San Diego County Parks and Recreation Department Mountain Lion Detection Software Design Specification

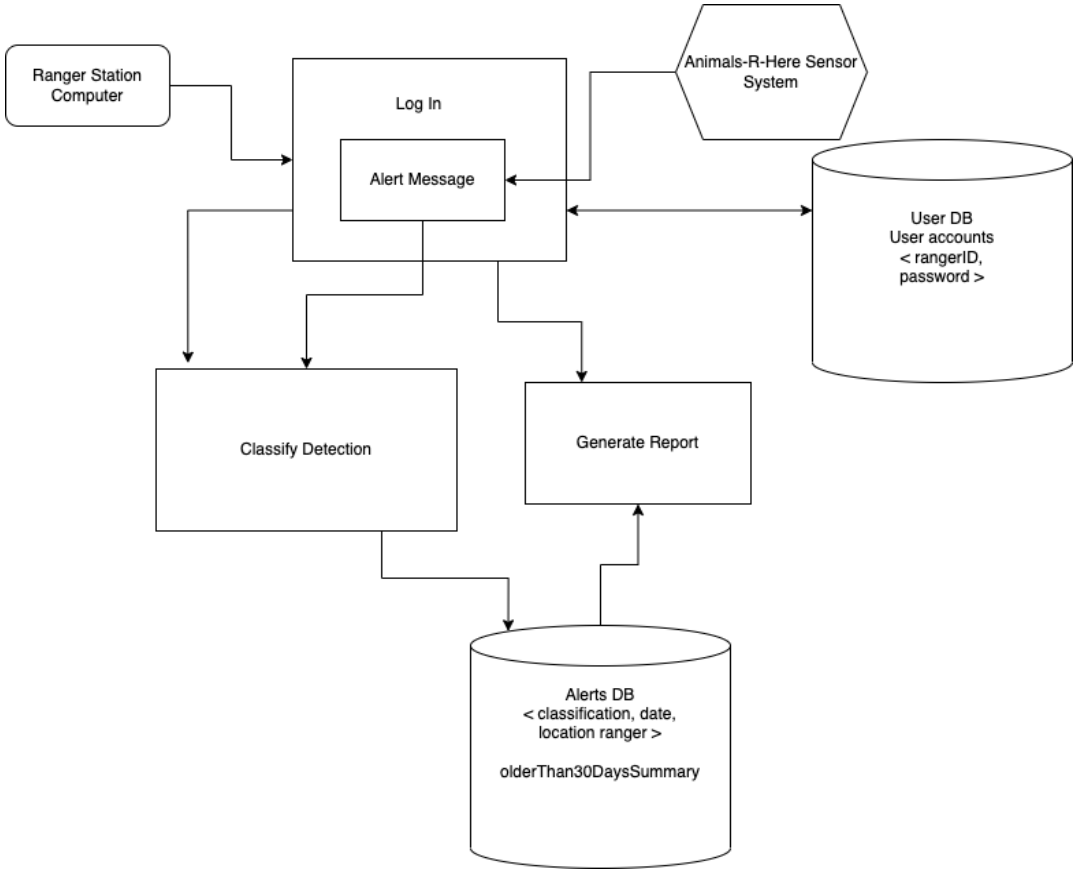
Prepared by: Gabriela Calvo, Shane Ahmad, Daniel Ha

System Description

Our system is designed for park rangers employed by the San Diego Parks and Recreation Department to use when handling data on mountain lion activity within a given park area. Our control program will facilitate receiving, classifying, and storing alerts sent from Animals-R-Here sensors that are placed throughout the parks and pre-programmed to detect mountain lions. Upon logging in, users will have the option to classify a detection or generate a report. Alerts received from the sensors will contain data on type, strength, and location of the detection but all alerts must also be reviewed by a ranger and classified as either definite suspected or false based on the probability that an actual mountain lion was detected. Within generating reports, the user has four options. The four types of reports are of detections organized by classification and date, detections organized by ranger, detections at a specific location, and detections indicated on a map that extends 2 miles beyond park boundaries.

Software Architecture Overview

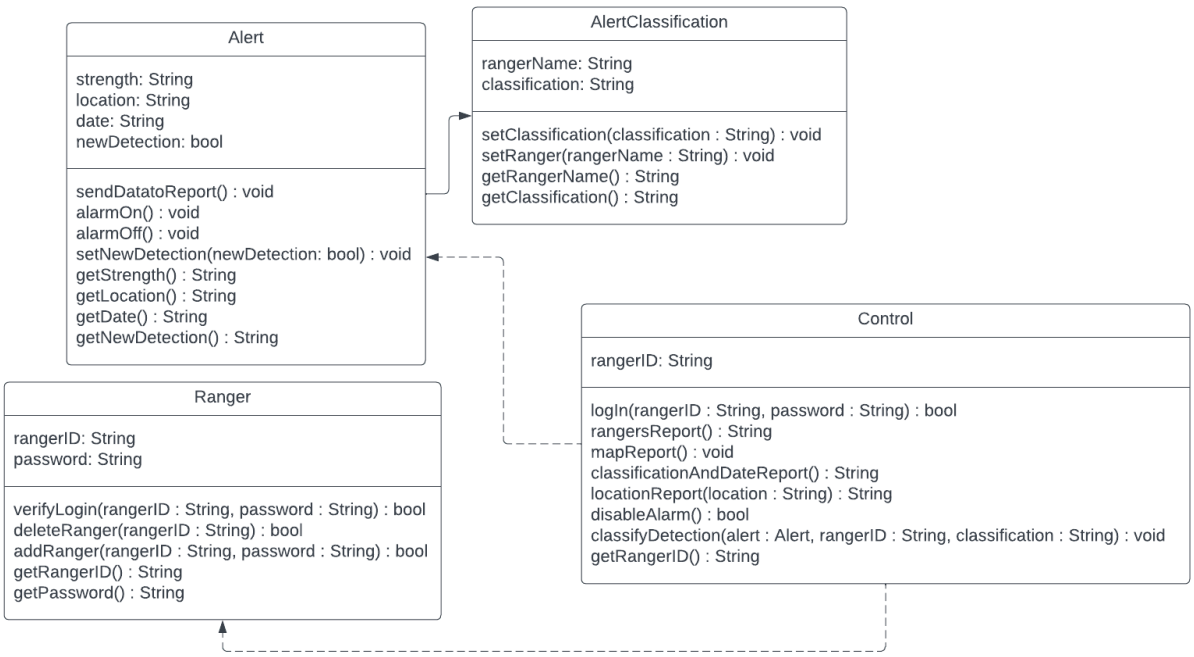
Software Architecture Diagram



The design of our system is centered around the two main types of actions that the user performs; classifying alerts and requesting reports. For our purposes, the user will always be a ranger on a ranger station computer which is represented by the box in the top left. That box connects to our login page which has a sub-category representing the case where the user is responding to an alert message. The alert message depends on data that is sent from the Animals-R-Here Sensor system which is represented in the top right corner. Whether or not they are responding to an alert, the user will have to log in with their rangerID and password and that

is where the login page will communicate with the User Database to verify their credentials, indicated by a double ended arrow. From there, the user will either classify a detection or generate a report both of which will interact with the Alerts Database that stores the data for previous detections. Classifying a detection sends data to the Alerts Database, which is depicted by a one directional arrow towards the database. Generating a report retrieves data from the Alert Database, depicted by a one directional arrow from the database.

UML Class Diagram



Description of Classes

Alert

The Alert class is designed to retrieve information from the third-party sensor that detects animal noises. This class includes the functionality for sending data to generate reports as well as controlling the alarm that goes off when a detection is received from the sensors.

AlertClassification

The AlertClassification class is an extension of the Alert class which adds variables for classification and the name of the ranger that reviewed the detection. This completes the information necessary for a detection to be included in a report, so each Alert should have an associated AlertClassification.

Ranger

The Ranger class represents all of the individual ranger profiles stored in the User Database depicted in the Software Architecture Diagram. This class will mostly be used to verify the rangers when they log in.

Control

The Control class includes the functions the rangers can perform on their computer. This class facilitates the user storing information related to the detections and retrieving reports.

Description of attributes

The Alert class has four attributes. The first attribute, animalNoise, is a String that defines the kind of animal that made the sound detected. The strength String attribute measures the intensity of the sound. The location String attribute is defined by the location of the sensor when it detects a sound. The date String attribute is determined by the date when the sound is detected. Finally, the newDetection attribute is a boolean variable that is set to true whenever a new detection is received from the Animals-R-Here system. The AlertClassification class builds onto the Alert class with two additional attributes, one for the classification of the detection set by the ranger and one for the ranger that reviewed the detection, both of which are String type. The Ranger and Control classes both use a String attribute for rangerID that is unique to each ranger. The Ranger class contains an additional String attribute for password, which will be used alongside rangerID when the user logs in.

Description of operations

The Alert class has seven functions. The first is sendDataToReport() which returns the information of a given detection to be included in a report. The class also has two operations associated with the alarm system, alarmOn() and alarmOff(). The alarmOn() function is triggered whenever the newDetection bool variable is true. The alarm will turn on and stay on until a ranger manually turns it off using the alarmOff() operation. The last 4 functions, getStrength, getLocation, getDate, and getNewDetection are all getter methods that return the current state of that attribute. The AlertClassification class has four methods. The first is setClassification() which allows the ranger currently at the computer to review the detection and classify it as

“definite”, “suspected”, or “false” according to how likely it was to be an actual mountain lion detection. The setRanger() function will associate the current ranger with the detection so it can be shown in the reports. The final two functions are getRangerName() and getClassification() are getter methods for the rangerName and classification attributes. The Control class has eight functions. logIn(String rangerID, String password) is used in the Control class to allow a ranger access to the database so they can request reports. classifyDetection(Alert alert) is used to select an alert to classify. There are four functions within the Control class that generate reports. The classificationAndDateReport() function retrieves a report that includes all mountain lion detections organized by date and classification. The locationReport(String location) takes one String argument indicating location and returns to the user a report including all mountain lion detections at a specific location. The requestMap() function provides a map of the park and the surrounding area indicating all stored detections. The rangerReport(string ranger) function takes one String argument indicating a specific ranger and returns to the user all detections reviewed by that ranger. Elsewhere in the class, disableAlarm() is used to stop the alarm from making sound. Finally getRangerID() returns the current state of the rangerID attribute which for our purposes will be the rangerID for the ranger that is currently logged in. The Ranger class includes three functions managing a ranger’s ability to use the system. deleteRanger(String rangerId) is used to stop allowing a particular ranger to access the database. The second function, addRanger(String rangerID, String password) is used to create a new Ranger in the system. Finally, verifyLogIn(String rangerID, String password) checks to ensure that a ranger’s ID and password match. All three of these functions return a boolean value indicating success (true) or failure (false). The final two functions for the Ranger class are getters for the rangerID() and password() attributes that will be used during the login process.

Unit Testing

setClassification

The setClassifition function can be fully tested using four cases. The only valid inputs for classification of a detection are “definite”, “suspected”, or “false” so we need to test that classification gets correctly updated for those inputs and does not get updated for any other inputs.

Preconditions:

AlertClassification alert1;

We assume that upon initialization classification will be set to “Not yet classified.”

Test : Definite detection

alert1.setClassification(“definite”);

Check that the classification attribute updates correctly using getClassification() == “definite”.

Test 2: Suspected detection

alert1.setClassification(“suspected”);

Check that the classification attribute updates correctly using getClassification() == “suspected”.

Test 3: False detection

setClassification(“false”);

Check that the classification attribute updates correctly using getClassification() == “false”.

Test 4: Bad input

setClassification(“maybe”);

Check that getClassification == ”Not yet classified” which is the default value for the classification attribute. We don’t want to allow the user to set the classification to anything other than “definite”, “suspected”, or “false”.

verifyLogin

The verifyLogin() function is intended to check that the username and password entered by the user match those which are stored for a Ranger in the database. If they do, the function should return true to indicate that the login credentials have been verified. If they do not match what is stored in the database the function should return false to indicate a failed login attempt. In a typical use case, this function will be called within the logIn() function of the Control class.

Preconditions:

Ranger ranger1;

ranger1.setRangerID(“12345”);

ranger1.setPassword(“mypassword”);

In order to test this function we first make an example Ranger object and initialize its rangerID and password attributes to example Strings in order to have something to compare the user input to.

Test 1: Correct inputs for both rangerID and password

```
if(ranger1.logIn(“12345”, “mypassword”)  == true){

        //test successful, logIn returns true if the arguments entered match the
        Strings stored in rangerID and password for ranger1

}else{

        //test not successful, logIn returns false even if rangerID and password are
        entered correctly

}
```

In this test case we enter the correct inputs for both rangerID and password. Since both inputs match what rangerID and password were initialized to, we expect the function to return true. If the function does not return true that means it is not functioning correctly.

Test 2: Incorrect inputs for both rangerID and password

```
if(ranger1.logIn(“0000”, “wrong”)  == false){

        //test successful, logIn returns false if the arguments entered do not match
        the Strings stored in rangerID and password for ranger1

}else{
```

```
        //test not successful, logIn returns true even if rangerID and password
        entered do not match the Strings for rangerID and password for ranger1
    }
```

In this test case we enter incorrect inputs for both rangerID and password. Since both inputs do not match what rangerID and password were initialized to, we expect the function to return false. If the function does not return true that means it is not functioning correctly.

Test 3: Correct input for rangerID, incorrect input for password

```
    if(ranger1.logIn("12345", "wrong")  == false){
        //test successful, logIn returns false if the arguments entered do not match
        the Strings stored in rangerID and password for ranger1
    }else{
        //test not successful, logIn returns true even if rangerID and password
        entered do not match the Strings for rangerID and password for ranger1
    }
```

In this test case we enter the correct input for rangerID but an incorrect input for both rangerID and password. Since one input does not match, we expect the function to return false. If the function does not return true that means it is not functioning correctly.

Test 4: Incorrect input for rangerID, correct input for password

```
    if(ranger1.logIn("00000", "myPassword")  == false){
```

```
        //test successful, logIn returns false if the arguments entered do not match
        the Strings stored in rangerID and password for ranger1
    }else{
        //test not successful, logIn returns true even if rangerID and password
        entered do not match the Strings for rangerID and password for ranger1
    }
```

In this test case we enter an incorrect input for rangerID and the correct input for both rangerID and password. Since one input does not match, we expect the function to return false. If the function does not return true that means it is not functioning correctly.

Integration Testing

setRanger

Focuses on testing the setRanger method to assign a ranger to a classification. It is important to make sure the method works well with integrated units such as databases and other classes (in this case, Control to set and get data from reports), first to check if it is a valid ranger, and second to store the ranger's name into the database with the intent that it will be retrieved later.

Test:

```
AlertClassification classifiedAlert1;
classifiedAlert1.setRanger("Jimmy");
List<Rangers> Park1Rangers; //assume list is from database
```

```
//test
bool setRangerTest(AlertClassification, alert1){
        if (Park1Rangers.contains("Jimmy")){
                //then we expect alert1.getRanger() to return "Jimmy"
                if(alert1.getRangerName() == "Jimmy"){
                        //setRanger successfully updated the ranger, test passed
                        return true;
                }
                else{
                        //setRanger did not update correctly, test failed
                        return false;
                }
        else {
                //setRanger should not update because the ranger name entered is not
                contained in the database of current rangers
                //we expect alert1.getRangerName() to return "" (an empty String is the
                default value)
                 return false;
        }
}
```

Good Case:

setRangerTest(classifiedAlert1, "Jimmy"), returns true

A true reading means the Ranger is within the database and can be assigned to a classification.

Likewise, a false reading means that the ranger is not within the database and can not be assigned to a classification.

classifyDetection

Focuses on testing the classifyDetection method to classify a detection based off the type of alert. classifyDetection should be able to work well well between integrated units such as both the Control and Alert Classes. When using the classifyDetection method, three parameters must be called (an Alert, a rangerID, and a classification) and it is important to also check that whatever is inputted is valid.

Preconditions:

```
Ranger ranger2;
ranger2.setRangerID("54231");
Alert Alert1;
Control User1;
User1.classifyDetection(Alert1, ranger2.getID(), classification);
```

Test 1:

```
if(classification.equals("definite")){
        if(Alert1.getClassification().equals("definite")){
                // true if classifyDetection has updated the Alert Class' detection
                successfully to match the new classification
        }
        else{
                // false if classifyDetection has not updated the Alert Class'
                detection successfully to match the new classification
        }
}
else if (classification.equals("suspected")){
        if(Alert1.getClassification().equals("suspected")){
```

```
                    // true if classifyDetection has modified the Alert Class' detection
                    successfully to match the new classification
            }
            else{
                    // false if classifyDetection has not modified the Alert Class'
                    detection successfully to match the new classification
            }
    }
    else if (classification.equals("false")){
            if(Alert1.getClassification().equals("false")){
                    // true if classifyDetection has modified the Alert Class' detection
                    successfully to match the new classification
            }
            else{
                    // false if classifyDetection has not modified the Alert Class'
                    detection successfully to match the new classification
            }
    }
    else {
            //The classification entered by the user/ranger is not one of the three
            classifications and should not be considered valid
    }
```

In this test, we are making sure that whatever the user/ranger decides to classify an alert as, it will automatically update the Alert Class' member variables. In this case, the classification the user inputs should be one of the three classifications ("definite", "suspected", or "false") and if not, the input should not be considered correct.

Test 2:

```
    if(classification.equals("definite")){
            if(classificationAndDateReport.contains("definite")){
```

```
                // true if classifyDetection has updated the Control Class' report
                successfully to match the new classification

        }

        else{

                // false if classifyDetection has not updated the Control Class'
                report successfully to match the new classification

        }

}

else if (classification.equals("suspected")){

        if(classificationAndDateReport.contains("suspected")){

                // true if classifyDetection has updated the Control Class' report
                successfully to match the new classification

        }

        else{

                // false if classifyDetection has not updated the Control Class'
                report successfully to match the new classification

        }

}

else if (classification.equals("false")){

        if(classificationAndDateReport.contains("false")){

                // true if classifyDetection has updated the Control Class' report
                successfully to match the new classification

        }

        else{

                // false if classifyDetection has not updated the Control Class'
                report successfully to match the new classification

        }

}

else {

        //The classification entered by the user/ranger is not one of the three
        classifications and should not be considered valid
```

}

Likewise, in this test, we are making sure that whatever the user/ranger decides to classify an alert as, it will also automatically update the Control Class' member variables. In this case, the classification the user inputs should be one of the three classifications ("definite", "suspected", or "false") and if not, the input should not be considered correct.

Note We can assume that the other parameters are correct. We can assume that the ranger involved in calling this method is valid since we called the tested setRanger method in the preconditions. Same assumption goes for the Alert Object.

System Testing

Getting a Location Report

This test is an overview of the functions required for a ranger to go to the computer and request a report about all alerts at a location. The user should be able to log in and then request a location report from the database. This involves using functions from all four classes.

Control LocationTest = new Control();

Alert newAlert = Alert()

rangerID = 1234;

String password = "pass_word";

String classify = false;

location = "Location 1";

```
newDetection = true;

date = “September 3, 2023”;

strength = “strong”

AlertClassification.setClassification(classify);

LocationTest.classifyDetection(newAlert);

// send the example data to the database

if (ControlTest.logIn(rangerID, password)) {

        // indicates attempt to log in, then call verifyLogIn

        if (Ranger.verifyLogIn(rangerID, password)) {

        //fails if pass_word is not the correct password for ID 1234

                if(locationReport(location).contains(“Strong, false alert on September 3, 2023”)){

                        return true;

                }

          }

}

return false;
```

This test returns true if the system was successfully able to log in the user and then allow the user to request a report listing all detections at a location, in this example “Location 1”. It also

indicates that there were no errors in performing these functions. Reasons the test would fail includes reporting the information from a wrong location, not being able to retrieve any information from the database, or not allowing the user to log in.

Responding to Alert

This test is an overview of the functions required for a ranger to respond to an alert at the computer. The user should be able to log into the computer, disable the alarm, and then classify the report so the report can be sent to the database. To do this, functions from all four classes must be used.

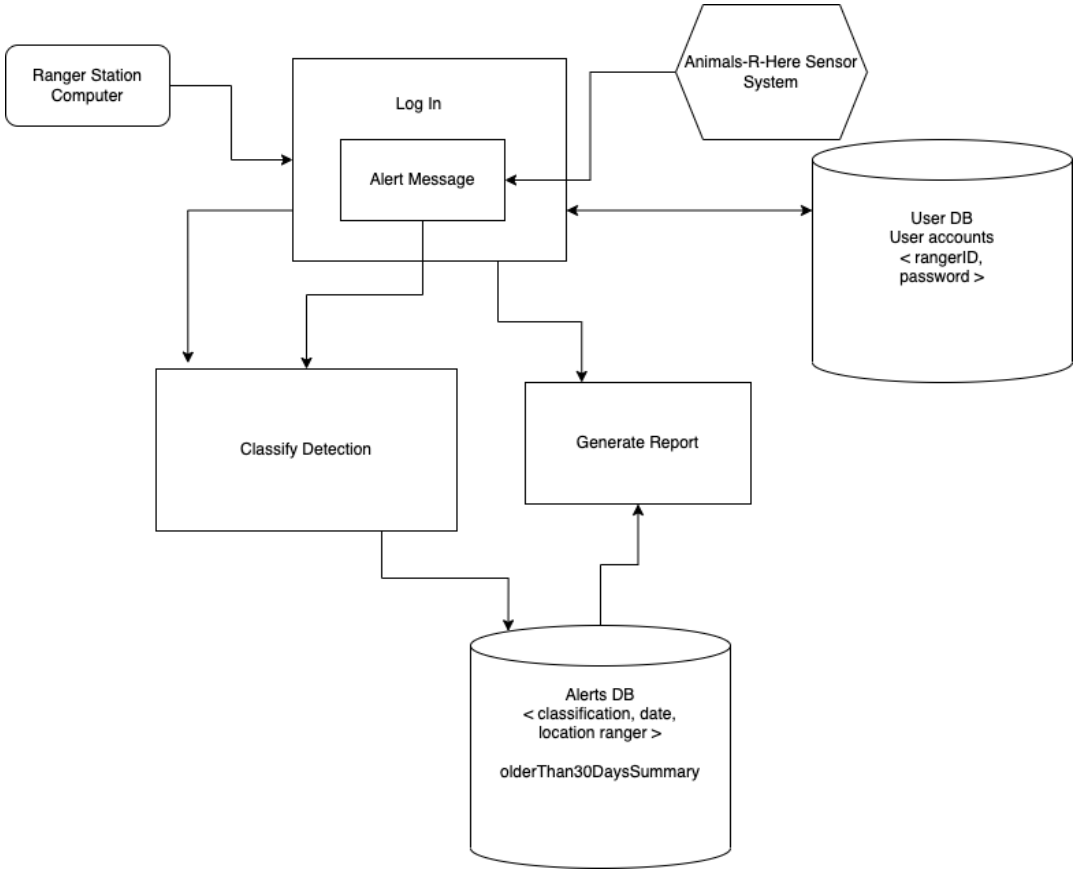
```
Control AlertTest = new Control();

rangerID = 1234;

String password = “pass_word”;

String classify = “false”;

if (AlertTest.logIn(rangerID, password)) {

        // indicates attempt to log in, then call verifyLogIn

        if (Ranger.verifyLogIn(rangerID, password)) {

        //fails if pass_word is not the correct password for ID 1234

                if (AlertTest.disableAlarm()) {

                        //checks if alarm is currently on, and then disables it and returns true
```

```
                    //if alarm was already off, return false
                    AlertClassification.setClassification(classify);
                    //defines the classification
                    AlertTest.classifyDetection(Alert);
                    //sends the report to the database
                    return true;
                    // indicates a successful test
                }
            }
}
return false;
// indicates a failure in testing
```

If this test returns true, then the test was successful. It indicates the user is able to log in, disable the alarm, and classify the report. It also indicates that there were no errors in sending the report to the database. There are a few possible reasons a test would fail. If the rangerID does not correspond with the entered password, the test will fail. If that rangerID is not an added ranger, the test will fail. If the user is trying to classify an alert but there is no alert to classify, the test will fail.

Data Management Strategy

Software Architecture Diagram



As a reminder, this software architecture diagram is the basis for our data management strategy. Refer to the section titled “Software Architecture Overview” for a more detailed description.

SQL Management Visualizations

Table View

ALERTS DB

detection_id	classification	ranger_id
00001	“false”	“ranger1”
00002	“suspected”	“ranger2”
00003	“false”	“ranger1”
00004	“definite”	“ranger3”
00005	“definite”	“ranger2”

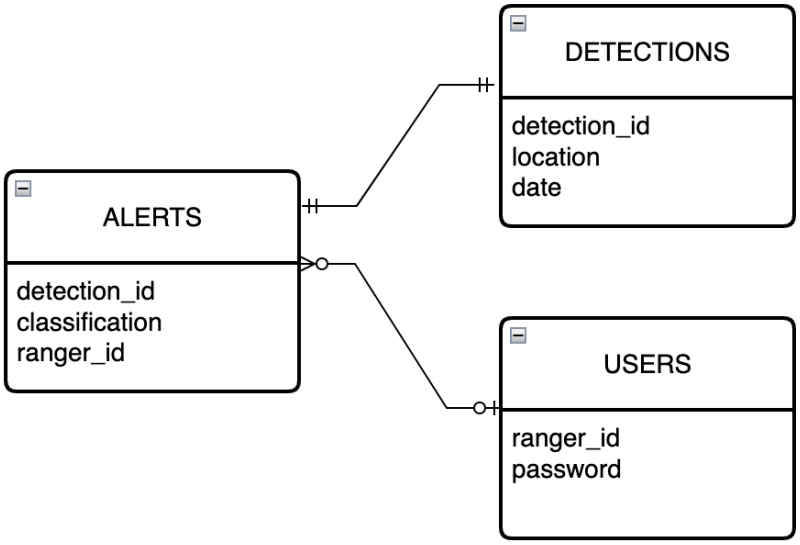
USER DB

ranger_id	password
“ranger1”	“mypassword”
“ranger2”	“myOtherPassword”
“ranger3”	“securePassword”

DETECTIONS DB [update software diagram]

detection_id	location	date	strength
00001	“Site 1”	“01/01/2023”	“strong”
00002	“Site 2”	“01/02/2023”	“weak”
00003	“Site 2”	“01/03/2023”	“weak”

Diagram View



Description

We chose to have 3 databases, titled Detections, Alerts, and Users. We feel that having these 3 databases allows for the minimum amount of storage use while still representing the entirety of our system. The Detection database is where the information gathered from the sensors is stored. No user input is involved in this data. The Alerts database stores information gathered from the user at the computer when they review and classify a detection. Finally, the Users database stores all of the login information for the rangers. Each of these databases is closely associated with the three main functions of our system; receiving data from sensors to classify a detection, retrieving detections to include in a report, and retrieving login credentials to verify a username and password.

Although we have decided to use a SQL database, it is certainly possible to use a noSQL strategy. Using NoSQL could help us handle a larger amount of structured data if we ever decide to add more fields or functionality that might increase the volume of data. Even for our current design that has relatively few fields, implementing a noSQL strategy early on could be helpful with scalability and availability and it pairs well with object-oriented programming. Although noSQL is a valid alternative, there are a number of tradeoffs to consider. Building off of ACID vs. CAP, NoSQL is not as capable of performing dynamic operations lacking ACID factors. Since our program does involve complex querying such as reporting, SQL is preferred. However, our system does not need to prioritize run-time complexity so NoSQL could also be used.