Project Report – Kili Trekker System

Group Members : Dante Viscuso, Eduardo Barraza,

MichaelJames Gurtiza, Animesh Srivastava

KILI TREKKER SYSTEM

TABLE OF CONTENTS

User Manual

This software system is going manage to all relevant and real-time information about the 12 available trails located at the Kilimanjaro National Park. Relevant trail information includes weather reports, trail conditions, ongoing park events and any rebellious or malicious activity within the park. This system will work in tangent with the already in place cellphone towers scattered across the park to connect all the ranger stations to the main server. Users of this system should be able to interact with and view all information as needed inside and outside the national park. It will be the park rangers' and guides' job to keep the systems information up to date via ranger station or web-based cell phones. This system is being implemented with trekkers, guides, and park rangers in mind to provide a smooth experience through the trails, have information readily available, and provide a safe experience through various alerts of rebel activity and correction of incorrect information posted on the system.

This section is meant to also outline how a user might interact with the system as well as the requirements for these users. For an average trekker, you would connect to the system, possibly via a cellphone, and then select whatever information you seek to obtain, and the system will send you a full report such as the weather with the temperature and condition of the area. Rangers and guides would have the same access to this information, but they can also use the system by editing the information as well as any emergency information that needs to be relayed to the users. Some of the more specific requirements for the users includes things like the system being interactive for a great user experience, being secure by having emergency notifications and updates sent to the users, and having all the data readily available and up to date is crucial for the user.

<u>Design Documents</u>

Functional Requirements: In essence, this system will function in 2 types of manners. It displays the information of the trails such as weather, events in the park, or emergencies which is consumed and used by anyone and everyone within the park. It must also require that the information is able to be updated, which is done solely by the park ranger and guide. Some of these requirements include making sure that the rangers have edit access when connecting to the system and that the information they inserted in is consistent throughout the entirety of the park. Another functional requirement would be that the trekkers are denied access to editing any of the information and if they somehow got in and did so, it would alert the other trekkers immediately. Furthermore, other requirements include more specific things like viewing ongoing evens or accessing the camera.

Non-functional Requirements: Seeing as how rebel activity is extremely prevalent within the area that this system is being implemented, most of the nonfunctional requirements revolve around security/safety and establishing the most secure system possible. For example, one of the requirements is to enforce security against rebel and malicious attacks on the system. This is to prevent rebels from not only accessing the information itself within the system but also shutting the entire system down from the inside. Another requirement that goes hand in hand with the previous is allowing access only to park rangers and guides so that these types of attacks are far less likely, although not impossible. A requirement that falls out of the scope of security but is just as important is about usability. The environment in which this system is established can be very harsh, so it is required that this system works under extreme weather conditions. This would allow for a much better uptime as well as more coverage on weather-based emergency situations.

Validation

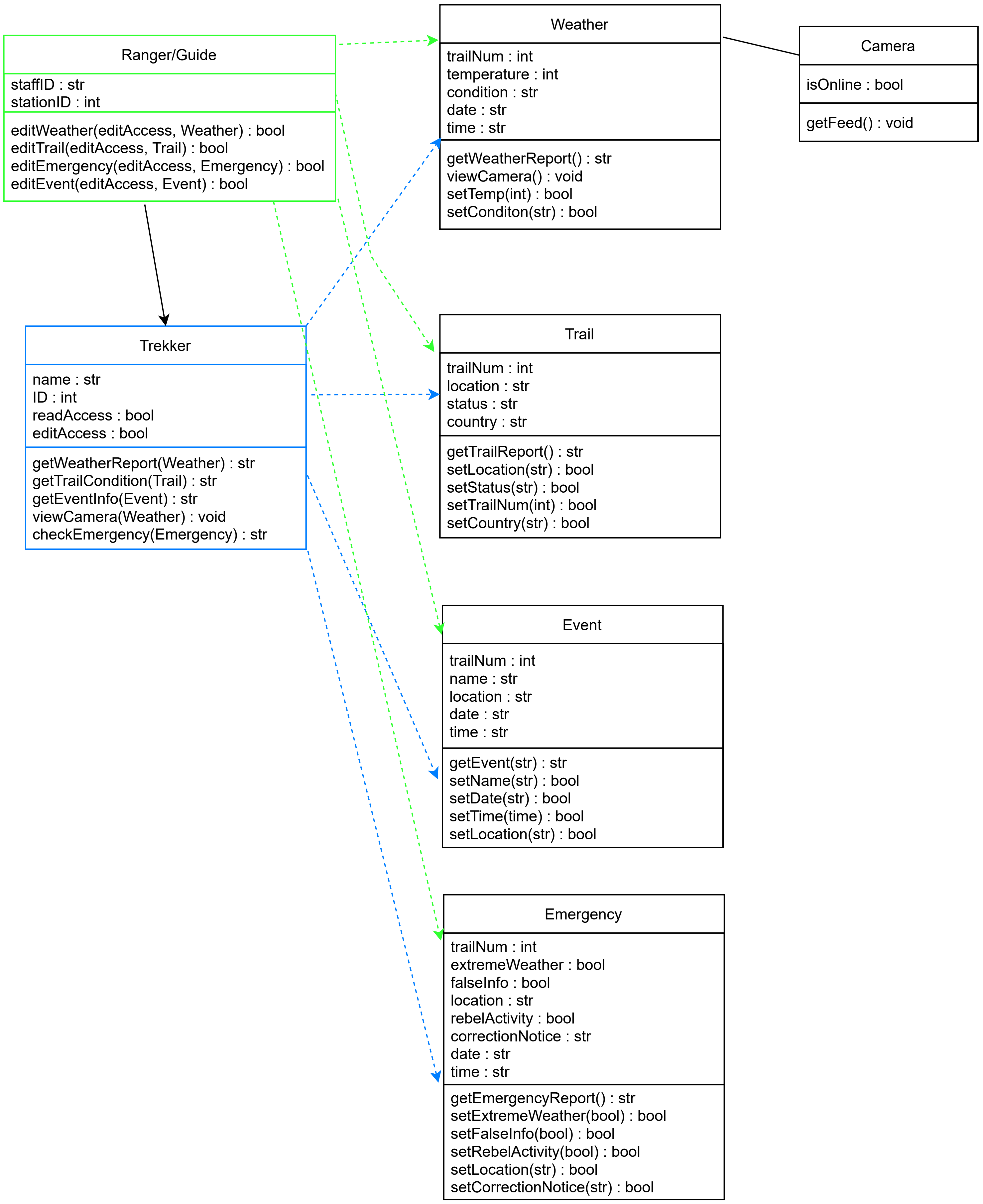
Validation will consist various uses cases that will be performed by 10 users, 5 trekkers and 5 rangers. They will perform normal usage such as getting information about the trails/park and use the system in different ways to ensure that the requirements are all valid

The first set of testing will consist of the 5 trekkers. The trekkers will need to access the weather, trail, event, and emergency section within the system to make sure all the information is readily available. For example, the trekker will need to access the weather section and ensure the temperature and condition are properly prompted and match the current weather and they would also need to access the camera and ensure that it is showing a live feed. Next, they will access the trail and event section and they will try and generate a report for each respective section. Finally, there will be a mock emergency to ensure the safety and notification requirements are fully functional

The second set of testing will include 5 rangers, these tests consist of changing and updating information on different sections All of them will attempt to access the system from various location in the park on their phones to make sure the towers are working properly and then change the information for each section to ensure consistency. Next, there will be wrong information added and it needs to give warnings to all current trekkers and a correction notice following. Finally, they will try to access the editing functions of the system without their ranger ID which should block them out immediately.

The last set of testing will simply be having all 10 users test the system not only during harsh weather but also during various times of the day, conditions, and even emergencies. A combination of all three of these sets of tests will ensure that the requirements are up to standard as per request of the client.

UML Class Diagram



UML Class Description

Class: Trekker

This is the basic user class which will determine if they can read/edit and keeps track of the user's name in order to keep track of all trekkers in the park. The class will outline all the functions that can be performed by the average user.

Attributes:
<u>name : str</u> – holds the name of the user to ensure they are a verified park goer
<u>ID : str</u> – an ID to identify trekkers at the park
<u>readAccess : bool</u> – true or false variable that will determine if they can read the information from the system (always true)
<u>editAccess : bool</u> – true or false variable that will determine if they can edit the information of the system (false for the basic user class)

Operations:
<u>getWeatherReport(Weather) : str</u> – uses the weather class in order to generate a weather report in the form of a string
<u>getEventInfo(Event) : str</u> – uses the event class in order to generate an event report in the form of a string
<u>getTrailCondition(Trail) : str</u> – uses the trail class in order to check the various trails and any current conditions with them in the form of a string
<u>checkEmergency(Emergency) : str</u> – uses the emergency class to check if there are any ongoing events, returns this info in the form of a string
<u>viewCamera(Weather) : void</u> – uses the weather class in order to connect to the mountain camera for a live feed view, returns nothing

Relationships:
<u>Trekker</u> uses Weather for the weather report and camera view
<u>Trekker</u> uses Event for information on current events
<u>Trekker</u> uses Trail for the trail conditions
<u>Trekker</u> uses Emergency to star ware of any ongoing emergencies

Class: Ranger/Guide

This is the ranger/guide class which extends the basic 'Trekker' class in order to give it specific functions unique to only this class. The class will outline all the functions that can be performed by the rangers and guides of the trails.

Attributes:
<u>staffID : str</u> – an ID or badge number to keep track of which ranger or guide is in use of this class
<u>stationID : str</u> – an ID that identifies the station ranger works at

Operations:
editWeather(editAccess,Weather) : bool – setter-like functions which allows the ranger/guide to input or edit the data relating to the weather class, such as temperature or the overall condition, returns true when successful
editEvent(editAccess,Event) : bool – setter-like functions which allows the ranger/guide to input or edit the data relating to the event class, such as a future event along with its name, returns true when successful
editTrail(editAccess,Trail) : bool – setter-like functions which allows the ranger/guide to input or edit the data relating to the trail class, such as the trail number along with its current status, returns true when successful
editEmergency(editAccess,Emergency) : bool – setter-like function which allows the ranger/guide to input or edit the data relating to the emergency class, such as if there is an emergency regarding harsh weather or rebel activity, returns true when successful

Relationships:
Ranger/Guide uses Trail to edit the trail conditions
Ranger/Guide uses Weather to input the current weather
Ranger/Guide uses Events to edit and input any information about events at the park
Ranger/Guide uses Emergency to update the system of any ongoing emergencies to keep the trekkers safe
Ranger/Guide is a Trekker class so it can do everything a trekker can, with the added edit functions

Class: Weather
This is the weather class which will contain all current and future information about the weather conditions. This class will outline the various weather-related function and the ways a user might interact with this class.

Attributes:
trailNum : int – an integer that will distinguish which of the 12 trails is being referenced
temperature : int –holds the temperature at current time
condition : str – holds the condition at current time
date : str – a string variable that will hold calendar date in which the weather is taken
time : str – a string variable that will hold the time of day the weather is taken

Operations:
getWeatherReport() : str – basic function that generates a weather report in the form of a string by using its private class variables
setTemp(int) : bool – setter function that will allow the input or editing of temperature at Mount Kilimanjaro National Park, returns true if change is successful
setCondition(str) : bool – setter function that will allow the input or editing of weather conditions, returns true if change is successful
viewCamera(Camera) : void – connects to the live feed camera by using the camera class

Relationships:
Weather is associated with the camera class to be able connect to the mountain camera for a live feed view
Weather is used by Ranger/Guide class to edit the class's information through its functions
Weather is used by the Trekker class to view a weather report

Class: Trail
This is the trail class which will contain all current information about the trail conditions. This class will outline the various trail-related function and the ways a user might interact with this class.

Attributes:
trailNum : int – an integer that will distinguish which of the 12 trails is being referenced
location : str – a string that will hold the geolocation of the trail start
status : str – a string that hold the current status of the trail
country : str – a string that will hold which country the trail belongs to

Operations:
getTrailReport() : str – basic function that generates a trail report in the form of a string by using its private class variables
setLocation(str) : bool – setter function that will allow the input of coordinate of trail's start as a string, returns true if change is successful
setStatus(str) : bool – setter function that will allow the input of the status of the trail as a string, returns true if change is successful
setTrailNum(int) : bool – setter function that will allow the input of a unique identification number for a trail as integer, returns true if change is successful
setCountry(str) : bool – setter function that will allow the input of which country the trail belongs to as a string, returns true if change is successful

Relationships:
Trail is used by Ranger/Guide subclass to edit the class's information through its setter function
Trail is used by the Trekker class to view a trail report of a given trail

Class: Camera
This is the camera class in which its sole purpose is to be called through the weather class by the trekker in order to show a live feed view of the trails. The class will outline the one function that is performed in this class.

Attributes:
isOnline : bool – true or false variable that will determine if the camera is online and ready

Operations:
getFeed() : void – simple function which is called by the weather class in order to connect to the mountain top camera

Relationships:
Camera is associated with the Weather class because the Weather class 'has a camera' in order to show a live weather feed
Camera is used through the weather class by the user to again show a live weather feed

Class: Emergency
This is the emergency class that will be responsible for dealing with any and all emergencies that occur within the park. This class will outline the different functions that can be performed by the trekker and the ranger/guide.

Attributes:
trailNum : int – an integer that will distinguish which of the 12 trails is being referenced
extremeWeather : bool – true or false variable that will determine if there is extreme weather within the area
falseInfo : bool – true or false variable that will determine if there is false information that has been inputted in the system
location : str – a string variable that will hold the location of the emergency
rebelActivity : bool – true or false variable that will determine if there is ongoing rebel activity within the area
correctionNotice : str – a string variable that will hold what the false information is and where is currently being shown
date : str – a string variable that will hold calendar date in which the emergency occurs
time : str – a string variable that will hold the time of day the emergency occurs

Operations:
getEmergencyReport() : str – uses the emergency class so that the trekker can see the emergency report in a form of a string
setExtremeWeather(bool) : bool – uses the emergency class so that the ranger/guide can set any emergency report, returns true when successful
setFalseInfo(bool) : bool – uses the emergency class so that ranger/guide can report if any info are false, returns true when successful
setRebelActivity(bool) : bool – uses the emergency class so that the ranger/guide can report any rebel activities, returns true when successful
setLocation(str) : str – uses the emergency class in order to set the location of the emergency in a form of a string
setCorrectionNotice(str) : bool – setter function to allow the input of a string to change correctionNotice, returns true when successful

Relationships:
Emergency is used by Ranger/Guide subclass to edit the class's information about any ongoing emergency through its setter function
Emergency is used by the Trekker class to check if there are any emergencies in the park

Class: Event
This Event class will be able to let trekkers know of information about events at the park. This class will also be able to let the Ranger/Guide edit/set any event that is happening at the park as well.

Attributes:
trailNum : int – an integer that will distinguish which of the 12 trails is being referenced
name : str – a string variable that will hold the name of the event
location : str – a string variable that will hold the location of the event
date : str – a string variable that will hold calendar date in which the event occurs
time : str – a string variable that will hold the time of day the event occurs

Operations:
getEvent(str) : str – uses the event class with string parameter to generate the event name in the form of a string
setName(str) : bool – uses the event class in order to set the name of the event in the form of a string, returns true when successful
setDate(str) : bool – uses the event class in order to set the date of the event in the form of a string, returns true when successful
setTime(str) : bool – uses the event class in order to set the time of the event in the form of a string, returns true when successful
setLocation(str) : bool – uses the event class in order to set the location of the event in the form of a string, returns true when successful

Relationships:
Trekker uses Event to determine the name, time, and location of the event
Ranger/Guide uses Event to set the name, date, time, and location of the current event

Test Plan

<u>Unit Tests</u>

Testing one component of the system for each test

In our test plan, we want our unit tests to mainly cover the class functions of the information holding classes such as the Event class, the Trail class, or the Weather class in order to ensure that the data is not only up to date, but also valid.

1. **setTrailNum(int) : bool**
   <u>Invalid input</u>: any number less than 1, any number larger than the total amount of trails (in this case it would be 12, null input
   - × {-1, false}
   - × {0, false}
   - × {“”, false}
   - × {234, false}

   <u>Valid input</u>: any number 1-max number of trails (at this time it is 12, but could increase)
   - ✓ {1, true}
   - ✓ {4, true}
   - ✓ {(max number of trails), true}

2. **setLocation(str) : bool**
   <u>Invalid input</u>: Invalid data would include anything besides a string containing numbers to representing to coordinate location. Strings that do not contain a comma separating numbers will be invalid. First number shall not be greater than 90 nor lesser than -90, representing latitude. The second number shall not be greater than 180 nor lesser than -180, representing longitude. The precision of these numbers shall not be anything other than 5 decimal places; allows for the use of the decimal degrees system used to represent GPS coordinates with about a precision of 1.11 meters. Adjacent coordinates shall not differ by more than 0.00010, 0.00010. Letters inside the string are invalid.

   - × {“-99.36193, 181.38353”, false}
   - × {{43.38291m, 34.28392m”, false}

   <u>Valid input</u>: Strings that contain 2 numbers separated by a comma. First number can be anything from -90 to 90, while second number is -180 to 180. Numbers shall have a precision of 5 decimal places. Each coordinate will only be 0.00010, 0.00010 from previous coordinate.
   - ✓ {“-3.06142, 37.31296”, true}
   - ✓ {“-3.28395, 37.52273”, true}

3. **setCorrectionNotice(str) : bool**
   Invalid input: any number such as int values and double values. It will also be invalid if nothing is entered.
   - × {-78, false}
   - × {24, false}
   - × {8.0, false}
   - × {" ", false} (" " represent an empty/nothing entered)

   Valid input: Any string will be valid. (errors in spelling or grammar will be valid so be careful)
   - ✓ {"There are 5 rebels nearby", true}
   - ✓ {"Trail 5 is unsafe due to falling rocks", true}
   - ✓ {"Weather is unsafe to hike today", true}

Integration Tests

Testing the relationship between components within the system for each test

In our test plan, we want our integration tests to cover the functions of the basic user class, trekker in this case. This is the most important part of the system: being able to retrieve the data such as the weather, or the trail condition - it is the whole reason this system even exist. With that said, it is crucial that we test the links between the user class and the information holding classes like Emergency and Event.

1. **getTrailCondition(Trail) : str**
   Invalid input: in this case, what is mainly being entered and the most crucial part would be the trailNum as it will create the report. So, like the unit test, its input would be any number less than 1, any number larger than the total amount of trails (in this case it would be a number larger than 12 or null input). However, since it is returning a string, the "false" equivalent would be a null string.
   - × {-5, ""}
   - × {0, ""}
   - × {"", ""}
   - × {93, ""}

   Valid input: any number between 1 and the max number of trails (at this time it is 12, but could increase). However, since it is returning a string, the "true" equivalent would be the trail report in the form of a string.
   - ✓ {6, "Trail 6 is open for trekking"}
   - ✓ {10, "Trail 10 is closed for trekking"}
   - ✓ {(number between 1 and max number of trails), "Trail (number) is open (or closed) for trekking"}

2. **getEventInfo(Event) : str**
   Invalid input: in this case the main component that will vary is the event name since the user would enter the name of an event, they want the info for, and the function will output a string which contains things like the location and date/time of said event. So invalid input would be a null name, anything other than a string such as a number, a string that doesn't contains letters or simply something that isn't an event and if it's invalid it would return a null string.
   - × {"", ""}
   - × {503, ""}
   - × {"+-..!#$", ""}
   - × {"milk", ""}

   Valid input: any valid event name, more specifically the name of an ongoing or future event that will happen in the park, and it would then return a full event report in the form of a string.
   - ✓ {"Summer BBQ", "Summer BBQ will take place on August 4th (12:15pm) at Mandara Hut"}
   - ✓ {"Jeanne's Yoga Class", "Jeanne's Yoga Class will take place on August 20th (08:15am) at Marangu Gate"}
   - ✓ {"(any valid event name)", "(Event name) will take place on (date) (time) at (location)"}

3. **getWeatherReport(Weather) : str**
   Invalid input: in this case, the wrong information or wrong order. It should be weather condition first then temperature; if these are inputted wrong, then it will be invalid.
   - × {"Tuesday, 95 °F, sunny", ""}
   - × {"thursday, sun, 94 °F"), ""}

   Valid input: a valid weather condition (rainy, sunny, cloudy, snowy), a valid day of the week and a valid temperature within the park. It would then return a full weather report in the form of a string.
   - ✓ {"sunny, Monday, 95 °F", "Monday will be sunny with a temperature of 95 °F"}
   - ✓ {"snowy, Friday, 32 °F", "Friday will be snowy with a temperature of 32 °F"}
   - ✓ {"(any valid weather condition), (temperature)", "(day) will be (condition) with a temperature of (average temperature)"}

System Tests

Testing the entire system

In our test plan, we want the system test to essentially be a collection of integration tests grouped together in which every single component is reached at least once. We want to make sure that after running this system test, we can be sure of the fact that each element in this system works to some degree (without seeing/writing the source code)

1. **editWeather(editAccess, Weather) : bool**
   editTrail(editAccess, Trail) : bool
   editEmergency(editAccess, Emergency) : bool
   editEvent(editAccess, Event) : bool
   viewCamera(Weather) : void

   Invalid input: There could be several invalid inputs when dealing with this large number of tests, but I think it is important to hit the main ones and perhaps the ones that could cause the most faults. First and foremost, if at any time editAccess is false, it is crucial that it returns false and allows no edits through. Another invalid input would be any case where you are trying to edit something like an event or a trail and enter a null string. Lastly, it would be hard to test since it returns void. But say that a user tries to view the camera and isOnline is false, it should show no video.

   × {true, “ ”, false} – leaving a blank field in the weather class
   {false, 12, false} – Correct trail number but no edit access
   {true, “21f3fea89”, false} – Invalid form of location for the emergency
   {false, “Trail 8 Cookout”, false} – Correct event name but no edit access
   {false, (no camera view)} – isOnline is false so the camera can’t show anything

Any combinations of these failure would then result in a failure of the system test, more specifically these are all examples of how to incorrectly use the system and how we expect it to react to such errors.

   Valid input: In terms of valid input for our particular system testing: it is necessary for editAccess to be true so that user can successfully edit the system’s information. Furthermore, trailNum’s input requires an actual trail number so that the input is valid. Additionally, valid input when testing to view the camera requires isOnline to be true so that camera can be accessed.

   ✓ {true, 40 , true } – Correct temperature format with edit access
   {true, 4, true} – Correct trail number with edit access
   {true, “-3.08138, 37.38907”, true} – Valid form of location for the emergency with edit access
   {true, “Trail 8 Cookout”, true} – Correct event name with edit access
   {true, (camera view)} – isOnline is true so the live camera feed is shown

2. getWeatherReport(Weather) : str
getTrailCondition(Trail) : str
editEvent(editAccess, Event) : bool
viewCamera(Weather) : void
checkEmergency(Emergency) : str

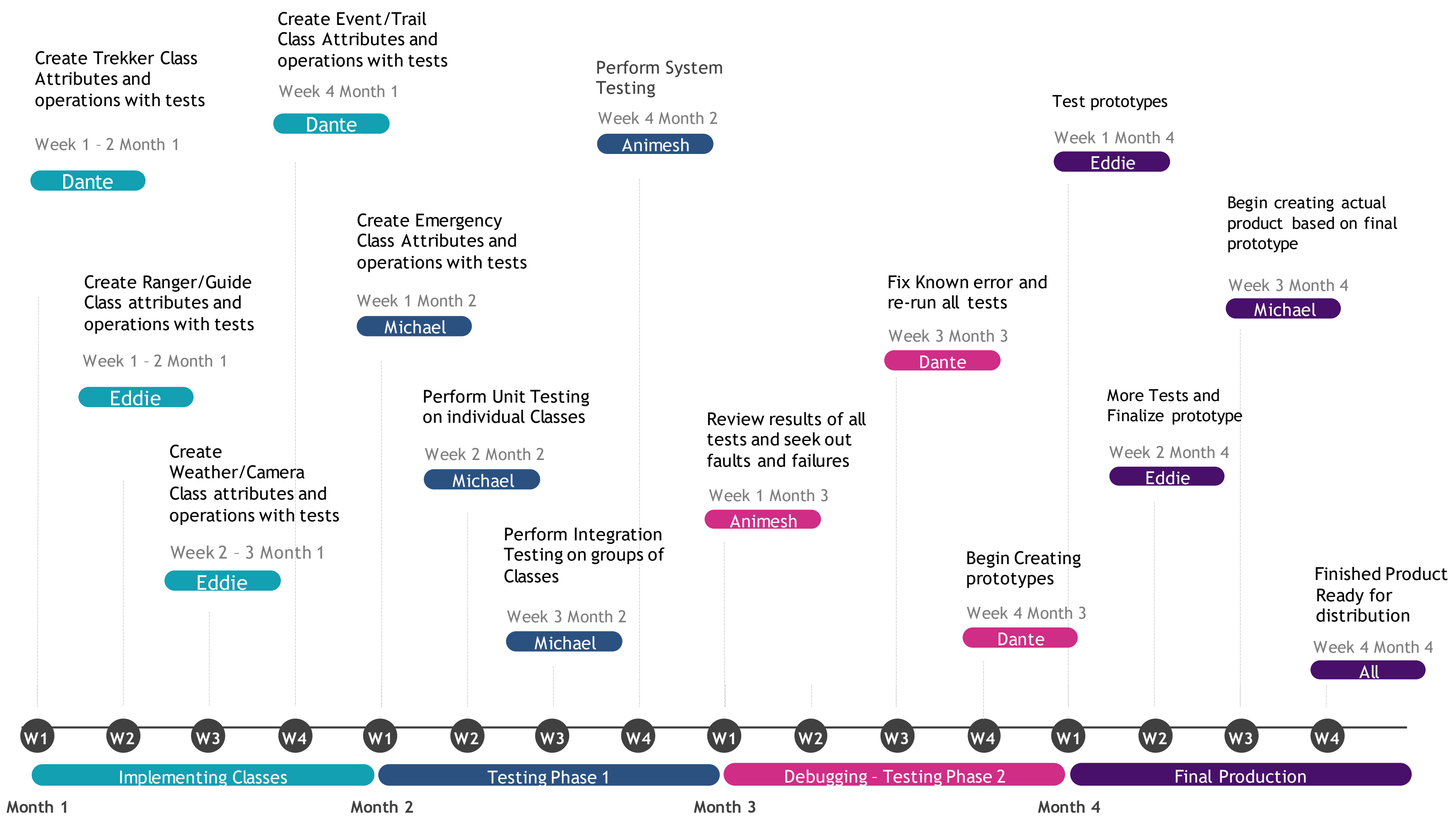
Invalid input: For the most part, this will be very similar to the previous system test. Things like editAccess being false at any point should return false for the edit functions. In terms of the get functions, formatting issues such as leaving fields null or inputting numbers for strings and vice versa. And once again, if isOnline is false then it will show no live feed of the camera.

× {" ", ""} – leaving a blank field in the weather class, results in an empty string being returned
{22, ""} – Nonexistent trail number inputted results in an empty string being returned
{true, "4th of July Fireworks", false} – Valid editAccess but invalid event entry results in a false Boolean being returned
{false, (no camera view)} – isOnline is false so the camera can't show anything
{"-2.9562, 37.4988", ""} – Not precise enough location (invalid), returns empty string

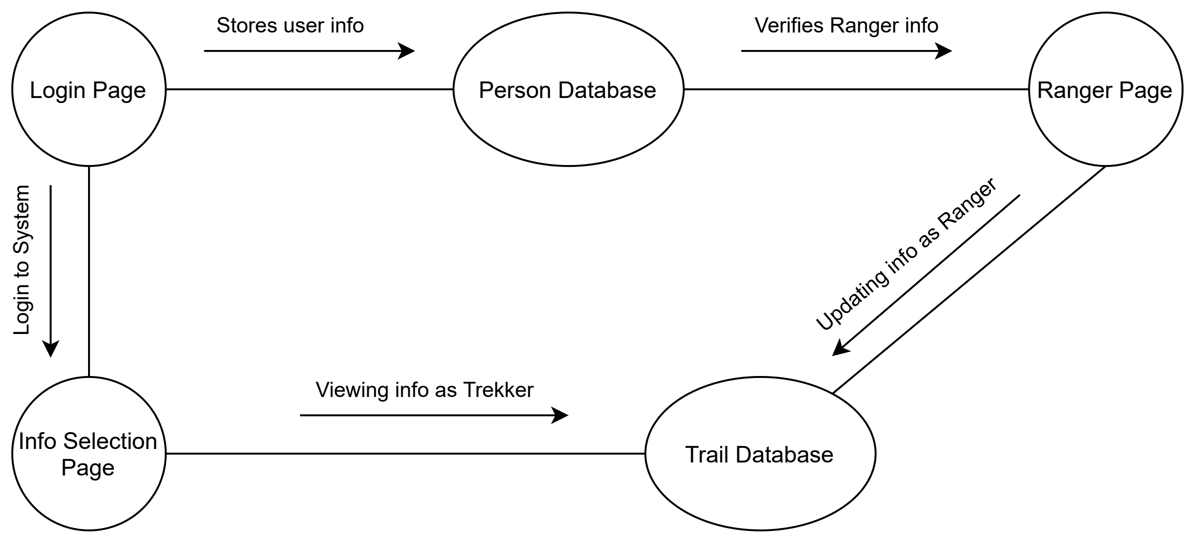
Valid input: Again, just like the last system test, valid input includes correct formatting of class attributes such as having a trail number be in the range of valid numbers, having real events, no null inputs and finally, having the camera's isOnline be true.

✓ {"cloudy, Thursday, 53", "Thursday will be cloudy with a temperature of 53 °F"}
{8, "Trail 8 is open for trekking"} – Valid trail number that is between 1 and number of total trails
{true, "Summer BBQ", true} – Valid editAccess with true event being accessed
{true, (camera view)} – isOnline is true so the live camera feed is shown
{"-2.95615, 37.49880", "Current rebel activity at this location"} – Valid form of GPS coordinate that then returns emergency notification as string

Kili Trekker System Timeline Nov 2021 – Feb 2022


Create Trekker Class Attributes and operations with tests
Week 1 - 2 Month 1
Dante
Create Event/Trail Class Attributes and operations with tests
Week 4 Month 1
Dante
Create Ranger/Guide Class attributes and operations with tests
Week 1 - 2 Month 1
Eddie
Create Weather/Camera Class attributes and operations with tests
Week 2 - 3 Month 1
Eddie
Create Emergency Class Attributes and operations with tests
Week 1 Month 2
Michael
Perform Unit Testing on individual Classes
Week 2 Month 2
Michael
Perform Integration Testing on groups of Classes
Week 3 Month 2
Michael
Perform System Testing
Week 4 Month 2
Animesh
Review results of all tests and seek out faults and failures
Week 1 Month 3
Animesh
Fix Known error and re-run all tests
Week 3 Month 3
Dante
Begin Creating prototypes
Week 4 Month 3
Dante
Test prototypes
Week 1 Month 4
Eddie
More Tests and Finalize prototype
Week 2 Month 4
Eddie
Begin creating actual product based on final prototype
Week 3 Month 4
Michael
Finished Product Ready for distribution
Week 4 Month 4
All
W1
W2
W3
W4
W1
W2
W3
W4
W1
W2
W3
W4
W1
W2
W3
W4
Implementing Classes
Testing Phase 1
Debugging - Testing Phase 2
Final Production
Month 1
Month 2
Month 3
Month 4

Architecture Diagram



Data Plan

Trail Database SQL

Trail
trailNum
location
status
country

Weather
trailNum
temperature
condition
date
time

Event
trailNum
name
location
date
time

Emergency
trailNum
extremeWeather
falseInfo
location
rebelActivity
correctionNotice
date
time

Person Database NoSQL
Document-Style

Trekker

name
ID
editAccess
readAccess

Ranger/Guide

name
staffID
stationID
editAccess
readAccess

Trekker Collection

Document 1

```
{
   "name": "John Smith"
   "ID": 1
   "editAccess": false
   "readAccess": true
}
```

Ranger/Guide Collection

Document 1

```
{
   "name": "John Smith"
   "staffID": 1
   "stationID": 1
   "editAccess": true
   "readAccess": true
}
```

Data Dictionary : Trail Database

Name : Trail
Description : Trail information
Type : Table
Primary Index : trailNum

Name : trailNum
Description : unique id for each trail
Type : Attribute/Int
Range : 1-12

Name : location
Description : location of the trail
Type : Attribute/String

Name : status
Description : current status of the trail
Type : Attribute/String

Name : country
Description : country in which trail is located in
Type : Attribute/String

Name : Weather
Description : Weather information
Type : Table
Primary Index : trailNum

Name : trailNum
Description : unique id for each trail
Type : Attribute/Int
Range : 1-12

Name : temperature
Description : the temperature of the trails in Celsius
Type : Attribute/Int
Range : 10-40

Name : condition
Description : the weather condition of the trails
Type : Attribute/String

Name : date
Description : current date
Type : Attribute/String

Name : time
Description : current time
Type : Attribute/String

Name : Emergency
Description : Emergency information
Type : Table
Primary Index : trailNum

Name : trailNum
Description : unique id for each trail
Type : Attribute/Int
Range : 1-12

Name : extremeWeather
Description : whether or not there is ongoing extreme weather
Type : Attribute/Bool

Name : falseInfo
Description : whether or not there is false info in the system
Type : Attribute/Bool

Name : location
Description : location of the emergency
Type : Attribute/String

Name : rebelActivity
Description : whether or not there is ongoing rebel activity
Type : Attribute/Bool

Name : correctionNotice
Description : correction notice for false information
Type : Attribute/String

Name : date
Description : date of the emergency
Type : Attribute/String

Name : time
Description : time of the emergency
Type : Attribute/String

Name : Event
Description : Event information
Type : Table
Primary Index : trailNum

Name : trailNum
Description : unique id for each trail
Type : Attribute/Int
Range : 1-12

Name : name
Description : name of the event
Type : Attribute/String

Name : location
Description : location of the event
Type : Attribute/String

Name : date
Description : date of the event
Type : Attribute/String

Name : time
Description : time of the event
Type : Attribute/String

Person Database Description

The Person Database is a NoSQL document-style database that is split up into two components, the Trekker document, and the Ranger/Guide document. We went with NoSQL because this database will be changing a lot in terms of people coming and leaving the trails so we wanted to be more flexible and adaptable with the document style because it will make it easy to remove people once they leave the park.

The Trekker document will hold all the info about a given hiker that enters the park, for example, the name which will be held in a string. The document will also hold the attribute of readAccess, in the form of a bool, so that they may have reading privileges while using the system. Furthermore, the document will also hold the attribute of editAccess, in the form of a bool, which will be used to check and prevent any writing or deletion of data by anyone that is not a ranger. Lastly, the document will hold an ID attribute, in the form of an int, which acts as their unique identifier in our person database.

In the Ranger/Guide will hold similar info to the trekker document but with added attributes that pertain to the duties of these individuals. These documents will contain the name of the ranger/guide as a string. The document will also hold the attribute of readAccess, in the form of a bool, to verify reading privileges. The document will also hold the editAccess attribute which will determine editing privileges and ensures they can update all info within the system. The ranger document will also contain the attribute of staffID in the form on an int that will serve as their unique identifier in this document database. Lastly, the document will hold the attribute of a station, which will hold the data of what station they are posted at and will also be an int.

System Security Measures

With the known rebels throughout the park, we are expecting a wide range of attacks that will pose potential threats based on the CIA categorization. We plan on carrying out many different counter measures to prevent and handle these incoming attacks.

The first type of attack we anticipate would hinder the confidentiality by trying to gain access of the information of people in the park. This could be done by faults in the person database and would need to have some level of prevention in this case. We propose that we not only keep the information of the trekkers as a number/ID but also when we do get their name for the person data base, we encrypt it to some degree in which it would be difficult to figure out their names if someone were to get access to this data.

The second type of attack would be aimed at the integrity of the system. We already addressed this issue to some extent with our emergency class and notifications within the system. To reiterate, this attack would try and edit the system and input false information. We deal with this problem by having constant checks of the system and if there were to be a successful attack that would harm the integrity of the system, we have correction notices sent out to the trekkers to inform them about it.

The last type of attack would try and prevent the availability of information. This could be done a few ways, such as gaining edit access and simply deleting the information. The way we plan on dealing with this type of attack would be to implement a firewall in order to prevent these types of attacks and provide a sort of shield access to internal network services. Another way we plan to address this problem is by running penetration tests to constantly gauge the level of security held by the system.

Life Cycle Model

The software development life-cycle model that we used while making our system was the waterfall method, which consists of one iteration of all 5 different steps. These steps include requirement gathering, design, implementation (which we skipped), verification and maintenance. It was a very straight forward developmental process while using this method, around a few weeks on each step spending time to not only make multiple drafts of each piece of documentation but also making sure that they are consistent with one another.

We gathered the requirements through client interviews and formed an exhaustive list of functional, non-functional, user, and other requirements. Next, we created a UML class diagram to model the general design of the system and to fulfill the gathered requirements. Then we created a test plan to make sure our UML class diagram works as desired through various unit, integration, and system tests. Finally, we created a plan for the data management in terms of which types of databases we would use and how they would be interconnected and formatted.

For our system this method felt like it simply got the job done and nothing more. It resulted in a finished product, but it is not as refined since this method only had one iteration, so something with multiple iteration of the 5 steps might have produced a much more polished result. For example, if we would have worked with the spiral method with more time on this project, it would have been much more complete and if it was real, less bugs. In terms of scalability, the method we used would not have worked since the increase in information would require many iterations to iron out the issues. In general, something with multiple iteration but still basic steps would work well in this case and would also scale well.