Developers:
Demetri Reid-Rumbolt, Sohil Pramij, Zahin Hasan Prangon, Simanto Rahman

Comp 2005
Activity Tracker

Created by Group 10

Program Summary: The **Activity Tracker** program is a smart device companion which receives data from a file and can display that data as statistical run data for the *user*.

Check the github repository to clone on master branch
https://github.com/spramij/cs2005.git

USER CONTENTS:

1. INSTALLATION AND HOW TO RUN:

In order to execute the program download the executable file from the github repository and clone/download the project onto your local desktop. This will download the project and then open up the project into your IDE(Eclipse). Make sure the latest JDK is installed in your computer and set it to the right path and system path.
Then click on **Run the project** which will execute the App.java and then the login screen will appear.

Make sure to add the icons/pictures in your build in order to not break the program before executing the program. The CSV files in the local storage of the user can be used to import data for that specific user.

Whenever a profile is registered it will store a instance of that profile in the Data folder in the root directory which can be used later in order to run the program.

2. Use Cases:

- **1.** Account Management
  - **1.1** Account Creation
  - **1.2** Login
    - **1.2.1** Wrong Username/Password
  - **1.3** Switch Account
- **2.** Data Import
  - **2.1** Import Data
    - **2.1.1** Wrong data format
- **3.** Data Access
  - **3.1** View Data
    - **3.1.1** No Data
  - **3.2** View Sorted Data
- **4.** Data Manipulation
  - **4.1** Edit data
    - **4.1.1** Invalid entry
  - **4.2** Save changes
- **5**. Daily Challenges
  - **5.1** view challenges
- **6.** Social
  - **6.1** View friends

6.1.1 No friends

2.1 Use Case Description:

1. Account Management
Brief Description: Actor creates an account if they don’t have one, otherwise they login or switch to another account.
Actors: The *user* of the application.
Preconditions: The actor launches the application.
Basic Flow: **1.1** If the actor does not already have an account they will be prompted to create one, filling in all essential profile information **1.2** *actor* logs into their profile using the information they have provided in the account creation process. **1.3** If a *actor* is signed in but would like to use another profile on the same system they can logout and log back in under a different username.
Alt Flow: **1.2.1** when the *actor* enters the wrong wrong information they will be told of the error and prompted to try again.
Postconditions: The *actor* has either created an account, signed in to their profile or has switched accounts.

2. Data Import

Brief Description: The actor imports data from a device/file
Actors: The *user* of the application.
Preconditions: Actor is signed in and has selected the option to import data, and has a file with rundata or a connected device.
Basic Flow: 2.1 The *actor* has connected a device (or entered a csv) which contains information about their runs, the system imports the data.
Alt Flow: 2.1.1 the data is in the incorrect format and is not imported into the system.
Postconditions: data from the file/device is brought into the system.

3. Data Access
Brief Description: The actor views their converted run data either in full or sorted by a specific time slot.
Actors: The *user* of the application.
Preconditions: Actor selects option to view data, and has imported data to be displayed.
Basic Flow: **3.1** after selecting an option to view their data, the Actor can see their runs as imported from their device/file. **3.2** *Actor* can view the averages for a specific time slot.
Alt Flow: 3.1.1 The Actor has not imported any runs and no runs are shown on screen.
Postconditions: Data is displayed by the system.

4. Data Manipulation
Brief Description: The Actor edits and saves the edit to a run.
Actors: The *user* of the application.

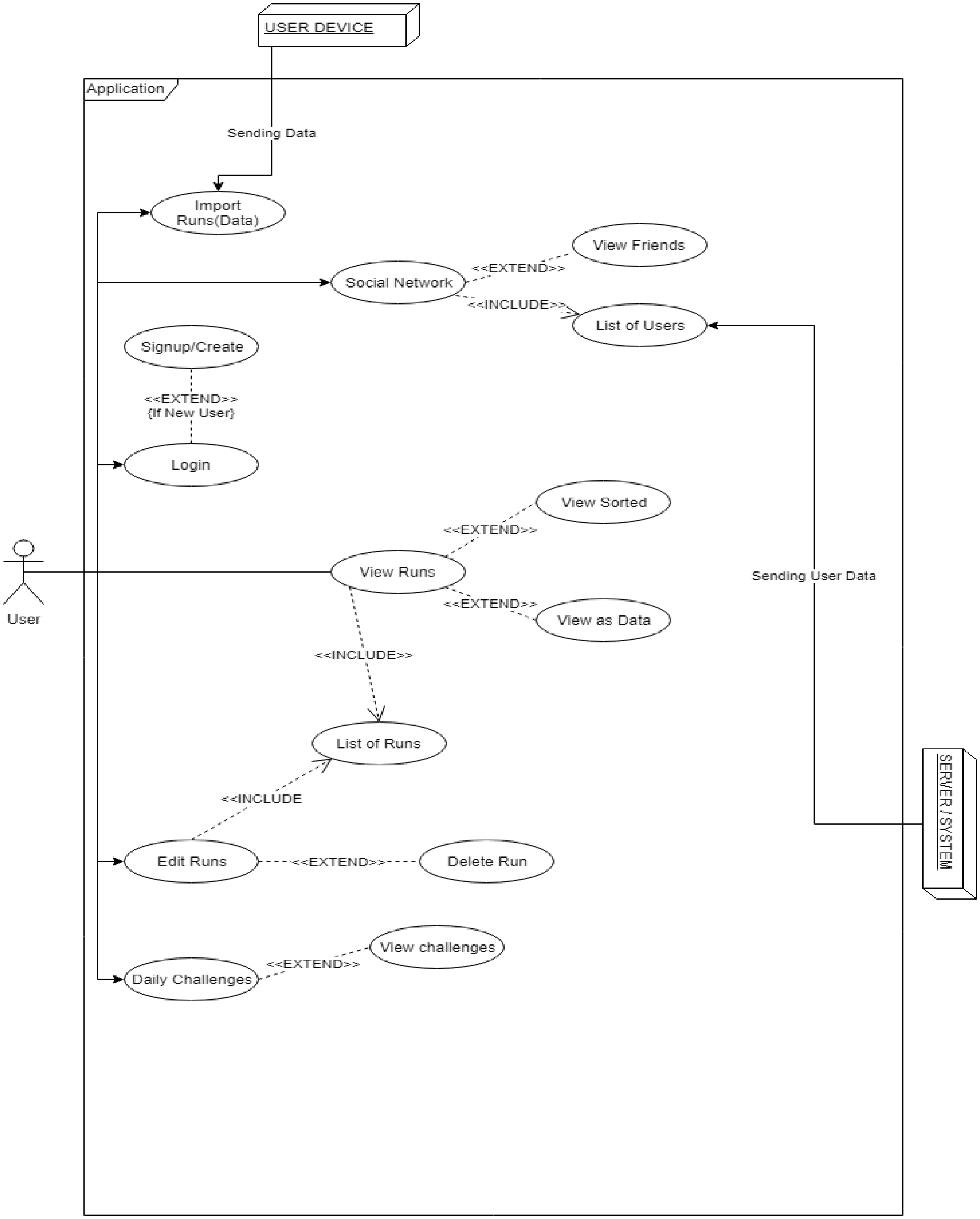
Preconditions: Actor selected an existing run that they wish to make changes to.
Basic Flow: **4.1** If the *Actor* wishes to change the data which is being displayed they can override existing data points to more match their needs. **4.2** If the *Actor* is happy with their entry they can permanently save it.
Alt Flow: 4.1.1 If the *actor* inputs an invalid data type, the entry will not change.
Postconditions: A run is edited and permanently saved.

5. Daily Challenges
Brief Description: The Actor views the challenges that they can work toward
Actors: The *user* of the application.
Preconditions: Actor selected option to view their challenges
Basic Flow: **5.1** The *actor* may view challenges that they can work towards.
Postconditions: The system displays the Actors challenges.

6. Social
Brief Description:
Actors: The *user* of the application.
Preconditions: The Actor selects the option to view their friends.

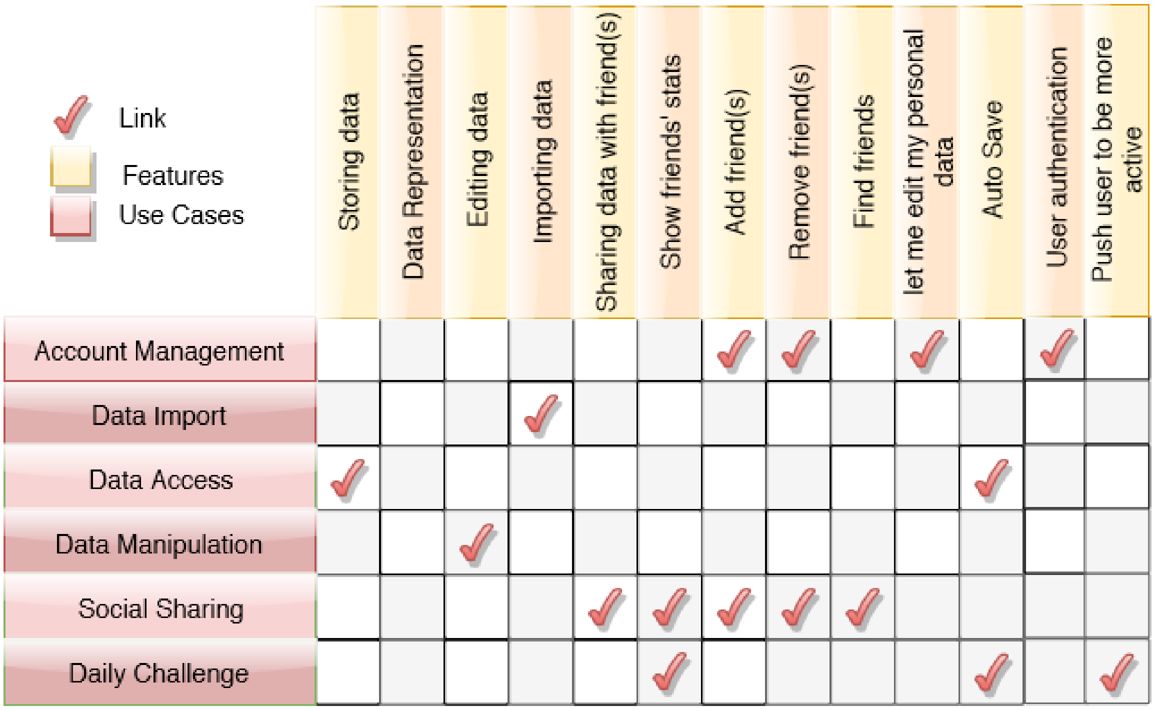
Basic Flow: **6.1** The *Actor* is able to view a list of friends.
Alt Flow: 6.1.1 The Actor has no friends and the system displays nothing
Postconditions: The system displays the Actors friends.

3. Use Case Diagram



4. Traceability Matrix

A traceability matrix is a document, usually in the form of a table, used to assist in determining the completeness of a relationship by correlating any two baselined documents using a many-to-many relationship comparison.



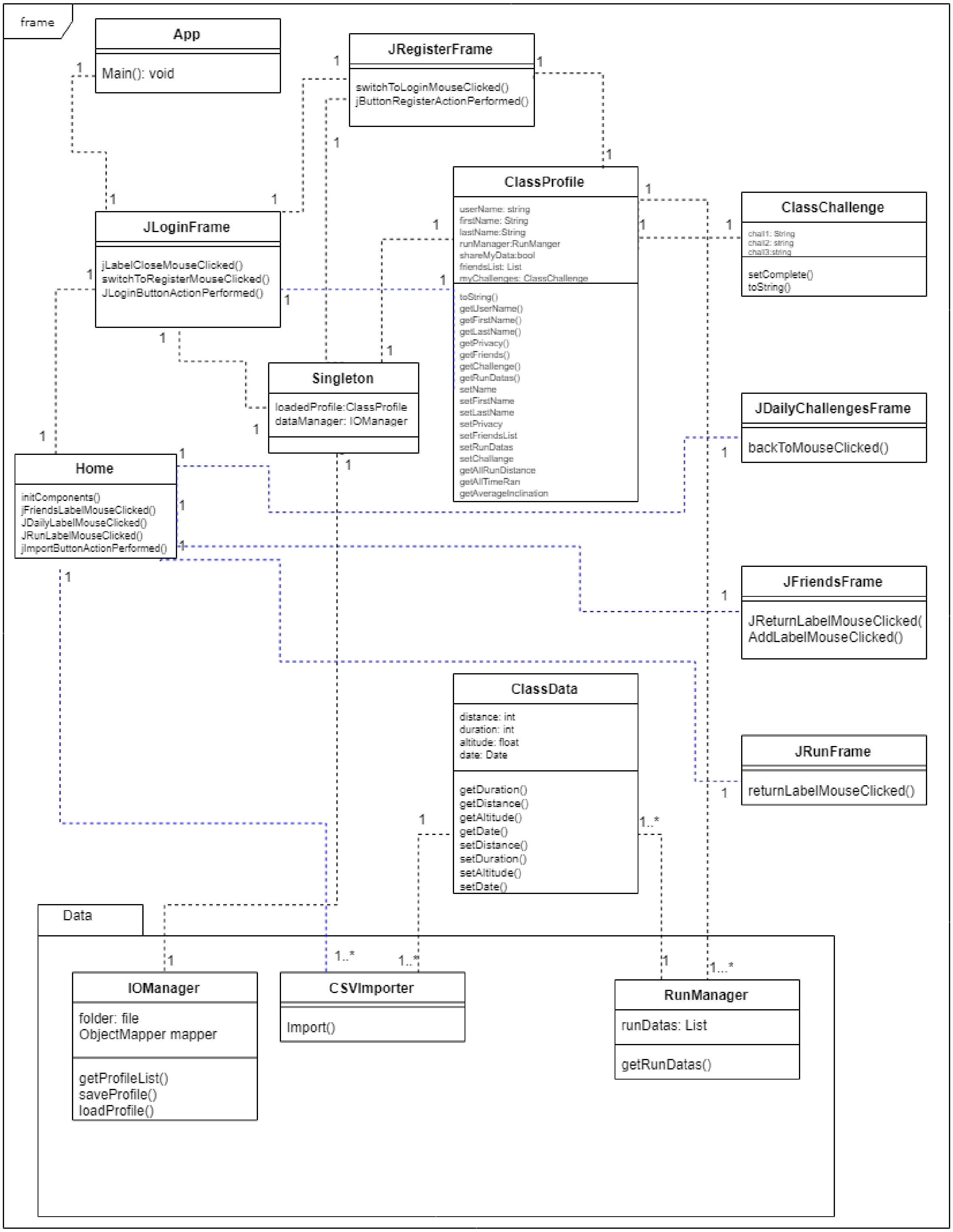
✓ Link

🟨 Features

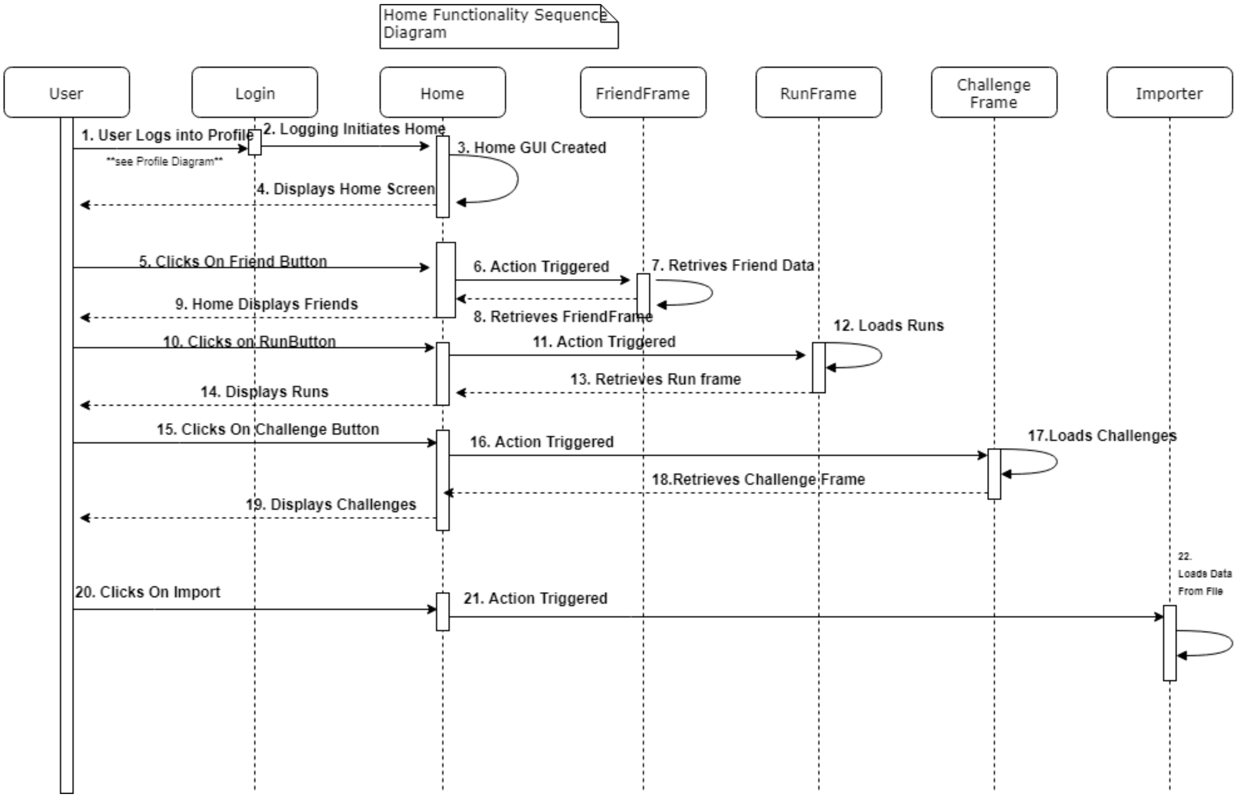
🟥 Use Cases

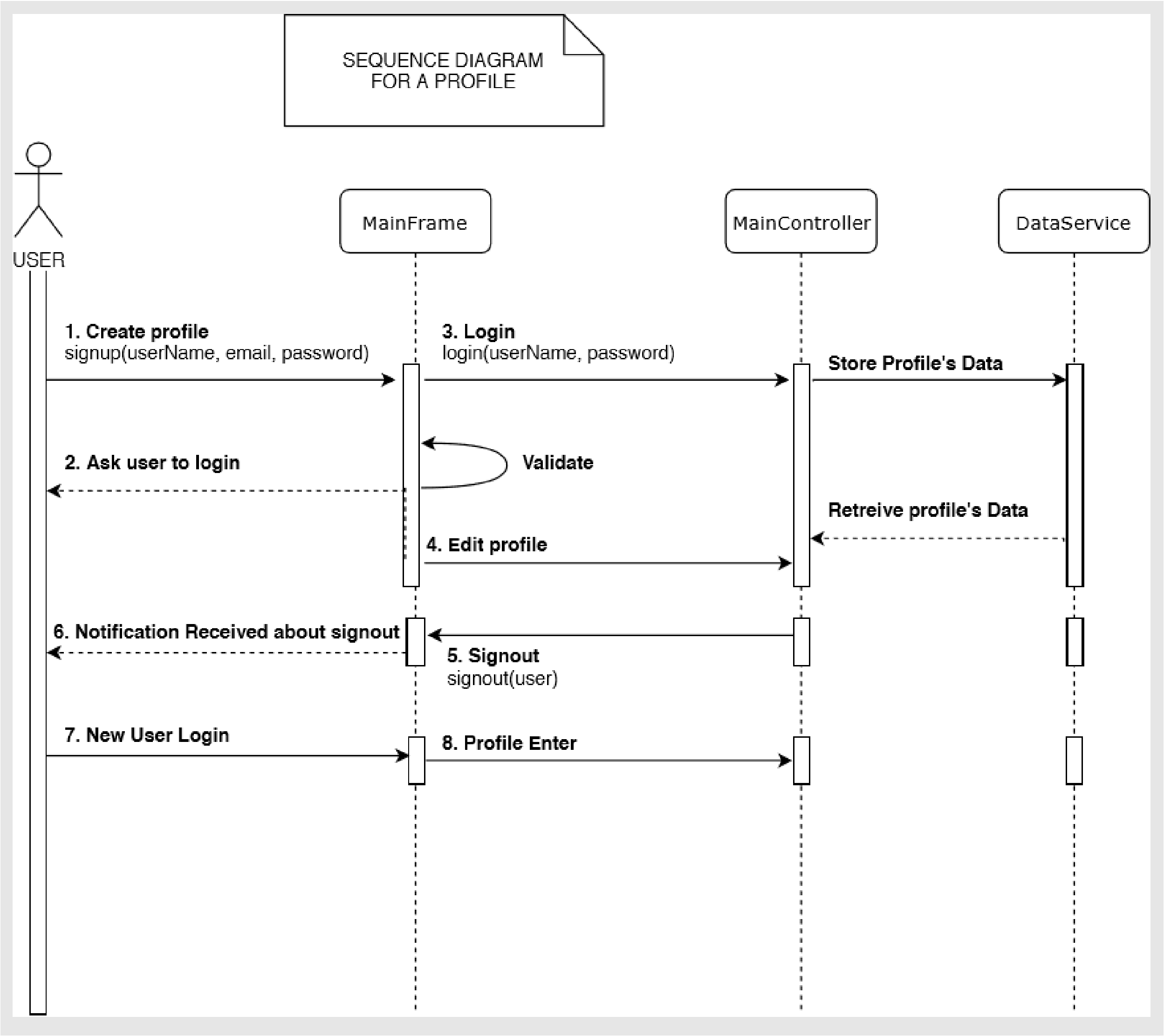
	Storing data	Data Representation	Editing data	Importing data	Sharing data with friend(s)	Show friends' stats	Add friend(s)	Remove friend(s)	Find friends	let me edit my personal data	Auto Save	User authentication	Push user to be more active
Account Management							✓	✓		✓		✓	
Data Import				✓									
Data Access	✓										✓		
Data Manipulation			✓										
Social Sharing					✓	✓	✓	✓	✓				
Daily Challenge						✓					✓		✓

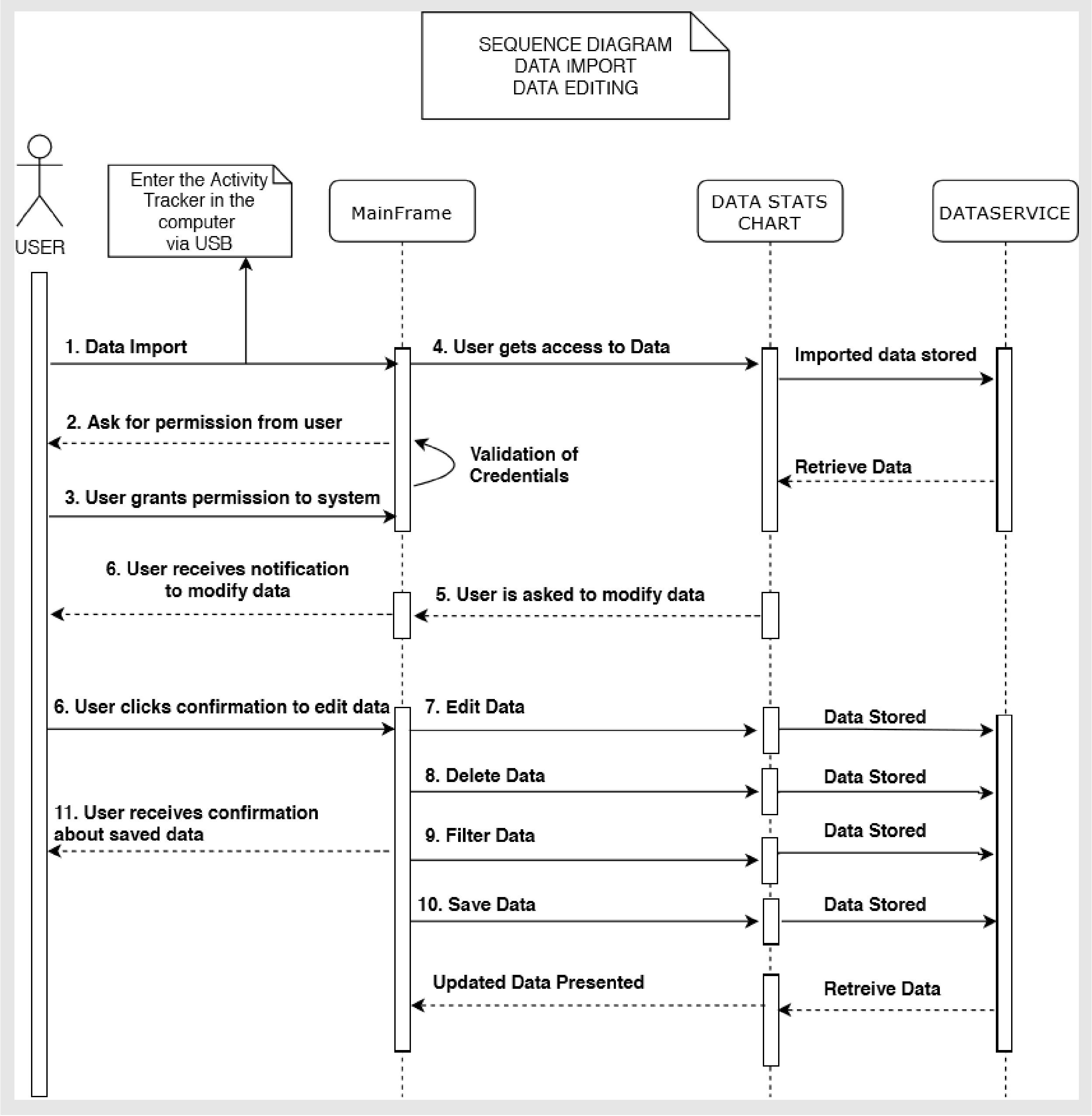
5. Domain/UML Model

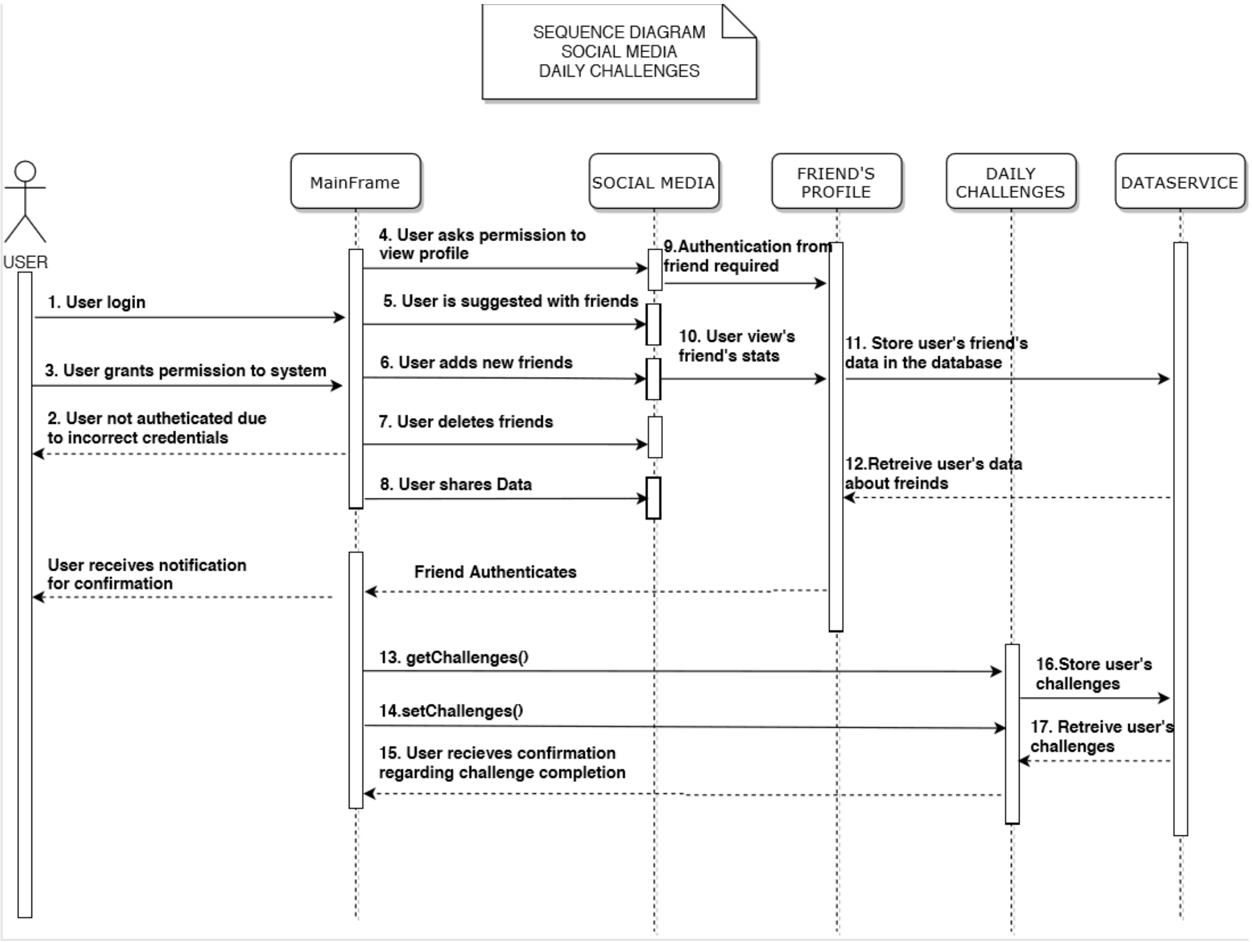

frame
App
Main(): void
JRegisterFrame
switchToLoginMouseClicked()
jButtonRegisterActionPerformed()
JLoginFrame
jLabelCloseMouseClicked()
switchToRegisterMouseClicked()
JLoginButtonActionPerformed()
ClassProfile
userName: string
firstName: String
lastName:String
runManager:RunManger
shareMyData:bool
friendsList: List
myChallenges: ClassChallenge
toString()
getUserName()
getFirstName()
getLastName()
getPrivacy()
getFriends()
getChallenge()
getRunDatas()
setName
setFirstName
setLastName
setPrivacy
setFriendsList
setRunDatas
setChallange
getAllRunDistance
getAllTimeRan
getAverageInclination
ClassChallenge
chall1: String
chall2: string
chall3:string
setComplete()
toString()
Singleton
loadedProfile:ClassProfile
dataManager: IOManager
Home
initComponents()
jFriendsLabelMouseClicked()
JDailyLabelMouseClicked()
JRunLabelMouseClicked()
jImportButtonActionPerformed()
JDailyChallengesFrame
backToMouseClicked()
JFriendsFrame
JReturnLabelMouseClicked(
AddLabelMouseClicked()
ClassData
distance: int
duration: int
altitude: float
date: Date
getDuration()
getDistance()
getAltitude()
getDate()
setDistance()
setDuration()
setAltitude()
setDate()
JRunFrame
returnLabelMouseClicked()
Data
IOManager
folder: file
ObjectMapper mapper
getProfileList()
saveProfile()
loadProfile()
CSVImporter
Import()
RunManager
runDatas: List
getRunDatas()
1
1..*
1...*

6. Sequence Diagrams


Home Functionality Sequence Diagram
User
Login
Home
FriendFrame
RunFrame
Challenge Frame
Importer
1. User Logs into Profile
see Profile Diagram
2. Logging Initiates Home
3. Home GUI Created
4. Displays Home Screen
5. Clicks On Friend Button
6. Action Triggered
7. Retrives Friend Data
8. Retrieves FriendFrame
9. Home Displays Friends
10. Clicks on RunButton
11. Action Triggered
12. Loads Runs
13. Retrieves Run frame
14. Displays Runs
15. Clicks On Challenge Button
16. Action Triggered
17.Loads Challenges
18.Retrieves Challenge Frame
19. Displays Challenges
20. Clicks On Import
21. Action Triggered
22. Loads Data From File


SEQUENCE DIAGRAM
FOR A PROFILE
USER
MainFrame
MainController
DataService
1. Create profile
signup(userName, email, password)
3. Login
login(userName, password)
Store Profile's Data
Validate
2. Ask user to login
Retreive profile's Data
4. Edit profile
6. Notification Received about signout
5. Signout
signout(user)
7. New User Login
8. Profile Enter


SEQUENCE DIAGRAM
DATA IMPORT
DATA EDITING
USER
Enter the Activity Tracker in the computer via USB
MainFrame
DATA STATS CHART
DATASERVICE
1. Data Import
4. User gets access to Data
Imported data stored
2. Ask for permission from user
Validation of Credentials
Retrieve Data
3. User grants permission to system
6. User receives notification to modify data
5. User is asked to modify data
6. User clicks confirmation to edit data
7. Edit Data
Data Stored
8. Delete Data
Data Stored
11. User receives confirmation about saved data
9. Filter Data
Data Stored
10. Save Data
Data Stored
Updated Data Presented
Retreive Data


SEQUENCE DIAGRAM
SOCIAL MEDIA
DAILY CHALLENGES
USER
MainFrame
SOCIAL MEDIA
FRIEND'S PROFILE
DAILY CHALLENGES
DATASERVICE
1. User login
2. User not autheticated due to incorrect credentials
3. User grants permission to system
4. User asks permission to view profile
5. User is suggested with friends
6. User adds new friends
7. User deletes friends
8. User shares Data
9.Authentication from friend required
10. User view's friend's stats
11. Store user's friend's data in the database
12.Retreive user's data about freinds
User receives notification for confirmation
Friend Authenticates
13. getChallenges()
14.setChallenges()
15. User recieves confirmation regarding challenge completion
16.Store user's challenges
17. Retreive user's challenges

7. Modules

Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as modules. A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements.

- Social package: Contains the classes that are used to get access into the social media and is directly used in the application.

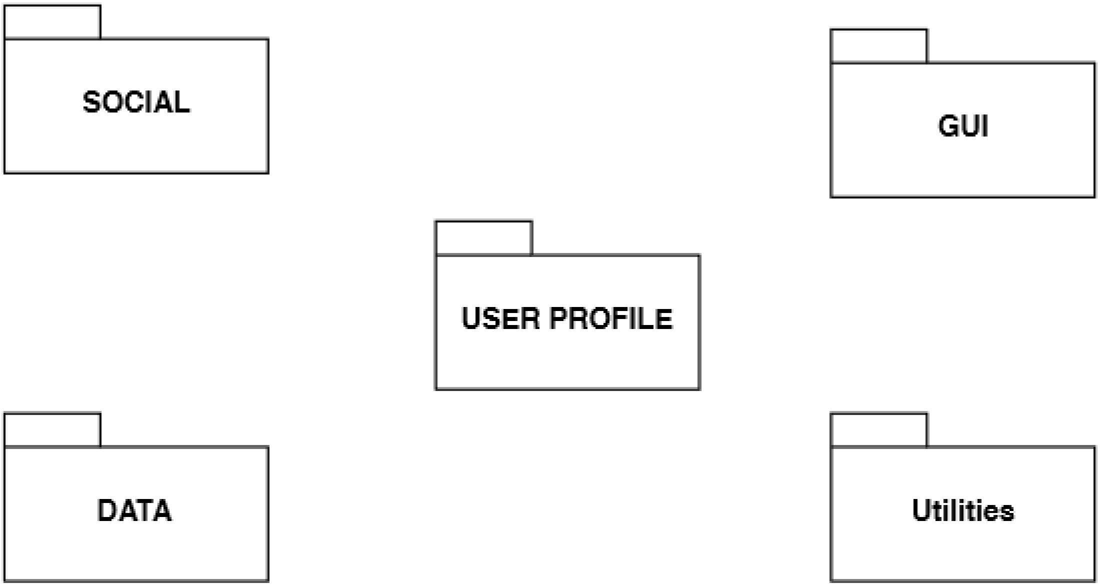
- Data package: Classes used to store imported data and view the statistics of user’s activity. Data when modified will also update in the classes accordingly

- User Profile package: Classes used to store the client’s profile and login credentials are added and updated here

- GUI Package: The imported libraries used to build the application’s interface goes here and the blueprint of the package can be viewed from here

- Utilities: Classes that provide generic behavior used by the other packages go here

MODULES



8. Design Analysis

Software architecture is the organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles and decisions that you use in the design of a system.

To ensure that key features and architecture is maintained and reusable across the entire lifespan of the application, strong design pattern will be essential. For this reason implementation of Builder Design Pattern. This would allow us fine tuned control over extension of the application and maintain interconnectivity between functionality and code bases. While defining all the priority classes in abstract form, we allow room for upgrades to existing functionality and new functionality without breaking the previous code base as the Abstract classes and Abstract Builders will remain the same, only the implementations will extend.

The major decision regarding the design process includes the prioritization of the creation of an Profile class, which is of highest priority and the essence of the Activity Tracker Application. The Profile class in relation to the database we are planning to implement whether it be a Backend data service or a java class for storing and retrieving data for profiles. With profile data separated from the profile class allows ease with debugging issues for profile retrieval.

The app.java class is used as a entry point for the program to execute. The Login screen allows an existing user to login and if not then lets the user to register using the first and last name along with a username which the user will use later to login. When clicked the login button later it takes user to a home screen where the user can navigate through several features of the application which includes adding new friends, removing friends and viewing the data of the permission. The user can then also check the progress of their daily challenges provided by the user.

Due to the commonality between the activities, we have chosen to design one class for Activity that contains all attributes related to a workout. The addition of activities other than running can be added by making a instance of the Activity class. This increases the flexibility of the software design to extend in the future by not repeating ourselves which is straight-adaption of the principle: Do Not Repeat Yourself. The activity data is saved to the DataService class where it is linked to a profile. The DataService class computes the calculations to display the statistics and then returns to the profile class.

Another design approach which follows the object-oriented design includes using of a single class named Data.java. All the activities have the same statistics; the only difference is the type of activity that was performed. Due to the commonality between the activities, we have chosen to design one class for Activity that contains all attributes related to a workout. The addition of activities other than running can be added by making a instance of the Activity class. This increases the flexibility of the software design to extend in the future by not repeating ourselves which is straight-adaption of the principle: Do Not Repeat Yourself. The activity data is saved to the DataService class where it is linked to a profile. The DataService class computes the calculations to display the statistics and then returns to the profile class.

9. Functionalities of All Classes:

- **App:** The *App* class is the initiates the *JLoginFrame* class using its main() method, this initiation allows for the program to begin.

- **JLoginFrame:** Class creates a frame containing all elements that would be necessary to login, or if user does not have an existing profile *JLoginFrame* has the method switchToRegisterMouseClicked() which switches over to *JRegisterFrame* for account creation. If the user does have an existing Profile they would type its username into the displayed text field and click the “Login” button, this action triggers JLoginButtonActionPerformed() which, using the *Singleton* class, compares the inputted username with the usernames of all profiles in the system and creates an instance of the *Home* class with all relevant information for the Profile.

- **JRegisterFrame:** Creates a frame with text fields retaining to crucial profile information i.e userName, firstName, lastName. Once the information is entered and the user clicks the button to create their account this triggers jButtonRegisterActionPerformed() method and creates a Profile object with all of these attributes. The class now loads back to *JLoginFrame* so the use may login.

- **ClassProfile:** This class handles the creation and retrieval of all attribute pertaining to a Profile object using their respective set__() and get__() methods. The class also has attributes that are not directly entered by the user and can be both created and retrieved by the system such as, Challenges, Friends List and a List of Runs. The class also takes a list of run data points and converts them in statistical run data.

- **Home:** The first frame displayed to the user after logging in, displayed in the center is the “Import” button which when pressed activates jImportButtonActionPerformed() and triggers the *CSVImporter* class and allows the user to select the .csv files they’d wish to bring into the system. The other buttons such as runs, friends and challenges when pressed would trigger the *JRunFrame*, *JFriendsFrame* and *JDailyChallengesFrame* respectively.

- **ClassData:** Retrieves data from the importer and creates individual data points for a run, these data points can be loaded into the *runManager* class which adds them to a list so that they can then be added to a Profile to be converted.

- **JRunFrame:** Loads a frame containing the Run information and statistics relating to the logged in profile. It also contains a sorting feature to view run statistics for a specific time frame.

- **JDailyChallengesFrame:** Creates 3 seperate challenges with progress bars for the user to work towards.

- **JFriendFrame:** Contains methods to add and remove profiles to a Profiles friendsLIst. Uses the *Singleton* class to check the userNames of all existing profiles on the system .

- **Singleton:** With the help of the *IOmanager* class the method loadedProfile() allows the system to access all Profiles that exist in the system.

- **ClassChallenge:** Contains methods to get and set challenges objects, which are strings, and a toString() method to return all 3 challenges.

10. Design Patterns and Principles

The design patterns used in this program includes several inspired from the textbook covered in the course. The activity tracker application uses principles which includes

Open Close Principle:

Software entities like classes, modules and functions should be open for **extension** but closed for **modifications**.

The ClassProfile.java is programmed using the OCP principle which is a generic principle allowing extension by creating a profile class everytime a new user logins into the system which is not modified on the ClassProfile.java but stored in the Data folder instead. The ClassData.java is also programmed in such manner which will only be open for extension by storing the data but not modifying the profile.

Do not Repeat Yourself:

A principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions or using data normalization to avoid **redundancy**.

This design principle allows the application to avoid redundancy by creating providing an abstraction and using it later as an instance. The singleton class uses the DRY principle by creating additional instance of the file rather than redundant code.

Single Responsibility Principle:

The SRP is used by the classes profile, data and app which serves a single purpose in the application

Singleton Design Pattern

We used Singleton Design Pattern for data persistence across the program. Any data that needs a single instance to work goes under ***Singleton*** class which keeps only one instance of each of the kind of objects that is needed. This includes:

- ❏ The currently loaded profile.
- ❏ The DataManager class that handles saving and loading of profiles.
- ❏ Importer that is responsible for importing run data.

Adapter Design Pattern

Using Decorator Design Pattern, we wrapped up functionalities in a way that would allow us to wrap up and call functionalities from internal instances of other objects. To make things modular, we provided modular classes for each modular functionality and using Decorator pattern we interconnected functionalities without having to actually go down a rabbit hole of intances. For example, run data is managed by the class ***RunManager*** which holds and provides the functionalities related to run data. But as a Profile should contain the reference to the ***RunManager*** that itself uses, profile provides wrapper methods for all the methods in ***RunManager***. So instead of:

```
ClassProfile.runManager.getRunDatas();
```

We can actually access the same functionality through:

```
ClassProfile.getRunDatas() {
      return this.runManager.getRunDatas();
}
```