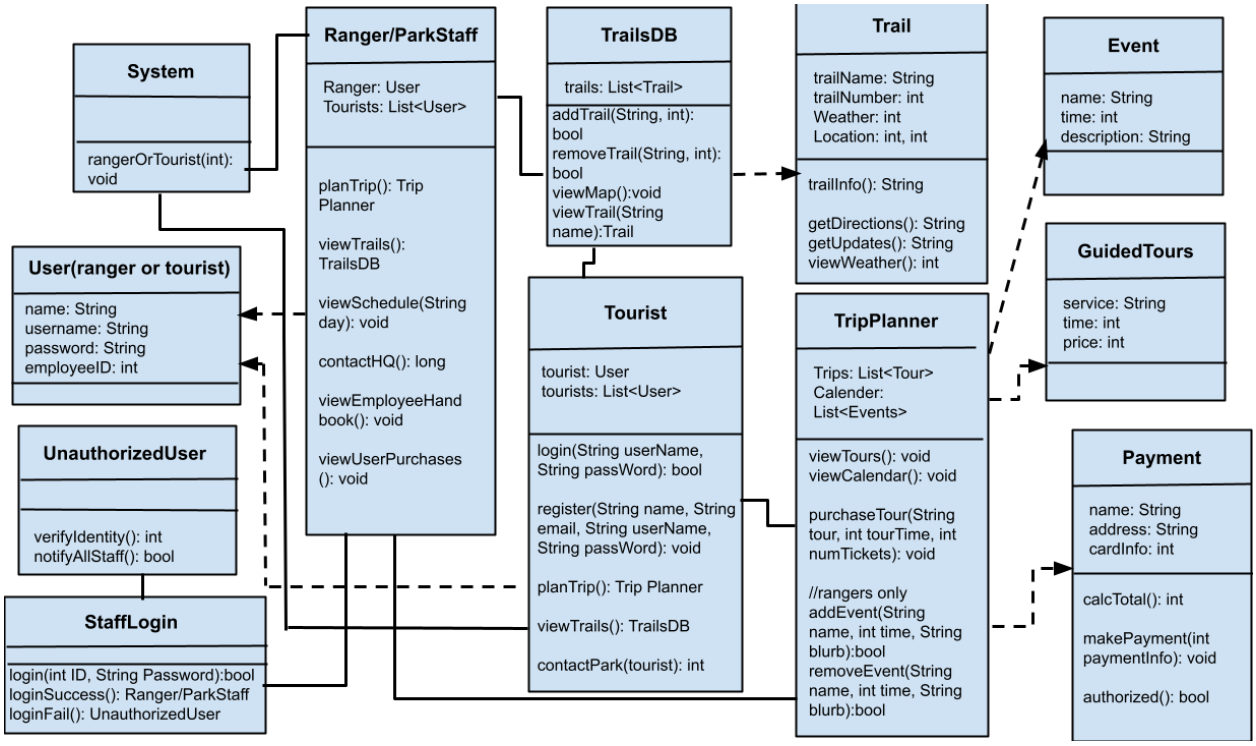
Kilimanjaro Trekker System Test Plan

Prepared by: Ella Suchikul, Sawyer Blakenship, Isabelle Bernal

March 21, 2023

1. Software Design Specification

UML Class Diagram: Kilimanjaro Trekker System



UML Class Diagram Description: The UML class diagram covers all the specific classes, attributes, and operations of the Kilimanjaro Tracking System software. The classes start off with the Kilimanjaro Trekker System login page then gives the option to go into the class of Ranger/Park Staff or Tourist/Trecker to get the best experience possible for the particular user. Within both of those classes, there are some similar classes they call to, but with different operations. This is due to rangers/park staff having the ability to write into the system, while tourists and trekkers have only read access. Those classes consist of guided tours, trails, trip planner, payment, and trails which are all necessary to meet the requirements of this particular system. Within the diagram, we have two different arrows which are an important part of how the system is

able to function. The solid black line symbolizes aggregations/associations (i.e. X has a Y). The dotted arrow line symbolizes dependencies (i.e. X uses Y). It points to a specific class that can store data for functions that have more detailed information such as the user's login information or trail specifications.

The StaffLogin class is needed as a particular login area just for the staff so they can get into the operations within the system that they either need to update or look at. We made the login operation a boolean so that the client knows whether the login was successful or not. If the user successfully logged in, they are sent to the Ranger/ParkStaff class. If the user fails to log in, they are sent to the UnauthorizedUser class.

The UnauthorizedUser class is used in conjunction with the StaffLogin class to ensure rebels aren't able to infiltrate the system and gain administrative privileges. Everytime a staff member logins unsuccessfully, the unauthorized user is called to verify the identity. The verifyIdentity method uses another camera to make sure the person is who they claim they are. The notify all staff method is used in certain cases, where the system thinks there is a possibility of the system being compromised.

The Ranger/ParkStaff class includes operations of planTrip, viewTrails, viewSchedule, contactHQ, viewUserPurchases and viewEmployeeHandbook so that the requirements needed by the staff are met and they have easy access to each of these operations. The operations return void if they will just be viewing things such as purchases and the handbook. Other operations return another class because they are interconnected. viewTrails returns the TrailsDB class, since that is where the list of trails is. planTrip returns the TripPlanner class, since that class holds all the operations for updating events. contactHQ returns a long since the phone number is 12 digits, which is larger than the range of an int. Specific attributes note mentioning are the name, employee id, and password since this is how the client will get into the system. The ranger and tourist classes both use the user class as an object to store information. There is a dotted arrow line representing aggregation because the ranger class uses the user class to store information. There is also a solid black line that points to the TrailsDB class and the Trip Planner class since these are the two pathways that the employees can choose to go to. We did not choose the dotted line symbol for this arrow because those classes do not store user information; it just provides more information about the specific class topic they chose.

The tourist class is used for the tourists to be able to access the other classes. It stores a list of all of the current users so it can verify logins by pointing the dotted line to the user class that is in charge of storing all the login information. Within the functions, in the class, it has two methods that will return Trip Planner & Trail DB so the tourists can

access the trip planning and trail viewing methods. This is always why there is a black line pointing to Trip Planner and trails DB since this is the class that holds all the information for the tourists to view. The tourist class also has a register method where it creates an account for the user if they do not have one, which returns all the registered information. The last method is named contactPark which is crucial because it returns the number of the park in case there are any issues or questions.

The TrailsDB class is used to give information in regard to what specific trails are currently available and their status. There are methods of addTrail, removeTrail, and viewMap that allow just for the specific trail to be updated by the rangers and for the tourists to be able to view their location. The add and removeTrail methods returns a boolean to verify the success of the addition and removal of the trail and viewMap is a void since it has all the information already stored in the class. Now the viewTrail method returns the Trail class which is very important because the Trail class has all the specifics about the trail. For instance, the Trail class has specific directions, weather, and allows for the park ranger to update anything they need to in regards to the trail. This is also why the TrailDB class points to the Trail class with a dotted line because all the information in regards to the trail is stored in the Trail class.

The Trip planner class is used to provide specific information about the different events/tours that the park has to offer to its visitors/tourists. The methods of viewTours, viewCalendar, and purchaseTours specifically returns a void because this is only used for the tourists to look at and nothing more. Compared to the methods of removeEvent and addEvent where they return a boolean for the purpose of rangers seeing if their changes were successful.

The Payment class is meant to process payments. Users will be able to pay for guided tours using it. The payment class needs to be able to authorize payment methods with the bank and process payments. It will be accessed by the Trip Planner class's purchaseTour method. The calcTotal method will determine the amount that the tour will cost and return it. The makePayment method will give payment info, and attempt to charge the user for their purchase. The authorized method will return whether or not the payment was successful.

2. Verification Test Plan

Test Plan 1: Ranger/Park Staff User

Unit:

Unit: TrailsDB
Feature: addTrail(String, int): bool
Test Cases:

// Test Case 1: Successfully adding a new trail
TrailsDB.viewTrail("Marangu Traverse");
TrailsDB.addTrail("Marangu Traverse", 6);
TrailsDB.viewTrail("Marangu Traverse");

Expected Output:
Null
True
Marangu Traverse

Observed Output:
Null
True
Marangu Traverse

// Test Case 2: Failed attempt at adding a new trail
TrailsDB.viewTrail("Marangu Traverse");
TrailsDB.addTrail("Marangu Traverse", 6);
TrailsDB.viewTrail("Marangu Traverse");

Expected Output:
Null
True
Marangu Traverse

Observed Output:
Null
False
Null

// Test Case 3: Trying to add a trail that already exists
TrailsDB.viewTrail("Marangu Traverse");
TrailsDB.addTrail("Marangu Traverse", 6);
TrailsDB.viewTrail("Marangu Traverse");

Expected Output:
Null
True
Marangu Traverse

Observed Output:
Marangu Traverse

False
Marangu Traverse

Unit Test Description: This test took TrailsDB as the unit, and addTrail(String, int): bool as the feature. The unit test ran several test cases on addTrail for three different scenarios: if the trail successfully added, if adding the trail failed, and if the trail already existed.

For the first unit test case, viewTrail is called first to verify that the trail has not been added yet. Null is returned, meaning the trail does not yet exist. Then, addTrail is called with the name and trail number to be added as inputs. The expected and observed output are true for the addTrail feature. Since the new trail was successfully added, true would be returned. Then to further confirm, viewTrail is called, where the trail just added would be returned, allowing access to the new trail's information in the Trail class.

For the second unit test case, viewTrail is again called first to verify that the trail has not been added yet. Null is returned, meaning that the trail does not yet exist. Then, addTrail is called with the name and trail number to be added as inputs. The expected output for the addTrail feature is true, however, the observed output is false. This indicates that the trail failed to add, which can be due to internet connectivity issues. When viewTrail is called for the trail that failed to add, null is returned since addTrail failed.

For the third unit test case, viewTrail is called first to verify that the trail has not been added yet. The trail's information is returned as available to access in the Trail class. This means that the trail is already in the database. Therefore, when addTrail is called with the name and trail number to be added as inputs, false is returned. addTrail will fail because the trail already exists, and an existing trail cannot be added again. However, calling viewTrail again afterwards will still return the trail's information since it already exists.

Integration:

Testing the connection between Ranger/ParkStaff, Trails DB, and Trail class and how these units work together.

1.Ranger/ParkStaff Class:
Ranger/ParkStaff.viewTrails();

Expected/Observed Output:
TrailsDB Class

2. TrailsDB Class:
TrailsDB.viewTrail("Marangu Traverse");

Expected/Observed Output:
Trail Class

3. Trail Class:
Trail.trailInfo();

Expected/Observed Output:
Marangu Traverse Trail Information

Length: 41.8 mi
Elevation Gain: 13,156 ft
Route Type: Out & Back
Challenge Level: Hard

Experience this 41.8-mile out-and-back trail near Kibosho Magharibi, Kilimanjaro. Generally considered a challenging route. This is a very popular area for backpacking, camping, and hiking, so you'll likely encounter other people while exploring. Note: dogs aren't allowed on this trail.

Integration Test Description: An integration test shows the interaction between units. This test took Range/ParkStaff, TrailsDB, and Trail as the units, and viewTrails(): TrailsDB viewTrail(String name): Trail, and trailInfo(): String as the features. The integration test demonstrates how these units work together.

In the (1.) Ranger/ParkStaff class, when viewTrails() is called, the TrailsDB class is accessed. This is where the complete trails list and map is located.

In the (2.) TrailsDB class, when viewTrail(String) is called, the Trail class is accessed, which is where the information on each specific trail is located. In this case, the String input was "Marangu Traverse," so all the details of this trail will be available to view.

In the (3.) Trail class, the user can then view all the details and information on individual trails after calling the previous two functions. In this test case, viewTrail("Marangu Traverse") was called. Therefore, when calling features of the Trail class such as trailInfo(), the information on the Marangu Traverse is returned (as seen in the test above).

The interactions between these different units are necessary to successfully access all the requested information.

System:

Ranger Use Case: Updating the website as an employee

Test Code:
StaffLogin.login(5791, “ilovehiking230”);
Ranger/ParkStaff.contactHQ();
Ranger/ParkStaff.planTrip();
TripPlanner.addEvent(“Smith’s Wedding”, 7, “The wedding will be held at the community center hall on September 9, 2023 from 7:00 - 11:00 pm.”);
TripPlanner.removeEvent(“Jonathan’s Birthday”, 12, “The birthday party will be held at the garden on February 12, 2023 at 12:00 pm);
Ranger/ParkStaff.viewTrails();
TrailsDB.viewTrail(“Machame Route”);
TrailsDB.addTrail(“Machame Route”, 1);
TrailsDB.viewTrail(“Machame Route”);

Output:
true // login successful
255272970404 // HQ phone number
TripPlanner class // TripPlanner class returned
true // event successfully added
true // event successfully removed
TrailsDB class // TrailsDB class returned
null // trail does not exist
true // new trail successfully added
Machame Route // viewing information on inputted trail

System Test Description: This system test shows a park employee calling multiple functions of the system. More specifically, the user will be updating the website, since it is their role to do so.

login is first called in the StaffLogin class with the employee’s ID and password as inputs. True is returned, indicating that the login to the website was successful. Because the login was successful, the user is taken to the Ranger/ParkStaff class.

contactHQ is then called which connects the employee to the Kilimanjaro Park headquarters in case of emergency or for other reasons needing contact. The HQ phone number is returned.

planTrip is called from the Ranger/ParkStaff class which gives this user access to certain functions that tourists/trekkers do not have access to such as addEvent(String name, int time, String blurb): bool and removeEvent(String name, int time, String blurb): bool. The TripPlanner class is returned.

Then, addEvent is called with the event, time, and event information as inputs. True is returned, indicating the event was successfully added to the website. Afterwards, removeEvent is called with the event, time, and event information as inputs. True is returned, indicating that the removal of the event was successful.

viewTrails() is then called from the Ranger/ParkStaff class which gives this user access to certain functions that tourists/trekkers do not have access to such as addTrail(String, int): bool and removeTrail(String, int): bool. The TrailsDB class is returned.

Similar to the unit tests, a new trail is added by calling viewTrail with a trail name as the input to verify the trail does not yet exist. Null is returned, indicating the trail is not in the database. Then, addTrail is called with the trail name and number as inputs. True is returned, indicating that the trail was successfully added. Finally, viewTrail is called again with the newly added trail name as the input. The trail name and information is returned, indicating that the trail exists and can be accessed.

Test Plan 2: Tourist/Trekker User

Unit:

Unit: Trip Planner
Feature: purchaseTour(String tour, int tourTime, int numTickets): void
Test Case: check that tour exists at given time, tour isn't full, sends to payment

//check that tours can be purchased
//add a tour
purchaseTour("Loop Tour", 1330, 3);

Expected/Observed Output
"3 tickets for Loop Tour tour at 13:30 were successfully purchased"

//check that nonexistent tour will not purchase anything
purchaseTour(String nonTour,1330,3);

Expected/Observed Output
Nothing

Unit Test Description:

The Unit is Trip Planner. The feature that is being tested is the purchase Tour method. I am testing it with various inputs to make sure it works correctly.

The first test case is simply checking that a tour can be purchased. First, the tester must add a tour with the name looptour, at 1:30 pm, and get 3 tickets. Then, to test the purchasetour method, the tester will call purchaseTour. If successful, the purchase tour will print that it was successful to the console. If nothing is printed, the tour was not purchased.

For the second test case, it is checking that it if the tour doesn't exist, it will not be purchased. It takes in the name nonTour, as a string name that no tour is named, along with a time of 1:30 and 3 tickets. If the tour was purchased, something would be printed to the console, so if the program does what it's supposed to, nothing will be printed to the console.

Integration:

Testing integration between Tourist, Trip Planner, & Payment
1.
Tourist: planTrip(): TripPlanner
//the plan trip method returns a static TripPlanner, which allows the tourist to run the trip planning methods.

Expected/Observed Output
TripPlanner TP

2.
//add tour
TripPlanner: purchaseTour("Loop Tour", 930, 1): void
//purchase that same tour

Expected/Observed Output
"1 ticket for Loop Tour tour at 9:30 was successfully purchased"

3.
Payment:
Payment: makePayment(int info)
Payment: authorized
//in the payment class make a payment
//if authorized is true, the purchaseTour was called successfully for an existing non full tour and the payment info was verified by a bank

Expected/Observed Output
true

Integration Test Description: This test shows how the Tourist, Trip Planner, & Payment classes interact with each other

1. In the tourist class, the plan trip method returns a static trip planner with which the tourist can access the trip planner's functions.

2. In the trip planner class, the purchase tour method accesses the payment class to inform it that a tour is going to be purchased.

3. In the payment class, the make payment method allows the user to make a payment. It verifies that the payment method is valid by accessing the bank. If the payment method is valid, the payment will be made, and the authorized method in the Payment class will return true

System:

Use Case 1: Tourist buying a tour
Test Code:
Tourist.login(user1,pass1)
TripPlanner TP = Tourist.planTrip()
TP.viewTours()
TP.purchaseTours("Loop Trail Tour, 1330,3)
Payment.makePayment(*credit card info*)
Print(Payment.authorized())
TP.viewCalendar()

Output:
True
*nothing
*Displays a graphic showing many tours and imposed onto a calendar
"3 tickets for Loop Trail Tour tour at 13:30 were successfully purchased"

*nothing True *Displays a graphic showing all the events and tours

System Test Description: The tourist logins first with the username and password that they have created. Then they want to access the trip planner so they call tourist's plantrip method.

The user calls the viewTours functions which displays all the tours that are coming up. The user decides which tour they want to pay for and they call the purchase tours methods to buy it.

Than the user accesses the payment class and calls the makePayment method to make a payment. The user also wants to make sure the payment was successful, so they call the authorized method. Finally, through the trip planner class, the user views the calender.

Test Plan 3: Kenyan Rebel / Unauthorized User

Unit:

Unit: StaffLogin
Feature: login(int ID, string passWord): bool
Test Case:

//Test Case 1: Successful login attempt
StaffLogin.login(5446, “stop12”);
Ranger/ParkStaff.contactHQ();

Expected Output:
True
255272970404

Observant Output:
True
255272970404

//Test Case 2: Unsuccessful login attempt
StaffLogin.login(5445, “stopp3”);
Ranger/ParkStaff.contactHQ();

Expected Output:
True
255272970404

Observant Output:
False
Null

//Test Case 3: Three failed attempts
StaffLogin.login(5445, “stopp12”);
Ranger/ParkStaff.contactHQ();
StaffLogin.login(5445, “stopp3”);
Ranger/ParkStaff.contactHQ();
StaffLogin.login(5445, “stopp10”);
Ranger/ParkStaff.contactHQ();
UnauthorizedUser.verifyIdentity();
UnauthorizedUser.notifyAllStaff();

Expected Value:
False
Null
False
Null
True
Null
False

Observant Value:
False
Null
False
Null
False
Null
255272970404
True

Unit Test Description: This test took staffLogin as the unit and login(int ID, string passWord): bool as the feature. The unit test ran different test cases to the two scenarios if the login was successful and if the login was unsuccessful.

The first unit test case, login is called to determine whether or not the information given is accurate or not. This returns a boolean of true or false if the login was successful or not. Then contactHQ is called after to determine if the user was able to gain access or not. In this case the expected/observant output is true and 255272970404 since this is a successful login attempt. This further ensures that the program is able to work properly with this specific scenario.

The second unit test case, login is called again to determine whether or not the information given is correct or not. Since the second unit test case is supposed to be an unsuccessful attempt the boolean returned is false. Then contactHQ is called to determine if the user was able to gain access or not and with this scenario it returns null. Since this is an unsuccessful attempt the observant output would be false and null but the expected value would be true and 255272970404 since realistically if you were to type in a login you are not planning on it to be wrong. This further ensures that the program is able to work properly with this specific scenario.

The third unit test case, login is called three different times since we are testing that the unauthorized user class will work as intended. The observant output for all three different times login is called is false since none of them were correct. Then we called contactHQ to make sure that when the login is incorrect it does not give access to any of the information in the ranger class so the observant output for all three times the contactHQ is called is also false. Lastly for the observant output verifyIdentity and notifyAllStaff are called where verifyIdentity gives the number to verify the user ca call to verify identity since the user failed login three times and notifyAllStaff returns true to let us know it successful notified staff of this user. The expectant output is different since after two tries the user would think the third try is correct. So the last time contactHQ and login is called it returns true with the HQ number. As well as the

notifyAllStaff and verifyIdentity where it would return false and null since the user is expecting to get the login right.

Integration:

Testing the connection between StaffLogin, Ranger/ParkStaff, and UnauthorizedUser class
1.
StaffLogin:loginSuccess()

Expected/Observed Output:
Ranger/Parkstaff class
//In the Ranger/Parkstaff class you are granted full access to view all the information about the park and allows for any updates needed to be made.
2.
StaffLogin:loginFail():

Expected/Observed Output:
UnauthorizedUser class
//In the UnauthorizedUser class, this allows the user to be given 2 more attempts to login then gives the option to verify their identity and automatically notifies staff of a possible rebel.

Integration Test Description: This test shows the interaction between the StaffLogin, Ranger/ParkStaff, and AuthorizedUser class.

1. The Staff Login class allows for the Ranger/ParkStaff class and the UnauthorizedUser class to be return depending on the success or fail of the login.
2. The Ranger/ParkStaff class can only be called by staff login which is the connection between these two classes and this is the only way to gain access to updating and changing the park system.
3. The UnauthorizedUser class can also only be called by staff login which is the connection between these two classes as well since without staff login the Unauthorized class would not be used. With the connection between these two classes this allows the staff to be made aware of potential security threats.

System:

System Test Description #1: The system is designed to prevent Kenyan rebels or other unauthorized users from gaining access and update the website with false information since this was a huge concern for the client. One system test that we could run is if there was a failed login attempt where if(login == false) then it would

automatically call the function loginFail() to allow for more attempts in the Unauthorized Class. This will allow for further options of verifying the identity and notifying all the staff right away after 2 more attempts failed. This allows for the system as a whole to flag whether or not there is potential security concern going on.

Code Test #1:

```
if(login == false){
loginFail():UnauthorizedUser
}
Main:
//something with wrong
StaffLogin.login(5446, “stoppp”);
```

System Test Description #2: Another system test that we could run is if a casual was able to log into an employee account. This is a huge deal because this will give the casual the power of an employee to remove anything or update anything they choose compromising the security of the tourists and staff. To potentially avoid this scenario there could be a security class implemented that will have two-factor authentication (2FA). In this class there would be two methods, one for the user to choose to have a text sent or a phone call verifying their information.

Code Test #2:

```
toFactorAuthentication()
{
int num =sendTextorSendCall();
bool real = checkCorrectness(num);
if(real == true)
    Security():Ranger/ParkStaff
Else if(real == false)
    Security():UnauthorizedUser
}
Main:
Ranger/ParkStaff.toFactorAuthentication();
```