Software Design Specification

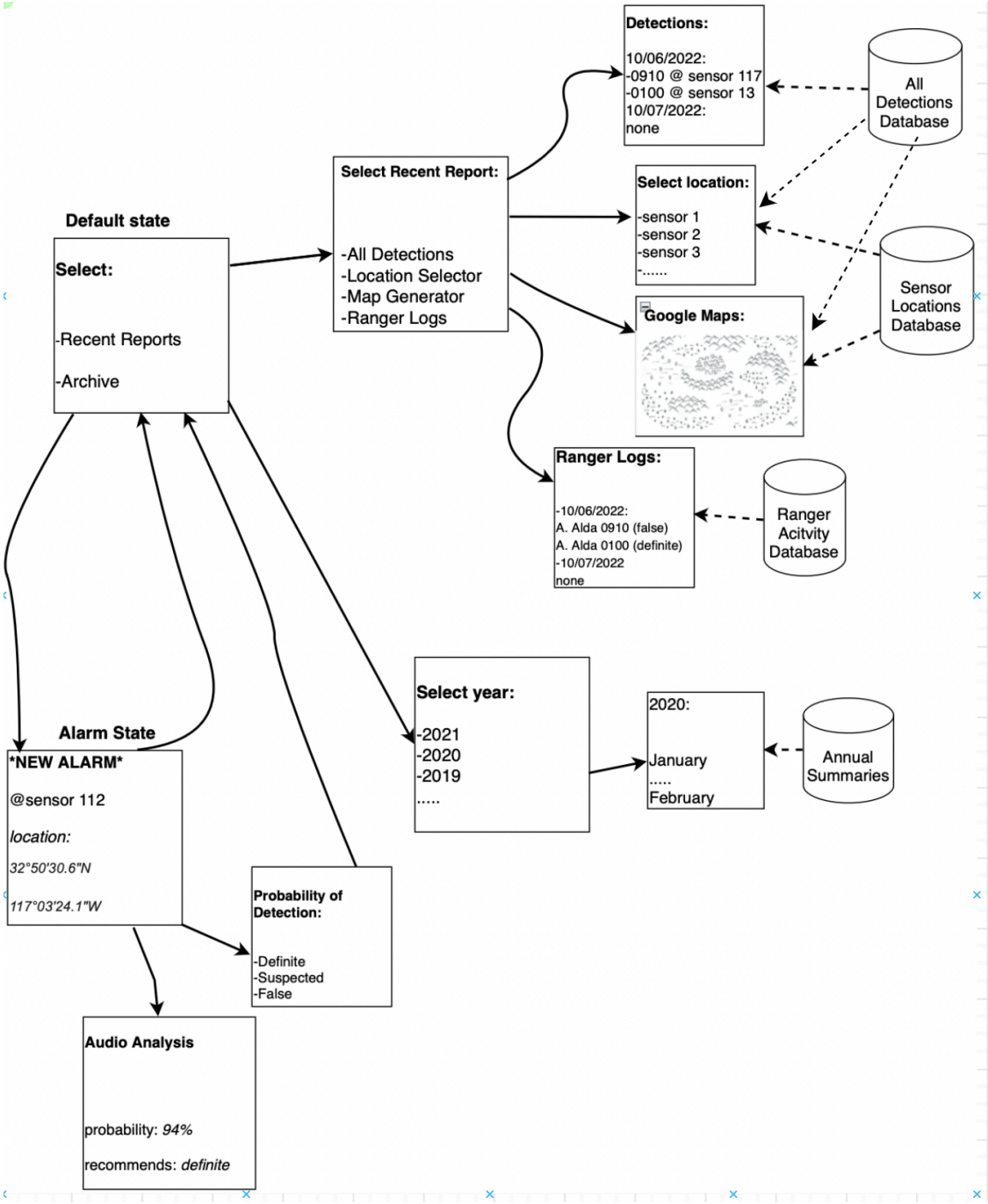
Mountain Lion Detection System

Ali K.
Chris F.
Zhiwei H.

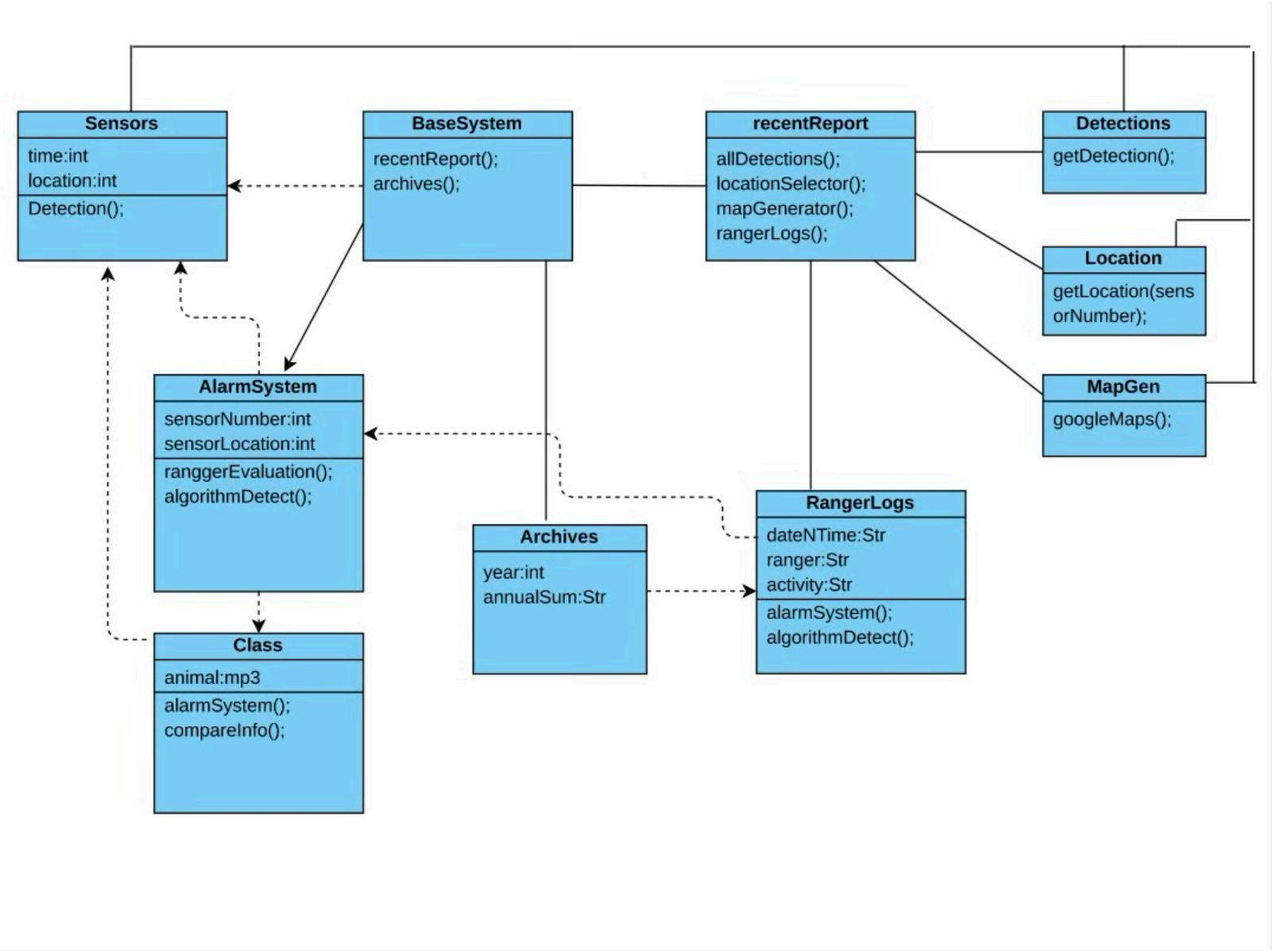
System Description

This system is designed to provide rangers with quick and reliable alerts about mountain lion activity within their parks. At the ranger station, users can receive alerts sent from one of hundreds sensors distributed throughout the park. The rangers' sole responsibility is to assign a probability classification to the detections as they are received by the system. The system also runs an algorithm automatically that generates a recommendation for the ranger based on the newly received audio file attached to the alert. Once the ranger has classified their new detection, the system reverts back to its default state which allows the user to navigate through all the past detections. Users can also filter the data by location or view the detections on a map of the park. Older reports from previous years can be accessed through the archives to generate annual summaries. For accountability, Ranger logs are kept to track all classifications performed by each ranger by date.

Architectural Diagram



UML Class Diagram



The following UML diagram is created to visually illustrate the implementation of the Mountain Lion Detection System. Starting off with the class BaseSystem, this serves to contain and direct to the main components and functions. The BaseSystem class contains two operations branching off to two different classes. The first operation is recentReports which directs to the RecentReports class. The second operation is archives which directs to the Archives class. Beginning with the RecentReports class, this class serves to store all the reports, and contains four optional operations. The first option is the allDetections operation which directs to the Detections class. Inside the Detections class there is a getter method called getDetection() which retrieves its

information from the operation called Detection present inside the Sensors class. The second operation is the LocationSelector which directs to the Location class. The Location class contains the getter method called getLocation with a parameter for the park Ranger to manually input a specific sensor number. The third operation is MapGenerator which directs to the MapGen class. The MapGen class contains the method googleMaps which is an API call allowing the user to check detection by the map. The fourth operation is rangerLogs which directs to the RangerLogs class. The RangerLogs class contains three attributes that are all string variables. These three attributes are called dateNTime, ranger, and activity. There is also an operation called alarmSystem inside the RangerLogs class which directs to the AlarmSystem class. Moving onto the Archives class, this class serves to provide yearly data and contains two attributes. The first attribute, year, is represented by an integer variable. The second attribute, annualSum, is represented by a string variable. Moving onto the Archives class which is the second branch of BaseSystem, the class Archives contains two variables. Year, which is an integer variable, and annualSum, which is a String variable. The variable annualSum provides a general overview specified by month of the detections that occurred throughout the year.

The AlarmSystem class contains two variables called sensorNum and sensorLocation which are both of type integer. This class also contains two operations. The first operation is rangerEvaluation. The dotted arrow from the class RangerLogs that is pointed at the AlarmSystem class indicates the dependency that RangerLogs has on AlarmSystem. The second operation is called algData(). The algData operation retrieves its info from the AlgorithmData class. The AlgorithmData class contains one attribute called animal, and two operations called alarmSystem and CompareInfo. The AlgorithmData class retrieves its information from the Sensors class and then using its animal mp3 attribute and CompareInfo operation it compares audio files to determine the severity of the threat. Which is why there is a dotted arrow from the AlgorithmData class going to the sensors class to indicate this dependency. If determined to be a serious threat then the method alarmSystem is triggered, which is why there is a dotted arrow from the AlarmSystem class going to the AlgorithmData class to indicate this dependency.

Development Plan & Timeline

This project will be developed by a team of seven highly experienced python programmers and data scientists. The estimated time required to deliver an alpha version of this system would be 4 months and at least six months for the final product.

A small group of three will be assigned the sole task of developing the algorithm needed to provide rangers with recommendations regarding the classification of new threat detections. A database of mp3 files containing animal noises will be purchased from a third-party company based in Chicago. This will be used to allow the algorithm to compare newly-received audio files transmitted from the sensors to those stock

animal noises. Afterwards, an evaluation is presented to the user (ranger) regarding the probability of the threat. This is a highly elaborate and specialized component of the system which should take at least four months to develop.

The remainder of the team will be working either individually or in close pairs to develop some of the simpler, less complex functions. One individual will be responsible for collecting sensor data and compiling them into the three databases: *All detections, sensor locations, and Annual summaries.* One pair will work closely together in designing an easily navigable user interface for the rangers. Their responsibilities extend to developing the main functions as well since they are fairly straightforward and rely on the work of their teammates to do most of the heavy-lifting. This should primarily involve calling on algorithms and retrieving from databases. The final member will be responsible for constant testing and code review.

Test Plan

Function Test for Captured Audio Analysis Algorithm

Comparison tests: the inputs will consist of various audio files that will be compared to the database of stock audio files of animal noises.

Input 1: Mountain lion roar audio that is nearly identical to stock audio (found in Animal Noises database) that is used by the algorithm to compare with new audio detections (input). This is a test in ideal conditions that should output a probability of *definite.*

Input 2: Mountain lion audio that has been distorted by other naturally occurring sounds (birds, insects, etc.) to further test the function's ability to recognize and distinguish the lion's sound out of a messy mix of forest noises. These harsher conditions are intended to output a probability of *suspected.*

Input 3: Audio will be inputted containing various forest wildlife recordings but with no mountain present. This test should should provide an output probability of *false.*

Once all these tests have been completed and confirmed to have the correct output, we will know that the algorithm is capable of providing a reliable recommendation on the probability of mountain lion detections being definite, suspected, or false.

Integration test for Map Generator

The map generator uses detection and sensor location data from the Detections and Sensor Locations databases respectively to provide the user with a map containing red pins, each representing a single detection from the past 24 hour period. To test this, we will dispatch various rangers at random sensor locations around the park where they will trigger them. Detection data will be sent from the sensors to the ranger station where it is recorded and used to generate a map. The Rangers were also asked to record their geographical coordinates on their mission which would be used to validate the Map Generator's accuracy. Each pin's position on the map will be compared to each ranger's recorded location. Since the audio sensors are accurate to approximately 5 meters, any detection that exceeds that margin of error will be considered a failure. Also, if any two locations should match (within 5 meters), then those would confirm the map's accuracy.

Unit Test for Archive Years

```
x = inputYear = 2021
y = maxYear = 2023
If (inputYear ≥ maxYear) {
        Year = fail;
}
Else {
          Year = pass;
}
```
(passed)

```
x = inputYear = 2023
#this is the year that will be inputted when the ranger creates the annual summary
y = maxYear = 2023
#this is the maxYear set to the current year + 1
If (inputYear ≥ maxYear) {
        Year = fail;
}
#the if statement checks if the inputYear is greater than or equal to the maxYear, if it is true, the test fails.
Else {
          Year = pass;
}
#the else statement is if the if statement is false, then the test passes.
```
(failed)

The inputYear variable is the year that will be inputted when the ranger requests the annual summary. MaxYear is the year set to the current year + 1
The if statement checks if the inputYear is greater than or equal to the maxYear, if it is

The else statement is if the if statement is false, then the test passes. This test makes sure that an incompatible input (e.g. ‘2026’) does not return anything and reminds the user (ranger) that there is a maximum year for input.

System Test for Ranger Logs Access

We will be testing if we can access the rangerLogs and if the rangerLogs are working properly and providing us with the correct information needed. From the baseSystem we will call recentReport(); which should allow for the call of 4 new functions, allDetections();, locationSelector();, mapGenerator();, and rangerLogs();. If using recentReports does not allow for those 4 new functions to be called the test fails if it does, it passes the first test. We will select rangerLogs from the 4 functions and it should give us a list of all activities logged by rangers with the following information, dateNTime as a string, ranger as a string, and the activity recorded as a string. If any of those are not provided in a string format the test fails. We will check the information from the rangerLogs to see if it matches a hardcopy of the data (Rangers will be asked to initially record their logs on paper as well). If any of the information shown from the system does not match the hardcopy the ranger has, it fails the test.

Unit Test for Sensor Location

```
Sensor s1 = new Sensor(); //creating 4 sensors to test
Sensor s2 = new Sensor();
Sensor s3 = new Sensor();
Sensor s4 = new Sensor();
s1.setLocation(97°80’30.6”N); //giving each sensor a location
s2.setLocation(14°50’11.2”S);
s3.setLocation(97°80’30.6”N);
s4.setLocation(15°90’18.4”W);
Sensors sensors[] = {s1, s2, s3, s4}; //populating array with created sensors

for(int i = 0; i < sensors.length; i++){ //starting at element zero
        for(int j = 1; j < sensors.length; j++){ //starting at element 1
                if(sensors[i].getLocation == sensors[j].getLocation && i != j){
                        FAIL //fail because there is a duplicate location in sensors
                }
                Else{
                        PASS //there are no duplicate locations
                }
        }
}
```

Here we created a scenario of 4 sensors and gave them each a specific location and then populated them into an array. For each element in the array we retrieve the location with the method getLocation. We use two for loops to iterate through the entire array to check if there are any duplicates of sensor locations. If there are duplicate locations after iterating through then we know the system test failed. If there are no duplicate locations then the system test passes. Here we expect the system test will fail because there is a duplicate of two coordinates (s1, s3).

System Test for Alarm State

The alarm system will be tested to determine if the system is justifiably activated. AlarmSystem will take two inputs, one given from sensors, and the other from a stored database of animal mp3 sounds. The given threshold for the sensor's trigger is 5 decibels. Below that, the alarm system will not be triggered. Rangers will be sent out and attempt to exceed that standard. If the system exceeds 5 decibels, the alarm system should be triggered. This minimum threshold was established based on the ambient background noise of the forest. Rangers will have to exceed this base line to test whether or not the alarm is going off for a justifiable reason. Rangers will also attempt to trigger the alarm at the same location more than one time. The alarm should not go off at the same location. Rangers will also be equipped with handheld decibel-o-meters to record their noise level each time they trigger a sensor. As long as the recorded successful triggers remain above 5 dB each time and and below 5 dB for failed triggers, then the alarm system is being justifiably activated.

Data Management Strategy

The company has opted to implement a data management strategy that uses SQL databases to store the information needed to run the application as well as store data generated by it. This info will be distributed among four databases: *All Detections, Sensor Locations, Ranger Activity, and Annual Summaries.*

Sensor Locations Database

Sensor #	Location
117	32°44'02.4"N 117°05'13.8"W
13	32°43'55.2"N 117°05'18.8"W
58	32°44'36.7"N 117°05'22.3"W

All Detections Database

Detection #	Date	Time	Sensor #	Location
000623	10/06/2022	09:00	117	32°44'07.2"N 117°05'27.5"W
000624	10/06/2022	10:50	13	32°43'58.1"N 117°05'26.2"W
000625	10/07/2022	15:30	58	32°44'22.4"N 117°05'13.5"W

Every detection is recorded and stored in the *All Detections* database in chronological order as soon as it is received by the sensor. Each detection is assigned an ID number and has four fields associated with it: date, time, ID number of the triggered sensor, and the approximate location of where the sound originated. The second database is fairly small and is only used to store the locations of all the Mountain Lion sensors distributed throughout the park. The third database stores all the detection classifications each ranger completed during their shift. At the end of each shift, the base computer records the ranger's name, ID number, when their shift started/ended and the classifications they performed. These classifications are stored as single strings with each letter representing a single classification of a specific probability. For example, "SSF" represents three detections: *2 suspected* and *1 false.* The final database, Annual Summaries, logs a summary of all the detections at the end of every month with a total number of definite, suspected, or false detections. It also provides a rate of error calculation for the detection system.

Ranger Activity Database				
Ranger Name	Ranger ID	Shift Start	Shift End	Classification
Williams, R.	1234567	7:01 AM	5:00 PM	FSSSD
Roberts, G.	7653241	9:00 AM	5:07 PM	SSF
Smith, K.	4327901	11:00 AM	8:00 PM	SSFS
Duval, R.	1930308	2:00 PM	8:30 PM	DFF

Annual Summaries Database					
Year	Month	definite detections	Suspected detections	False detections	Rate of Error
2019	September	80	12	30	37.50%
2019	October	120	16	50	41.66%
2019	November	60	15	20	33.33%
2019	December	90	19	30	33.33%
————————————————————————————————————					
2020	January	160	16	30	18.75%

Tradeoff discussion

We found it most practical for the development team to use table-based databases with pre-defined schema since we do not anticipate there will be a need to add/remove certain field criteria. We understand this option is costly to scale, but that has been remedied by the development budget allocated for this project which is quite substantial. And based on the technical requirements needed for this data management strategy, we expect a fairly gradual vertical expansion plan. There will be intermittent hardware upgrades to sustain this scaling. We also had to acknowledge the user's (Park Ranger) perspective and they should appreciate SQL's ability to retrieve records quickly and easily. Alternatively, our team could have opted for a NoSQL approach. This would have provided us the added benefit of protection against changing schema criteria. However, as mentioned before, it is quite unlikely that this will happen. We can also forgo NoSQL's main advantage, being adept with big data, since our storage requirements are relatively lightweight. Unlike SQL, it also lacks standardization which was an issue raised by the client out of fear that some system failures might arise during migration.