Project Proposal

**Elijah Basile**

**Project Summary**

**Final Project Title: Event Manager**

**Project Description:** The purpose of this project is to construct a functioning, interactive environment for event coordinators to plan projects, event facilitators to receive such event info and use it for running the events and inviting participants, and event participants that will receive the information needed and partake in the events occurring at the occasion.

**Intended User:** Event Coordinators, Facilitators, Participants

**Reason for Project**: If an event does not have a central source of completing, reviewing, and distributing information, plans, information, and instructions can become muddled in between multiple sources causing day-of disconnects or slowing the before-day process significantly. This project aims to solve that issue by constructing a central database to hold all the necessary necessary info for an event.

**Required Technology:** This project will require a database that can hold several users, storing their data and permission accordingly.

**Case Analysis**

**Note: the basic initial UI is a basic listing of events planned by you, and events that you are a member of in some capacity, below lists the information presented once your username and password has been entered and you have entered that event**

**Event Coordinator:** On the GUI, they will need several necessary tools

1. **Edit Event Information**
2. **Invite Users**

**Event Facilitator/Volunteer:** On the UI, they have limited access to the particular event

* **They can only Invite Users**

**Event Participant:** The most limited group member, may or may not require permission to view event in question

* **All members including the participants can view the Event data**

**No Permission:** Any user not listed on the event’s permission listing

* **No access**

**Data Design**

What data does program deal with? **Coordination Information Necessary to Display Events**

* **Event Creation**
* **Event Information**
* **Invitations to Users/ Permission Adjusting**
* **Address of Event**
* **Date of Event**
* **Necessary Contact Info (could be phone, email ... both!)**

What is the best way to represent the data? **Through a combination of a UI, several linked lists and associated files necessary**

Will the data need to be persistent? **Yes.** How will you make that happen? **Using object serialization for a linked list of Users and Events and a file listing the user names already taken.**

Will the data need to be aggregated into a larger structure? How? **The Data will be aggregated per event, per user, but other than that basic hierarchy (as listed earlier) the aggregation of data will be rather intuitive as looking at program tree**

**UI Design**

The UI will be through the command line interface.

The option will be presented as follows:

**Login:**

Welcome to EventManager. Please login or Sign-up.

1: Login.

2. Sign-up

**Menu:**

Choose from the following options.

1: Create a new Event

2: Edit an existing event

3: Invite another user to one of your events

4: view your events

5: delete an event

6: Quit

**Algorithm (in relative object hierarchy)**

**Users**

**Initializer will be prompt the user to enter his username/password and if a new combination, will create a new User in the User LinkedList in Users. If username already in the database (will search usernames and passwords in the database) and the password is false, block entrance. Variables will be private instances (we don’t want anyone viewing them other than the specific user)**

**LinkedList<User> userRoot;**

**LinkedList<User> userRoot;**

**Int c\_uid**

**-**current user id, don’t have to pass in any id numbers throughout program run

**LLNode findEvent(int id)**

**Goals: find and return an event in the root**

**Input: the id of the Event**

**Output: the Event in question**

**Steps:**

1. **Set the current to head**
2. **If the list is empy, return null**
3. **While there are still items in the list, search for the event with the same as the id selected**
4. **If list runs to the end without success, return null**

**LLNode findUser(String \_username)**

**Goals: to find the user in the list**

**Input: username of user in question**

**Output: the User in question**

**Steps:**

1. **Set the current to the root of the list**
2. **If list empty, return null**
3. **While the list isn’t empty, run through each username, checking to see if they match with input**
4. **Return if found**
5. **Return null if you run to the end of the list**

**saveEvents()/saveUsers()**

**Goals: to the save the roots of the Database**

**Input: file name**

**Steps:**

1. **Serialize the LinkedList objects**
2. **Write objects to the file name inputted**

**loadEvents()/loadUsers()**

**Goals: to load the Users and Events from previous sessions**

**Input: file name**

**Steps:**

1. **Deserialize the Users/Events from the file name entered**
2. **Set the eventRoot to the object read in**

**User login()**

**Goals: to allow a new user to sign up or an existing user to log in and access their events**

**Output: User**

**Steps:**

1. **Welcome user**
2. **User can choose to login or sign up**
3. **login**
   1. **user enters a username**
   2. **the username is found in database**
   3. **user enters correct password to receive access to account**
   4. **the current user id is set to the User node’s id**
4. **sign up**
   1. **new user object created**
   2. **User object inserted into the userRoot**
   3. **Current user id set to the head (as the head is the new user)**
5. **Returns user**

**listEvents()/listEventsByPermission()**

**Goals: to list the events relevant to the User**

**Steps:**

1. **Go to the top of the Event root**
2. **Run down the list, checking if the user has permission to access them**
3. **If user is either not denied (listEvents) or has that permission type (byPermission), print our Event info, including the Event ID**

**createEvent()**

**Goals: to create a new Event**

**Steps:**

1. **Create a new Event, adding the coordinator status to the user**
2. **Insert the Event in the eventRoot**

**editEvent()**

**Goals: to edit a chosen event**

**Steps:**

1. **User chooses which event they would like to edit**
2. **User edits Event accordingly**

**deleteEvent()**

**Goals: to delete an event in the list**

**Steps:**

1. **User chooses event by id**
2. **Node is deleted**

**inviteToEvent()**

**Goals: to grant another user a permission level for an event**

**Steps:**

1. **Lists the events that you can invite people to**
2. **Select event by entering Event ID**
3. **Set the user id in that event to yours**
4. **Checks for the permission level of the accessor**
5. **If they are of coordinator or facilitator permission level, let them pass**
6. **Facilitators can invite more facilitators and more participants**
7. **Coordinators can invite any type**
8. **Sets the info String to an invite receipt**
9. **A set new invite is sent to the user node in question**

**viewInvites(String \_username)**

**Goals: for the User to view their invites to events**

**Steps:**

1. **finds the user with inputted username**
2. **displays the invites**

**int printMenu()**

**Goals: to print the general UI for logged in user on event manager**

**Output: an integer displaying the user’s choice**

**Steps:**

1. **displays the following message**
   1. Choose from the following options.
   2. 1: Create a new Event
   3. 2: Edit an existing event
   4. 3: Invite another user to one of your events
   5. 4: view your events
   6. 5: delete an event
   7. 6: Quit
2. **User inputs choice**

**Main()**

**Goals: to handle the skeletal structure of the program**

**Steps:**

1. **Loads the events**
2. **Loads the users**
3. **Initializes a loop that loops until user termination**
4. **Gets choice from printMenu and User selection moves program forward**
5. **User can choose to exit the program after logging out.**
6. **Loop terminates**
7. **Save Users & Events**

**User**

**Handles the user controls and user object**

**String password**

**String username**

**Contact contact**

**String inviteInfo = “”;**

**Constructor User()**

**Goals: to set the username, password of user**

**Steps:**

1. **User enters username that is tested to be unique**
2. **User enters password at least 8 characters long**
3. **New username written to file of all used usernames**

**Boolean checkPasswordLength()**

**Goals: make sure the password is 8 characters long at least**

**Output: true or false**

**Steps:**

1. **Checks length of password**
2. **If longer than or equal to 8, return true**
3. **If shorter, return false**

**Boolean checkUniqueUsername()**

**Goals: check if username is unique**

**Output: true or false**

**Steps:**

1. **Opens usernameList.txt**
2. **Sees if the username matches any in the List**
3. **If so, return false**
4. **If not, return true**

**newInvite(String info)**

**Steps: append new string to inviteInfo**

**viewInvites()**

**Steps: prints out the inviteInfo**

**setContact(String phone, String email)**

**Steps: sets the contact phone to string phone and contact email to string email**

**setPassword(String \_password)**

**Steps: set the password**

**setUsername(String \_username)**

**Steps: sets the username**

**String getPassword()/getUsername()**

**Steps: return password or username respectively**

**Event**

**Will be constructed once a user decides to create one, they will automatically be granted a new coordinator object associated with the event so they have full access**

**Date date**

**Address address**

**Contact contact**

**String name**

**int uid**

**LinkedList<Integer> coordinators, facilitators, participants**

**Constructor Event(int \_uid)**

**Goals: to fully initialize the Event object and all its elements and set access to the user who made it**

**Input: user id (\_uid)**

**Steps:**

1. **Sets the current user to the coordinator**
2. **Sets the name**
3. **Sets the address**
4. **Sets the date**
5. **Sets the contact**

**setUId (int \_uid)**

**Steps: sets the uid to the inputted integer**

**PL checkForPermission()**

**Goals: to check to see if the user can access that element**

**Output: Permission Level => PL**

**Steps:**

1. **Search for the user id in the coordinators**
2. **Search for the user id in the Facilitators**
3. **Search for the user if Participants**
4. **If none of them were found** 
   1. **Return the Permission Level Denied**

**printInfo()**

**Goals: print the Event info < mainly a test function so I won’t dwell**

**String getDate()/getContact()/getAddress()**

**Goals: return formatted object**

**Output: string**

**Steps:**

1. **If user have any permission level**
   1. **Returns the formatted date/contact/address**

**setDate()/setContact()/setAddress()**

**Goals: to change or set the date, contact, address objects belonging to the event**

**Steps:**

1. **Each object element is entered in by user after user was prompted**
2. **Set the object element to entered value**

**setName()/getName()**

**Goals: to set and return the name values of the Event**

**Steps:**

1. **setName sets the name to the user input**
2. **getName returns the string**

**addCoordinator()/addFacilitator()/addParticipant()**

**Goals: to add the user id to the permission listing**

**Input: user id that will be added \_uid**

**Steps:**

1. **inserts the new int \_uid to the linkedlisted items**

**Main**

**Goals: to test the object <- no further explanation needed**

**LinkedList<T>**

**LLNode<T> head;**

**int id;**

**search(T value)**

**Goals: checks for the existence of an item in a list**

**Input: the value that matches the value of your LinkedList**

**Steps:**

1. **if the list is empty, return false**
2. **run through the list until item found (return true) or list reaches end**
3. **if never found, return false**

**LLNode<T> getNode(int uid)**

**Goals: get the node in the list**

**Steps:**

1. **set the current to the head**
2. **run through list until node found (by matching id and uid combo)**
3. **return null otherwise**

**deleteNode(LLNode<T> toBeDeleted)**

**Goals: delete a node in the linkedList**

**Input: node to be deleted**

**Steps:**

1. **if node to be deleted is the head, set the new to the element after it**
2. **otherwise**
   1. **set the before to the before of the deleted**
   2. **set the after to the deleted after**
   3. **set the node before toBeDeleted to point to after**
   4. **set the node after toBeDeleted to point behind it to before**

**LLNode<T> getHead()**

**Steps: return the head of the list**

**Insert(LLNode<T> inserting)**

**Goals: to insert a new element onto the list**

**Input: inserted linkedlist item**

**Steps:**

1. **set the inserting’s after to head**
2. **if the head is not null**
3. **set the head’s before to inserting**
4. **set the id of inserting**
5. **set the head to inserting**
6. **increment the id counter**

**LLNode<T>**

**T value**

**LLNode<T> before**

**LLNode<T> after**

**int id**

**Constructor LLNode(T value)**

**Steps: sets the value to the value in the parentheses**

**setId(int \_id)**

**Steps: sets the id**

**setAfter/setBefore(LLNode<T> \_after/\_before)**

**Steps: sets the nodal reference to after/before**

**getAfter()/getBefore()**

**Steps: return the after and before LLNodes**

**getValue()**

**Steps: returns value contained in the Node**

**getId()**

**Steps: returns the id value**

**Address**

**Contains getters and setters for the following variables**

**String street**

**String apartment**

**String city**

**String state**

**String zipCode**

**Date**

**Contains getters and setters for the following variables**

**int day**

**int month**

**int year**

**Contact**

**Contains getters and setters for the following variables**

**String phone**

**String email**