

# SENG 300: Assignment 3

Celina Ma, John Ngo, Omar Qureshi

June 21, 2018

## Deliverable #1

Need to adjust this section for our software process model.

### 1-1: Functional Requirements (FRs)

1. "The app must be able to receive data from the muon detector."

This app must be able to communicate with the external piece of hardware in order to meet the criteria given by the client.

2. "The app must be able to display a 'live' reading from the muon detector."

Along with receiving input from the detector, it then must be able to display the stream of values outputted by the detector onto the phone.

3. "The app must be able to average out the readings when pressing the 'Summary' button"

While a live reading is displayed, the 'Summary' button takes a snapshot of the readings average after recording is finished.

4. "The app must be able to record the averaged 'get readings' in a table."

After pressing the 'get reading' button, the value is averaged and stored into a table that is accessible in another part/menu of the application.

5. "The app must be able to export out the saved data to a computer so that further data interpretation and processing can be performed."

The client has specified that the readings once in tabled format must be able to export (via .csv for example) to another computer. This would progress the learning experiences of future students who want to work with the data more comfortably once it has been collected via the phone.

## 1-2: Non Functional Requirements (NFRs)

1. "The app should present the readings with three significant digits."

This allows the user to experience a non cluttered UI, removing unnecessary information. This NFR would fall under the usability.

2. "The app should record the readings by making a smooth visual animation."

A transition animation is often desired to express user action in a clearer way. This NFR would fall under the usability category.

3. "The app should record the readings within 0.5s after pressing the 'get reading' button."

This ensures the user doesn't have to wait an unusually large amount of time to get the reading. It also stops the user from getting frustrated and pressing the button again needlessly. This NFR would fall under the response time category.

4. "The app should emit a feedback sound when the user presses the 'get reading button'"

Giving the user direct confirmation of their action being applied is helpful and again stops user from pressing the button unnecessarily. This NFR would fall under the usability category.

5. "The app should present a message if the muon detector is connected or not."

This makes sure that a precondition of using the app is met, which is having the miniUSB cable plugged into the phone and detector. This NFR would fall under the learnability/usability category.

## **Deliverable #2**

After considering both use cases and user stories, we conclude that the user stories method is a better way to represent the systems requirements for our project. Just as with the selection of our software process model, this comes from consideration of our circumstances compared to the strengths and weaknesses of each model.

The largest, most important draw of the user stories model is that it is extremely simple and to the point. Its structure, where a given user wants to do something for a given reason, is very, very close to how our client expresses what he wants out of the software, and as such is extremely attractive. Along with the emphasis on action and reason, user stories allow for the communication of overarching requirements goals as opposed to more nitty-gritty details, which fits with the iterative model of development we have chosen, since iterative models work towards a general end goal but don't care about details at each stage.

The largest, most important draw of the use cases model is that done correctly, it can map out the structure of the very application needed, at least on some level. However, this backbone given and its greater emphasis on specific details compared to user stories does not lend it quite as well to an iterative model; and furthermore, it is more technically designed than user stories, which are closer to natural language. As such, while not a terrible alternative, it just doesn't provide nearly as many advantages as with user stories.

Therefore, we have selected user stories to represent our system requirements.

### **2-2: User Stories**

The three core functional requirements that the user (in our case Professor Donev) needs to accomplish are as follows: connect muon detector to phone, view muon detector data and save data from muon detector to a log. As such, three corresponding story cards are created;

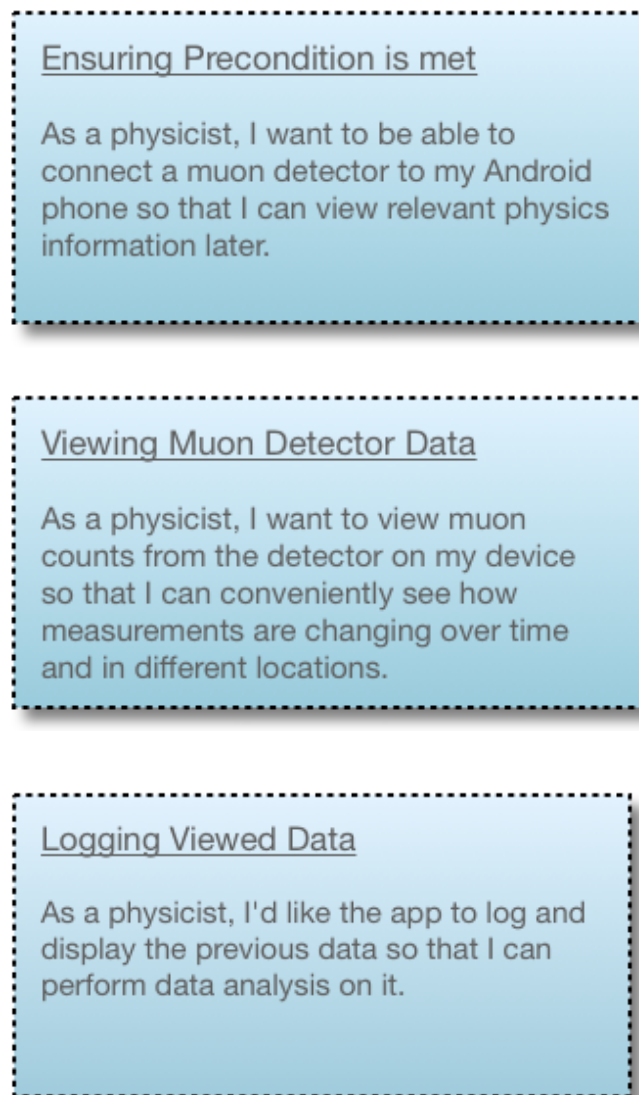


Figure 1: Individual story cards are first created to document core requirements, upon which further cards will be used to document how a user goes about accomplishing those tasks

## 2-3: Story Map

With the use of the user story cards above, a collage/story map can be created that details further on how a user can interact with the app to accomplish their requirements. The story map can be viewed on the next landscape page;

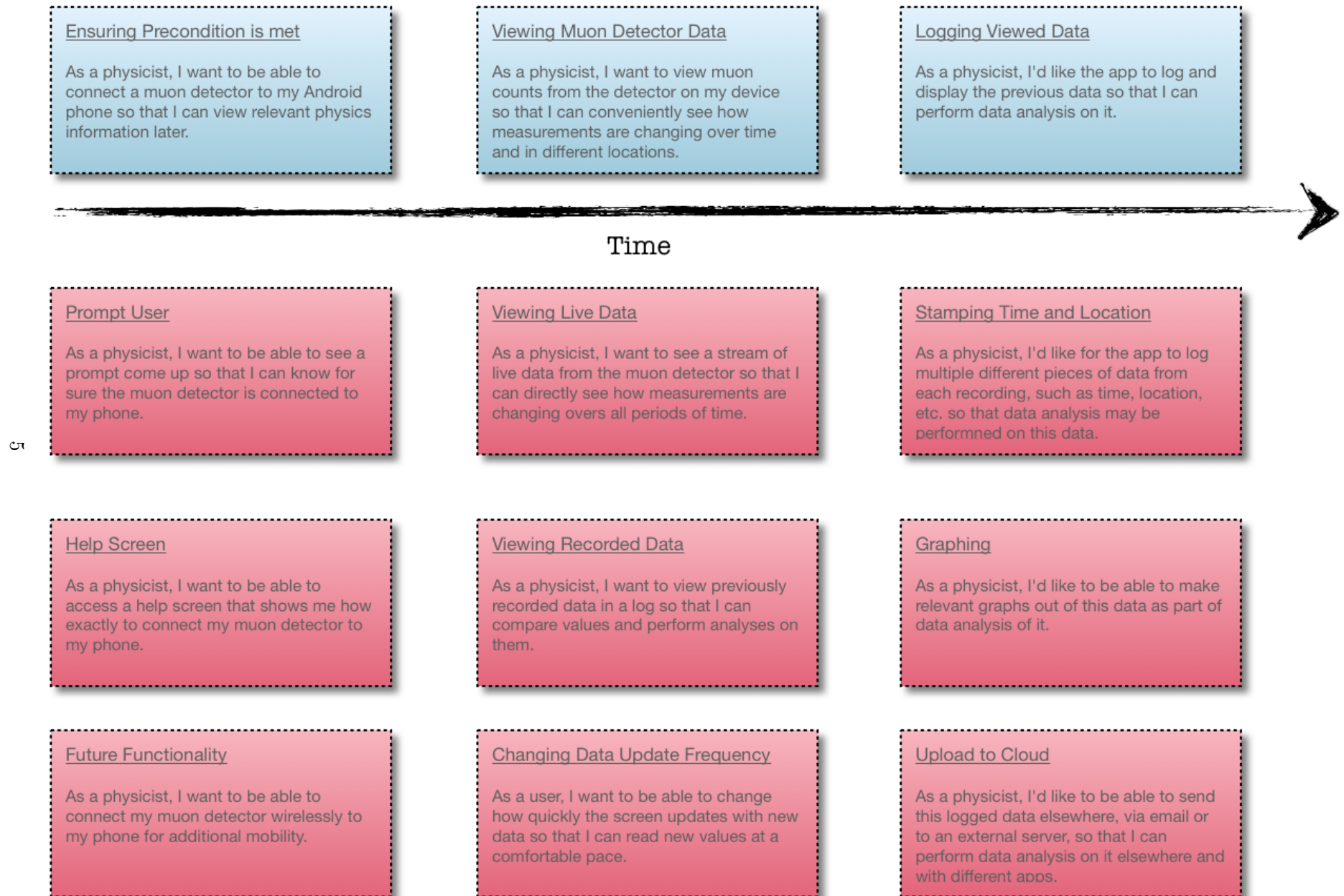


Figure 2: A story map is used to organize requirements in order of priority and to illustrate how user interacts with app

## Deliverable #3

### 3-1: Overview Sketches

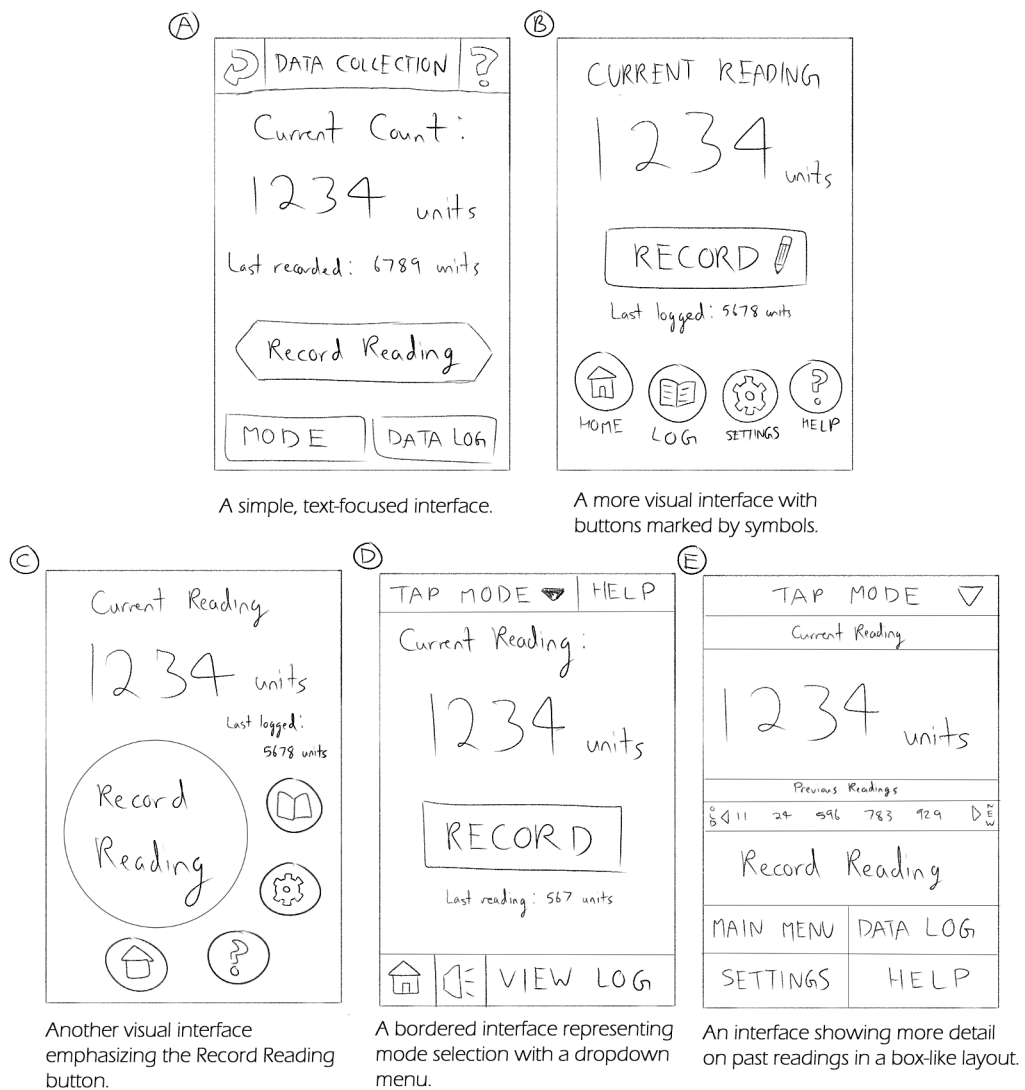


Figure 3: A variety of design approaches for the recording screen

## Why We Chose Sketch A

We chose sketch A to elaborate on due to its clarity and emphasis on the most important aspects of this screen. The main buttons the user would interact with (Record, Mode and Data Log) are large and displayed clearly. Compared to sketches B and C, which have smaller icons that the user may accidentally mis-press, the interface of A is more focused on what is important to the user. This is key to enhancing the intuitiveness of the app. Sketch D emphasizes similar points to sketch A, but the mode selection button is somewhat less obvious in this design. Although sketch E shows more information about Previous Readings than the other sketches, it is too cluttered; additionally, that information is repeated in the Data Log screen in more detail, so it is not needed for the Data Collection screen. Thus, our main reason for choosing sketch A was its relative simplicity and correctly emphasized functions compared to the other designs.

### 3-2: Elaborating Sketches

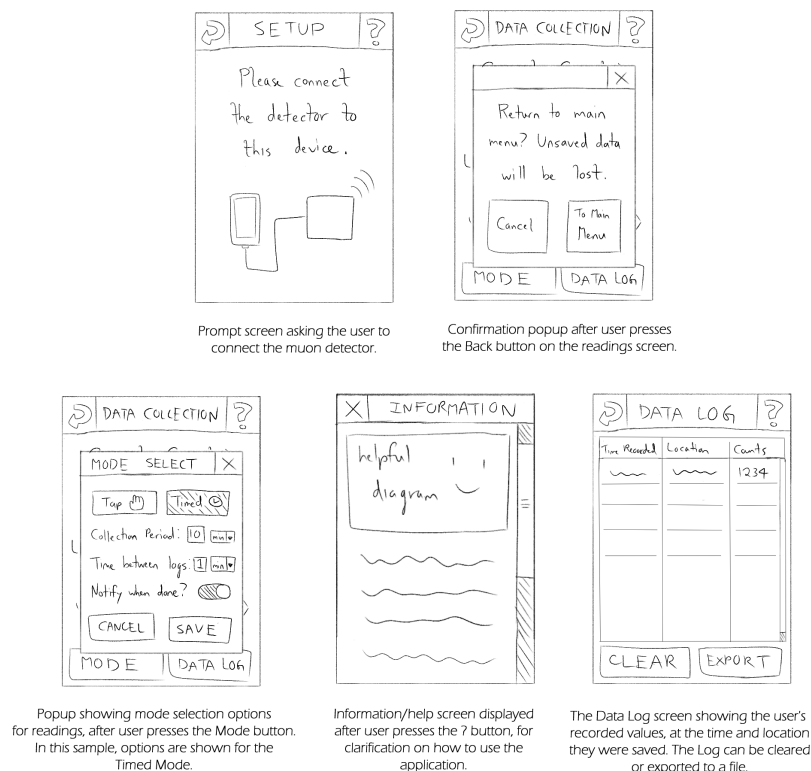


Figure 4: Extended sketches for other parts of the apps UI screens

### 3-3: Storyboard Sketches

#### Storyboard: Recording a new value to the Data Log

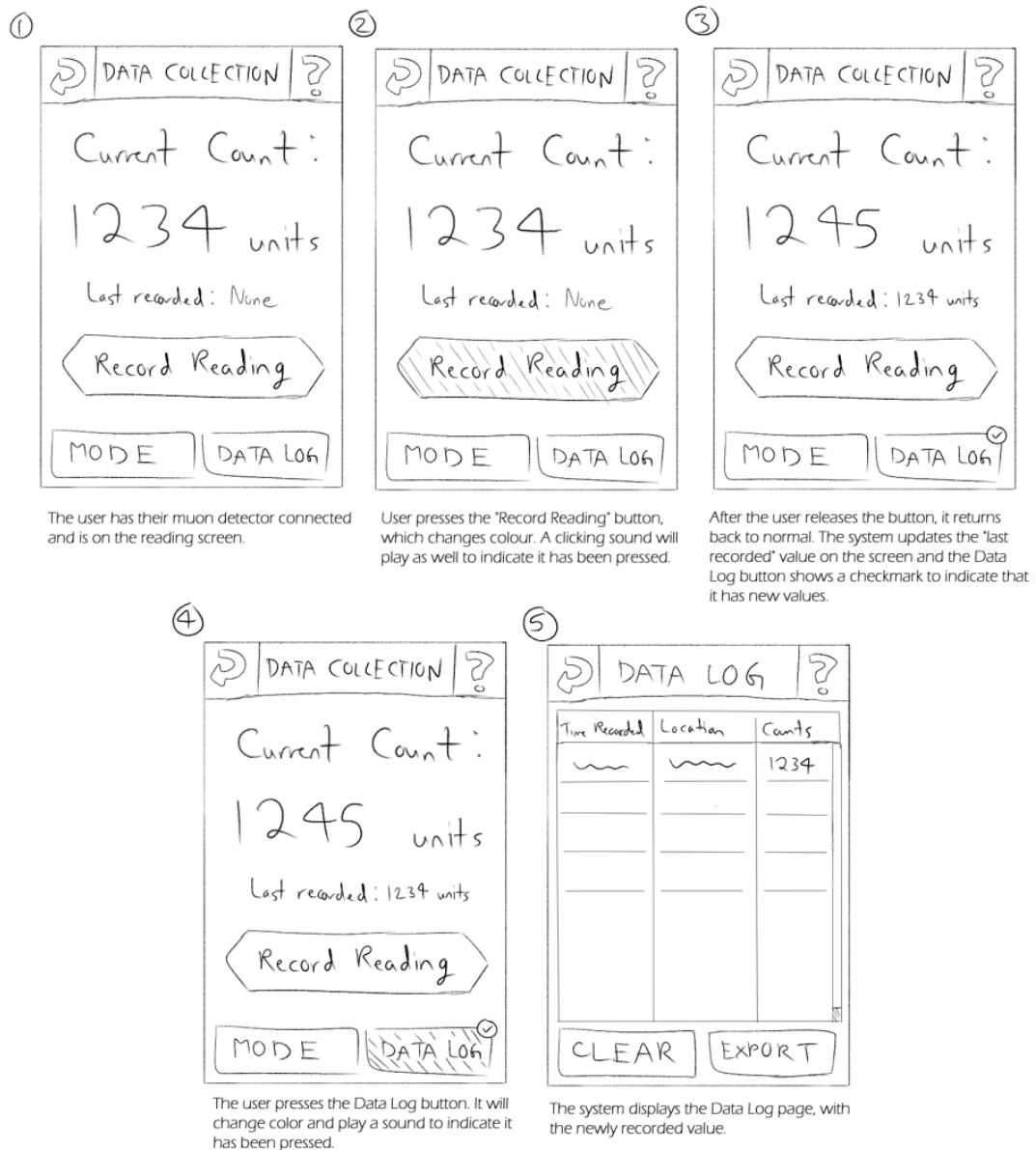


Figure 5: The recording screen responding to user input



### 3-4: Wizard of Oz Demo Video

Demo videos can be viewed below:

1. Full Video 30min ([link](#))
2. Short Video 1min ([link](#))

## Deliverable #4

### 4-1: Poker Effort Estimation

For our planning poker session, we chose to create tasks for the following functional requirement:

**The app must process event data from the muon detector to determine the number of events per minute over a specific time interval.**

We generated six tasks for this functional requirement and individually rated the difficulty of each task from one to ten. During our planning session, we generally had consensus for our ratings, with some variance between us.

For example, the task, “Create an addEvent method to store processed MuonEvents to an array”, was considered to be relatively easy by all of us. However, one person rated the difficulty slightly higher and suggested that this method may need to check if the event data is valid (eg. if the timestamps are in an order that makes sense). We discussed some other issues that may occur, such as what should happen if a maximum number of events had been reached. These additional factors helped clarify the difficulty of the task.

The task of “calculating the difference in minutes between two timestamps” was more divisive. One of our group members believed the task would be quite difficult (7) but another believed it would be easy (3), since Java likely contained some built-in functionality to help with the task. We learned that this was indeed the case and adjusted our rankings accordingly. We all agreed that the task of “calculating events per minute over a specific time interval” would be difficult. During our discussion, we contemplated how the number of events per minute could be displayed “live” on our app screen while collecting data, since that would involve frequently saving new timestamps for the calculation. We also wondered if events per minute should only be calculated over some fixed time interval (eg. the last 10 seconds of recording) or use the whole time spent recording.

Overall, the planning poker session was useful for evaluating our list of tasks by raising new questions on generating ideas for test cases and structure.

## 4-2: Silent Grouping Effort Estimation

Once our planning poker session was completed, we moved along to do silent grouping effort estimation. We now had to define features of the functional requirement rather than the tasks. The definition of a feature we used was an element the user can see or interact with, corresponding to the functional requirement defined in *Deliverable 1-1*. The features we came up with were:

1. The user is able to press the 'Start Recording' button to begin when data collection starts.
2. The user is able to see events being displayed on the recording screen after each new event is detected and added.
3. The user is able to stop the recording midway if desired.
4. The user is able to clear the screen once a session is done to start a new recording session.
5. The user is able to see when a recording is in progress by noticing the green Start Recording button changes into a red Recording (duration remaining in s) button.

After coming up with the above list, we wrote them down on some post-it notes and used a table rather than a wall to do the sorting;

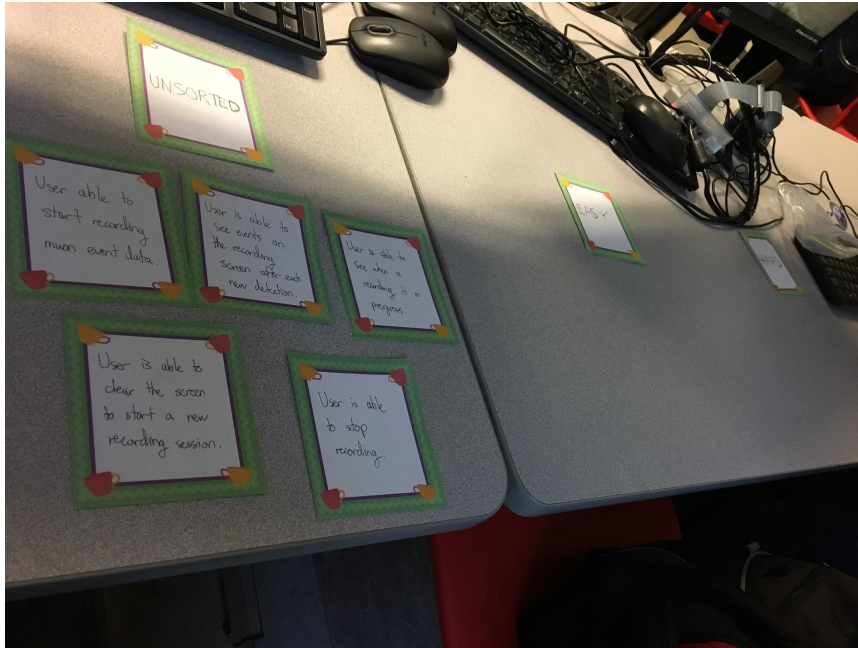


Figure 6: Post it notes started initially in the unsorted pile and a table was used for convenience

One of the issues we noticed with the silent grouping method was the lack of granularity of the piles. A feature was either easy or hard whereas the planning poker method was able to give a finer sense of difficulty. As such, clearly easy or hard features were the first to get picked and sorted which is seen in the below in progress shot;

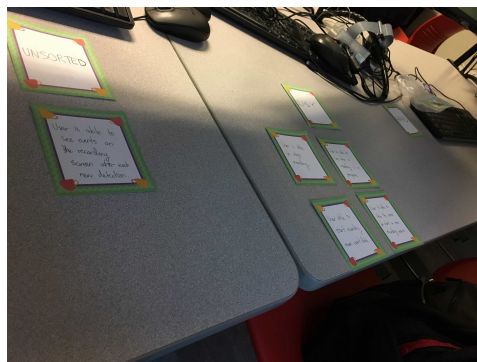


Figure 7: Silent grouping effort estimation in progress

We then attempted to categorize some of the harder features such as displaying the live event being detected and added to a running total. This aligned with our previously defined hard task in *Deliverable 1-1* and a consensus

was reached that this would indeed be difficult.

Afterwards, a feature that was hard to reach an agreement on was stopping the recording half way. Stopping it mid way may prove to be difficult so after a few back and forth for that feature, we decided to put it in a middle 'hold' section as seen below:

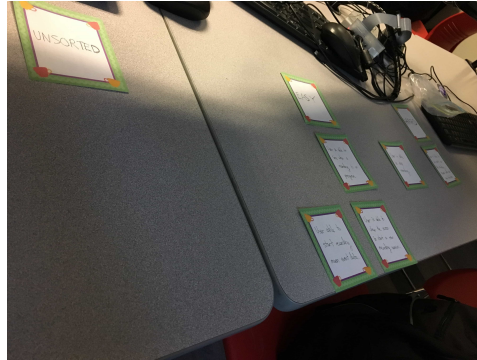


Figure 8: Completed session with one of the features being placed on hold

Overall, silent grouping proved to be a valuable medium of communication in allowing team members to interact with one another. An exchange of different views can ultimately help in creating a better application.

### 4-3: Task Breakdown

Functional Requirements	Short Name
As a physicist, I'd like to process data received from the muon detector to my phone	Data processing
As a physicist, I'd like my phone to be able to connect to the muon detector via miniUSB	USB Manager
As a physicist, I'd like to be able to view transmitted data from the muon detector onto my phone UI	Display View

Our first table shows the three main functional requirements picked for the app and their corresponding short names to be used in the following tables.

Task Name	Functional Requirement	Estimated Effort (person-hours)	KDLOC
Detect if a USB device is connected or not	USB Manager	2.0	0.02
Check: Correct USB device (optional?)	USB Manager	1.0	0.03
Adjust USB reading parameters to fit the correct device	USB Manager	1.0	0.05
Toggle on/off reading from the USB device	USB Manager	0.5	0.01
Read from the USB/ Device	USB Manager	3.0	0.02
Process USB Strings into usable information	USB Manager	5.0	0.10
Code to send to a processing class the event noticed; sending the signal detection upstream somehow	USB Manager	6.0	0.03
Read and Append data from this USB device into an array	USB Manager	0.5	0.01
TOTAL	-	19	0.27

Our second table corresponds to the USB Manager functional requirement. The table format serves as a visual aid for breaking down requirements into tasks and effort, which also provides a documentation reference in the future for cost estimation.

Task Name	Functional Requirement	Estimated Effort (person-hours)	KDLOC
Create a MuonEvent class to store information about individual events (eg. timestamp, location, altitude)	Data processing	0.25	0.01
Create appropriate getter methods for the MuonEvent class	Data processing	0.5	0.02
Create an addEvent method to store processed MuonEvents to an array	Data processing	0.5	0.01
Create a clearEvents method to delete unneeded event data	Data processing	0.25	0.005
Create methods to store the timestamps when a data collection period begins and ends	Data processing	0.5	0.01
Create a method that calculates the difference in time (minutes) between two timestamps	Data processing	1.0	0.015
Create a getEventsPerMin method that examines a MuonEvent array and calculates how many events occurred per minute over a specific time interval	Data processing	1.0	0.02
<b>TOTAL</b>	-	4	0.09

Our second requirement is similarly broken down into individual tasks and their corresponding estimated effort.

Task Name	Functional Requirement	Estimated Effort (person-hours)	Estimated Lines of Code
Create an introductory menu to the app with two buttons near the centre middle with title 'Muon Detector' on top, left justified bolded	Display View	1.0	0.05
Create an 'About' button on menu	Display View	0.25	0.01
Create an about screen with minor author details that is able to go back to menu screen, 'About' title top left justified bolded	Display View	0.5	0.05
Create a 'Start' button on menu	Display View	0.25	0.01
Create a recorder screen with a 'Start Recording' button at the bottom, 'Detector' title top left justified bolded	Display View	2.0	0.07
Create a 'Start Recording (1 min)' button with green background, centre bottom of the screen	Display View	0.25	0.01
Create a 'Recording (time remaining in s)' button with red background while a reading is in progress	Display View	0.25	0.02
Show events (if any) in text centre left justified as recording progresses	Display View	0.5	0.01
<b>Total</b>	-	5	0.23
<b>GRAND TOTALS</b>		28	0.59

Finally, our third table for the final display functional requirement is also broken down.

Overall, this exercise in the design process helped create estimates on the challenges to be faced when completing functionality of the application. This also helped plan our schedules since we now had a general idea of time required for each step.

## Deliverable #5

### 5-1: Initial Tests

For the data processing requirement, we will focus on testing the Processor class. This class will store event data as MuonEvents occurring over some time period. Here, we see seven initial test cases that focus on checking if the correct number of events have been stored (by the `getEventCount()` method) and on calculating the number of events per minute. None of the test code is complete save for the expected values:

```
class ProcessorTest {  
  
    @Test  
    void startTimeTest() {  
        Date date = new Date(2018, 6, 30, 6, 30, 0);  
        Processor data = new Processor();  
        assertEquals(date, data.getStartTime());  
    }  
  
    @Test  
    void eventCountFifty() {  
        Processor data = new Processor();  
        assertEquals(50, data.getEventCount());  
    }  
  
    @Test  
    void eventCountLarge() {  
        Processor data = new Processor();  
        assertEquals(500, data.getEventCount());  
    }  
  
    @Test  
    void eventCountZero() {  
        Processor data = new Processor();  
        assertEquals(0, data.getEventCount());  
    }  
  
    @Test  
    void eventsPerMinFive() {  
        Processor data = new Processor();  
        assertEquals(5.0/6.0, data.getEventsPerMin());  
    }  
  
    @Test  
    void eventsPerMinOne() {  
        Processor data = new Processor();  
        assertEquals(1.0/60, data.getEventsPerMin());  
    }  
  
    @Test  
    void eventsPerMinNone() {  
        Processor data = new Processor();  
        assertEquals(0, data.getEventsPerMin());  
    }  
}
```



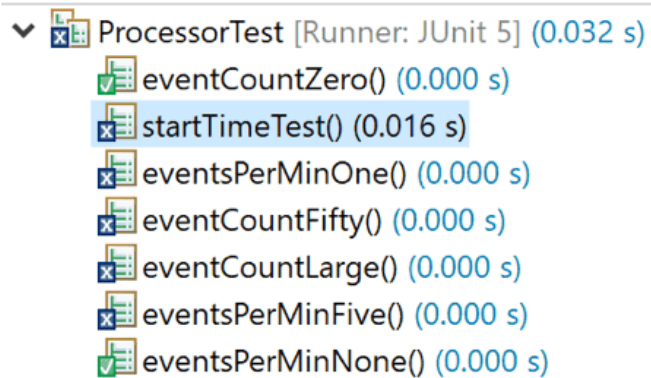


To compile the test code, a basic skeleton of the Processor class was created:

```
public class Processor {  
    public int getEventCount() {  
        return 0;  
    }  
  
    public Date getStartTime() {  
        return null;  
    }  
  
    public double getEventsPerMin() {  
        return 0.0;  
    }  
}
```

When the previously created test class was now run, all but two of the tests failed. The only tests that passed already expected "0" as a result:

Runs: 7/7    ✖ Errors: 0    ✖ Failures: 5



ProcessorTest [Runner: JUnit 5] (0.032 s)

- ✓ eventCountZero() (0.000 s)
- ✖ startTimeTest() (0.016 s)
- ✖ eventsPerMinOne() (0.000 s)
- ✖ eventCountFifty() (0.000 s)
- ✖ eventCountLarge() (0.000 s)
- ✖ eventsPerMinFive() (0.000 s)
- ✓ eventsPerMinNone() (0.000 s)

## 5-2: Tests with Partial Implementation

While working on the implementation, we realized that we needed more data and methods to flesh out the Processor class, resulting in an increased number of test cases. For example, to calculate events per minute, we needed to save the timestamps when data collection began and ended to know the length of time spent recording data. This functionality is performed by the new switchRecording method.

Although the full implementation of this requirement will use real muon event data from a connected muon detector, the test cases created fake data to simulate how real events would be saved by the Processor.

The following screenshots show some of the test cases after implementation:

```
class ProcessorTest {  
  
    @Test  
    void startTimeTest() {  
        Date date = new Date(2018, 6, 30, 6, 30, 0);  
        Processor data = new Processor();  
        data.switchRecording(date);  
        assertEquals(date, data.getStartTime());  
    }  
  
    @Test  
    void stopTimeTest() {  
        Date startDate = new Date(2018, 6, 30, 6, 30, 0);  
        Processor data = new Processor();  
        data.switchRecording(startDate);  
  
        Date endDate = new Date(2018, 6, 30, 6, 35, 0);  
        data.switchRecording(endDate);  
  
        assertEquals(endDate, data.getStopTime());  
    }  
  
    @Test  
    void eventCountFifty() {  
        Processor data = new Processor();  
        for (int i = 0; i < 50; i++) {  
            data.addEvent();  
        }  
        assertEquals(50, data.getEventCount());  
    }  
  
    @Test  
    void timeDifOneMin() {  
        Processor data = new Processor();  
        Date startTime = new Date(2018, 6, 30, 6, 29, 0);  
        Date endTime = new Date(2018, 6, 30, 6, 30, 0);  
  
        double diff = data.timeDifference(startTime, endTime);  
        assertEquals(1.0, diff);  
    }  
}
```

The new `timeDifference` method was created to calculate the difference between two timestamps in minutes. This is used in conjunction with the `getEventsPerMin` method:

```
@Test
void eventsPerMinFive() {
    Processor data = new Processor();

    Date startTime = new Date(2018, 6, 30, 6, 29, 0);
    data.switchRecording(startTime);

    for (int i = 0; i < 5; i++) {
        data.addEvent();
    }

    Date endTime = new Date(2018, 6, 30, 6, 35, 0);
    data.switchRecording(endTime);

    assertEquals(5.0/6.0, data.getEventsPerMin());
}

@Test
void eventsPerMinOne() {
    Processor data = new Processor();
    Date startTime = new Date(2018, 6, 30, 6, 29, 0);
    data.switchRecording(startTime);

    Date time = new Date(2018, 6, 30, 6, 30, 0);
    data.addEvent();

    Date endTime = new Date(2018, 6, 30, 7, 29, 0);
    data.switchRecording(endTime);

    assertEquals(1.0/60, data.getEventsPerMin());
}
```

The Processor class now has partially complete methods. However, it does not yet store instances of each muon event:

```
public class Processor {

    private Date startTime;
    private Date stopTime;

    private int eventCount;    // Number of events over a collection period

    private boolean isRecording = false;

    /**
     * Gets the number of events that have occurred since recording began.
     */
    public int getEventCount() {
        return eventCount;
    }

    /**
     * Records a new instance of a muon event.
     */
    public void addEvent() {
        eventCount++;
    }

    /**
     * Returns when the last data collection period started.
     */
    public Date getStartTime() {
        return startTime;
    }

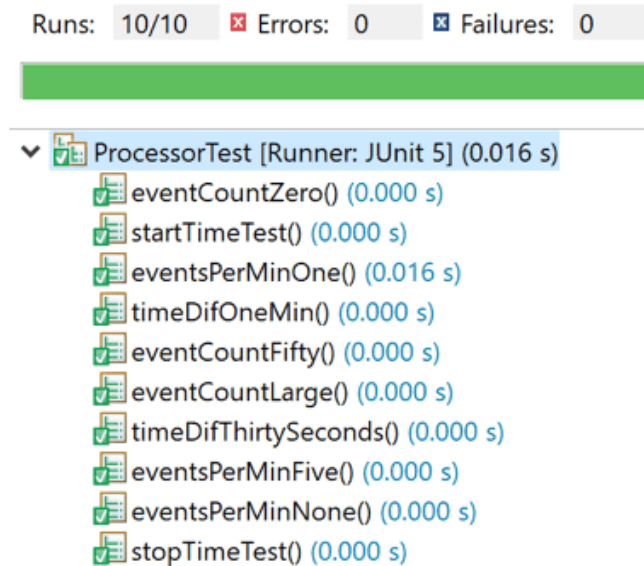
    /**
     * Called when the user chooses to Start/Stop Recording. Saves the appropriate timestamp
     * for starting or ending recording and toggles the isRecording flag.
     * @param timestamp the time the user chose to Start/Stop Recording
     */
    public void switchRecording(Date timestamp) {
        if (isRecording) {
            stopTime = timestamp;    // Stop recording
        } else {
            startTime = timestamp;    // Start recording
            stopTime = null;
        }
        isRecording = !isRecording;    // Toggle recording
    }

    /**
     * Calculates and returns the current events/minute based on the total number of events
     * that occurred during the last data collection period, and the time that has elapsed.
     * @return
     */
    public double getEventsPerMin() {
        Date endTime;

        if (isRecording) {
            endTime = new Date();    // If this is called while recording, use current time for calculation
        } else {
            endTime = stopTime;    // Else use the time the last recording ended
        }

        double result = (double)(eventCount)/timeDifference(startTime, endTime);
        return result;
    }
}
```

Now there is enough code to pass all the test cases:



### 5-3: Tests with Full Implementation

Our demo video demonstrating the TDD approach with full implementation along with explanations can be viewed at the following ([link](#)).

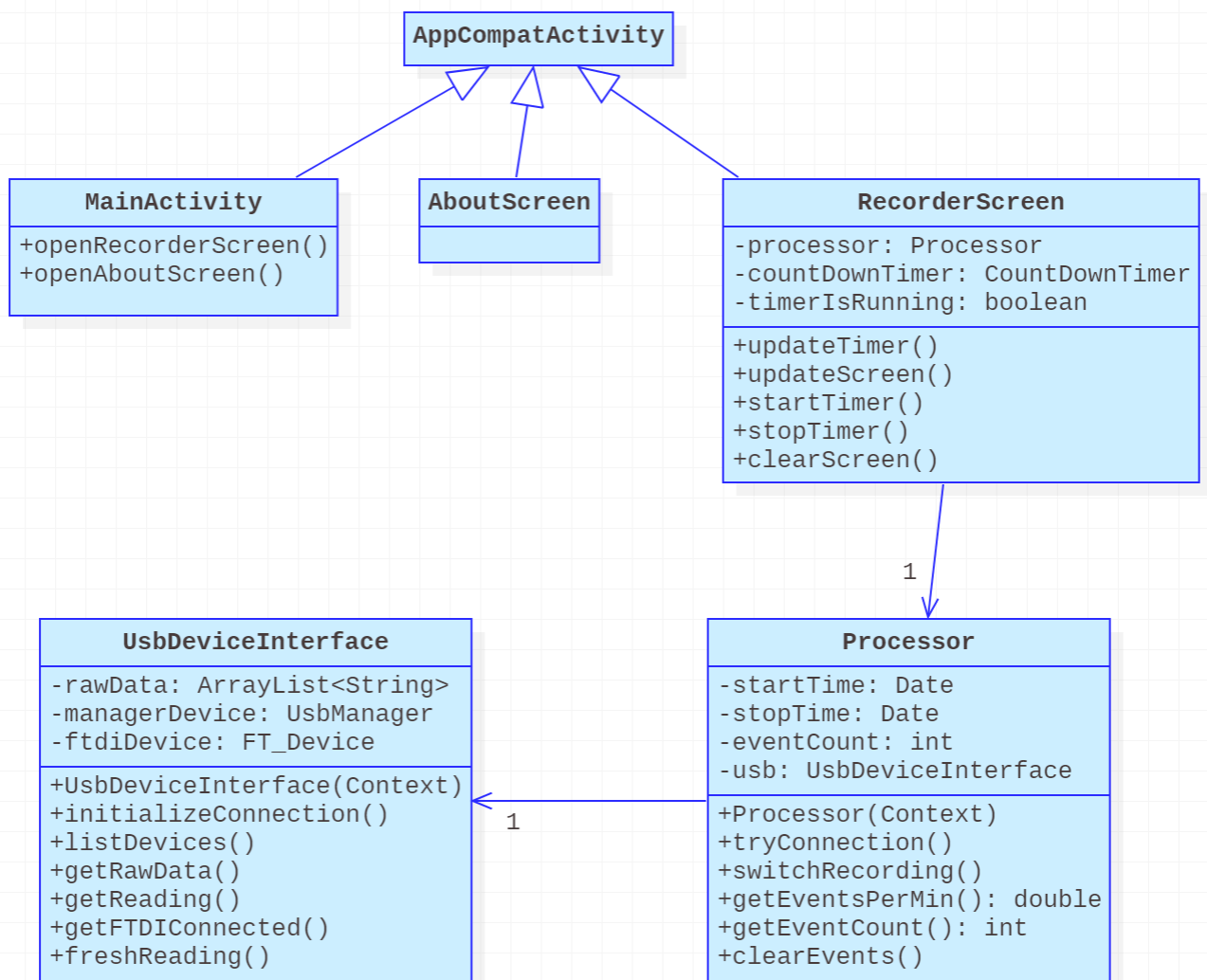
### 5-4: Two Extra Functional Requirements

Our demo video featuring all three functional requirements can be viewed at the following ([link](#)).

## Deliverable #6

### 6-1: Class Diagram

#### Cosmic Watch Communicator



## 6-2: Detailed Explanation

At the top of our diagram, there are three classes extending the class `AppCompatActivity`. This indicates that these are the main Activities of our app, representing the screens the user can access; `MainActivity`, `AboutScreen` and `RecorderScreen` correspond to the Main Menu, About, and Recorder screens respectively.

The Main Menu screen allows the user to access the About and Recorder screens. Thus, `MainActivity` has the methods `openRecorderScreen` and `openAboutScreen`. The About screen simply displays information, so it does not have relevant attributes and operations to show.

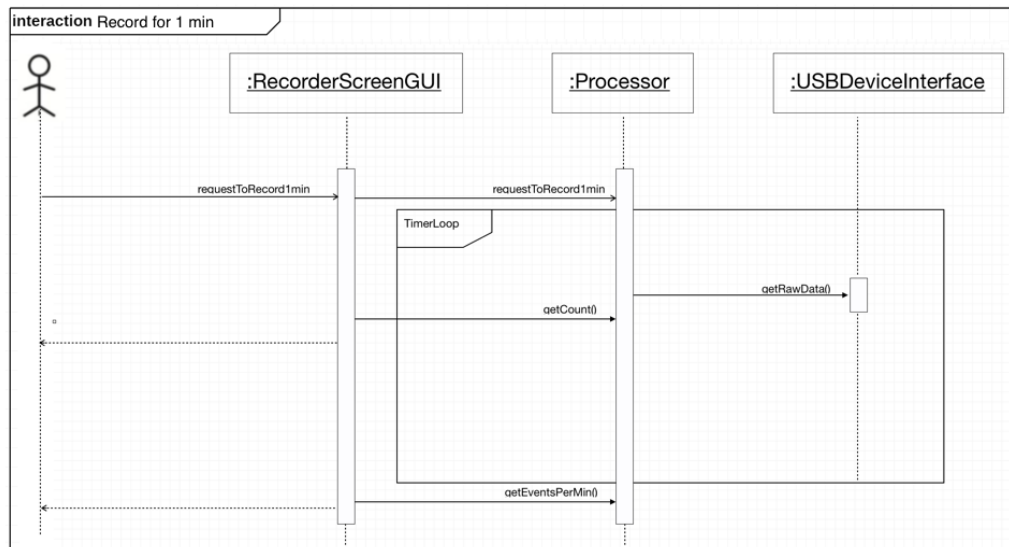
`RecorderScreen` maintains a reference to a `Processor` object, which communicates between the `UsbDeviceInterface` and the `RecorderScreen`.

The `RecorderScreen` starts a timer when the user chooses to start recording. This is conveyed to the `Processor`, which will save the time recording starts and initialize the `UsbDeviceInterface`. The `UsbDeviceInterface` will open a wired connection to the muon detector and store raw data from it. When this data needs to be interpreted and displayed on the `RecorderScreen`, the `Processor` will make requests to the `UsbDeviceInterface` and update `eventCount` according to the amount of raw data saved.

When the recording session timer ends or when the user wishes to clear their saved data, the `RecorderScreen` sends another signal to `Processor`. The `Processor` will then reset its saved data and the raw data in `UsbDeviceInterface` as needed.

## Deliverable #7

### 7-1: Sequence Diagram 1 with Explanation



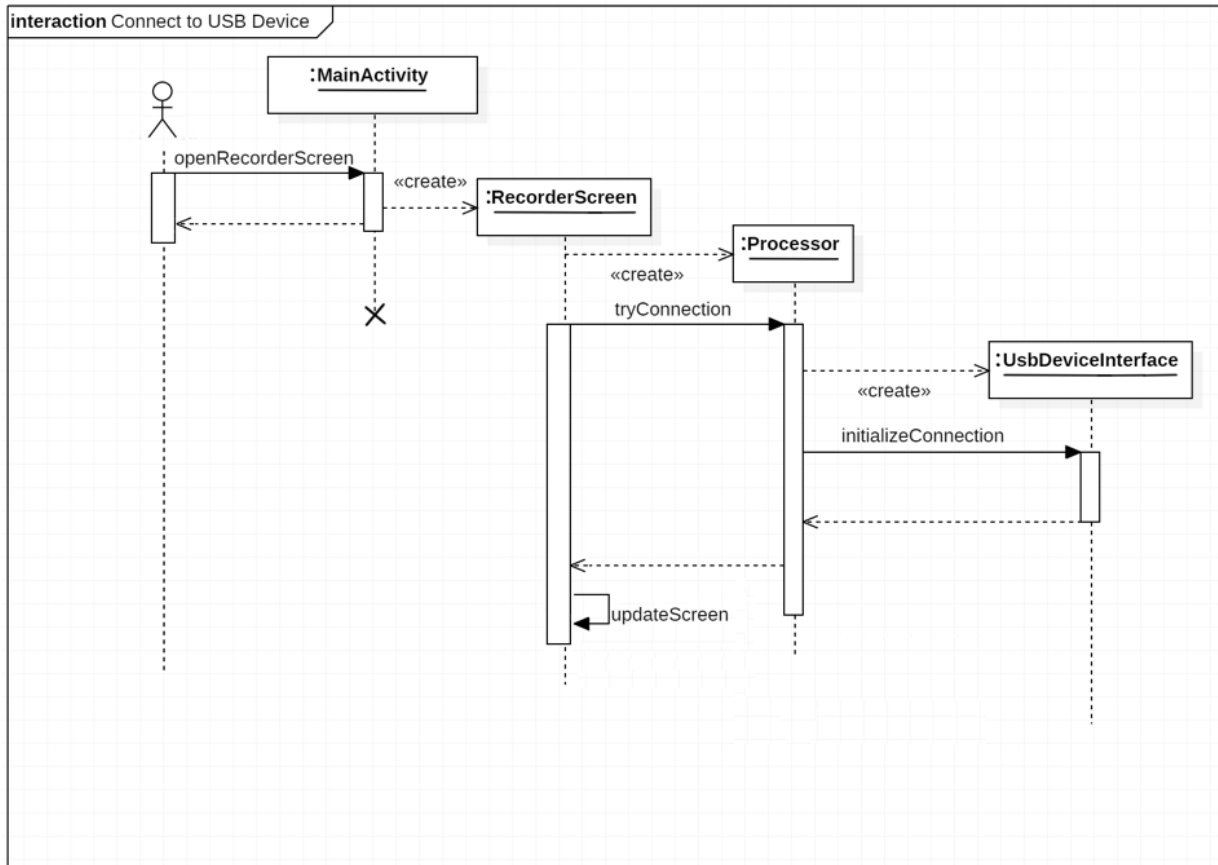
Our first sequence diagram demonstrates the sequence of tasks where a user initiates a recording for 1 minute. This is done by the user first being able to interact with a RecorderScreenGUI interface, specifically a button. The button communicates with the Processor class which creates a corresponding time stamp for the start of the recording session and enters into a Timer-Loop for 60 seconds. During those 60 seconds, the Processor communicates with the USBDeviceInterface, which is a class used to communicate with an external piece of hardware known as a muon detector.

As the muon detector picks up events, the Processor class calls getRawData() method from the USBDeviceInterface class increment events, which increases the private variable count in the Processor class. This happens every time a muon event occurs during those 60 seconds. The RecorderScreenGUI class then updates the count displayed by calling the getCount() method from Processor. At the end of the TimerLoop, the RecorderScreenGUI calls the getEventsPerMin() method of the Processor class which calculates the total events that occurred divided by the total duration the recording took place by using the date stamp time difference.

All of this information is then finally presented to the user in the RecorderScreenGUI. This entire sequence of tasks demonstrates a key concept in software engineering/computer science which is a layer of abstraction with all of the subsystems working behind the scenes while the user only interacts with a friendly UI.



## 7-2: Sequence Diagram 2 with Explanation



This sequence diagram shows the use case of the user successfully connecting to the muon detector through a USB port. When the user is on the Main Menu screen (MainActivity), they will choose to open the Recorder screen. This results in an instance of RecorderScreen being created, which also creates a Processor. MainActivity is no longer needed.

Upon creation, RecorderScreen will ask the Processor to try to connect to the detector using the UsbDeviceInterface. The Processor will initialize the UsbDeviceInterface, and the UsbDeviceInterface will form the connection with the muon detector. After this connection is formed successfully, initializeConnection and tryConnection will return to the RecorderScreen. Finally, the RecorderScreen will display a message using updateScreen to show that the device is connected.

## **Deliverable #8**

**8-1: Class Diagram with Design Pattern**

**8-2: Explanation**

## **Deliverable #9**

See PowerPoint presentation in the submitted .zip file.

## **Deliverable #10**

Collaboration with client blurb.