Face Recognition with Raspberry Pi

Matthew Menten & Eli Jacobshagen

CSCI 5/4448 OOAD

Montgomery

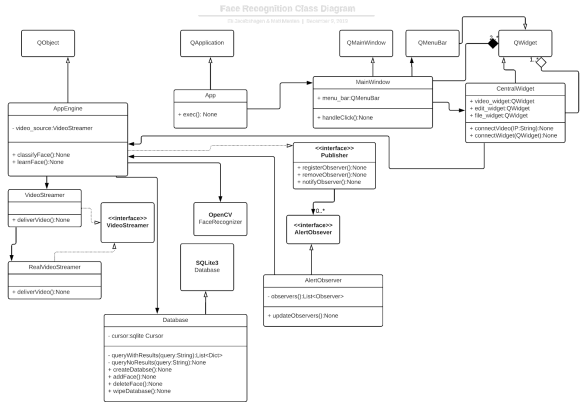
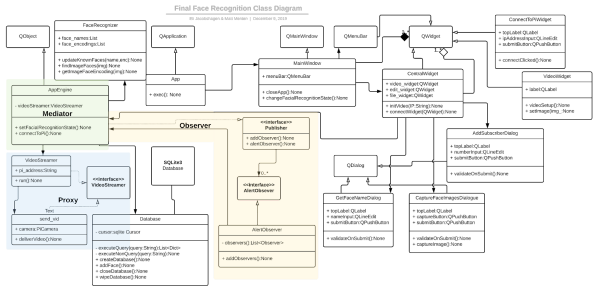
Project 6 Final Report

Final State of System:

This project was designed to be a proof-of-concept for a home security system that provides a live video feed via Raspberry Pi camera module, and alerts the user of unknown intruders near their home. As of writing this report, our system provides all the features that were included in the initial project design. The system consists of a Raspberry Pi with a camera module, which is meant to be installed on or inside a home, and a client, which is meant to allow the user to view the feed, manage the app, and send alerts. The Raspberry Pi functionality is simple; it initializes a video stream and sends it to a TCP socket. When the client starts, it prompts the user to enter the IP of a Raspberry Pi that is actively streaming video. After the user enters the IP of the desired Pi, the client creates a socket connection with said Pi, and displays the video stream to the user. One of the central features of this software is facial recognition via several machine learning techniques. The ML algorithms used in this project were provided by PyImageSearch (*face_recognition*, *dlib*) and *OpenCV*. This feature is necessary for both the detection of trusted faces and the detection of intruders. Additionally the system contains a SQLite database which maintains a persistent set of trusted faces between independent runs of the client. The client allows the user to add new faces to the database from the video stream, and delete faces that already exist in the database. SMS alerts are sent using the Twilio API

whenever an unknown face is detected by the client. In theory this would allow a user to be remotely notified of an intruder even if they have no access to the client. Overall, all of the functionality in our design has been implemented.

Final Class Diagram & Comparison:

Initial UML Class Diagram	Final UML Class Diagram
	

For the most part, our class/object design stayed constant throughout the development of our app. Still there are a few changes/additions that should be pointed out. Three dialog objects inheriting from QDialog were added. Two widget objects were also added. In addition, certain member, method, and class names changed. Other class descriptions were made more thorough. In the final diagram, the three design patterns we used are highlighted as well. Nailing down a strong class design at the very beginning of the project allowed us to not have to redesign much, so the diagrams are mostly identical. Full size copies of the above diagrams are at the end of this document.

Third-party Code Statement:

The completion of this project would be impossible without the use of several third-party libraries and APIs. First, the Python package 'face_recognizer' (PyImageSearch) by Adrian Rosebrock provided all the facial recognition functionality needed. This package employs *dlib* and *OpenCV* to run images through a Convolutional Neural Network that is trained to detect human faces, and then to compare detected faces to a user-created set of trusted faces. These machine learning algorithms are complex and involved, so implementing them from scratch was out of scope for our project. Rosebrock wrote a very in-depth tutorial for this package which was extremely useful. This tutorial is credited and linked in our FaceRecognizer object.

A significant amount of development time was dedicated to installing and setting up the Nvidia CUDA Toolkit, which can be leveraged by *dlib* in order to use eligible clients' GPUs to improve performance of ML classification algorithms. This process proved to be difficult because the client computer was originally running Windows 10, and as we discovered, it's incredibly difficult to setup a sufficient development environment on Windows 10 without installing a myriad of large, clunky applications (like Visual Studio or Cygwin). Virtual machines or the Windows Subsystem for Linux (WSL) were not viable workarounds because those systems are only exposed to virtual hardware. This limitation made compiling the CUDA library impossible in those environments, so the next best option was installing Linux as a standalone operating system on the client. From our brief testing, running our application on a client with a CUDA GPU significantly improves its performance.

Next, we used the Twilio REST API to facilitate the sending of SMS messages to observers. The Twilio documentation provides tutorials and code examples, which made it very simple to send text messages to a phone number.

Raspberry Pi code draws from the Python PiCamera package, as well as several tutorials on sending video streams over a network socket. The database engine we used is SQLite3. We used the PyQt5 framework to build the GUI. Additional Python packages like *numpy* and *pickle* were also employed. All code examples in the aforementioned tutorials had to be changed substantially in order to fit the needs of our application, and all third-party code is attributed appropriately within the source code.

Links to documentation used:

https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/

https://www.pyimagesearch.com/2015/03/30/accessing-the-raspberry-pi-camera-with-opencv-and-python/

https://picamera.readthedocs.io/en/release-1.10/recipes1.html

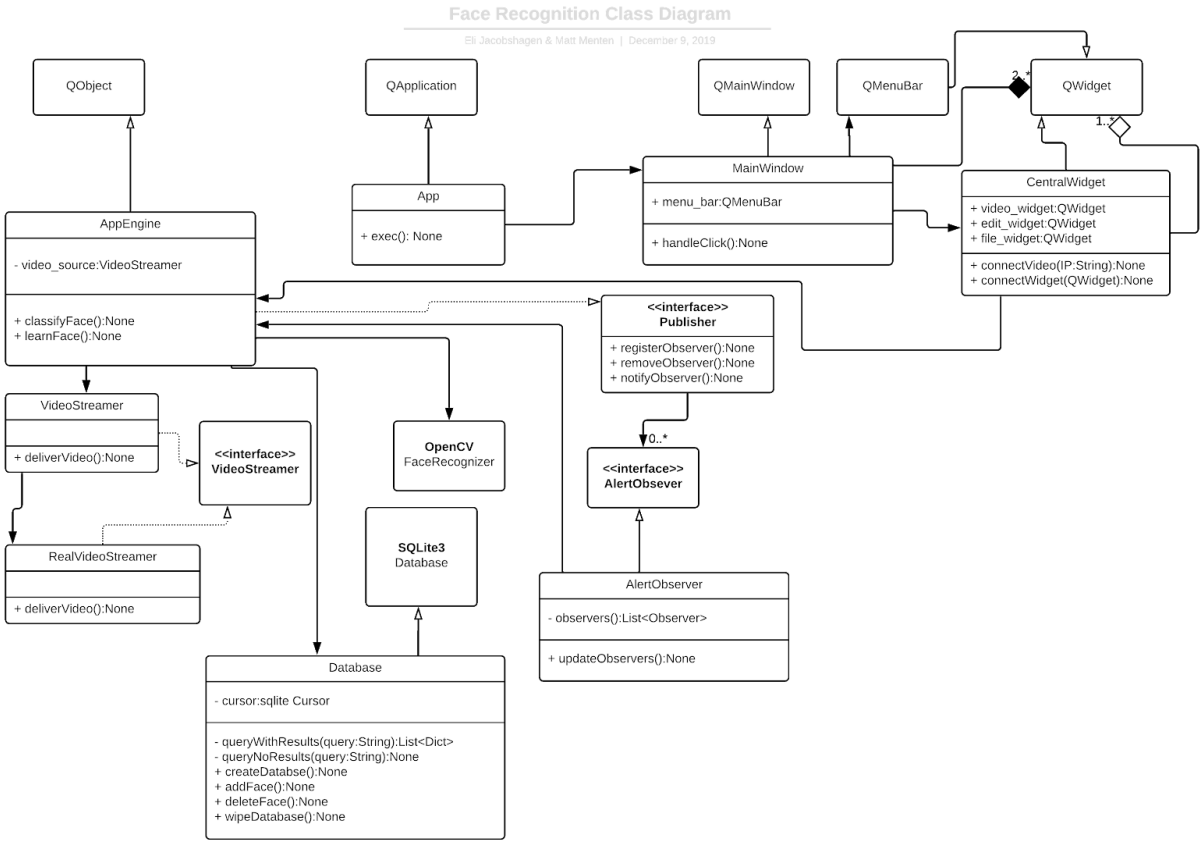
https://www.twilio.com/docs/sms/quickstart/python

Reflection on OOAD Process:

The principles of Object-Oriented Design helped us throughout the execution of this project. One of the OO principles we used was the idea of designing and planning objects and system architecture upfront, before implementation begins. This was useful for a couple of reasons. First, it allowed us to create a high-level conception of our system and the individual pieces within it. With so many working pieces, a decent initial design allowed us to

conceptualize how everything was going fit together. Additionally, our thorough design served as an invaluable resource anytime we ran into issues or confusion throughout the execution of the project. We tried to follow the 'Single Responsibility' principle as much as possible; most objects do one core thing, and one thing only. 'Design by Contract' was followed as well; once code implementation began, we started writing skeleton code for the high-level objects dictated in our design, and then iteratively filled in the details. The 'Hollywood Principle' was followed in our signal passing throughout the app. Objects in our app *notify* other objects when they have information to share, as opposed to *requesting* information from other objects. One of the core aspects of Object-Oriented Design is the idea of patterns, and we tried to use patterns to our advantage in our implementation. After coming up with the high-level objects and responsibilities in the app, it was clear that a few patterns applied directly to our needs. As such our application uses Observer in the SMS alert functionality (and general signal passing throughout our app), Mediator in the 'engine' of the app to facilitate communication between the smaller components, and Proxy in the socket communication. Thinking about these patterns in the context of our code allowed us to abstract away the details of fairly complex functionality, while simultaneously providing us with tried-and-true solutions to problems that are encountered often in code. Following these OO principles resulted in a smooth development process and a successful end product.

Initial UML Class Diagram:



Final UML Class Diagram: