Avi Lance, Isaac Perkins, Jaxon Simmons, Aleks Stevens
https://github.com/Avi-Lance/Painted-Penguins-Stegord

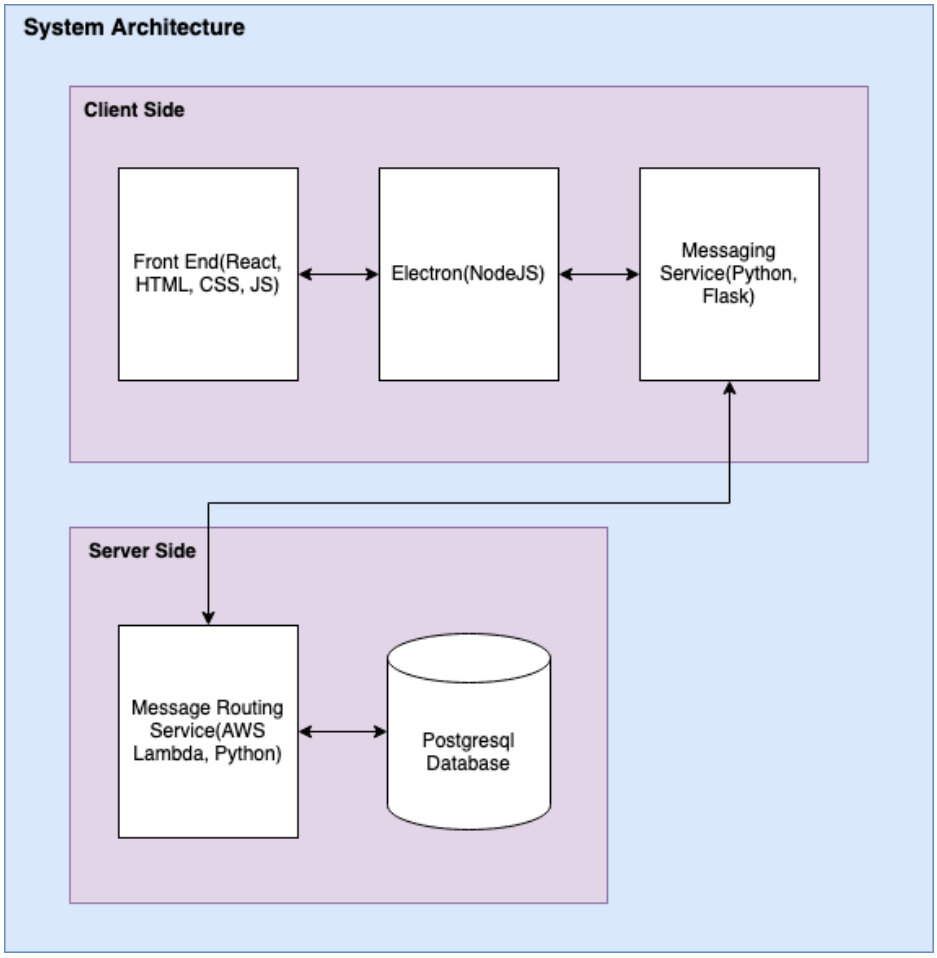
This lists every modification to the document. Entries are ordered chronologically.

Date	Author	Description
5/14/2023	Aleks S.	Created template and first draft of the SDS
5/14/2023	Jaxon S.	Added System Overview Section
5/19/2023	Isaac P.	Added reference, contributors, and repo link. Change to standard font (Times New Roman).
5/19/23	Jaxon S.	Finished Alternative Designs and Dynamic Models of Operation sections.

Digital communication is currently the fastest and most efficient way to send information long distances. However, sending and receiving data, especially messages through the internet can be

dangerous for both an individual or a company. The intention of Stegord is to provide people with secure digital messaging by utilizing steganography to encrypt messages. Not only will users be reassured that their messages and files will be secured in transit, it will also be accompanied by a very simple and easy to use interface. Users of Stegord will create their account, add friends, and start sending messages.

Stegord consists of 4 named components: User account information, a login screen, friend screen to add friends, messaging screen to message friends and groups. The login screen is meant for new or existing users to set up and use accounts. Account information is for users to create a username, add a profile picture, bio, etc. Existing users can use the login screen to access previously created accounts and the user information and activity stored with it. The friends screen is where users will connect with people through friending them. Then the messaging system is where users will send messages to friends and/or groups of friends, which will be encrypted using steganography.



1. Components
    a. Client Side Components
        i. Front End - This is all of the software to design the look of the chat application without regards for functionality. This is written in React with Typescript, Redux, HTML, CSS, and Javascript.
        ii. Electron Component - This is the middleman software between the functional backend(ie Messaging Service Component) and the front end. That is, electron has access to operating system functionality where the Front End does not, so it can talk to the messaging service(which is running as a python daemon) on behalf of the front end, then relay any information back to the front end for display.
        iii. Messaging Service Component - This is a python daemon that exposes a Flask API in order to allow interprocess communication between itself and the Electron Component. This component is responsible for talking to the server side where it fetches any received messages and sends any messages to be sent. In addition, the Messaging Service Component is responsible for decoding messages that have been hidden using steganography for the network transfer: this ensures the rest of the client side can act as if all messages are received in plaintext.
    b. Server Side Components
        i. Message Routing Service - This is the primary backend server that will be a REST API hosted as AWS Lambda functions. This is responsible for providing the API that the client side Messaging Service Component interacts with. That means providing functionality for receiving messages to be sent, sending messages to users, looking up user accounts to start a chat, creating new user accounts, etc. In addition, this service is expected to communicate using the steganography protocol that hides the network traffic.
        ii. Postgresql Database - This is the database management system we'll be using for storing user account information, user messages, etc.
2. Interactions and Rationale

    The system architecture was designed in a way that 1.) minimizes the number of interactions between different software components in order to increase maintainability, 2.) maximize abstraction so development of components can be done in parallel by different developers with minimal communication needed, and 3.) uses familiar and popular technologies that are well documented so that we can iterate quickly.

4.1.1. Module Role

The role of this module is to consolidate all of the front end code for user interaction written in React, HTML, CSS, and Javascript into a single space. This module also provides the communication infrastructure for the front end to receive chat messages from the backend(via the Electron Middleman). The Electron middleman is included in section 3.1 because the two modules are highly interrelated and the Electron module will be written in conjunction and by the same developer as the React/Front end module.

4.1.2. Module Connections

This module has two connections. First is the connection between the Front End and the Electron Middleman(this is done via Electrons IPC library), and the second is between the Electron Middleman and the Client Messaging Service Module which is the primary client side 'Backend' component.

4.1.3. Models(N/A)

Standard React with Typescript, Redux, HTML, CSS, JS, and Ant Design. No abstract software architecture.

4.1.4. Design Rationale

The design rationale came from the need to build a desktop application with a modern looking front end. The industry standard these days is to write the front end using web technologies(ie React with Typescript, HTML, CSS, JS) and to package the application into a desktop application using Electron as the middleman to communicate with the operating system.

4.2.1. Module Role

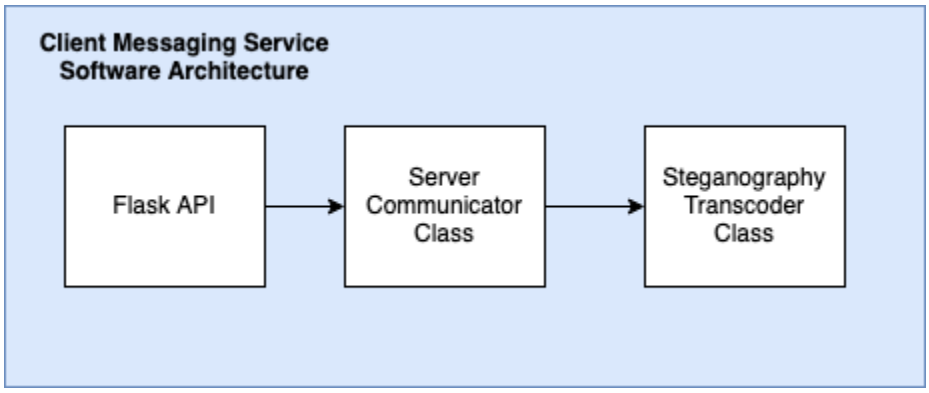
The role of this module is to act as the main backend module on the client side. This means 1.) handling the communication with the server module for receiving messages, sending messaging, creating new chats, etc, 2.) Handling steganographic decoding of messages into plaintext, 3.) providing an easy to use API for the Electron Middleman component to communicate with.

4.2.2. Module Connections

This module is connected to two other components, 1.) The Electron Middleman talks to the Flask API exposed within the Client Messaging Service Module, 2.) The Client Messaging Service Module talks to the REST API exposed by the Server Message Routing Service Module for receiving and sending messages.

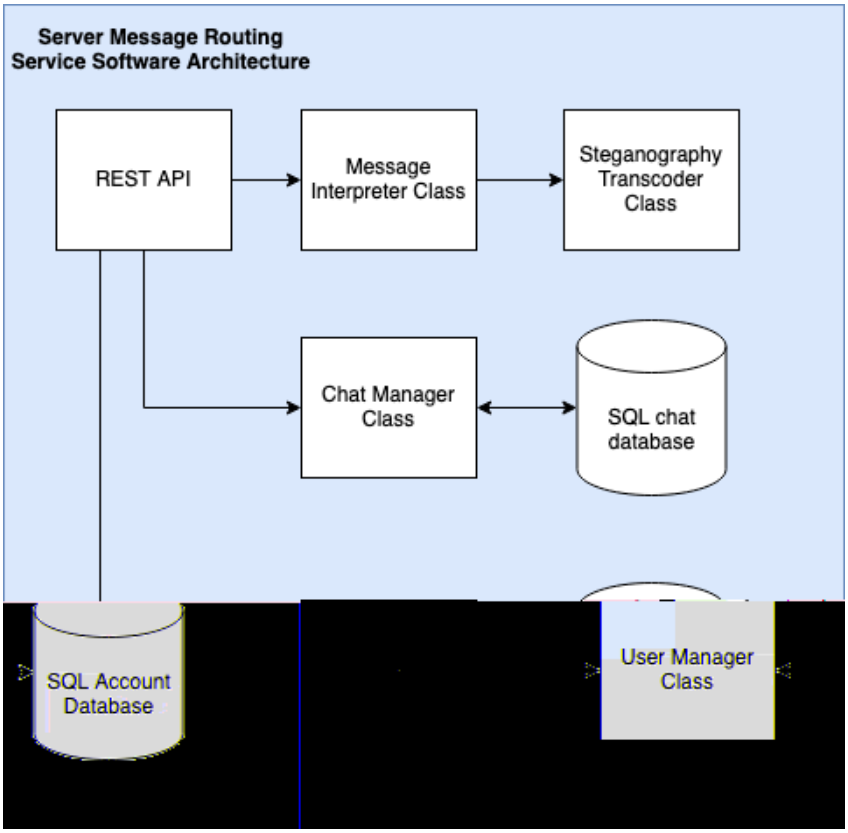
4.2.3. Models

4.2.3.1. Client Messaging Service Software Architecture



4.2.4. Design Rationale

We decided on this three class software architecture for the Client Messaging Service because 1.) It minimized module connections and dependencies which increased maintainability, 2.) Consolidated decoupled functionality which also increased maintainability, and 3.) Utilizes technologies we are familiar with so we can iterate quickly.



4.3.1. Module Role

The role of this module is to consolidate all the server side functionalities into a service that is accessible via a REST API. This makes it easy for the Client Messaging Service to interact and send/receive messages to other users of the chat application. The service is also responsible for allowing the creation of new user accounts and the looking up of users by name.

4.3.2. Module Connections

This module is connected to the Client Messaging Service which is being run on different client computers, as well as connected to a PostgreSQL database where user information and messages are kept.

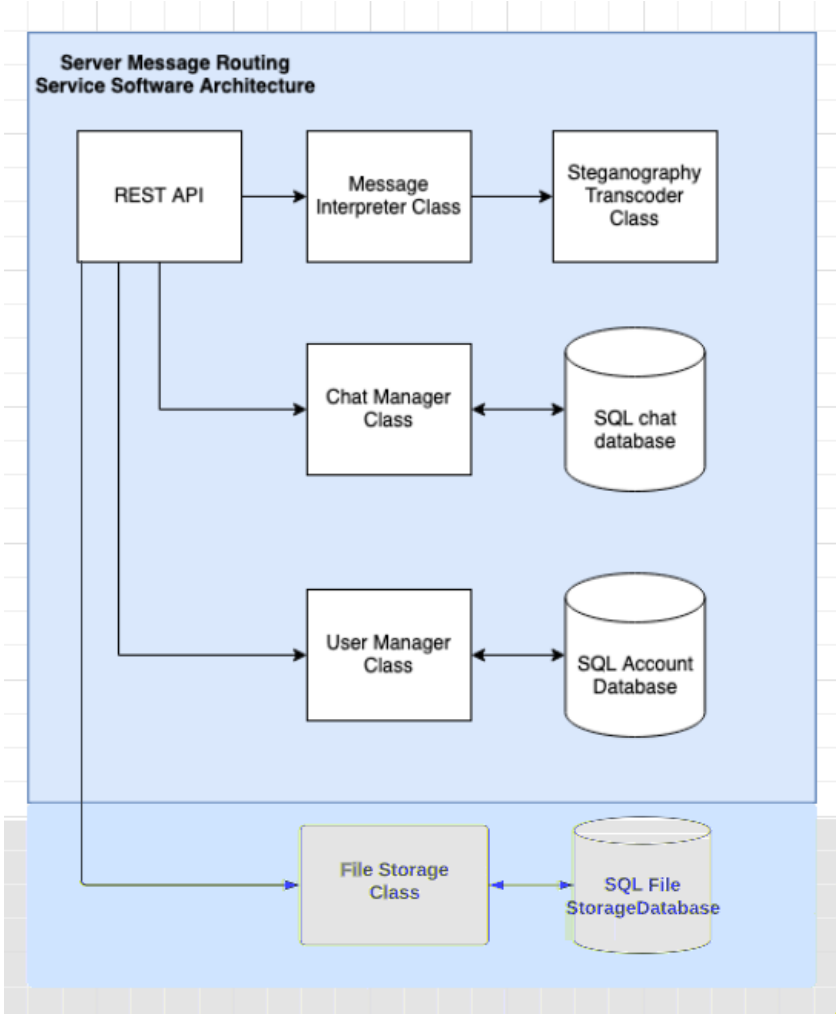
4.3.3. Models

4.3.3.1. Server Message Routing Service Software Architecture

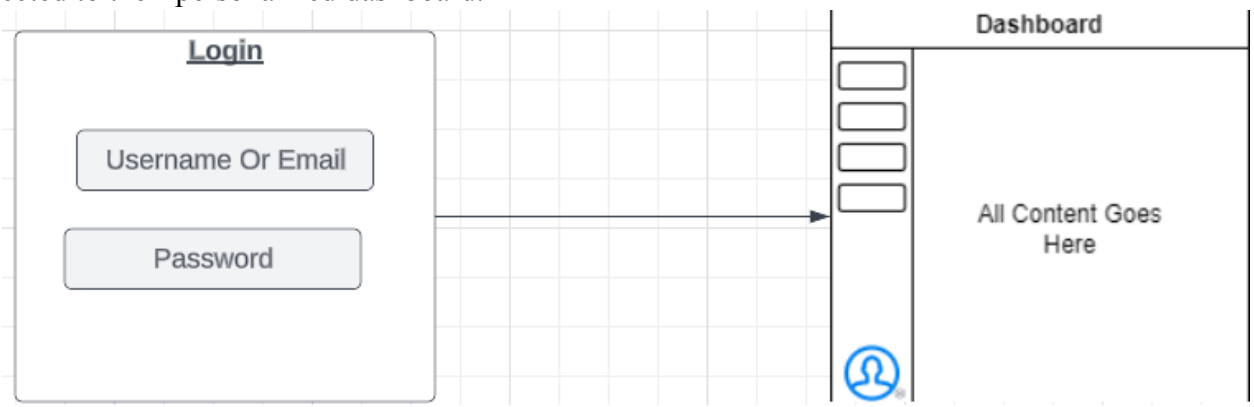
4.3.4. Design Rationale

We decided to design the Server Message Routing Service with this software architecture as it consolidated decoupled functionality into different classes with clear interfaces which allows us to develop in parallel, as well as increases future maintainability. In addition, by not having many cross dependencies, we can have new requests to the server be handled in parallel as there is minimal resource overlap between software components.

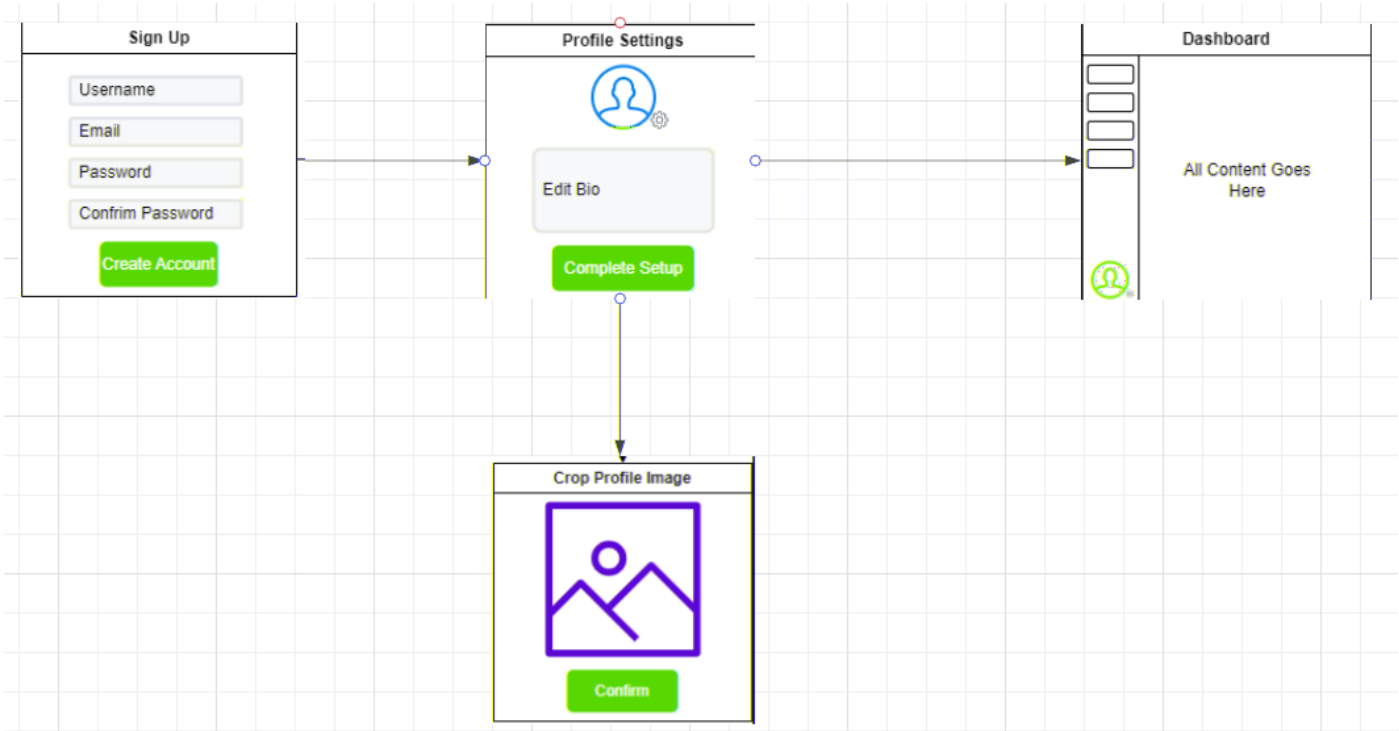
Originally, we were planning on having Stegord feature file storage and file sharing. This would require a whole new class to handle files of variable lengths, and a database in order to track who's files belong to who. However, we found that not only would it be very costly to send whole files that need to both be encrypted and decrypted using a stenography image, it would be out of the scope of this project due date.



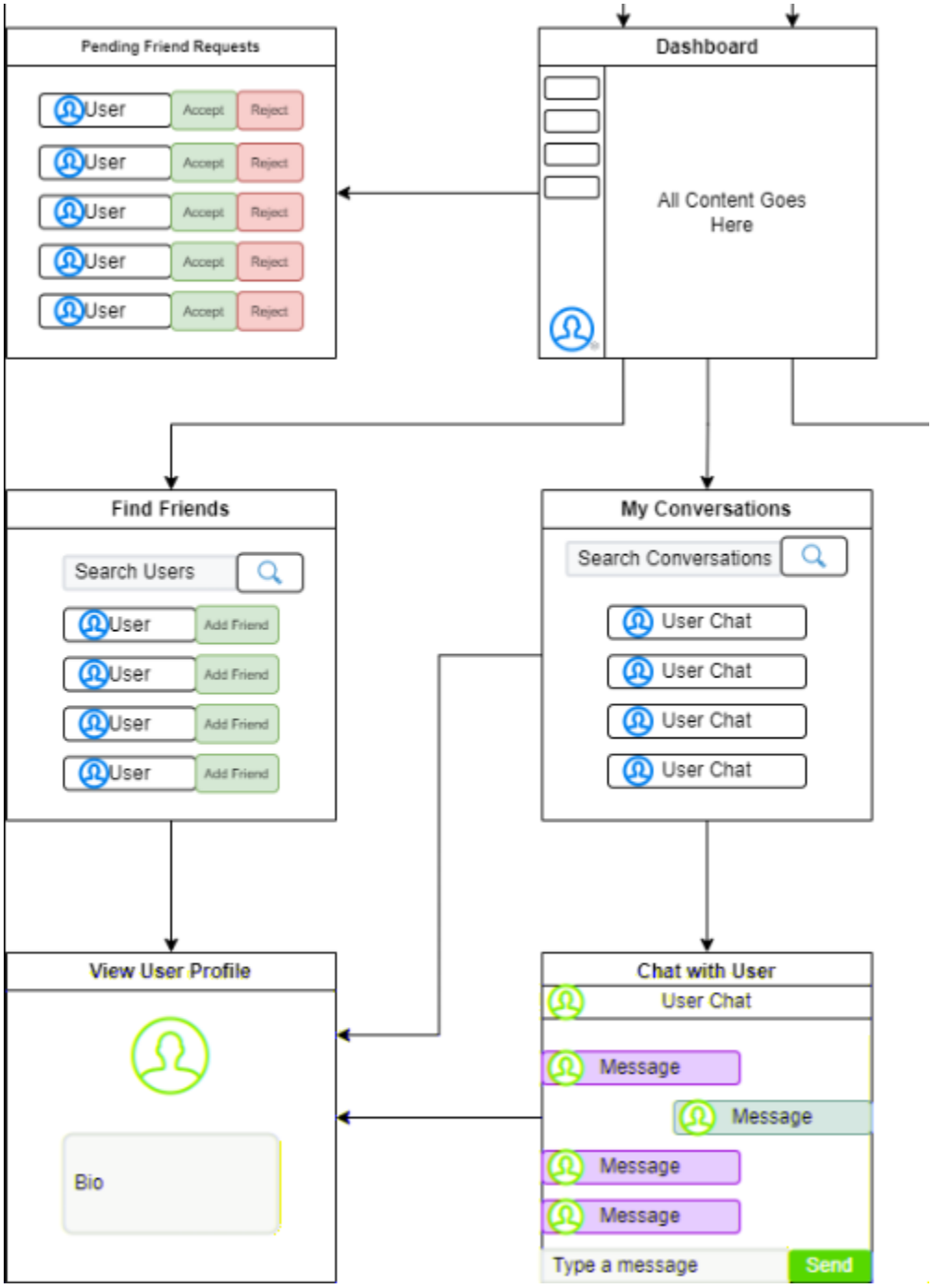
Users will login with either their username or email, and a password. Then they are redirected to their personalized dashboard.



Users trying to sign up, will create username and password, using a predefined email. Then they are redirected to their profile settings to add a bio, and they can optionally upload a profile image. Afterwards they will be redirected to the dashboard.



Once users are in their dashboard, they can check their pending friend requests. Also, from the dashboard they can search their friends list to find a friend's profile and view their bio. When users want to chat with friends, from the dashboard, they can open up their conversations, and from there; either view someone's profile, or continue chatting with another user.



Class Diagram. In *Wi i edia*, n.d. https://en.wikipedia.org/wiki/Class_diagram.

Sequence Diagram. In *Wi i edia*, n.d. https://en.wikipedia.org/wiki/Sequence_diagram.

UML. In *Wi i edia,* n.d. https://en.wikipedia.org/wiki/Unified_Modeling_Language

We would like to thank Professor Juan Flores for providing the template which this software design specification is built on.