ICriterion C: Development

The program's main purpose is to fetch data from other services and display them in a centralized application. This means that it relies on the other's services capabilities to serve that data to me. In this case I used the APIs from the other programs to get that data. Each service had a different method of getting the information I wanted. In my product, I had two services that I had programmed, Steam and League of Legends. For the Steam API, I had to make two calls to the steam servers. The first call was a call to get the users friends list, then a second call was made to get the online status of the user's friends. Although I needed several friend's statuses, I could get them all in one call. This made the steam calls very efficient. These efficient calls are directly opposite of League of Legends. League of legends has no way to get the friends list or the status of their friends, so I had to find a workaround. The first workaround was to scrape the users game history to look for other players that the user plays with often, it then assumes them to be the user's friend. It also grabs a list from a file that the user can set. To then determine if they are online, I check if they are currently playing a game, or if their game was wishing the last few minutes. Despite the massive difference between how the services acquire the data, I designed it to be easy to integrate them into the program. Each service is defined in a python module, and the module just needs to have a function called get_all_friend_statuses(api_key, user_id), and have that function return a list formatted as defined in the README.md. The program reads these dynamically, so that me as the programmer just needs to create these files and place them into the correct folder. This also allows users to customize what services they need, or expand upon the product as they see fit.

The API calls themselves caused trouble while I was programming. In order to protect their servers, the services limit the number of calls a person can make in a set amount of time. To mitigate this problem, I added in a queuing system. The modules add their call to a queue, and the queue ensures that the calls are being made at the correct interval. In a typical computer science fashion, this fixed one issue but created another. The initial design of my program had the APIs as blocking calls, the program could to continue while they were being made. These long API calls froze the GUI and led to a bad user experience. To resolve this I multithreaded the program. The GUI was coupled in one computer thread, and the API calls would be made in another thread, so the blocking calls would not affect overall program responsiveness. This thread was created in a very simple way, I called the function in another thread, and the function contained an infinite while loop that made the API calls.

All these API calls requires a lot of prior knownledge, such as the username of the user, and other predefined friends lists or data. In order to make the application as user friendly as possible, this data can be edited in three ways. The first is a small setup script that prompts the user for data then inputs the data into the appropriate files. This setup script is run the first time the program is run. The second way to edit the data is by directly modifying the files

themselves. Most data is stored as plain text, and thus can be modified with simple and widespread programs. This does leave room for the user to make mistakes and break the program. There is currently no way to repair the files so this method is discouraged. The third and final method to edit the data is with the options window in the program. This method will be what the user will use the most, and it fits with the design of the rest of the program.

The user statuses are output into an listbox format. From there the user can modify the sorting and filtering. The first sorting method is by service. From each service, it creates a section that shows the username and status. The user can sort it that way, or sort it by person. To sort by person, the user must first define a list of people, and the usernames they go by. This is stored in a file, and is parsed when the application goes to sort the data. Additionally, the user can filter out results by clicking on the logo buttons in the left column. This is done through an if statement in the sorting method. The sorting methods are stored in the tools module despite only being used by the graphics module. See screenshot1.png.

The program was designed to be as modular as possible. They're are 3 main parts to the source program. The "main.py" file, the library directory, and the plugins directory. The library directory contains two files, "graphics.py" and "tools.py". The plugins directory contains all the ".py" files that define the modules for all the service APIs. The program works when you call the "main.py" file. It sets up an instance of the AppManager, which in turn sets up an instance of the AppWindow (The GUI). The AppManager then starts up the thread that makes the API calls. The program determines what services to use by walking through the plugins directory, and it attempts to use each file in the folder as an API module. The plugins folder allows the user to customize what items they want to use. All other files, such as data files, images etc. are stored in another directory called "files". A flowchart of the logic order of the program is available in "logic_flow.png".

To make the product easier for the user to use, the program is packed up into a single executable, on Windows it is "nomen.exe" and on OSX it is "nomen.app". On Linux, the program is compiled into c files, but still must be executed by command line.