CoolCall: A Cold-Call Assist Program
Software Design Specification

Arden Butterfield, Quinn Fetrow, Derek Martin, Amy Reichhold, Madison Werries

January 30, 2022

1

Table of Contents

1. SDS Revision History

Date	Author	Description
1-9-2022	mgw	Created the initial document from Prof. Hornof's template.
1-10-2022	qif	Added Interface Specifications for all modules
1-10-2022	all	Finished initial submission
1-19-2022	qif	Updated Interface Specifications for all modules
1-21-2022	qif	Added sequence diagrams for Instructor Interaction Model
1-27-2022	ar	Added static and dynamic diagrams for Log Manager
1-28-2022	ab	Added Static Diagram for Instructor Interaction Model
1-29-2022	qif	Added static and dynamic diagrams for all other modules
1-30-2022	qif	Finished final SDS writeup draft
1-30-2022	mgw	Reviewed and edited final draft for submission

2. System Overview: The Cold-Call Assist Program

The general specifications described in the System Overview are derived and inspired by the document "*Develop a Classroom Cold-Call Assist Software Program"* (Hornof, 2022).

The overall product to be built will be a system designed to assist an instructor with "cold calling" on students in a classroom, focusing both on ease-of-use for instructors and fairness for the students being cold-called. "Cold calling" means requesting input (thoughts, questions, or answers to questions) from a specific student who did not raise his or her hand.

The CoolCall system is launched through a command in the Terminal. By default, 4 students will be chosen at a time to be placed "on deck," and the names of these students will be displayed to both the instructor and students via an onscreen name display which can sit on top of lecture notes, such as Microsoft PowerPoint slides.

The system will be designed to equally distribute the cold calls across all students across multiple days. The system displays the current list of students who are at the front of the queue and considered to be "on deck". The instructor uses arrow key input to navigate through the list of students and remove students from the queue as they are called on, regularly writing to log files which store information about the cold calls.

The instructor may call on any student in the visible queue and select that student by pressing the left and right arrow keys. At startup, the first student on deck is selected.

The right-arrow key moves the highlighting to the right, and it does not cycle back around to position 1. The same applies to the left-arrow key: it moves the highlight to the left, and it does not cycle back around to position 4.

Pressing the down arrow key removes a highlighted name from the on-screen list *without* a flag. Pressing the up arrow key removes a highlighted name while *raising a flag* for that removal instance. When a name is removed from the list, all of the remaining names on the list shift to the left (a new name appears at position 4).

The system will save a daily log file containing information about the day's cold calls. The top of the file will indicate that this is the daily log file for the Cold Call Assist Program. Under that heading will be the date, followed by one line for each cold-call that the instructor made that day.

3. Software Architecture

Components:

1. **Student Object -** Holds all student information
2. **Student Queue Manager -** Maintains the order of the Student objects to be called on. Manipulated by the Instructor Interaction Model, accepts student data from Student Roster Manager.
3. **Student Roster Manager -** Imports/Exports roster file, and creates Student Objects. Provides student data to the Student Roster Manager.
4. **Instructor Interaction Model -** Handles all internal logic, making calls to each module Interacts with every module, making calls based on user input.
5. **Log Manager -** Writes to output files (Daily Logs and Performance Summary File). Called upon by the Instructor Interaction Model each time a student is called on.
6. **Random Verification Manager -** Launches Random Verification mode based on keystrokes. Adds keys to Key Sequence for each user input, launches if Key Sequence matches the target sequence
7. **Key Sequence -** Keeps track of all keystrokes, manipulated and read by the Random Verification Manager
8. **Cold Call GUI Manager -** Manages the graphic user interface of the program. Data is sent from the Student Queue Manager and the Instructor Interaction Model.

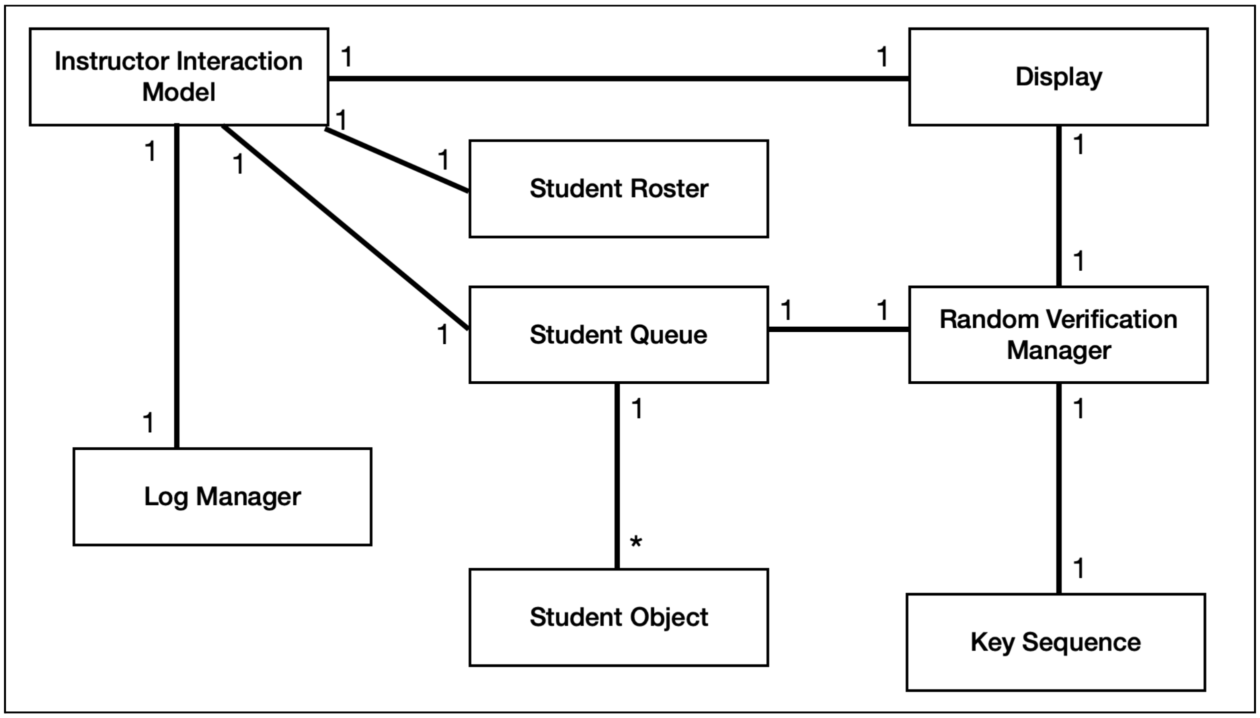

Figure 3.1: UML Abstract Class Structure of Cold Call System

This system is loosely based on the Gang of Four: Model View Controller design pattern, modified to fit the CoolCall system requirements as follows:

A. The Model being the **Student Queue Manager** (which holds instances of the **Student** object)
B. The View being the **Cold Call GUI Manager**
C. The Controller being the **Instructor Interaction Model**.

This was chosen to be the best design for the system because the main purpose of the system is to allow the user to view and manipulate a model (list of students).

Some functionalities of the system do not fit into the Model View Controller design pattern. The **Random Distribution Verification** mode is a subroutine launched based on user input, so it needs it's own module and sub-module(**Key Sequence**). The system must also write output data based on user interaction, so it must make calls to the **Log Manager.** Finally, the system must support functionality for the user to import and export a roster file, implemented by the **Student Roster Manager.**

4. Software Modules

The following documentation describes each of the CoolCall program's modules, including:

1. The module's general role and primary function.
2. The module's interface specification, which includes the module's variables, public functions, and private functions.
3. The design rationale for the module, including the alternatives considered during the development of the module's design.
4. Static and Dynamic UML diagrams to describe each module.

4.1 Student Class

The module's role and primary function:

The Student module keeps track of all student data. Instances of the student object are created by the **Student Roster Manager** and the **Student Queue Manager**. Students hold all identification information provided by the user, as well as an integer and array to keep track of how many times the student has been called on, and the dates they have been called upon

Interface Specification:

- ***Strings: first_name, last_name, UO_ID, email_address, phonetic_spelling***
- ***reveal_code* -> String *:***
  String that is used by ***include_on_deck()*** to determine if the student should be included in the queue.
- ***total_num_flags* -> Integer:**

Is incremented by **call_on()** function if boolean **flagged** = True. This integer keeps track of how many times the student has been flagged

- ***dates_called*** **-> [Date]:**
  Every time the **call_on()** function is called, the current date is appended. This date is received from the datetime module.
- ***call_on(*flag*):***
  Called from the **Student Queue Manager**, indicates the instructor has called on the student. If **flag** is true, the ***total_num_flags*** integer is incremented. The current date is appended to ***dates_called***
- ***get_name():***
  Returns a string of the first and last name of the student.
- ***include_on_deck():***
  Called from the **Student Queue Manager**, returns true if the student's **reveal_code** is equal to "0". This indicates the student should be included in the queue

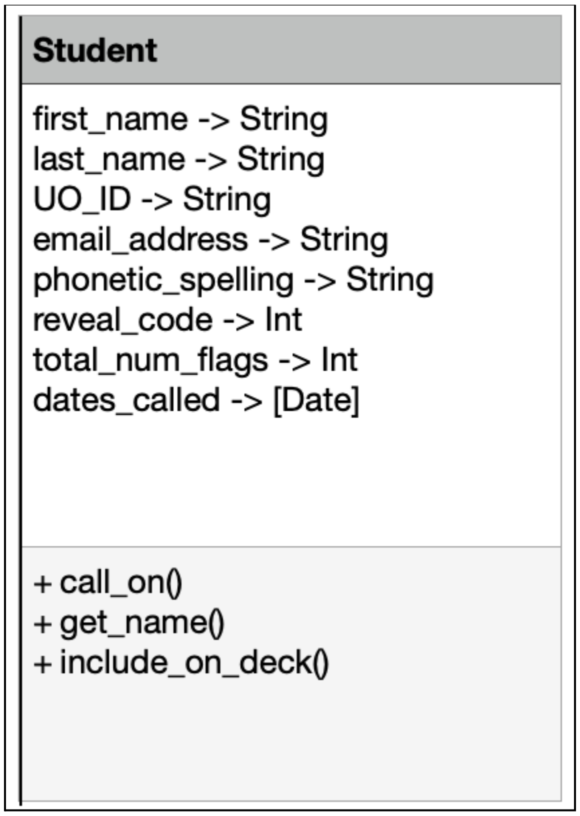

Figure 4.1.1 UML Static Diagram of Student Class

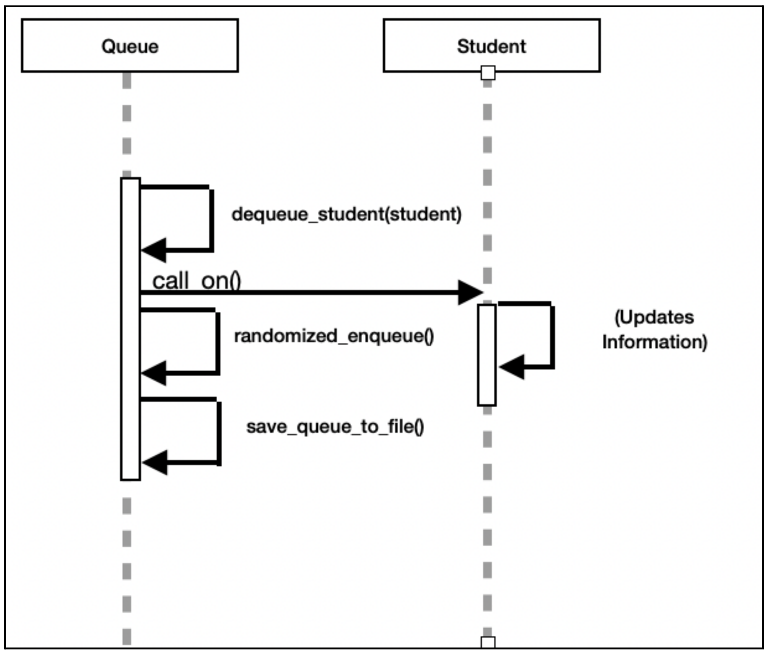

Figure 4.1.2 UML Sequence Diagram of Student Class

Design Rationale and Alternatives Considered:

The Student module was designed with the intention to hold all data. It makes the process of saving, loading, and writing information simple through providing a format that allows the

Student Queue Manager, Student Roster Manager, Cold Call GUI Manager, and Instructor Interaction Model to access and manipulate data.

One topic of debate was whether the Student module should keep track of the number of times the student was called on/flagged, or if that should be held by a dictionary within the Queue. We decided against this idea, as holding all of this information in one place makes it easier to store the data between runs.

4.2 Student Queue Manager

The module's role and primary function.

The Queue Manager loads and saves student data, managing the order in which students are placed "On Call" by the instructor. The Student Queue Manager is responsible for adding students to and dropping students from the "On Deck" list as well as randomizing the queue. The module loads an array of student objects from a previously saved file or an instance of the roster. Student objects contain information about each student. The module can perform actions that manipulate the array. These actions are performed through commands given by the **Instructor Interaction Model**.

Attributes

- ***student_queue*** **[array of student objects]**

Public Methods

- ***queue_from_roster(roster*** **-> Roster*):***
  Called from **Instructor Interaction Model** if there is no save file, generates queue from the given roster module
- ***load_queue_from_file(file_name):***
  Reads and stores array of student objects to **student_queue** from pickle file ***file_name***
- ***get_on_deck()***
  Called from **Instructor Interaction Model.** Returns the first N students in the Queue
- ***take_off_deck(student*** **-> Student*):***
  Called from the **Instructor Interaction Model,** if a name has been cold-called, the queue removes the selected name and reinserts it randomly in the bottom "n" percent of names by calling ***dequeue()*** and ***randomized_enqueue()***. Finally, ***save_queue_to_file()*** is called to store the updated queue.

Private Methods

- ***dequeue_student():***
  Called from ***take_off_deck()*** , removes a student from the queue
- ***randomized_enqueue(student*** **-> Student*):***

Called from ***take_off_deck(),*** enqueues a student back into the queue at a random index in the bottom N percent

- ***save_queue_to_file(file_name)*:**
  Self-called from ***take_off_deck(),*** saves the entire queue to a pickle file.
- ***shuffle_queue():***
  Randomizes all elements in the Queue
- ***shuffle_front_and_back():***
  Called on startup, randomizes the front and back of the queue separately

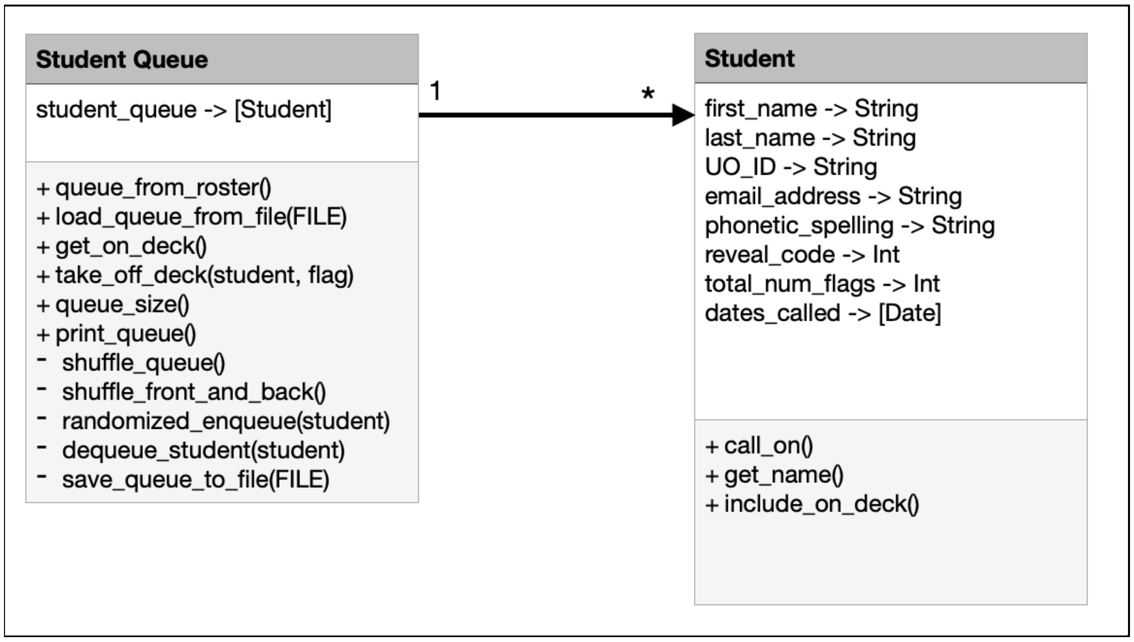

Figure 4.2.1: UML Static Class Diagram of Student Queue

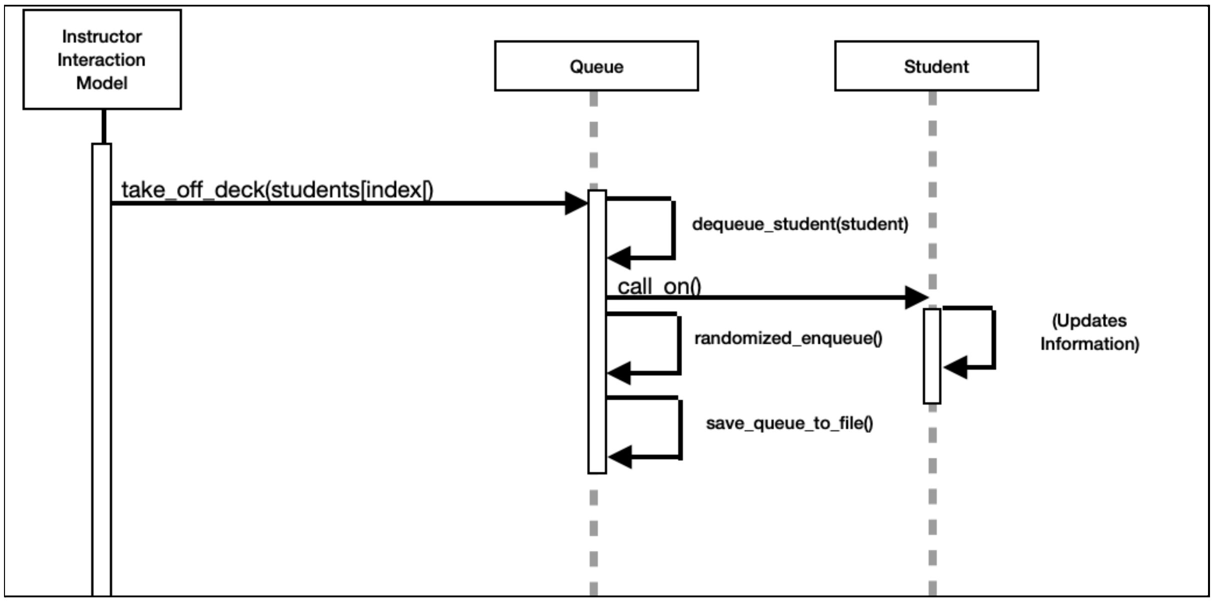

Figure 4.2.2: UML Sequence Diagram of Student Queue

Design Rationale and Alternatives Considered

The Student Queue was designed based on the Gang of Four: Model View Controller design pattern to be the Model. The Model is manipulated by the controller (the Instructor Interaction Model) and displayed by the view (the Cold Call GUI Manager).

One topic of debate when designing the Student Queue Manager was whether or not it should hold all of the student information, or if it should hold an array of IDs that map to student information within the roster. The hashmap would reduce the amount of information that would have to be saved between runs by only saving an array of IDs and not student objects. This idea was rejected for two reasons:

1. The requirements specify a student should not be mapped to personal information (UO ID/ Email)
2. Information about the amount of times the student has been called on/flagged needs to be stored internally, and the Queue holding Student Objects simplifies this process

Another design that was considered was storing the queue information as plain-text to be deciphered on startup. This was considered as it would eliminate the need for the Pickle module. This was decided against as using Pickle is faster and simpler.

4.3 Student Roster Manager

The module's role and primary function.

The Student Roster Manager collects and verifies student information from the user-created "Roster File". This module primarily functions to create arrays of student objects based on information from the imported Roster File, and send them to the **Student Queue Manager**. When an import command is called, the Student Roster Manager creates an array of student objects that will be sent to the **Student Queue Manager** to verify/use.

Attributes

- ***students:*** **[array of Student Objects]**

Public Methods

- ***import_roster_from_file(filename):***
  Called from the **Instructor Interaction Model**, after checking the file format with ***get_errors(),*** saves an array of student objects created from parsing information in the filename to **students**.
- ***export_roster_to_file(filename):***

Called from the **Instructor Interaction Model** when the user presses export button**,** creates a new file and writes data from the current roster to a new file "[filename]"

- ***compare(roster, other_roster):***
  Called from the **Instructor Interaction Model** when the user wants to upload a new roster, returns an array of Student objects containing changes that will be made to the old roster
- ***save_internally():***
  Writes the contents of the roster to an internal file
- ***get_errors():***
  Called from ***import_roster_from_file(),*** verifies that the provided input file is in correct format

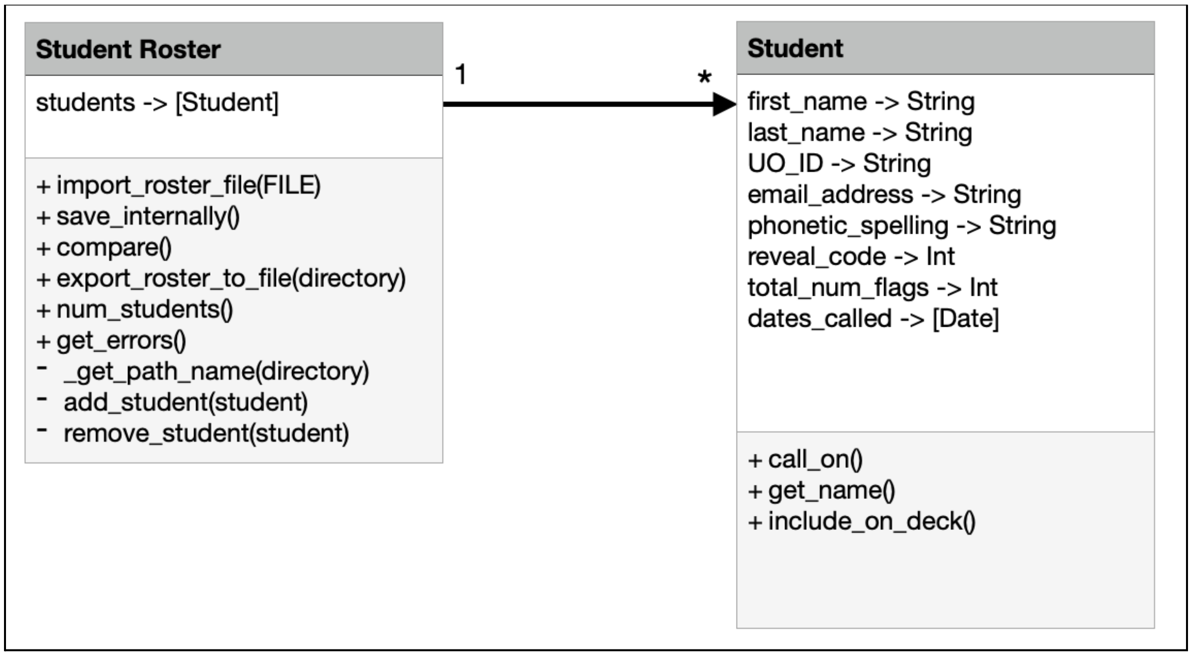

Figure 4.3.1: A Static UML Diagram of the Student Roster Module

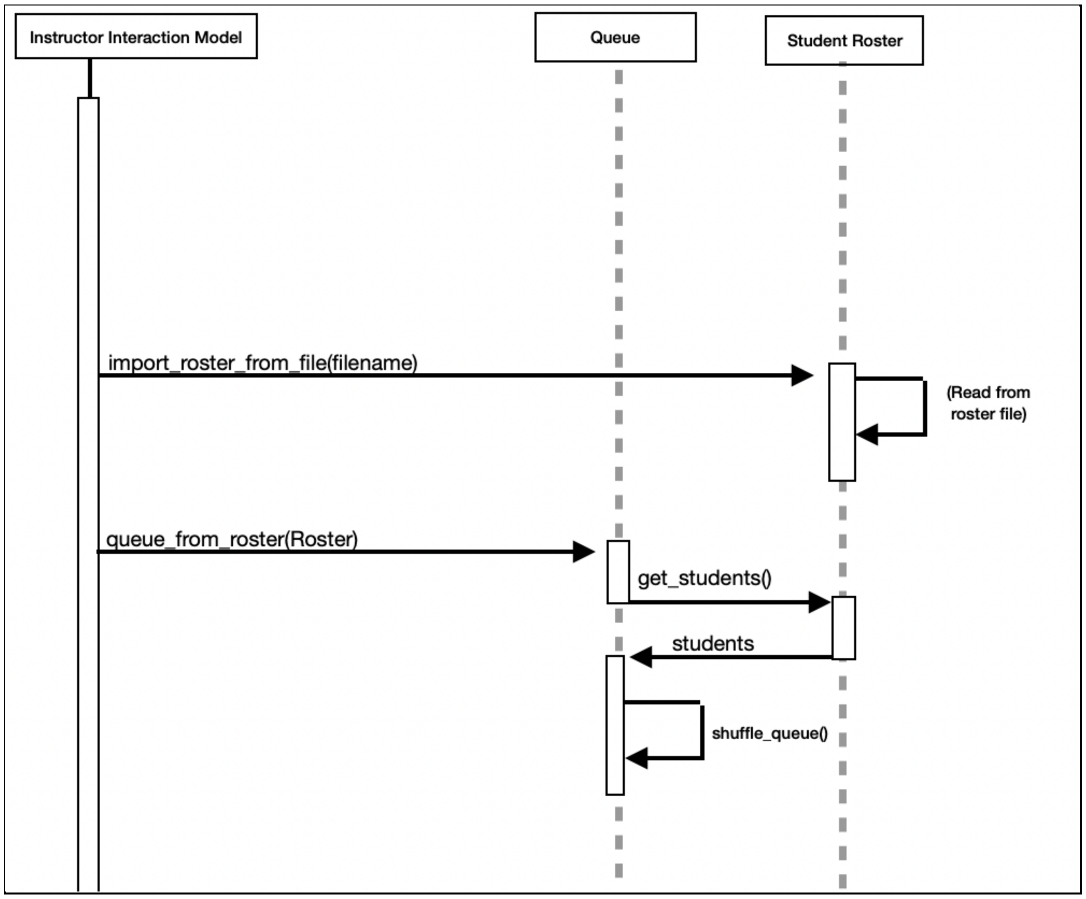
Instructor Interaction Model
Queue
Student Roster
import_roster_from_file(filename)
(Read from roster file)
queue_from_roster(Roster)
get_students()
students
shuffle_queue()

Figure 4.3.2 A UML Sequence Diagram of the Student Roster on Startup

Design Rationale and Alternatives Considered

The Student Roster Manager stores the roster internally, and serves as a buffer between the user inputted roster and the Student Queue. The reason the system is designed this way is so that if the user deletes, moves, or modifies the import file, the Cold Call program can still load the unedited internally stored file.

One alternative that was considered was merging the Student Roster Manager and the Student Queue Manager functionality. This alternate system would rely on the Student Queue Manager to read the initial file and store all student information. In this system, the roster would never be stored internally. We decided against this system as storing the roster internally simplifies the process of exporting.

4.4 Instructor Interaction Model

The module's role and primary function.
The Instructor Interaction Model controls the other segments of the project, and takes it through the various states of operation. It takes input from the keyboard and from buttons pressed on the Tkinter windows.

Attributes

- ***index -> int***
- ***roster*-> (**instance of **Roster Module)**
- ***display***-> (instance of **Cold Call Display Module**)
- ***queue*-> (**instance of **Queue Module)**

Private Methods

- ***__init__()***

  This function is called on startup. First it creates instances of the **Key Sequence, Student Roster, Cold Call Display** and **Student Queue** modules. Then it checks to see if there is already queue file, if there is, it calls ***load_queue_from_file()*** on **Student Queue Manager,** and if there is not it checks to see if there is a roster file. If there is no roster file, ***import_roster()*** is self-called. The queue is then filled with ***queue_from_roster(*Roster*).***
- ***shift_index_left()***

  This is called when the left arrow is pressed, decreases the index variable by 1. ***draw_main_screen()*** is called on the **Cold Call Display Manager** to update the display.
- ***shift_index_right()***

  This is called when the right arrow is pressed, increases the index variable by 1. ***draw_main_screen(index)*** is called on the **Cold Call Display Manager** to update the display.
- ***remove_without_flag(student-*>Student*)***

  This is called when the down arrow is pressed. It calls ***write(student, flag*=False*)*** on the **Log Manager,** and ***take_off_deck(student)*** on the **Student Queue Manager**. ***draw_main_screen(index)*** is called on the **Cold Call Display Manager** to update the display.
- ***remove_without_flag(student-*>Student*)***

  This is called when the up arrow is pressed. It calls ***write(student, flag*=True*)*** on the **Log Manager,** and ***take_off_deck(student)*** on the **Student Queue Manager**.
- ***import_roster()***

  Called when the import button is pressed, this function opens a window that prompts the user to select a file to import from. The selected file is saved to internal data, and the **Student Roster Manager** is sent the new_data(filename) command with the selected file.

If the file is formatted correctly, The **Student Roster Manager** returns an array of differences between new and old data. These differences are displayed and the user is prompted to continue the importing process.
If, however, the file is not formatted correctly, the controller opens a window telling the user that the file was not readable. At user input, it loops back to the start of this function, and opens the file selection window again

- ***export_roster()***

  Called when the export button is pressed, this function opens a window that prompts the user to select a filename and location, and calls **export_roster(**filename) on the **Student Roster Manager**

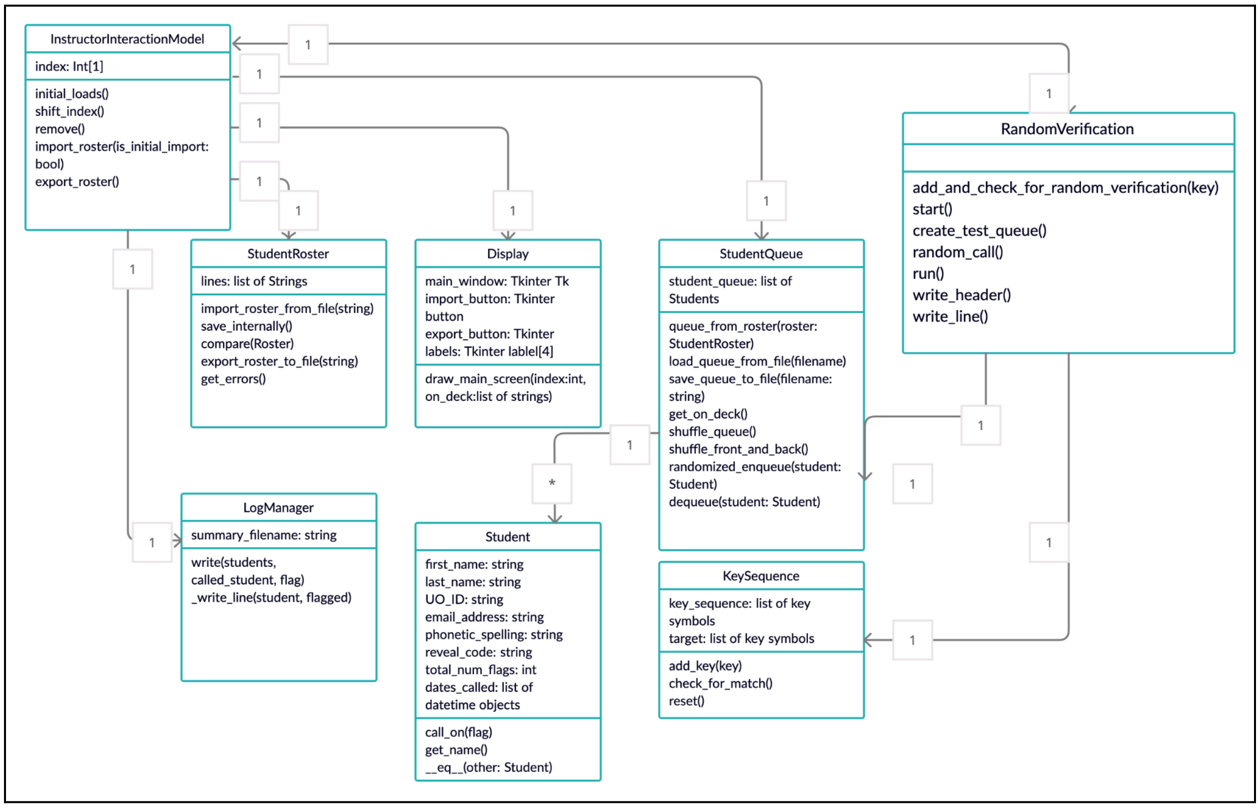
InstructorInteractionModel
index: Int[1]
initial_loads()
shift_index()
remove()
import_roster(is_initial_import: bool)
export_roster()
StudentRoster
lines: list of Strings
import_roster_from_file(string)
save_internally()
compare(Roster)
export_roster_to_file(string)
get_errors()
Display
main_window: Tkinter Tk
import_button: Tkinter button
export_button: Tkinter
labels: Tkinter lablel[4]
draw_main_screen(index:int, on_deck:list of strings)
StudentQueue
student_queue: list of Students
queue_from_roster(roster: StudentRoster)
load_queue_from_file(filename)
save_queue_to_file(filename: string)
get_on_deck()
shuffle_queue()
shuffle_front_and_back()
randomized_enqueue(student: Student)
dequeue(student: Student)
RandomVerification
add_and_check_for_random_verification(key)
start()
create_test_queue()
random_call()
run()
write_header()
write_line()
LogManager
summary_filename: string
write(students, called_student, flag)
_write_line(student, flagged)
Student
first_name: string
last_name: string
UO_ID: string
email_address: string
phonetic_spelling: string
reveal_code: string
total_num_flags: int
dates_called: list of datetime objects
call_on(flag)
get_name()
__eq__(other: Student)
KeySequence
key_sequence: list of key symbols
target: list of key symbols
add_key(key)
check_for_match()
reset()
1
*

Figure 4.4.1: A UML Class diagram of the Instructor Interaction Model and its interactions to the other classes.

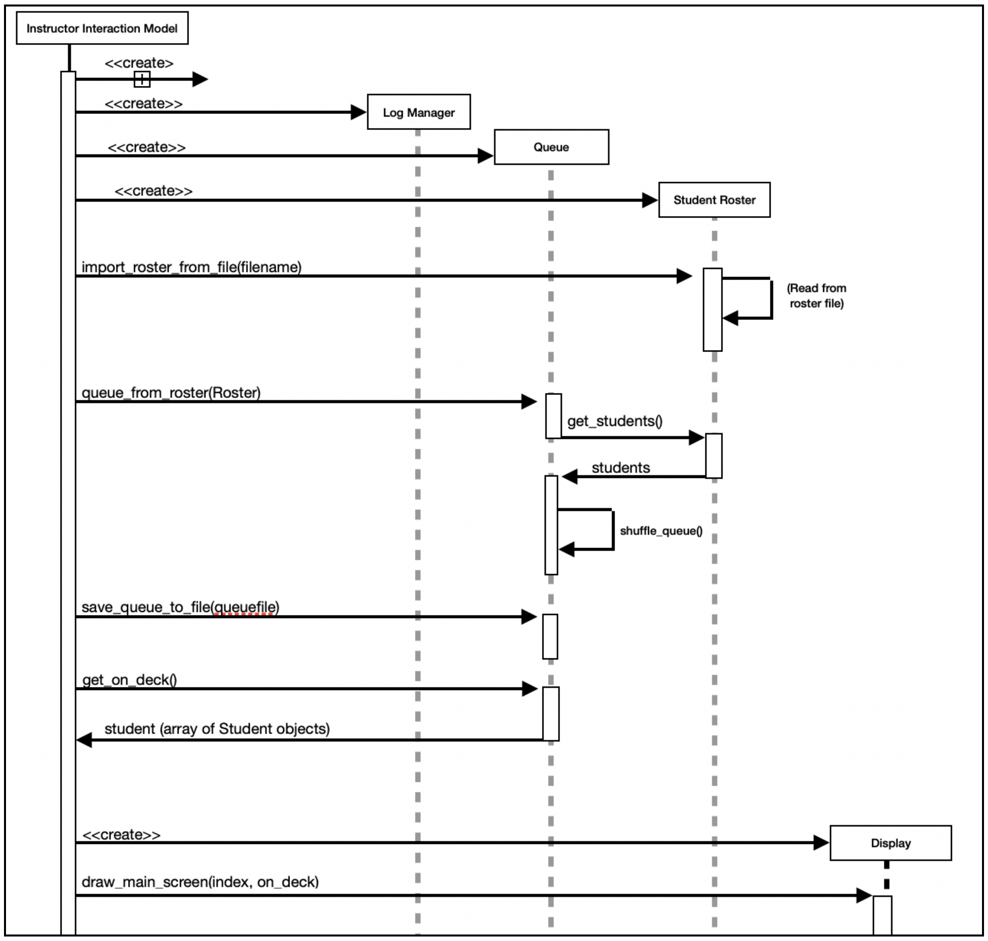
Instructor Interaction Model
<<create>
<<create>>
Log Manager
<<create>>
Queue
<<create>>
Student Roster
import_roster_from_file(filename)
(Read from roster file)
queue_from_roster(Roster)
get_students()
students
shuffle_queue()
save_queue_to_file(queuefile)
get_on_deck()
student (array of Student objects)
<<create>>
Display
draw_main_screen(index, on_deck)

Figure 4.4.2: A UML Sequence Diagram of Instructor Interaction Model on Startup

Design Rationale and Alternatives Considered

The Instructor Interaction Model integrates all other modules and handles all of the logic of the system. It is designed to be the Controller of the system, based on the Gang of Four: Model View Controller (MVC) design pattern. It manipulates information within the Model (Student Queue Manager). The Instructor Interaction Model does not strictly adhere to this design pattern however, as it sends information from the Queue to the Cold Call GUI Manager, rather than having the Cold Call GUI Manager request this information for itself.

One idea that was considered was having the Instructor Interaction Model hold an array of all of the students who are “on deck” (the first 5 students of the array). It would query the queue for the students every time anything updated, and then send them to the display module. This proposal was dismissed as it is simpler to have the queue hold a method that returns an array of the on-deck students that is called as an input for the draw_main_screen() function.

4.5 Log Manager

The module's role and primary function.

The Log Manager is in charge of creating daily log files, and maintaining the performance summary file. It records cold calls in the format <response_code><tab><first name><last name>"<"<email address>">", one per line, or in the case that there were no cold calls made that day, it will record a single line underneath the date that says "No cold calls made."

Attributes

- ***summary_filename (string):***
  The path to the summary file.

Public Methods

- ***write(students, called_student, flag)***
  Called from the **Instructor Interaction Model** each time a student is cold called. Calls ***write_line(called_student)*** to append information to the daily log file, as well as overwrites the previous summary file with a new file given updated data from the ***students*** queue. The summary file is in the format of <total times called><number of flags><first name><last name><UO ID><email address> <phonetic spelling><reveal code><list of dates> with a tab between each field and a linefeed at the end of each line.

Private Methods

- ***write_line(student, flag)***
  Appends a line of the following form <response_code><tab><first name><last name>"<"<email address>">" to the daily log file, creating the file as necessary. If the ***flag*** argument is false, the <response_code> field is blank, and if the ***flag*** is true, then the <response_code> field is the character "X" to indicate that the cold call is flagged.

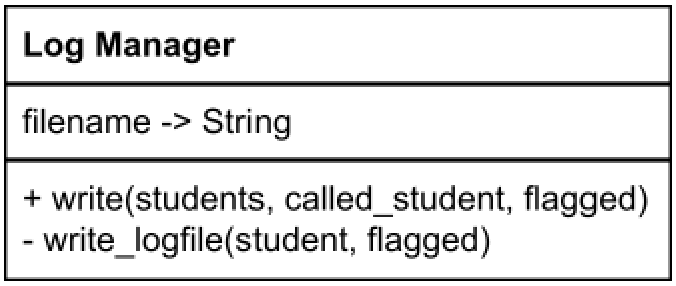
Log Manager
filename -> String
+ write(students, called_student, flagged)
- write_logfile(student, flagged)

Figure 4.5.1: UML Static Class Diagram of Log Manager

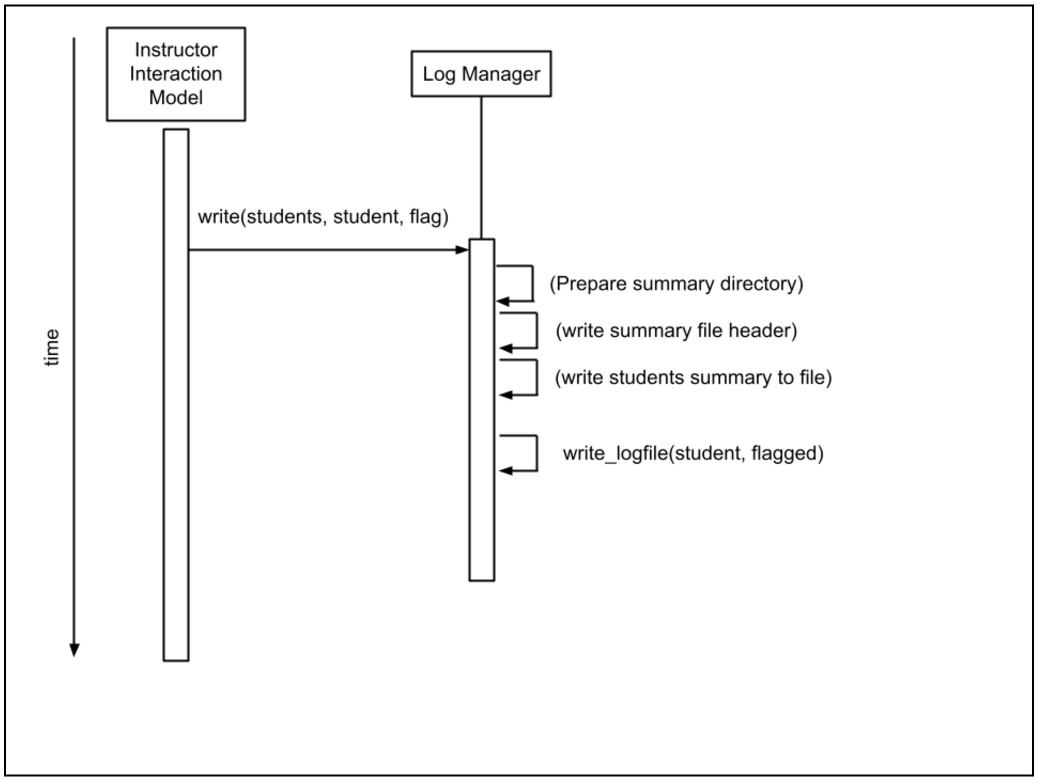
time
Instructor Interaction Model
Log Manager
write(students, student, flag)
(Prepare summary directory)
(write summary file header)
(write students summary to file)
write_logfile(student, flagged)

Figure 4.5.2: A UML Sequence Diagram of Log Manager when a student is removed from being on deck

Design Rationale and Alternatives Considered

The Log Manager was designed to be a module that handles all text output with one function call. The Log Manager was designed this way because for each call, the Daily Log and Performance Summary file must be updated, so it would be efficient to have both operations performed with one method.

The initial plan involved breaking the Log Manager into two separate modules: The Daily Log Manager and the Performance Summary Manager. The Daily Log Manager would write to the log file, and the Performance Summary Manager would write to the Performance Summary file. It was decided that it would be simpler to create a module that writes all log information with one function.

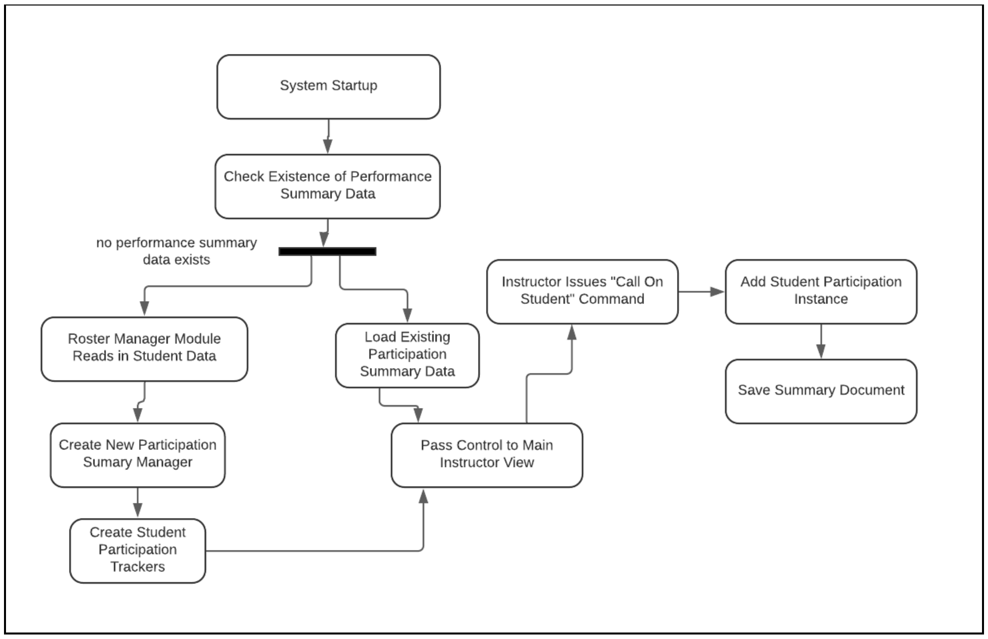
System Startup
Check Existence of Performance Summary Data
no performance summary data exists
Roster Manager Module Reads in Student Data
Load Existing Participation Summary Data
Instructor Issues "Call On Student" Command
Add Student Participation Instance
Save Summary Document
Create New Participation Sumary Manager
Pass Control to Main Instructor View
Create Student Participation Trackers

Figure 4.5.3: An alternative Design for a Performance Summary Manager

4.6 Random Distribution Verification Manager

The module's role and primary function.

The Random Verification Manager relies on the object **Key Sequence** that stores an array containing user input. Each user input is sent from the **Cold Call GUI Manager** to the Random Verification Manager. The Random Verification Manager then sends the input in string format "LEFT","RIGHT","UP", or "DOWN" to the **Key Sequence**. If the array held within the **Key Sequence** matches the target sequence, random verification mode is launched, which randomizes the queue by cold calling students 10,000 times, shuffling the queue every 100 calls. Information about each cold-call is written to an output file.

Attributes
- ***key_sequence*** **-> (instance of Key Sequence object)**
- ***test_queue*** **-> *(*instance of Queue object)**

Public Methods
- ***add_and_check_for_random_verfication(key-*>int):**

  Appends a new key to the **Key Sequence** object and checks to see if it is equal to the target sequence, if it is, ***start()*** is called.

Private Methods
- ***start():***

Prompts the user to start the program with a tkinter dialogue box. If the user accepts, calls ***write(), create_test_queue(),*** and ***run()***

- ***create_test_queue():***
  Creates an instance of **Student Queue** with information from the saved Queue file if it exists. If the Queue file does not exist, user supplied roster text file is used.
- ***run():***
  Performs 100 calls of ***random_call()*** and shuffles the queue 100 times for a total of 10,000 operations.
- ***random_call():***
  Randomly chooses a student from the queue and calls ***test_queue.take_off_deck(***chosen student). Then calls ***write_line(***student***).***

- ***write_header():***
  Called on startup, writes a header that indicates what the output file contains and when it was written

- ***write_line():***
  Writes information about each cold call to the output file.

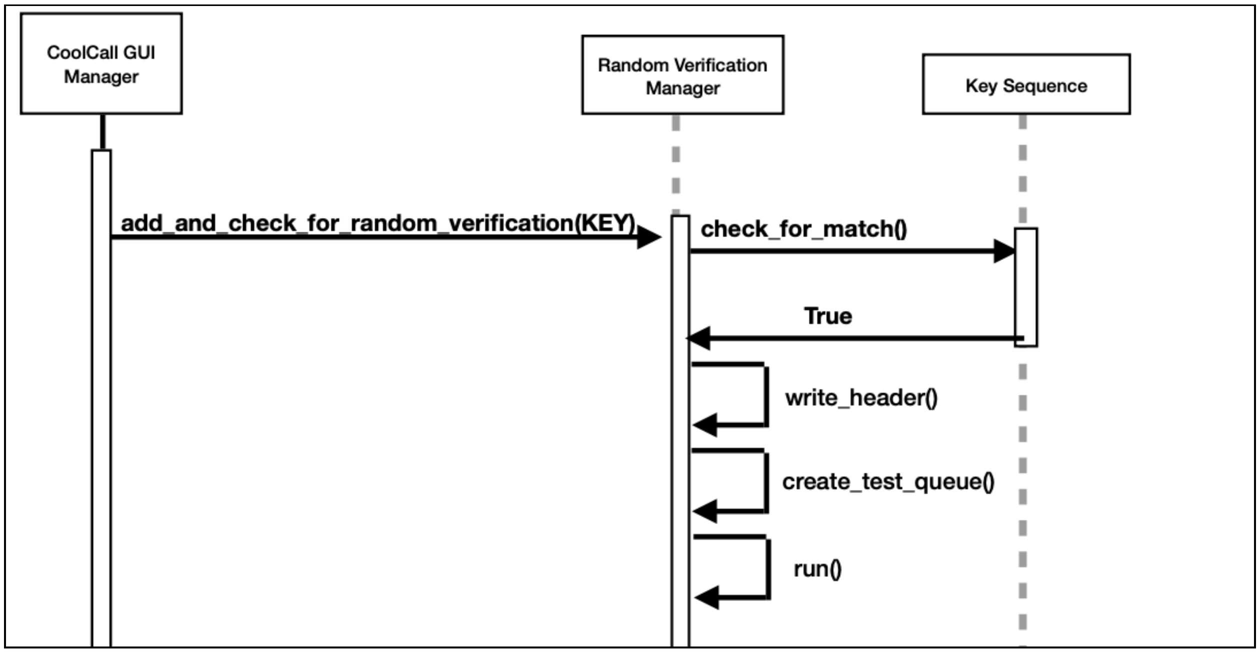

Figure 4.5.1: UML Sequence Diagram of Random Verification Manager when Target Sequence is Entered

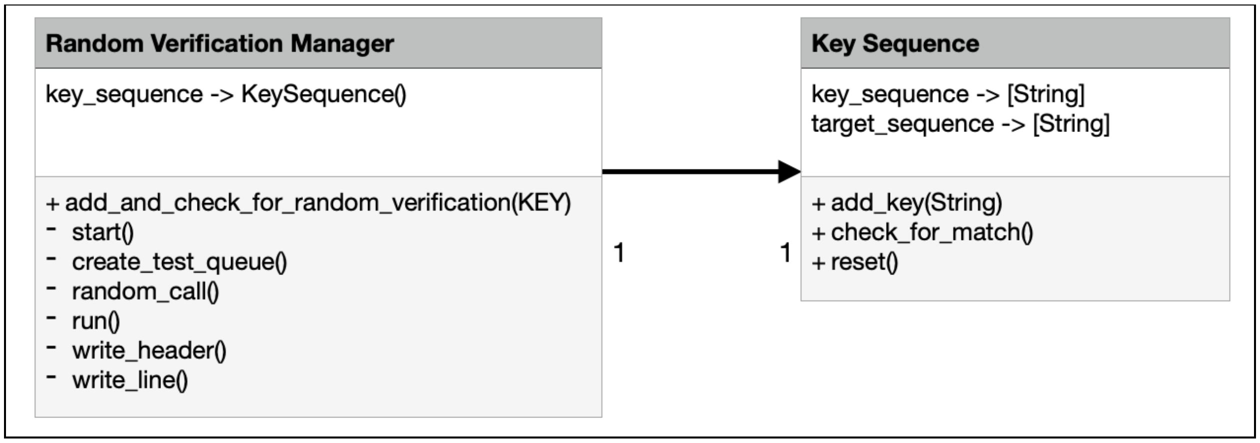

Figure 4.5.1: UML Static Class Diagram of Random Verification Manager

Design Rationale and Alternatives Considered

The Random Verification Manager is designed to prove that the system is truly randomly calling upon students. The default target of 10 left key-presses was chosen to ensure the instructor does not accidentally start the verification process.

The idea of having the random verification process be triggered via a button on the home screen was considered, but rejected as the process was intended to not be a prominent feature.

One alternative considered was having add_and_check_for_random_verification() be called from the Instructor Interaction Model instead of the Cold Call GUI Manager. This was considered because it felt unorthodox for the GUI Manager to hold an instance of something that was unrelated to the display. This was decided against as it would involve adding calls of add_and_check_for_random_verification() to each input-dependent function in the Instructor Interaction Model.

4.6 Key Sequence

The module's role and primary function.

The Key Sequence accepts input about what keystrokes are being pressed from the **Random Verification Manager.** It stores this information and checks for the target sequence (defaulted to 10 consecutive Left Key inputs but can be changed from the constants file).

Attributes

- ***target_sequence -> [String]:***
  Array that holds the target sequence of keystrokes.

- ***key_sequence -> [*String*]:***
  Array that holds strings of all keystrokes: “Left”,”Right”,”Up”,”Down”.

Public Methods

- ***add_key(*key*):***
  Called from the **Random Distribution Verification Manager,** appends the given **key** to the array.
- ***check_for_match() ->* Bool*:***
  Called from the **Random Distribution Verification Manager,** returns True if the array matches the target sequence.
- ***reset():***
  Called from the **Random Distribution Verification Manager,** resets the data in the ***target_sequence*** array.

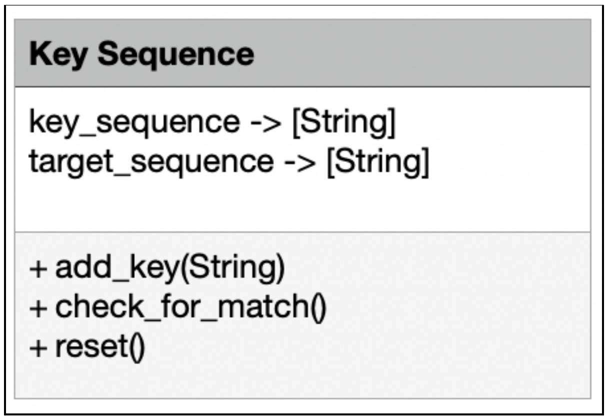

Figure 4.6.1: UML Static Class Diagram of Key Sequence

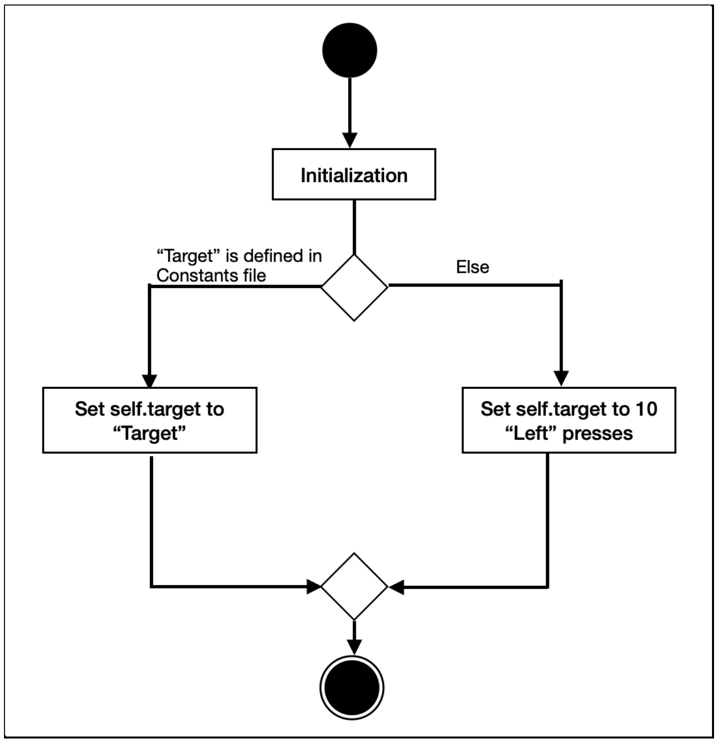
Initialization
"Target" is defined in Constants file
Else
Set self.target to "Target"
Set self.target to 10 "Left" presses

Figure 4.6.2: UML Action Diagram of initialization of Key Sequence

Design Rationale and Alternatives Considered

The Key Sequence object was designed to be able to work with any target sequence of inputs. This target sequence is defaulted to 10 "Left" key presses, but can be changed by the user in the global constants file.

One alternative considered was having the key sequence hold an integer that represents the amount of "LEFT" key presses. This integer would be incremented for each "LEFT" key-press, and reset to zero for any other key_press. This system was considered because it would eliminate the need to compare arrays after each input. This idea was rejected because it only functions if the target sequence is the default sequence (10 Left key-presses), and does not support the user being able to change the target sequence in the CONSTANTS file.

4.7 Cold Call GUI Manager

The module's role and primary function.

The Cold Call Display Manager is in charge of updating the display based on changes in the system. It relies on calls from the **Main Controller** that indicate which student is selected, and which students are on deck.

Interface Specification

Attributes

- ***rdv* -> Instance of Random Verification Manager**
- ***main_window* -> TKinter Widget Object**
- ***import_button* -> TKinter Button Object**
- ***export_button* -> TKinter Button Object**

Public Methods

- ***draw_main_screen(index*->int, *on_deck_students*->[Student])**

  Called from the **Main Controller** any time the User interacts with the system. Updates the display with the first names of the list of ***on_deck_students*** as well as highlights the name of the student at ***index.***

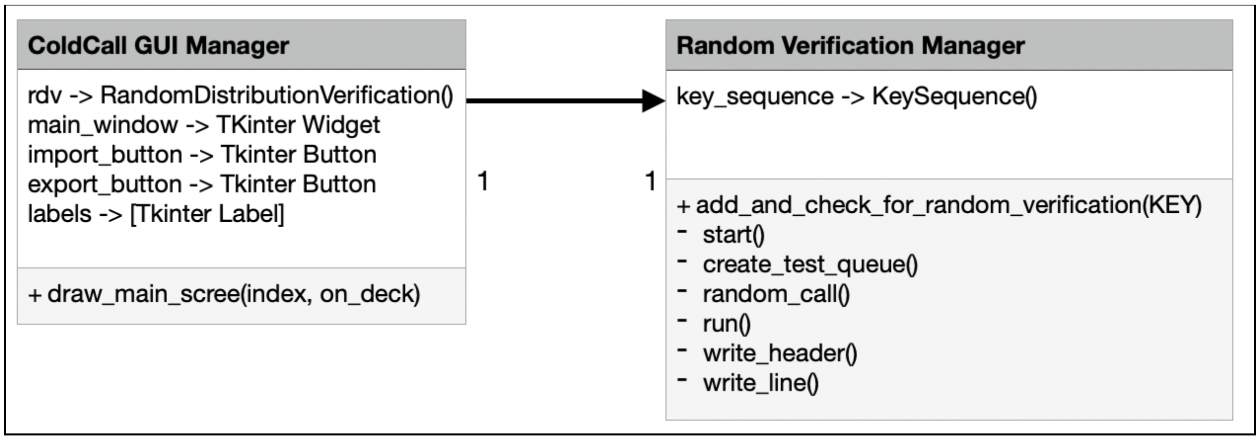
ColdCall GUI Manager
rdv -> RandomDistributionVerification()
main_window -> TKinter Widget
import_button -> Tkinter Button
export_button -> Tkinter Button
labels -> [Tkinter Label]
+ draw_main_scree(index, on_deck)
1
1
Random Verification Manager
key_sequence -> KeySequence()
+ add_and_check_for_random_verification(KEY)
- start()
- create_test_queue()
- random_call()
- run()
- write_header()
- write_line()

Figure 4.7.1: UML Static Class Diagram of ColdCall GUI Manager

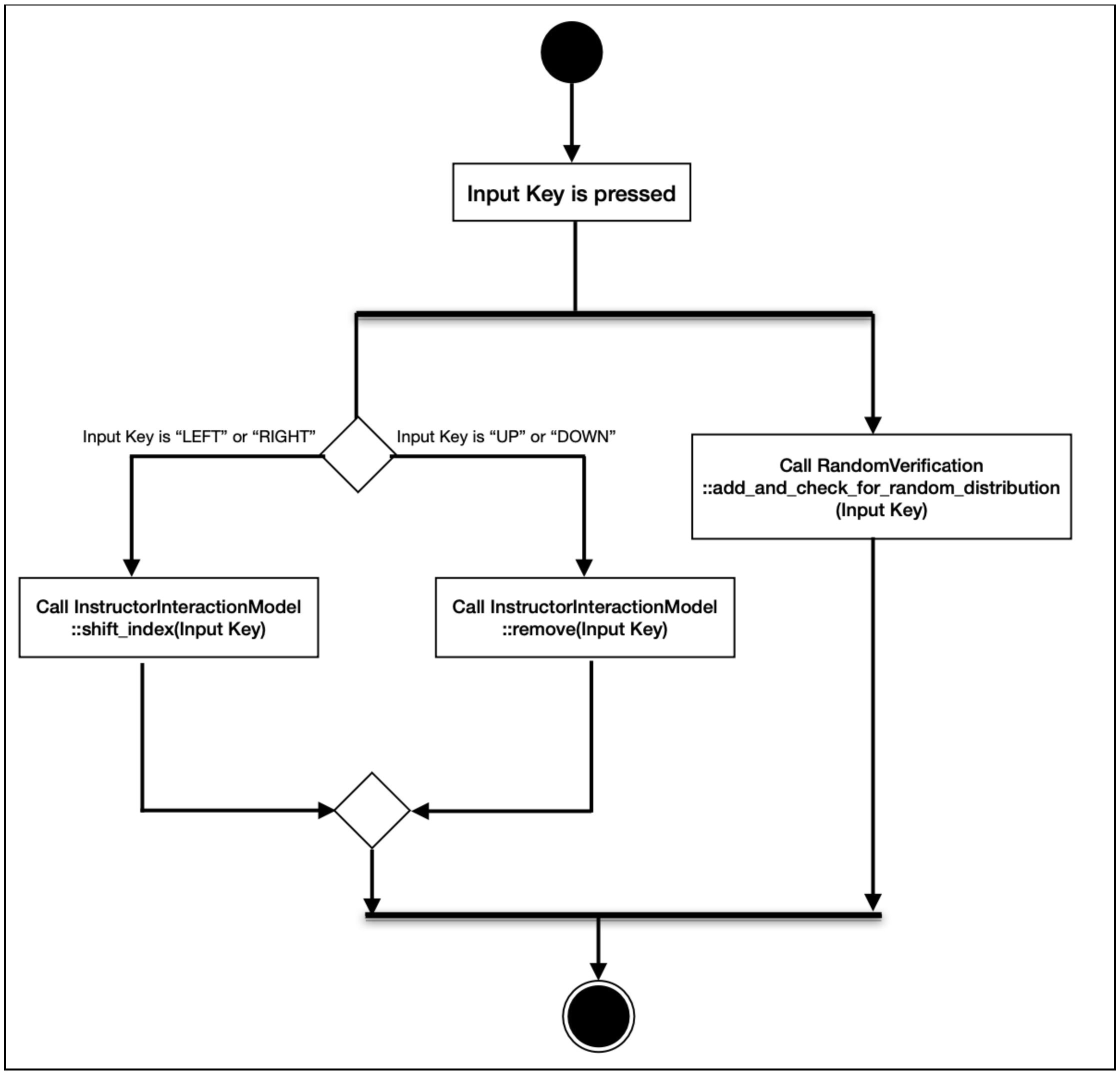
Input Key is pressed
Input Key is "LEFT" or "RIGHT"
Input Key is "UP" or "DOWN"
Call RandomVerification ::add_and_check_for_random_distribution (Input Key)
Call InstructorInteractionModel ::shift_index(Input Key)
Call InstructorInteractionModel ::remove(Input Key)

Figure 4.7.2: A UML Action Diagram of Cold Call GUI Manager upon Receiving Input

Design Rationale and Alternatives Considered

The Cold Call GUI Manager was designed to be the "View" of the Gang of Four: Model View Controller design pattern. It displays the on deck students through a window that stretches across the user's screen. This "thin line" design was chosen to minimize the amount of space that the program takes up on the screen, as the user will most likely have the system running alongside many other applications. Thin windows that stretch across the entire screen seem to fade into the background.

One alternative that was considered was having all of the functionality of the Cold Call GUI Manager be done by the Instructor Interaction Module. This was considered as most of the functions of the Instructor Interaction Module depend on user input, which is accepted by the TKinter widget. This idea was dismissed because using the TKinter function "bind_all()" binds user input to functions from any module.

5. Dynamic Models of Operational Scenarios

5.1 Roster File is Imported

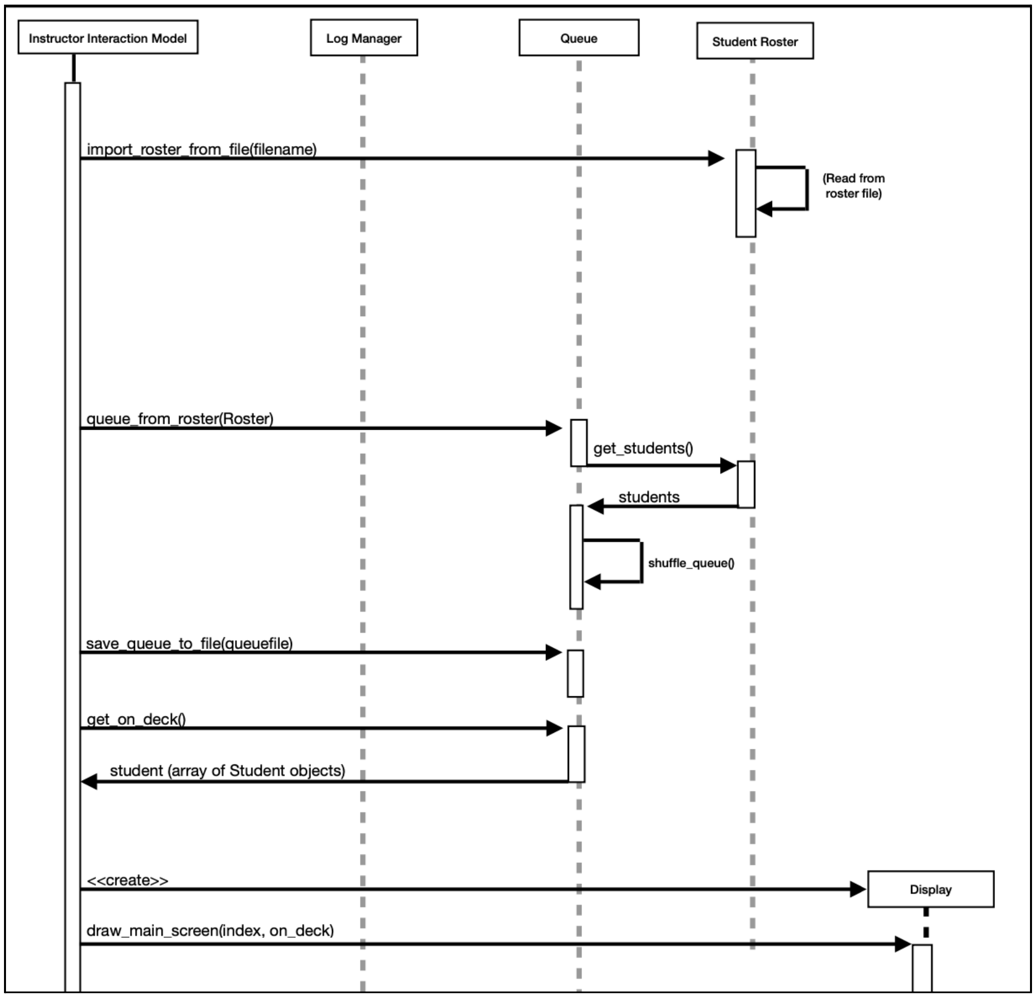
Instructor Interaction Model
Log Manager
Queue
Student Roster
import_roster_from_file(filename)
(Read from roster file)
queue_from_roster(Roster)
get_students()
students
shuffle_queue()
save_queue_to_file(queuefile)
get_on_deck()
student (array of Student objects)
<<create>>
Display
draw_main_screen(index, on_deck)

Figure 5.1.1: UML Sequence Diagram of the System when a Roster File is Imported

5.2 A Student is Called On

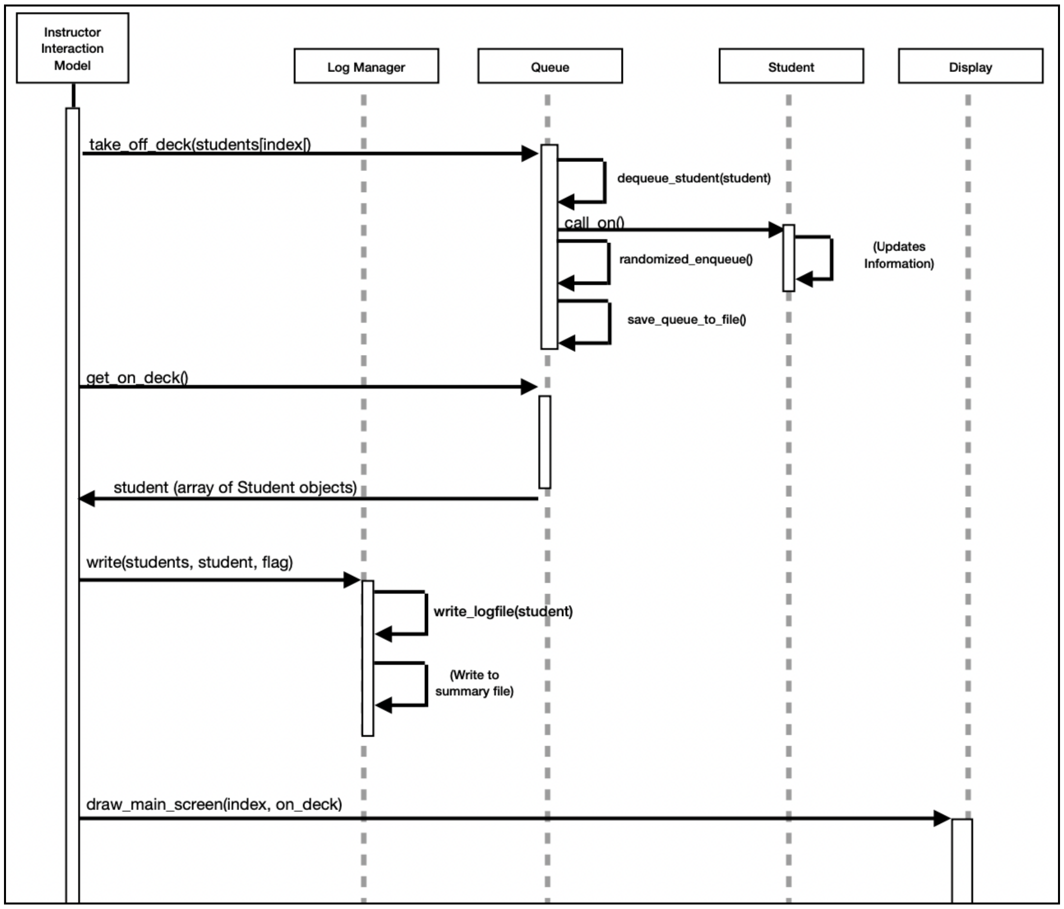

Figure 5.2.1: A UML Sequence Diagram of the System when a Student is Called

5.3 Random Verification Mode is Launched

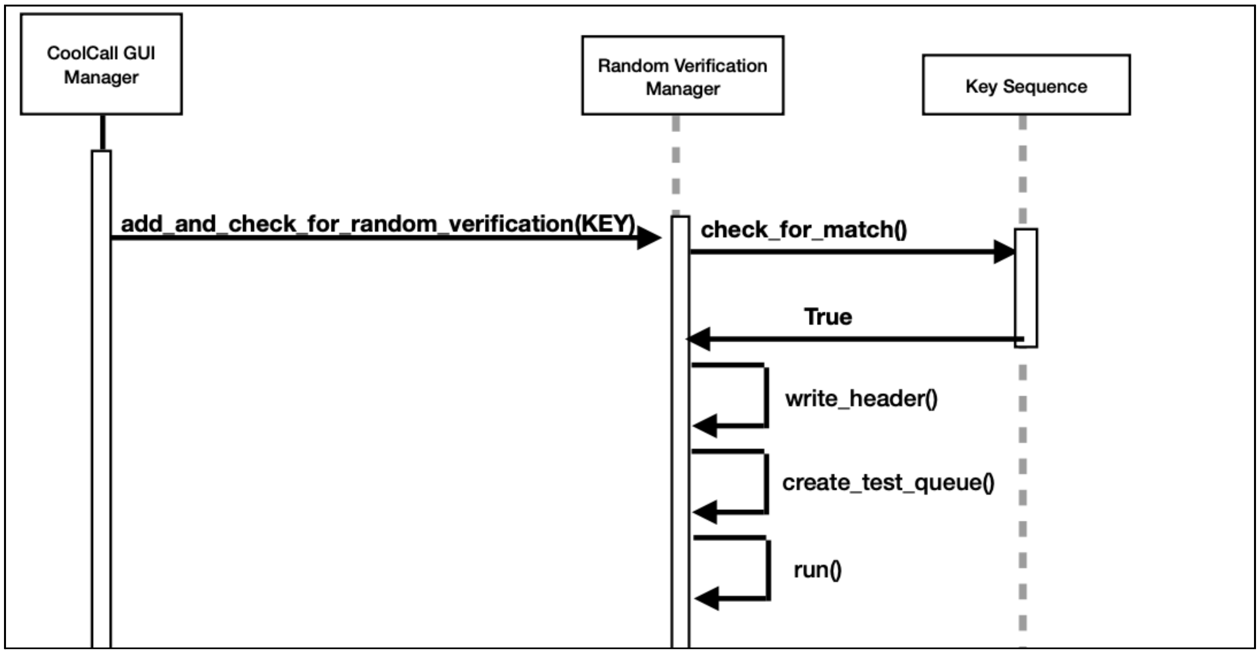

Figure 5.3.1: A UML Sequence Diagram of the system when Random Distribution Verification Mode is Launched (Same as Figure 4.5.1)

6. References

Faulk, Stuart. (2011-2017). CIS 422 Document Template. Downloaded from https://uocis.assembla.com/spaces/cis-f17-template/wiki

Hornof, A. (2019). CIS 422 Software Design Specification Template. Available at: https://classes.cs.uoregon.edu/22W/cis422/Handouts/Template-SDS.pdf

Hornof, A. (2022). Project 1 - Develop a Classroom Cold-Call Assist Software Program. Available at: https://classes.cs.uoregon.edu/22W/cis422/Handouts/Cold_Call_System_SRS.pdf

Kieras, David (Fall 2019) UML Notation Template. Downloaded from http://websites.umich.edu/~eecs381/ in 2022.