inVøke

Technical Documentation

February 3rd, 2020

Zachary Hoffman, James Kang, Sam Lundquist, Owen McEvoy, Mason Sayyadi

Table of Contents

1. Outline

This documentation should assist the programmers in running, modifying and understanding the inVøke program. We will give an overview of:

1) Systems requirements and dependencies
2) Installing the program
3) Running the software
4) Relationship between files
5) Inner workings of all code, including classes and its member functions

2. System Requirements

2.1 Command-Line Interface Requirements

The command-line interface can run on any OS that supports std C++ 11 and ncurses. Our decision to construct the project in this language has given us this flexibility. The program does not need any extra libraries to be installed, and only has one imported library, which is ncurses to display the interface.

2.2 Xcode Cocoa Interface Requirements

In order for the program to function, the client must have a MacOS on version 10.14, and basic computer attributes (mouse, keyboard, a monitor). If they wish to modify/ read the Xcode project files, then it is recommended to have Xcode development. Any other dependencies used within the Xcode project are already built into the INVOKE.app and it is not necessary for the user to install them.

Each component and its versions are as follows:

1. MacOS: version 10.14 or newer.
2. Xcode: version 11.0 and above

Special Note: The requirements specify that macOS version 10.14 (or newer) is required to run the project. This is true in our current version. In order to run the project on macOS 10.13, the INVOKE.app would need to be rebuilt within Xcode with the target set to macOS 10.13.

3. Installation Instructions

3.1 The command-line interface

Unfortunately, the front end implementation of our project in XCode is not complete at this time. This is due to the tremendous difficulty involved in properly linking C++ code within a Swift based program. We realized this far too late and were not able to come up with a proper fix within the given timeframe.

In response, we have set up a command-line interface solution using the ncurses library. This allowed us to showcase the functionality of the back-end system effectively within the given time constraints.

Installation:
The program will be installable via a makefile using the following commands:

```
$ make //Builds the program
$ make run //Runs the command-line interface
```

3.2 The Xcode cocoa interface

Our main goal when building this piece of software was for it to be convenient for the client. As such, the main program should be very easy to access and run on the client's system. We decided to set up a dmg installer, more information about what they are and how they work can be found here: Apple Disk Image.

Once the dmg file has been downloaded and located on the user's system, it needs to be installed. Here is a handy article that helps with this process:

How to Install Programs from DMG Files on Mac

Note: This was the intended functionality of our program, due to issues with the linkage of c++ code in swift, the dmg version of the program is not complete. The alternative user interface solution that we have implemented is a command-line interface.

4. User Documentation

To run the command-line interface, you must first make the project by typing the following in your terminal from the project directory:

```
 $ make //build the application and needed directories from Makefile
```

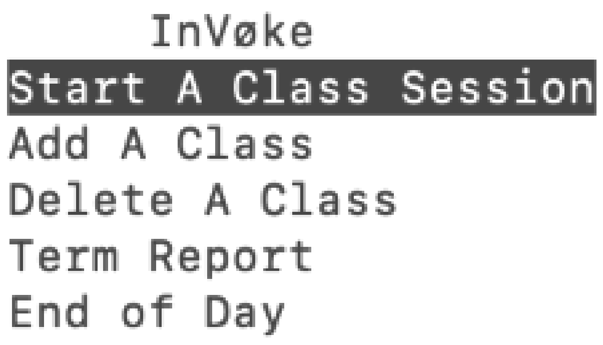
Once the project has been built, a directory structure will be added to your Documents folder. This is where all files will be stored for the program. Before you can run the program you will need to locate this directory.

Once you have found the Invoke directory, you will need to add a roster file to the Rosters folder. This is important because, without this file, you will not be able to import a class roster into the program. So, get the roster file of your choice and then we can move on to the next step, running the program.

To run the program, we call the following in the terminal from the program's directory where we called 'make' earlier.

```
$ make run //runs the application that we built previously
```

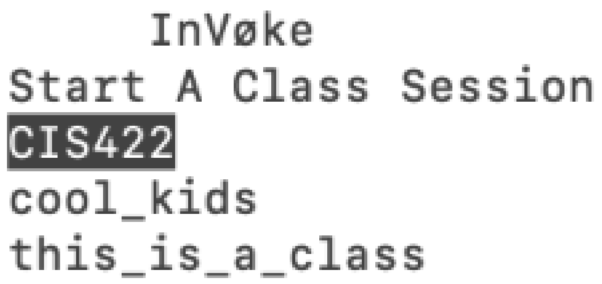
Now that the program has been run successfully, you will be greeted by the ncurses interface in your terminal as displayed in the figure :



```
     InVøke
Start A Class Session
Add A Class
Delete A Class
Term Report
End of Day
```

The user then has a choice to start the class session, add a class, delete a class, term report and end the day. In order to navigate through the options, the user will press the Up and Down arrow key on their keyboard. To select the highlighted option the user must press the Right arrow key. There are five different cases.

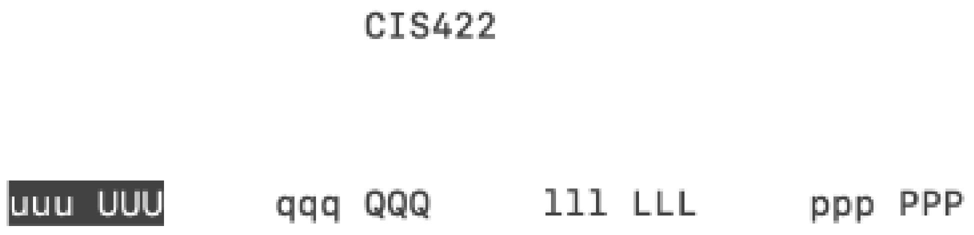
Case 1:



```
     InVøke
Start A Class Session
CIS422
cool_kids
this_is_a_class
```

If the user selects the first option (Start A Class Session). They will be directed to another menu.

The menu will consist of our entire list of classes that are stored in our data (a reference to this last in this paper). Similarly to our initial menu, the user can use Up and Down arrow keys to navigate through the class list and press the Right arrow key to select the class. If the user wishes to select no class and return to the main menu, they will press the Left arrow key. Pressing the right arrow key will trigger a new environment that looks as follows.



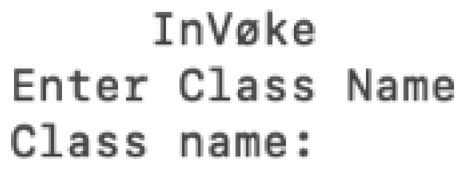
```
              CIS422


uuu UUU     qqq QQQ     lll LLL     ppp PPP
```

At the top will be the name of the class. Followed by this are four different students' first and last names. To navigate through the students the user must press Left and Right arrow keys. To flag a student (for whatever reason) the user must press the Up arrow key. To give the student a point for participation the user must press the Down arrow key. Doing either of these actions (down and up arrow keys) will result in the student getting placed in the last 25% of the queue randomly. It will also increase the student's participation by one (will be able to tell at the end of the day report). To exit this environment the user must press the Space keystroke to return to the main menu.

Case 2:

If the user selected the second option (Add a Class) the user will be directed to the following screen.



```
      InVøke
 Enter Class Name
 Class name:
```

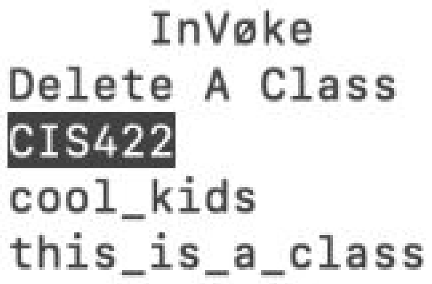
In this screen, the user can type in the class name that they are trying to add. Only numeric values and English characters are accepted. Adding a class doesn't necessarily mean that all of the data has been added as well. To add the class data the user must add the roster of the class to the "Rosters" directory located in the Invoke directory. The user must confirm that the file is of type .txt, is delimited by either a tab or comma, and the name of the class matches the name of the file (before the .txt). It is important that the user uses the first line of the file to create relevant headings for each column of data because the first line of the file will not be imported into the roster. The format of the file must follow this.

<first_name> <t> <last_name> <t> <student_id> <t> <email> <t> <phonetic_spelling> <t> <reveal_code>

Only rows with adequate data will be added to the class roster. After entering the class name the user must press the enter key to add the class into our list. If the user wishes to exit this environment and go back to the main menu they must press the Left arrow key.

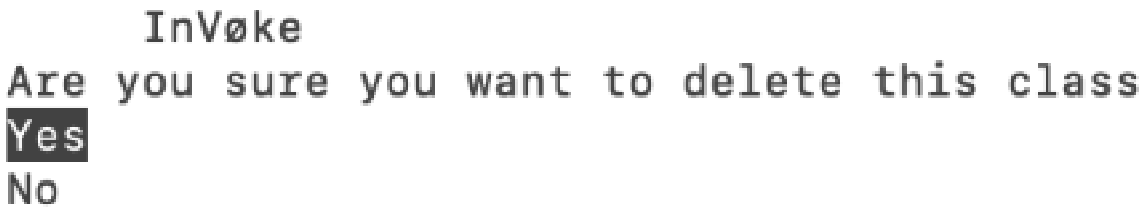
Case 3:

If the user selected the third option (Delete a Class) the user will be directed to the following screen.



```
     InVøke
Delete A Class
CIS422
cool_kids
this_is_a_class
```

Just like in the first part of case one the user must choose which class they would like to delete. To navigate through the list of classes the user will press the Down and Up arrow keys. To return to the main menu the user will press the Left arrow key. To select the following highlighted class (to delete) the user must press the Right arrow key. After the user presses the right arrow key, the user will be directed to the following
screen.

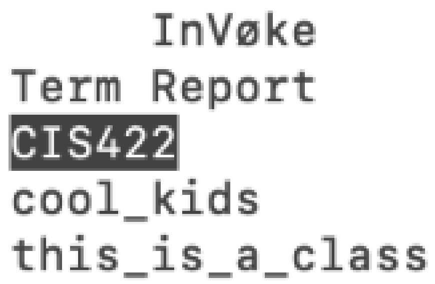


```
     InVøke
Are you sure you want to delete this class
Yes
No
```

If the user wishes to delete the class that they have selected they must simply press the Right arrow key (since the highlight is by default on the “Yes”). If the user wishes to not delete the selected class they could either navigate to the “No” (by pressing down arrow key followed by right arrow key) or they could press the Left arrow key to go back to the initial case 3 screen.

Case 4:

If the user selected the fourth option (Term Report) the user will be directed to the following screen.



```
     InVøke
Term Report
CIS422
cool_kids
this_is_a_class
```

Just like in the first option the user must navigate through the class list and pick the class that they would like the end of the term report for. To do this the user can press the Down and Up arrow keys to navigate through the list. Then they must press the Right arrow key to choose that class (which they want the end of the term report for). If the user wishes to exit back to the main menu they must press the left arrow key. Pressing the right arrow key will trigger one of two actions. If the data of the class does not exist, the program will print an error message with why the command didn't work (the only case is if the file inside ~/Documents/Invoke doesn't exist). If it did work then the program will store the end of the term report to ~/Documents/Invoke/InvokeReports and return to the main initial screen.

Case 5:

Finally, if the user selects the fifth option (End of Day). The program will exit.

5. Programmer's Documentation

5.1 Backend Component Outline

The C++ code files are displayed below, with a short detail about what each file does and how it communicates with the other files.

1. Student.cpp/.h: A file containing the 'Student' class and all of its functions. The file doesn't depend on any other files.
2. ColdCallClass.cpp/.h: A file containing the 'ColdCallClass' and all of its functions. The files depend on the Student.cpp/.h and ColdCallClassQueue.cpp/.h

3. ColdCallClassList.cpp/.h: A file containing the 'ColdCallClassList' class. The file depends on ColdCallClass.cpp/.h
4. ColdCallQueue.cpp/.h: It contains all the information about the class "ColdCallQueue". The file depends on the Student.cpp/.h
5. Reporting.cpp/.h: It contains code to extract information that the User wants. The file has two main functionalities: the first is appendDailyLog(), which maintains a log file of user events, the second is exportTermData(), which pulls all of the student data from a specific class to help the instructor with grading. The file is dependent on the Student.cpp/.h
6. RosterIE.cpp/.h: It contains code to import and export the roster into a text file stored in a separate directory (without participation data). The file depends on the Studfent.cpp/.h
7. StatePersistence.cpp/.h: It contains code to save the current state of the program (before the program quits), this file also contains functions to extract the last saved data into a running program (when we start up the program again)

5.2 Back End Code Breakdown

ColdCallClass.cpp/h

The Class type "ColdCallClass" has 4 private attributes and one private member functions. The class also has 10 public member functions.

Private:

string **class_name:**
is the name of the class (with type string).

vector<shared_ptr<struct Student>> **roster:**
is the current roster of the students in the class. Its type is a vector, and it stores data type student.

vector<shared_ptr<struct Student>> **newRoster:**
is the potential new list of students. It's just like Roster however it might have different Information in case someone dropped the class and such.

***Class ColdCallQueue* <u>call_queue:</u>**

is a queue of students. It is of type Cold Call Queue and this class will be addressed in the next part of the writing.

<u>compareShared:</u>

is a private member function that compares the **Roster** to the **newRoster**. This member function does not compare the array participation of each student however it compares its basic attributes. If there happens to be a change then the function will return a Boolean (True for a change, False for no change). Its arguments are pointers to vectors that contain the information of all students who added the class, updated their main attributes, or removed the class. Its arguments are empty when the member function is first called. After the function runs if the Boolean attribute returned was True then at least one of the vectors will have some information in it.

Public:

void* <u>loadState(</u>const string* **class_name):**

Populates all of the private variables in the Cold Call Class. This function takes a string argument for a “class name” and updates the **Class Name** of the Cold Call Class variable with it as well as the Cold Call Queue’s **class_name**. Load state also calls on statePersistence.cpp’s load Class function in order to populate an empty array of “Student” structs (*var.* <u>loaded</u>) given the specified name of the class from **class_name**. The **roster** of the ColdCallClass also gets updated from the newly loaded array of **Student** structs (*var.* <u>loaded</u>)**.** The loadState function also loads the ColdCallClass’ student queue with its own **getStudent()** function.

***string* <u>getClassName()</u>:**

Getter function that returns the **class_name** value of the **ColdCallClass**

***ColdCallQueue* *<u>getQueue()</u>:**

Getter function that returns the address of the **ColdCallClass’ ColdCallQueue**

vector<struct *Student> **<u>getStudents()</u>:**

Creates a new vector of Student struct pointers (var. **class_list**) that iterates through the **ColdCallClass roster**, appending the **roster** values' pointers. Returns class_list.

bool **<u>compareRoster(</u>***vector<struct Student *>** **addedStudent*****,***
*pair<struct Student *, struct Student *>* **updateStudents**
*vector<struct Student *>* ***removedStudents):**

Compares the class' current **roster** to the previously imported **newRoster** and stores the results in three vectors. Returns true if there are differences between the two, indicating that there is something to update. This function is used by external classes.

bool **<u>compareRosterShared(</u>**
vector<shared_ptr<struct Student>> ***added*****,***
pair<shared_ptr<struct Student> , shared_ptr<struct Student>> **modified,**
vector<shared_ptr<struct Student>> ***removed):**

Compares the class' current **roster** to the previously imported **newRoster** and stores the results in three vectors. Returns true if there are differences between the two, indicating that there is something to update. This function is used internally by the **ColdCallClass** to identify what needs to be changed when the **newRoster** data is integrated into the existing **roster**.

void **<u>updateRoster(</u>***const bool* **should_update):**

Updates the **ColdCallClass**' existing **roster** with information from the previously loaded **newRoster** (if **should_update** is true) or clears out the **newRoster** information to cancel the import of a roster (if **should_update** is false). Uses the **compareRosterShared()** function to get a specific list of what actions need to be taken: adding students, modifying students' basic attributes, and removing students. Once any updates have been made, they are saved to persistent storage using the **storeClass()** function.

void **<u>exportRoster(</u>***string* **filename):**

Exports the **ColdCallClass' roster** to a file with the given the **filename**

void **addStudent**(*shared_ptr<struct Student>* **student**):

Adds the student struct pointer to the **ColdCallClass roster** as well as import the address of that to the **ColdCallQueue**.

void **exportRoster(***string* **report_file):**

Calls the exportTermData() function on the ColdCallClass roster to write to a file with a title associated with **report_file**

ColdCallClassList.cpp/h

Returns and does similar functionality as ColdCallClass, except populates and modifies various class lists. This program also loads and saves information for cold call class lists.

Private:

map<string, shared_ptr ColdCallClass>**classes:**

It provides a way to store and lookup classes (ColdCallClass) using their class name (string).

Public:

void **loadClasses():**

Creates a vector of strings (var. **class_names**) that is populated by calling the **statePersistence loadClassList()** function. Once loaded, the class uses the **ColdCallClassList** function **addClass** to add and load in any of the data.

void **saveClasses():**

Creates a vector of strings (var. **class_names**) and populates the **ColdCallClassList**'s private **classes** map variable at the first index (the string name). The **statePersistence saveClassList()** function is called to save state information.

void **addClass(*string* class_name):**

Function checks to see if classes exist, and exits if they don't exist, exit function. If classes exist, load in class from the file using a new sharedPointer **ColdCallClass** struct (var. **addClass**) and calls the load state from **statePersistence**. The **statePersistence saveClassList()** function is called to save state information.

void **deleteClass(*string* class_name):**

Calls itself on the class_name variable, and erases the **ColdCallClassList classes** variable with the index matching "**class_name**" as the key values

vector<string> **getClassList():**

Creates a new vector<strings> called **class_name_list**. Function iterates through the **ColdCallClassList's** classes map at the 2nd position (shared_ptr **ColdCallClass**) and sets it equal to a new shared_ptr<struct **ColdCallClass**> (var. **currentClass**). **Class_names_list** populates itself with names by calling the **getClassName()** function from the **currentClass**. Returns the list of class names in **class_name_list**

Class ColdCallClass ***getClass(*string* class_name):**

Creates a new **ColdCallClass** pointer with a value set to nullptr (var. **gotten**). Function checks to see if the **ColdCallClassList's classes** map is populated, and proceeds to set the classes map value to the key of the class at (**class_name**) to a new shared pointer **ColdCallClass** struct (var. **gotten_sptr**). Returns the populated **ColdCallClass** pointer "**gotten**"

ColdCallQueue.cpp/h

Focuses on the queue of data structures that the program depends on. This file also contains functions for adding, and removing students from the queue. This file also contains functions that load class and student queues, as well as removes the students on deck.

Private:

deque<struct Student *> **student_queue:**
Creates a deque for the students in the queue

string **class_name:**
Name of the class the queue is for

Public:

String **getClassName():**
Getter function that returns the private string **class_name** in **ColdCallQueue.**

void **setClassName(***string* **class_name):**
Setter function that sets the private string **class_name** in **ColdCallQueue** to the argument **class_name.**

void **loadQueue(***vector<struct Student *>* **newList):**
Loads a vector of Student struct pointers (var. **newList**) into the private **student_queue** deque and returns none.

void **exportQueue():**
Creates a new vector of student struct pointers (var. **exported**) and assigns them to the private **student_queue** deque in **ColdCallQueue**. **storeClass()** is called from **statePersistence** on the private string **class_name** of **ColdCallQueue** and the **exported** vector. The data is written from the vector of student struct pointers (**exported**) to a file.

void **addStudent(***struct Student* ***student):**
Setter function that adds a Student struct argument (var. **student**) to the end of the private **student_queue** deque in the **ColdCallQueue**

vector<struct Student *> **getOnDeck(***const unsigned int* **requested_count):**

Creates a blank vector of student struct pointers (var. **on_deck**) that will get populated in a for loop (where int i = 0 with an increment of 1 each time) by the **ColdCallQueue** private **student_queue** deque at index[i]. The range of “i” is from 0 to **requested_count** (num specified) number of times). Function returns populated deck of student pointers.

void **removeOnDeck(***const unsigned int* **position,** *const char* **flag):**

Checks to see if the position argument is smaller than the total size of private **student_queue** variable. Once checked, function creates a Student struct pointer (var. **participated**) that will be set to the value of the private **student_queue** variable at the index of **position.** The function records the timestamp information and stores it to the **participated** student pointer struct, and adds the information (along with the private **class_name** and **flag**) to the **Reporting** file’s **appendDailyLog()** function. The function erases the student from the **student_queue** using his position. The function then randomized the addition of the removed student to the back half of the **student_queue**.
Calls the **exportQueue()** from the **ColdCallQueue** class.

Reporting.cpp/h

This file focuses on writing any type of evaluation files. Files export term data as well as tells who participated in class events for the daily log file.

Public:

void **appendDailyLog(**string **class_name,** struct Student ***student,** time_t **time,** char **flag):**

Creates a character buffer and checks to see if the state of the program is possible to call the function. Function populates a string (var. **path**) with a pathway by calling the **getPath(0)** function. Function also concatenates the file name and the buffer to the end of the path in a blank string (var. **filename**). Using an output file stream (var. **dailyLog**), the program opens the given pathname (**filename**) and writes the **student** argument’s **first_name**, **last_name**, **email**, **buff**, **flag** to the **dailyLog**.

void **exportTermDate(***vector<shared_ptr struct Student>* **roster,** *string* **filename):**

Function populates a string (var. **path**) with a pathway by calling the **getPath(0)** function. Function also concatenates the file name and the buffer to the end of the path in a blank string (var. **filename**). Function uses an output file stream and a for loop with **exportFile**(**filename**), to open and write the "# of times participating", "First Name", "Last Name", "Student ID", "email", "Phonetic Spelling", "Reveal Code" of each student (**rosterSize** amount of times) to **exportFile**. Prints an error message if the file cannot be opened and calls the **createDir()** to create the directories if they do not already exist.

Student.cpp/h

Class for a student object that contains a student_id, first name, last name, email, phonetic spelling, and reveal code. This also includes a participation tuple that keeps track of time and character flag values.

Public:

string **student_id:**

The student's id number

string **first_name:**

Student's first name

string **last_name:**

Student's last name

string email:

Student email

string **phonetic_spelling:**

Student phonetic_spelling

char **reveal_code:**

Student reveal_code

vector**<tuple<time_t, char>> participation;**

Contains the time_t and character values to be accessed

bool **<u>compareBasicAttributes</u>(**const struct Student ***b):**

Function sets a boolean flag (var. **Is_equal** = False) Compares self to the argument Student struct argument (**b**) on the following: **student_id**, **first_name**, **last_name**, **email**, **phonetic_spelling**, **reveal_code**. If one of the comparisons isn't equal, then return the boolean False (that the students aren't equal), otherwise, the function returns true.

void **<u>updateBasicAttributes</u>(**const struct Student ***b):**

Compares self attributes to the attributes of argument Student struct (**b**) on the following: **student_id**, **first_name**, **last_name**, **email**, **phonetic_spelling**, **reveal_code**. If one of the self-comparison's isn't equal, the function will set its self attribute to **b's attribute.**

void **<u>participate</u>(***time_t* **time,** *const* **char flag):**

Creates a tuple of the **time**, and **flag** arguments and appends it to the **Student** tuple **participation** variable.

statePersistence.cpp/h

Focuses on the consistency of the classes, as well as the loading, saving and storing of each of our classes.

Public:

void **<u>deleteClass</u>(**string **filename):**

Sets the string **filename** to a function that deletes the file. If the file cannot be deleted, the function prints an error message.

void **createDir():**

Function calls the **statePersistence getPath**(1) function (var. **meta**) function to create a pathway. The pathway is used to populate a string **testFile** (var. **testFile**) and create a file out of the string using by concatenating "text.txt" to the end of the pathway. C++'s output filestream command (ofstream) opens the file and returns the directory path if the stream doesn't open. If **testFile** was open, delete it, and get the pathway of the other directories. Continue testing the paths of the other 2 directories to see if they all exist. If the directories don't exist, create them with the system("mkdir ~/Documents/Invoke/Roster"). If they do already exist, then remove them with the C++ remove function.

string **getPath(*int* hidden):**

Creates a char (var. **dir**) that is set to equal the C command getenv(). The function also creates the startPath(dir). Compares the **hidden** argument to 2 separate if statements. If hidden is a '1', the function will return the path to .invokeData directory. If hidden is a '2', the function will return the pathway to the Roster directory. Any other number will return the pathway to the InvokeReports directory which is where the daily logs and term report files are written to.

void **saveClass(*vector<string>* class_name):**

Call **getPath(1)** (var. **path**) to create a path to /.invokeData, and is concatenated with "listOfClasses" at the end of the path, and opens the pathway with ofstream. Returns an error if the pathway cannot be open. For every class, print the class names. Once printed, close the pathway stream.

vector<string> **loadClass():**

Call **getPath(1)** (var. **path**) to create a path to /.invokeData, and is concatenated with "listOfClasses" at the end of the path, and opens the pathway with ifstream. Return error if the file can't open. If the file is open, the function uses a nested while loop and the getLine() function to iterate through each line in the file, as well as every word (separated by a delimiter) for each line. The function makes a blank vector of strings (var. **entry**) and indexes the following entries: first_name, last_name, student_id, email, phonetic_spelling, reveal_code. For every student, a tuple with the timestamp and date values (var. **times**) are pushed to the student's participation vector.

vector<string> **loadClassList():**

Function creates a vector string (var. **class_name**) Call **getPath(1)** (var. **path**) to create a path to /.invokeData, and is concatenated with “listOfClasses” at the end of the path, and opens the pathway with ifstream. Return error if the file can’t open. If the file is open, the function uses a nested while loop and the getLine() function to iterate through each line in the file, as well as every word (separated by a delimiter) for each line. Append the line to the **class_name** function in the nested loop and return class names;

RosterIE.cpp/h

char **getDelim(string line):**

This is called in the importRoster roster function, which passes the first line of the file, and determines the delimiter of the file. The two acceptable delimiter types are tabs and commas. If the function does not recognize either of these two options, it returns a 0 and the roster does not get imported.

std::vector<std::shared_ptr<struct Student>>
importRoster **(string file):**

This function is used to import a class roster and is utilized in the ColdCallClass **loadRoster** function. Upon success, the function returns a loaded vector of students. If the file has inadequate data, the user is informed of what the specific issue is and at what row the error occurred. Entries with inadequate data are not added to the vector of students. Should the file have an unrecognized delimiter, the function returns an empty vector.

void
exportRoster(std::vector<std::shared_ptr<struct Student>>, std::**string file):**

This function takes in a class roster, or a vector of students, and the corresponding class name. The general purpose of this function was to allow the instructor to export the class roster, modify it, and then reimport the updated roster in the event a student was either dropped from the class or added in.

Main.cpp/.h

This file contains all of the front end functionalities and executes them.

Int StartS():

This function is the initial function that gets called to display the window containing the main menu information. Its return integer is the index of which option the user would like to choose.

Int ClassListSR(ColdCallClassList ourclasses):

This function is to display the list of classes for starting up the program. Its only argument is the class type **ColdCallClassList** which will have the list of the classes we need within it. It will return the index of the class that the user would like to start. S stands for Start R stands for Running.

Int ClassListSD(ColdCallClassList ourclasses):

This function displays the class list for the deletion of the function. Its only argument is the class type coldcallclasslist which will have the list of the classes we need within it. It will return the index value of the class that the user would like to delete. S stands for Start D stands for deleting.

Int ClassListReport(ColdCallClassList ourclasses):

This function is to display the list of classes that the user would like an end of term report for. Its only argument is the class type coldcallclasslist which will have the list of the classes we need within it. Its return value is the index of the class that the user would like the end of the term report for.

Int classSession(class ColdCallClass * selectedClass, string className):

This function is to actually run the on deck interface. Given a pointer to the cold call class we want to open up the class to start the on deck interface. The second argument is there because the class name within the cold call class is private data so we cannot access it directly. If this function runs without any errors it should give the user a cold-call system for the class they selected in Class List SR.

bool areyousure()

This function simply asks the user if they are sure they want to make the deletion they wanted to make. It either returns TRUE (for the “Yes” option) or FALSE (for the “No” option).

int addClassname(ColdCallClassList &ourclasses)

This function asks the user for input. Given the input, the function stores the string value into the single argument we have(ourclasses). Its return value is always the main screen.

Int main():

This is our main function which uses all of the implementation specified above.