FuelMe

COMP4905 – Honours Project

Antony Tamer
Antony.Tamer@carleton.ca

Supervised by: Tony White, PhD

Carleton University
School of Computer Science
5354 Herzberg Laboratories
1125 Colonel By Drive
Ottawa, ON, K1S 5B6
Abstract.

Natural gas has been around for forever. It was first discovered in America in the 17th century. The first really practical car ever invented ran on gasoline. Ever since then, gas has become one of the most important natural resources that drives our society’s economy worldwide. The greatest way that humans interact with gas, is by filling up their transportation vehicles. It happens so often, that it effects everyone with a vehicle and with that being said, it raises certain problems that we as humans face everyday. In this paper, I introduce a newly designed application as a solution to a long lasted problem involving filling up gas from gas stations. The readers will acquire knowledge regarding basic software definitions, design patterns, architectural styles, functional and non-functional requirements, basic user interface designs associated with the application and tests conducted on the product to analyze important aspects of the software such as the usability. By the end of this paper, readers will have learned of the motivation and rationale behind the app, the methodologies and approaches that were used and they will understand the results achieved.

Acknowledgments

Before delving in to the main topics of this research paper, I’d like to thank all the individuals who were willing to participate in the sets of experiments used to support arguments and ideas presented in this paper. These people have signed a consent form to remain anonymous for the sake of this research and they took time out of their day to volunteer in filling out surveys and getting involved in use case studies. Their insights and expertise had a great impact on the research and final product. So once again, a big thank you goes out to all those people.
# Table Of Contents

Introduction .................................................................................................................. 5

Rationale ....................................................................................................................... 6

Functional Requirements ............................................................................................ 8

Non Functional Requirements ..................................................................................... 8

Background .................................................................................................................. 9

  Xcode ......................................................................................................................... 9

  Google Places .......................................................................................................... 10

  AWS/Parse .............................................................................................................. 10

Related Work .............................................................................................................. 11

System Design ............................................................................................................ 12

Architectural Style ...................................................................................................... 16

User Interface/Design ................................................................................................. 21

Test ............................................................................................................................. 25

Results ......................................................................................................................... 30

  Issues Encountered ................................................................................................. 30

  Achievements .......................................................................................................... 31

  Future Work ............................................................................................................ 32

Conclusion ................................................................................................................. 35
List of Figures

Delegation Pattern Model ................................................................. 13
Protocol Function .............................................................................. 14
Delegate Implementation of protocol ............................................... 15
Architectural Style Representation .................................................. 17
Parse configuration ............................................................................ 19
Million Dollar Homepage ................................................................. 22
Center Stage design ........................................................................... 23
Hamburger Menu Example ............................................................... 24
Time Taken for Users to Accomplish Task bar graph ....................... 26
My Location Button ............................................................................ 27
Direction Tip ....................................................................................... 28
Time Taken for Users to Accomplish Task after Modification .......... 29
Information Button to Rate ............................................................... 33
Rating Interface .................................................................................. 33
1.0 Intro:

Your on your way to work and BAM! The gas light in your car turns on. There are numerous gas stations on your way however, which one will you get the most bang for your buck from? One gas station might be cheaper, the other might sell higher quality gas, another may have better services, or maybe you’ll go to the nearest one, the one that is the least out of your way. We all share relatively similar problems when low on gas, thankfully, FuelMe is a solution to all these difficulties. FuelMe is an iOS social media application that locates local gas stations within your vicinity while sharing the exact locations of these gas stations. It shows you the closest gas stations while notifying you of the most convenient ones. The social media aspect is convenient for users as it allows them to create gas related posts and read other user posts that will benefit them by providing them with knowledge. This knowledge can relate to understanding the most advantageous gas stations you should be filling up at, or insight on what time of the day and what part of the city drivers should be filling up at to get gas at its cheapest price. This application targets anyone who regularly drives a car. It helps by locating them to the nearest gas stations along with providing them with great insights (i.e gas prices, gas quality, etc) about local gas stations via other user posts.
1.1 Rationale

Before starting this project, I wanted to see how much in demand an application like this actually was. The reason for this is because I was building this application for more reasons than learning purposes. Not only was I building this application for the purpose of my honours project, but I also had plans and high hopes for this application in the app store. In order to analyze its potential, I’ve created a simple survey. This survey is to rest assured that there exists a problem in which my application will solve. The participants of this survey were males and females of all ages who drive regularly (minimum once a day).

The following are the questions asked in the survey:

1. How often do you fill up gas?
   A. More than once a week
   B. Approx. once a week
   C. Once a week - once every two weeks
   D. Less than once every two weeks

2. Does a minimum difference of 5 cents/litre matter to you?
   A. Yes
   B. No

3. When needing to fill up gas, do you go to just any gas station (i.e the first one you see)?
   A. Yes
   B. No

4. Do you have a preferred gas station?
   A. Yes
   B. No
5. In your opinion, is gas too expensive?
   A. Yes
   B. No
   C. I don’t know

6. Do you know how to save money on gas?
   A. Yes
   B. No

7. Do you know a site/app that provides insight (excluding the price) on gas (stations)?
   A. Yes (it’s called: ____________________)
   B. No (100% chose B)

8. Would you/do you browse and read through this site? (follow up on question 6)
   A. Yes
   B. No

9. Do you contemplate which gas station to fuel up at?
   A. Yes
   B. No

10. Does gas quality matter to you (Would you spend 5cents/Litre extra for better quality gas)
    A. Yes
    B. No

The highlighted fields are the ones that were most frequently picked. After analyzing this data, I came to a conclusion that people will want to use an app like FuelMe. From this survey, I
learned that filling up gas is definitely a regular routine to many people. It happens frequently enough to the point that it is fair to say that filling a car’s tank has an impact on most people’s lives. Whether the impact is primarily related to the expenses of gas or just the time taken out of one’s day, filling up a tank can have a negative effect one way or the other. Also, after analyzing the survey, it is clear that as expected, most people care about how much money they spend on gas and that they try to pick gas stations that they assume would have better gas quality to price ratio. Last but not least, all the contestants who took the survey stated that they do not know of a website or any app that provides great insight on gas whether it may be through companies or the public, however most people would like it if such an application was available. After inspecting the answers given from the participants, I was assured that FuelMe was a great app idea that could actually solve every day struggles experienced by anyone who regularly drives a car.

2.0 Functional Requirements

Before creating this application, I figured that it was a good idea to come up with a collection of functional and non-functional requirements that FuelMe will have. The functional requirements define what the system must do. The functional requirements that needed to be fulfilled by FuelMe are as follows: allow users to register/login, determine the user’s geographical location via GPS, display and specify gas stations near the user’s vicinity, specify directions to the tapped gas station, display the distance in kilometres and time taken to get to the tapped gas station, list all user posts, allow users to like/dislike posts, permit users to follow/unfollow each other and last but not least, FuelMe should provide a simple user interface to allow users to post their thoughts on gas station related topics.

2.1 Non-Functional Requirements

The non-functional requirements define the constraints that apply to the system. These requirements “specify criteria that judge the operation of a system, rather than specific
behaviours’’ [Ulf Eriksson, 2012]. Essentially, it describes how the system works. FuelMe fills several non-functional requirements including: 1) usability. When the application loads, users are directed to the login page where they can login with their credentials or create an account. After logging in, the system segues straight to the map view where users see most functionalities of the app and will therefore easily learn how to interact with the application. Another non-functional requirement that is fulfilled is 2) performance, more specifically speed and reliability. It was difficult to measure the overall speed of the application but I found that the response time of all features was fairly quick. FuelMe, when run on a regular iPhone 6 (my phone), ran with very minimal delays with an average load time of one to two seconds. Loading the gas stations from Google Places as well as posts from the server (which will vary when more posts are uploaded) was quick and only required the transfer of very few bytes of data per request. Another non-functional requirement that was focused upon was 3) capacity, specifically the maximum size of the database that stores user information and user posts. When choosing the company to host the server, I chose Amazon Web Server (AWS) and one of the main reasons why I chose them was because there was no limit to how much data you could store in your server. The server dynamically allocated more space whenever needed. Last but not least, 4) Security was greatly focused upon when creating this system since I’d want to be certain that user information was not easily accessible. This was the second reason why I chose to work with AWS which has multiple built-in security features and security groups that act as virtual firewalls to control traffic to the database on the cloud. That way, it ensures confidentiality, data integrity and authentication.

2.2 Background:

Xcode:

To create this application I used Xcode 8, an apple platform for developers to create iOS apps. The software provides an interface to visually edit the app’s storyboard. The storyboard is

---

“a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens”² [Anon, 2013]. Each scene on the storyboard is represented by a view controller where developers can design the user interface and create connections through segue objects. I programmed the entire project in Swift 3 which is one of two major languages (excluding other versions of swift) thus far, used to create iOS apps. Swift comes along with a standard library that helps solve complex problems in a simple and higher performance kind of way. Similar to other languages, Swift 3 comes with a base layer of functionalities such as fundamental data types, common data structures (i.e. Array, Set, Dictionary, etc), global functions (i.e. print(), abs(), etc), protocols and more. These functionalities facilitates the way software is created through Swift 3 and with them, developers are able to more effectively build iOS applications.

Google Places:

In order to dynamically display all of the gas stations around the user’s current location, I used Google Places API. To use APIs provided by google, I retrieved an API key from their site. It allows me as a developer to easily communicate with google services along with other services that they are integrated to. I embed the API key into my program and got rid of all places that are not marked as “gas stations” so that annotations on the map view landed only on gas station locations.

AWS/Parse:

In terms of the social media aspects in which FuelMe provides, I used Parse server which is open source and free to use. Parse server allows me to use MongoDB as a database which is used to store all information relevant to the application such as user information and shared content such as user posts. The reason why I chose Parse server is because it comes with many great features. Unlike Firebase (another widely used server), Parse provides the owners with a neat dashboard interface where their users can have a visual representation of the database and

make changes directly to them. Another great feature about Parse is it already does some error checking for you. For example, when registering users, I prompt them for their email and Parse automatically throws an error if the email is non-existing. In order to host the server, I use Amazon Web Services (AWS) with the elastic compute (EC2) cloud package. I used AWS for its enhanceability and reliability. Many very well-known companies such as Netflix and Kellogg’s use it as their backend service. In terms of enterprises, “Netflix was the most prominent early user of AWS, adopting it in 2009” [Benjamin Wootton, 2017] and “the ongoing pressure from Netflix, combined with Amazon's willingness to improve its service and meet its customers' requirements, pushed AWS to develop into the full, enterprise-scale integrated set of services that it is today”3 [Benjamin Wootton, 2017]. This means AWS had to have worked really hard to improve its services and to gain the trust of multiple big and successful companies. The EC2 package provides secure, reliable and resizable computing capacity in the cloud. It is inexpensive (comes with a one year free trial), easy to start and most importantly it gives owners complete control over their server-side computing.

The work objective was therefore to use apple’s Xcode 8 platform, Google’s APIs and Parse Server along with Amazon Web Services to create an application that can fulfill all the functional and non-functional requirements that were described above, given the time restriction.

2.3 Related Work

FuelMe is a combination of many related works. It contains certain functionalities that are present in already existing applications. A couple of the most related works are Google Maps and an application called Gas Buddy.

---

Like Google Maps, FuelMe can show user location and nearby gas stations. Google Maps is also able to specify directions to any given gas station that their API can pick up. In order for Gas Buddy to specify directions, it’d need to open the mobile’s default maps application and show users directions through it unlike FuelMe that will display directions directly through the app.

A feature that exists in Gas Buddy is that it can also include the amenities. For example you can filter on which gas stations you want listed near you based on the hours it is open or whether or not they include washrooms, ATMs, car washes, etc. Another functionality that exists on both Google Maps and Gas Buddy, but not FuelMe, is that they include gas prices. These prices are added by the active users of the application. So, once the users drive by a gas station, they can upload the price associated with the given station and this allows for all users to know the most recently posted price of any given gas station.

My application differs from these related works in many different ways. These applications that were previously described are solutions to many problems but they do not solve the problem that my application will attempt to resolve. Since FuelMe is strictly dedicated to gas stations, it makes finding and getting directions to any given gas station much easier compared to any similar work. Directions are displayed through a single click on an annotation. Another major difference is the fact that FuelMe is a social media app. None of the applications mentioned above include a platform to allow users to connect and socialize about important issues regarding gas/gas stations. FuelMe is not only convenient in helping users get to a gas station, but it also allows them to learn about concepts relating to them. That way, users will acquire more knowledge about something they do regularly!

3.0 System Design:

Design patterns are solutions that can be reused on common problems related to software design. It’s like a template to help developers more easily write understandable and reusable
The design pattern in which this whole application revolves around is mostly the delegation pattern (sometimes known as the subject/observer pattern). The delegation pattern is one of the most common design patterns in iOS because it is “heavily used by Apple's frameworks and even the simplest iOS application leverages delegation to do its work”\(^4\) [Bart Jacobs, 2015]. In this pattern, “the delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it”\(^5\) [Anon, 2015]. Therefore, this pattern allows for a class to give some control to another class by handing it some of its responsibilities. This design was selected because it best fits the system design since in many cases, objects in the program need to act on the behalf of others in order to effectively achieve the main goals of the app. With swift 3, I utilize an abstraction layer called protocol (similar to abstract classes in Java) to achieve the delegation design pattern. These protocols are abstract because there’s no implementation details provided in their declarations. Figure 1.0 illustrates the important factors involved in the pattern and the way in which the different classes communicate.

---


Here is an example that is provided in order to furthermore explain this iOS pattern:

The delegation pattern can be clearly represented in the map view of the app that is shown as soon as users login. The map displayed on the screen is of type MKMapView which is its own class and responsible for providing many protocol methods to manipulate, in anyway, the iOS map view. When the map view loads up, users will see custom annotations representing gas station locations and an important feature they’ll notice are the events that take place when they tap on the annotations, including a line displayed on the map showing users direction. This means that Swift recognizes when users have tapped on an annotation. The way this was done, is through the delegation pattern. In order to make changes to the map view (add a poly line to the map view) once users tap on an annotation, we’d need to give the current class (the class that has the map displayed) access to the MKMapViewDelegate attributes and methods so that we are able to manipulate the things that appear on the map such as customized annotations and poly lines that show users directions to a selected gas station location. This class is known as the “delegate class”.

The way the program provides these customizations is by using protocols from the MKMapViewDelegate class. Figure 1.1 shows the protocol function that is referenced by the delegator class. Notice how there is no implementation to the function provided in the delegator because it is a protocol.
Many protocols like the “didSelect” function shown in Figure 1.1 are used in the delegate class (class that displays the map) in order to provide implementations to them. Figure 1.2 shows an implementation given to the didSelect function.

```swift
//Method that gets called when you select an annotation
func mapView(_ mapView: MKMapView, didSelect view: MKAnnotationView) {
    print("Annotation tapped \{(view.annotation?.title)\}
    destination.append({(view.annotation?.coordinate)!!}
    drawLine(source: srcPlaceMark)
}
```

**FIGURE 1.2 - Delegate Implementation of Protocol: Shows “drawLine” function being called every time users tap on an annotation to display directions from the user’s current location, to the gas station destination.**

Without an actual implementation, the “didSelect” function will still be called every time the users tap on an annotation, however nothing would actually happen.

As part of the design I’ve grouped together everything that was related and separated the unrelated functions to properly encapsulate my code. Encapsulation is used in object-oriented programming to “[bind] together the data and functions that manipulate the data, and [to keep] both safe from outside interference and misuse”[6] [Anon, 2017]. Every different screen presented on the system has its own class to hold attributes and functions corresponding to it. The main “ViewController.swift” class represents the main screen containing the map that users go to in order to find a nearby gas station. There are a few TableViewController classes that also provide an implementation to protocol functions to handle the listings of users who’ve created an account and user posts. There are also separate classes that handle user logins and signups as well as a separate class that takes care of incorporating the google places API to the application which allows us to place the gas station annotations in the correct locations on the map.

---

This design was selected because by defining a protocol that encapsulates the delegated class’ responsibilities, we are easily able to use functions from some built-in classes of iOS to accomplish many functional and non-functional requirements of our app. Also, it allowed for us to keep related code in the same class and separated from the unrelated code while still being able to allow one class act on behalf of the other. Before creating the application, while still trying to decide on a proper design, I was going to use “composite” as the design pattern but it required the composition of multiple objects as one which wasn’t a great idea since there isn’t really too much data that this app needs to handle. Also, I didn’t use the strategy design pattern because for this app, it is unnecessary to define a family of algorithms to make them interchangeable and there’s no need to completely change any sort of behaviour at runtime, which is what the strategy design pattern is really meant for. Therefore both the composite pattern as well as the strategy design pattern were discarded.

3.1 Architectural Style

Before developing this application, much thought was put in to determining which type of architectural style it will implement. This is important when designing an app because “an architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems” [Anon, 2012]. This means that when developers are strict with the kind of architecture that their system implements, it could save them a lot of work and effort in the long run. The architecture used for this application is a blend of three distinct architectures: model-view-controller, object-oriented and event based (where methods are implicitly invoked by publishers using Swift’s delegation pattern). The aspect of client-server interaction is also very important within the application, so it also can be described as taking as a three-tiered architectural pattern of a state logic display.

---

In terms of justification, applications developed on Apple’s iOS platform follow a model view controller pattern where the components are models, views and controllers. The models contain/update data for the view to use in rendering interfaces for the user of the application. The controller is effectively the region of the application where all of the logic relating to individual views is performed, and where the controller manipulates the model in addition to interacting with the view to respond to certain events. Figure 2.0 displays a visual representation of the MVC model to help better understand the interactions within it. Underneath the image is an explanation of how the MVC is applied in the map view of the application where users can see nearby gas stations.

**FIGURE 2.0 - Architectural Style Representation:**

The **model** is responsible for information related to users. It also handles all the data and logic such as gas station locations.

The **view** is in charge of visual representation of user location, nearby gas stations and basically all objects that users can see and interact with (i.e interface).

The **controller** allows the view to display the correct information using the data from the model. It can also update the model based on the user’s interaction with the view, so essentially, it mediates between the model and view and handles user input.
In particular, what was found while working on the application was that in effect the controller also has the ability to receive events delegated to it by other application interfaces, such Apple’s MapKit and Google’s Places API, and the phone’s location manager which publishes events relating to where the user is located. Thus, the MVC architecture is intertwined with an event-driven (or event-based) architecture as well.

Similarly, object oriented programming is also very important within the context of our architecture as it clearly relates the models (such as Places on the map) to actual “Place” objects encapsulating data, and allows consistency across the application when referring to the concept of a “Place” on the map (which traditionally has a latitude, longitude, type, name, etc.).

While this may be delving into the design in some aspects, object oriented programming, specifically inheritance, was crucial to getting some features of the application to work at all. For example, one problem I ran into is that it was only possible to add annotations (the type Annotation) to the map and not Places (model data). Thus, when an Annotation (or marker) is added to the map, the concept of a Place and an Annotation are not at all related, so when a Place becomes an Annotation it loses some of its data (an Annotation cannot store the data associated with being a Place). So, instead of adding an annotation to the map it is instead easier to make a Place inherit from an Annotation, and thus be down casted from an Annotation to a Place whenever receiving an Annotation object during a published MapKit event.

It turns out that a model-view-controller architecture pattern for a mobile application is actually very much so a useful construct. I can technically fulfill every one of the functional and nonfunctional requirements, since an application’s interface is intuitively directly related to an associated controller, which can also be related to other components within the application.

Interacting between views is also very intuitive. A segue event is fired whenever a view transitions between one view to the next and this bridge event (prepareForSegue) provides data
as its parameters of the view it is transitioning from, and the view it is transitioning to such that any sort of interaction between the two can be made possible and quite easy to use.

**Style specific to Server Side:**

As previously stated, I use Parse as my server to store information that enables users to create accounts, login, write posts, view other user posts, etc. A common style of communication that parse uses is RESTful interactions, however this style was not needed to create this system. Thanks to iOS’ Cocoa design, I was able to use a language specific SDK (iOS Parse SDK) which opens a connection to the parse server using an app ID and client key specific to the server in order to interact with it. Figure 2.1 shows how and where these values were set. I was able to use the SDK methods to communicate synchronously or asynchronously to the server whether it was for manipulating the database or querying through it.

```swift
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]? ) -> Bool {
    // Enable storing and querying data from Local Datasstore.
    // Remove this line if you don't want to use Local Datasstore features or want to use cachePolicy.
    Parse.enableLocalDatastore()

    let parseConfiguration = ParseClientConfiguration(
        block: {
            (ParseMutableClientConfiguration) -> Void in
                ParseMutableClientConfiguration.applicationID = "b29e813c9e0ba3708bb4ec861a52eacf4e677be77"
                ParseMutableClientConfiguration.clientKey = "595891cbbd7d44e1c3ab3b2f9be074b6a194a516b"
        }
    )

    Parse.initialize(with: parseConfiguration)
```

**FIGURE 2.1 - Parse Configuration:** You can see that the `applicationID`, `clientKey` and `href` is set in the `AppDelegate.swift` class inside a function called “`didFinishLaunchingWithOptions`” which tells the delegate when the actual application has finished launching. By setting the appropriate values, the system is able to communicate with parse as soon as it has launched.

The database on the server keeps track of four important tables that store the necessary information to fulfill the important requirements. One of these tables is the “User” table that is used to keep track of all the users who’ve registered for an account on FuelMe. Whenever a user registers, they are added to the User table and are given an object ID which is unique to all users. This is important because in order to sign in to the system, the program queries through the User
table to see if the username and password that was typed already exists. Also, when registering
users, the application checks to see if the user already exists, and if so, users are given an alert to
notify them that a user with the same username already exists. Another table is the “Followers”
table. This keeps track of the following relationships among other users. It is important when the
application provides a list of other users signed up with the app to the current user, displaying to
them who they are following and who they can follow. It’s also important in order to see only the
posts of users you follow. The “Posts” table stores all user posts. It associates each posts with
users by using their object ID that is given to users as soon as they register and are therefore
added to the server. This is important in order to specify the users that posted each post. It also
contains a number field that keeps track of the number of users who liked the post. Last but not
least, the server automatically creates a table called “Session”. The session table keeps track of
every time users log in or register.

Thus, in terms of producing an application that would specifically be able to load gas
stations, direct users to them, find all of the ones within a certain radius of the user, filter them
appropriately, allow users to follow others, permit the users to post and like/dislike posts and
perhaps in the future let users submit their own gas stations (given that there are some gas
stations on private property for instance), the architectures described above makes this very
reasonable.

Of course, in terms of architecture, I had a limited amount of choices. The reason for that
is that developing on an iOS device was a relatively new process for me meaning I did not have
much experience. Naturally, I went with the “default” MVC architecture which made my life a
little simpler. My goal wasn’t necessarily to re-invent the wheel, or delve into some overly
complex architecture for a mobile application, but to understand what was already available and
learn something new in the process.
3.2 User Interface/Design

An important duty as a mobile developer (or any other developer) is to help users carry out a certain task in the most efficient way possible. To do this, I had to decide on which user interface/design I should use in order to help users focus on the important aspects of the app. The interface pattern used in FuelMe is “Center Stage”. This pattern is used whenever you have an important aspect in the app and want to make it the most visible to users immediately after they log in. When users are carrying out a task, they need a central area to perform actions all while being able to instantly access other features of the app. This is exactly how my application is built. One of the most important parts of the application is the map view which displays user location, nearby gas stations, directions and is accompanied with a toolbar for easy access to different view controllers.

The reason why “Center Stage” fits so well with this application is because having the most important aspect of the app in the largest subsection immediately disrupts the users tendency to get distracted by many possible options that can overflow the page. When I first read about this design, I instantly thought of a webpage that displays the exact opposite of the Center Stage.

Figure 3.0 displays a website made by Alex Tew that advertises numerous companies. The main reason why this website was made was to raise money for university tuition. The creator charged a dollar per pixel and people who wanted their companies advertised would buy pixels to display their company logo, in which users can click on in order to be directed to the company’s main website. And of course the more pixels bought, the bigger your logo was.
As you can see, things are all over the place in the webpage shown in figure 4.0. This website may have benefitted the big companies who were able to afford more pixels but the main issue lies within the user interactions. Users may use this website in order to search for unknown companies that may provide a specific service. However, the second users arrive to this webpage, their eyes will be all over the place and it will be near impossible to find exactly what their looking for. When creating an app that is centred around user needs, this is the perfect example of what not to do and the point of “Center Stage” design is to solve this issue.

FuelMe is a great example on how a great design should look like using “Center Stage”. I want users to quickly learn how the application works and by putting the map view on the main screen along with a toolbar to maneuver through the app, users will quickly adapt to the application. Below, figure 3.1 shows how the application is indeed associated with the center stage design.
FIGURE 3.1 - Centre Stage Design: Shows how the application looks like immediately after users log in. The most important part takes up most of the screen.

After some brainstorming, I was able to determine the most important part of the user interface, make it the largest on the screen, and arrange the buttons on the toolbar to help users carry out other desired actions. I draw user attention towards the map which displays their location and nearby gas stations by centre staging the map. Unlike, the million dollar homepage, the user’s eyes will not be scattered all over the place. This design guides the user’s eyes immediately to the most important feature of the application, rather than letting them wander through different pages with confusion. Users will be able to focus on the main aspects of the app and will assess the stuff in the periphery in terms of how they relate to what’s in the centre. This indeed makes the application much more usable and learnable.

Another UI pattern that is being displayed in figure 3.1, is bottom navigation. Bottom navigation is displayed within the toolbar in the bottom of the page, holding four buttons in
which users tap to explore between different pages of the application. When first designing the app, I had a difficult time choosing between the hamburger menu (three horizontal lines, usually at the top left of the page) displayed in figure 3.2 and bottom navigation. One of the main reasons why I included a toolbar at the bottom as a way of navigation is because this design allows users to interact with this app while using their thumb comfortably. This enhances usability simply because it reduces the user’s need to change the method of holding the device and requires less strain. Another reason why I chose the bottom navigation over the hamburger menu is because it requires only one click to change from one page to another thereby making it easier to explore and switch between top-level views in a single tap. Meanwhile, the hamburger menu requires more taps which according to cognitive friction, may cause frustration.

**FIGURE 3.2 - Hamburger Menu Example:** This figure shows the hamburger menu circled in red at the top right. It consists of three horizontal lines in which user click on in order to allow the page to slide over to show more options to navigate through.

Last but not least, to furthermore improve the UI design, I have provided customized annotations in the map view. Most gas stations have their own custom annotation which is represented by a pin along with the logo of the gas station hovering on top of it. This makes it
much easier on the users eyes when trying to locate which gas station to go to. However, since I was unable to cover all types of gas stations, I use a default logo to represent most third party gas stations.

3.3 Test

Software testing determines the quality of the system after creation. The quality of the application is very important because the better it is, the longer it will last, the more consistent it will be and the better it will perform. The main aspect that I wanted to focus on is the usability of the application. Usability testing requires “evaluating a product or service by testing it with representative users” [Anon, 2014]. The main idea of usability testing is to give the participants a certain task to perform while the tester (myself) sits back, listens, watches and takes notes. As I observe, I try to identify any problems users experience by collecting qualitative and quantitative data.

After completing my app (or after I thought I had completed my app), I made it a priority to manually test it by creating multiple scenarios and having potential users act through them with the motivation to complete the given task. Creating scenarios to have users act upon is great for testing usability because it can be used to articulate existing and envisioned work done with the application. One of the user stories were simple. It was: “The user logs in to the application, and searches for direction from his/her current location to any nearby gas station”. Some quantitative data that I collected were how long it took users to log in to the system and how long it took them to get directions to any gas station.

It’s always a good idea to calculate task times because it is great for diagnosing usability problems. I even used the think aloud technic where users were saying what was on their mind while they performed the task because studies show that this may actually increase user speed. If

users take too long to complete a certain task, it usually means that there are problems with the interaction within the interface. During the usability test, I was focused on many different details. Below is a list of some of the most important data collected and characteristics that I was focused on:

1. Check if participants were able to complete the task at hand.
2. If task is completed, calculate time it took to complete the task.
3. Get user feedback and probe them after the task is complete. (Did it seem easy? Were they satisfied with the interface?, What didn’t they like about it?, etc).
4. Analyze the results, and see if it meets usability objectives or if changes must be made.

All the participants did not have any trouble logging in to the system. It is relatively straightforward and universal. However, things started to get a little tricky for the representatives when trying to get directions to any of the gas stations. It didn’t immediately strike the users that in order to display a poly line from their current location, to any given gas station, you simply tap on the annotation corresponding to that gas station. I had a total of eight representatives participate in my usability test. Below is a histogram representing the time taken for each user to accomplish the tasks.
FIGURE 4.0 - Time Taken for Users to Accomplish Task: Histogram representing the time, in seconds, taken to accomplish two tasks in the user scenario.

As shown in figure 4.0, the time taken to get directions to a gas station greatly surpasses the time taken to log in, however it shouldn’t. The reason why it should not is because getting directions is done with a tap of an annotation and should therefore be almost instantaneous compared to typing in your username, password and clicking “log in”. The task of getting directions was simply not as explicit as it should have been. When developing my application, I only thought of how I can make getting directions so convenient but I never thought of how it could be so implicit to the users using the application. I collected qualitative data through the participants’ feedbacks and through analyzing their thinking, since it was done out loud. It was obvious that getting directions was not so clear. I learned that the representatives would think that in order to get directions, they would need to click one of the buttons on the toolbar and noticed that most of them clicked the button shown in figure 4.1, which is associated with having the map centre on user’s current location in case users went too far off the map.

FIGURE 4.1 - “My Location” Button: This figure usually represents the user’s location but was mistaken as a button used to display directions from the current location to a given destination.

The fact that users took too much time to complete the user scenario indicates that there is a problem with the user interface and the interaction within it. I knew that changes needed to be made but the tough part was figuring out how these changes are to be made. After some brainstorming, I needed to conclude on the best way to make getting directions to any given gas station extra clear. Some ideas were to have a small label in the top corner of the map view instructing users how to use the functionality, another was having an information button that leads to another view to instruct users on how to use the whole application (including all the
other functionalities). The reason why I didn’t go with the first idea is because it looked messy and unprofessional. I also didn’t go with the latter idea because if an application like mine requires an instructions page, it clearly means it’s a confusing app that no one would want to use to begin with. Most application that require a page for instructions are applications such as games that introduce a new type of entertainment. Finally, I concluded on a great idea. Once users begin to communicate with this app for the first time and get the hang of it, it’ll be very easy to use any other time. I figured that users may need to be told how to retrieve directions only once. So, I decided to create an alert that pops up once users log in and are directed to the map view. This alert will instruct users what happens once they click on an annotation. However, the neat part about this functionality is that it only appears on the first time users use the application. That way, the pop up won’t get annoying and too repetitive. It’s a simple gesture to get directions and users only need to be told once how to achieve this goal. This will make users more efficient and effective while using the app. Figure 4.2 represents the pop up that appears in the map view.

![Map View with Alert]

**FIGURE 4.2 - Direction Tip: Representation of what happens in the map view when users use the application for the first time.**
After adding the little alert, I did another usability test with the same user story but different representatives. The time taken to log in was relatively the same as the first group of participants which was expected. However, the time taken to get directions after reading the pop up and clicking on the annotation, dropped by a decent amount. Figure 4.3 is a bar graph that shows the time taken for each user to complete the task after logging in.

![Bar graph showing time taken for users to accomplish task after modification](image)

**FIGURE 4.3 - Time Take for Users to Accomplish Task After Modification: Bar graph representing the time taken to display directions after log in.**

As shown in the figure above, the time taken to display directions from user location to the tapped gas station has greatly decreased after modifying the app. The average time has decreased from 7.9 seconds to 4.1 seconds. Another user story that was put to test to measure usability was: “The user logs in to the application to write a post about the most recent gas station they attended”. I used the same participants in this usability test as I did in the last one and the participants accomplished the task with ease, so no changes were made regarding that given scenario.
Aside from qualitative data (time taken to log in and get directions), I also collected quantitative data from the participants after they ran through the user scenarios. This form of data describes “information about qualities; information that can't actually be measured” [Anon, 2008]. Qualitative data was collected through direct interaction with the participants, more specifically through a structured interview where I had preplanned questions ready to ask them. Questions that were asked in order to learn more about the quality of the application were related to learnability and what they thought of the overall application as a whole (i.e. Did you find this application easy to use? Do you feel that an application like this one is actually useful? What kind of people do you feel would benefit from this application? Would you download this application and use it?). The qualitative data helped me learn that this system would indeed solve the problem that it is intended to solve since all participants’ answers were pro creation of this application.

4.0 Results

Issues encountered:

Throughout the development of this app, I have encountered many unimaginable issues. Most of the major issues encountered were due to the limited resources online regarding the problems I had. My biggest issue was related to the back end. I encountered this issue while trying to implement the “like” feature on users’ posts. Every post created by users has its own tuple stored in the “Posts” database on the server. The row contains key information such as the title of the post, the body, the author and the number of likes. I was having problems updating the “Likes” record which is an integer that stores the number of likes associated with the post. When I queried the object that I wanted to update, the system would give the following error:

“Optional(Error Domain=Parse Code=101 "Object not found." UserInfo={code=101, temporary=0, error=Object not found., NSLocalizedDescription=Object not found.})”. I had no

idea why I was getting this error and after a lot of research I understood the problem, however I found no solution to it. I ended up posting on a forum called stackoverflow asking for help regarding my situation. I finally understood the solution. Every object created in Parse has a column called “ACL”. This column is in charge of the kind of access you can have to the table. The kinds of accesses it controls is read and write. The problem was that the value “write” was set to “false” meaning it does not give me, as the developer, permission to overwrite information on the table. Due to the interface provided by Parse, I was able to set the value of write to “true” directly on the dashboard rather than through code. This was an issue I had due to my lack of knowledge on Parse, but thankfully, I was able to resolve it on time and stay on schedule.

Another functional requirement that I found difficult to implement was displaying a poly line on the map view that shows users directions from their current location to a tapped gas station annotation. There were quite a bit of documentations on it, however most of them are from previous versions of Swift and it wasn’t easy to translate it to Swift 3. Also I had issues figuring out how to update a poly line whenever the users are moving. This functionality simply required a lot of problem solving and debugging but after a lot of hard work, this functionality finally started working properly.

Achievements:

FuelMe is used to help users in need of filling up gas. Throughout the time I had to complete this project, I was able to achieve all the goals that I had planned ahead of time. The application displays user location, nearby gas stations, directions and the distance and time needed to get to the most desired gas station. It also stores user information in order to allow users to create an account which is used for taking part in public discussions on forums regarding gas stations. The application comes with a great interface designed to make the app learnable and easy to use.
I built FuelMe, an iOS application that contains all the functionalities and proper interface needed for it to be useful, efficient and usable. People in the real world are continuously facing difficulties when it comes to filling up their tank. These difficulties include choosing the right gas station (based on price, quality of gas, etc), locating any given gas station and more! Even till this day, a lot of drivers are still blindly filling up at any nearby gas station without even realizing the advantages of buying gas from others while other drivers are still contemplating which gas station to go to. FuelMe is an app that solves this problem. It warns its users of the importance of properly choosing which gas station to attend and it allows anybody to make the right decisions about filling up gas. It’s the first app of its kind to provide users with an interface in order to communicate about gas station related subjects. Through participating and reading forum discussions, users will finally understand the importance of wisely choosing which gas station to fuel up at and they’ll use the map view interface to get there.

Future Work:

With the delegation design in place along with encapsulation, the system is better able to accommodate for change. There may be times where implementing new features to the application would be a great idea. For instance, a feature that I’d like to add in the near future is allowing the clients to rate gas stations as they use them. This would allow users to more easily choose between nearby gas stations. The rating feature may require a whole new screen and to stick to the design pattern, I can easily set up a new view controller to display the interface for rating gas stations and we’d link this view controller to a whole new class called Rate.swift. This class would contain the attributes and required functions needed for allowing users to provide feedback through a 5-star rating system.
To implement the rating feature, the user ratings will be stored on a server in order to average out all of the ratings that users have placed for a particular gas station. I can store an array of integers that vary from one to five that represent all the ratings a certain gas station has and can compute the average. The server that will be used is Parse, hosted by Amazon Web Server, which is the same server and host used for the social media aspect of this application.

If ever I needed an already existing class to access the features to this new class, thanks to the design pattern and architectural style used, I can easily create an instance of the “Rate” class in the already existing class in order to have control of its methods and attributes. If I needed to

FIGURE 2.0 - Information Button to Rate: Shows the map view. Notice that when users click on an annotation, an “information” button appears. There, users will be able to navigate to a rating view controller.

FIGURE 2.1 - Rating Interface: The view controller that users will be directed to in order to rate the gas station.
use a delegator, its delegation can be set equal to the current class. This way, the existing class that holds the “Rate” object can provide the ratings of gas stations through through methods and variables in Rate.swift.

Another functional feature that I will look into implementing is allowing active users to post gas stations in which they may have happened to stumble upon and that aren’t picked up by google places’ API. I’d need a separate view controller that would allow users to take a single picture of the gas station and submit it. For this I’d need a basic black box for a server that could take in POST requests, save the image (of the gas station) and location data appropriately and serve back all of the added gas stations as JSON data. Pictures of the gas stations would need to be taken as proof and since the gas stations would have to be approved first to avoid using the API for malicious intent, an interface would need to be developed for admins (me) to approve/delete gas stations on the go to ensure accurate results subject to my opinion of the app. The architecture for this would be a simple program with subroutines (connect to the database, output records, check if the user is logged in, compare against some hard-coded password in the PHP script).

A way that this system will evolve in the future is whenever the number of gas stations all over the world increases. The only external API used is the Google Places API to extract the gas stations over the world and the way we can support these changes is by renewing the API key about every month to make sure our application stays updated in order to keep up with added and removed gas stations for quality assurance. This API is located in a separate swift file that handles everything that has to do with embedding the Google Places API to dynamically locate gas stations. Since we encapsulated it in such a way, the small changes will only be made in one file.

Last but not least, in the future I would work on designing more custom annotations for each gas station company instead of assigning them the third party gas station. The more custom
annotations there are, the easier it is for users to distinguish the difference between the gas stations in their vicinity.

4.1 Conclusion

As the research has demonstrated, the development of FuelMe as a whole was a success. There exists a problem that every day individual stumble upon, some of them without even knowing it. When launched on the app store, the plan is that FuelMe will serve to fix these problems in the most effective way possible. This report highlights in detail the motivation towards the application, the methodologies used in creating this project and the results achieved. The design patterns, architectural styles, user interface designs and tests performed on this application played a huge role in making this application as successful as it is. Of course, just like in many other applications, there is always room for improvement and I will continue to add new features and create new tests to furthermore polish this application in order to make it as good as it can get before hitting the app store. Will more individuals realize how critical it is to choose the right gas station? Soon, this mystery will be unravelled.
References:


