Cold Call Helper SDS

Authors: Nolan Cassidy, Andrew Evelhoch, Jake Gianola, Kyle Kincaid, Kylie Quan
Last edited 4/29/2019
(Adapted from a class handout by A. Hornoff)

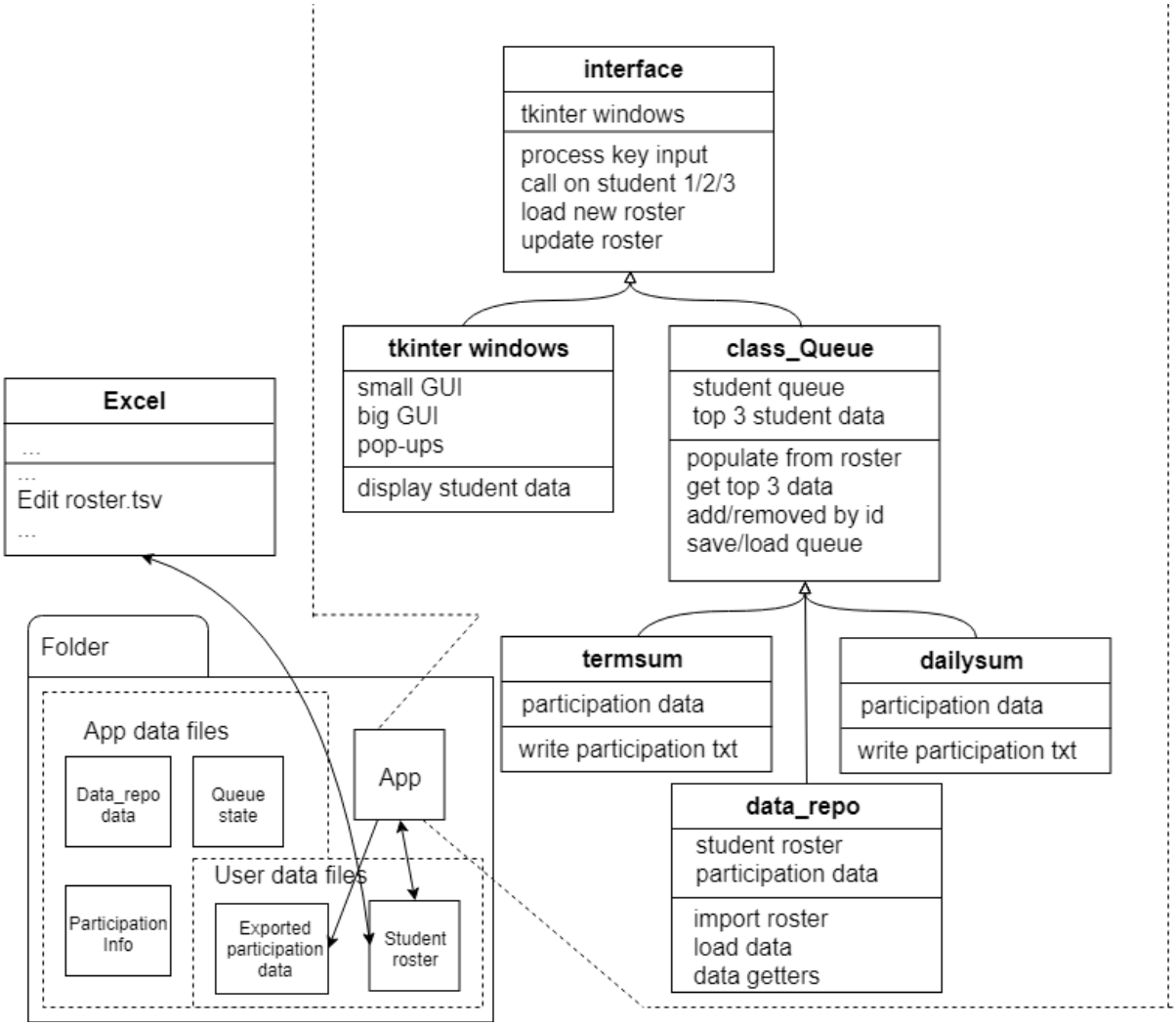
This will be a simple solution to cold calling and class participation. The design will not draw the classes eyes away from other applications like powerpoint on the screen.

- There will be a smaller mode to simply show the next three students, for use when projecting the screen in front of the class.
- There will be a full screen mode to show the image, name, and phonetic spelling in bold readable font for the next three students, for full-screen use when the screen is not being projected.

On both modes there will be a menu bar to access all the functionalities of the buttons: advance student 1/2/3 (optionally with mark for lost participation or needs follow-up), start new term, export term summary to date, export daily summary, change GUI mode.

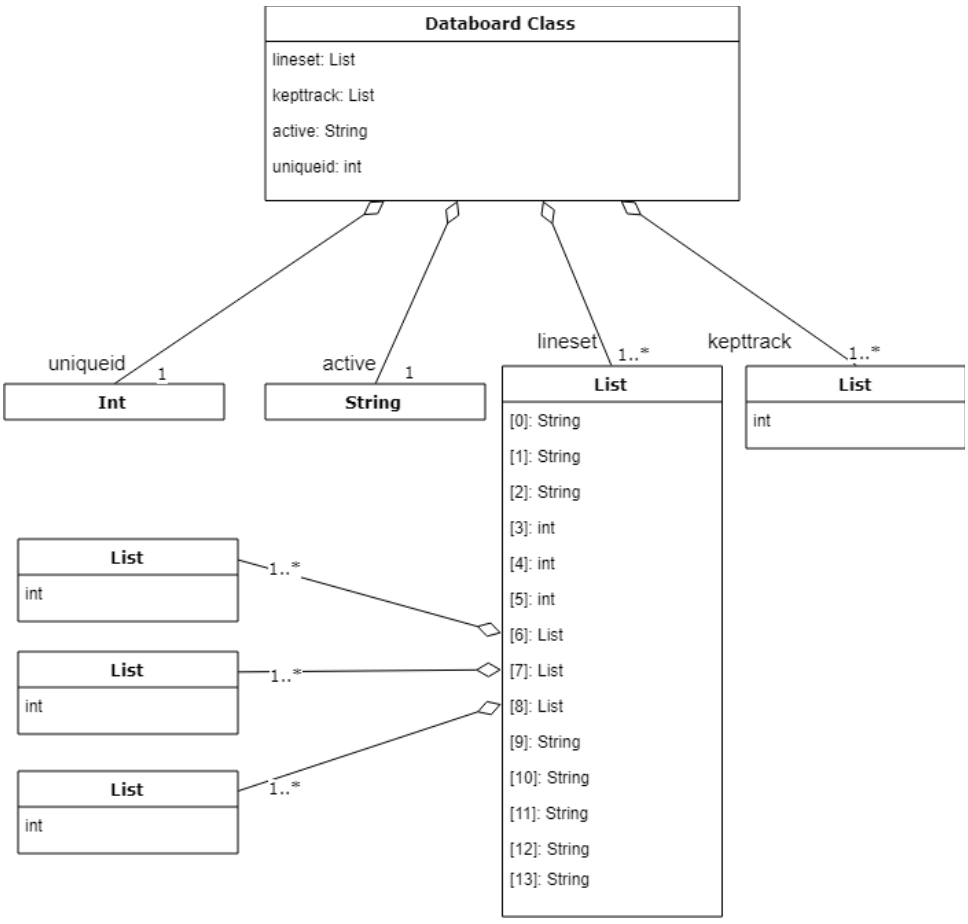
The major parts of our system will be the Interface, the Queue and the Data Repo. The instructor will edit a roster file in excel and add all the data for the students in the class. This file will be loaded into the data repo and the repo will handle the data for the class. In addition to storing the information for the roster, it will also keep track of participation marks. The queue will randomly order the list of students in the class and prompt each one to be called on once before any repeats happen. The GUI has 2 display modes: small and large. The small mode will display simply the students preferred names and last names. The large mode will display the students full names, phonetic pronunciation and preferred names. Finally, there will be an export option for the daily or term summary of student performance. (Shown in System Structure Diagram, next page)

System Structure



This is a simplified UML class diagram, only functions that are called by external classes/input are included in each class description and only the most important attributes. Check the individual module class diagrams to see all of the functions and attributes of that class.

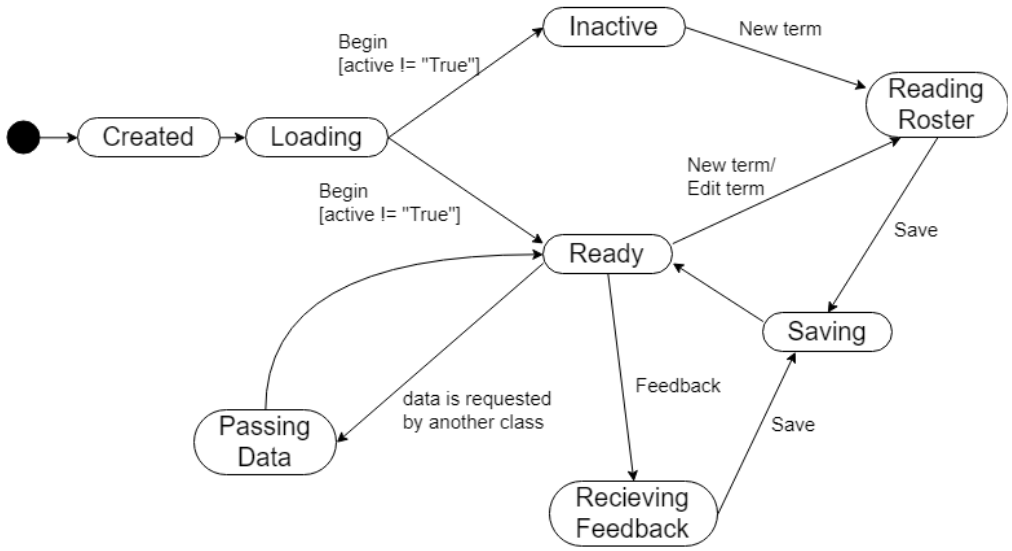
Data Repo



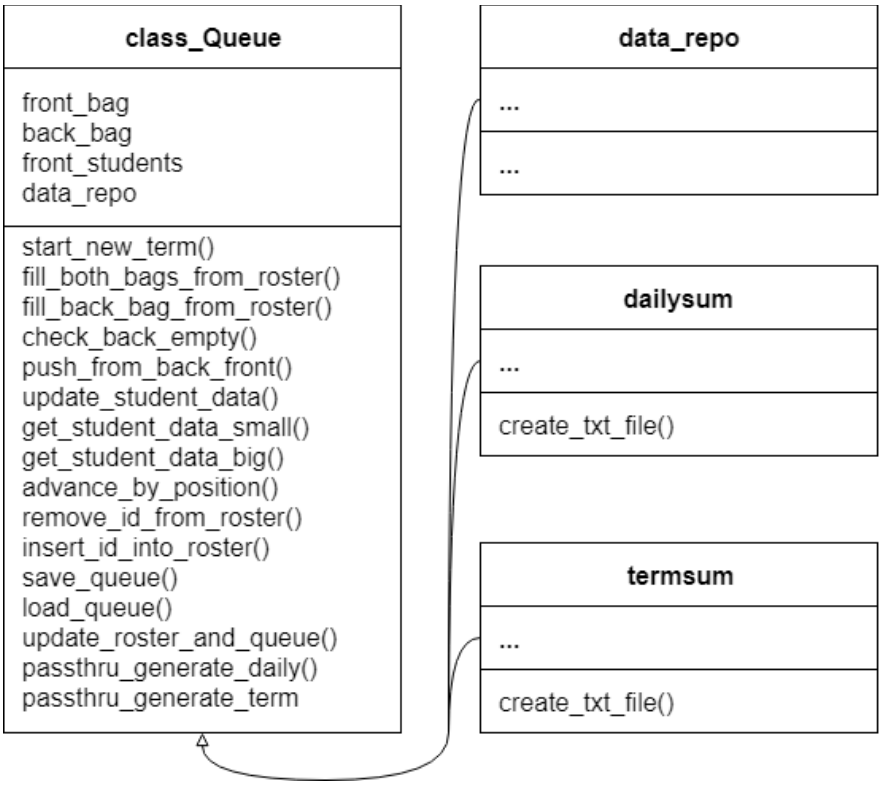
UML class diagram for the databoard class, showing the datastructures

The main data repo is used for storing all the data of each student in class, including their roster data (name, student number, etc) and their daily and overall participation logs. At any given time the static state of the data repo will contain a representation of the student data which is up-to-date, and various necessary pieces of data for the operations on that student data. This data is also written to two .tsv files to ensure consistency between class sessions. The behavior of the data repo is primarily to retrieve, change and then save the student data for the class. It is modeled in the state diagram on the next page.

Databoard Class Object
State Machine Diagram



Queue

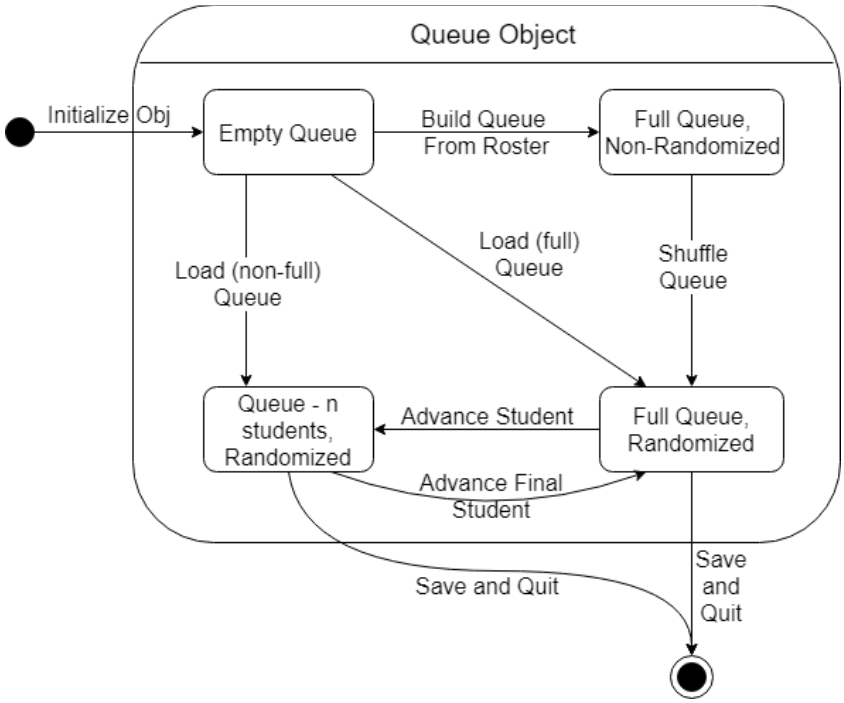


Class diagram for Queue

The Queue is tasked with randomizing the order of the students as they are called on. The queue uses a bag-based randomizer that ensures that each student will be called on once before any student is called on again- barring any student's turn being passed over a significant amount of times. At any given time, the static state of the queue will include:

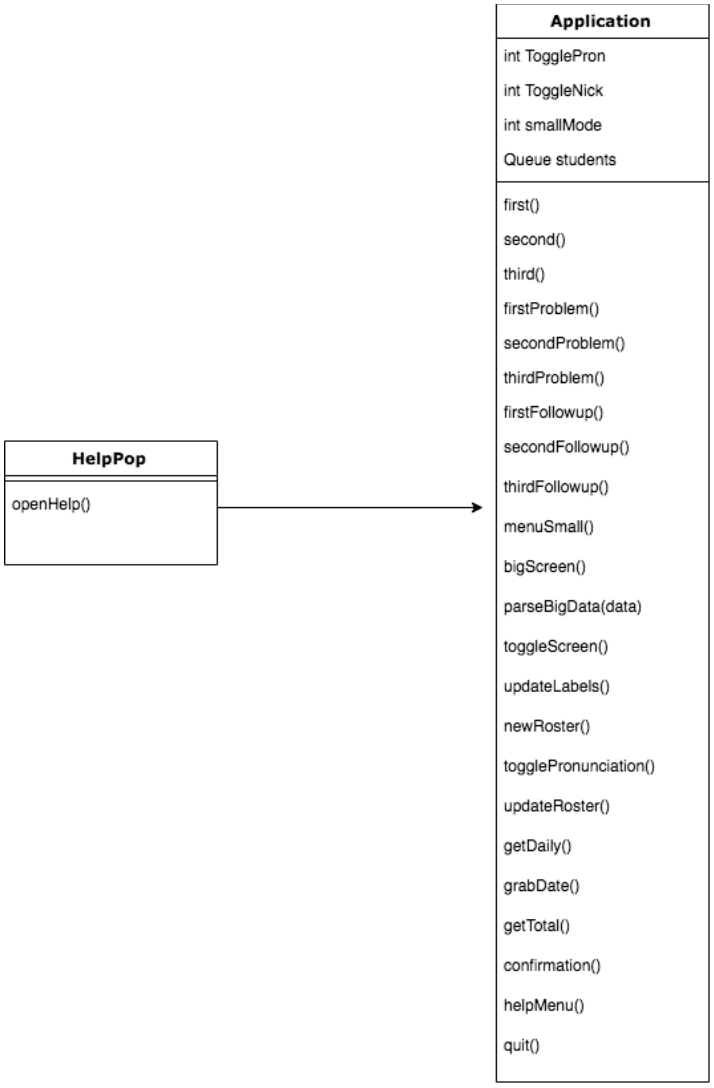
-The order of students to be called on is stored into front_bag and back_bag.
-The data that the GUI will display for the first 3 students in the queue is stored in front_students
-The data_repo object that holds and updates all data on the students and their participation.

This state diagram shows the state of the queue during normal use of the program. Any transition between Queue states in that diagram is a result of a student being called on and removed from the queue, which also contacts the data repo and tells it to store that participation data.



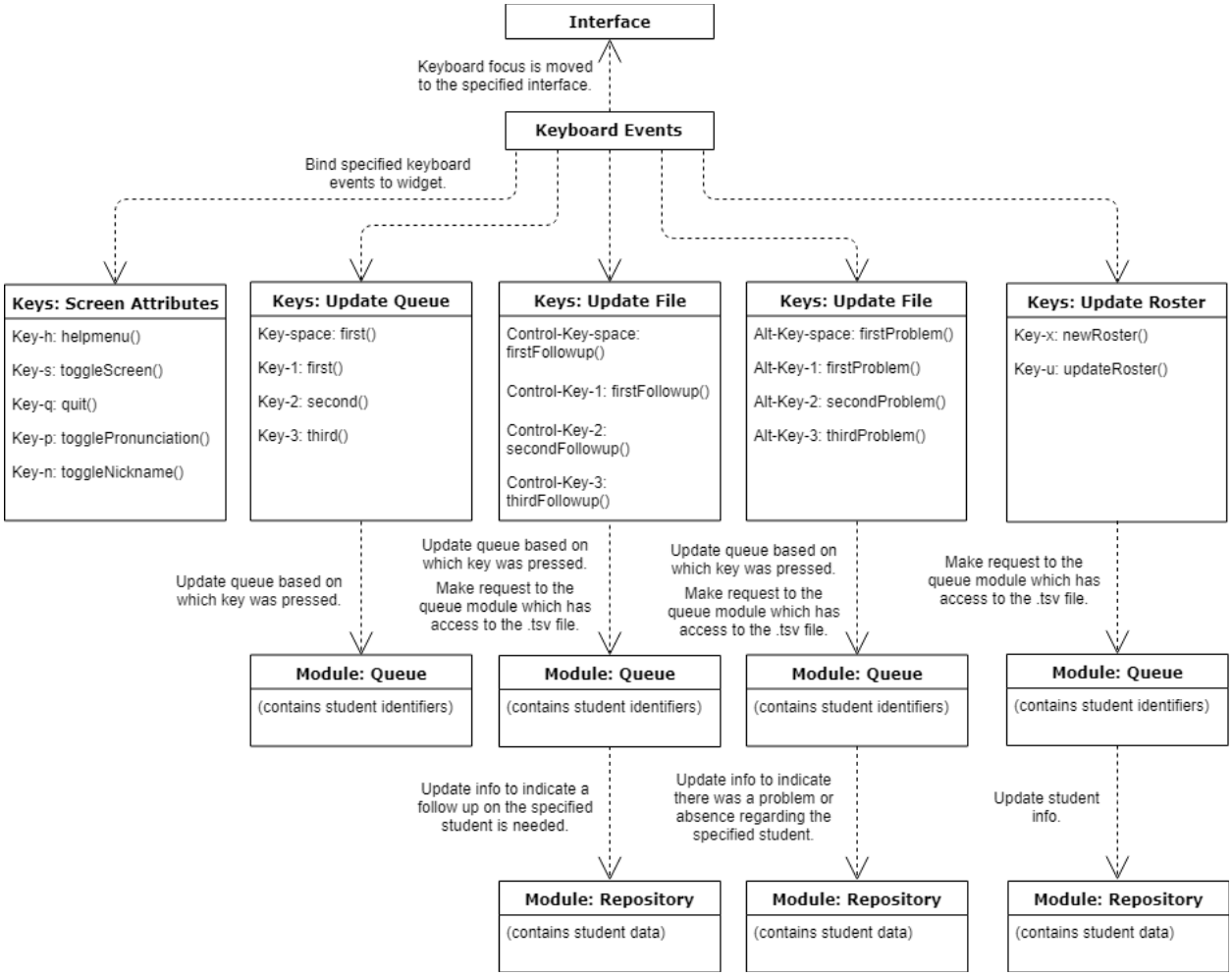
State diagram for Queue

GUI



Class diagram of the GUI class

The GUI is integral for the user to interact with the system. Any static state for the GUI will include the primary tkinter window that displays the data on the students, pair of variables for toggling display of nicknames and pronunciation and a variable to determine if the GUI is in large mode or small mode. The final part of the static state is that it holds the queue, which holds the data repo, so the GUI can be looked at as the top level of Cold Call Helper. Most of the dynamic behavior of the GUI concerns getting input from the keyboard or the tkinter menus and calling corresponding functions through the queue. This is represented in the activity diagram on the next page. Additionally, it has functions to gather data from the queue (which, in turn, gathers it from the data repo) to display it to the user, they are invoked every time user input calls a function to change the queue or data.



Activity diagram of the keyboard input

Export Data

The modules used to export files with summaries pertaining to either a specific day or the overall term, while separate, are incredibly similar. The only difference between the two is the module for exporting the text file for a daily summary requires an argument to specify which day the professor would like an overview of. However, both modules will be given data exported by the main repository. The extraction of certain fields from said repository make for another common functionality between the two. Such fields include each student's contact information as well as the total amount of times the student was called on, the total amount of times the student had difficulty answering a question, and whether a follow up on the student is necessary. Note the files exported by the two modules may vary depending on the data provided.

Design Rationale

Our rationale for choosing this design was to break it into the small modules with loose coupling, and then compose them all together into a single program. By separating into these parts, we were able to parallelize our work for the first half of the project, as each person could work on a module alone until we needed to start composing them together. As for our choice of the modules to be broken down into, we determined that each of the modules we chose was a separate problem that has easily identifiable interfaces to other modules.

The GUI needs to solve the problem of displaying data that is passed to it in a variety of formats, and to invoke functions in a class that exists under it. The GUI interfaces with the queue by getting the data it needs to display, and invoking functions in that class. The queue needs to solve the problem of randomly calling on students in a class in a fair way. The queue interfaces with the GUI by sending it the data it will need to display, and interfaces with the data repo by getting the data to send to the GUI, and routing requests the user makes through the GUI to alter the data, such as marking participation on a student. The data repo needs to solve the problem of loading in a roster file and keeping track of student data. It interfaces with the queue by giving the queue the data that it asks for.

Another advantage of this modularity is that it allows major changes to a module to improve functionality or reliability without having to change the rest of the program, as long as the interfaces stay the same. This approach seems to have worked, as none of the problems we encountered during development were related to the architecture of our software.