



SPO AUTO-LOG

Auto logging application: release 1

Abstract:

While working as a student patrol officer at Elizabethtown College, I was inspired to take a new course and attempt to use the knowledge of Computer Science and programming to automate the process of logging and documenting data. This is my Senior project.

Jamal D. Scott
jamaldscott@gmail.com

Proposal:**Aim:**

The purpose of this project is to develop software to be used in house at Elizabethtown College for the Campus Security Department, specifically for the student patrol officers (SPO). This software shall replace traditional means of student information logging which requires students to carry around a log sheet to fill in various forms of information such as current location, times, and the paths that the patrol officers have taken to their destinations.

This software shall:

- Support administrated user registration:
 - The current student patrol coordinator (SPC) will be able to maintain and manage SPO accounts. This entitles them to add or remove SPOs based on employment. Users who are not a part of the employment database will not be able to register or submit logs to keep the integrity of the logs safe (people unemployed shall not be able to access any information.)
 - Eligible accounts will be listed in an encrypted and be compared with users registering. If a user's information does not appear in the file, then they cannot register. This prevents unemployed users from registering.
- Support user registration:
 - Currently employed SPOs will be able to register for an account to use the app. This will make appending information to the log easier.
- Support user troubleshooting:
 - Given that a user may forget their log in information, the software will support user account recovery through means or recovery questions and resetting the password.
- Secure all information:
 - All passwords received will be hashed through SHA-256 hashing functions so that the passwords cannot be read by 3rd parties.
- Generate logs:
 - The logs will offer options to record all equipment used, places visited, paths taken, and areas checked by the SPOs. The app will then format the log, and save is as a file on the system to be printed out for documentation later.






Logging algorithm:

- The main function of this software is to log information. This will NOT utilize GPS APIs (application programming interface) as through trials of GPS paired with WiFi/Data, battery usage, and application execution and processing, the phone's battery life is not expected to last the length of a shift which is typically 4 hours; 5 on weekends.
 - Processes with GUI (graphical user interface) screen displays that are in constant use tend to greatly consume power, thus reducing phones battery charge significantly. Pairing that with the necessity of GPS (which already consumes a decent amount of power) and the possible necessity of the device needing to disconnect and reconnect from WiFi to mobile data and back (depending on where the user is on campus; WiFi does not reach all corners of the campus) battery consumption will be too high. If the phone does not last then the log will not be generated correctly.

- To combat the issue of battery consumption, Dijkstra's algorithm will be used to mathematically calculate the path that the users have taken from node to node (location to location).
- Each major and minor landmark or location on campus will be mapped to a node type and placed into a graph where with each transfer from node to node, Dijkstra's algorithm will be executed to find the path that the user took. It will note all landmarks that the user passed and log them just as a user would on the traditional logs.



File Contents:

Within the SPO Log generator folder, there will be several files and folders. These files cannot be altered, renamed, or removed as the program will not function properly without them. The contents of the main folder should look like this:

Name	Date modified	Type	Size
 Log_lib	4/27/2017 8:38 PM	File folder	
 UserLogs	4/27/2017 8:38 PM	File folder	
 Users	4/27/2017 8:41 PM	File folder	
 VerifiedUsers	4/27/2017 8:41 PM	File folder	
 Log	4/27/2017 8:38 PM	Executable Jar File	45 KB

In the order of which the elements are depicted in the image, the folder:

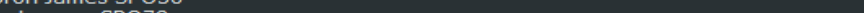
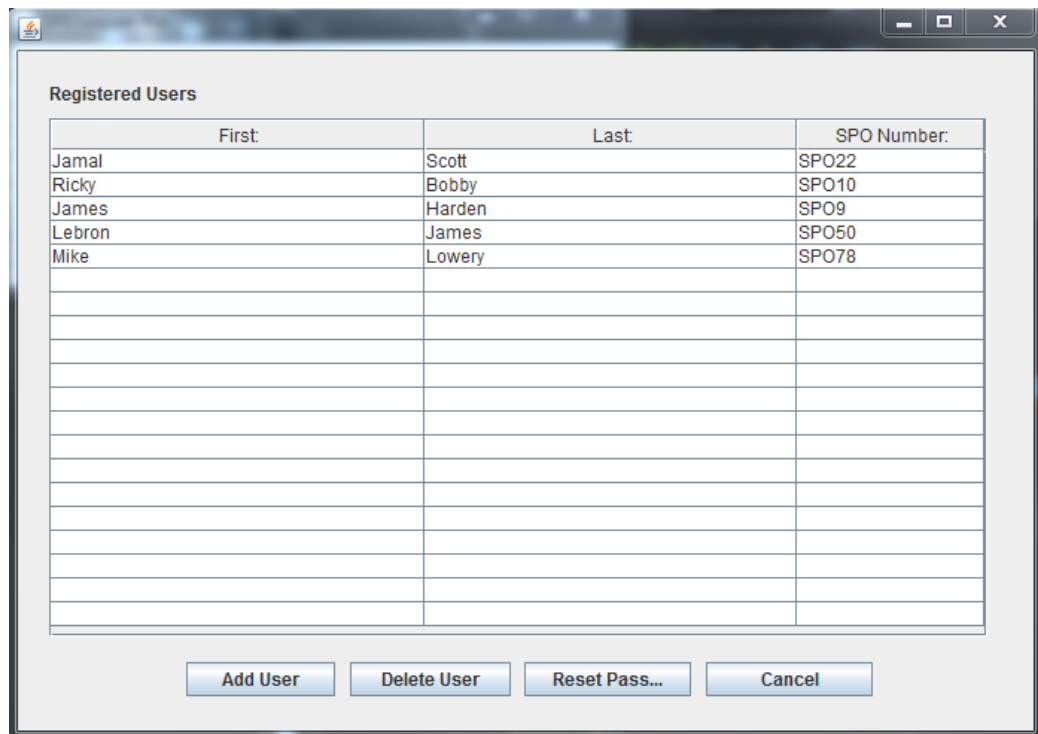
- **“Log lib”:**
 - This is where all of the included source code libraries will be place (if any exist.) This file must remain unaltered as when the source code was compiled into the .JAR file, the compiler specified that this folder is where libraries will be built; the program will check for this directory and fail to function normally if it cannot find this folder.
- **“UserLogs” folder:**
 - When a SPO completes their log, they will select the option to export the log for later use. When that occurs, the log is then drawn into an image file and the program will look for this directory to place it in; if the directory is not found due to it being altered or removed, the program will cease to function.
- **Users:**
 - This folder is where the generated user account files will be placed.

 admin.SPOFile	4/27/2017 8:39 PM	SPOFILE File	1 KB
 scottjd.SPOFile	4/27/2017 8:41 PM	SPOFILE File	1 KB

 - When registering, or logging in, the program will promptly search for this directory and search for the SPOFILE to read from. The generated SPOFILE contains the user created user, password, name, SPO number, security question, and security question answer.
 - **(For readability, the passwords were not hashed to show that elements are being place)**

```
scottjd.SPOFile
1 scottjd pizza Jamal Scott SPO22 0 r
2
```

- Username: scottj
- Password: pizza
- First Name: Jamal
- Last Name: Scott
- SPO Number: (String Append) “SPO”+22
- Associated Security Question Integer: 0 (corresponds to the first security question in an array of questions.)
- Associated Security Question Answer: r (The answer to the selected security question.)
- **VerifiedUsers:**
 - This folder will house the “Validated Users.vuf (Validated User *F*ile) which contains a list of all acceptable users.
 - This file must be created by the admin by logging in as an admin and adding users to the list.



A screenshot of a Notepad window titled "Validated Users.vuf". The window contains a list of five validated users, each on a new line and preceded by a line number (1 through 5). The text is as follows:

```

1 Jamal Scott SPO22
2 Ricky Bobby SPO10
3 James Harden SPO9
4 LeBron James SPO50
5 Mike Lowery SPO78








```

- The contents of the file are as depicted:
- First Name, Last Name, SPO Number.

- Without appearing on the list, a user cannot register. This prevents unwanted users on the application.

- **Log Generator.JAR:**

- This file is the executable JAR file that will compile and run the source code. It will serve as the “startup button” to the graphical user interface (GUI.)
- The JAR relies on several source code files:

 adminPage.java	4/27/2017 11:41 AM	JAVA File	6 KB
 DataAdder.java	3/29/2017 2:30 PM	JAVA File	7 KB
 FrontEnd.java	4/18/2017 10:54 AM	JAVA File	10 KB
 Log.java	4/27/2017 8:56 PM	JAVA File	7 KB
 RegisterWindow.java	4/27/2017 11:36 AM	JAVA File	10 KB
 SHA256.java	9/26/2016 6:30 PM	JAVA File	5 KB
 ShortestPath.java	4/27/2017 11:30 AM	JAVA File	6 KB

- **adminPage.java**
 - This page is only brought up when an admin logs in, it allows the admin to allow users to register, delete users, etc..
- **DataAdder.java** (back-end code)
 - This source code file will append and display all of the entry data that a user enters such as, the date, who they're working with, and which equipment is checked out.
- **FrontEnd.java**
 - This pulls up the main user interface and handles cases such as logging in, interfacing into the registration API, checking log in credentials, and password recovery.
- **Log.java**
 - This is where the log application window will be displayed, it incorporates a fully interactive GUI that allows a user to select locations, see the generated path, and export their log.
- **RegisterWindow.java**
 - This source code will handle registration cases such as creating the SPOFILE's and checking to see if an account already exists.
- **SHA256.java** (back-end code)
 - This is the SHA256 source code for generate a 256-bit hash code for user passwords.
- **ShortestPath.java** (back-end code)
 - This source code file utilizes Dijkstra's algorithm of finding the shortest path between nodes in a graph.

Application Uses:

To start the program and bring up the GUI, execute the runnable JAR file: Log Generator.JAR, this should begin the program. If your system does not allow execution by double clicking the JAR, you must run the JAR from the command line.

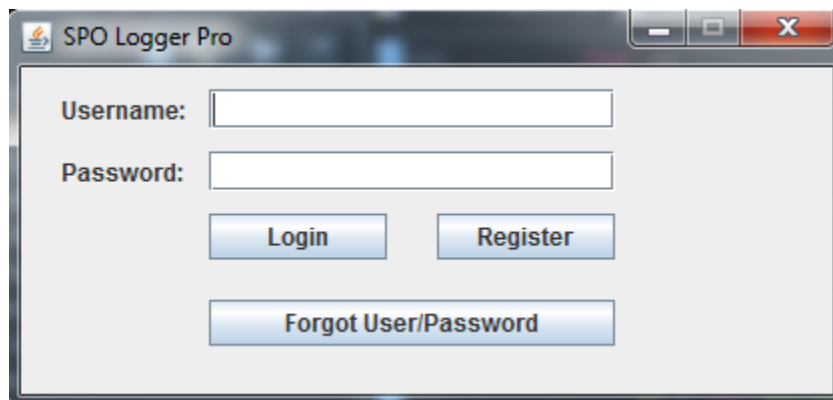
Windows:

Start -> (Search) "Command line" -> CD\ C:*path to jar*
 Java -jar Log Generator.java

Linux:

Open Terminal \$> java *filename*

Upon execution, the program will then bring up the main GUI:



To get started for first time use, we need to set up a student admin account to begin adding the names of acceptable users. Click on the "Register" button on the main GUI screen, this will pull up the registration window. An important thing to note is that once an admin account is created, another user cannot make another admin account because the SPO number 1 will already be taken as well as the username "admin." Those are the precursors to allow the Log Generator to pull up the admin page. Also note that the admin will have to add their own personal account for use to so that they can utilize the logging features if they are working as well.

To create an admin account enter the following credentials:

- First: Student
- Last: Administrator
- Username: admin
- Password: *select a password*
- SPO Number: 1
- Security Question: *select a question*
- Answer: *provide an answer*

Remember these details as they will be used each time you log into the system as an admin. If all of the provided information was acceptable, you will be notified of your successful account creation.

After registering, you will be redirected back to the log in GUI where you can then enter your credentials. If entered correctly, you will be notified of your successful login, and

then the administration page will be displayed. The page is simple; when using the application for the first time, the table should be empty. Acceptable users will be displayed in this section when adding users. At the bottom of the page exist several buttons for utilization. For now, we will only focus on adding new users. Select the option to add a new user and follow the prompts. Remember the information that you enter as it will be referenced later.

Upon completion of the prompts, the user's first name, last name, and SPO number should be displayed in the JTable. We have now enabled ourselves to add new users. As a back-end function of the application, you can allow any user to use the application. To do so, navigate to the internal directory \VerifiedUsers\ValidatedUsers.vuf, this file can be opened in any text editor (for readability, all contents of the file will be displayed in plain text for this example.)

At the top of the top of the page, add the tag "#Open-Beta/ALL" as so:

```
1 #Open-Beta/ALL
2 Jamal Scott SPO22
3
```

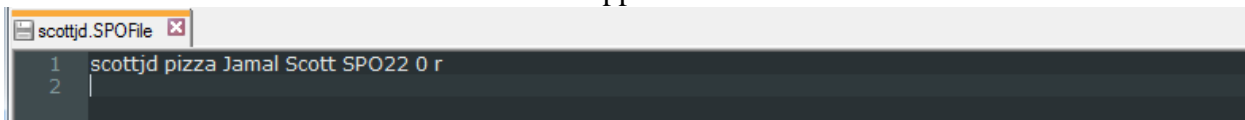
When a user attempts to register, the program will read from this file to verify that they are on the list. If the program comes across the tag "#Open-Beta/ALL", it will tell the calling function that all users are acceptable by always returning true until the tag is manually removed again.

```
if(account.contains("#Open-Beta/ALL"))
    return true;
```

For registration testing (after testing user registration through administration) it would be very useful to add this tag to speed up the process; note users created after this tag was added will not appear within the VUF file, or the administration table. If the desired results are achieved, refer back to the new test user that we created. Proceed to log in the application with their credentials. If you forgot the credentials, you may always refer back to the user files located: \Users*YOUR ACCOUNT*.SPOFILE

(Again, note that nothing in the test application is encrypted or hashed for readability. If you wish to see the end product, please refer to the "Log-Secure" application.)

The information in the SPOFILE will appear as such:



```
1 scottjd pizza Jamal Scott SPO22 0 r
2
```

Username	Password	First name	Last name	Spo #	Sec "?"	Sec answer
----------	----------	------------	-----------	-------	---------	------------

Alternatively, you may use the "Forgot Password" feature, and then follow the prompts. The application will pull this information from the files for you and display it on screen in plaintext.

If all information was successfully recovered and logging in was successful, the following GUI display should be present:

[illegible]

On the purpose of this display screen, is to document any additional information regarding the SPO officer's materials used, as well as their partner and their materials. These fields are optional. Any enter text will be formatted and displayed in the "Log Preview" text display area. Omitted data will still be place as empty strings in the template. If a user is required to add or change any of the data submitted, the user may edit what is shown in the text box.

Upon submitting or omitting the log info, the user will be brought to the main application page. On this page, similar to the physical logs that the student patrol officers would carry, the log consists of two rows of location logging slots containing 12 possible slots per entry. To the right of the location selection fields is the actual what will make up the actual log; the log information and the path and check table. Depicted below is what the main application's appearance is.

On startup, all of the area selection boxes except for the first one will be set as inactive, this will keep consistency in location selection as the entries on the log table appear in the order that the will based on which area selection box has been toggled. When one selection box has been used, it will be set to inactive and the next box in descending order will activate until the end of the column is reached. The selected information will appear in the log table to the right.

The paths are generated using Dijkstra's algorithm for determining the shortest path between two points in a graph. The technicality of the code will be explained after this tutorial. If a user has arrived to a location and wishes to perform a check on the location, they can simply reselect the location that they are at in the next available location selection box and click the "Check" button at the top of the application. This action will change the current table entry into the location + "check" which signifies an area check. Along with the path generation, the application will also log the times of the check for the student patrol officer. Below is an example of the log in use.

Log v1.0

Check **Generate PDF**

Locations

Schlosser Brown

Myer Solar

Ober Solar

Brinser Quads

Founders Quads

Disc

Disc

Library

Library

BSC

BSC

Log Preview:

SPO Information: 05/11/2017
 Jamal Scott
 SPO22
 In-time: 22:05

Partner Name:
 Partner SPO Number:
 In-time:

Jamal Scott's equipment:
 Keys: Radio: Flashlight:

Partner equipment:
 Keys: Radio: Flashlight:

Patrol Assignment:

From:	To:	Activity:
X	22:05	Office
22:05	23:06	Office -> ResLife -> Royer -> Schlosser
23:06	23:06	Schlosser -> Myer
23:06	23:06	Myer -> Ober
23:06	23:06	Ober -> Brinser
23:06	23:07	Brinser -> Founders
23:07	23:07	Founders -> Disc
23:07	23:07	Disc check
23:07	23:07	Disc -> CHL -> Ober -> Library
23:07	23:07	Library check
23:07	23:07	Library -> BSC
23:07	23:07	BSC check
23:07	23:07	BSC -> AQuads -> LOT4 -> Lake -> Brown
23:07	23:07	Brown check
23:07	23:08	Brown -> Lake -> YC -> Solar
23:08	23:08	Solar check
23:08	23:08	Solar -> Track -> Quads
23:08	23:08	Quads check

When the student patrol officer's shift has been completed, they can select the option to "Generate" the log. The application will pull up a preview of the log, and a confirmation as such:

Log Preview:

SPO Information: 05/11/2017
 Jamal Scott
 SPO22
 In-time: 22:05

Partner Name:
 Partner SPO Number:
 In-time:

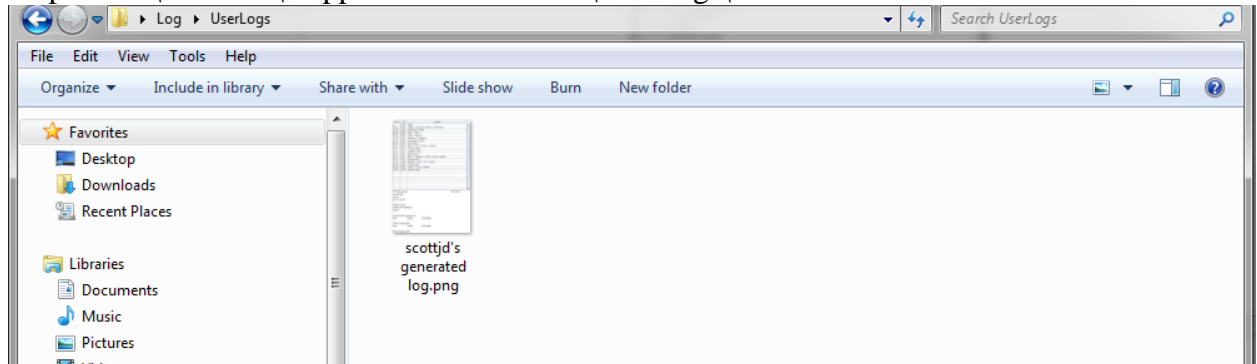
Jamal Scott's equipment:
 Keys: Radio: Flashlight:

Partner equipment:
 Keys: Radio: Flashlight:

Patrol Assignment:

OK

If satisfied with the log, the can then proceed with the publishing of the log. Their log will be output to C:*USER**Application Location*\UserLogs\



Utility & Purpose Redefined:

As mentioned in the abstract, the purpose of this program was to create centralization of data and make the logging process for the student patrol officers easier and scalable. Previously, students were required to rely on paper logs which could be lost or torn as they were working. Handwriting and methodology of logging was also different among the student patrol officers which resulted in each SPO essentially writing and completing their logs in different ways. This created a lack of consistency in the log writing which would then be passed onto the new SPOs in training, thus confusing the new trainees and resulting in new adaptation of incorrect documentation. With an application, all users will be required to use the application the same way and the paths taken to locations will be correct as opposed to speculation given that users fill uphold integrity and honesty by logging where they actually are.

In addition to centralization and consistency of data, the elimination of paper use is a positive outlook on this application as saving paper and saving trees from destruction in the environment is always a plus. Initially in release 1, the logs were to be formatted and emailed to the users to be printed and submitted into the collection bin at the office. This functionality was removed as it proved to be meaningless to print the documents which were analyzed by the coordinator, logged into a computer than shredded. Rather, the documents can be saved on a device, digitally transferred to the coordinator where they can analyze the documents digitally and dispose of them digitally.

Functionality and Source Code Explained:

In this section, we will review the major (not all) pieces of the code that give the app it's functionality, explain how they work, and list all of the source code:

FrontEnd.java:

Primarily, we'll start by defining all of the variables that we will need and use within the application:

```
public class FrontEnd
{
    private JFrame frame;
    public static String username = "";
    private String password = "";
    private String firstName;
    private String lastName;
    private String ModSPONumber;
    private String SAnswer;
    private JTextField userNameField;
    private JPasswordField passwordField;
    private String[] securityQuestions = {"What was your first pet's name?", "What city were you born in?", "What is your favorite color?"},

    /*SPO Information contains the information from the .SOFiles, it is public so we can use across
    *multiple classes without having to make new arrays.
    */
    public static String[] SPOInformation = new String[7];
    public static File[] listOfFiles;
    private boolean loggingIn = false;
}
```

For simplification, all of the strings that are private variables will be the information that we will only be handling in this class file, these are variables to hold the data input by the users for logging in, and password recovery. The public variables, the array of strings more importantly, are to hold information for computation heavy methods that we only need to compute once, such as reading in data from a specific file and storing it for the duration of that user's application use. The string array SPOInformation will hold all of the user data from the file as described earlier in the tutorial.

When the application is booted up, the JFrame will be created and populated with all of its button elements. If the user decides to select the button to log in, the program will execute this code:

```
/* Stores the password and user name entered into their respective variables.
 * passwordField.getText(); (scratched) hides the password.
 */
SecureHash256 SHA256 = new SecureHash256();
username = userNameField.getText();
password = SHA256.Hash(passwordField.getText());
/*
 * Lets the program know that the user is logging in, this helps the program
 * to determine what functions to use in terms of reading in.
 */
loggingIn = true;

/*
 * Calls the methods to determine if the user exists and that the
 * given password matches the user name. If both conditions are true
 * execute statements inside the 'if'.
 */
if(CheckUserNamePW() && ReadIn())
{
    /*
     * Close the main window application to open up another
     * Tell the program that we're not logging in anymore.
     * Open up the log window.
     */

    if(username.equals("admin"))
    {
        loggingIn = false;
        adminPage adminP = new adminPage();
        adminP.administration();
    }
    else
    {
        frame.dispose();
        loggingIn = false;
        Log log = new Log();
        log.log();
    }
}
```

This snippet of code will take what the user has input into the text fields and store them in variables. In the case of the secure application, the password will run through a hashing algorithm as passwords are never stored in plaintext on a system or database. In the parameters of the “If” statement, two Boolean functions are called that will determine if the log in was successful. By utilizing the “AND” logical operation, both Boolean functions must return “true” in order to log in successfully. If the inputted username was “admin” and the log in credentials were correct, the admin will be taken to a different application screen as described in the tutorial, otherwise, the application will redirect the user to the log page.

The purpose of the function “CheckUserNamePw()” served as protective error checking so that the system may not crash, the contents of the function are rather trivial:

```
private boolean CheckUserNamePw()
{
    if(username == null || username.contains(" ") || username.isEmpty())
    {
        JOptionPane.showMessageDialog(frame, "Invalid Entry on 'username!'");
        return false;
    }
    if(password == null || password.contains(" ") || password.isEmpty())
    {
        JOptionPane.showMessageDialog(frame, "Invalid Entry on 'password!'");
        return false;
    }
    return true;
}
```

We are simply checking to see if the user enters invalid data, or any data at all. If any negative condition is met, the function returns “false” and notifies the user of their mistake. If everything was acceptable, the function returns “true” which yields one successful requirement to logging in.

Next there is the ReadIn() function:

```
private boolean ReadIn() throws IOException
{
    SecureHash256 SHA256 = new SecureHash256();

    BufferedReader in;
    char c;
    String fileContents = "";

    //Reads in all of the names of files in the directory
    File folder = new File(System.getProperty("user.dir")+"\\Users\\");
    ListOfFiles = folder.listFiles();

    //Loops through the list of the file names
    for (int i = 0; i < ListOfFiles.length; i++)
    {
        //Opens each file and puts the file contents into a string
        in = new BufferedReader(new FileReader(ListOfFiles[i]));
        while(( c = (char)in.read() ) != (char)-1 )
        {
            fileContents += c;
        }
        //Sees if the string of file contents contains the information we're looking for
        //If it does contain the information, then it is the file/ account that we need to recover.
        if(loggingIn && fileContents.contains(username))
        {
            PopulateInformation(fileContents);

            if(SPOInformation[0].equals(username) && SPOInformation[1].equals(password))
            {
                JOptionPane.showMessageDialog(frame, "Log in successful.");
                loggingIn = false;
                return true;
            }
            else
            {
                SPOInformation = new String[7];
            }
        }
        else if(!(loggingIn) && fileContents.contains(firstName) && fileContents.contains(lastName) && fileContents.contains(ModSPONumber))
        {
            PopulateInformation(fileContents);
            //Confirms that this is the file that we need to recover and all contents are filled in.
            return true;
        }
        else
        {
            //Resets the contents in case we didn't find the file.
            fileContents = "";
        }
    }
    return false;
}
```

The objective of this function is to select the correct user .SPOFILE from the working directory. To do this, we must manually look through the list of files to see if the username's file exists. If it does, we must read the data from that file and store it. This is achieved through making another function call within the "ReadIn()" function.

```
if(loggingIn&&fileContents.contains(username))
{
    PopulateInformation(fileContents);
}
```

```
private void PopulateInformation(String fileContents)
{
    char currentChar;
    String currentWord = "";
    int j = 0;

    while(j < SPOInformation.length)
    {
        //Separator the contents of the file (which is stored in a string) into their respective array cells
        //Arr[0] = username Arr[1] = password Arr[2] Firstname Arr[3] = Lastname.....
        for(int k = 0; k < fileContents.length(); k++)
        {
            currentChar = fileContents.charAt(k);
            currentWord += currentChar;
            if(currentChar == ' ' || currentChar == '\n')
            {
                SPOInformation[j] = currentWord.replaceAll("\\s", "");
                j++;
                currentWord = "";
            }
        }
    }
}
```

In this function, the contents of the correct file are passed over and parsed into individual tokens which have been delimited by white space. These tokens are then stored into the "SPOInformation" string array. Back tracking to the "ReadIn()" function, we will then compare the user input to the contents of the file to see if the inputted username and hashed password match the file's contents. If it does, we can then tell the main function that we are not attempting to log in anymore and return "true." If the information was incorrect, the user will be notified and the array will be cleared via "SPOInformation = new String[7];" (Writing a new, clear, array over the old one in memory

```
if(SPOInformation[0].equals(username)&&SPOInformation[1].equals(password))
{
    JOptionPane.showMessageDialog(frame, "Log in successful.");
    loggingIn = false;
    return true;
}
else
{
    SPOInformation = new String[7];
}
```

Also, an important thing to note is that this function is used two ways, one for logging in and one for password recovery. We tell the function which we are doing through the Boolean variable "loginIn." The "ReadIn()" function will know when we are attempting password recovery if the variable "loggingIn" is set to "false."

```

else if(!(loggingIn)&&fileContents.contains(firstName)&&fileContents.contains(lastName)&&fileContents.contains(ModSPONumber))
{
    PopulateInformation(fileContents);
    //Confirms that this is the file that we need to recover and all contents are filled in.
    return true;
}

```

If the user opts to recover a password, the previous steps are taken except, error checking is implemented on the user prompts:

```

public void actionPerformed(ActionEvent arg0)
{
    firstName = JOptionPane.showInputDialog(frame, "What is your first name?");
    lastName = JOptionPane.showInputDialog(frame, "What is your last name?");
    ModSPONumber = JOptionPane.showInputDialog(frame, "What is your SPO number?");

    if(CheckInput())
    {
        try
        {
            if(ReadIn())
            {
                //String tota = SPOInformation[5];
                int recoveryAnswer = Integer.parseInt(SPOInformation[5]);
                SAnswer = JOptionPane.showInputDialog(frame, "Answer the security question that you selected upon registration: "+securityQuestions[recoveryAnswer]);

                if(SPOInformation[2].equals(firstName)&&SPOInformation[3].equals(lastName)&&SPOInformation[4].equals(ModSPONumber)&&SPOInformation[6].equals(SAnswer))
                {
                    JOptionPane.showMessageDialog(frame, "Account recovered:\n\nUsername: "+SPOInformation[0]+" \n\nPassword: "+SPOInformation[1]);
                }
                else
                {
                    JOptionPane.showMessageDialog(frame, "Sorry, we could not verify your account.");
                }
            }
        }
        else
        {
            JOptionPane.showMessageDialog(frame, "Sorry, we could not verify your account.");
        }
    }
}

```

```

private boolean CheckInput()
{
    if(firstName == null || firstName.contains(" ")||firstName.isEmpty())
    {
        JOptionPane.showMessageDialog(frame, "Invalid Entry on 'first name!'");
        return false;
    }
    if(lastName == null || lastName.contains(" ")||lastName.isEmpty())
    {
        JOptionPane.showMessageDialog(frame, "Invalid Entry on 'last name!'");
        return false;
    }
    if(ModSPONumber == null || ModSPONumber.contains(" ")||ModSPONumber.isEmpty())
    {
        JOptionPane.showMessageDialog(frame, "Invalid Entry on 'SPO Number!'");
        return false;
    }
    else
    {
        try
        {
            //int num = Integer.parseInt(ModSPONumber);
            ModSPONumber = "SPO"+ModSPONumber;
            return true;
        }
        catch(NumberFormatException e)
        {
            JOptionPane.showMessageDialog(frame, "Error! Invalid SPO number.");
            return false;
        }
    }
}

```

RegisterWindow.java

Similar to the error checking in logging in (checking to see if the string is valid or empty) registration goes through the same process for each form of input: username, password, spo number, etc..

```
try
{
    if(CheckUser()&&CheckPassword()&&CheckFirstLastName()&&CheckSPONumber()&&CheckSecurityAnswer()&&IsValidUser())
    {
        RegisterUser(username,password);
        JOptionPane.showMessageDialog(frmRegister, "Your account has been created.");
        frmRegister.dispose();
    }
}
```

The “CheckUser()” function is the most essential in the system of function calls for the **RegisterWindow.java** class. This function will read through all of the file and the currently read file’s contents to see if a user or SPO number already exists as no two users can be the same, nor can two SPOs share the same unique ID.

```
private boolean CheckUser() throws IOException
{
    username = usernameField.getText();
    if(username.isEmpty())
    {
        JOptionPane.showMessageDialog(frmRegister, "Username cannot be empty!");
        return false;
    }
    if(username.contains(" "))
    {
        JOptionPane.showMessageDialog(frmRegister, "Username cannot contain spaces!");
        return false;
    }

    File folder = new File(System.getProperty("user.dir")+"\\Users\\");
    File[] listOfFiles = folder.listFiles();
    BufferedReader in;
    String fileContents = "";
    char c;
    if(CheckSPONumber())
    {
        SPONumber = Integer.parseInt(spoNumberField.getText());
    }
    else
        return false;

    String ModSPONumber = "SPO"+Integer.toString(SPONumber);

    if(listOfFiles != null)
    {
        for (int i = 0; i < listOfFiles.length; i++)
        {
            if(listOfFiles[i].toString().equals(System.getProperty("user.dir")+"\\Users\\"+username+".SPOFile"))
            {
                JOptionPane.showMessageDialog(frmRegister, "ERROR: This account already exists!");
                return false;
            }

            in = new BufferedReader(new FileReader(listOfFiles[i]));
            while(( c = (char)in.read() ) != (char)-1 )
            {
                fileContents += c;
            }
            if(fileContents.contains(ModSPONumber))
            {
                JOptionPane.showMessageDialog(frmRegister, "ERROR: This SPO number already exists!");
                return false;
            }
        }
    }
}
```


Then we must check to see if the user is acceptable for application use through the Verified Users documents described in the tutorial. If the admin has not set up these documents first, then users will be unable to access the logs unless the VUF file has the tag #Open-beta/ALL which allows any user unrestricted registration. This was achieved through the following source code:

```
private boolean isValidUser()
{
    String account = "";
    BufferedReader in;
    char c;

    if(username.equals("admin"))
        return true;

    try
    {
        File file = new File(System.getProperty("user.dir")+"\\VerifiedUsers\\Validated Users.vuf");
        if(file.exists() == false)
        {
            JOptionPane.showMessageDialog(frmRegister, "Error, could not initialize registration, the system admin needs to set up accounts for access.");
            frmRegister.dispose();
            return false;
        }

        in = new BufferedReader(new FileReader(file));
        while(( c = (char)in.read() ) != (char)-1 )
        {
            account += c;
        }
    }
    catch(IOException ioe){}

    String searchString = firstName+" "+lastName+" "+SPO+SPONumber;
    if(account.contains(searchString))
        return true;

    if(account.contains("#Open-Beta/ALL"))
        return true;

    JOptionPane.showMessageDialog(frmRegister, "Error, you're not authorized to register for this application");
    return false;
}
```

Here, we read in the list and see if the user's name and spo number are present, if they are not then the application will remain unusable for them until further noticed, otherwise, we can write out their .SPOFILE:

```
File file = new File(System.getProperty("user.dir")+"\\Users\\"+username +".SPOFile");
FileWriter out = new FileWriter(file,true); //the true will append the new data
String ModSPONumber = "SPO"+Integer.toString(SPONumber);
out.write(username+" "+SHA256.Hash(password)+" "+firstName+" "+lastName+" "+ModSPONumber+" "+SQuestion+" "+SAnswer+System.getProperty("line.separator"));
file.setReadOnly();
out.close();
```

Log.java

Apart from much of the UI elements, the main algorithmic logic which pertains to the actual functionality is achieved with the following block of code:

```
int j = 120;
int k = 120;
for(int i = 0; i < 24; i++)
{
    boxes[i]= new JComboBox(locations);

    boxes[i].addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {

            end = boxes[index].getSelectedItem().toString();
            if(end.equals("RA Checks") || end.equals("Sec Checks"))
            {
                end = JOptionPane.showInputDialog("Where will you end?");
                from = to;
                table.setValueAt(from, column,0);
                table.setValueAt(boxes[index].getSelectedItem().toString(), column,2);
                start = end;
                to = timeFormat.format(time = new Date());
                table.setValueAt(to, column,1);
                boxes[index].setEnabled(false);
                column++;
            }
            else{
                ShortestPath pathFinder = new ShortestPath();
                String path = pathFinder.find(start, end);
                from = to;
                end = start;
                table.setValueAt(from, column,0);
                table.setValueAt(path, column,2);
                start = boxes[index].getSelectedItem().toString();
                to = timeFormat.format(time = new Date());
                table.setValueAt(to, column,1);
                boxes[index].setEnabled(false);
                column++;
            }
            if(index <= 22)
            {
                index++;
                boxes[index].setEnabled(true);
            }
        }
    });

    if(i == 0)
        boxes[i].setEnabled(true);
    else
        boxes[i].setEnabled(false);
}
```

This function is rather simple, it exists in a loop where for each area selection box that is selected, this segment of code will be executed, it's purpose is to simply get the time of the

selected item and place it in the log table along with the selected item and it's path from the previous selected item. When a combo box has been selected, it will deactivate itself and the next one in the list will activate.

ShortestPath.java

The method of tracking paths between markers was achieved using Dijkstra's algorithm of finding the shortest path between two nodes on a weighted graph. Simply put, the algorithm will look at all of the connections from the starting point to the ending point and calculate their total weights; the path that weighs the least (path weight = distance between markers) will be the chose path.

The algorithm for this portion of the code is fairly complex and non-trivial as with the SHA256 hashing algorithm, for these reasons and the sake of readability, source code will not be embedded within the discussion but rather be include at within the Log Release 1 zip file for analysis.



In order to get an idea of how the node system works, a map of the college was used to highlight the areas which a SPO officer must check and down scaled distances of their locations relative to their neighbors.

These nodes were then manually entered into a graph in Java:

```
g.addVertex("Office", Arrays.asList(new Vertex("Myer", 2), new Vertex("ResLife", 1)));
g.addVertex("ResLife", Arrays.asList(new Vertex("Office", 1), new Vertex("Royer", 1)));
g.addVertex("Royer", Arrays.asList(new Vertex("ResLife", 1), new Vertex("Schlosser", 3)));
g.addVertex("Schlosser", Arrays.asList(new Vertex("Royer", 3), new Vertex("Myer", 5), new Vertex("AQuads", 4)));
g.addVertex("Myer", Arrays.asList(new Vertex("Office", 2), new Vertex("Schlosser", 5), new Vertex("Alpha", 3), new Vertex("Ober", 2)));
g.addVertex("Ober", Arrays.asList(new Vertex("Myer", 2), new Vertex("Library", 2), new Vertex("CHL", 1), new Vertex("Brinser", 1)));
g.addVertex("Alpha", Arrays.asList(new Vertex("Myer", 3), new Vertex("Library", 1), new Vertex("AQuads", 4)));
g.addVertex("Library", Arrays.asList(new Vertex("Brinser", 3), new Vertex("Alpha", 1), new Vertex("Ober", 1), new Vertex("BSC", 1)));
g.addVertex("AQuads", Arrays.asList(new Vertex("Schlosser", 4), new Vertex("Alpha", 4), new Vertex("BSC", 1), new Vertex("GYM", 1), new Vertex("LOT4", 1)));
g.addVertex("BSC", Arrays.asList(new Vertex("Library", 1), new Vertex("Brinser", 1), new Vertex("BSC", 1), new Vertex("AQuads", 1), new Vertex("GYM", 3)));
g.addVertex("GYM", Arrays.asList(new Vertex("BSC", 3), new Vertex("AQuads", 1), new Vertex("BSC", 1), new Vertex("AQuads", 1), new Vertex("Leffler", 1)));
g.addVertex("LOT4", Arrays.asList(new Vertex("AQuads", 1), new Vertex("Leffler", 1), new Vertex("Lake", 2)));
g.addVertex("Leffler", Arrays.asList(new Vertex("LOT4", 1), new Vertex("GYM", 1), new Vertex("Lake", 2), new Vertex("Track", 1)));
g.addVertex("Lake", Arrays.asList(new Vertex("LOT4", 2), new Vertex("Leffler", 2), new Vertex("Brown", 1), new Vertex("YC", 1)));
g.addVertex("Brown", Arrays.asList(new Vertex("Lake", 1)));
g.addVertex("YC", Arrays.asList(new Vertex("Track", 1), new Vertex("Lake", 1), new Vertex("Solar", 2)));
g.addVertex("Solar", Arrays.asList(new Vertex("YC", 2), new Vertex("Track", 3)));
g.addVertex("Track", Arrays.asList(new Vertex("Leffler", 1), new Vertex("YC", 2), new Vertex("Solar", 3), new Vertex("Quads", 5)));
g.addVertex("Quads", Arrays.asList(new Vertex("Track", 5), new Vertex("Soccer", 1)));
g.addVertex("Soccer", Arrays.asList(new Vertex("Quads", 1), new Vertex("Stadium", 3)));
g.addVertex("Stadium", Arrays.asList(new Vertex("Soccer", 3), new Vertex("Apts", 2)));
g.addVertex("Apts", Arrays.asList(new Vertex("Stadium", 2), new Vertex("Founders", 3)));
g.addVertex("Founders", Arrays.asList(new Vertex("Disc", 10), new Vertex("Apts", 5), new Vertex("Brinser", 5)));
g.addVertex("Disc", Arrays.asList(new Vertex("Founders", 10), new Vertex("CHL", 7)));
g.addVertex("CHL", Arrays.asList(new Vertex("Disc", 7), new Vertex("Ober", 1)));
g.addVertex("Brinser", Arrays.asList(new Vertex("Founders", 5), new Vertex("Ober", 1), new Vertex("Library", 3), new Vertex("BSC", 1)));
```

When a user selected a location, the corresponding location will be mapped to a preexisting node in the graph and the path from the current node to the previous will be calculated. For accuracy, a user should always mark the closets objects to them within reason to avoid the log generating an improper path. For example, if a user started at the at point A and needed to go to point E, they may have actually taken the following path:

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

But if a user incorrectly uses the app to log the path, the application may generate the path as being:

$$A \rightarrow D \rightarrow E$$

Where A to D to E was the shortest path, not necessarily the path with the pit stops as the user actually took. When crossing large distances, the user must be sure to mark their land markers as they go.