‘Close By Places’ iOS App
Cal Poly Senior Project

Chris Yerina
Computer Engineering 2018
Advisor: John Bellardo

Abstract

The name of this app is Close By Places. This app allows users to save their favorite restaurants or stores in the app and allows them to find the closest location quickly and easily. The user types the names of their favorite places to be saved in the app. When the user wants to find the closest location of one of their favorite places, they select it. Then the app presents a map view that zooms in to show the user's location and the closest location of the selected place. The map view also allows users to zoom out to see other nearby locations. The user can tap a button which will open Apple Maps to get directions for the selected location. This app is available for download on the iOS App Store.

Introduction

I originally wrote this app in 2015 using Swift 1.2. The main goals of this project were:

- Integrate the new version of Swift (4)
- Rewrite some components of the app for better performance
- Add a couple new features

I still use my app. For my senior project, I wanted to continue to work on this app and continue to make it useful for others as well. I knew what I wanted to do in order to redesign some of the features that I originally wrote for this app. I also wanted to add some new features. One new feature I wanted to add to my app was custom ordering of items on the main screen. The main screen shows a list of the user's saved favorite places. The first version of my app only appended items to the end of the list and the user could not reorder them. Implementing this feature required some design decisions because there are multiple ways to implement this. The second major feature I wanted to add was autocomplete when the user types in places to save. The key design decisions for this feature revolved around which APIs I would use and how I would integrate it in my app.

App Store Link: https://itunes.apple.com/us/app/close-by-find-your-favorite/id1028800432?ls=1&mt=8

Background

Previous Work

I originally wrote version 1 of this app in 2015. I was not as experienced as a software engineer as I am now. There were many different components of the app that were not written properly or robustly. It is amazing that I was able to make the app work by making certain hacks. I made these hacks because there were certain concepts that I did not fully understand. Now that I have more experience programming, I wanted to rewrite parts of my app to write the code properly, and to also improve the performance.

Key Design Decisions

Redesigning the Local Storage

When I originally wrote this app, I used Realm to store data locally on the device. Realm is an open source object database geared towards mobile platforms (iOS/Android). The data stored by the app is the list of the user's favorite places (see Figure 1). I originally used Realm because that is what I used when I was learning how to store data on the device. I was able to make it work for the app originally. However, there were a couple of reasons for why I decided to use something else.

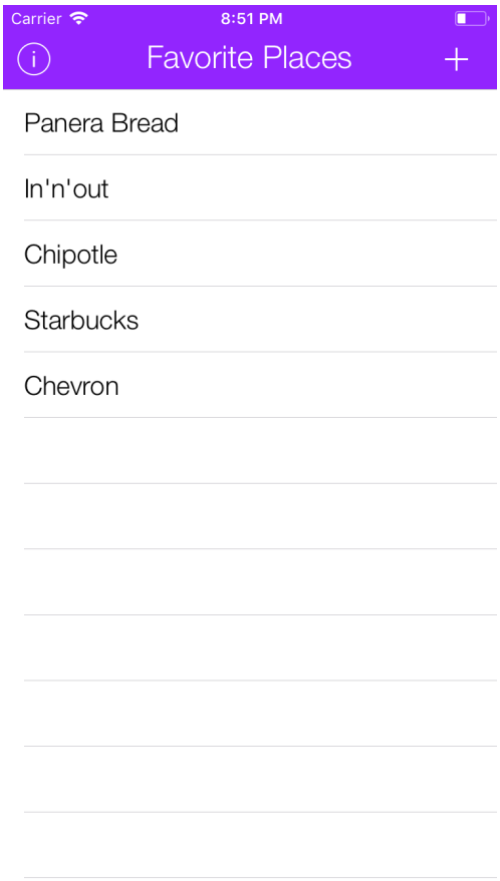

Figure 1: Main screen that displays list of favorite places

It is possible that the Realm APIs might have changed since I originally wrote the app, but the following discussion deals with version 0.93.2 of Realm/RealmSwift. When I worked with Realm in this app and other apps, I was required to use a custom Realm collection type. It worked in this app, but in other apps it was not easy to figure out how to use a different collection type when I needed one other than a Realm list. It was also not possible to use a built in Swift collection type with Realm. The other big reason why I chose to replace Realm was because of the maturity of Swift. I originally wrote the app in Swift 1.2 and the language is still being developed, as Swift 4.2 is the current version. The language changing led to Realm changing. I wanted to try and eliminate it as a

dependency because I did not want to make significant changes every time Swift updated.

In order to redesign this feature, I utilized NSCoding, FileManager, and NSKeyedArchiver/NSKeyedUnarchiver. All of these APIs are built-in to iOS in the Foundation framework. The following descriptions can be found in the iOS Developer Documentation:

- NSCoding - A protocol that enables an object to be encoded and decoded for archiving and distribution.
- FileManager - An object that provides a convenient interface to the contents of the file system.
- NSKeyedArchiver/NSKeyedUnarchiver - A coder that stores an object's data to an archive referenced by keys/A decoder that restores data from an archive referenced by keys.

In my code, each item in the list of the user's favorite places is a Favorite class. Currently, the only instance variable is a string, but I wanted to make it a class in case I wanted to add other functionality later. The Favorite class implements the NSCoding protocol. The methods that are part of the protocol take in an NSCoder as a parameter. The methods either encode or decode the instance variables of the class based on a key which usually matches the name of each instance variable. The FileManager gave me access to the file system on the device. I accessed the documents directory within the user's home directory. Then I appended "favorites" to the document directory path in order to create an archive URL to a file. This archive URL is used by the NSKeyedArchiver/NSKeyedUnarchiver so it knows where in the file system to save and read the list. It also calls the NSCoding methods implemented by the Favorite class.

Redesigning the local storage this way provided the following benefits:

- I could use the built in Swift types to work with the data that I was saving.
- The code to implement the interaction with the file system/local storage was simple.
- These APIs have been around since iOS 2.0 and will not be changing. Therefore, there will be less overhead for me as the developer, the next time Swift changes.

Redesigning the Map Zooming

The main function of my app is to perform a local search around the user. This functionality is provided by an API that is part of MapKit called, MKLocalSearchRequest. This is an asynchronous request to the network and takes a completion handler as a parameter. When I originally wrote this app, I did not understand the concept of closure. However, completion handlers rely on closure to make network requests work smoothly for the UI. Closure is the ability of a function to capture variables from the scope in which it was created. In this specific example, the completion handler of the request can access the variables in the enclosing method and class. The completion handler is called once the request has finished so that the data can be unpacked and the UI can be updated.

On the screen that displays the map, there are three main tasks that happen:

1. Locate the user
2. Perform the search
3. Prepare the UI with the data and for the user to interact with

In the first version of my app, I was not getting the user's location correctly. The way I implemented it was a hack and was not the proper way to do it. MKMapView has a delegate method that gets called when the user's location is updated on the MapView. Originally, I was using this method, but I was not using it correctly. I rewrote the logic in this method to check if the user's location is available (not nil). If it is available, then I would perform the local search.

When I originally wrote the app, I did not write the code to perform the search correctly. Since I did not understand closure, I did not understand that the completion handler gets called when the data is available and the rest of the code continues to execute. I ended up adding a delay in my code in order to get it to work properly. This was a hack and needed to be redesigned. For redesigning this part, I unpacked the data and did not make many modifications to that part of the code. After the data is unpacked, the code enables user interaction with the buttons on the UI and the MapView. I rewrote the completion handler to include unpacking the data and allowing the user to interact with the UI. That way both of these things would happen once the request completes. Close By Places also zooms in on the MapView to show the user's location and the closest location to the selected place (see Figure 2). In the redesign, I got rid of the delay, making the code cleaner as well as improving the performance of the app.

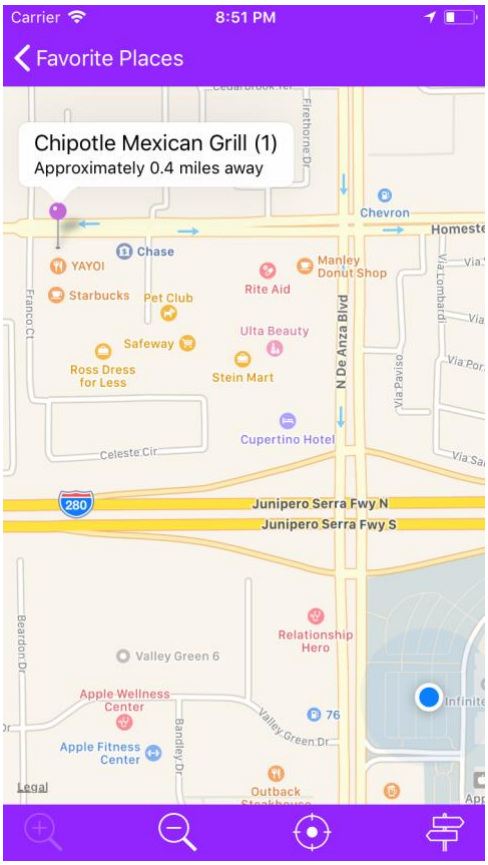

Figure 2: The map presented to the user

In version 1 I also used a 3rd party map overlay called Mapbox. I did this because I wanted to customize the look of the map. The Mapbox map did not render very fast. The search would complete and the map would still be rendering. Removing this from Close By Places and using the regular MapKit map improved performance because the MapKit map renders very fast.

Custom Ordering on Main Screen

One feature that I wanted to add to my app was custom ordering of items. I wanted to allow the user to customize the order of their favorite places on the main screen. For example, the user could put more frequently visited places at the top. Version 1 of the app did not have custom ordering. Every new item was appended to the end of the list. I had implemented an editing style allowing the user to swipe left to delete an item. At least they could add to the list and delete from it. See Figure 3.

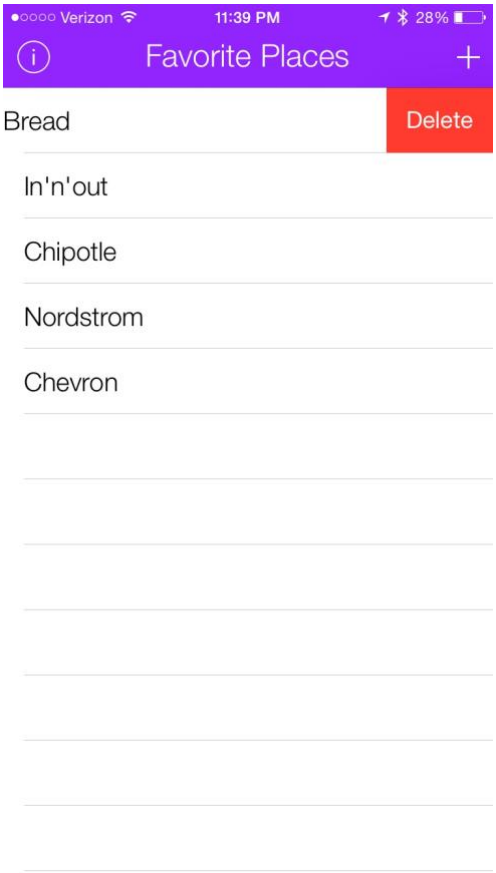

Figure 3: Swipe to delete on a UITableView

At first, I thought that this feature would be pretty easy to implement since TableView editing has been part of iOS for a while. When I was figuring out the design for how I was going to implement this feature, I came across two methods. See Figure 4. The first method was to put the TableView in editing mode and allow one of the editing modes to be reordering. The second method was to implement drag and drop on the TableView cells using gesture recognizers. I could not design this feature using editing mode because using editing mode places an edit button in one of the upper corners of the screen. As shown in Figure 3, I already have two buttons in both upper corners of the screen. I designed this feature using a long press gesture recognizer on the TableView.

This implementation was much more complicated than using editing mode. Luckily, I found a tutorial that teaches how to do this in Swift 3. After stepping through the code and making some conversions to Swift 4, I was able to get this feature to work. The procedure is described below:

- Add a UILongPressGestureRecognizer to the TableView
- Get the state of the gesture, the location in the TableView (coordinate), and the indexPath for the row at that location
- If the state is began:
    - Get the cell at the index path
    - Take a snapshot of the cell
    - Add the snapshot as a subview to the TableView

- If the state is changed:
    - Move the cell in the list
    - Move the cell to the new row in the TableView
- If the state is ended:
    - Remove the snapshot from the TableView

Taking a snapshot of the cell basically created an image of the pressed cell, turned it into an ImageView, and then returned the ImageView. The tutorial that I followed helped a lot. The code used a lot of methods in UIKit that I was not that familiar with.

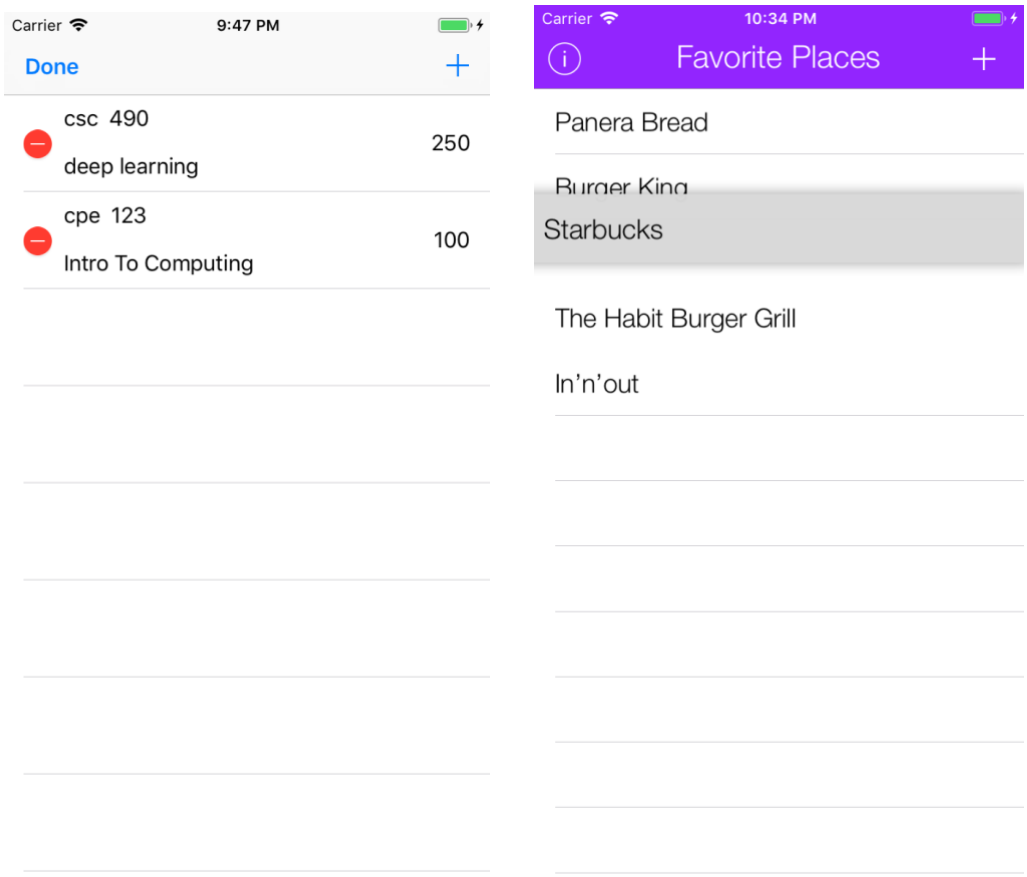

Figure 4: Editing mode vs. Long Press

As part of custom ordering, I wanted to include automatic scrolling when the user drags a cell near the top or bottom of the screen. The tutorial that I used did not include how to do this so I included this part myself. In order to implement this, I included some more logic in the "changed" state. This logic would check the location of the long press and the row, and if it was near the top or bottom of the screen, the TableView would scroll in the right direction. Getting this to work right required a lot of testing. It was difficult to find a threshold for how close the user's finger could get to the top of the bottom of the screen before scrolling.

Favorite Place AutoComplete Suggestions

Another new feature I wanted to add to my app was improved autocomplete suggestions. When the user wants to add a new place to their list of favorites, I wanted to provide suggestions that they can select. I had this functionality in version 1 of my app but it was not dynamic. I wanted to design this feature differently and provide the user with better suggestions.

In version 1 of my app, I provided the user with autocomplete suggestions. However, the list of suggestions was static and hard-coded into the app. This did not give the user the best experience because it was very likely that they would type in a name that the app could not suggest.

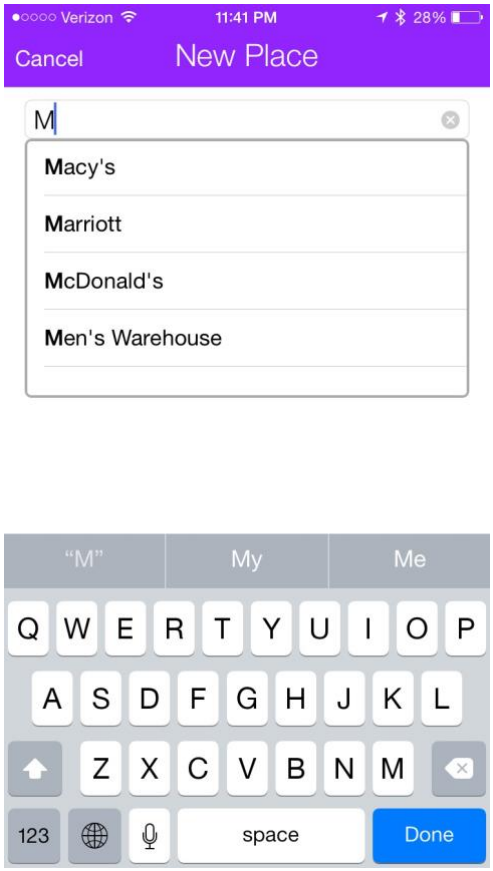

Figure 5: Screen from version 1 that allows user to add a place

The first change I made to this screen started with the UI. In Figure 5, the user types in a UITextField. I redesigned the UI to so that that user types in a UISearchController. I wanted to do this because I wanted to take advantage of the full screen to show the results. The UISearchController also has a lot of functionality built in for displaying and filtering the results (See figure 6). The next step for implementing this feature was deciding what type of data source to use that would provide the user with suggestions.

There are many places/business APIs available to developers. After doing some research, there were three possible candidate APIs that I could include in my app. A short analysis of each is described below:

- Google Places API – This API is very good for providing data for places/businesses. However, it is required that data from this API be shown on a Google Map. I could not use this API because one, Close By Places uses Apple Maps, and two, the results are not shown on a map.
- Foursquare API – This API's strength comes from the fact that a lot of the data is provided by users. It also has good coverage in Asia. This was an important detail because according to iTunes Connect, most of the Close By Places users are in Asia.
- Yelp API – This API has good coverage in the US. It also has many businesses using their platform.

I wanted to use the Foursquare API to support the users in Asia. On the other hand, I wanted to use Yelp because it is widely adopted in the US. I could not decide which one to use. My advisor provided some excellent advice and suggested that I use both. Before this suggestion, I had really only considered using one. I decided to support both APIs and the app would choose which API to use depending on the locale of the user.

Both APIs are REST APIs and are used by making an HTTP network request. Each API offered various different endpoints. For both of them, I used the autocomplete endpoints. Implementing support for both APIs required some object-oriented design. I started by creating a class called RequestModule. This class was designed to handle the TableView delegate methods for the SearchController, the filtering for the SearchController, and making the API calls to get the suggestions. I implemented functionality for the Foursquare API first. That helped me understand what I would need in order to handle both Foursquare and Yelp without having to write redundant code. The biggest challenge for being able to support both APIs was due to using a JSONDecoder. It decodes the data being received by the API. The JSONDecoder relies on taking a custom defined struct type as a parameter. This struct type tells the decoder how to decode the data. The responses from Foursquare and Yelp were different, and I didn't want to check which API was being used before being able to decode the data. I solved this problem by using object oriented design.

I made two different classes for each API, FourSquareClient and YelpClient. Both of these classes inherited from a parent class Client, since they had some common functionality. This design worked really well because I was able to perform the same actions with both APIs and could customize each one. However, I still had to add to the design because I wanted functionality in the parent that is implemented differently in each child. In other object-oriented languages, this can be accomplished with an abstract method. I discovered that Swift does not have abstract methods. Instead, I was able to solve this problem by using a protocol. A Swift protocol is equivalent to a Java interface. I ended up creating a protocol called ApiClient. Half of the methods were implemented in Client and the other half were implemented in FourSquareClient and YelpClient, which each implemented the protocol. This allowed me to use methods in both the parent and children.

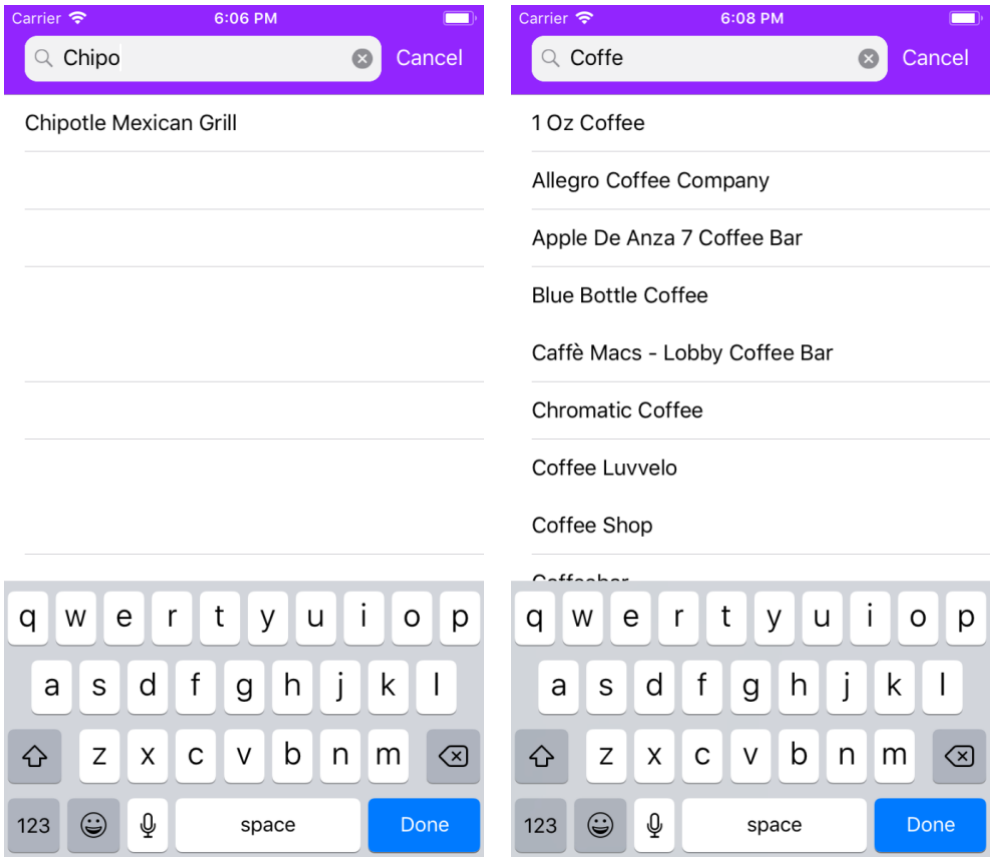

Figure 6: Giving suggestions to the user with a UISearchController

After working with both APIs, I wanted to do a comparison of them. The main differences between them can be found in Figure 7. Foursquare allows significantly more requests per day then Yelp. This shouldn't be a problem for users because once they add a place to their list of favorites, it will be saved. Foursquare allowed a limit parameter to specify how many results it would return. This customization allows the developer to request more or less results. On the other hand, Yelp did not have a parameter like that. Foursquare also had various options for specifying the geographical region to search. This is nice because requesting location from the device can use a lot of resources depending on the accuracy desired. The other parameters allow a developer to search a more general area. For example, the request can ask for results near a city. Or in a specified region defined by an upper corner and lower corner coordinates of a region. On the other hand, Yelp requires a latitude and longitude and that's it. Overall, Foursquare is a better API, but I still want to support both.

API	Foursquare	Yelp
Usage	99.5k Calls / Day	5k Calls / Day
Limit for Returned Results	max = 50	No 'Limit' parameter
Geocoding parameters	latitude/longitude, near city, upper corner/lower corner of region	latitude/longitude

Figure 7: Comparison of the Foursquare vs Yelp autocomplete APIs

Future Work

I could iterate on the new feature I added which provides the user with suggestions of places from Foursquare and Yelp. Close By Places will use Foursquare or Yelp depending on the locale. The locale is part of the user's system settings. It does not represent a general physical location of the user. In a future version of the app, I might change this so that Close By Places will pick an API depending on the time zone the user's device is currently using. I have also considered adding a setting that would allow the user to select a preference for where they want the suggestions to come from.

I also want to include some features that take advantage of the haptic feedback on some of the new iPhones. The new iPhones include a haptic engine hardware component that gives feedback like a vibration. I could give feedback for certain actions in Close By Places. I have also considered adding 3D touch shortcuts to allow the user to find a favorite place by 3D touching the app icon from the home screen.

I have also thought about adding emoji to Close By Places. Emoji are very popular and I think they could be used as symbols in Close By Places. I might include them either as ways to assign places in categories or displaying them in a marker on the map.

Reflections

If I could do this project again, I would work on it add a steadier pace. I had a lot going on with other classes and job searching. If I stayed on track, I could have implemented some extra smaller features or a more custom UI.

Conclusions

Overall, I accomplished the main goals I had for this project. I was able to improve parts of the app as well as implement a couple new features. I really enjoy developing apps for iOS. I continued to build upon my knowledge of Swift and enjoyed using it while working on this project.

Appendix

Analysis of Senior Project Design

The following sections are responses to the Computer Engineering department's analysis of senior project.

Summary of Functional Requirements

Refer to **Abstract**.

Primary Constraints

The most difficult part of working on this project was implementing the autocomplete suggestion feature. Switching over to a UISearchController was more difficult than I anticipated. This was because I was using the SearchController that wasn't originally on top of a TableView. It was also challenging to design for using two different APIs. Using object oriented principles, I was able to integrate both of them. For a more detailed discussion about this, see *Favorite Place AutoComplete Suggestions* in the **Key Design Decisions** section.

Economic

Since this project was mostly software based, there were no costs for components of the project. There was one cost of $99 for the annual Apple Developer Program Membership. This membership is required to release apps on the App Store. Close By Places was on the App Store before the start of this project. Version 2 was released on the App Store during this project. Close By Places was tested using the built in Xcode device simulator. It was also tested on my own personal devices. There was no cost associated with this additional equipment. The original estimated development time is summarized below:

- Updating to new version of Swift – 3 weeks
- Refactoring old code – 3 weeks
- List reordering – 3 weeks
- Resubmitting to App Store – 2 weeks
- Autocomplete suggestion feature – 5 weeks

This adds up to 16 weeks. Two quarters is 20 weeks. I thought that would give me some time in case something took longer than I expected or some time to add a few small extra features. The actual development time is summarized below:

- Updating to new version of Swift – 3 weeks
- Refactoring old code – 4 weeks
- List reordering – 3 weeks
- Resubmitting to App Store – 2 weeks
- Autocomplete suggestion feature – 10 weeks
    - UISearchController – 4 weeks
    - API Integration – 6 weeks

This summary doesn't add up to 20 weeks because I worked on some of the tasks in parallel. It is a good thing I gave myself some extra time. Refactoring the old code and implementing some of the new UI features took a little longer than I expected. Implementing the new suggestion autocomplete feature took a lot longer than I expected. I faced a few challenges while changing the UI and integrating the APIs.

Manufacturing

There is no manufacturing associated with this project. The app is distributed by the iOS App Store.

Environmental

This app has no environmental impact.

Manufacturability

This app is entirely software and does not require manufacturing.

Sustainability

There are two main challenges with maintaining Close By Places. The first challenge is the Apple Developer Program membership. This membership cost is annual. In order to keep the app on the App Store for users to download, I have to retain my membership. The second challenge with maintaining Close By Places is the evolution of the Swift programming language. Swift is a new programming language as it was announced in 2014. It is also open source so that the software community could help develop it. Since Swift is not that mature yet, new releases of the language can bring significant changes. This will present a challenge for me to maintain Close By Places. Xcode helps with this and can help migrate old Swift code to the current version. Changes in Swift also lead to changes for any 3rd party software dependencies. That is why I removed some 3rd party software dependencies from the app. Issues caused by both of these problems will need to be resolved before updating the app.

Refer to the **Future Work** section for a discussion on possible future updates to Close By Places.

Ethical

Close By Places is on the App Store. The source code is in a private repository on GitHub. This is to prevent someone from copying the code, slightly modifying it and then putting it on the App Store.

Health and Safety

Close By Places is grouped in the Navigation category. The only safety issue could arise if a user is trying to use this app while driving. If the user is trying to find a place already saved, then it should be safe. This app is meant to help the user quickly and easily find one of their favorite places and should not be that distracting.

Social and Political

This app has no social or political implications.

Development

I had experience working with Swift and Xcode prior to this project and did not independently learn any new skills.