Project Title: Pumpkin Pi

Team Members: Kamiar Coffey, Amari Hoogland

Code Submission: https://github.com/kamiarcoffey/PumpkinPi

Final State of Project

The iOS app, the cloud database, and the Raspberry Pi are fully functional and integrated.

The app includes functional views for navigation and data display. It also includes two reactive graphs that will update as new cloud data is fetched. The app includes asynchronous fetching from the database as well as CRUD functionality to update the database with an admin override. The Firebase database has been set up with customized read-write permissions and a secret access key.

The Raspberry Pi was initially set up to use Tensorflow Lite to analyze and project the COCO SSD MobileNet v1 model object labels onto the live video stream. However, though a baseline of live detection was achieved, integrating the live object detection to trigger updates to the database proved to require implementations out of scope.

Instead, a simulation script that randomly triggers entry/exit events and pushes the events to the database was created to speed up integration and testing. This simulation script is randomly triggered throughout the parking garage's open hours via a bash script automated to run in real time (thereby maintaining the database in real time). Further, a random number of events are created every time the script is triggered to keep car activity consistent with real world data.

Ultimately, the system can use the Raspberry Pi to monitor the parking activity of a given garage via live video stream and update the database when entry/exit events are triggered. Firebase is connected to the application and displays the live data.

Features - Implemented

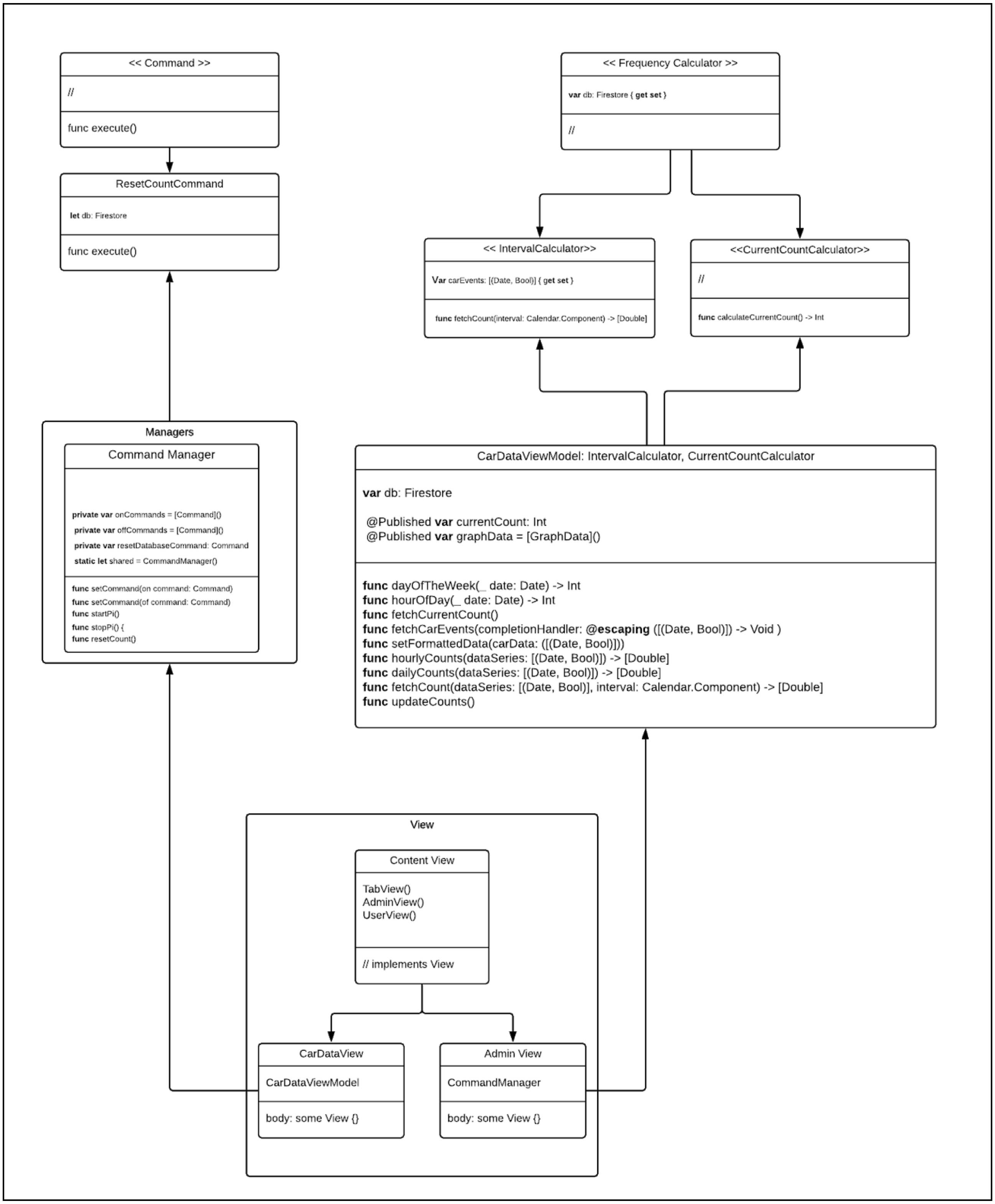
The Raspberry Pi was uploaded with a trained machine learning model to recognize cars entering and cars leaving a parking garage as separate events. The Pi can also live stream it’s video data locally. The Pi can access the cloud database to push events in real time and can simulate random parking garage activity in real time.

The iOS app can pull events on command and has the following use cases: 1) viewing the number of cars currently in the lot, 2) viewing the busiest days, 3) viewing the busiest hours and 4) updating/overwriting the database manually with admin authentication from the app.

Features - Not Implemented

Features relating to the object-oriented integration of the system overall were prioritized. Thus, features more specific to their respective technology were ultimately ignored. This includes turning the Pi on and off from the front end application and training personalized object detection models from the camera.

Final Class Diagram and Comparison Statement



<< Command >>
//
func execute()
ResetCountCommand
let db: Firestore
func execute()
<< Frequency Calculator >>
var db: Firestore { get set }
//
<< IntervalCalculator>>
Var carEvents: [(Date, Bool)] { get set }
func fetchCount(interval: Calendar.Component) -> [Double]
<<CurrentCountCalculator>>
//
func calculateCurrentCount() -> Int
Managers
Command Manager
private var onCommands = [Command]()
private var offCommands = [Command]()
private var resetDatabaseCommand: Command
static let shared = CommandManager()
func setCommand(on command: Command)
func setCommand(of command: Command)
func startPi()
func stopPi() {
func resetCount()
CarDataViewModel: IntervalCalculator, CurrentCountCalculator
var db: Firestore
@Published var currentCount: Int
@Published var graphData = [GraphData]()
func dayOfTheWeek(_ date: Date) -> Int
func hourOfDay(_ date: Date) -> Int
func fetchCurrentCount()
func fetchCarEvents(completionHandler: @escaping ([(Date, Bool)]) -> Void)
func setFormattedData(carData: ([(Date, Bool)]))
func hourlyCounts(dataSeries: [(Date, Bool)]) -> [Double]
func dailyCounts(dataSeries: [(Date, Bool)]) -> [Double]
func fetchCount(dataSeries: [(Date, Bool)], interval: Calendar.Component) -> [Double]
func updateCounts()
View
Content View
TabView()
AdminView()
UserView()
// implements View
CarDataView
CarDataViewModel
body: some View {}
Admin View
CommandManager
body: some View {}

Figure 1: iOS App Final Class Diagram

Changes from Project 4

The front end changed from using an MVC architecture to an MVVM architecture for viewing the car Data and an MVC for sending commands. The User use cases discussed in project 4 were transitioned to MVVM to take advantage of Apple's Observable features for asynchronous updates.

We also added protocols to handle calculating hourly and daily intervals from a list of all car entry and exit events (filtering and accumulating by day of the week). This means we also added another collection to the Firebase database that stores a full car entry or exit event along with it's time stamp. Previously we had just been storing the current count in the database and would have relied on the app to keep running tallies. That old architecture is much more tightly coupled than the new architecture.

The scope of the project also changed from Project 4-5. Mainly, feature implementation was abandoned on processes not directly related to the object-oriented system. Following, the Flask app control of the Pi and the machine learning side of object detection was scaled back. These have been moved to future integration plans.

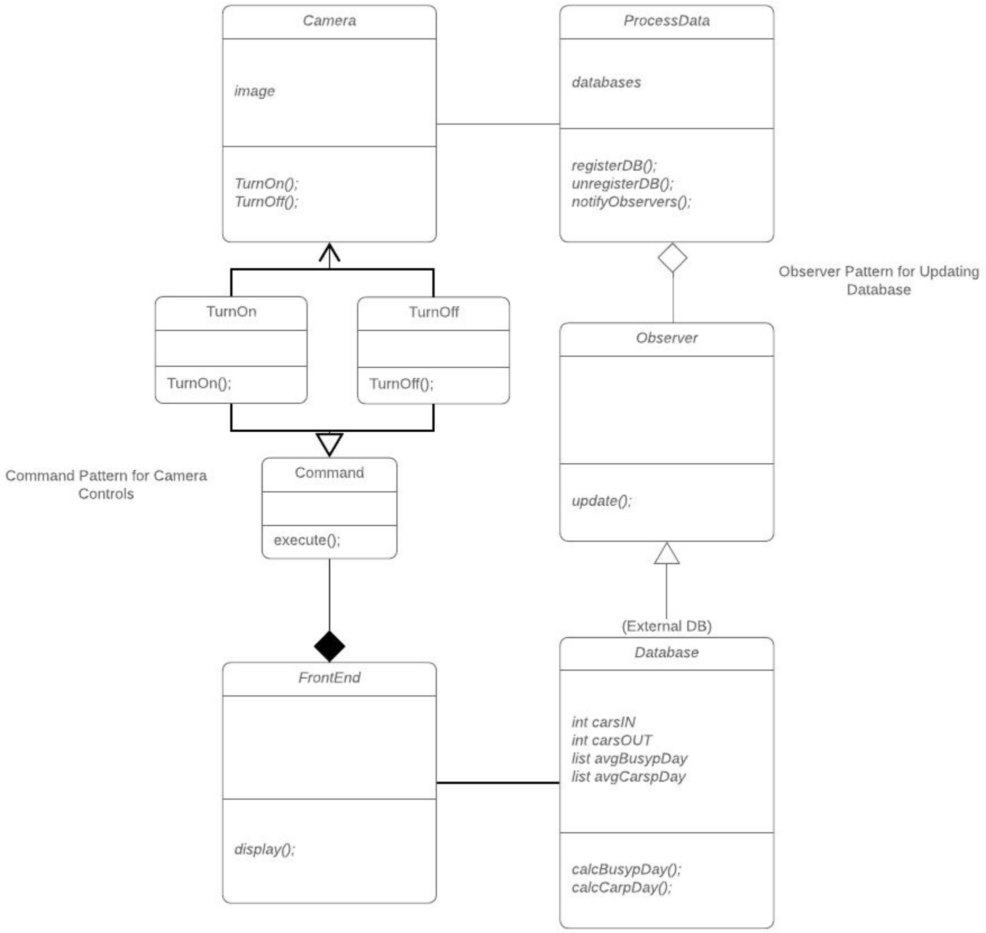


Figure 2: System Architecture

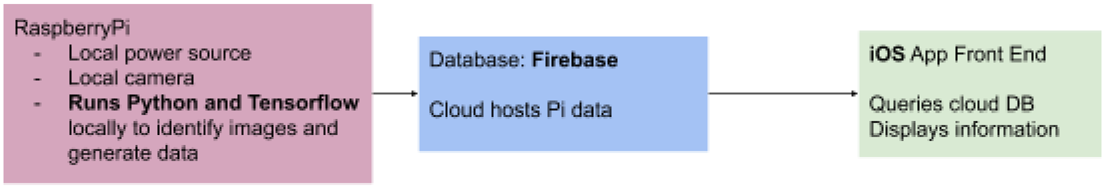
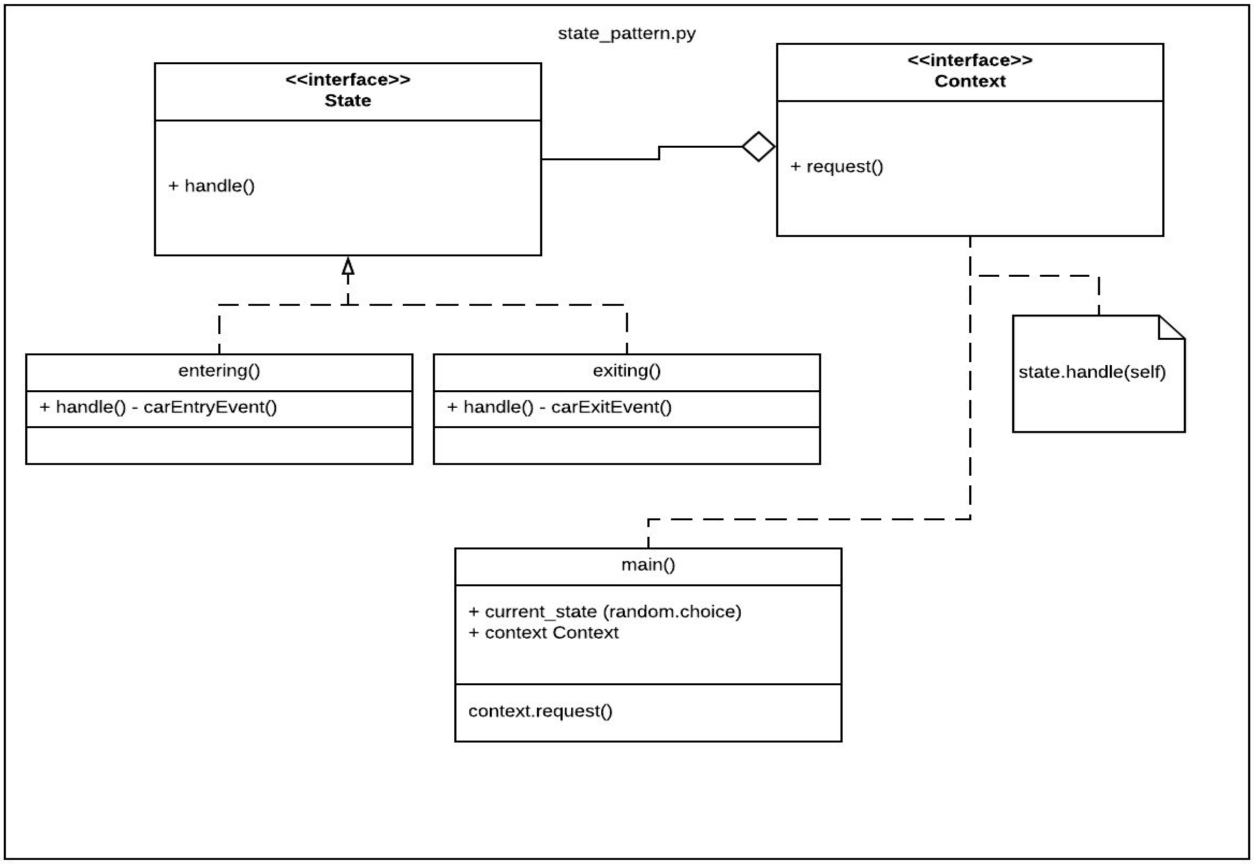


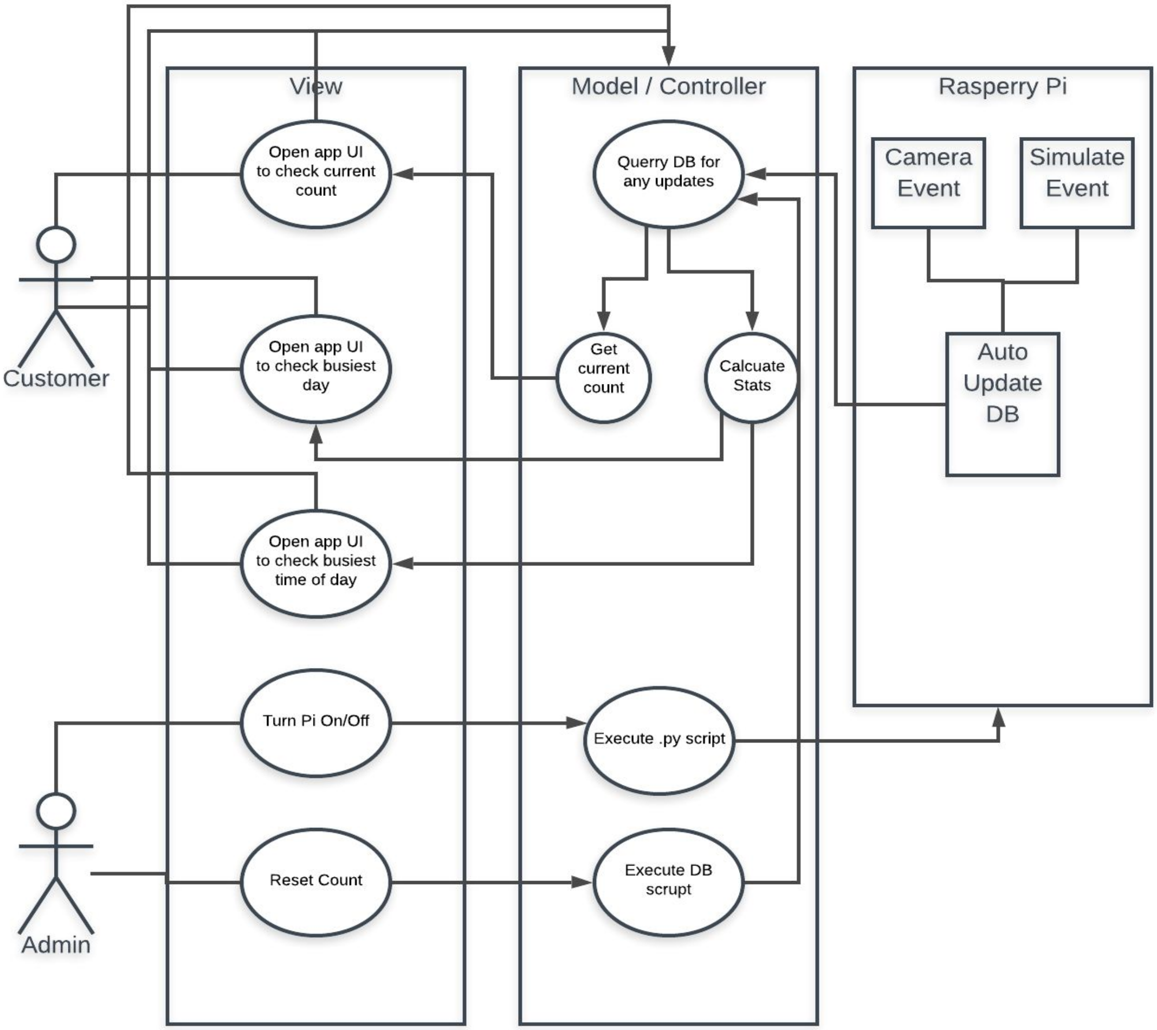
Figure 3: System Stack diagram



The system stack diagram remains the same as from Project 4.

Figure 2: Pi Diagram (state pattern for integrating into system database), independent features unrelated to system left out.

Figure 3: System Use Cases



Original Code Statement

All code is original in the iOS app with the exception of UI formatting for the bar charts. Creating bar charts is done via building shapes within a constrained view. We used an MIT licensed library published by a personal author on GitHub who has written some code to create shapes that look nice and respond to user touches for additional style. The original source code was edited to accommodate our needs, but the UI formatting remains original to the author. Proper source citing is included in the files which reference others' work, BarChartView.swift. All sources referenced for setting up and integrating the Raspberry Pi are cited in the Pi documentation below. All code relating to the object-oriented system is original.

Raspberry Pi tutorial references:
https://desertbot.io/blog/headless-raspberry-pi-3-bplus-ssh-wifi-setup
https://desertbot.io/blog/how-to-stream-the-picamera
https://gist.github.com/miguelmota/d1d70d19fbd763afdd65c5c1375ddbde

Statement on OOAD Principles Used

The app uses an MVVM architecture.

- Within that, the Observer Pattern is used to keep the views data in sync with the published car data that comes in over the network.
- It also uses the Singleton pattern to manage the database connection.
- A Command pattern is used to implement functionality for controlling the Pi as well as CRUD updates to the Firebase database.

The Raspberry Pi uses a State Pattern to trigger entry/exit events and push to the Firebase database with specialized behavior.

Three key design process elements / issues the team experienced:

1. Integrating existing parts of a system can be hard. For example, each task we set for ourselves at the beginning of Project 4 could be accomplished, but it was harder than expected to make each piece work together because defining how two those pieces interact places design restrictions on them.

2. It's good to change design patterns, but much more dangerous to change an entire architecture pattern. For example, we changed Command and State

patterns between project 5 and project 6, which was great -- however, we never moved away from using an app front end, and a Flask stack (the system “architecture”).

3. Technical communication is key when working with a team. It’s crucial to be able to explain what design pattern you’re using in a shared vocabulary so the other person knows what to expect from your software in terms of both abilities and API interaction.