RU Healthy?

RU Healthy?
REPORT 2:

Full Report #2

Class
ECE 567 – Software Engineering 1

Semester
Fall 2017

Group #2
Tahiya Chowdhury, Tina Drew,
George Koubbe, Aymen Al-Saadi,
Himabindu Paruchuri and Ramya Tadepalli

Github Repository
[RU Healthy?]

1 Effort Breakdown

Report	Aymen	George	Tahiya	Tina	Himadindu	Ramya
Interaction Diagrams						
UML Diagrams (10)	50%	50%				
Prose description of diagrams (10)					50%	50%
Alternate Solutions (10)			50%	50%		
Class Diagram and Interface Specification						
Class Diagrams (4)				100%		
Data Types and Operation Signatures (5)	50%	50%				
Traceability Matrix (1)				50%	50%	
System Architecture and System Design						
Architectural Styles (5)						100%
Identifying Subsystems (2)			100%			
Mapping Subsystems to Hardware (2)			100%			
Persistent Data Storage (3)					%100	
Network Protocol (1)	50%	50%				
Global Control Flow (1)	50%	50%				
Hardware Requirements (1)	50%	50%				
Algorithms and Data Structures						
Algorithm	20%	20%	20%		20%	20%
Data Structures	20%	20%	20%		20%	20%
User Interface and Design						
User Interface and Design	20%	20%	20%		20%	20%
Design of Tests						
Design of Tests	20%	20%	20%		20%	20%
Project Management						
Merging Contributions	20%	20%	20%		20%	20%
Project coordination and Progress report				100%		
Plan of work				100%		
Breakdown of responsibilities				100%		
References	100%					

2 Interaction Diagrams

2.1 UML Diagrams

When drawing these more detailed diagrams, we always tried to follow the most important design principles at the objects level: *Expert Doer, High Cohesion* and *Low Coupling*.

<u>UC-2: Health Data</u>

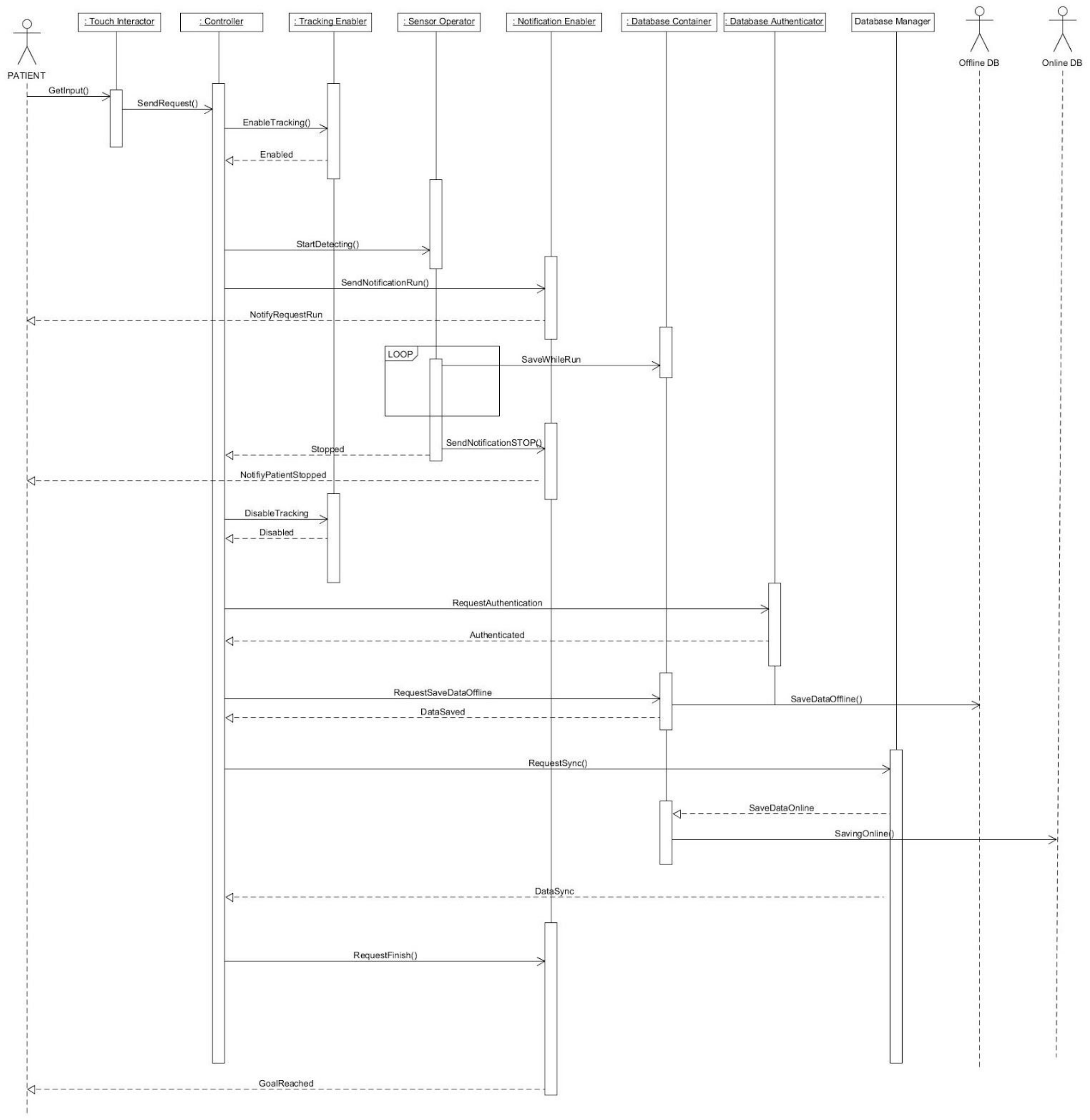


Figure 2.1.1 Health Data

The figure above shows the diagram for the main use case UC-2: Health Data, which monitors the Patient's activity. While sketching the design, we realized that we were missing a conceptual software object that allows direct interaction with the Patient. Thus, we added the concept *"Touch Interactor",* and assigned it the said responsibility. The Patient then touches the screen to start, and *Touch Interactor* communicates this to *Controller*. Even though *Controller* seems to handle several communications, we favored the Expert Doer principle for it because after all, Controller needs to know everything at all times to be able to delegate tasks effectively. If the Patient wants to start monitoring his activity, *Controller* asks *Tracker Enabler*, who acts like a switch, giving permission for *Sensor Operator* to start detecting. *Controller* then allows *Notification Enabler* to start sending notifications to the Patient about his current activity.

By the Expert Doer principle, we keep observing that every concept has its specific task, and they only communicate through *Controller*, in accordance with the Low Coupling principle. We analyzed the option of letting *Controller* perform the *Tracking Enabler* task, but in order to keep a High Cohesion, we ended up using the former.

Sensor Operator is going to continuously obtain the data and send it to *Database Container*. When the exercise finishes, the loop is over and Controller is notified, along with the Patient. Another option here is to let Notification Enabler communicate directly with *Tracking Enabler*, but *Controller* needs to know that the activity finished anyway, and to favor the design principles, so we decided not to go with the option.

Finally, *Controller* starts the process of authenticating and storing the data to the Offline Database, and request syncing online. When this is done, the Patient gets notified for the last time that all of his information was saved online.

UC-3: GetVitalSigns - Phone

The following figure explains the interaction between concepts for the next important use case, UC-3: Get Vital Signs Phone. Here, the main goal is to allow the Patient retrieve their information from the local database, so he/she can access his past activities and records.

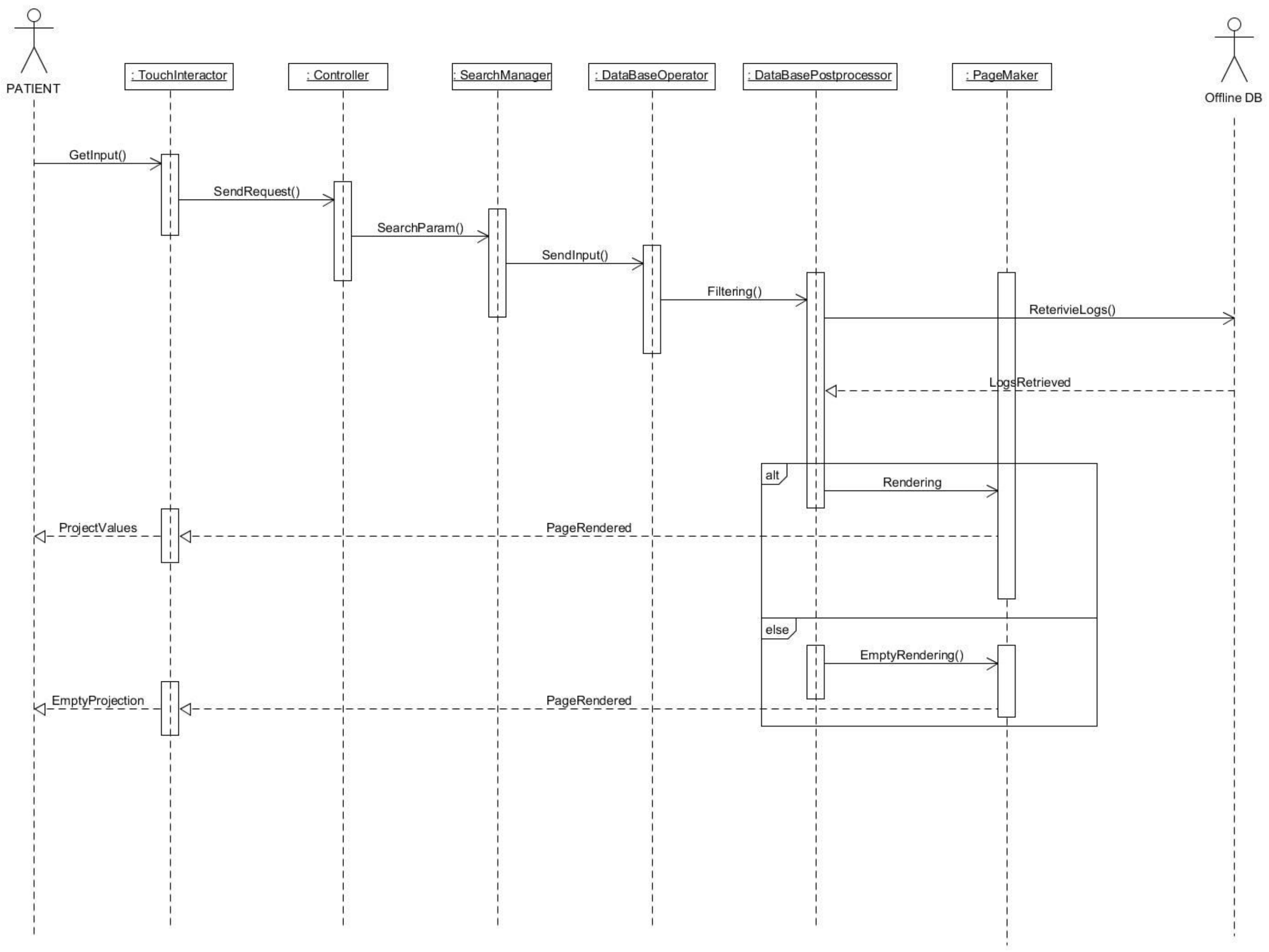


Figure 2.1.2 Get Vital Signs - Phone

UC-4: GetVitalSigns - Web

Essentially, the design for this diagram is the same as UC-3. The difference is that now, the actors are going to be Physician and Online Database, rather than Patient and Local Database.

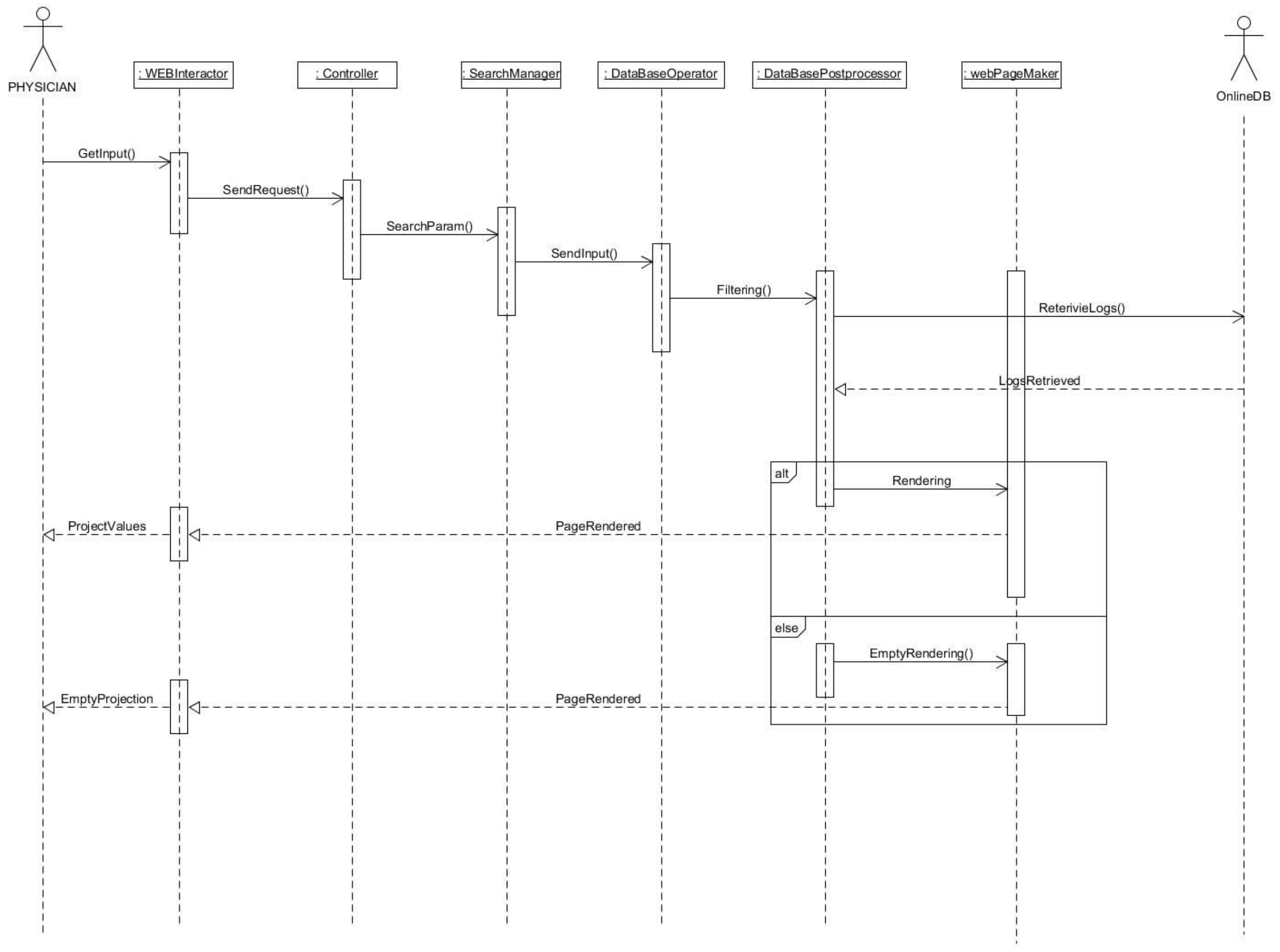


Figure 2.1.3 Get Vital Signs - Web

UC-11: Physician Sign in

The following figure explains the interaction between concepts for UC-11: Physican Sign In. Here, the main goal is for the doctor to sign in and edit the database by adding or removing patients.

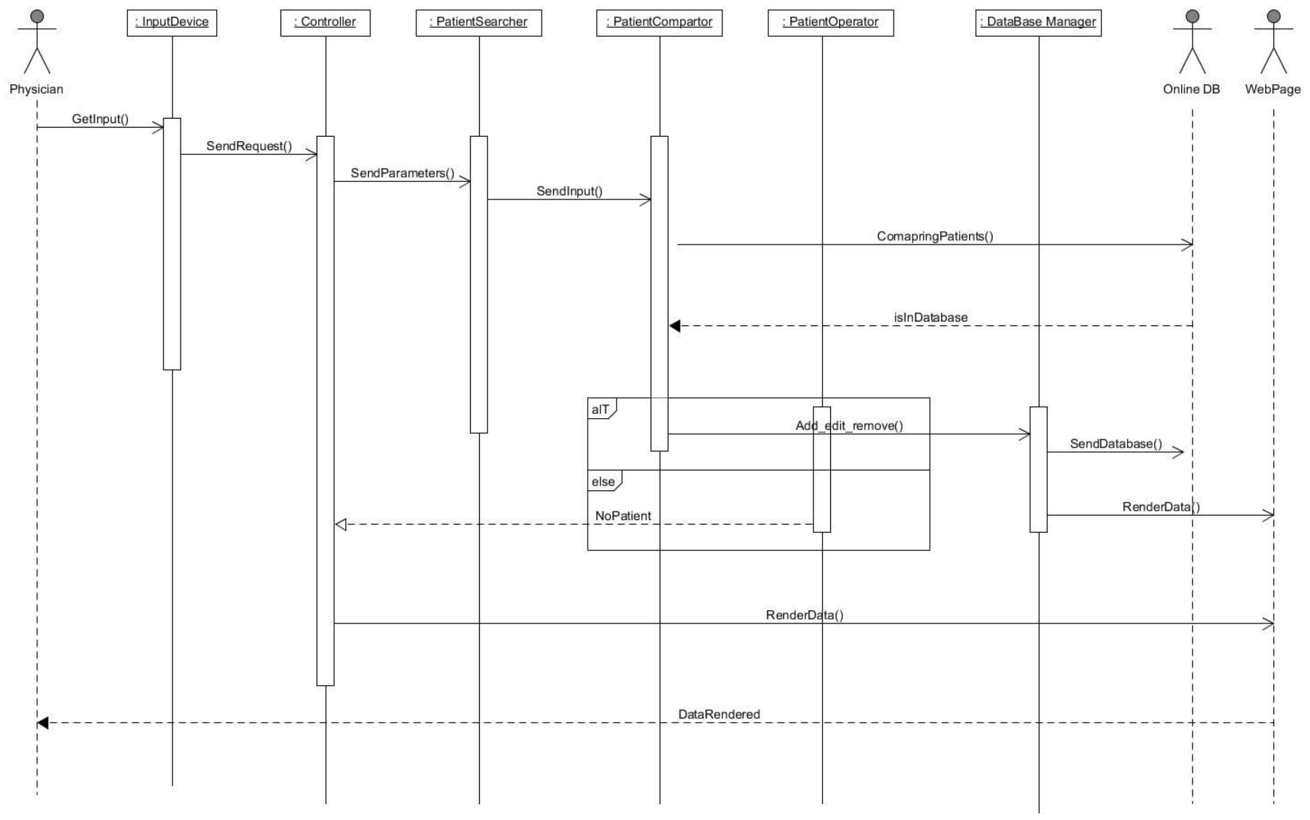


Figure 2.1.4 Physician Sign In

2.2 Prose Description of Diagram

Use Case-2 :Health Data

The user can monitor his/her current activity using the mobile application. The user/patient opens the application and initiates the tracking function. When initiated, the Touch Interactor gets the input from the user and sends corresponding request to the Controller. The Controller then notifies the Tracking Enabler to start tracking. It also requests the Sensor Operator to start detecting using the sensor and simultaneously sends request to Notification Enabler to notify the patient to start running. The Sensor Operator saves the data in the Data Container in loop. When the patient stops running, the sensor operator sends Stop notification to Notification Enabler and the Controller send disable tracking request to the Tracking Enabler. The Controller sends authentication request to the Database Authenticator to verify user authenticity and after the authentication is done, the controller requests to save data offline from the Data Container in the Offline Database.The Database Manager ensures that the user data is synced and updated in the Online Database, when it receives a sync request from the controller. Also, The Controller sends a finish request to the notification Enabler , when a specific goal is reached and displays the corresponding message to the patient.

UC-3 : GetVitalSigns - Phone

The User may wish to see his current health data or history of his previous activity. This Use Case refers to the historic data of the user. The User opens the app and presses the History button to view his previous activity details. The Touch Interactor then talks to the Controller to retrieve the User's records. The Controller sends in basic User information which serves as a Search criteria in the database. The Database Operator then queries the Database for the User's records using this search criteria. After the information has been obtained from the database, the information is assigned to specific components by the Postprocessor and the PageMaker is an xml file that determines how the Display page will look like. This is then available for the user to view. We have used a Publisher-Subscriber Design Pattern here with no specific central controller. The control flow traverses from object to object. Even though we do have a controller, it does not have direct communication with all components and each part of our code is reusable.

UC-4: GetVitalSigns - Web

This Use Case is similar to UC-3, in that most components interact in the same way. The Physician must be signed in to use this functionality. The Physician can only view records of patients he is authorized to access. Hence, his login is authenticates and provides access to certain records in the Online database. The Physician requests for patient history by clicking on the appropriate button. The Controller then sends the basic patient data (for example, ID) serving as a search criteria to obtain information from the Online database, which is then displayed to the user. The design pattern and control flow are the same as described in UC-3.

Use Case 11: Physician Sign In

This use case describes how the physician can manage patient records by performing operations like add/edit/remove. The physician is assumed to be signed into his/her account and view the web page.The Physician sends the input to the Input device which in turn sends the corresponding request to the Controller. The Controller sends the input parameters to the Patient Searcher. The Patient Searcher in turn sends the input to Patient Comparator which compares the input with the Online Database and returns the required response back to the Comparator. After checking the Database for the patient records, the Controller sends the add/edit/remove request to the Patient Operator. Then Patient Operator sends the commands to the Database manager and renders the data to the Web Page for display.

Design Patterns and Responsibilities:

We aim to implement software that is reusable as individual components and each component is more or less independent, in a way that changes made in one component do not heavily affect a lot of other components. In other words, we plan to minimize dependencies between objects. This helps us in not only dividing the work between members of the group, it also helps us when we put together all the components, and when we perform Software Testing. Since, we will also continue modifying and refactoring the code to provide all the functionalities we proposed, having this kind of design principles will be more convenient.

The most important components of our software are the Controller and Database Manager. This does not mean that most work is done by these two components; it simply means that since the controller initiates most functionalities and since the database manager is required for interaction with our Database, and helps in communication between our Web and Mobile app, they have some key roles and most number of interactions with other objects. We have, however, decentralized the control flow and have assigned specific roles to other components. For example, the Sensor Operator performs the function of getting information from the sensors and passes it onto a Data Container object to store it. This in turn talks to the Database Manager to help store this information in the Online Database. Similarly, we have assigned specific roles to other objects. With such requirements, we have planned to implement our software mainly using the design principles of Expert Doer - to have short communication links between objects, High Cohesion - decentralized flow of control with each component having its own specific role which in turn supports Low Coupling - having minimum number of interactions for each component.

2.3 Alternate Solution Description

2.3.1 Basic Design Process

While designing the flow of interactions between the concepts of the system, it is a common practice to follow responsibility-driven approach. This gives a way to assign the responsibilities among the elements of the design in a balanced, fluid and optimal way. While designing ours, we considered a design to be optimal that manages to articulate the interaction flow while maintaining the following design principles [15]:

- Expert Doer Principle - Short communication chains between the objects
- High Cohesion Principle - Balanced workload across the objects
- Low Coupling Principle - Low degree of connectivity among the objects

At global design level, we also considered additional implicit principles for several segments of our design to give it more flexibility for transition to higher fidelity design [21]. Some of them are:

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - With changes made in one part of design, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

In the initial process, we crafted multiple design solutions for the use cases of our system that achieve the corresponding design goal. However, by employing design principles and design heuristics from developer's point of view, we compared the alternatives, check whether they violate any of the design principles, considered necessary trade-offs and decided to abandon the ones that failed to achieve the benchmark. That being said, in cases with conflicts, we reached a conclusion by compromising with our reasoning and a design solution has been chosen based on the developer's judgement for this context. While it can be argued whether our chosen design is the best one or not, we do state that within the design constraints and in our judgement, we have tried to conclude with the optimal choice.

2.3.1 Design Solution Description

UC-2: Health Data

We considered alternate solutions for the responsibility SendRequest() to the Controller for starting the tracking ability. We considered assigning this responsibility directly from the initiating actor (Patient) to Controller. However, following Expert Doer and Low Coupling principle, we decided to incorporate this interaction between Patient and Controller via an additional UI element. Hence, Touch `was included to receive the request to start tracking from the Patient and then convey this to Controller [figure 2.1.1].

Another decision we had make was regarding the responsibility of notifying the tracking activation (EnableTracking). Initially we considered conveying the StartDetecting responsibility directly from Controller to Sensor Operator. However, in order to incorporate feedback in the process for the Controller (Controller needs to be notified of the success in enabling the tracking operation), we decided to include Tracking Enabler that notifies Controller. Adding a new object in the system had the risk of including longer chain of communication between the design objects. As the current design promised higher cohesion, improved readability and maintainability, we decided that it was a necessary choice. We employed this logic process to reach design decision for responsibility of notifying the patient that running stopped (NotifyPatientStopped) and reaching the running goal for the day (GoalReached).

One more instant where we considered alternate choice was assigning the responsibility of saving the data. In initial design prototype, we considered saving the data to the Offline database directly via *Database Manager*. In that case, Offline Database needs to save the real-time data sent from the sensor and later after the session ended, it needs to produce session data and send to online database. In later iteration, we discovered this introduces two potential problem:

- For longer session, there is a chance of missing data when data being sent to Offline Database in real-time.

- *Controller* should be assigned the responsibility to make command for requesting to save data online or offline.

Considering these two drawbacks of our design, we decided to save the real-time data in *DatabaseContainer*. Once a session ended, Controller sends request to *DatabaseContainer* to save data in the Offline Database. This design ensures each element has only one responsibility at a time (Expert Doer) and *DatabaseContainer* lightens the responsibility of Online Database (High Cohesion).

UC-3: GetVitalSigns - Phone

In this particular use case, we needed to employ our better judgement to make a choice of optimal design from the alternatives we have. As shown in the interaction diagram [figure 2.1.2], we needed to assign the responsibility of sending request from *Patient* to *PageMaker* for displaying the *VitalSigns*. The initial and final point of the flow of interaction were known: Controller and PageMaker. The decision of responsibility on receiving the data on *VitalSigns* was straightforward: *PageMaker* directly send the related page to the UI element (*TouchInteractor*) and allows the *Patient* to interact.

However, we had alternatives in the case of sending the request. We had the following alternative design choices:

Option 1: Request from *Patient* will be directly conveyed to *PageMaker* via *Controller* and *PageMake*r will return the page containing the most update report of *VitalSigns* (received from Offline Database).

Option 2: Request from *Patient* will come to the *Controller*. *Controlle*r will delegate the task to several additional elements to assign one responsibility per element to extract specific information requested by *Patient*.

In option 2, in later iteration, we discovered that we need to incorporate further objects to make process functional and balanced. We incorporated *SearchManager*, that will search for the specific parameters sent by *Controller* (additional object for managing the search operation). Based on the input in *SearchManager*, *Database Operator* was assigned the responsibility to send filtering request to the Database Postprocessor. Finally *Database Postprocessor* executed the task of rendering the filtered data to the PageMaker (to be conveyed for display in the UI).

When we employed the design principles on the two options, it was obvious that option 1 violated Expert Doer and High Cohesion principle. On the other hand, option 2 had a long chain of communication between the newly incorporated objects. We decided to compromise in this case and chose the design that offers optimal solution with less severe violation. This leads us to choose option 1. We believe that the current design ensures that each element knows what it is performing (Expert Doer), doing one task at a time (High Cohesion) and communicates effectively while staying within the design constraints.

UC-4 Get Vital Signs Web

A possible alternate design strategy for the "Get Vital Signs Web" use case, (UC-4), was to exclude the *SearchManager* object. Instead, *Controller* handles these responsibilities. The controller not only communicates with the database, but it also determines and manages the search parameters. The advantage of this solution is that there is a shorter communication distance between the Controller and the Database

We choose not to go with this design option for a few reasons. First of all, this design does not meet the high cohesion principle because *Controller* has too many responsibilities. This version of the use case requires that the controller to designate the search parameters in addition to handling all communication responsibilities. The search request definition and communication are not related attributes. Thus, *Controller* would be responsible for two functions instead of one.

Additionally, this solution reduces the design's demonstration of the expert doer principal. This principle means that "the one who knows should do the task". Although *Controller* knows the user's input for the request they are not the expert in how to arrange this input as a search request to the database. Adding *SearchManager* as an object, allows this object to be the expert in delegating search request parameters. This in turn allows the controller to continue to be the expert doer in communication only rather than trying to be an expert in another task.

UC-11 Physician Sign In

One design consideration for the physician sign-in use case was to add an object, *PatientEditor* to the design. In our interaction diagram this object would reside between the *PatientOperator* and *DatabaseManager* objects. It responsibilities would include adding and removing patients. We considerd this option because we thought it would reduce the number of responsibilities for the *PatientComparator* object and thereby promote the high cohesion principle.

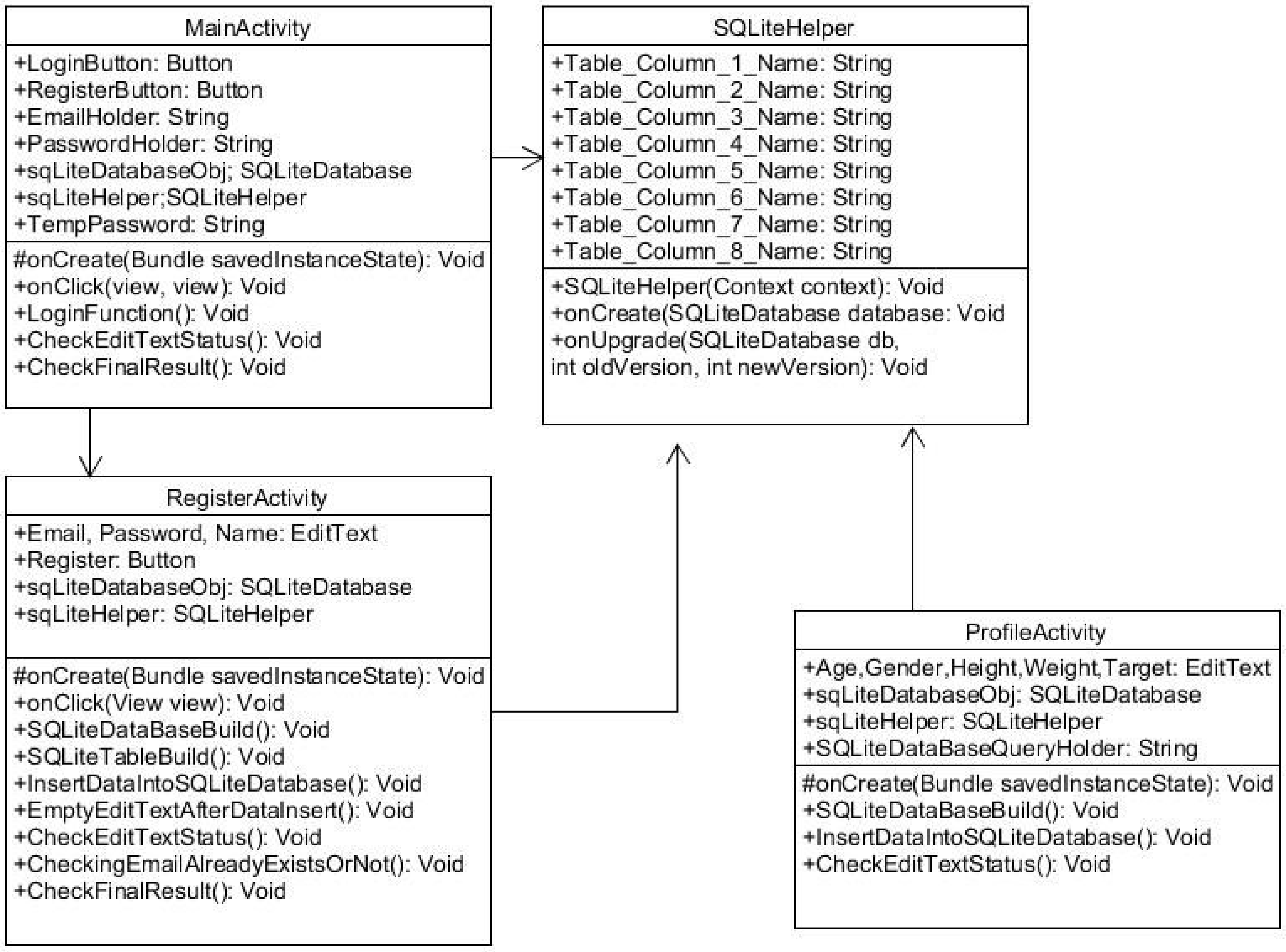
However, the following communication interactions would have to be added to support this object: 1) *PatientComparator → PatientEditor* and 2) *PatientEditor → OnlineDatabase*. Additionally, the *PatientComparator* object would still have to communicate with the online database. Therefore, this would increase the overall number of communication interactions among objects.These additional communication responsibilities defy the "low coupling principle". Furthermore, since the *PatientComparator* already has the information required to perform the add/remove patient request, it makes sense for it to do so.

3 Class Diagram and Interface Specification

3.1 Class Diagram

The class diagrams are separated into two sections: Database access and Health Monitoring Activity. This was done so that the diagram would not appear cluttered. The Database Access Class Diagram displays classes that access and modify the SQL database. This includes the classes with function related to Login, Registration, and profile management. The Health Monitoring Activity Class diagram displays classes related to the detection of activity and health information. This includes the classes with functions related to activity monitoring (step counter) and heart rate detection.

Database Access Class Diagram



MainActivity
+LoginButton: Button
+RegisterButton: Button
+EmailHolder: String
+PasswordHolder: String
+sqLiteDatabaseObj; SQLiteDatabase
+sqLiteHelper;SQLiteHelper
+TempPassword: String
#onCreate(Bundle savedInstanceState): Void
+onClick(view, view): Void
+LoginFunction(): Void
+CheckEditTextStatus(): Void
+CheckFinalResult(): Void
SQLiteHelper
+Table_Column_1_Name: String
+Table_Column_2_Name: String
+Table_Column_3_Name: String
+Table_Column_4_Name: String
+Table_Column_5_Name: String
+Table_Column_6_Name: String
+Table_Column_7_Name: String
+Table_Column_8_Name: String
+SQLiteHelper(Context context): Void
+onCreate(SQLiteDatabase database: Void
+onUpgrade(SQLiteDatabase db,
int oldVersion, int newVersion): Void
RegisterActivity
+Email, Password, Name: EditText
+Register: Button
+sqLiteDatabaseObj: SQLiteDatabase
+sqLiteHelper: SQLiteHelper
#onCreate(Bundle savedInstanceState): Void
+onClick(View view): Void
+SQLiteDataBaseBuild(): Void
+SQLiteTableBuild(): Void
+InsertDataIntoSQLiteDatabase(): Void
+EmptyEditTextAfterDataInsert(): Void
+CheckEditTextStatus(): Void
+CheckingEmailAlreadyExistsOrNot(): Void
+CheckFinalResult(): Void
ProfileActivity
+Age,Gender,Height,Weight,Target: EditText
+sqLiteDatabaseObj: SQLiteDatabase
+sqLiteHelper: SQLiteHelper
+SQLiteDataBaseQueryHolder: String
#onCreate(Bundle savedInstanceState): Void
+SQLiteDataBaseBuild(): Void
+InsertDataIntoSQLiteDatabase(): Void
+CheckEditTextStatus(): Void

Figure 3.1.1: Database Access Class Diagram

Health Monitoring Activity

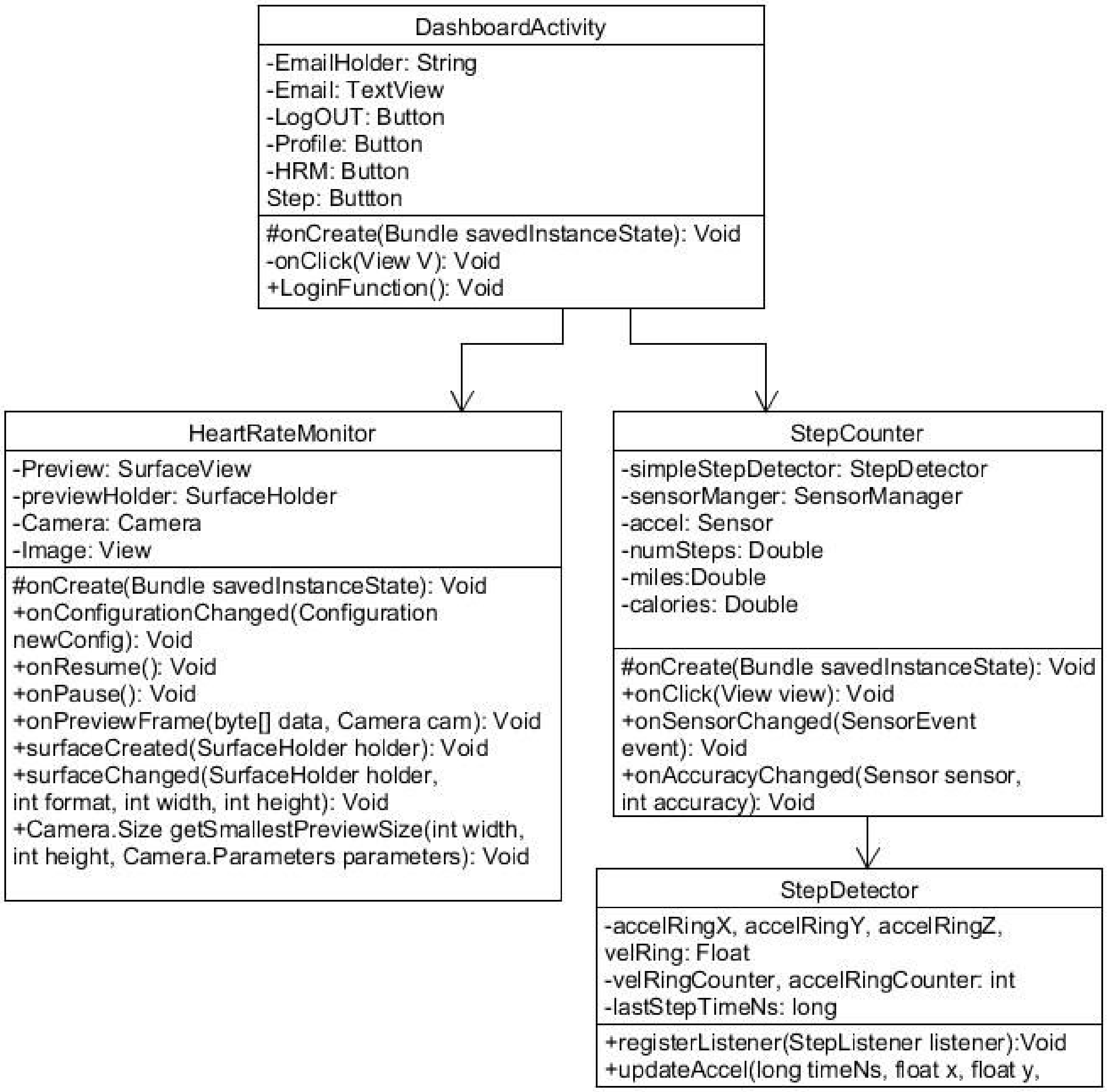

DashboardActivity
-EmailHolder: String
-Email: TextView
-LogOUT: Button
-Profile: Button
-HRM: Button
Step: Buttton
#onCreate(Bundle savedInstanceState): Void
-onClick(View V): Void
+LoginFunction(): Void
HeartRateMonitor
-Preview: SurfaceView
-previewHolder: SurfaceHolder
-Camera: Camera
-Image: View
#onCreate(Bundle savedInstanceState): Void
+onConfigurationChanged(Configuration newConfig): Void
+onResume(): Void
+onPause(): Void
+onPreviewFrame(byte[] data, Camera cam): Void
+surfaceCreated(SurfaceHolder holder): Void
+surfaceChanged(SurfaceHolder holder, int format, int width, int height): Void
+Camera.Size getSmallestPreviewSize(int width, int height, Camera.Parameters parameters): Void
StepCounter
-simpleStepDetector: StepDetector
-sensorManger: SensorManager
-accel: Sensor
-numSteps: Double
-miles:Double
-calories: Double
#onCreate(Bundle savedInstanceState): Void
+onClick(View view): Void
+onSensorChanged(SensorEvent event): Void
+onAccuracyChanged(Sensor sensor, int accuracy): Void
StepDetector
-accelRingX, accelRingY, accelRingZ, velRing: Float
-velRingCounter, accelRingCounter: int
-lastStepTimeNs: long
+registerListener(StepListener listener):Void
+updateAccel(long timeNs, float x, float y,

Figure 3.1.2: Health Monitoring Activity Class Diagram

3.2 Data Types and Operation Signatures

Registration

For the registration part of our code, the classes are as follow:

MainActivity

This class starts the user instance by initializing login objects. It hosts relevant object calls from different files

Attributes:

+ **LoginButton**:Button.

+**RegisterButton**:Button.

+**EmailHolder**:String that will receive the current user email input (Main Activity)

+**PasswordHolder**:String that will receive the current user password input (Main Activity) .

+**sqLiteDataBaseOb**j:SQLiteDatabase is index that resides the main,temporary or attached Database.

+**sqLiteHelper**:SQLiteHelper:SQLiteHelper is context String Name that creates object helper to (open/edit /manage Database).

+**TempPassword**: String that will Store the received password from the user input in a temporary way.

Operation:

#**onCreate(Bundle savedInstanceState)**: Void. This method gets called when the activity is first created and runs only once.

+**onClick(View V)**: Void. This method just adds click listener the the buttons.

+**LoginFunction()**: Void. Same as above, just adds click listener to the logIn button.

+**CheckEditTextStatus()**: Void .This method will get the text from the email holder convert it to string and check if the received data is empty or not.

+**CheckFinalResult()**: Void.This method will check the Final entered Value and match it if it valid or not and approve the Final Result status .

RegisterActivity

In this class is where the user creates a new account. The database is opened for entering new data.

Attributes:

+**Email,Password,Name**:EditText.Stores the information that the user enters

+**Register**: Button

+**sqLiteDataBaseOb**j:SQLiteDatabase is index that resides the main,temporary or attached Database.

+**sqLiteHelper**:SQLiteHelper:SQLiteHelper is context String Name that creates object helper to (open/edit /manage Database).

Operation:

#**onCreate(Bundle savedInstanceState)**: Void. This method gets called when the activity is first created and runs only once.

+onClick(View V): Void. This method just adds click listener the the buttons.

+**SQLiteDatabaseBuild**(): Void.This Method will initialize the Database .

+**SQLiteTableBuild():** Void. This method will create a Database Sql Table if does Exist.

+InsertDataIntoSQLiteDatabase(): Void. This method inserts data into the database.

+**EmptyEditTextAfterDataInsert**: Void. This method will empty EditText After done inserting process.

+**CheckEditTextStatus():** Void. This method just checks if EditText is empty or not.

+**CheckingEmailAlreadyExistsOrNot**: Void. This method will Check if Email is already exists or not.

+CheckFinalResult(): Void.This method will check the Final entered Value and match it if it valid or not and approve the Final Result status .

ProfileActivity

The Patient creates his/her profile. All of the details and information are stored in the database.

Attributes:

+**Age,Gender,Height,Weight,Target**: EditText. Stores the information that the user enters.

+**sqLiteDatabaseObj**:SQLiteDatabase. Is an index that resides in the main, temporary or attached Database.

+**sqLiteHelper**: Is a context String Name that creates object helper to open/edit /manage Database.

+**SQLiteDataBaseQueryHolder:** String. Is a query to insert the data into the table.

Operation:

#**onCreate(Bundle savedInstanceState)**: Void. This method gets called when the activity is first created and runs only once.

+SQLiteDataBaseBuild(): Void. This method creates the database.

+InsertDataIntoSQLiteDatabase(): Void. This method inserts data into the database.

+**CheckEditTextStatus():** Void. This method just checks if EditText is empty or not.

DashboardActivity

Here is where all the application options are located: Profile, Heart Rate Monitor, Step Counter.

Attributes:

-EmailHolder: String. Store the user’s email address sent by MainActivity.

-Email: TextView. Displays email address to the user.

-LogOUT: Button.Triggers the log out event when clicked.

-Profile: Button. Opens ProfileActivity class when clicked.

-HRM: Button. Opens HeartRateMonitor class when clicked.

-Step: Button. Opens StepCounter class when clicked.

Operation:

#onCreate(Bundle savedInstanceState): Void. This method gets called when the activity is first created and runs only once.

-onClick(View V): Void. This method just adds click listener the the buttons.

+LoginFunction(): Void. Same as above, just adds click listener to the logOUT button.

SQLiteHelper

In this class we define all the attributes of the databases.

Attributes:

+Table_Column_1_Name: String. Represents the name column in the table.
+Table_Column_2_Name: String. Represents the email column in the table.
+Table_Column_3_Name: String. Represents the password column in the table.
+Table_Column_4_Name: String. Represents the age column in the table.
+Table_Column_5_Name: String. Represents the gender column in the table.
+Table_Column_6_Name: String. Represents the height column in the table.
+Table_Column_7_Name: String. Represents the weight column in the table.
+Table_Column_8_Name: String. Represents the target column in the table.

Operation:

+SQLiteHelper(Context context): Void. A helper subclass to manage database creation and version management.

+onCreate(SQLiteDatabase database): Void. Takes care of opening the database if it exists, or create it if it does not.

+onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion): Void. Upgrades the database as it is necessary.

Step Counter

For the step counter functionality, the implemented classes are as follow:

StepCounter

This class starts StepDetection by calling relevant objects.

Attributes:

-simpleStepDetector: stepDetector. Stores the value of the accelerometer sensor.
-sensorManager: SensorManager. Variable used to manage the smartphone's sensors.
-accel: Sensor. Temporary variable used to store the initial accelerometer values.
-numSteps: Double. Stores the actual number of steps
-miles: Double. Stores the miles traveled so far.
-calories: Double. Stores the calories burned so far.

Operation:

#onCreate(Bundle savedInstanceState): Void. This method gets called when the activity is first created and runs only once.

+onClick(View view): Void. This method just adds click listener the the buttons.

+onSensorChanged(SensorEvent event): Void. Updates the accelerometer values every time the sensor detects a change.

+step(long timeNs): Void. Calculates the steps, miles and calories.

StepDetector

The sensor values are read and the steps calculations are made.

Attributes:

-accelRingX,accelRingY,accelRingZ,velRing: Float identifiers for the Accelerometer sensor.

-velRingCounter,accelRingCounter:integer identifiers for the Accelerometer sensor.

-lastStepTimesNs: long identfiers to hold the last number of steps that detected from the user.

Operation:

+registerListener(StepListener): void. This Method will register the unusual behaviour between the sensor data and millisecond.

+updateAccel(long timeNs, float x,float y, float z); Void. This method will update the the old Accelerometer values with new upcoming ones .

Heart Rate Monitor

For the Heart Rate monitoring functionality, the implemented classes are as follow:

HeartRateMonitor

This class initializes all necessary sensors: camera, wake screen, flash, etc. The, it gets the preview frame and process red values using rolling average filter.

Attributes:

-Preview: SurfaceView. SurfaceView takes care of placing the surface at the correct location on the screen.

-PreviewHolder:SurfaceHolder. Is identifier for The holder of the surface.

-Camera:Camera. provides an interface to individual camera devices connected to an Android device.

-Image: View. preview images are sent to SurfaceView or TextureView.

Operation:

#onCreate(Bundle savedInstanceState): Void. This method gets called when the activity is first created and runs only once.

+OnConfigurationChanged(Configuration newConfig): Void. This method will identify any new configuration and will make new configured environment based on the received configuration.

+**onResume()**: Void. This method is called everytime the MainActivity is hidden and try to resumed.

+**onPause()**: Void This method is called everytime the main activity is stopped and switched to another activity.

+onPreviewFrame(byte[] data,Camera cam): Void. This is a Callback interface used to deliver copies of preview frames as they are displayed.

+SurfaceCreated(SurfaceHolder holder , int format , int width , int height):Void .This method will hold and store the current Surface with its dimensions

+SurfaceChanged(SurfaceHolder holder , int format , int width , int height): Void. This method will update the current surface with new one (dimensions of the surface).

+Camera.Size getSmallestPreviewSize(int width, int height, Camera.Parameters parameters): Void. This method will sets the dimensions of the picture.

3.3 Traceability Matrix

The table below contains the list of the domain concepts and how they map to the classes which have been implemented till now:

	Main Activity	Register Activity	Profile Activity	Dashboard Activity	SQLite Helper	Step Counter	Step Detector	Sensor Filter	HeartRate Monitor
Controller	X								
Sensor Operator					X		X		X
Search Manager									
Tracking Enabler					X				
Data Container		X	X		X	X			X
Database Authenticator			X	X	X				
The domain concepts below have not been implemented yet.									
Database Manager									
Database Operator									
Database Postprocessor									

Interface									
Pagemaker									
Patient Searcher									
Patient Operator									
Patient Notifier									
Notifications Enabler									

Our classes differ from our domain model because we decided to take a different approach in the code implementation. Our domain model was originally based on a *pipe and filter* approach. We centered the model around how one object would process the data before presenting it to the next object. So for example, the SearchManager filtered and processed the search parameters from the controller and passed the data to the DatabaseOperator. Similarly, the DatabaseOperator processed the data from the SearchManager and output the filtered information to the DatabaOperator.

However, in implementing the code we choose to “divide and conquer” and addressed the more complex tasks of the project first. Thus, we made data collection from sensors (used for the step counter) and database communication a priority. As a result, we shifted the software architectural style to a more task oriented approach. This leads to different class objects and different naming conventions. We wanted to name the classes according to their associated task or activity. Thus we have the classes names such as StepCounter and RegistrationActivity.

Although the class names and functionality differ from the domain model, we are ensuring that the vital domain is/will be addressed. Due to time constraints, some of the optional use cases will not be implemented. Thus these domain concepts will not link to a particular class.

4 System Architecture and System Design

4.1 Architectural Styles

Our design is comprised of three main components. A mobile application, a database and a web application. The Architectural styles that we have used to design our web application is two-tier Client-Server model. In this type of Architectural style, clients can communicate with a server over a computer network. To ensure data consistency, we are using an online database as part of the server. The client which is user running the web application can initiate a communication session to this server and retrieve patient health information as required. This is done by connecting to the server using TCP/IP socket connection. The advantage with this kind of model is that we have a centralized control and easy maintenance as changes to server will not affect the client end. In our system, the web application will display patient details to the doctor/admin. The online database stores patient information. This includes basic profile information in addition to step activity and heart rate which we get from the mobile application.

The mobile application uses Model View Presenter (MVP) architectural pattern. The biggest advantage this model gives us is code organization. The Model layer controls how data is stored and modified in accordance to the business logic. View (Android Activity) is a passive interface that displays data and user actions to the presenter and defines flow logic. Presenter retrieves data from Model and show it in the View. It also processes user action forwarded to it by the View. This kind of abstraction lets us define clear roles for each component. Object oriented design is being used as an integral design style in coding. In addition, we have defined clear responsibilities for each Activity so that changes to one component will not affect another component. This is especially useful for big applications which involve multiple contributors. As mentioned before, an online database is used for data consistency. This also helps when the app is used by multiple users and on multiple devices.

4.2 Identifying Subsystems

Design Considerations

We followed OMG UML instructions to identify all subsystems and packages for our system [22].

Package

A package is the grouping of model elements of different kinds including other packages to hierarchy of model. Packages are mostly used to create an overview of a large set of model elements, to organize a large model, to group related elements and offer separate namespaces to avoid confusions among multiple libraries.

In our system, when identifying packages for UML package diagram, we followed the following principles:

1. Gather model elements with strong cohesion in one package
2. Keep model elements with low coupling in separate packages
3. Minimize associations between model elements in different packages.

Subsystem

Subsystem is a grouping of model elements that represent a certain technique used for decomposing a large system into several smaller systems. Subsystems are very useful to express how a set of components are composed into a large system for component based project development. A subsystem generally offers two aspects-

Specification elements - It offers an external view showing the services provided by the subsystem. It mainly describes the interface of the subsystem.

Realization elements - It offers an internal view showing the realizations of the subsystems. It contains the actual contents and functions of the subsystem.

There are multiple approaches for performing the specification techniques: Use case, logical class, and operations approach. A specific or a mixed approach may work for different systems. Mapping between specifications and realizations defines the role to be played when a task to be performed.

In our system, when identifying subsystems, we followed the following principles:

1. Defining a subsystems for each separate part of the large system
2. Choosing specification technique depending on the kind of the system and its subsystems. In this case, we chose use case approach.

Identifying subsystems involves backtracking, evaluation and revision of possible solutions. The heuristics we used to identify subsystems are:

- Creating a dedicated subsystem for objects in one use case into the same subsystem
- Creating a dedicated subsystem for objects used for moving data among subsystems
  Ensuring all objects in same system are functionally related
- Minimizing number of association or interaction between subsystems

We further considered the following criteria:

- The kind of services provided by the subsystems
- The contents of the subsystems
- High Cohesion - Each subsystem will perform only one activity
- Loose Coupling - Minimum dependency between the subsystems

In general, objects that are identified during requirement analysis were grouped into subsystems. The degree of cohesion and coupling between subsystems are used to perform subsystem decomposition.

The Subsystems

As we observe the system sequence diagram of our system design, we noticed that our system has a web application on the physician end and a mobile application on the patient end. The mobile application only interacts with the local database, while web application interacts with online database. Furthermore, we observed that local database has access to online database for updating and storing information.

Considering the heuristics mentioned in the above section, we decomposed our system into following subsystems:

1. Mobile application
2. Local database
3. Web application
4. Online database

Based on our observation, the diagram of the subsystems and the containing packages looks following:

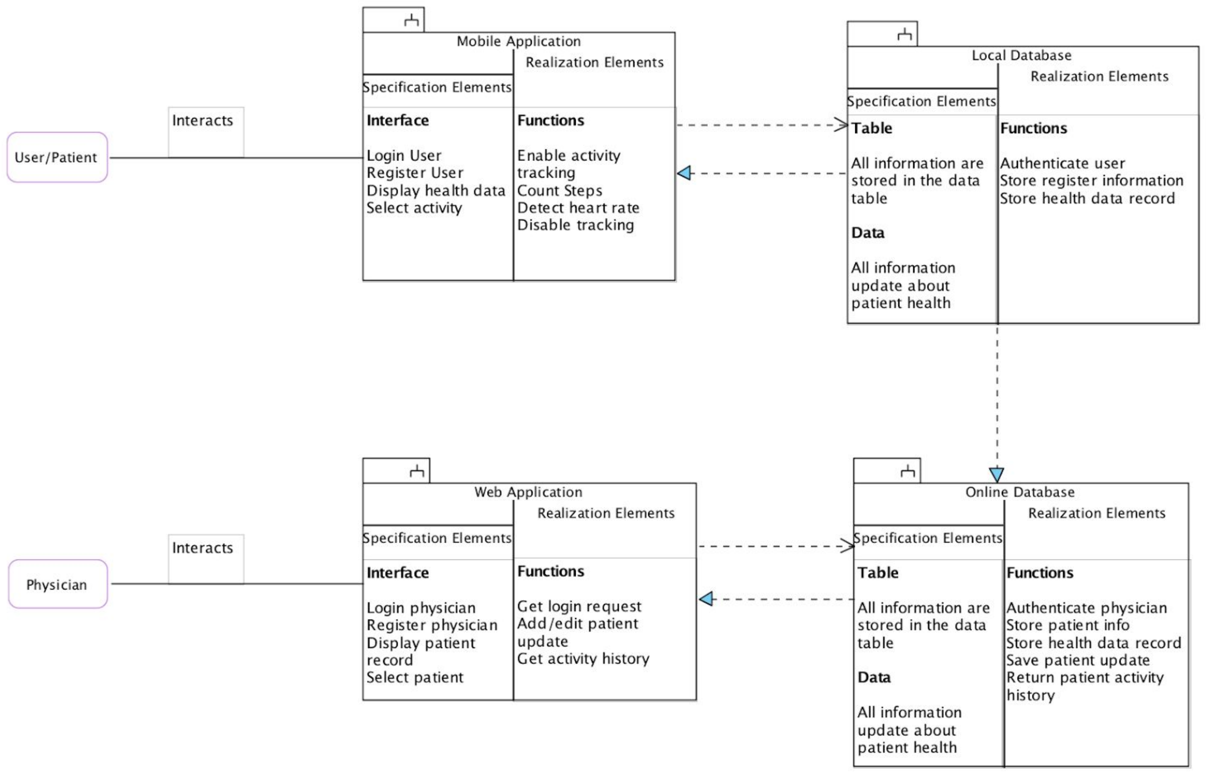


Figure: UML package diagram of subsystems

4.3 Mapping Subsystems to Hardware

Mobile application

The mobile application initiates the vital parts of the patient end of the system: tracking user activity, detecting user's steps, detecting heart rate of the user and disable activity tracking. The accelerometer sensor and camera sensors are crucial for the functions performed inside this subsystem. The interface allows the user to enter information for login and register purpose. It also displays the log of health vitals during particular activity and retrieved record of health data from the local database.

Our application has been developed for Android operating system based handheld (mobile devices). The hardware for this subsystem will be any Android mobile device that has API level 15 or higher. It should also have its accelerometer sensor working and functional medium resolution back camera for the step detecting and heart rate monitoring service.

Web application

The web application initiates the vital parts of the physician end of the overall system: receiving login and register request, updating patient information (add/edit) and retrieving the activity history of a particular patient. The web page interface allows the physician to enter information for login and register purpose. It also allows physician to choose any particular patient and displays health data and information of all patients retrieved from the local database.

The web application has been developed to be displayed via any web browser (e.g. Safari, Mozilla, Chrome, etc.). Any computing device (e.g. Desktop, laptop, tablet or mobile phone) with internet connection is capable of providing the hardware support for the web application subsystem. The web application does not provide offline service in the current design.

Local Database

The real-time functionality of the mobile application depends on the performance of local database. In our case, we used SQLite, the built-in database offered in Android because it is a highly reliable, public- domain and self contained relational database. Data tables are used to to store two types of information regarding the patient: the basic profile information and the health vitals during activity. In each session of activity, the local database fills out the datatable with information of the most recent session and stores this updated information the database. It also sends the information requested by the user about a selected activity to be displayed on the mobile UI. As we are using a built-in database within Android, our hardware for this subsystem is also the Android mobile device used by the user (patient).

Online Database

The functionality of the web application depends on the performance of the online database. It receives activity information of different patients from the local database residing in the mobile device of each patient and stores them. From the physician side, it stores the basic information of each patient that are entered by the doctor. It also the registration-related information of the patient that is requested for authentication each time the physician logs in. Finally, the online database is also responsible for storing the patient information updated by the physician. This database is crucial for implementing the activity scheduling feature of our system, which is yet to be implemented. We plan to use Google Firebase for this subsystem as it offers realtime syncing and storing of data.

The following diagram gives an overview of the current mapping of the subsystems of our system into hardwares:

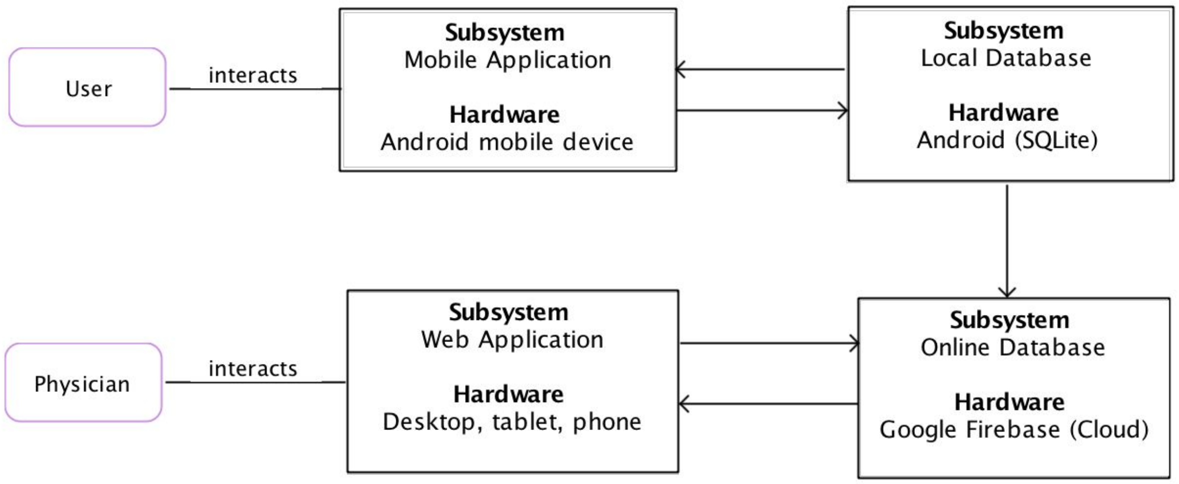


Figure: Mapping of subsystems into hardwares

4.4 Persistent Data Storage

For the initial stages of our project, we used SQLite Database which is preferred for Android applications. The SQLite database is simple, convenient and robust. But, for the future work, we will be switching to Firebase Database which is cloud-hosted database.The data is stored as JSON and can be used by cross-platform apps whether Android , iOS or web applications.

In our project, the data collected is separated into three parts: User Profile, Step Count and Heart Rate(Beats Per Minute). User profile includes user data such as name, email , password, age, gender, weight, height and target step count. Step Count includes email id of specific user, date, number of steps taken, miles, and calories burnt. Heart Rate includes email id of specific user, date, and heart rate. All this data can also be accessed from the web application when a physician logs into the website with proper authentication. Physician can extract the specific details of each patient and notify the patient accordingly.

4.5 Network Protocol

Several network protocols were implemented for our project:

1. **HTTP Protocol:** We used this type of protocol (Hypertext Transfer Protocol) in order to support the connection for the login/sign up in a web page and to support the connection between the user and the physician part, to stream the data request between the user terminal and physician terminal.

2. **SSH Protocol:** SSH provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client application. This protocol is uncensored because we are using the SQLite which is based on the SSH client – Server Stream tunneling.

3. **TCP/IP Protocol:** Nearly all computers and smart devices today support TCP/IP. This is not a single networking protocol, and will also be uncensored because this protocol will be used on the Physician (Computer) end and the Patient (Android Phone) end as a TCP/IP4 (wifi, 4G, 3G, 2G).

4. **Named Pipes Protocol:** This protocol is used only on SQL Database communications**,** it will allow network access to SQL Server Express by supporting numerous network protocols, including NetBEUI, TCP/IP.

4.6 Global Control Flow

The system is event driven which is based on the user interaction with the device so when the user interacts with the smartphone he/she will be able to choose the activity to perform. Also, once the user wants to check his/her previous activity log history, he/she will need to press the specific activity so the data will be transferred and shows up to the patient.

Moreover, there will be no need for timer in this current system, although it will be concerned about real time (it is a real-time system), but it is not periodic.

4.7 Hardware Requirements

For the Physician Part

1. **Operating System:** it can be Microsoft Windows (7, Vista,8.1,10), Mac OS, Linux.
2. **Display:** Any output Source can perform well (not requiring any special sources), with any resolutions.
3. **Minimum Memory :**1 GB of RAM (in order to open the Physician web page and performs the job correctly without any glitches).
4. **CPU:** Single core CPU and UP**.**
5. **Storage:** no need for Local storage because all the information will be stored on online Database for the Physician so it will be retrieved from the server based on the Physician Request.
6. **Network:** 56 Kbps (minimum) in order to perform the request in sufficient way (less will take longer and will cause data lost sometimes).

For the Patient Part

1. **Operating System:** Android (based on Linux OS) with minimum SDK (2.2).
2. **Display:** it will require colored Screen with at least 640 x 480 pixels and more.
3. **Minimum Memory:**1 GB of RAM to perform the OS and Application internal requests to be performed well without glitches.
4. **CPU:** Dual Core CPU and UP.
5. **Storage:** it will need sufficient amount of space (100MB-500MB) to store the Application resources and temporary files and also to store the local backed-up database.
6. **Network:** Wifi-4G-3G-2G.

5 Algorithms and Data Structures

5.1 Algorithms

5.1.1 Step Counter

The step counter uses the phone's accelerometer to count the steps. The accelerometer uses the standard 3-axis sensor coordinate system to see the data values. This is defined relative to the screen of the phone in default orientation. The step detector triggers an event each time the user takes a step. The latency or delay is expected to be around 2 seconds. The accelerometer sensor measures the force in m/s^2 , which is applied to the device on all the three axes(x,y and z), including the force of gravity. These values are normalized so that the value is close to zero. And then the estimated velocity can be checked with a given threshold. If the estimated velocity is more than the threshold, it will be counted as a step. After getting the step count, we have used the below formulas to calculate the miles runs and the calories burnt.

Miles = number of steps * 0.0005, Calories = Miles* 0.0128

5.1.2 Heart Rate Monitor

We have used an open source algorithm for determining heart rate. The algorithm uses the phone camera to capture the image of user's fingertip and calculates the heart rate based on redness of the image. A brief description of the algorithm is as follows:The flash and the wake screen lock are acquired when the camera is turned on. The preview frame is taken as an image of the YUV420SP format. This is then converted to the known RGB image format. This is done by a set of equations:

$R = 1192*y + 1634 * u ; G = (1192 * y - 833 * v - 400 * u); B = (1192 * y + 2066 * u);$

Where R,G,B correspond to the RGB components and y,u,v correspond to the YUV420SP format.
After the redness amount is obtained as explained above, this is stored as the average red value. This is done repeatedly for as long as the fingertip is placed on the camera. A rolling average (or moving average) filter is used to smooth out this data. When the average is higher or lower than the average, heart beat can be obtained and displayed.

5.1.3 Recovery Time

As part of the final application, we will be adding a feature to determine the "recovery rate" of the patient. As our app focusses on the exercise part of the user's activity, this is a very good indicator of the user's fitness. When the user is exercising, their heart rate is spiked up. The number varies on exercise time and intensity of exercise. When the user stops exercising, heart rate returns to normal heart rate in about 3-5 minutes, with a majority of drop in the first minute. A lot of studies show that this "recovery time" is an important indicator of the user's fitness. Giving this information to the physician will help in better diagnosis. It is also especially useful for patients who have undergone surgery and are in their recovery period or for personal trainers to track their client's fitness levels. Since this requires learning the user's heart rate values every day, we would like to use Artificial Neural Networks (ANN). We use the feed forward, back-propagation algorithm to predict future recovery time values. When constructing ANN models, one of the primary considerations is choosing activation functions for hidden and output layers that are differentiable. This is because calculating the back-propagated error signal that is used to determine ANN parameter updates requires the gradient of the activation function gradient. We use the sigmoid function for this purpose.

$$a(x) = \frac{1}{1+e^{-x}}$$

This is used for the feed forward network, along with our input values and random weights assigned to each link. We use the actual value for the final data point as the target and compare with our predicted value. This lets us determine the error. We can now determine delta values by multiplying this error with activation function values. The weights are then adjusted using the back-propagation algorithm.

*derivative = output * (1.0 - output)*
*error = (expected - output) * transfer_derivative(output)   (for a neuron)*
*error = (weight_k * error_j) * transfer_derivative(output)   (for the hidden layer)*

Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron.
Network weights are updated as follows:
*weight = weight + learning_rate * error * input*
The network is trained like explained for daily values till we can accurately determine the target value. This is used to predict the next day's value. When the patient is consistently having lower time than predicted value, the physician can be alerted to check on him.

5.2 Data Structures

Currently, we use **integer arrays** to store intermediate values in step counter and heart rate monitor. In Step Counter, we use arrays to store acceleration values for the three axes x, y and z. The steps counted, the calories burnt and miles walked are of integer, decimal and decimal data types respectively. In Heart Rate Monitor, we use arrays for storing average image redness, conversion to RGB and beats count since we have a set of values to store and process. Similarly, for ANN, since we have multiple data points as input values, we anticipate to use arrays to store every set of values in feedforward and backpropagation. We will use 2 dimensional integer arrays for weights of links between input and hidden, and hidden and output nodes. The rest of the intermediate variables are almost all of type **integer** or **float**. We also will use list or array data type to obtain values from the database to the web application for analysis and display, and android application to display history.

6 User Interface Design and Implementation

During brainstorming session prior to start working on the implementation, we analyzed several similar apps that are available in the market and identified what are the likeable features popular among the users. Based on these observations, In the first iteration of report, we identified UI requirements for our system which are trivial for the functional purpose of our system, but crucial for the user to succeed in achieving user goals with ease. The requirements were following:

The interface of the mobile app will be easy to navigate so that patient can change between menus with minimal effort even when working out.

Both the mobile and web application shall be intuitive; so that general experience with any Android or web app will suffice to use the basic features effectively.

We designed the initial mock-ups for the mobile and web application user interface keeping these requirements in mind. The mock-ups were designed for the use cases that are most important for the system to be functioning to serve user's needs. We tried to include in these mock-ups the features that are not only essential for functional purpose but also make the system easy to use for the human user.

6.1 Modifications made during Implementation

During the implementation phase, we realized that though our mock-ups were well thought out and were designed with user-centric focus in mind, several changes were required to be made. This was basically required due to repeated revision made on our initial idea as we went deeper into the implementation phase.

6.1.1 Mobile Application

***Information required for registration and login**:* One significant change we made was the information required for the initial account information for the user. In our mock-up screen for registration, full name of the user, a username and password were sufficient for a new user to open an account with the mobile application. During implementation, we realized that it is crucial for the authentication purpose to require the user enter his/her e-mail address during account initiation. As our system had a web application on the physician end and we plan to sync the mobile and web

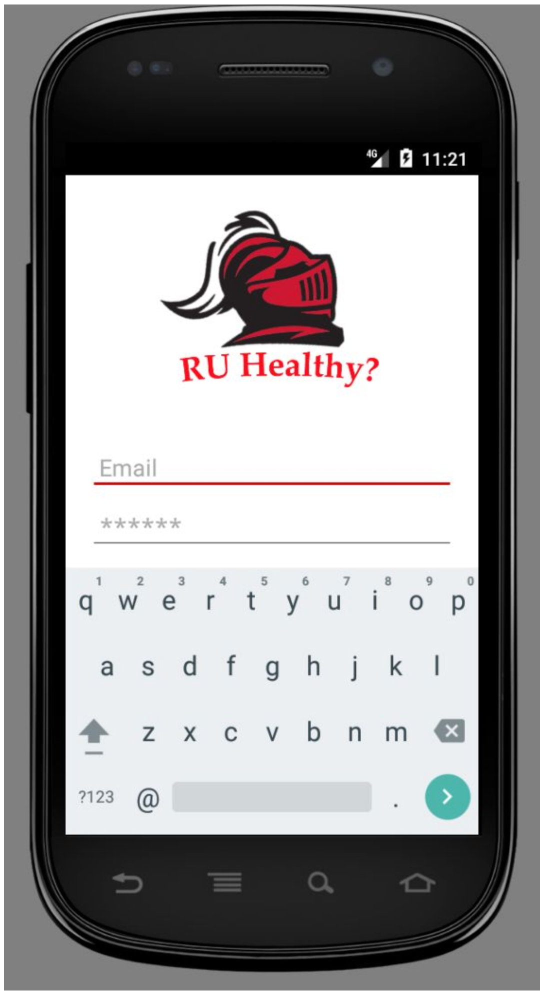


Figure 6.1.1 Login Page (Mobile)

applications to operate in harmony with the online database, inclusion of the e-mail address seemed a wise choice for security purpose. We also decided to include a field for the user to confirm their password to avoid cases where the user may have pressed a wrong key when typing the password. Confirming the password a second time will ensure that the user entered the password they intended to.

Inclusion of Dashboard page: Furthermore, in our initial mock-up design, when user logs into his/her account with credentials, the user is directed to profile page containing the basic information like age, height, etc. In our current design, once logged in, user gets to the Dashboard page, which contain tabs like Profile, Heart rate monitor, Step counter, Log out, etc. The idea is that instead of directing the user to the profile, we will allow the user to choose wherever he/she wants to navigate from the Dashboard page.

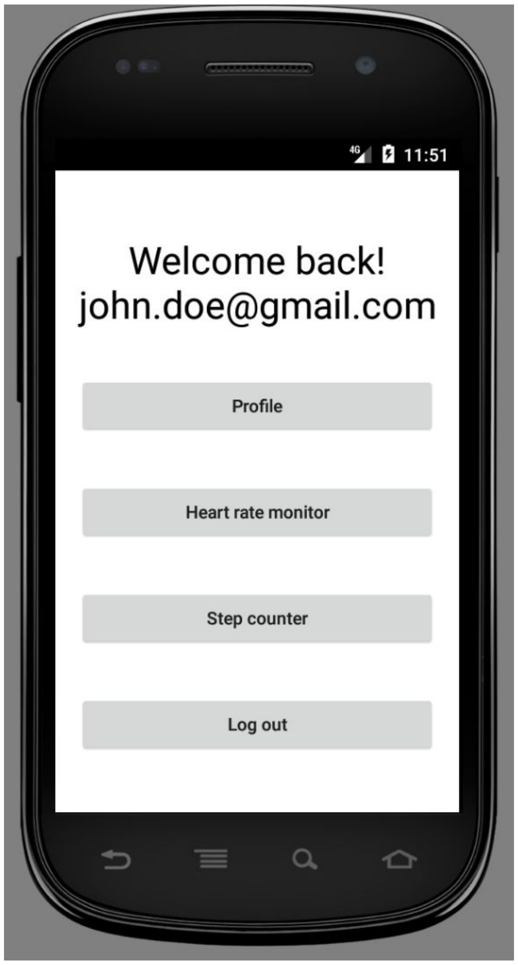
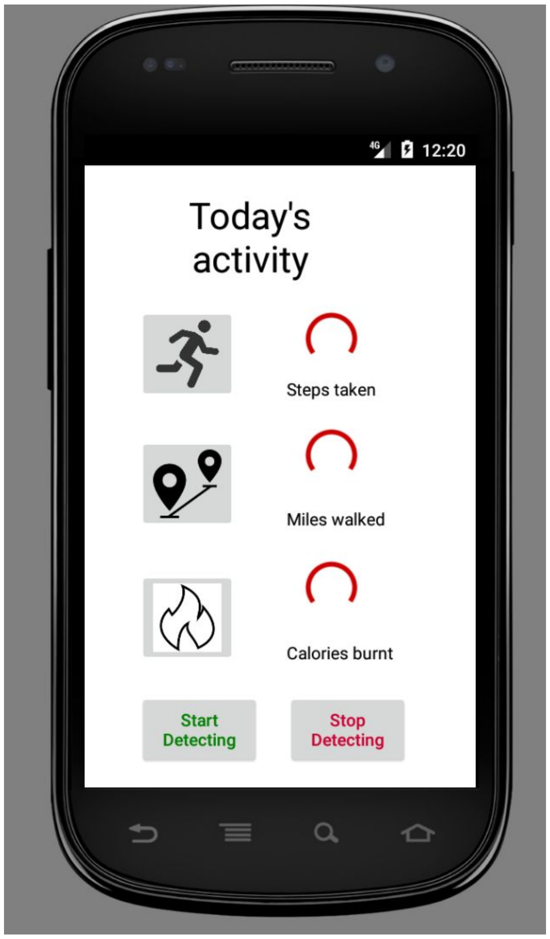


Figure 6.1.2 Dashboard

Inclusion of additional page for heart rate monitor: Another major change we have brought to our UI design is a separate page for the heart rate monitor. In the initial mock-up we excluded this feature as it was not among the use cases we were focused on. However, towards the end of first iteration, we decided to include the heart rate monitoring feature along with step counting feature. The dashboard was a vital part for this decision, as by including the additional page called Dashboard, we allowed user to navigate to all the basic features of our app as needed. This certainly reduces user effort, as user can view the feature they need to one at a time with a single click on the corresponding button.



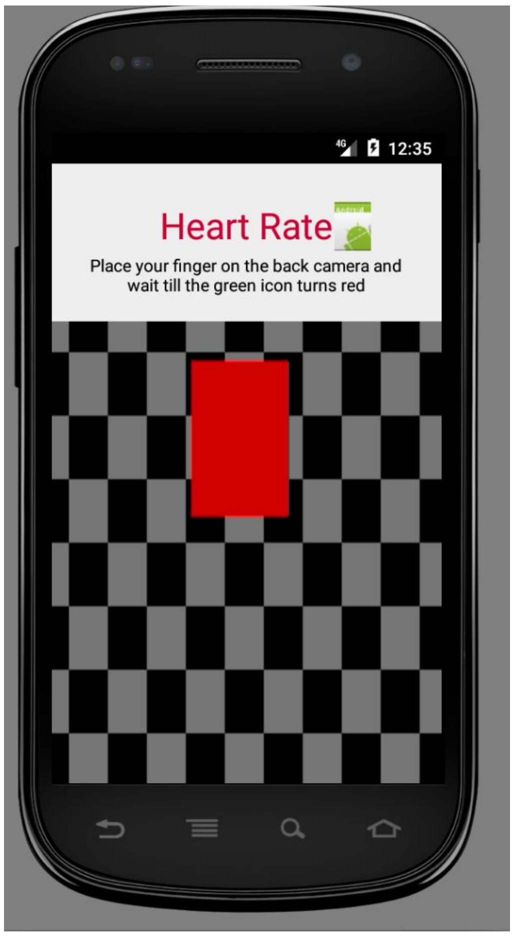
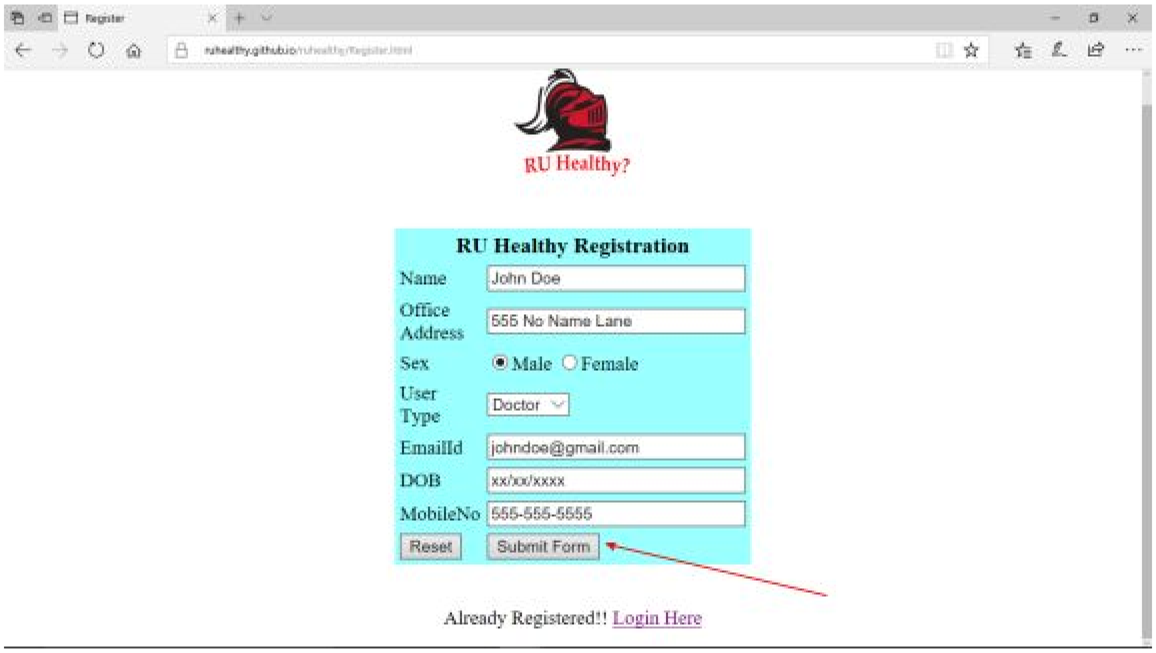

12:35
Heart Rate
Place your finger on the back camera and wait till the green icon turns red

Figure 6.1.3 and 6.1.4 Step counter (left) and Heart rate monitor (right)

6.1.2 Web application

***Information required for registration**:* Similar to the mobile application, a significant change for the web application was the initial information required for registration. In our mock-up, we only included user's name, email and password as the required information. But during implementation phase, we realized that as the web application will be mostly used by health professionals, more information requirements will give the system authenticity. In the current version, users will need to enter name, office address, contact number, etc. while opening an account. Furthermore, we implemented the web application in such a way so that user can choose the type of role they play in the system; they can select whether they are a patient or a physician. In initial mock-up, the web application was specifically designed for physicians only. This enables the user to use RU Healthy services from a web browser along with the mobile application used during workout.

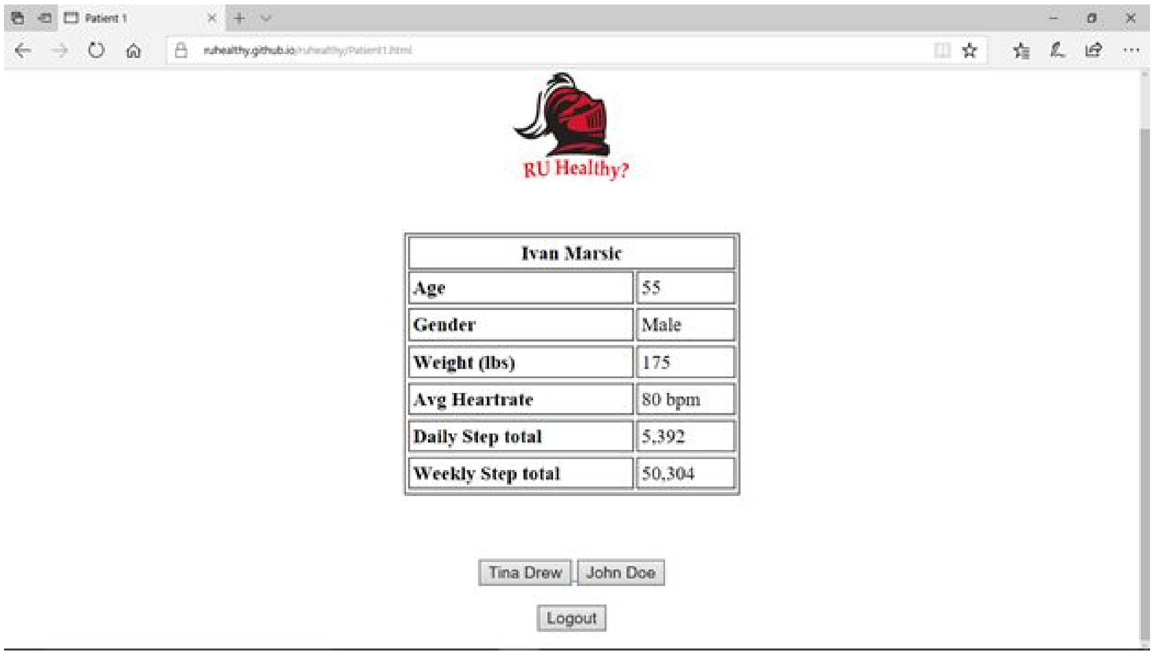


RU Healthy Registration

Name	John Doe
Office Address	555 No Name Lane
Sex	[✓ Male] Female
User Type	Doctor
EmailId	johndoe@gmail.com
DOB	xx/xx/xxxx
MobileNo	555-555-5555

Already Registered!! Login Here

Figure 6.1.5 Registration (Web)



Ivan Marsic	
Age	55
Gender	Male
Weight (lbs)	175
Avg Heartrate	80 bpm
Daily Step total	5,392
Weekly Step total	50,304

Figure 6.1.6 Patient summary

Revised Patient summary page: Additionally, we customized the visual look of the patient summary page. In the initial mock-up, the patient's weekly activity summarized patient's performance in a graph and displayed a one-line summary of the missed sessions for the week. We later realized that the physician would require a summary on a daily basis to provide more accurate health advice. In the current version of implementation, patient activity for the day is summarized in a table which contains step count, heart rate, weight, etc. As part of our future work, we plan to sync the web application with the database so that the doctor can view real time data collected from the patient's phone sensor to track patient activity.

6.2 Thematic approach for Graphical User Interface

In our initial mock up, we developed a basic theme for our interface based on the three colors: Red, Black and Grey. We further used green for the scheduling page, which is yet to be implemented. This decision was influenced by our logo which is taken from the logo of Rutgers University.

In keeping with this theme, our current UI for the mobile app 'Black' for text inputs and 'Grey' for visual or textual cues. In the Step counter page, we used 'Green' and 'Red' for the detecting buttons to give visual cue for the type of function (green denotes start, red denotes stop). We employed the same principle for the Heart rate monitor page. The Android icon and screen remains green initially. As the user puts a fingertip on the camera sensor, the icon on the screen turns red to denote that sensor data is processing.

We further plan to use this color-themed cue for the activity page. In the initial mock up we used horizontal progress bar to visually represent user's accomplishments for the day. The ides was: progress bar will represent daily target (for steps, distance or calories) for the user while the already accomplished portion will be in green and the remaining portion of target for the day will be displayed in red. However, in our current design, we used round progress bar as we find this will provide more intuitive feedback for the user.

We plan to work towards making the interface more easy to navigate, easy to use and learn for the rest of the semester. A simple, yet functional and intuitive user interface is crucial for the mobile application of our system, as it involves user operating on it during workout. It is not easy to perform simple tasks on a touch screen when a person is walking fast or jogging. While it can be argued whether the design approach we are taking is the best way for minimize user effort and maximize ease of use, we do state that within the resource constraints and trade-offs and in our better judgement, we have tried to provide the optimal user-centric design solution.

7 Design of Tests

Testing design is a number of techniques that add certain testability features to the system. Testing design is often viewed as executing a program to see if produces the correct output for a given input which wrongly implies that testing should be postponed until late in the lifecycle[1]. Actually, errors are introduced during the early stages of the software lifecycle, hence, testing activity should be started as soon as possible.

Testing uses different combinations of inputs to detects faults, but trying all possible inputs exhaustively will incur in a huge cost. In order to detect enough faults within a limited cost, we decide to make the testing design to follow the hierarchical structure of the system. We will start the test from individual components, unit testing, then the composition of these components, integration testing, and finally the whole system, system testing, ensuring the system suits the function and non-functional requirements.

So there are three different kind of testings that should be made here:

1. **Unit testing:** For unit testing, we mainly use the strategy of boundary testing and for several specific cases, we just use the equivalence testing, like the input process in sign-in.

2. **Integration testing:** the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing.

3. **System testing:** is a level of the software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

We begin with the design of our unit testing cases. Because there are too many, we'll provide the basics on the most important units of each class.

7.1 Unit Testing

1. **Test Coverage: Sign In** (Android App, Web App)

Tested Unit	**LogIn**
Assumption	The application interface showed up and the sign in input screen is waiting for the user input.
Expected Input	User Email or User Name, User Password

Expected Output	Sign in successfully done with proper input, Sign in unsuccessfully done with improper, and make alter the same time.
Pass	Function fit the requirement.
Fail and The Probable Error	1. Sign in still success with improper input. Error related input checking. 2. Sign in success with proper input but no related data show in database. Error related to connection with database.

2. **Test Coverage: Registration** (Android App, Web App)

Tested Unit	**Register**
Assumption	The Application interface showed up. The signUp input screen is waiting for the user input.
Expected Input	User Email, User Name ,User Password
Expected Output	Sign up and Information registration to database successfully done with proper input, Sign up unsuccessfully done with improper, and make alter the same time.
Pass	Function fits the requirement.
Fail and The Probable Error	1. Sign Up still success with improper input. Error related input checking. 2. Sign Up success with proper input but no functional connection was found , Error related to connection with database.

3. **Test Coverage: Step Counter** (Android App)

Tested Unit	**Step-Counter**
Assumption	User selects Step Counter Activity. Application is waiting for action
Expected Input	Click on start button
Expected Output	Step Counting-Miles Counting and Calories Burnt Calculations starts successfully and turns to other functions.
Pass	Step Counting-Miles Counting and Calories Burnt Calculations shows real and Actual calculations and Numbers.
Fail and The Probable Error	1. Click on start button, but nothing happens. Errors related training start function.

	2. Click on start button, but No Hardware Detected Message Showed up..

4. **Test coverage: Heart Rate** (Android App)

Tested Unit	**Heart Rate**
Assumption	User selects Heart Rate Activity. Application is waiting for user's actions.
Expected Input	User's fingerprint.
Expected Output	Number of heartbeats per minute (BPM).
Pass	The BPM corresponds accurately to that of the user.
Fail and The Probable Error	1. Incorrect BPM. Error related to fail in the algorithm. 2. No BPM is given. Error related to absence of finger to read.

5. **Test Coverage: HistoryActivity** (Android App)

Tested Unit	**GetUserVitalSigns**
Assumption	User selects the option to see his/her activity history for a given time.
Expected Output	All of the user activities in the selected period.
Pass	All relevant information is showed.
Fail and The Probable Error	1. Some information is missing. Error related to connection with the database.

6. **Test coverage: DashboardActivity** (Android App)

Tested Unit	**LogOut**
Assumption	User is in the Dashboard Activity and selects the Logout function.
Expected Output	User profile is closed. Application shows Main Activity.
Pass	User's profile is successfully logged out of the system. No user's information is left.

Fail and The Probable Error	1. Display shows is logged out but when returning to the application, the profile is still open. Error related to connection to the database.

7. **Test coverage: PatientInfo** (Web app)

Tested Unit	**GetPatientList**
Assumption	User in the main page selects, get patient list.
Expected Output	Webpage navigates to a page that returns a list of patients that that have selected this physician.
Pass	User's profile is successfully logged out of the system. No user's information is left.
Fail and The Probable Error	1. Patient list returns no data. 2. Patient list does not reflect the patients that selected the doctor. 3. Patient lists includes additional patients.

-

8. **Test coverage: PatientInfo** (Web app)

Tested Unit	**GetVitalSignsWeb**
Assumption	User in the patient list page, selects the desired patient
Expected Output	Webpage navigates to summary page with patients health information.
Pass	Summary page returns the correct information.
Fail and The Probable Error	1. Patient health data is not returned. 2. Incorrect health information is displayed. 3. Webpage navigates to incorrect page.

7.2 Integration Testing

7.2.1 Strategy

For integration testing, the basic hierarchy of levels is described below:

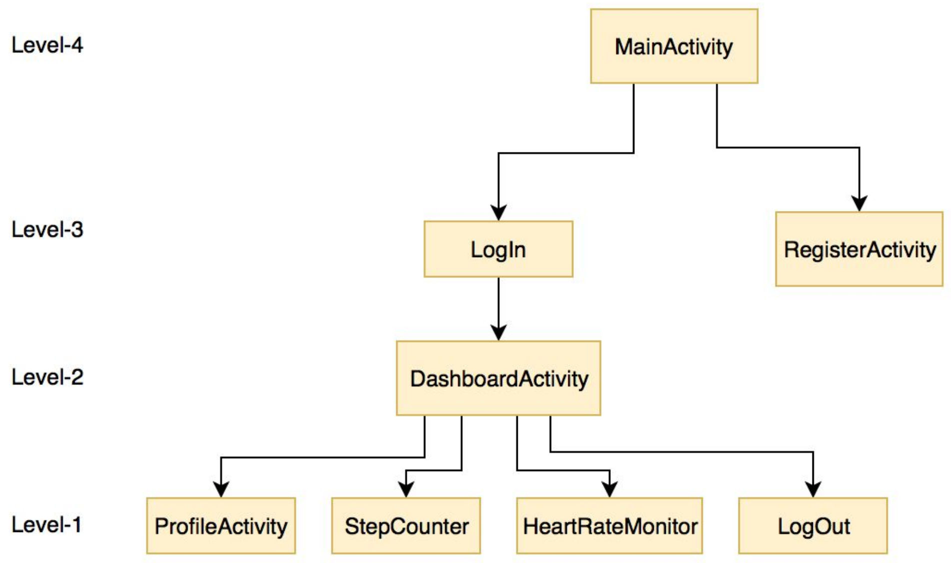


Figure 7.2: Testing Hierarchy

For simplicity, the easiest way of integration testing is going to be “big bang”, in where we link all the units and to try to test it at once. Every unit has been already tested individually, so their integration is expected to produce good results.

7.2.2 Plans

<u>Unit Testing</u>

For unit testing we completed automated tests. We did this by creating java code to run failure scenarios. The automated code for the registration module tested the following conditions:

1. whether a valid email address (with ‘@‘) is accepted as input and results in pass.
2. whether an invalid email address is rejected as input and results in fail.
3. whether a name that contains an invalid character (‘*’) results in fail.
4. whether a password containing invalid character ‘^#%’ results in fail.

Fore the remaining unit tests, we intend to continue with this strategy.

Integration Testing

Up to this point, most of our integration testing has been done manually. For manual tests, we tested the functionality of each use case by navigating through the specific pages and monitoring the results. For example for the registration we followed the steps below:

1) Open application main page
2) Navigate through the registration process
3) Complete registration

   Test Criteria

4) Verify page navigation
5) Verify database storage

We intend to extend the test criteria to include more stringent test conditions. For example, we will test the heart rate monitoring under non-optimal conditions. These manual tests will follow the basic steps below:

1) Open application main page
2) Navigate through the appropriate processes process
3) Verify that results meet criteria

8 Project Management and Plan of Work

8.1 Merging the Contributions from Individual Team Members

Our report documentation is available in a google drive that to which everyone has access. Each week Tina formats the reports with section headers and outlines so that each group member can edit the document with their contributions. The reports are edited and merged collaboratively using Google docs. Tina and Tahiya are usually responsible for formatting, style consistency, and modifications to the document appearance.

The report and code distribution are decided collaboratively among the group. One of the challenges that we faced was evenly distributing the report and coding so that everything would be responsible for a relatively equal amount of points. We attempted to divide tasks according to people's strengths while not overwhelming any particular members of the group. We addressed this challenge of point distribution by reviewing the points weekly and assigning sections accordingly.

As far as code distribution we delegated tasks according to each team member's strength. Ramya and Himabindu have have experience with socket programming; so they are responsible for code integration. George and Aymen have have strong programming skills; therefore they were responsible for various code sections including sensor detection and step counting. Additionally, they have completed some of the project management tasks such as maintaining the Github site. Tahiya is specializes in user interface; so, she has been assigned with the responsibility of the UI pages. Moreover, she has served as the project leader and completed various project management tasks such as coordinating meetings. Tina does not have a strong programming background but has experience in program management. Thus, she has been involved in a lot other project management activities including report formating and google drive management. She has also been responsible for much of the web development.

8.2 Project Coordination and Progress Report

8.2.1 Use case progress

Implemented Use Cases

We have implemented the functionality of the application that are the central part of the project: the collecting information from the phone sensors, the ability to monitor health data, and granting user access to this health data. This has resulted in the following use cases being implemented:

UC-1: Toggle Activity
UC-2: Health Data
UC-3: Get Vital Signs Phone
UC-12: User Sign-up
UC-13: User Sign-in.

Use Cases Under Development

Currently we are tackling online database management. We have integrated the physician registration to the website page with Firebase. So, when the physician registers, their data, including username and password, is stored in the Firebase database. However we have not linked the database to user sign in yet. We need to add capability to verify the sign in credentials based on information in the Firebase database. Moreover, we need to include the Firebase database on the android side for the user. Right now the patient data is stored locally in the SQL database on the Android. In order for physician to access the information via the web application it should be available and updated online. So, we are left with the following use cases under development:

UC-4: Get Vital Signs Web
UC-11: Physician Sign-in

Non-implemented Use Cases

Due to time constraints, we are not able to implement all of the use cases. As a result, we decided to exclude some of the optional use cases. These non-implemented cases are listed below:

UC-5: Enable Notifications
UC-6: Get Notes
UC-7: Manage Schedule

UC-8: Toggle Alarm
UC-9: Enable Weekly Updates
UC-10: Send Notes

8.2.2 Project Coordination Activities

Basic information

Team Leader: Tahiya
Primary Communication Method: WhatsApp
Meeting Time: Wednesdays, 11:30 am to 12:00 am
Meeting Location: CoRE 100
Document Repository:
https://drive.google.com/drive/folders/0B0NuBl5TDP7_N2RaRk9BaUEyOTA?usp=sharing
Project Repository: https://github.com/karahbit/RU_Healthy

Activities

As a group, much of our time has been devoted to completing documentation required for the class assignments. Additionally, we have worked on the project management portion by mapping out task assignments. We have all contributed to these sections. We meet weekly in person or by Whatsapp to delegate tasks and share information about our current coding status. The bulleted list outlines the non-coding tasks that we have completed:

- Proposal - **All**
- Report 1, Part 1 - **All**
- Report 1, Part 2 - **All**
- Report 1, Full report - **All**
- Report 2, Part 1 - **All**
- Report 2, Part 2 - **All**
- Demo Documents - All
- Coordination of team meetings - **Tahiya**
- Delegation of report sections - **All**
- Delegation of coding assignments - **All**
- Delegation of report updates - **All**
- Delegation of demo assignments - **All**
- Creation of Github Folder - **Aymen, George**
- Organizing Github Folder - **Aymen, George**
- Uploading Documents to Github Server - **All**
- Creation of Github site for webpage - **Tina**
- Management Github website repository - **Tina**
- Gantt chart - **Tina**
- Creation of firebase account for website database management - **Tina**

8.3 Breakdown of Coding Responsibilities

Each group member has contributed to coding, developing, and testing some portion of the project. The delegations of these modules and pages are listed below:

Android Modules
MainActivity.java - **Ramya/Himabindu**
RegisterActivity.java - **Ramya/Himabindu**
ProfileActivity.java - **Ramya/Himabindu**
DashboardActivity.java - **Ramya/Himabindu**
SQLiteHelper.java - **Ramya/Himabindu**
StepCounter.java - **Aymen/George**
StepDetector.java - **Aymen/George**
StepListener.java - **Aymen/George**
SensorFilter.java - **Aymen/George**
HeartRateMonitor.java - **Ramya/Himabindu**

Android UI pages
Activity_dashboard.xml - **Tahiya**
Activity_main.xml - **Tahiya**
Activity_register.xml - **Tahiya**
Hrm.xml - **Tahiya**
Profile.xml - **Tahiya**
Step_count.xml - **Tahiya**

Website Code
Login.html - **Tina**
Registration.html - **Tina**
ProfilePage.html - **Tina**
FirebaseConfig.js - **Tina**
FirebaseOperationsUser.js - **Tina**

Each member will contribute in some aspect to the code integration. Ramya and Himabindu will be responsible for socket programming and integrating different functional sections of the android code. Tahiya will be responsible for integrating the user interface with the androd pages. Tina will be responsible for integration the website with the android database. George and Aymen will be integrating the test code with the Android system. Additionally each group member will contribute to the integration testing.

8.4 Plan of Work

8.4.1 Projected tasks

In the next few weeks we will prepare the documentation and demonstration for the class assignments. The list below details the tasks and projected dates for these items

Class Assignments

- Report 3 - **Dec. 10**
- Demo 2 - **Dec. 13**
- Electronic Project Archive - **Dec. 16**

Web Application

- Add password credential check to registration page - **Nov 15**
- Link user login and registration information to database - **Nov 20**
- Link patient information page to data on android server - **Nov 30**

Android Application

- Fix bugs Stop of Submit from profile page bug - **Nov. 18**
- Integrate user information into Firebase database - **Nov. 22**

Testing

- Unit Testing - **Dec 1**
- Integration testing - **Dec 10**

Additional tasks, projected dates, and task assignees are included in the Gantt chart in available in Microsoft project.

8.4.2 Gantt Chart

We discovered that the software does not easily allow users to output in a format that would be easily readable in this report. As a result, we added the Microsoft Project PDF output as Appendix A at the end of this report.

9 References

1) http://answers.google.com/answers/threadview?id=758572%22
2) http://www.mayoclinic.org/healthy-lifestyle/fitness/in-depth/exercise/art-20048389
3) https://www.cbsnews.com/news/cdc-80-percent-of-american-adults-dont-get-recommended-exercise/
4) http://www.upedu.org/process/gcncpt/co_req.htm
5) https://en.wikipedia.org/wiki/FURPS
6) https://www.merriam-webster.com/dictionary/obesity
7) https://www.tomsguide.com/us/pictures-story/482-best-fitness-apps.html#s10
8) https://www.cdc.gov/physicalactivity/basics/adults/index.htm
9) Sullivan, Alycia N. and Lachman, Margie E. (2017), Kostkova, Patty, ed., "Behavior Change with Fitness Technology in Sedentary Adults: A Review of the Evidence for Increasing Physical Activity", Front Public Health 4: 289, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5225122/
10) National Center for Health Statistics. Health, United States, 2016: With Chartbook on Long-term Trends in Health. Hyattsville, MD. 2017.
11) http://myphonefactor.in/2012/04/sensors-used-in-a-smartphone/
12) https://healthyceleb.com/wp-content/uploads/2012/10/Health-app-android-iOS.jpeg
13) https://www.google.com/search?biw=1240&bih=572&tbm=isch&sa=1&q=health+apps&oq=he&gs_l=psy-ab.3.0.0i67k1l4.73873.76835.0.78238.16.13.1.0.0.0.264.1860.0j5j4.10.0....0...1.1.64.psy-ab..10.5.874.0..0.147.QsrScBdLFfs#imgrc=C6IWjX4kIqVArM:
14) https://www.wrike.com/project-management-guide/faq/what-is-a-stakeholder-in-project-management/
15) Ivan Marsic, (2012). Written at Rutgers University, New Brunswick, New Jersey, *Software Engineering*, http://www.ece.rutgers.edu/~marsic/books/SE/
16) https://www.cmcrossroads.com/sites/default/files/article/file/2013/XUS234033087file1_0.pdf
17) https://en.wikipedia.org/wiki/System_sequence_diagram
18) http://photos-ak.sparkpeople.com/7/0/b707669348.jpg
19) http://stg-tud.github.io/eise/WS14-EiSE-12-System_Sequence_Diagrams.pd
20) Cameron et al, (2004), "Energy Expenditure of Walking and Running", Medicine & Science in Sport & Exercise.
21) Design Principles, Object Oriented Design, http://www.oodesign.com/design-principles.html
22) www.uml.org

Appendix A

This Appendix contains the Gnatt chart of the project tasks for the RU Healhy? App