SENG 300: Assignment 2

Celina Ma, John Ngo, Omar Qureshi

June 15, 2018

Deliverable #1

1-1: 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.

1-2: 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;
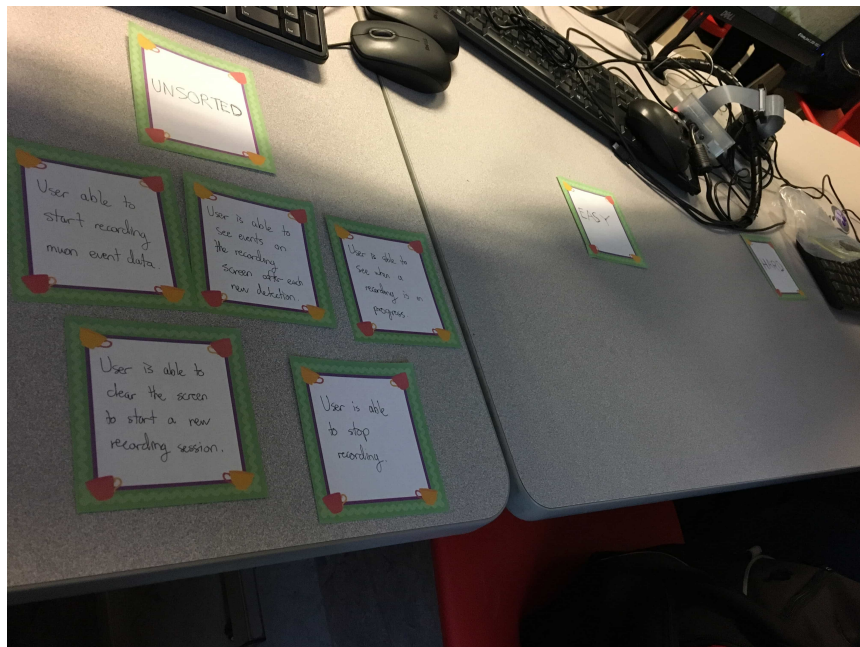

Figure 1: Post it notes started initially in the unsorted pile and a table was used for convenience

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;
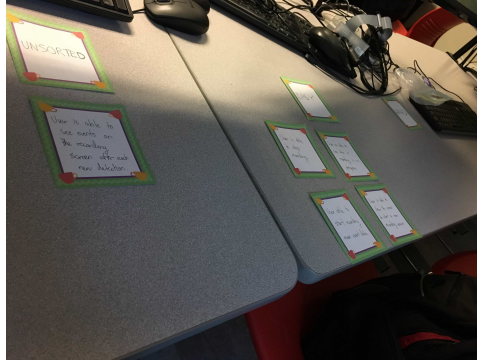

Figure 2: Silent grouping effort estimation in progress

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:
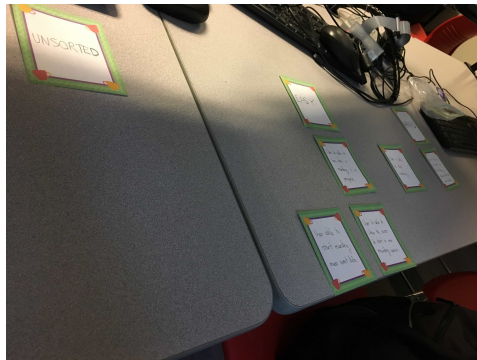

Figure 3: Completed session with one of the features being placed on hold

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.

Deliverable #2

2-1: 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.

4

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
TOTAL	-	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
Total	-	5	0.23
GRAND TOTALS		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.

2-2: Effort Estimation using Intermediate COCOMO Model

Project type: Semi-Detached (a = 3.0, b = 1.12)

Justification: Our project can best be described as a semi-detached project. While part of our project involves data processing and data retrieval, the other half is hardware based in that it depends on communication through USB with a scientific instrument. The overall project is ambitious for our team, since we have fairly little experience with much of the components of the project, incl. Android programming and USB interaction. The app is being developed by a small team, and ultimately aims to have real-time display of detection events as fed through by the USB link. As such, straddling the lines of organic and embedded as it is, it stands to reason that the project is semi-detached.

The following table represents our Cocomo factors along with their associated cost ratings, which will be used in the equation to calculate our person-hours:

Cocomo Factors	Rating
Analyst Capacity: Low (Beginning Students)	1.19
Applications Experience: Very Low (<=4 Months)	1.29
Application of SENG/Programming Practices: Nominal (Doing the assignment)	1.0
Complexity of the Project: High (Event based programming; weakest estimate)	1.15
Computational Operations: Nominal	1.0
Device Dependent Operations: High (or Nominal)	1.0
Data Management Operations: Nominal	1.0
Memory Constraints: Nominal	1.0
Programming Language Experience: Low (<4 Months)	1.07
Required Turnabout Time: Low	0.87
Required Software Reliability: Nominal or Low	1.0
Required Development Schedule/Schedule Constraint: High	1.04
Runtime Performance Contraints: Nominal	1.0
Database Size: Low (for now)	0.94
Software Engineer Capacity: High	0.86
Use of Software Tools: High/Very High (IDE)	0.91
Virtual Machine Experience: Low	1.10
Volatility of the Virtual Machine: Very Low (Do not expect changes anytime soon)	0.87
Total Multipler	1.17184642

Of the cost driver factors, this is the justifications for why some of the factors may be different from nominal:

1. Analyst Capacity: Our analyst capacity is low, because we are software engineering students in our first software engineering class, just beginning to apply what we've learned here. As such, we have little confidence in our own analysis capacities.

2. Applications Experience: We selected very low here because none of us have worked on a comparable project before. We haven't worked on android before, we haven't worked with USB input with Java before, and anything we did have experience on, such as coding and UI, was for less than 4 months since that's how long a university semester is.

3. We selected that the project has high complexity because of the requirement for event-based programming, some multithreading, USB hardware communications, and high level programming operators in the form of classes interacting with one another. In any case, all lower level descriptions do not fit and all higher level ones demand more than is actually involved here.

4. Our programming experience with Java is low since our only experience with Java was during CPSC 233 for a short period of time.

5. The required turnabout time for processing our code is very low, due to using modern computers.

6. Our development schedule is high due to this being an extremely rushed semester.

7. Our database size needed is low, mostly because we don't need one within the current scope of the project.

8. Our software engineer capacity is high, because we are all fairly effective and productive coders who can organize meeting often and plan well together

9. Our use of software tools is high, because almost all the features described are embedded as part of the Android Studio IDE.

10. Our experience with the Virtual Machine we are programming for is low; the android platform we have but do not know the deeper level coding behind it, and the detectors we are programming for are simple but also not interacted with very much before.

11. The volatility of the virtual machine is very low, primarily because Android is a stable operating system with backwards code comparability, and the scientific instrument we are coding for doesn't have code that will change within out current software development timeframe.

With our multiplier now calculated along with constants a and b given, we can use the KDLOC total from our previous table in *Deliverable 2.1* to do a person-month calculation:

$$a * KDLOC^b * multiplier = 3 * (0.59)^{1.12} = 1.95$$

Once we have our above person-months figure, we can convert it to a person-hour figure via a simple calculaton, starting with some assumptions. We first assumed that we would work about 18 days over the course of the month on this project, with 3 hours spent each day. This would then give us 54 hours of work per person. We then multiply this number with the person-month figure above to give us:

$$1.95 \ personmonth * 54 hours/person = 103.5 \ personhours$$

The above number is then used as a topic of discussion below, comparing it with our previously calculated value via summing total in the tables.

2-3: Comparison Between Two Approaches

After completing our Intermediate Cocomo Model effort estimation calculation, we noticed a stark difference in person hours compared to the previously calculated totals in the tables of *Deliverable 1-1*. To recall, the value calculated via the table totals was 28 person hours vs. 103.5 person hours via Cocomo.

Some thoughts as to why the Cocomo number may be larger than actual could be because we used inflated estimates for our lines of codes. The Cocomo estimation is very sensitive to the KDLOC number due to the exponential and as such this contributes to the large number, whereas in actuality our lines of code may be smaller.

On the other hand, some thoughts supporting the Cocomo estimate is that there are a lot of cost drivers taken into account when coming up with the multiplier number. Factors such as applications experience, programming language expertise, schedule constraints and complexity of project are all taken into consideration whereas via the table approach those variables were all ignored. Usually it's always safer to prepare for the worst case so having this figure calculated will serve as a reminder that quite a bit of effort, time and determination will be required to provide the client with a product that meets their requirements.

Overall, our group members are glad we are provided with a tool such as the Cocomo effort estimation equation to come at an estimation for the time possibly required for the project. This again helps us better plan our weekly schedules and allows us to be more on the safer side for time management.

Deliverable #3

3-1: 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:

```java
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());
    }

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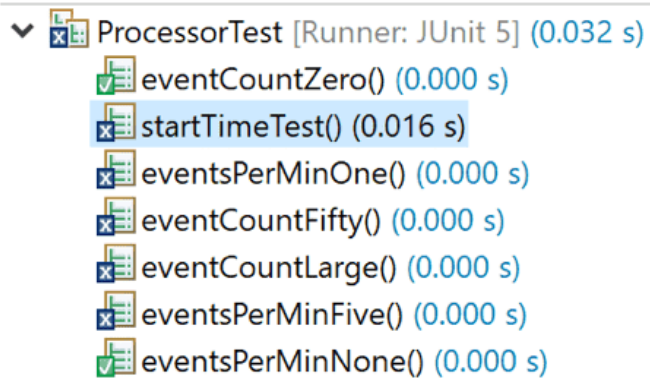
To compile the test code, a basic skeleton of the Processor class was created:

```java
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)

3-2: 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:

```java
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:

```java
@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:

```java
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;
    }
```

14

Now there is enough code to pass all the test cases:
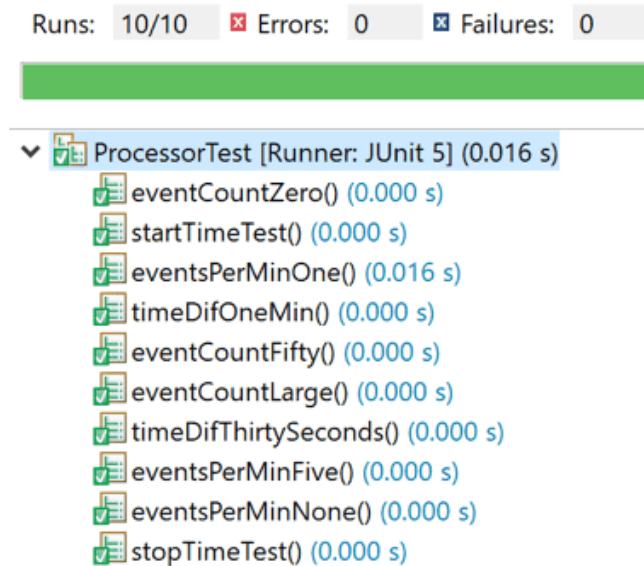
3-3: Tests with Full Implementation

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

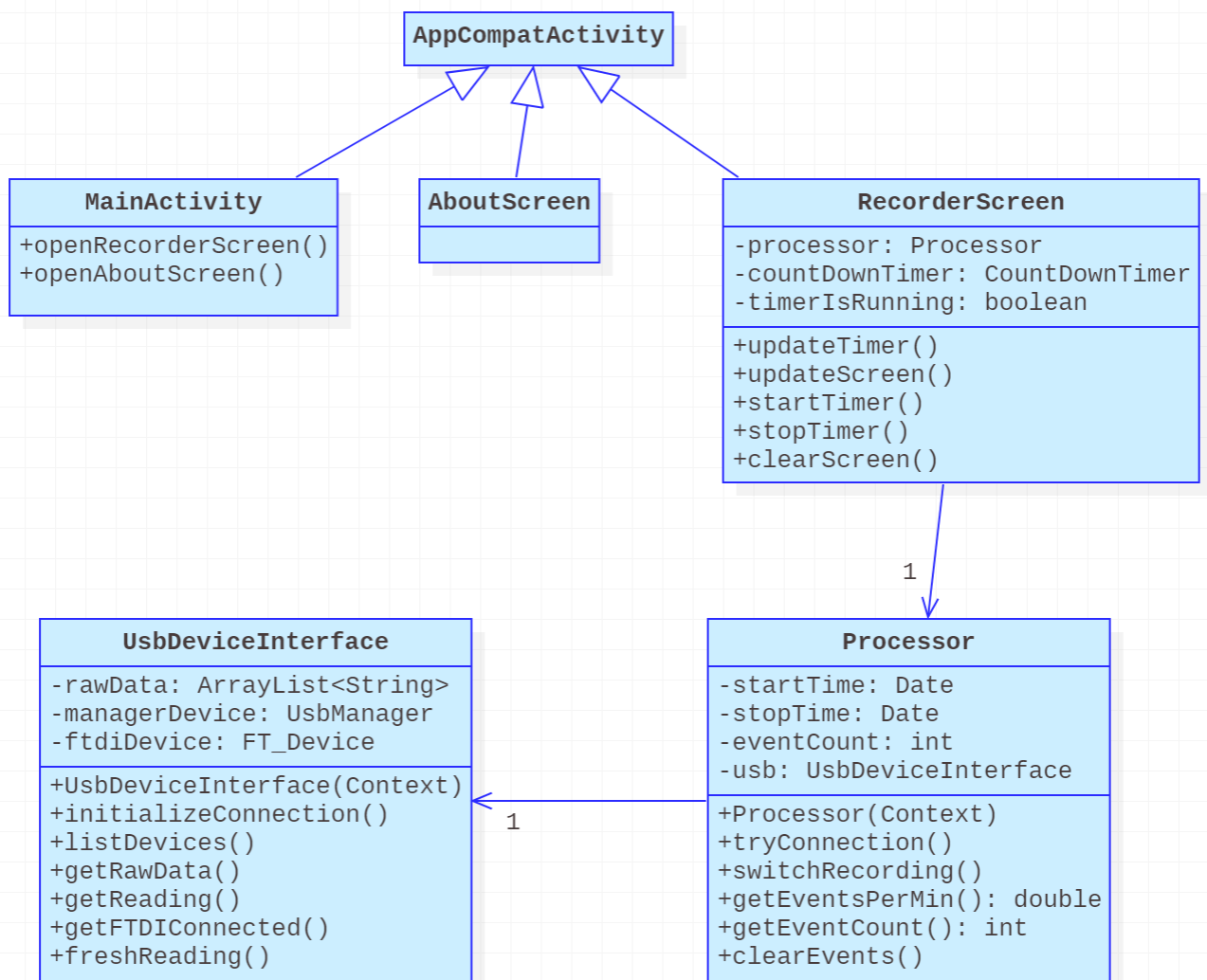### 3-4: Two Extra Functional Requirements

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

Deliverable #4

4-1: Class Diagram

`Cosmic Watch Communicator`

4-2: 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 #5

5-1: Sequence Diagram 1 with Explanation
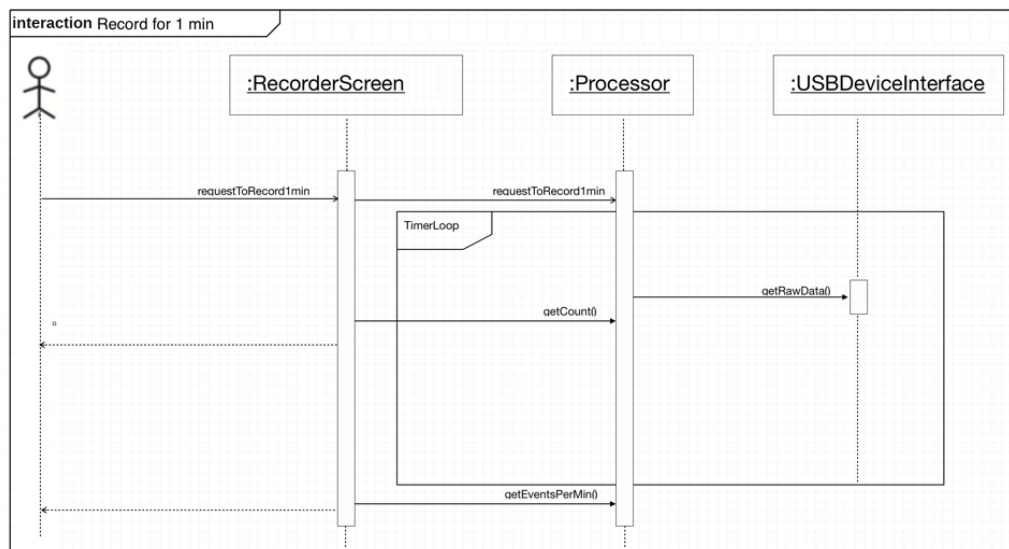
Our first sequence diagram demonstrates the sequence of tasks where a user initiates a recording for 1 minute. This is done by the user first being able to interact with a RecorderScreen interface, specifically a button. The button communicates with the Processor class which creates a corresponding time stamp for the start of the recording session and enters into a TimerLoop 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 getRaw-Data() method from the USBDeviceInterface class which increases the private variable count in the Processor class. This happens every time a muon event occurs during those 60 seconds. The RecorderScreen class then updates the count displayed by calling the getCount() method from Processor. At the end of the TimerLoop, the RecorderScreen 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 Recorder-Screen. 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.

5-2: Sequence Diagram 2 with Explanation
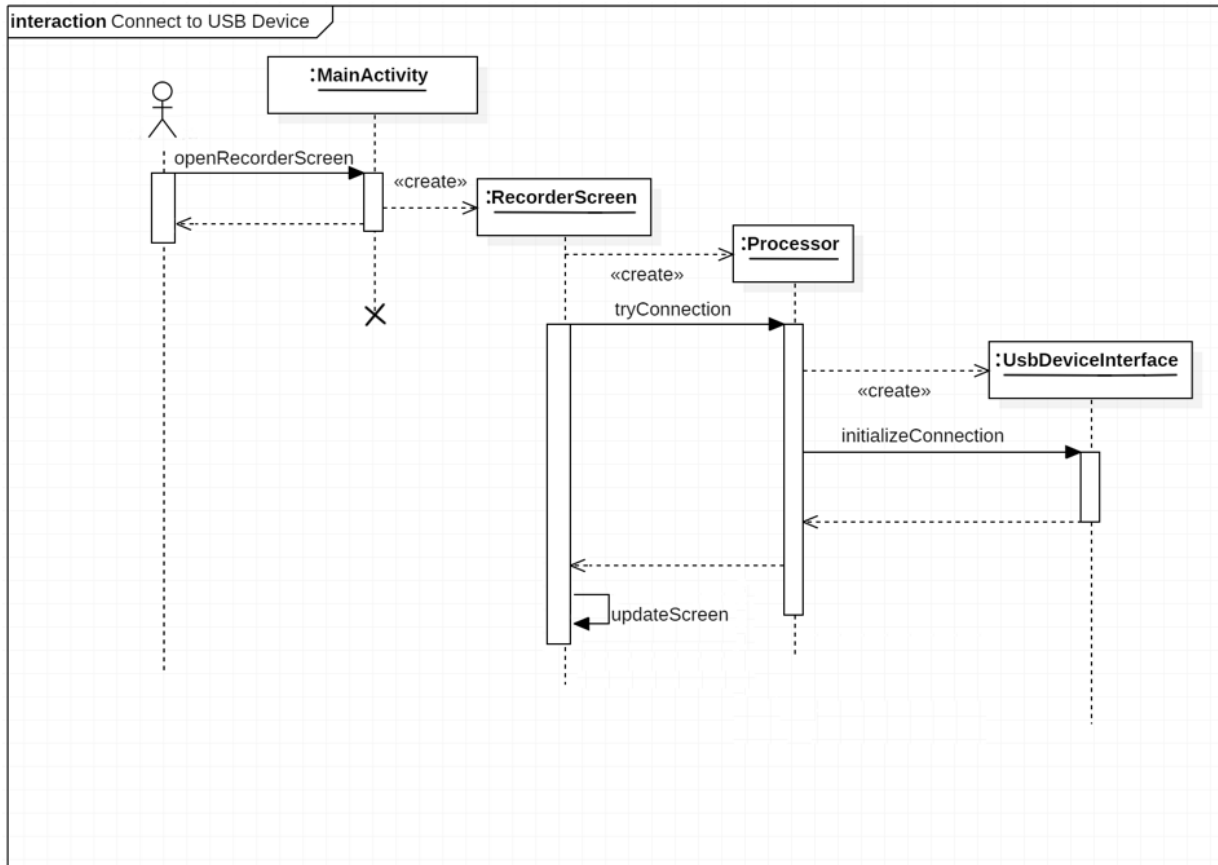
This sequence diagram shows the use case of the user successfully connecting to the muon detector through a USB port. When the user is on the Main Menu screen (MainActivity), they will choose to open the Recorder screen. This results in an instance of RecorderScreen being created, which also creates 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 #6

6-1: Client Meeting

We initially agreed to meet with Dr. Donev shortly after the completion of Assignment 1. However, we soon realized that we would not have much of our product complete by then, so we decided to hold the meeting later;

After we made more progress with implementation, we arranged a new time;

For our meeting, our app had two versions demonstrating different functional requirements that had yet to be integrated. We first showed that the app was able to connect to the muon detector via a mini-USB to mini-USB wired connection. We then showed the second version of the app, which could display fake data in a useful format to simulate what real data read from the muon detector would look like. We explained that we would combine these components to obtain one complete, functional product.

Dr. Donev was pleased by our progress and provided helpful feedback on our work. We allowed him to directly interact with the app's simple interface on an Android tablet. He commented that a setting for the app that prevents the device from going to sleep while recording data would be useful; this was a feature we had not considered until testing the app on a real device. During this session, we also discovered some bugs with the interface. At the end of our meeting, we agreed to talk again on Monday to hopefully present a more polished product.

Our greatest challenge in communicating with our customer was finding convenient times for all four of us to meet in-person. Although we spoke through email, some ideas could not be easily transferred through text alone. Due to our classes and Dr. Donev's many obligations, we were not able to meet frequently. However, we made the most out of each meeting we had by focusing on obtaining as much feedback as possible from our client.