Mountain Lion Detection System

Hisham Alasadi, Dylan Bernier, Hanh (Hannah) Nguyen

A. User Manual:

I. Functionality Requirements:

The system must ask users to enter their credentials to login and verify the authentication. Then users (rangers) will be able to view, create, edit, and classify reports into pdf files from alerts generated by the system. Rangers will be able to see detailed daily alerts within 30 days. The detailed daily alerts include the information about time, location and status description to write the daily report. Rangers can sort the alerts by time or location. Also, they can review previous yearly reports.

When the system detects the noise of a lion, it will display an alarm on the screen and rangers will have the option to save the alert (in case of false alarm detected). When there are multiple alerts, the system will place them to a queue in order by time. Through the system, rangers can also see detections on a graphical map within 2 miles of the park for better identification.

II. System Requirements:

Three databases will be used for user information, alerts and reports. The database of user information must be encrypted for security and purposes. The authentication must use a symmetric authentication key for integrity and confidentiality. The system must have the capability to hold a database of detailed daily alerts within 30 days and alerts of previous years. Finally, the report database must be able to classify reports into groups of time or location.

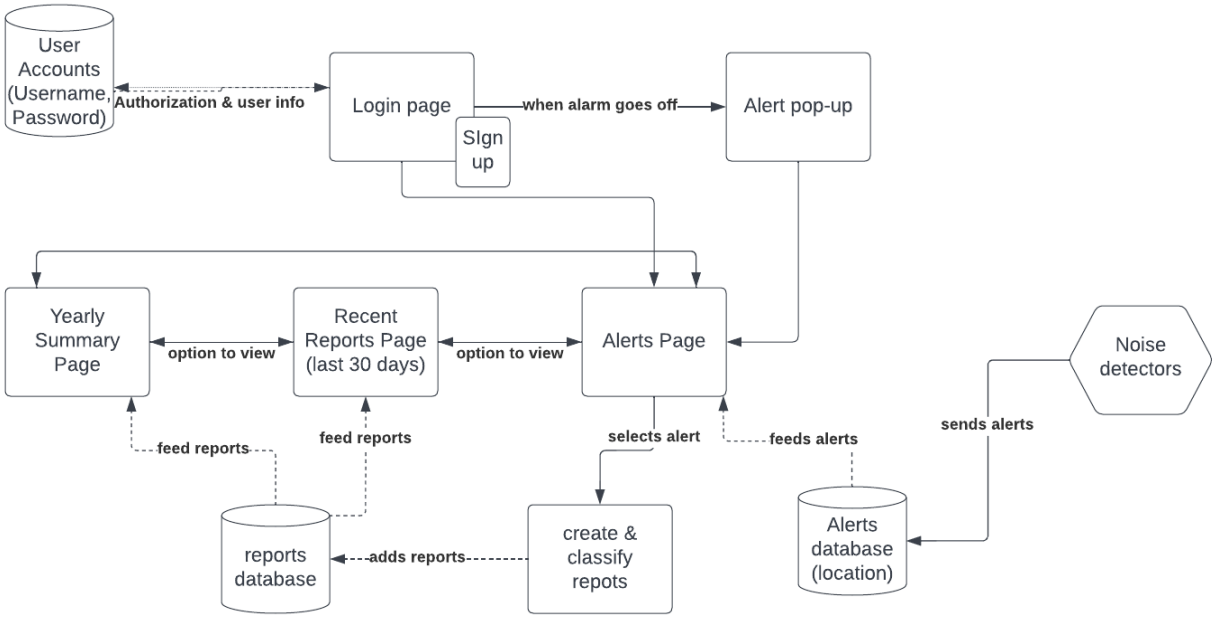
B. Design Documents:

I. System Description:

The Mountain Lion Detection System is a detection system for San Diego county parks needed to detect dangerous mountain lions in order to better equip park rangers with knowledge of mountain lion habits. The goal of this system is to detect mountain lions in county parks and take safety precautions using the noise detection software. Another goal is to not disrupt the wildlife but rather to monitor the wildlife. The system will be integrated with a scripted program to detect levels of audio frequencies to classify sounds of animals. Each noise detection sensor will be placed within an area of 5 square miles. And they have the capability to automatically send alerts to the controlling computer at the park ranger station. The main users of this system would be park rangers who can review and edit the system database through a controlling computer at the park. The controlling computer houses a database of reports sent by noise detectors placed throughout the park.

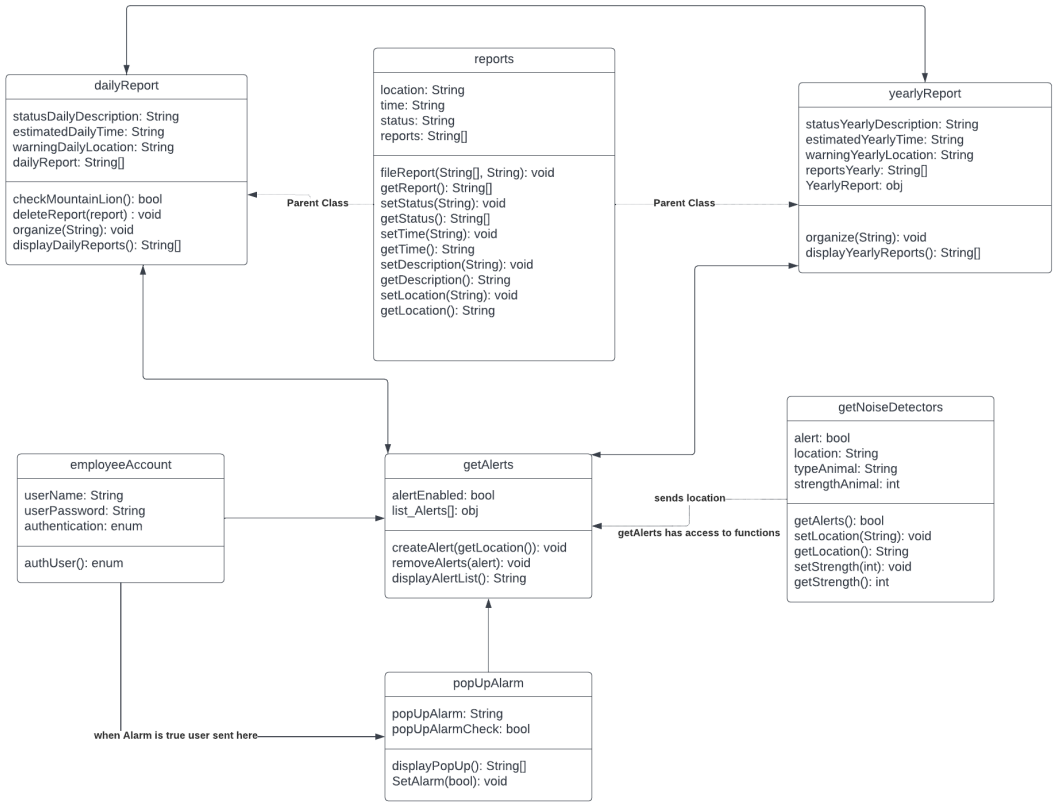
II. Software Architecture Overview:

The following are the architectural and the UML class diagrams for all major components. The architectural diagram details the pages that the application will include. There are three databases: the first one will store users' information such as login details and authorization codes, the second one will store all the reports (daily and yearly), the third one will store the alerts information which include locations and time. Users (rangers) can choose to start on the login page that, when authorized, will enable them to classify, turn off each alert, edit and view several daily and yearly reports.



III. UML Class Diagram:

The class diagram shows the different classes that will be utilized to develop the application. The employeeAccount will allow users to sign in into the application and use the getAlerts and popUpAlarm classes to access relevant information (location, time) about the detection. The getNoiseDetectors will receive the information based on types of animal noise, the strength of the detected animal noise, and the location of the detected noise to within 3 meters. The getAlert class will provide information which includes the location and time received from the getNoiseDetectors class to send alerts.When the rangers confirm the right alerts, they will send a pop-up alarm by using the popUpAlarm class to create it. Otherwise, they will remove alerts by using the function removeAlerts() if they are false. The dailyReport and yearlyReport include all information about the location, time and description of the status. And these reports will be organized by date and time. The reports class is used to classify all daily reports and yearly reports with all details, which are provided from dailyReport and yearlyReport.



❖ **Description of classes:**

1. ***employeeAccount:***
    - **Attributes:**
        - userName: String
        - userPassword: String
        - authentication: enum
    - **Operations:**
        - authUser(): enum
    - **Description:** This class creates employee (ranger) accounts. The userName, userPassword variables are for employees to create their login credentials. They are both strings. Authentication is an enum. They need to be authenticated through authUser(): enum. This class also allows users (rangers) to create new accounts for new users.
2. ***reports:***
    - **Attributes:**
        - location: String
        - time: String
        - status: String
        - reports: String[]
    - **Operations:**
        - fileReport(String[], String): void
        - getReport(): String[]
        - setStatus(String): void
        - getStatus(): String[]
        - setTime(String): void
        - getTime(): String
        - setDescription(String): void
        - getDescription(): String
        - setLocation(String): void
        - getLocation(): String
    - **Description:** In this class, we want to get the information about the location, time, description and status. We will get this detailed information by using getters and setters. The variables used are location, time, status with String type and reports variable is defined by String[] type. The get methods will take String and return string, while the set methods will not return anything. There will be an array called reports that will include all report filenames. fileReport(String[], String) will take an array of reports and add a new report to this array when given a String file name. getReport() will return a full array of reports.
3. ***dailyReport:***
    - **Attributes:**

    - statusDailyDescription: String
    - estimatedDailyTime: String
    - warningDailyLocation: String
    - dailyReport: String[]
- **Operations:**
    - checkMountainLion(): bool
    - deleteReport() : void
    - organize(String): void
    - displayDailyReports(): String[]
- **Description:** The purpose of this class is providing the information about time, location and status description to write the daily report. The dailyReport variables are statusDailyDescription, estimatedDailyTime, warningDailyLocation and dailyReport. Within this class, the checkMountainLion(): bool function will return true or false value which we need to check whether the sensor is about mountain lion or not based on the definite alerts. If they are not, then users can delete them by using the deleteReport(): void function, this function requires the report that is to be deleted. The function organize(String): void is used to arrange the report date by date for users to look up. displayDailyReports(): String[] function will return a string array including all information to be used in the reports class.

4. ***yearlyReport:***
    - **Attributes:**
        - statusYearlyDescription: String
        - estimatedYearlyTime: String
        - warningYearlyLocation: String
        - reportsYearly: String[]
        - yearlyReport: obj
    - **Operations:**
        - organize(String): void
        - displayYearlyReports(): String[]
    - **Description:** The purpose of this class is providing the information about alerts received older than 30 days, but within the year. The yearlyReport variables are statusYearlyDescription, estimatedYearlyTime, warningYearlyLocation, reportsYearly, and yearlyReport. The function organize(String) is used to arrange the report date by date for users to look up. displayYearlyReports() function will return a string array including all information to be used in the reports class.

5. ***getAlerts:***
    - **Attributes:**

    - alertEnabled: bool
    - list_Alerts[]: obj
- **Operations:**
    - createAlert(getLocation()): void
    - removeAlerts(): void
    - displayAlertList(): Alerts
- **Description:** The purpose of this class is to get alerts to inform the park rangers and to prioritize if there is a mountain lion or not. The variables for this class are alertEnabled: bool which will define whether the alert is either true or false and list_Alerts[] which is an array containing alerts.The functions will be used in this class are createAlert(getLocation()): void which creates an alert to notify the park rangers, removeAlerts(alert): void usually removes the alerts that have already been dealt with or if alerts are false, and displayAlertList(): Alerts display the alerts on the main computer so that the park rangers take appropriate measures to deal with the alerts.

6. ***getNoiseDetectors:***
    - **Attributes:**
        - alert: bool
        - location: String
        - typeAnimal: String
        - strengthAnimal: int
    - **Operations:**
        - getAlerts(): bool
        - setLocation(String): void
        - getLocation(): String
        - setType(String): void
        - getType(): String
        - setStrength(int): void
        - getStrength(): int
    - **Description:** This class is responsible for detecting noise which happens to send an alert to the main computer. getAlerts():bool which creates an output whether it is true(detects noise) or false(does not detect noise). setLocation(): String specifies where the location of the noise is coming from which helps the park ranger. getLocation(): String is responsible for getting the location of the noise detector. setStrength(int): determines the strength of the noise and getStrength(): returns the strength.

7. ***popUpAlarms:***
    - **Attributes:**

    - popUpAlarm: String
    - popUpAlarmCheck: bool
- **Operations:**
    - displayPopUp(): String[]
    - SetAlarm(bool): void
- **Description:** The purpose of this class is that once park rangers log in with their info they will get a pop-up alarm to notify them that there is a mountain lion right away so that they won't have to waste time looking for the alert. The variables are popUpAlarmCheck: bool which outputs if the alarm is true or false so that the rangers deal with it swiftly and popUpAlarm: String which will include the alarm message to pop-up. displayPopUp(): String[] is usually what's responsible for displaying the pop-up as soon as the park ranger logs in the display pop-up will show so that they can be notified right away. SetAlarm(bool): void is used for setting up the alarm which the park ranger sees as soon as they login and does not return anything. If it is true they will be displayed the pop-up if it is false then nothing will happen.

IV. Development Plan and Timeline:

1. Partitioning Tasks

To create the system in an orderly fashion we would split up the creation of classes amongst ourselves. Dylan would handle the reports class and its respected children classes, dailyReport and yearlyReport classes. He would be responsible for the creation of the systems major database. Hannah would create the employeeAccount class. She will have to handle the ranger's account information and creation of the employee account portal. She is also responsible for the users database. Finally, Hisham would make the getNoiseDetectors class, the popUpAlarm and getAlerts classes. He would have to tackle the detection part of our system needed to use integral geolocation tools for the noise detectors.

2. Team member responsibilities

The entire team is responsible for completing this system within a timely manner. We believe that a deadline of three months should be sufficient time to complete our respective tasks and a final 2-4 weeks for testing and working out any bugs to create a user friendly and functional Mountain Lion Detection system.

V. Future Modifications:

In the future, the system can be upgraded to expand its functionality. The system can extend its graphical map of detections from within 2 miles to 5-10 miles. The noise detectors can be upgraded to recognize more frequencies to detect other types of animal, not just limited to lions. Developers can create an application for visitors to use to alert them that their location is within the detection of lions. Moreover, the system can add another database for pending alerts - those alerts that rangers cannot verify successfully, and rangers can send those pending alerts to a third party to examine.

VI. Potential Threats and System Vulnerabilities:

Using the Confidentiality, Integrity, and Accessibility (CIA) categorization for threats and vulnerabilities, this system is vulnerable to attacks on its database information. The attackers may try to obtain user information for malicious purposes. Also, wildlife smuggling criminals may try to get information from the reports for their criminal activities.

Two potential techniques that attackers typically use to get user information are the Brute-Force Attack and SQL injection. The system's network may also be penetrated by other external hackers.

VII. Verification Test Plan:

1. ***Verification Test Plan - Unit:***
   employeeAccount:

These tests will validate that each unit of the employeeAccount class's functions performs as expected. The employeeAccount is supposed to authorize the user's login information based on username and password.

- Target components:
  - authUser(): enum
- Test Strategies:

  authUser(): enum - will authenticate the user given a user-inputted username and password

  if(authUser() != authentication) {fail}
  else {pass}
- Test Vectors:
  - Enum employeeAccount = "user, password"; Result = "user, password"
  - Enum employeeAccount = "MikeWilliam, mike2879"; Result = "MikeWilliam, mike2879"

- Tests:
    - employeeAccount MikeAcc
        - username: MikeWilliam
        - password: mike2879
        - enum MikeAcc(MikeWilliam, mike2879)
    - void TestAcc1() {Assert.IsEqual(authUser(), MikeWilliam, mike2879)}
    - employeeAccount HaleyAcc
        - userName: HaleyDunphy
        - password: haley2563
        - enum HaleyAcc(HaleyDunphy, haley2563)
    - void TestAcc2() {Assert.IsEqual(authUser(), HaleyDunphy, haley2563)}

getNoiseDetectors:

These tests will validate that each unit of the getNoiseDetectors class's functions performs as expected. This class is supposed to receive and send location information with the strength of the noise to create an alert which will notify the park rangers.

- Target components:
    - getAlerts(): bool
    - setLocation(String): void
    - getLocation(): String
    - setStrength(int): void
    - getStrength(): int
- Test Strategies:
    - getAlerts(): bool - will determine the alert is true or false
        - if(currAlert != testAlert) {fail}
          else {pass}
    - setLocation(String): void - will set the location where the noise detector recognizes lions.
        - if(location != testLocation) {fail}
          else {pass}
    - getLocation(): String - will get the location where the noise detector recognizes lions.
        - if(getLocation() != testLocation) {fail}
          else {pass}
    - setStrength(int): void - will set the strength of the noise by integers.
        - if(currStrength != testStrength) {fail}
          else {pass}
    - getStrength(): int - this function will return the strength of the noise.
        - if(currStrength != testStrength) {fail}
          else {pass}
- Test Vectors:

    - getAlerts
        - boolean testAlert = true. Result: currAlert = true
        - boolean testAlert = false. Result: currAlert = false
    - setLocation
        - String testLocation = “Parking 12”. Result: currLocation = “Parking 12”.
        - String testLocation = “Parking 3”. Result: currLocation = “Parking 3”.
    - getLocation
        - String testLocation = “Parking 12”. Result: getLocation() = “Parking 12”.
        - String testLocation = “Paring 3”. Result: getLocation() = “Parking 3”.
    - setStrength
        - int testStrength = 3. Result: currStrength = 3.
        - int testStrength = 5. Result: currStrength = 5.
    - getStrength
        - int testStrength = 3. Result: getStrength() = 3.
        - int testStrength = 5. Result: getStrength() = 5.
- Tests:
    - void TestGetAlerts1 {Assert.IsEqual(getAlerts(), true)}
    - void TestGetAlerts2 {Assert.IsEqual(getAlerts(), false)}
    - void TestSetLocation1 {Assert.IsEqual(setLocation(“Parking 12”, “Parking 12”)}
    - void TestSetLocation2{Assert.IsEqual(setLocation(“Parking 3”, “Parking 3”)}
    - void TestGetLocation1 {Assert.IsEqual(getLocation(), “Parking 12”)}
    - void TestGetLocation2 {Assert.IsEqual(getLocation(), “Parking 3”)}
    - void TestSetStrength1 {Assert.IsEqual (setStrength(3), 3)}
    - void TestSetStrength2 {Assert.IsEqual (setStrength(5), 5)}
    - void TestGetStrength1 {Assert.IsEqual (getStrength(), 3)}
    - void TestGetStrength2 {Assert.IsEqual (getStrength(), 5)}

getAlert:

These tests will make sure every unit function in the getAlert class performs as expected. By using these functions, the system will be able to create, remove and display all alerts getting from the noise detection system.

- Target components:
    - createAlert(getLocation()): void
    - removeAlerts(alert): void
    - displayAlertList(): String

- Test Strategies:
    - createAlert(getLocation()): void - will create an alert to notify the user.
        - if(currAlert != testAlert) {fail}
          else {pass}
    - removeAlerts(alert): void - remove the alerts that have already been dealt with or if alerts are false
        - if(currRemoveAlerts != testRemoveAlerts) {fail}
          else {pass}
    - displayAlertList(): String - display the alerts on the main computer so that the park rangers deal with the alerts.
        - if(currAlertList != testAlertList) {fail}
          else {pass}
- Test vectors:
    - createAlert(getLocation())
        - String testAlert = "NOTICE: There are lions near parking 12.".
          Result: currAlert = "NOTICE: There are lions near parking 12."
        - String testAlert = "NOTICE: There are lions near parking 16."
          Result: currAlert = "NOTICE: There are lions near parking 16."
    - removeAlert(alert)
        - String testRemoveAlert = "Remove the current alert." Result: currRemoveAlert = "Remove the current alert."
    - displayAlertList()
        - List<alert>= testAlertList. Result: currAlertList = List<alert>
- Tests:
    - void TestCreateAlert1 {Assert.IsEqual(createAlert("Parking 12"), "NOTICE: There are lions near parking 12.")}
    - void TestCreateAlert2 {Assert.IsEqual(createAlert("Parking 16"), "NOTICE: There are lions near parking 16.")}
    - void TestRemoveAlert1 {Assert.IsEqual(removeAlert(alert1), "Remove the first alert.")}
    - void TestRemoveAlert2 {Assert.IsEqual(removeAlert(alert2), "Remove the second alert.")}
    - void TestDisplayAlertList() {Assert.IsEqual(displayAlertList(), "Parking 12, Parking 3, Parking 16")}

2. Verification Test Plan - Functional/Integrational:

This test employeeAccount will be responsible for checking each application feature performs as expected such as adding new username, adding new password, defining new

authentication, and also deleting all that information. While doing the tests, each function is compared to the corresponding requirements to ascertain whether its output is consistent with our expectations.

employeeAccount
username password authentication
List<String>: username List<String>: password List<enum>: authentication
void addUsername(...) void deleteUsername(...) void addPassword(...) void deletePassword(...) void addAuthentication(...) void deleteAuthentication(...)
Test Vectors: employeeAccount.addUsername (“MikeWilliam”). Result: username = “MikeWilliam” employeeAccount.addPassword(“mike2879”). Result: password = “mike2879” employeeAccount.addAuthentication(“MikeWilliam”, “mike2879”). Result: authentication = “MikeWilliam - mike -2879”

This test is responsible for getting alerts which display whether they are true or false with true meaning a mountain is detected and false being no mountain lion is detected. It also creates an alert with the location of where the mountain lion is detected which in our case is (Near Parking 12).

getAlerts()
alertEnabled: bool list_Alerts[]: obj
list<Alert>:alertsList
void addAlert(...) void removeALert(...)
Test Vectors:

alertEnabled.getAlerts(). Result: alertEnabled = true alerts.addAlert(“NOTICE: Near Parking 12”). Result: alerts = “NOTICE: Near Parking 12”

*The test (popUpAlarm) is connected to getAlerts, as soon as they log in,if the popUpAlarm returns true then this alarm creates a popUpAlarm that notifies the ranger which saves time and lets them know if there is a mountain lion in the area right away, it will display the message(*NOTICE: there are lions near parking 12) *so they can take appropriate action to close off the area.*

popUpAlarm()
popUpAlarm: String popUpAlarmCheck: bool
list<Alarm>: popUpAlarmList
void addPopUpAlarm(...) void deletePopUpAlarm(...) void addAlarmCheck(...) void deleteAlarmCheck(...)
Test Vectors: test.addPopUpAlarm(“NOTICE: there are lions near parking 12”). Result: popUpAlarm= “NOTICE: there are lions near parking 12” test.addAlarmCheck() = true. Result: popUpAlarmCheck = true.

This test is responsible for getting noise detectors which send an alert to the main computer, the alert could be true or false. The location outputs where the alarm is coming from which in our case is (Parking 3). The typeAnimal string is responsible for returning the type of the animal that has been detected. And finally, the strength of the animal returns an int.

getNoiseDetectors
alert: bool location: String typeAnimal: String strengthAnimal: int

List<Alert>: alertList List<String>: locationList List<String>: animalList List<int>: strengthList
void addAlert(...) void deleteAlert(...) void addLocation(...) void deleteLocation(...) void addAnimalType(...) void deleteAnimalType(...) void addAnimalStrength(...) void deleteAnimalStrength(...)
Test Vectors: currAlert.addAlert(true). Result: alert = true currAlert.addLocation("Parking 3"). Result: location = "Parking 3" currAlert.addAnimalType("Lions"). Result: typeAnimal = "Lions" currAlert.addAnimalStrength(3). Result: strengthAnimal = 3

3. ***Verification Test Plan - System*:**
   MountainLionDetection:
   (All rangers can look at information and can edit it)

This test is responsible for making sure all parts of our system are connected correctly so that the rangers can view each of the pages listed below and maneuver through the system easily, overall being user friendly. All of these pages and functions listed below will be viewable by all rangers who login. No information or functions are hidden from other rangers such as admin privileges

Employee Login (employeeAccount)
Alerts Page (getAlerts)
Recent Reports Page (dailyReport)
Yearly Report Page (yearlyReport)
Pop-Up Alarm (popUpAlarm)
Noise Detectors (getNoiseDetectors)

This test is to ensure that the rangers can also perform these functions when viewing each page, such as creating reports or creating new ranger profiles to login which are essential for them to complete their job.

- Add Ranger - User (authentication)
- Add Username
- Add Password
- Add Report
- Add Mountain Lion Status to Report
- Add Report Location
- Add Time to Report
- Add Pop-Up on/off
- View Recent Reports
- View Yearly Reports
- View Alerts
- Get Alerts (for Filing a Report)
- Get Report(s)
- Edit Report
- Edit Alert
- Delete Report(s)
- Delete Alert(s)

VIII. Validation Test Plan:

1. ***Validation Test Plan - Checking the Alerts and Reports on the controlling computer:***

When a detector detects a mountain lion's voice, it will automatically send the signal to the controlling computer. The rangers who monitor the controlling computer will see the alert. They can examine the alert by looking at its details. The rangers can also see the reports in the controlling computer to gather information for various purposes.

2. ***Validation Test Plan - Creating a new report:***

When the rangers successfully verify the alert, they have the option to edit it. Then, they can save it and create a new report in the controlling computer.

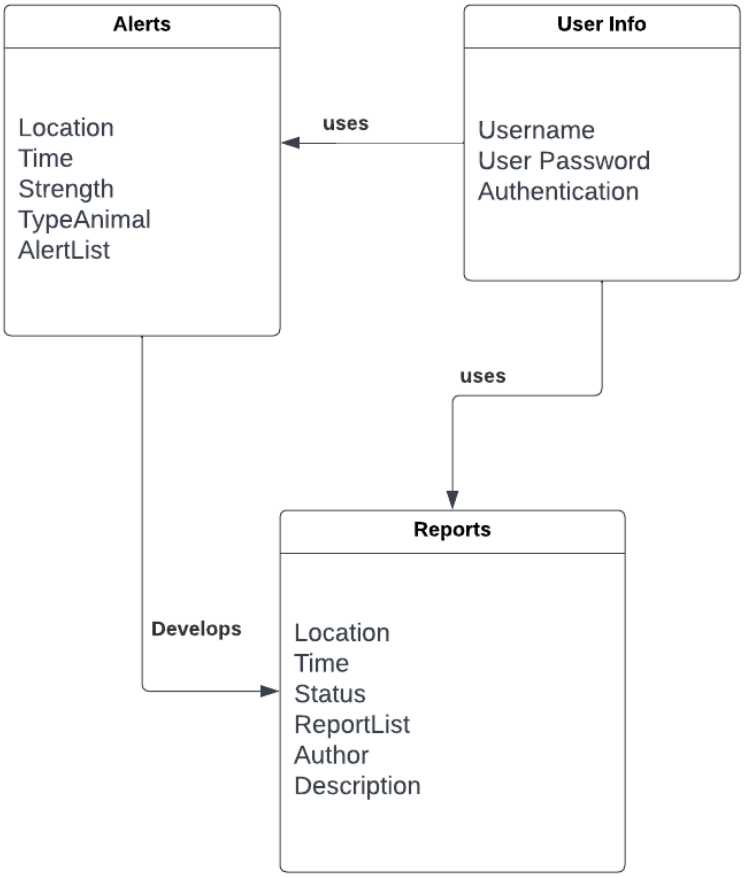
3. ***Validation Test Plan - Checking when the alarm is set off at the controlling computer:***

The rangers can also see the configuration at the controlling computer about the system. They can perform a check test to see if the configuration is correctly set up for the alarm.

IX. Data Management Strategy:

The Mountain Lion Detection System will have three separate databases. The first database will be the report database with the NoSQL column database structure, the second database will be the user accounts or login information with the SQL database structure, the last one will be the alerts database with the SQL database structure. Alerts database will only be accessed by the appropriate authorized users (rangers) of the park which contains location, time, type of animal, and strength of the noises. Report database is also accessible by authorized users only which contains daily reports and yearly reports. Both these databases will use the login database to ensure proper authorization for accessing all the information. Since all the databases are separated as they are to emphasize and improve the security. They will only allow a selected group of users such as the rangers of the park to access the alerts and report database. Using three different databases provides faster speeds and easier access since all information is divided into separate storage with specific details.

The data dictionary as well as the drawing for the data dictionary are shown below. The drawing shows how each of the three databases are divided and listed their contents. The report and alerts databases both use the login information database. This is important to ensure that only authorized users are able to access the information in the databases. The login information database is an important step to the database implementation which is necessary for them to use the function. This drawing is an overall description of the databases. To be more specific, the data dictionary goes into detail about how the database is implemented and provides the variable name, data type and brief description. It would be efficient and well organized to use when designing the system. By using the login information database, the authentication process will be verified to allow the user access to the other databases which helps the system to improve the security.



Data Dictionary:

Name: reports	Name: userInfo	Name: alerts
Name: Reports Type: No SQL Description: it is responsive for getting reports Name: Location Type: String Descriptio: it gets the	Name: LogInfo Type: SQL Description: username,password Name: username type:String description:Loginfo	Name: Alerts Type: SQL Description: it gets alerts Name: Location Type: String Description: gets the location of the alert

location of the report Name: Time type :string Description: it outputs time of the report Name: Status Type: string Description: shows the status of reports name : ReportList Type: string Description: displays the list of alerts Name: Author Type: string Description: displays the who the author is for the report Name: Description Type: string Description: shows the description of the chosen report.	name: authentication Type: enum Description: verification for LogIn	Name: Strength Type: int Description: gets the strength of the animal Name: TypeAnimal Type: String Description: gets the type of the animal Name: Alertlist Type: string Description: displays the list of alerts

We chose our reports database to be in noSQL because the benefits of noSQL outweigh those from SQL. We expect the reports database to eventually become a long and expansive list of reports spanning multiple years worth of documentation. NoSQL benefits from databases full of documentation. NoSQL is also flexible and can easily be adjusted for new information. There is also the possibility of this system scaling to parks all over the US and noSQL databases are generally cheaper and have much better scalability making it easy to scale if need be.

Our userInfo database was chosen to be SQL mainly because of how secure it is. All of the information stored on these databases are for the eyes of the park rangers and SQL allows the user to revoke and grant access to a database. SQL will be better for restricting access to the other databases. Not to mention that we wouldn't have an incredibly long list of users who can have access at a given time. While park ranger numbers can be large they won't number in the hundreds.

Lastly the Alerts database was chosen to be SQL. We decided this mainly because of how quick SQL can be in query processing. Whenever a ranger is creating a report they should

be able to gather what alert they are looking for in a quick manner. This database would also not be in anysort of hierarchy as it should be simply a list of alerts.

C. Life - Cycle Model:

In this project, to optimize the development of the system, we chose to use the waterfall life-cycle model. The waterfall model is broken down into sequential phases: Requirements, design, implementation, verification, and maintenance. We began by searching and gathering the requirements of the Mountain Lion Detection System. After organizing the requirements, we divided them into sections and started to design the system. Doing this helped us to keep track of the progress. After testing and implementing, we were able to identify the verification and acceptance of our system. In the final phase, we identified ways to maintain the system and suggestions for future improvements. We also discussed security and potential threats of the system. It is straightforward for us to do this project because we used the waterfall model and it worked really well for us. However, because of its sequential phases, we had to go back and forth between the phases to revise the changes. Also, the diagrams helped us to clearly understand the system. If the system is more complex, we think we may choose a different life-cycle model such as the spiral model, because we have to make a lot of changes when testing and implementing.