routines and days from the schedule and with them creates data points for a graph. The days are returned as x-values, so that they can be used to show change over time. The routines are transformed into scores, which differ depending on which exercises they contain and how difficult they were, and are returned as y-values.

2.3.1 Diagrams

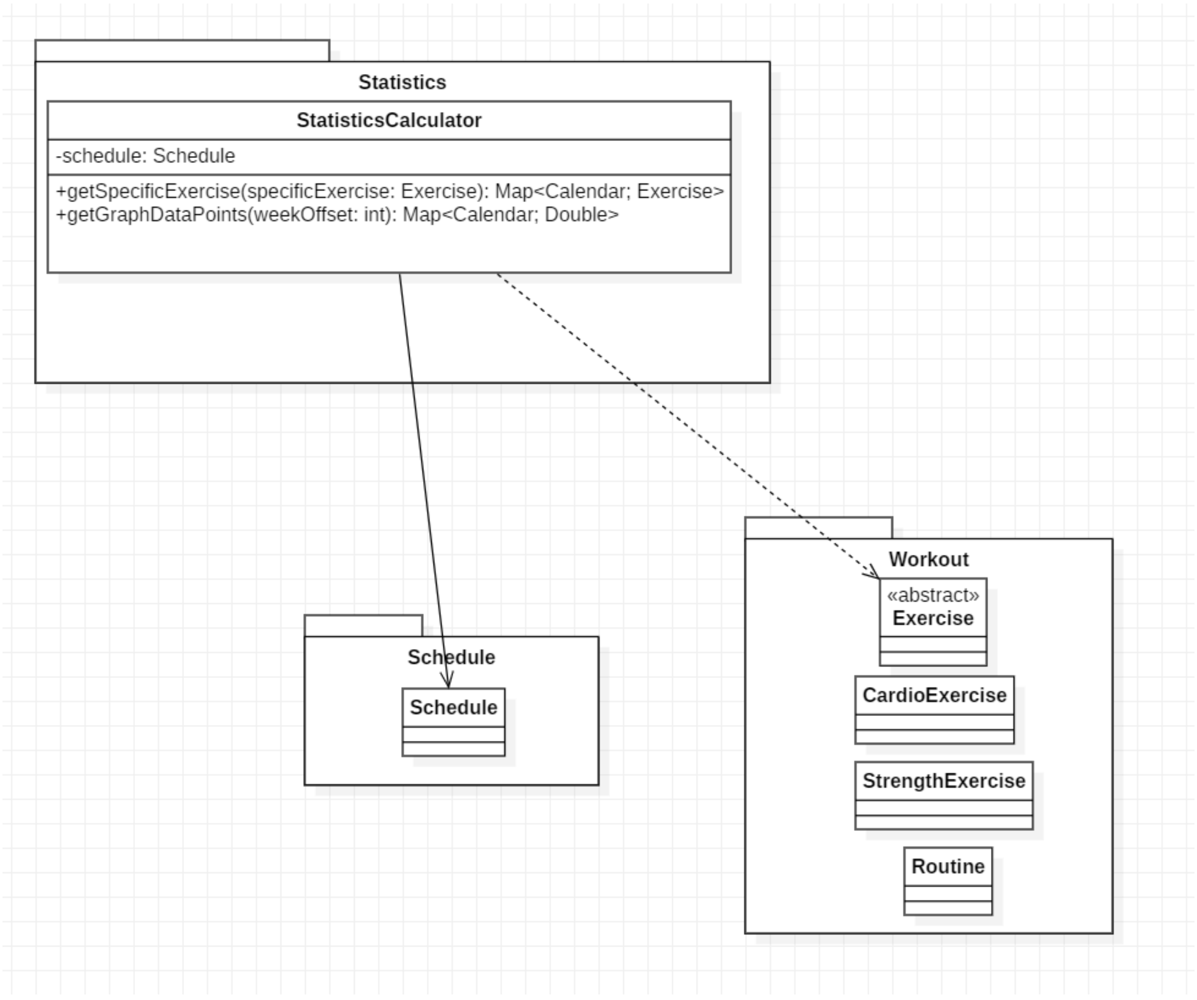
Statistics
StatisticsCalculator
-schedule: Schedule
+getSpecificExercise(specificExercise: Exercise): Map<Calendar; Exercise>
+getGraphDataPoints(weekOffset: int): Map<Calendar; Double>
Schedule
Schedule
Workout
«abstract»
Exercise
CardioExercise
StrengthExercise
Routine

Figure 2: Statistics Package UML

2.4 Strategies

This package contains logic for sorting algorithms. The classes in the package define different strategies as part of the Strategy design pattern. The strategy pattern makes it possible to choose an algorithm at run-time, while also making it easy to create and implement new algorithms. In the app the strategies are divided into two categories, **FilterStrategy** and **SortingStrategy**, which are both represented by interfaces that their respective type of strategy can implement. There are two different filter strategies and four different sorting strategies.

2.4.1 Filter Strategies

The filter strategies are **BeginnerFilter** and **MixedFilter**. Their task is to filter a list in a specific way.

- **BeginnerFilter** is used if the user wants to filter out everything except the easiest Routine/Exercise from every muscle group.

- **MixedFilter** on the other hand is a completely random filter that returns a randomised set of Routines/Exercises.

2.4.2 Sorting Strategies

The sorting strategies are **AscendingAlphabetic**, **AscendingDifficulty**, **DescendingAlphabetic** and **DescendingDifficulty**. Their task is to sort a list in a specific way.

- **AscendingAlphabetic** sorts a list of Routines/Exercises according to their names in alphabetical order starting from the first letter of the alphabet.
- **DescendingAlphabetic** sorts a list of Routines/Exercises according to their names in reverse alphabetical order starting from the last letter of the alphabet.
- **AscendingDifficulty** sorts a list of Routines/Exercises according to their difficulty from lowest to highest.
- **DescendingDifficulty** sorts a list of Routines/Exercises according to their difficulty from highest to lowest.

2.4.3 Diagrams

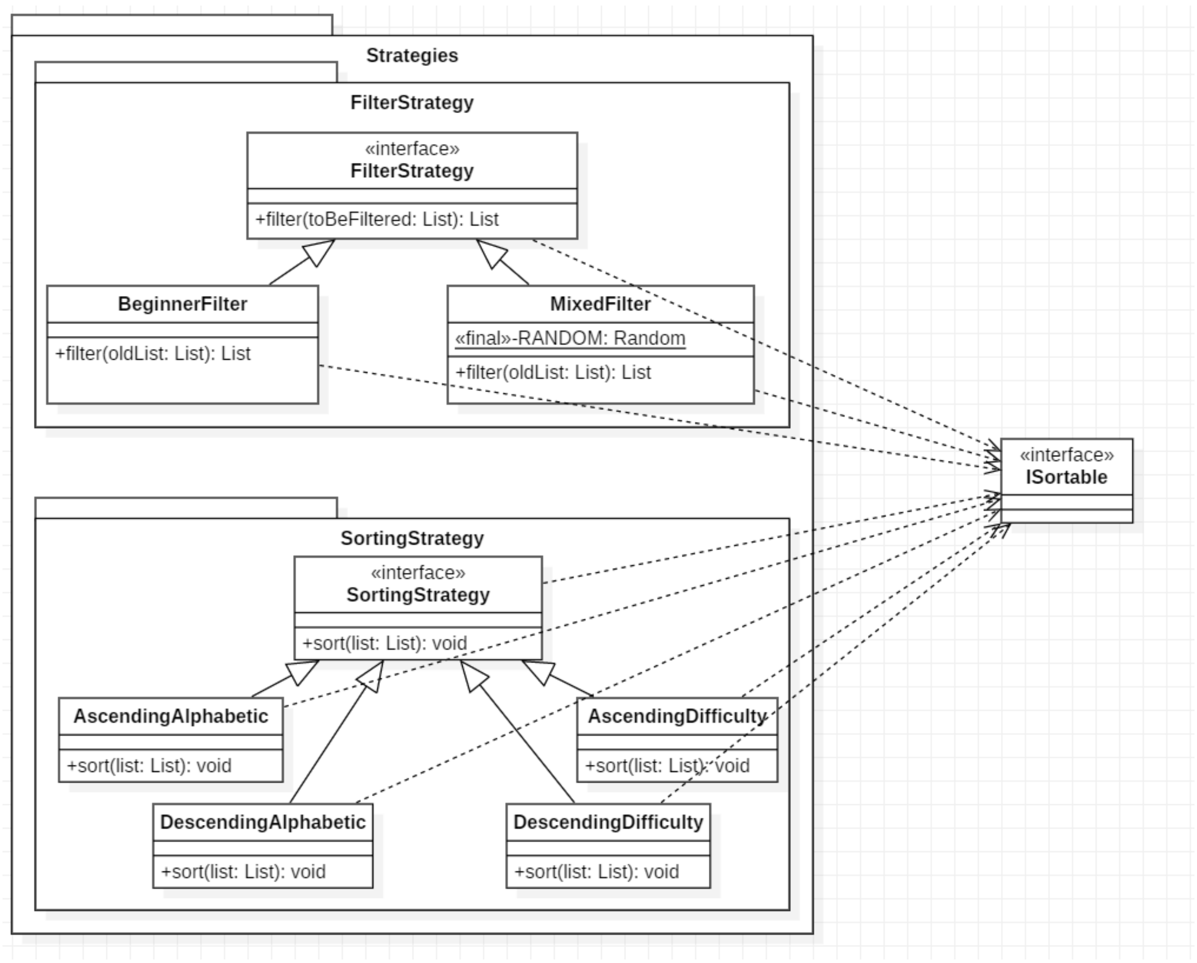

Figure 3: Strategies Package UML

2.5 User

This package contains a single class responsible for handling user related actions. The user class contains all the information about the user, their schedule, their own routines and their completed routines. This information is saved locally, so that it is still there after the user exits the app.

2.5.1 Diagrams

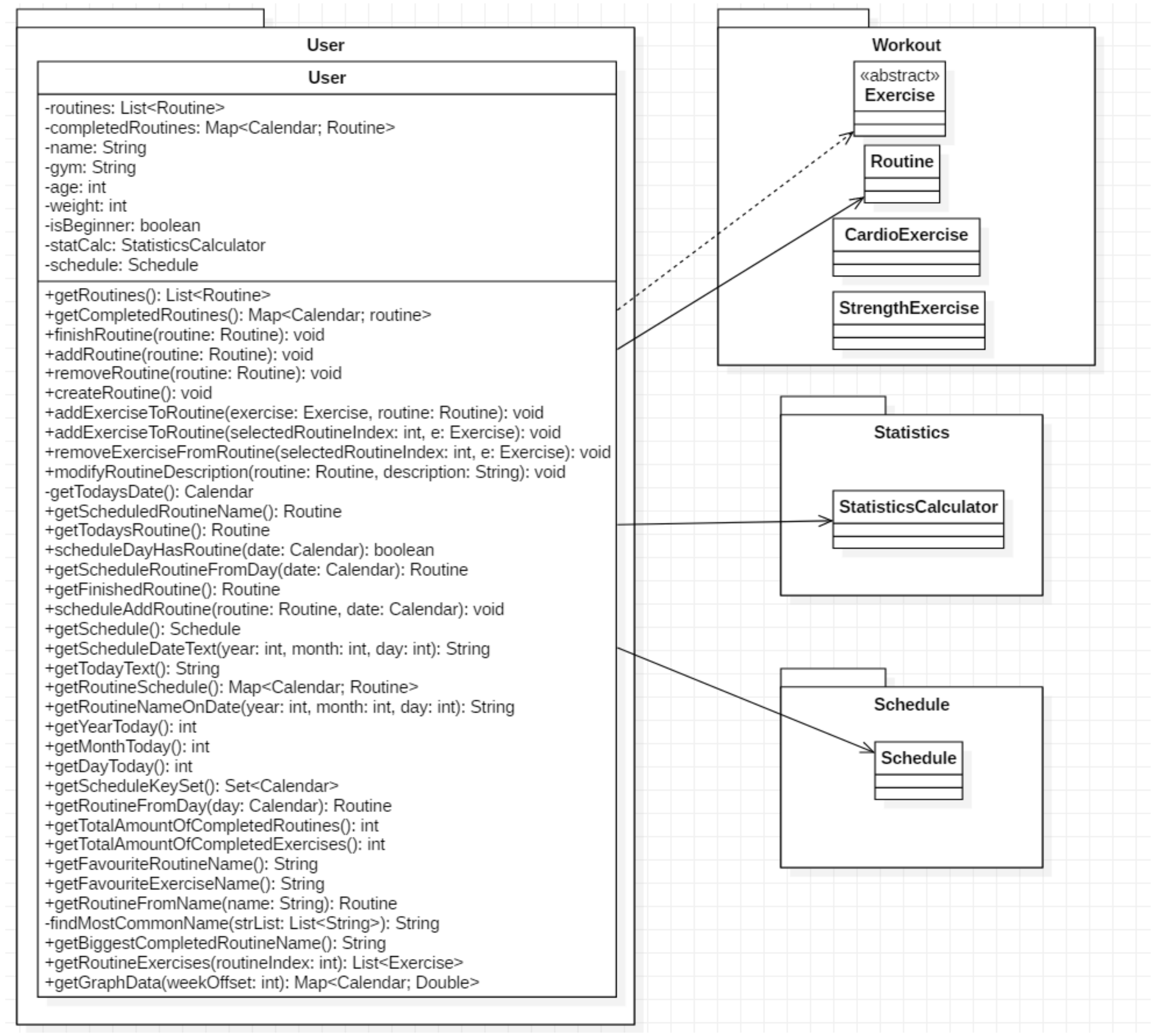
User
User
-routines: List<Routine>
-completedRoutines: Map<Calendar; Routine>
-name: String
-gym: String
-age: int
-weight: int
-isBeginner: boolean
-statCalc: StatisticsCalculator
-schedule: Schedule
+getRoutines(): List<Routine>
+getCompletedRoutines(): Map<Calendar; routine>
+finishRoutine(routine: Routine): void
+addRoutine(routine: Routine): void
+removeRoutine(routine: Routine): void
+createRoutine(): void
+addExerciseToRoutine(exercise: Exercise, routine: Routine): void
+addExerciseToRoutine(selectedRoutineIndex: int, e: Exercise): void
+removeExerciseFromRoutine(selectedRoutineIndex: int, e: Exercise): void
+modifyRoutineDescription(routine: Routine, description: String): void
-getTodaysDate(): Calendar
+getScheduledRoutineName(): Routine
+getTodaysRoutine(): Routine
+scheduleDayHasRoutine(date: Calendar): boolean
+getScheduleRoutineFromDay(date: Calendar): Routine
+getFinishedRoutine(): Routine
+scheduleAddRoutine(routine: Routine, date: Calendar): void
+getSchedule(): Schedule
+getScheduleDateText(year: int, month: int, day: int): String
+getTodayText(): String
+getRoutineSchedule(): Map<Calendar; Routine>
+getRoutineNameOnDate(year: int, month: int, day: int): String
+getYearToday(): int
+getMonthToday(): int
+getDayToday(): int
+getScheduleKeySet(): Set<Calendar>
+getRoutineFromDay(day: Calendar): Routine
+getTotalAmountOfCompletedRoutines(): int
+getTotalAmountOfCompletedExercises(): int
+getFavouriteRoutineName(): String
+getFavouriteExerciseName(): String
+getRoutineFromName(name: String): Routine
-findMostCommonName(strList: List<String>): String
+getBiggestCompletedRoutineName(): String
+getRoutineExercises(routineIndex: int): List<Exercise>
+getGraphData(weekOffset: int): Map<Calendar; Double>
Workout
«abstract»
Exercise
Routine
CardioExercise
StrengthExercise
Statistics
StatisticsCalculator
Schedule
Schedule

Figure 4: User Package UML

2.6 Workout

This package contains classes for modelling a workout routine and its exercises. The exercise model is divided in an exercise hierarchy.

2.6.1 Routine

Routine is a class that models a workout routine. It's primary functionality is to house a list of different exercises. The user is able to add and remove exercises from a routine.

2.6.2 Exercise

Exercise is a class that models a training exercise. Exercise is represented by a hierarchy with an abstract **Exercise** class at the top and two implementations **StrengthExercise** and **CardioExercise** below it. The abstract **Exercise** class makes use of the template method design pattern by defining abstract ”skeleton” methods **calculateScore** and **clone**, and then letting it's subclasses define concrete implementations.

2.6.3 Diagrams

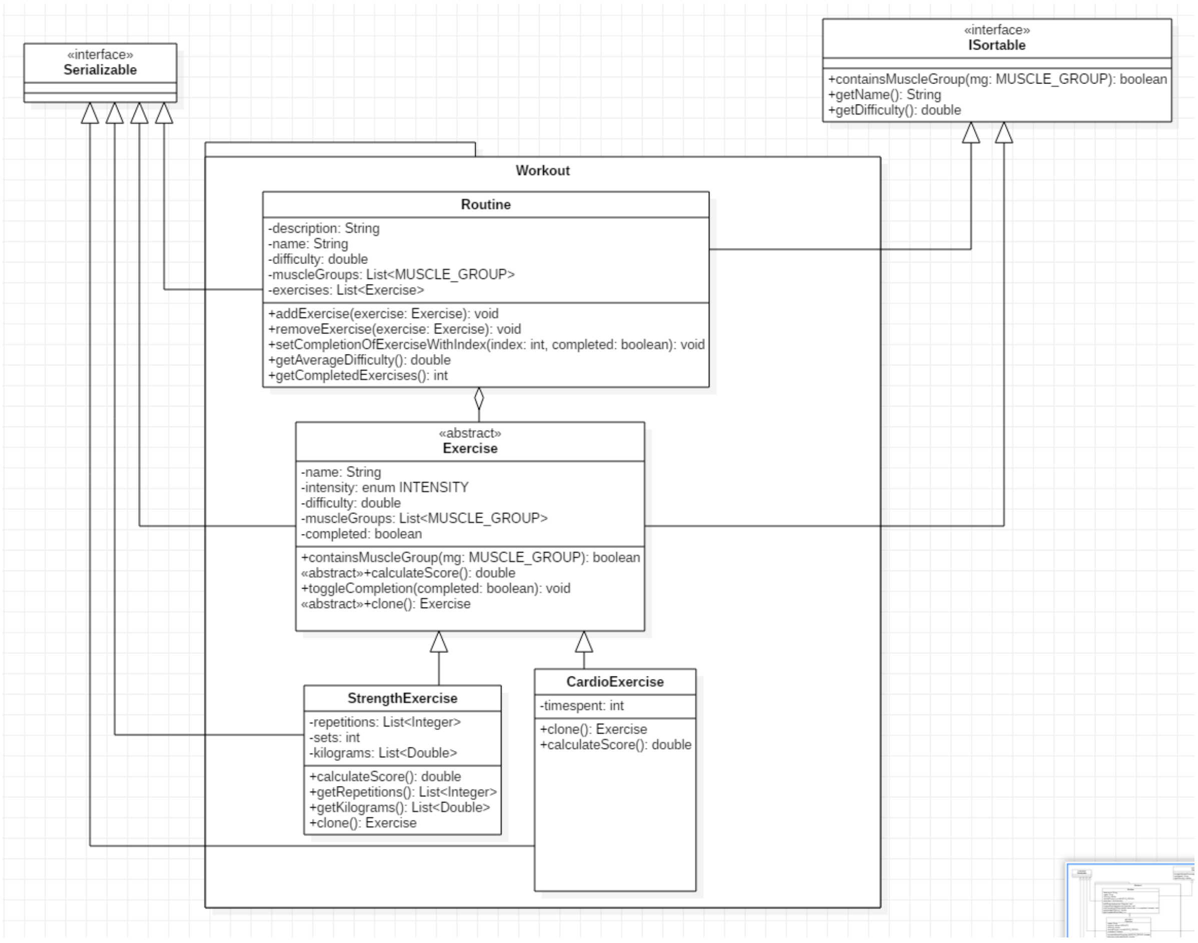
«interface»
Serializable
«interface»
ISortable
+containsMuscleGroup(mg: MUSCLE_GROUP): boolean
+getName(): String
+getDifficulty(): double
Workout
Routine
-description: String
-name: String
-difficulty: double
-muscleGroups: List<MUSCLE_GROUP>
-exercises: List<Exercise>
+addExercise(exercise: Exercise): void
+removeExercise(exercise: Exercise): void
+setCompletionOfExerciseWithIndex(index: int, completed: boolean): void
+getAverageDifficulty(): double
+getCompletedExercises(): int
«abstract»
Exercise
-name: String
-intensity: enum INTENSITY
-difficulty: double
-muscleGroups: List<MUSCLE_GROUP>
-completed: boolean
+containsMuscleGroup(mg: MUSCLE_GROUP): boolean
«abstract»+calculateScore(): double
+toggleCompletion(completed: boolean): void
«abstract»+clone(): Exercise
StrengthExercise
-repetitions: List<Integer>
-sets: int
-kilograms: List<Double>
+calculateScore(): double
+getRepetitions(): List<Integer>
+getKilograms(): List<Double>
+clone(): Exercise
CardioExercise
-timespent: int
+clone(): Exercise
+calculateScore(): double

Figure 5: Workout Package UML

2.7 DataStorage

This package contains logic for parsing, loading and saving user data. The package consists of two classes, LocalDatabase and Parser. LocalDatabase handles serializing and deserializing user objects. Parser translates premade routines and exercises JSON-data into objects during runtime.

2.7.1 LocalDatabase

The LocalDatabase class loads and saves a User object from a local file by using the built-in Serialized java interface. The application occasionally saves the current user, and LocalDatabase is the class that manages the save action. The user is loaded into the application in the beginning of the application lifecycle.

2.7.2 Parser

The Parser class parses JSON-data stored in files located in the 'res/raw' folder. Each JSON-object and its properties is sent to their respective class constructor and creates an object with matching fields. New premade routines and exercises can be created by adding new JSON-objects in these files.

2.7.3 Diagrams

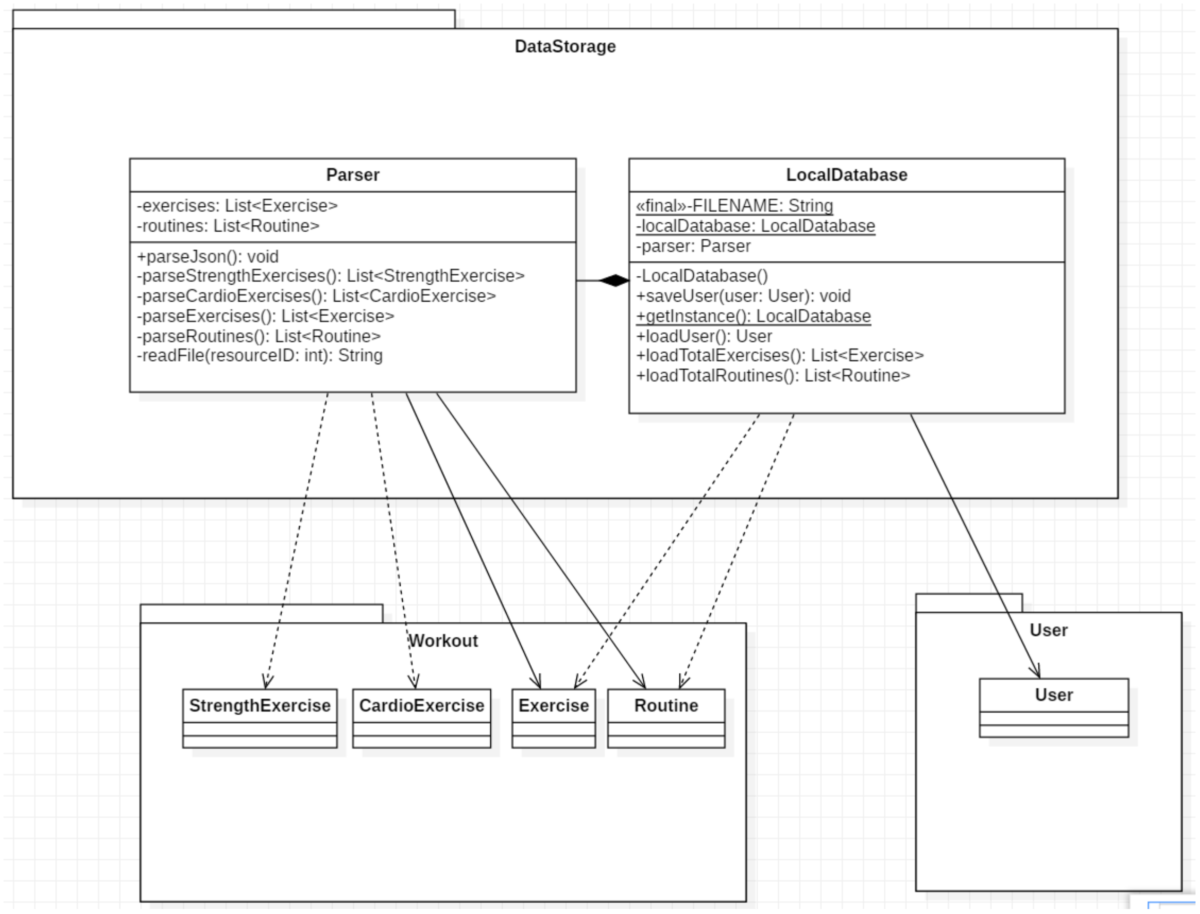
DataStorage
Parser
-exercises: List<Exercise>
-routines: List<Routine>
+parseJson(): void
-parseStrengthExercises(): List<StrengthExercise>
-parseCardioExercises(): List<CardioExercise>
-parseExercises(): List<Exercise>
-parseRoutines(): List<Routine>
-readFile(resourceID: int): String
LocalDatabase
«final»-FILENAME: String
-localDatabase: LocalDatabase
-parser: Parser
-LocalDatabase()
+saveUser(user: User): void
+getInstance(): LocalDatabase
+loadUser(): User
+loadTotalExercises(): List<Exercise>
+loadTotalRoutines(): List<Routine>
Workout
StrengthExercise
CardioExercise
Exercise
Routine
User
User

Figure 6: DataStorage Package UML

3 Persistent data management

The application stores user data by serializing a User object into a sequence of bytes in a local file. The object is then deserialized from the file into an object that can be used during the application's runtime. Here the built-in Serialized interface is used within User and its fields to enable such data management. The saved user is loaded when the application starts and thus initiates the state of the user. The user is saved on occassions when user data is modified. The class LocalDatabase is reponsible for saving and loading User objects, giving other classes the ability to access the user data.

The premade routines and exercises used by the application are stored in files with JSON-structure. The files are divided in "routines.json", "strength_exercises.json" and "cardio_exercises.json", each containing the respective content for each class. Each JSON-object consists of properties that are then sent in to an object constructor, with the use of the library 'gson'. The class responsible for parsing these files is called Parser.

All resource files, such as images, string values, layouts and user files, are located in the "res" folder inside the project's "main" folder. An essential part of the application's design consists of layout files in the folder "res/layout". These XML-files are named after their use in different application pages and they are used to create the intended layout design of a page.

4 Access control and security

Access control and security is not applicable to the current application, and the following sections explains the motivations as to why that is.

4.1 Access Control

The application does not have different roles, since the need of an admin is not necessary. It's not possible to cheat in the application, since the users themselves decide how they want to use the app and what they do affect no other users. There is no user interaction either, so there is no need for anyone the moderate how a user uses the application.

4.2 Security

Since accounts were never implemented, the applications security is entirely non-existent. No log in is needed after the application is started, and the user is put directly at the home page of the application. Information about the user is stored directly on their device, and almost none of that information could be considered sensitive and in need of encrypting.

5 References