CIS 422 Project 1:
Classroom Cold-Call Assist Software
Software Design Specification

Bethany Van Meter (bvm), Mikayla Campbell (mc), Joseph Goh (jg), Olivia Pannell (op), and Ben Verney (bv)

January 14th, 2020 - v0.1

Table of Contents Change page numbers - FIXME

1. SDS Revision History

Date	Author	Description
1-14-2020	mc	Created the initial document.
1-14-2020	mc	Started writing sections for v0.1

2. System Overview

2.1. Description of System

This system will provide an instructor with the basic capabilities of "cold calling." "Cold calling" is a method in which an instructor calls on a student who hasn't recently participated in the class's discussion. This software will also provide the students with a "warm up period." This is established through a queue that displays the top 4 students that may be "cold called"

next. The instructor will also have the capability to remove students from the queue when they have participated in class discussion and flag students they want to contact after class. The main goal of this system is to "cold call" on students efficiently in order to improve class discussion and overall participation. This software also has the capability of aiding an instructor with learning their students names through flashcard mode that is styled much like Quizlet. Each flashcard has the student's picture on one side and their name on the other side.

2.2. System Organization and Interaction

The system is divided into three functionalities: "cold calling," name flashcards, and emailing students. These three functionalities are broken up into a total of 8 files.

1. First file (main.py): ties the program together and is the main program file that runs the overall program.
2. Second file (io.py): reads and parses the class text files.
3. Third file (ui.py): contains the user interface graphics.
4. Fourth file (create_queue.py): contains the algorithm that sorts the students in a class into a queue that is used to cold call the students.
5. Fifth file (save_queue.py): saves the state of the queue and its data when the program is either paused or exited.
6. Sixth file (email.py): gives the professor the ability to keep track of students they want to email and allow them to email them.
7. Seventh file (learn_names.py): holds contents necessary for the user to learn the students names much like the format of Quizlet.
8. Eighth file (controls.py): holds and creates all the controls of the program. Such as removing a student from the queue after they have spoken and flagging a student to email later.

These eight files we split into four steps. During the first step the files io.py, ui.py, and create_queue.py will be created. During the second step we the files save_queue.py and controls.py will be created. During the third step the main.py will be created. Lastly, during the fourth step the files email.py and learn_names.py will be created.

The system is split into these steps in order to account for functionality, ease of testing, and priority. The first step implements the basic individual functions of the program. These are established first so that the smaller features can be tested individually. The second step implements the more advanced attributes that build off of the first step and sets up the user controls for the program. This allows the control of the smaller functions to be tested before the program is entirely linked. Finally, the third step connects the individual functionalities into one connected program. This aspect was saved to be implemented closer to the end of the project creation in order to first allow the operation of more minor components. The final step saves the least important functionalities for last in case there isn't enough time to implement each aspect of the program.

3. Software Architecture

Software Architecture

The description of the software architecture should capture and communicate the important design decisions regarding how the system is decomposed into parts, and the relationships among those parts (Faulk, 2017). The architecture should describe:

1. The set of components. This should be both in easy-to-read list form, and also in a diagram.

2. The functionality provided by (or assigned to) each component. This should be included in the list of components.

3. Which modules interact with which other modules. Describe how the components work together to achieve the overall system functionality. This should be indicated in the architectural diagram, and also described in brief paragraph form after the list of components.
   This should be at a level of abstraction that you would use to explain to a colleague how the system works. It should be abstract and static, such as "The client database holds all of the client information and interacts with the data-cleaning module to ensure that no sensitive data gets released through the online system...." It should not be a detailed textual description of a dynamic flow of control such as "module A passes the record to module B which removes the user ID and then passes the record to module C...."

4. The rationale for the architectural design. This should be in paragraph or list form. Explain how and why this architecture was decided to be the best to solve the problem. See “Design Rationale” in the next section.

Do not keep your architecture simple just to reduce the number of modules you need to describe later in the SDS. If your architecture has only one module, your project may be too small for the purposes of an SDS, or too small for a class project, or there may be a problem with your design.

Use descriptive names for components.

Every module should have a name that is specific to the project. Do not use generic names such as "User Interface", "Model", "View", "Controller", "Database", "Back end", or "Front end". Instead, use names specific to the functionality of this system such as "Instructor Interface", "Student Interface", "Roster", "Student Records", "Roster View", "Grade View", and so on. Every module name should in some way convey the module's role *in this project*, not the role in a generic software design.

Do not name modules "client" or "server". The roles of client and server are relative to a particular service. A component can be a server with respect to one service and client with respect to another. (Faulk, Young)

4. Software Modules

4.1. <Module Name> (Include one subsection for each module.)

Each module should be described with:

a. The module's role and primary function.
b. The interface to other modules.
c. A static model.
d. A dynamic model.
e. A design rationale.

The module's role and primary function.

Every module needs to be described first abstractly in terms of its function or role in the system, and then in more detail such as in terms of its data structures and functionality. When describing such details, lists and diagrams will probably provide be more readable and searchable (for the eyes) than paragraphs of text.

Interface Specification

A software interface specification describes precisely how one part of a program interacts with another. (Faulk, Young) This is not the user interface, but instead a description of services such as public methods in a class, or getters and setters. Describe the software interface that each module will make available to other software components, both internal and external. This will help to explain how components will interact with each other, what services each module will make available, how to access a module, and what needs to be implemented within a module.

Find the right abstraction for each interface specification. For example, describe the services that will be provided but not how they will be implemented within the module. An interface that reveals too much information is *over-specified* and limits freedom of implementation. (Faulk and Young, 2011)

A Static and Dynamic Model

Every software module should be described with a static or dynamic model, and probably both.

Each diagram should use a specific design *language*. There are enough languages that have been developed such that you will not likely need to devise your own. Most diagram languages emphasize a static or dynamic representation, and do not typically mix the two. For example, class diagrams are primarily static, though they hint at time-based dynamic activities in their

method names. Sequence diagrams emphasize activities over time, but also list the static entities (such as the classes) that are interacting. But you do not typically see a dynamic activity indicated on an association between two classes in a class diagram.

Every diagram should be introduced with a caption that appears immediately below the diagram and should what is show in in that diagram. Each caption should starts with "Figure <x>." and be referenced in the body of the text as "Figure <x>." The caption should briefly describe what the diagram shows, such as "Figure 3. A sequence diagram showing the 'Feed a child' use-case."

Diagrams can be hand-drawn and scanned in. There are some advantages to hand-drawn diagrams such as they are in some ways easier to modify. But they should ideally be scanned-in rather than photographed by hand, in part to keep the file sizes down. Do not fill an SDS with large (>1MB each) high-resolution photos of hand-drawn diagrams.

Design rationale

There will be a reason that each module (and the architecture) is designed as it is. The "design rationale may take the form of commentary, made throughout the decision process and associated with collections of design elements. Design rationale may include, but is not limited to: design issues raised and addressed in response to design concerns; design options considered; trade-offs evaluated; decisions made; criteria used to guide design decisions; and arguments and justifications made to reach decisions." (IEEE Std 1016-2009) Your design should find a good separation of responsibility for each module. One design rationale required by the IEEE standard (1016-2009) is "a description of why the element exists, ... to provide the rationale for the creation of the element."

Alternative designs

Your SDS for this class should include alternate designs that were considered for the architecture, and for each system or subsystem. This should result from either (a) considering multiple alternative designs considered during the project or (b) your design evolving over the course of the project.

These alternative design ideas and diagrams can be placed in a separate section entitled "Alternative Designs" or "Earlier Designs", or could be placed immediately after each current design, and clearly marked as an alternative or earlier design.

This would not be part of a typical SDS but is included here to reinforce the idea that design is a process.

If you do not consider alternatives, you are not doing design. If you are not recording the alternatives that you consider, you are not engaged in a good design practice.

Describe alternative architectural decisions that were considered. (If there were no alternatives, then no "design" work was done.)

5. Dynamic Models of Operational Scenarios (Use Cases)

There are three uses for this system. These uses include "cold calling" as defined previously, learning the names of an instructor's students, and emailing students to the system.

5.1. "Cold Calling" Use Case

This use case is used to "cold call" on students

1. User opens application
2. User interface is displayed
3. User selects class to "cold call"
4. System loads the class's data and the class's past queue information
5. User starts the queue
6. The "cold calling" feature is now active
7. Queue is displayed
8. User removes and flags students
9. User ends system
10. Queue data is saved
11. Email data is created
12. User can now restart queue, close application, learn student's names, or email students

5.2. Learning Names Use Case

This use case is used to learn the names of an instructor's students

1. User opens application
2. User interface is displayed
3. User selects class
4. System loads the class's data and the class's past queue information
5. User opens students flashcards
6. System loads class's flashcards
7. User flips, skips, and removes and flashcards from deck
8. User ends system
9. User can now restart queue, close application, learn student's names, or email students

5.3. Emailing Students Use Case

This use case is used to email students

1. User opens application
2. User interface is displayed
3. User selects class
4. System loads the class's data and the class's past queue information
5. User opens email feature

6. System loads flagged students
7. Flagged students are displayed
8. User emails students
9. User ends system
10. Email data is updated
11. User can now restart queue, close application, learn student's names, or email students

6. References

Faulk, Stuart. (2011-2017). CIS 422 Document Template. Downloaded from https://uocis.assembla.com/spaces/cis-f17-template/wiki in 2018. It appears as if some of the material in this document was written by Michal Young.

Hans van Vliet. (2008). *Software Engineering: Principles and Practic*e, 3rd edition, John Wiley & Sons.

IEEE Std 1016-2009. (2009). IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. https://ieeexplore.ieee.org/document/5167255

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), 1053-1058.

7. Acknowledgements

This template builds slightly on a similar document produced by Stuart Faulk in 2017, and heavily on the publications cited within the document, such as IEEE Std 1016-2009.

The project creation process is based off of ideas found in the textbook: Hans van Vliet. (2008). *Software Engineering: Principles and Practic*e, 3rd edition, John Wiley & Sons.