

Riding Mechanic

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Dezheng Wang

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May 2017

Riding Mechanic

By Dezheng Wang

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

<<Primary adviser's name>>
Examination Committee Chairperson

Date

<<Examiner's name>>
Examination Committee Member

Date

<<Examiner's name>>
Examination Committee Member

Date

Abstract

Dezheng Wang, “Riding Mechanic”, Master of Software Engineering, Feb 2017 (Elliott Forbes).

On Board Diagnostics II (OBD-II) was first adopted for all cars and light trucks since 1996 and lots of OBD-II products had come out in the past few years. Some early developed OBD applications did not make it very clear what problems you were confronted with because most developers had little knowledge about On-Board Diagnostics systems. As time went by, newer apps had become more and more comprehensive. There were also high-performance OBD apps that provided drivers with detailed information of cars, but users need to pay for it. So, this document would talk about how to develop a free mobile OBD-II application to gather detailed diagnostic information over a long period that could be used to monitor a vehicle’s operation and discover potential concerns before they became serious faults. The application could also help drivers identify the effect certain configuration changes have on their car’s operation by recording driving behaviors form one route to another route.

Acknowledgements

I would like to express my sincere thanks to my project advisors Dr. Elliott Forbes for his patience, kindness and valuable guidance. I would also like to express my thanks to the Computer Science Department and the University of Wisconsin-La Crosse for providing the computing environment for my project.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	vii
List of Tables	viii
Glossary	ix
1. Introduction	1
1.1 Onboard Diagnostic systems	1
1.2 Onboard Diagnostic II scanner	2
1.3 Onboard Diagnostic Parameter IDs	4
1.4 Diagnostic Trouble Code	6
1.5 AT command set	7
1.6 Thesis outline	8
2. Requirements analysis	9
2.1 User requirement analysis	10
2.2 Functional requirements analysis	12
2.3 Non-functional requirements analysis	19
3. System architecture design	20
3.1 Goals and constraints of architecture	21
3.2 Key scenarios	22
3.3 Application overview	23
3.4 Key issues and solution	24
4. Detailed system design	27
4.1 Database design	27
4.1.2 Database table design	27
4.1.2 Database password encryption and decryption	30
4.2 Use case diagram	31
4.3 Class diagram	33
4.4 Sequence diagram	34
4.5 How driving distance was calculated	41
4.6 User interface design	42
5. System implementation	48
5.1 Login module	48
5.2 Registration module	50
5.3 Forget password module	51
5.4 Device connecting module	51
5.5 Trip information module	53
5.6 Health scan module	54
6. Testing	56

6.1 Testing plan.....	56
6.2 Unit testing.....	56
6.2.1 Testing for server module	56
6.2.2 Testing for client module	58
6.2.3 Code coverage.....	66
6.3 Integration testing	68
6.4 System testing	69
7. Conclusion and Future Work	72
7.1 Conclusions	72
7.2 Future Work	72
8. Bibliography	74
Appendices.....	75

List of Figures

Figure 1. Relationship between OBD-II scanner and project	4
Figure 2. High-level architecture	26
Figure 3. Entity relationship diagram	29
Figure 4. User table in database	30
Figure 5. Trip table in database	30
Figure 6. Use case diagram	32
Figure 7. Class diagram	33
Figure 8. Sequence diagram of login	35
Figure 9. Sequence diagram of registration	36
Figure 10. Sequence diagram of resetting password	37
Figure 11. Sequence diagram of connection	38
Figure 12. Sequence diagram of reading data	39
Figure 13. Sequence diagram of searching a trip	40
Figure 14. Sequence diagram of checking a car	40
Figure 15. Sequence diagram of sending initial commands	41
Figure 16. Example of using a tab bar	44
Figure 17. Example of using navigation bar	45
Figure 18. Example of using table view	46
Figure 19. Example of using an alert	47
Figure 20. Code coverage	67
Figure 21. data comparison in coordinate system	69

List of Tables

Table 1. OBD-II PIDs used in the application	5
Table 2. Meanings of a DTC' prefix	6
Table 3. How first DTC character was encoded	7
Table 4. How second DTC character was encoded	7
Table 5. AT commands used in this application	8
Table 6. Frequency of sending different OBD-II PIDs.....	34
Table 7. Test cases for the server side	58
Table 8. Test cases for session module	59
Table 9. Data comparison and margin of error	68
Table 10. Achievements of requirements	71

Glossary

Onboard Diagnostic systems

Onboard Diagnostic systems (OBD) is a vehicle-mounted system used to monitor the running status of engine components and working status of emission control system.

Onboard Diagnostic II scanner

Onboard Diagnostic II scanner (OBD-II scanner) is a tool used to help getting access to the data provided by OBD system.

Onboard Diagnostic II Parameter IDs

Onboard Diagnostic II Parameter IDs (OBD-II PIDs) are hexadecimal codes sent to a vehicle's Onboard Diagnostic system to request data.

Diagnostic Trouble Codes

Diagnostic trouble codes (DTCs) are alphanumeric characters saved by a vehicle's Onboard Diagnostic system to point out different types of faults.

Attention command set

Attention Command set (AT command set) is a command language used by modems to send different types of commands.

ELM327

The ELM327 is a programmed microcontroller produced by ELM Electronics for translating the on-board diagnostics (OBD) interface found in most modern cars [1].

1. Introduction

In our daily life, it became more and more difficult to image life without driving. When you must go to work in a different city, go to school far away from your house, plan to travel to another state with your friends or you just want to make transportation faster and easier for you and your family, taking a car was really a good choice.

We almost drive every day but we have little knowledge about our cars. In that case, it was a good solution to use on-board diagnostics systems to help us get close to our vehicles. It was possible for users to use an OBD app along with an OBD scanner to read data from the car. Most OBD apps in the market could provide drivers with cars' real-time information and diagnostic trouble codes along with what kind of kinds of problems. Instead of investigating the problems yourself, those apps simply told us what kinds of problems along with categories that they belonged to.

As mentioned above, my target was to develop a free mobile OBD app to gather detailed diagnostic information over a long period that could be used to monitor a vehicle's operation and discover potential concerns before they become serious faults. The app could also help drivers identify the effect certain configuration changes have on their car's operation by recording driving behaviors form one route to another route.

1.1 Onboard Diagnostic systems

On-Board Diagnostics, or "OBD," was first introduced in early 1980s. Early OBD systems were developed to remind drivers of losing control and occurring malfunctions. As time went by, On-Board Diagnostics II, an advanced version over early OBD models, was adopted mandatory for all cars manufactured in the United States since 1996, as required by the Clean Air Act Amendments. Since that time, OBD system became an important role in helping fight emissions of greenhouse gasses and engine failures.

On-Board Diagnostics is additional computer software that monitors the emission control and emission-related components/systems, along with certain engine components that provide vehicle operational information [2]. OBD systems can monitor both running situations of engine and working status of emission control systems all the time. Once over standard emissions were detected, OBD systems would give alerts to users. If any

systems in a car were broken, “Check Engine” or “Malfunction Indicator Lamp” light on the dashboard of the car. Meanwhile, OBD systems would store Diagnostic Trouble Codes(DTCs) which were related to those broken systems. By using a standard scan tool connected to vehicle’s OBD-II connector, an automotive technician can quickly know what the trouble is and where it occurs. This did save plenty of time for both drivers and technicians.

The biggest difference between OBD-II system and previous OBD systems is that OBD-II system is extremely strict with emissions. It can detect over standard emissions by monitoring power system and emission control system. OBD-II system also has unique features. The first one is that all data link connectors(DLC) in cars are standard 16-Pin OBD-II connectors which brings lot of convenience for people. The second one is that OBD-II DLC can help to transmit data between cars and scan tools. The third one is that OBD-II defines universal Diagnostic Trouble Codes(DTCs) and its meanings for all kinds of vehicles. The fourth one is that it has similar functions as car driving recorder (car DVR). The fifth one is that users can use scan tools to clear DTCs. The last one is that OBD-II can show those cleared DTCs. Although it is very efficient to monitor emissions by using OBD-II system, sometimes drivers don not accept or just ignore alerts from OBD-II system. For this reason, a more advanced version, OBD-III had been developed. The main purpose of OBD-III is to combine vehicle detection, maintenance and management. When over standard emissions occur, OBD-III will prevent users from keeping driving with high emissions. However, OBD-III techniques are still under developing. We may have opportunity to use OBD-III in the future.

1.2 Onboard Diagnostic II scanner

Since OBD-II scanner was a standard tool used to help getting access to the data provided by OBD system, it had been playing an important role in the process of reading data from a vehicle. An OBD-II scanner does not make any sense when it is used as a single object. In the other words, an OBD-II scanner should always work with an OBD-II system. But how an OBD-II scanner works with an OBD-II system. The answer is that the OBD-II scanner can connect to an OBD-II port provided by the OBD-II system. Then

the question is that does my car have an OBD-II system and corresponding OBD-II port. According to United States Federal Law, every car or truck on the road manufactured since 1996 was mandatorily required to have OBD-II system and OBD-II ports. You do not need to worry about that OBD-II system have different OBD-II ports. Since OBD-II systems were standardized after 1996, OBD-II ports had been standardized by SAE J1962. When users want to use an OBD-II scanner to connect to an OBD-II port, the OBD-II port must be plugged into an OBD-II scanner. After the OBD-II scanner plugged in, it began to act as a special server. There are two types of OBD-II scanners so that a server usually has two types. The first type is Wi-Fi server which can provide its own Wi-Fi signal. The other one is Bluetooth which can provide its own Bluetooth signal. Connection between a mobile device and an OBD-II scanner can be built easily by simply connecting to that scanner's Wi-Fi or Bluetooth. After building connections successfully, users can use an OBD app to read data from a vehicle.

Figure 1 showed how an OBD-II scanner acted as a medium between the application and a car. As shown in the picture, the OBD-II scanner had two components-one was ELM327 and another was a kind of Wi-Fi module which could send and receive Wi-Fi signals. ELM327 was the core part of an OBD-II scanner and it was designed to build connections between an OBD-II port and a standard serial port. So, it almost supported all standard OBD-II protocols and it had universal serial interface.

After the scanner plugged in, it began to act as a server and kept sending Wi-Fi signal. A mobile device also had a Wi-Fi module which was responsible for receiving that signal. Once connection had been built, the application used a socket to send commands to the Wi-Fi module on the phone through Universal Asynchronous Receiver/Transmitter (UART). The Wi-Fi module on the phone received those commands and could send them to the Wi-Fi module on the scanner by using wireless communication transfer protocol (WTCP). Then Wi-Fi module on the scanner received commands and sent them to the ELM327 through UART. Then the ELM327 used those commands to interact with OBD-II system in the car through the universal serial port. The OBD-II system could identify those commands (next part will talk how to do it) and quickly send responses to the

scanner. As soon as the scanner received responses from the car, it would send them to the application by using the same mechanism. The application would keep receiving messages. Once received, it began to analyze those commands and transfer them into universal data used in our daily life.

The reason why I selected Wi-Fi scanner was that users could not connect to a vehicle's Bluetooth and a scanner's Bluetooth at the same time. As we know, some people like to connect to the vehicle's Bluetooth to listen to music. If they used Bluetooth based applications and they could not listen to music any more. So, my choices were using an OBD-II Wi-Fi scanner and developing an OBD-II app to build Wi-Fi connection.

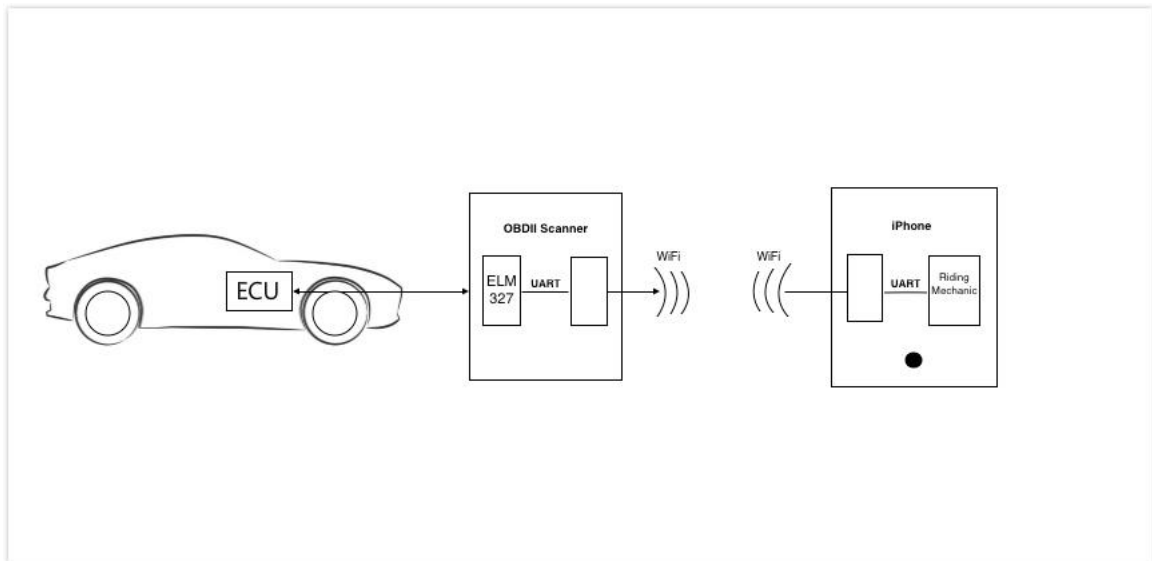


Figure 1. Relationship between OBD-II scanner and project

1.3 Onboard Diagnostic Parameter IDs

OBD-II PIDs are hexadecimal codes sent to a vehicle's Onboard Diagnostic system to request data. OBD-II PIDs were commands which were sent by a mobile device. SAE J1979 defines many standard universal OBD-II PIDs that were used by many car manufacturers, but car manufacturers also defined their own specific PIDs to their vehicles. This application would focus on those universal PIDs instead of all OBD-II PIDs because it was hard for developers to collect self-defined PIDs from corresponding manufacturer.

As it mentioned above, all codes that could be sent by the application were PIDs described in the latest OBD-II standard SAE J1979. There are 10 modes of operations defined in the standard, including “01: Show current data”, “02: Show freeze frame data”, “03: Show stored Diagnostic Trouble Codes”, “04: Clear Diagnostic Trouble Codes and stored values”, “05: Test results, oxygen sensor monitoring (non CAN only)”, “06: Test results, other component/system monitoring (Test results, oxygen sensor monitoring for CAN only)”, “07: Show pending Diagnostic Trouble Codes (detected during current or last driving cycle)”, “08: Control operation of on-board component/system”, “09: Request vehicle information” and “0A: Permanent Diagnostic Trouble Codes (DTCs) (Cleared DTCs)”. This application was aim to get current data and read diagnostic information from a vehicle. So, mode ‘01’ and ‘03’ were two most important codes for this project.

As defined by SAE J1979, OBD-II PIDs were consist of ASCII codes. There were many kinds of PIDs under mode ‘01’ while ‘03’ did not have any subset. Table 1 showed OBD-II PIDs had been used in the application. All those OBD-II PIDs were from by SAE J1979. The table just made some changes to the description.

OBD-II PIDs	Description
0100	Get supported PIDs [01-20]
0120	Get supported PIDs [21-40]
0140	Get supported PIDs [41-60]
0105	Get engine coolant temperature
010C	Get engine RPM
010D	Get vehicle speed
0110	Get MAF air flow rate
0142	Get control module voltage
03	Get stored DTCs

Table 1. OBD-II PIDs used in the application

1.4 Diagnostic Trouble Code

DTCs were fault codes detected by the OBD-II systems and those trouble codes would be stored by the OBD-II system. Users could use query tools on the internet to get diagnostic information associated with DTCs. OBD-II DTCs were alphanumeric codes which were consist of five characters, one letter followed by four digits (e.g., P1000). The first character always describes which system has that fault. Table 2 gave specific information of each letter's meaning [3].

Letter	System
P	Powertrain (Includes engine, transmission, drive shafts, differentials and the Final Drive)
U	User Network (Wiring Bus or UART)
B	Body (Includes A/C and Air Bag)
C	Chassis (Includes ABS)

Table 2. Meanings of a DTC' prefix

The second character was an integer that indicates whether the trouble code was an OBD-II generic code or not. If the second digit was zero, it was a generic code. If the second digit was one, it was a manufacturer specific code. The third character pointed out which subsystem in car had that fault. The fourth and fifth characters worked together and were responsible for the explaining that fault detected in that subsystem. Let's use an example of C0300. The first letter C indicates a chassis problem. The zero indicates that it is a generic code. The number three indicates that the failure is in the misfire monitoring system. The last two digits indicate that it is rear prop shaft circuit malfunction.

Referring to my previous argument, the application could receive diagnostic messages from a vehicle. Those messages need to be encoded to become universal DTCs. The prefix of DTCs were encoded as two bytes and the second character of DTCs were also described by another two bytes. Table 3 and table 4 showed how those two character were encoded [4]. The last three digits were hexadecimal characters and each of them could be either a number (0-9) or a letter (A-F). All three character were encoded by four

bytes and they followed the rules when a hexadecimal character was converted into a binary character.

First two bytes of a diagnostic code	First DTC character
00	P- Powertrain
01	C- Chassis
10	B- Body
11	U- User Network

Table 3. How first DTC character was encoded

Following two bytes of a diagnostic code	First DTC character
00	0
01	1
10	2
11	3

Table 4. How second DTC character was encoded

1.5 AT command set

AT command set was originally developed to control specific modern. As time went by, it became a commonly used command language with a series of short text strings which could be used to manage moderns' behaviors. Although AT commands and OBD-II commands (OBD-II PIDs) were both sent from the application, they were totally different. Since Wi-Fi connection had been built between a vehicle and a phone, users could use the application to send AT commands to change settings of the OBD-II scanner. As mentioned above, EML 327 was the core part of an OBD-II scanner. In addition, it was fully configurable with AT commands. An AT command was firstly created by the application and then it was transferred from the phone to the scanner by using UART and WTCP. Once the scanner had received that AT command, it could read it and know what to do next. It would also give a response to the sender to let it know whether the command works or not. In a word, AT commands were used to communicate

with the OBD-II scanner and OBD-II PIDs were commands used communicate with OBD-II systems.

Every AT command used by modem began with "AT" or "at" and following characters were different. In this project, the app would use AT commands to communicate with an OBDII- scanner. Table 5 showed some AT commands used in the app [5].

Version	Command	Description	Group
1.0	@1	Display the device description	General
1.3	@2	Display the device identifier	General
1.3	SP 00	Set protocol to auto and save it	OBD
1.0	Z	Reset all	General

Table 5. AT commands used in this application

1.6 Thesis outline

Thesis outline simply talked about the contents of following chapters.

- Chapter two showed how to analyze users' requirements from different perspective. Functions that could be performed by users were described in detail in functional requirements. Non-functional requirements were related to system qualities.
- Chapter three described both goals and constrains of system architecture design. Then it gave an overview of what the application would look like when it was completed. It also pointed out some key issues and its solutions.
- Chapter four presented detailed system design including database table design and use of UML diagrams to analyze user scenarios, relationship between classes and interactions between each component in the system.
- Chapter six showed implementation of main functions. It used advantages of pseudocode to make logic of each process clear.
- Chapter seven presented testing strategies and how they were used in testing. It also showed the process of testing and testing data included in each kind of testing.

2. Requirements analysis

This section lists specific requirements for Riding Mechanic. Requirements are divided into the following sections:

1. User requirements. These are requirements written from the point of end-users and they describe the functions that the product must have. User requirements are usually expressed in user stories.

2. Functional requirements. These are detailed specifications describing the basic functions the system must have. Functional requirements are defined by developers to satisfy user requirements.
3. Non-functional requirements. These are requirements defining characteristics the system should have. Non-functional requirements are also called system features (e.g., Usability).

2.1 User requirement analysis

In section one, we briefly talked about why OBD apps are needed in our daily life. This part would describe the requirements in detail from end-user's perspective. A smart vehicle could optimize its performance to reduce fuel consumption, show the best with considering current external conditions and detect errors in the engine. It was a little bit hard for users to maintain the performance of cars because users could only view some basic information (e.g., speed) which gave little help to user to improve the performance. What people should do if they want to get familiar with their cars without going to the auto shop? As mentioned above, users could use an OBD-II adapter to read information from a car's OBD-II system. However, to handle all kinds of data from OBD-II device users still need OBD-II apps to figure those categories out. The question was what capabilities users want those apps to have.

To communicate with a car, the very first step for users was to connect to an OBD-II scanner through Wi-Fi or Bluetooth. So, users want an OBD app to help them get access to the car. After building connection successfully, users could communicate with the car through the scanner. It was necessary for users to think about what kinds of information were needed so that users could take advantage of those information to help them. To improve driving habits, the best way was to analyze driving records to draw conclusions. So, users want the app to have functions of recording driving information. All data in driving records must be specific and accurate to guarantee correctness of those conclusions. Speed and RPM were two kinds of significant information in driving records. Although we could easily view speed and RPM on the dashboard in a car, they were not specific. So, users want the app to have specific speed and RPM in driving

records. In addition, driving time and driving distance were very important for users to know how long and how far they had been driving. So, users want the app to count time and distance in records. Besides, average speed could help users to identify comfortable driving speed in different situations. So, users need that information in records. What's more, real-time MPG could let users know how fuel consumption depends on road conditions and average MPG could help users assess fuel consumption of a car. So, real-time MPG and average MPG were both needed in driving records. Moreover, fuel consumption and fuel cost gave user intuitional feelings of how much a car cost. So, those two kinds of information should be included in records. Furthermore, engine coolant temperature could help user identify if a car was at risk of damage from overheating and control module voltage could help user identify if a car was at normal voltage level. So, users want engine coolant temperature and control module voltage to be included in records. At last, driving records should contain bad driving habits like sharp acceleration times and sharp braking times. So, users want those two kinds of information to help them overcome bad driving habits.

When users are driving on the road, bad driving habits or unknown malfunctions may result in accidents. Users may want the app to remind them of those bad driving habits or unknown malfunctions. Users want to set up speed limit to build over speed protection. Meanwhile, users also want to limit driving hours in case of tired driving. In addition, users also want to set up water temperature limit to remind them of high engine temperature. Besides, users want to set up fuel price to help them calculate how much they would pay for a journey. What's more, different users used different units of measurement. So, they want to find their familiar units of measurement in the app. Because users want to record all kinds of driving information, they need to register accounts and then using them to log into the app. Meanwhile, they want the app to memorize their accounts at very first time so they did not need to type username and password again and again. Only when users used log out function would they be required to type username and password for next time's login.

Once users had multiple driving records, they might want to search records for a date and got total fuel consumption, total fuel cost and total driving distance on that day. So, the app should show all records on that day. Besides, total fuel consumption, total fuel cost and total driving distance on that day should be included. Another important requirement for users is to make full inspections of cars. Users want to check their cars to find if there exist problems in cars. External problems (e.g., headlight malfunction) are easily to find and internal problems are hard to be detected without using professional tools. Users need the app to help them identify those malfunctions.

2.2 Functional requirements analysis

Index: Access.1

Name: Register

Purpose: To register an account for Riding Mechanic.

Input parameters: Username, password, confirm password

Action: Ensure that username doesn't exist in the system.

 Ensure that password and confirm password should not be empty.

 Ensure that password and confirm password are same.

 Show information of registering an account successfully.

Output parameters: None

Exceptions: Username has already existed.

 Password or confirm password is empty.

 Password and confirm password are different.

Remarks: Application should show proper error message when users make mistakes.

Cross-references: None

Index: Access.2

Name: Login

Purpose: To login into Riding Mechanic.

Input parameters: Username, password

Action: Ensure that username exists in the system.

Ensure that username and password match with the corresponding entries stored in database.

If they match, jump to main screen.

Output parameters: None

Exceptions: Username does not exist.

Username and Password don't match.

Remarks: Checking the format of inputs
before validating them and input validation.

Cross-references: None

Index: Access.3

Name: Logout

Purpose: To logout from Riding Mechanic.

Input parameters: None

Action: Ensure that a user has already logged in.

Ensure that the user does not have access to the system.

Terminate all pending activities.

Output parameters: None

Exceptions: Logout action takes long time.

Remarks: None

Cross-references: Access.2

Index: Access.4

Name: Forget

Purpose: To reset password.

Input parameters: Username

Action: Ensure that a user has that username associated with an account

Ensure that that username is in email format

Output parameters: None

Exceptions: Username doesn't exist

Username is not in email format

Remarks: None

Cross-references: None

Index: Connection.1

Name: Connect to a car

Purpose: To connect to a car.

Input parameters: None

Action: Ensure that an OBD-II adapter has already plugged into a car.

Ensure that Riding Mechanic is using the OBD-II adapter's Wi-Fi.

Jump to main screen and change status to connected.

Output parameters: None

Exceptions: The OBD-II adapter hasn't plugged into a car.

Riding Mechanic is not using the OBD-II adapter's Wi-Fi.

Remarks: Let users perform actions step by step.

Cross-references: Access.2

Index: Get.1

Name: Get supported PIDs

Purpose: To get supported OBD-II PIDs of a car.

Input parameters: None

Action: Ensure that Riding Mechanic has already connected to a car

Store those information into database.

Output parameters: None

Exceptions: Riding Mechanic hasn't connected to a car.

Remarks: Perform this activity in background.

Cross-references: Connection.1

Index: Get.2

Name: Get driving information

Purpose: To get real-time data while driving

Input parameters: None

Action: Ensure that Riding Mechanic has already connected to a car

 Ensure that Riding Mechanic has already got supported PIDs.

 Keep sending PIDs to the car, process return data and show it to users.

Output parameters: Real-time information of the car

Exceptions: Riding Mechanic hasn't connected to a car.

 Riding Mechanic hasn't got supported PIDs.

Remarks: None.

Cross-references: Get.1

Index: Save.1

Name: Save driving information

Purpose: To save driving information

Input parameters: None

Action: Ensure that Riding Mechanic has already connected to a car

 Ensure that Riding Mechanic has already got supported PIDs.

 Ensure that Riding Mechanic is keeping reading data from the car.

 Choose to stop reading data and save it.

Output parameters: None

Exceptions: Riding Mechanic hasn't connected to a car.

 Riding Mechanic hasn't got supported PIDs.

 Riding Mechanic is not keeping reading data from the car.

Remarks: Give users choice to save or not save.

Cross-references: Get.2

Index: Search.1

Name: Choose specific date
Purpose: To choose a specific date for searching
Input parameters: None
Action: Ensure that users have already logged in.
Output parameters: None
Exceptions: Users haven't logged in.
Remarks: None
Cross-references: Access.2

Index: Search.2
Name: Search driving information
Purpose: To search driving information for a specific date.
Input parameters: None
Action: Ensure that users have already logged in.
 Ensure that users have already chosen a specific date.
Output parameters: Driving Information
Exceptions: Users haven't logged in.
 Users haven't chosen a specific date
Remarks: None
Cross-references: Search.1

Index: Check.1
Name: Check a car
Purpose: To check a car to find malfunctions
Input parameters: None
Action: Ensure that users have already logged in.
 Ensure that Riding Mechanic has already connected to a car.
 Start checking and show results.
Output parameters: Malfunctions

Exceptions: Users haven't logged in.

Riding Mechanic hasn't connected to the car.

Remarks: Give hint if no malfunctions occur.

Cross-references: Connection.1

Index: Alert.1

Name: Set up alert

Purpose: To close or open an alert for Riding Mechanic

Input parameters: None

Action: Ensure that users have already logged in.

Open or close an alert.

Output parameters: None

Exceptions: Users haven't logged in.

Remarks: None

Cross-references: Access.2

Index: Alert.2

Name: Set up a speed limit alert

Purpose: To set up a speed limit alert while driving

Input parameters: Speed

Action: Ensure that users have already logged in.

Enter valid limited speed.

Output parameters: None

Exceptions: Users haven't logged in.

Limited speed is invalid.

Remarks: None

Cross-references: Access.2

Index: Alert.3

Name: Set up tired driving hours alert

Purpose: To set up tired driving hour alert while driving

Input parameters: Hour

Action: Ensure that users have already logged in.

Enter valid tired driving hour.

Output parameters: None

Exceptions: Users haven't logged in.

Tired driving hour is invalid.

Remarks: None

Cross-references: Access.2

Index: Alert.4

Name: Set up water temperature alert

Purpose: To set up water temperature alert while driving

Input parameters: Water temperature

Action: Ensure that users have already logged in.

Enter valid water temperature.

Output parameters: None

Exceptions: Users haven't logged in.

Water temperature value is invalid.

Remarks: None

Cross-references: Access.2

Index: Set.1

Name: Set up fuel price

Purpose: To set up fuel price to calculate fuel cost

Input parameters: Fuel price

Action: Ensure that users have already logged in.

Enter valid fuel price.

Output parameters: None

Exceptions: Users haven't logged in.

Fuel price is invalid.

Remarks: None

Cross-references: Access.2

Index: Set.2

Name: Set up unit of measurement

Purpose: To set up unit of measurement

Input parameters: None

Action: Ensure that users have already logged in.

Select unit of measurement.

Output parameters: None

Exceptions: Users haven't logged in.

Remarks: None

Cross-references: Access.2

2.3 Non-functional requirements analysis

This section would discuss non-functional requirements for this application from following aspects.

The first one was platform constraints. Because this application was developed for iOS users, it could only be installed and run successfully on an iPhone. Meanwhile, this application was using some application programming interfaces(APIs) supported by Apple Inc. and those APIs were only included in software development kit (SDK) whose iOS version was equal to version 8.0 or higher than 8.0. If users want to use this application without having any troubles, they need to upgrade their iPhones' system to iOS 8.0 or higher. The second one was operating constraints. Users need to register an account for this app and then used the account to login. The third one was accuracy and precision. This application would show real-time information of a car. Some of real-time

data came from the car and the other data was decided by developers. So, the accuracy and precision of the data was in allowable error scope. The fourth one was reliability. In case of system failure, this application would give proper error message to users when they had unexpected operations. The next one was security. This application would read data from a car. Taking data protection into consideration, the application would allow users use their personal accounts to login. The last one was usability. This application would give users appropriate message when users went to a different screen. Users could easily know what functions this screen had and how to go to other screens. What's more, this system would allow users perform functions step by step instead of giving all functions at very beginning.

3. System architecture design

Software architecture design is the process of finding solutions to deal with all kinds of requirements and optimization of common quality attributes such as usability, maintenance, and reliability. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application [6]. This section would firstly identity goals and constraints of architecture. Then, it would focus on figuring key scenarios out. After that, it would overview whole application including

using what kinds of architecture styles and using what kinds of tools and technologies. Finally, it would identify key issues that may occur during implementation and talk about how to deal with those issues.

3.1 Goals and constraints of architecture

One of the most important goals was to build a prototype for this project which could help users to understand developers' suggestions for the final product. Meanwhile, users can easily describe and prove requirements that have not been considered and then communicate with developers to figure issues out. This system had used incremental model and by the help of incremental prototyping the time gap between user and software developer had reduced a lot. Users had opportunities to provide feedback while some components were still in development. In addition, that way could influence the outcome of further development. During development, Dr. Elliott Forbes, advisor of this project, played a role of a customer and met with me very week to give his advice for each development phase in this project.

Another important goal was to identify key technical risks. The first key technical risk was to identify how to connect to a car through a mobile application. This process may involve hardware issues and would take some time to make it clear. After building connection successfully, another key technical risk was to identify how to read data from a car. This may take long time to figure out both sending requests to a car and receiving data from a car. The next key technical risk was to guarantee accuracy of the data read from the car. This phase allowed developers to keep recording data read from the car and compared it with data displayed in the car. The last key technical risk was to take thread safety into consideration. While the application was running, both UI changing and communication between a car and the application must be executed in different threads and those threads should not have conflicts with each other.

As mentioned in proposal, this application would use an OBD-II Wi-Fi scanner to read data from a car. In the other word, users must use a Wi-Fi type scanner to get access to the car and could not connect to normal network. This was the biggest constraint in this project. Another constraint was that the application doesn't have enough storage to keep

data and it should be saved in server database. The last constraint was that users need to buy an OBD-II Wi-Fi adapter and then they could use this application. There was no way to connect to a car directly without using any hardware products.

3.2 Key scenarios

A scenario is a broader and more encompassing description of a user's interaction with the system, rather than a path through a use case [6]. Key scenarios in this system included registering for an account, logging in, resetting password, connecting to a car, reading data from a car, saving data into database and searching history data.

When a user want to register for an account, he or she should go to registration screen and enter username and password and confirm password. There was no restrict for a user to enter username first or enter password first. The restrictions were different formats for username and password. Both username and password should not be empty. In additional, username should be email address and should not be same with any usernames stored in database. After a user complete those steps, he or she would register for accounts successfully.

When a user want to login, he or she was required to enter username and password. Username should be email address and be same with one of usernames stored in database. Meanwhile, username should match with password. After account verification, a user could login and perform functions in the system.

When a user forgot password, and want to reset password, he or she was required to enter username. Username should be email address and be same with one of usernames stored in database. The system should automatically generate a password for that username and send it to that user. Meanwhile, server database should update that account.

When a user want to connect to a car, he or she should have logged in. The first step was to open device's Wi-Fi in connection screen. Then an OBD-II Wi-Fi scanner need to be plugged into an OBD-II port on the car. Finally, connecting to OBD-II scanner's Wi-Fi. A user must finish those actions step by step.

When a user want to read data from a car, he or she was required to connect to the car first. After building connections successfully, the application would keep reading data

from the vehicle and show it to the user. If the user want to stop reading data, they just need to click stop button in the screen and then they would go back main screen.

When a user want to save data into database, he or she should have stopped reading data from a car and they system would remind them of saving date. The system would save data into temporary database instead of server database because network was unavailable when the user had connected to a car through an OBD-II Wi-Fi scanner. Temporary database would upload data to server database after login.

When a user want to search history data, he or she was required to go to trip analysis screen. Temporary database would request data from server database according to that account after login. The user could select specific date to search run history. If the user did not select any date, system would automatically show current date's trip information.

3.3 Application overview

This section would create an overview of what the application would look like when it was completed. As mentioned in capstone proposal, this software was designed to be a mobile application and would have a server database. The biggest deployment constraint was that this application could not have access to the server database while it was running on an iPhone. Some running data could only be stored in temporary database. Fetching and saving data must be executed between after login.

This application would use a combination of architecture styles. It would involve a separate server system and multiple clients. When a client sent one or more requests, the client would receive those requests and send corresponding responses to the client. There were two main benefits to use the client/server architectural style. The first one was higher security. Because all data was stored on the server, it was easy for admin to get a better control of security than client machines. The second one was ease of maintenance. Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network [6]. Another important architecture style was object-oriented architectural style which would divide system responsibilities into individual objects. Each individual module would take responsibilities for separate functionalities which can be used by other modules by providing corresponding

interfaces. There are several advantages to use objective-oriented design. The first one is reusable. Some classes can be used by other classes through polymorphism and abstraction. The second one is testable. Encapsulation of classes can help users to do both unit testing and integration testing. The last one is highly cohesive. By locating only related methods and features in an object, and using different objects for different sets of features, you can achieve a high level of cohesion [6].

This software was a mobile application running on iPhone. The programming language used for client part was objective-c and the integrated development environment was Xcode. As for the server part, the main programming language was Node.js and the integrated development environment was Visual Studio Code. This product would use MySQL as server database and use MySQL Workbench to manage the database.

3.4 Key issues and solution

It is meaningful for developers to identify key issues based on quality attributes where problems often occur when designing an application.

The first quality attribute was system quality which would analyze the whole system from supportability and testability. This product could not run individually without using an OBD-II Wi-Fi scanner. This was hardware restriction and could not be solved in software development. Besides, this product was required to run on an iPhone. There might exist unknown problems when users launched it on an iPad. As for testability, it was not too hard for developers to do functional testing, unit testing, integration testing and system testing because this product would take object-oriented architecture design.

The second quality was run-time quality which would analyze the system from manageability, performance, reliability, scalability and security. Manageability describes how easy it is for system administrators to manage the application. This system would take client/server architecture design to make it easy to manage the server database. Performance is an indication of the responsiveness of a system to execute any action within a given time interval [6]. In general, system should give proper message when the system was downloading data or uploading data. Some actions which took tens of seconds or a few minutes would be executed in the background instead of keeping users

waiting for a long time. Reliability defines how stable the system is when it is running. The system should give corresponding error message instead of breaking down suddenly whenever users make mistakes. Scalability is the ability to create a more advanced version of the system. This product would use generic OBD-II PIDs which could only request some basic information from a car. More functionalities could be designed if developers had access to manufactures specific OBD-II PIDs. Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, providing good access for disabled users, and resulting in a good overall user experience [6]. This product would use incremental model which can help users communicate with developers directly. Users had more information about the product while it was under development.

Another important key issue was thread safety. This application would send multiple standard OBD-II PIDs to the vehicle but the car could not receive more than two commands at the same time. The main thread of the application would take charge of updating UI and background threads would be responsible for sending OBD-II PIDs. The orders of sending different OBD-II PIDs should be sent based on their priorities. Next chapter would talk how to implement it.

As mentioned in users' requirements, a user could use this application to record vehicle information for a journey. Some information, such as speed and RPM, could be easily retrieved by sending corresponding OBD-II PIDs. But other information like driving distance could not be read directly from the vehicle. So, this kind of information would be handled by developers. Next chapter would describe how the system dealt with that kind of information.

As for communication between server side and client side, the system used the Hypertext Transfer Protocol (HTTP) to send request. Figure 2 showed a high-level design for this application. Both information from server side and information from client side must be configurable to be meaningful. What's more, the server side should give client side proper response whenever a client submitted an HTTP request to the server. For example, if a client sent a request to do account verification, the client received that

request and dealt with that request. If no such account existed, the server would respond no such account existed. If network crashed suddenly, the server would respond internet occurred error. If a request is in wrong format, the server would respond method was not supported. If that account existed in database, the server would give response together with that account information to the client.

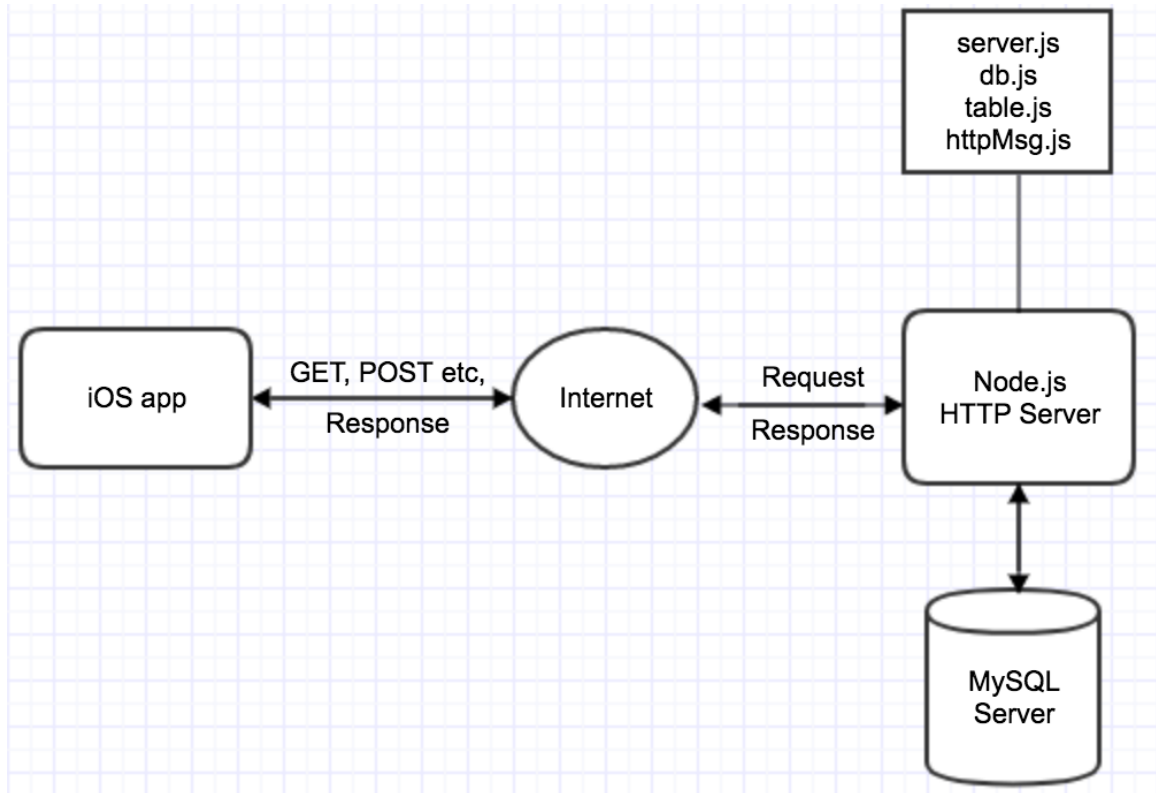


Figure 2. High-level architecture

4. Detailed system design

Detailed design focuses on all implementation details necessary to implement the architecture that is specified. Detailed system design is a low-level design and the main goal of it is to prepare a blue print of a system that meets the goals mentioned in system architecture design. This section described procedures and methods which would be used in detailed system design. According to the requirements of Rational Unified Process (RUP) [9] and Unified Modeling Language (UML) [10], detailed system design would define the behavior of each architectural component in detail. This section would take advantage of use case diagram to give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions were interacted. Class diagram would be used to shows internal classes in the system, attributes and operations of each class and the relationship between each class. Sequence diagram would be used to show interactions between objects, functions taken by each object and sequence of messages sent or receive by each object.

4.1 Database design

4.1.2 Database table design

A well-structured database can save disk space by eliminating redundant data, maintain data accuracy and integrity and provide access to the data in useful ways [11]. The first step in database design is to find out “entities” which exist in four kinds: people, things, events, and locations. Everything people could want to put in a database fits into one of these categories. In this project, user is absolutely one of the entities. Another entity is trip which contains information of a car during this trip.

The next step was to determine the relationships between the entities and to determine the cardinality of each relationship. In this project, a user could record a trip or multiple trips. Multiple users could not record the same trip.

The third step was to find out “attributes” which would be saved as data elements for each entity. For user table, it used auto-increment id as primary key. Unique “username” and “password” were also needed. Besides, “totalAlertSwitch” should be one of the “attributes” because it could determine if a user wants to open or close alert during a trip. In addition, “speedLimit” and “speedAlertSwitch” were also “attributes” because “speedLimit” determines the warning speed value and “speedAlertSwitch” determines if a user want to open speed limit alert or not. Similarly, we could get “attributes” like “tiredDrivingHour”, “tiredDrivingAlertSwitch”, “waterTemperatureLimit” and “waterTemperatureAlertSwitch”. What’s more, “fuelPrice” should be one of the “attributes” because users could use it to calculate fuel cost. “unit” should also be one of the “attributes” because users should be able to change unit of measurement. About a trip, it should use auto increment id as primary key and uses “userID” as a foreign key. In addition, it should have “startDateTime” and “endDateTime” to represent when users recorded the trip. Besides, “drivingDistance”, “averageMPG”, “averageSpeed”, “fuelCost”, “sharpAccelerationTime” and “sharpBrakingTime” were also “attributes” because they were necessary information for users to analyze a trip.

After finishing above steps, Entity Relationship Diagram (ERD) could be easily retrieved. ERD gave a graphical overview of the database. Figure 3 showed ERD for this project.

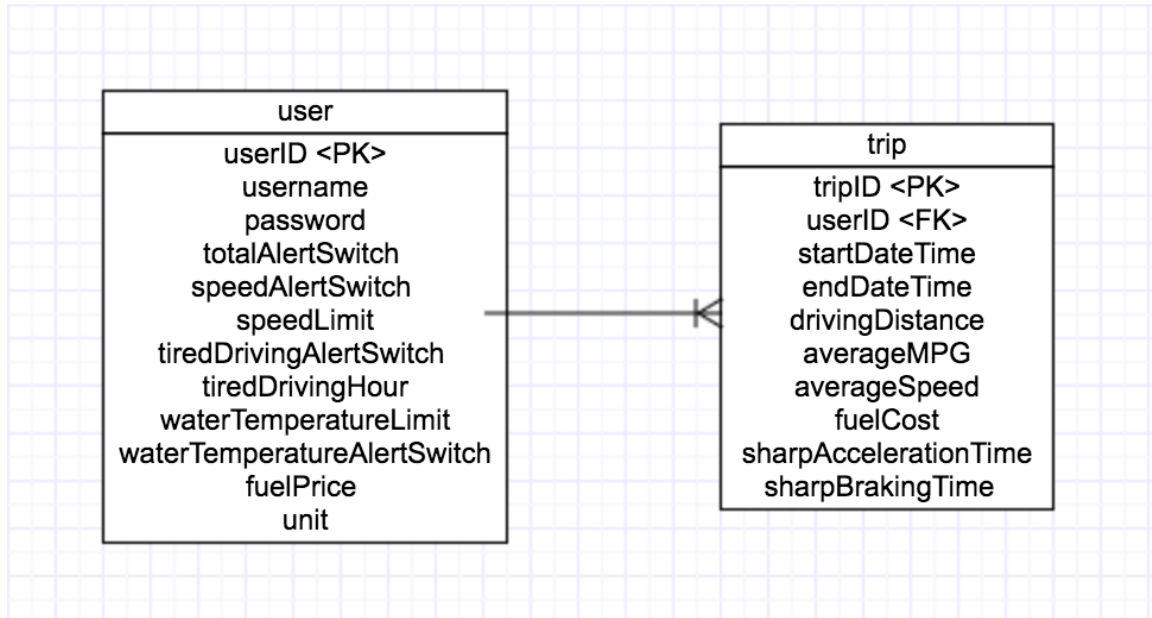


Figure 3. Entity relationship diagram

The final step was to define attribute's data type. There were a lot of different data types. This project would use MySQL as a server database, so all types of data were defined by MySQL. For user table, all attributes should not be null. "userID" should be unique positive self-increasing integer. Type of "username" and "password" were "VARCHAR" and "username" should be unique. "totalAlertSwitch", "speedAlertSwitch", "tiredDrivingAlertSwitch", "waterTemperatureAlertSwitch" and "unit" had the same type. The type was "TINYINT" which was equal to Boolean type. The reason to use "TINYINT" instead of using "Boolean" was that MySQL did not have internal "Boolean" data type. "speedLimit", "tiredDrivingHour", "waterTemperatureLimit" and "fuelPrice" used the type of "Double" because all of them were float values. For table trip, "tripID" should be unique positive self-increasing integer. It used "userID" as foreign key and "userID" should be not null and unique. Besides, "startDateTime" and "endDateTime" represented specific date and time so their type was "DATETIME". "drivingDistance", "averageMPG", "averageSpeed" and "fuelCost" were float values and their type was "DOUBLE". "sharpAccelerationTime" and "sharpBrakingTime" recorded how many times users had sharp acceleration and sharp braking, so their type was "INT".

After those steps finished, we could build the database through MySQL Workbench. Figure 4 showed user table and figure 5 showed trip table.

Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G	Default / Expression
userID	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
username	VARCHAR(100)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password	VARCHAR(100)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
totalAlertSwitch	TINYINT(4)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
speedAlertSwitch	TINYINT(4)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'80'
speedLimit	DOUBLE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
tiredDrivingAlertSwitch	TINYINT(4)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
tiredDrivingHour	DOUBLE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'4'
waterTemperatureAlertSwitch	TINYINT(4)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'
waterTemperatureLimit	DOUBLE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'220'
fuelPrice	DOUBLE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2.39'
unit	TINYINT(4)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'1'

Figure 4. User table in database

Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G	Default / Expression
tripID	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
userID	INT(11)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
startDateTime	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
endDateTime	DATETIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
drivingDistance	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
averageMPG	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
averageSpeed	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
fuelCost	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
sharpAccelerationTime	INT(11)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
sharpBrakingTime	INT(11)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Figure 5. Trip table in database

4.1.2 Database password encryption and decryption

Password encryption was designed to grantee account safety. When a user had created an account for this application, the password would be encrypted when it was insert into server database table. This technology was supported by MySQL. The encryption method was “AES_Encrypt (password, key)”. By using this method, the password in the table would be a messy code and no one could read useful information form that messy code.

The key of password worked as an index. The application could only find the password through the key. This project would use unique username as the key of the password.

It was easy for developers get the password which was not encrypted. Once it had been encrypted, the most important thing was decryption. The decryption method was “AES_Decrypt (password, key)”. This decryption key should be same with the key for encryption. As mentioned above, it was the username. It was not enough to get the real password because the password would be converted into binary value after using that method. So, another method, “cast (AES_Decrypt (password, key), AS CHAR)”, should be used after initial decryption. So, the first step of decryption was to convert a messy code into binary value by using “AES_Decrypt (password, key)”. The second step was to convert that binary value into char value by using “cast (AES_Decrypt (password, key), AS CHAR)”. Then, the real password could be retrieved.

4.2 Use case diagram

All use cases were retrieved from user requirements and the use case diagrams gave an overview of usage requirements for a system. Figure 6 showed the use case diagram of this project.



Figure 6. Use case diagram

4.3 Class diagram

Class diagram is the core part of UML diagram and it describes the structure of a system by modeling its classes, attributes, operations, and relationships between objects. There are several benefits by using class diagram. It can help developers to illustrate data models for information systems and have a good understanding of a general overview of system. In addition, it provides an implementation which contains independent description of types used in a system and passed between its components. Figure 7 showed the class diagram of this project.

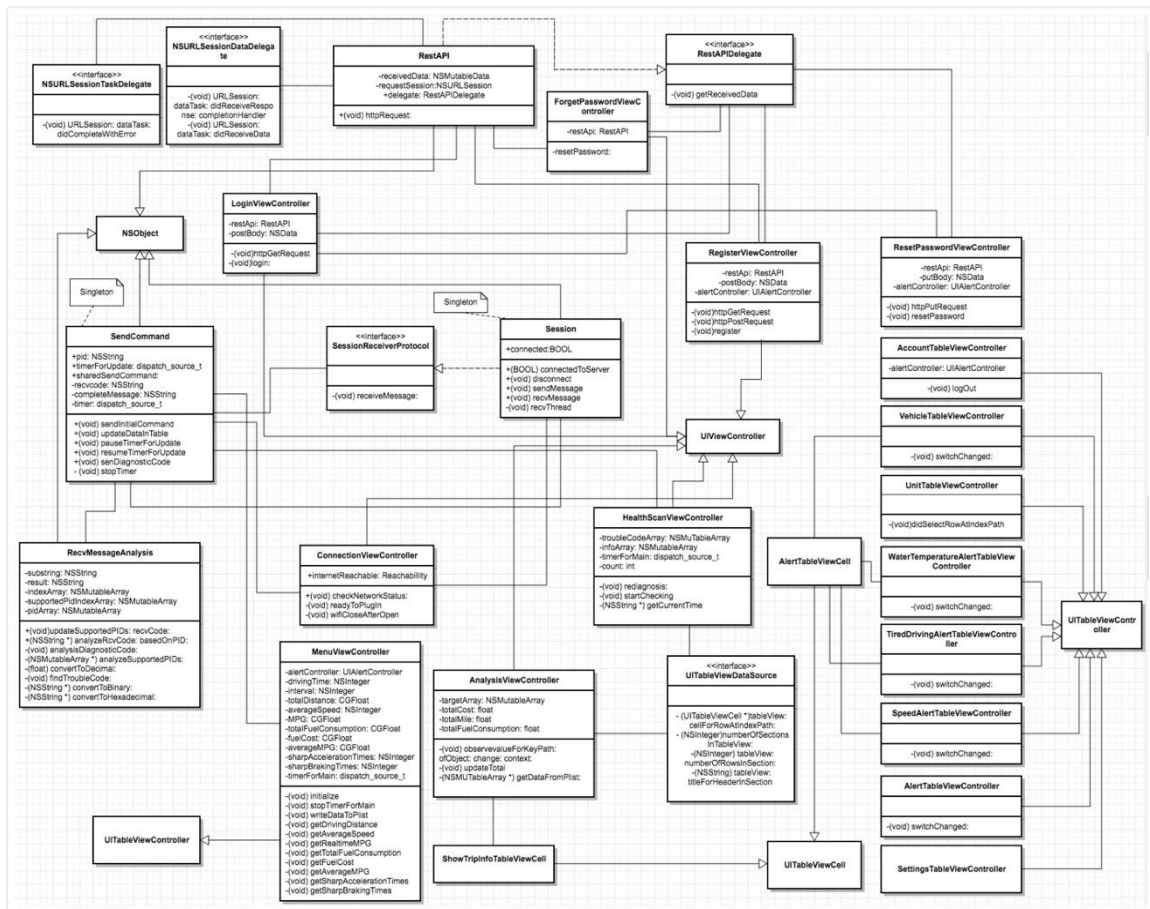


Figure 7. Class diagram

4.4 Sequence diagram

This section would mainly show the sequence diagrams of some important activities in this application. By using sequence diagram, we could clearly see how different components interact with each other and how messages passed between components in different activities.

Figure 8 showed login scenario; Figure 9 showed registration scenario; Figure 10 showed forgetting password scenario; Figure 11 showed connection scenario; Figure 12 showed reading data scenario; Figure 13 showed searching data scenario; Figure 14 showed checking a car scenario. Figure 15 showed orders of sending initial OBD-II PIDs to get supported OBD-II PIDs.

While the system was sending OBD-II PIDs to the car to get real-time information, it was very important to decide the order of sending different OBD-II PIDs because the OBD-II system could only receive and give response to one OBD-II PID each time. The application would receive message and deal with that message every half second. In the other word, the application could only receive two kinds of message each second which meant the application could only send two kinds of message each second. Table 6 showed the frequency of sending each kind of OBD-II PIDs.

	1s	2s	3s	4s	5s	6s	7s	8s	9s	10s	Rule
0105 (Coolant Temperature)										√	Time% 10==0
010C (RPM)	√		√		√		√		√		Time is odd
010D (Speed)	√	√	√	√	√	√	√	√	√	√	Time% 1==0
0110 (MAF Air flow rate)		√		√				√			Time is even and! Time% 10==0 and! Time% 6==0
0142 (Control module voltage)						√					Time% 6==0

Table 6. Frequency of sending different OBD-II PIDs

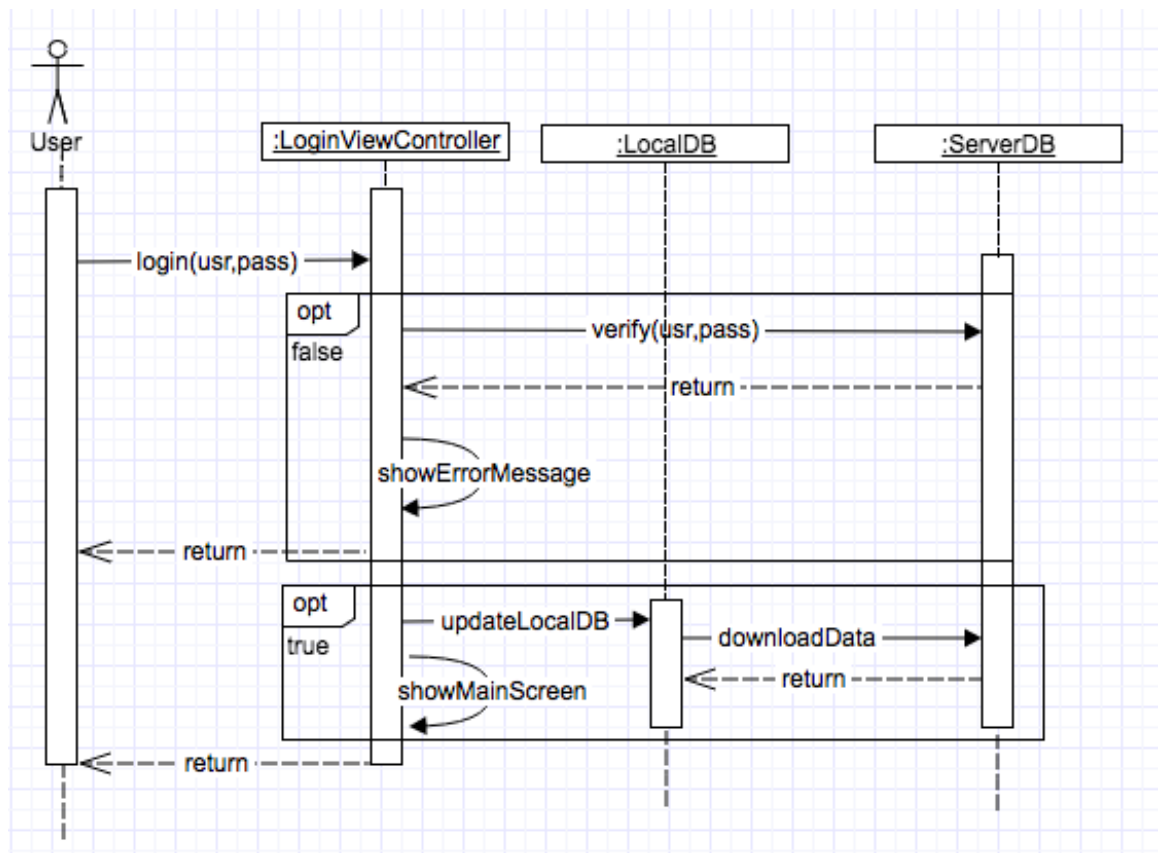


Figure 8. Sequence diagram of login

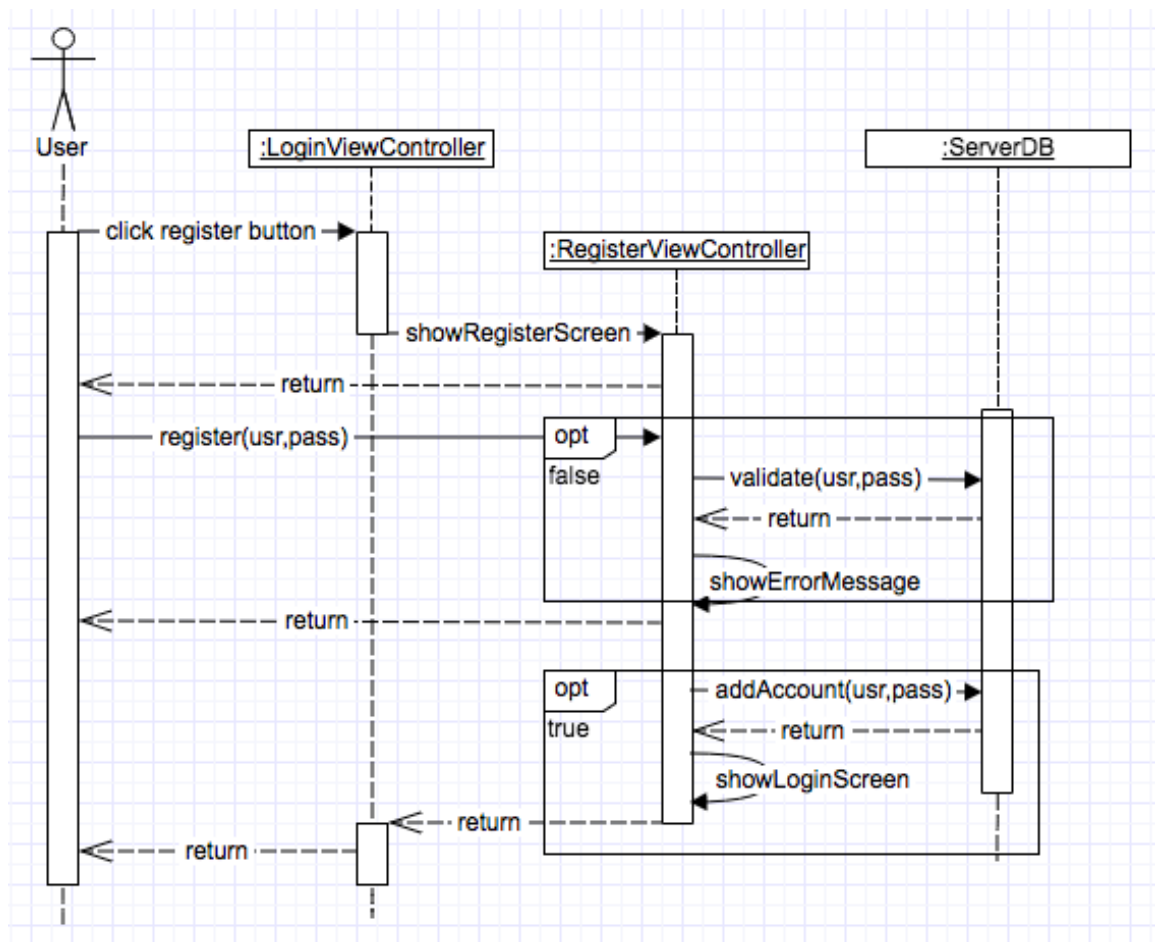


Figure 9. Sequence diagram of registration

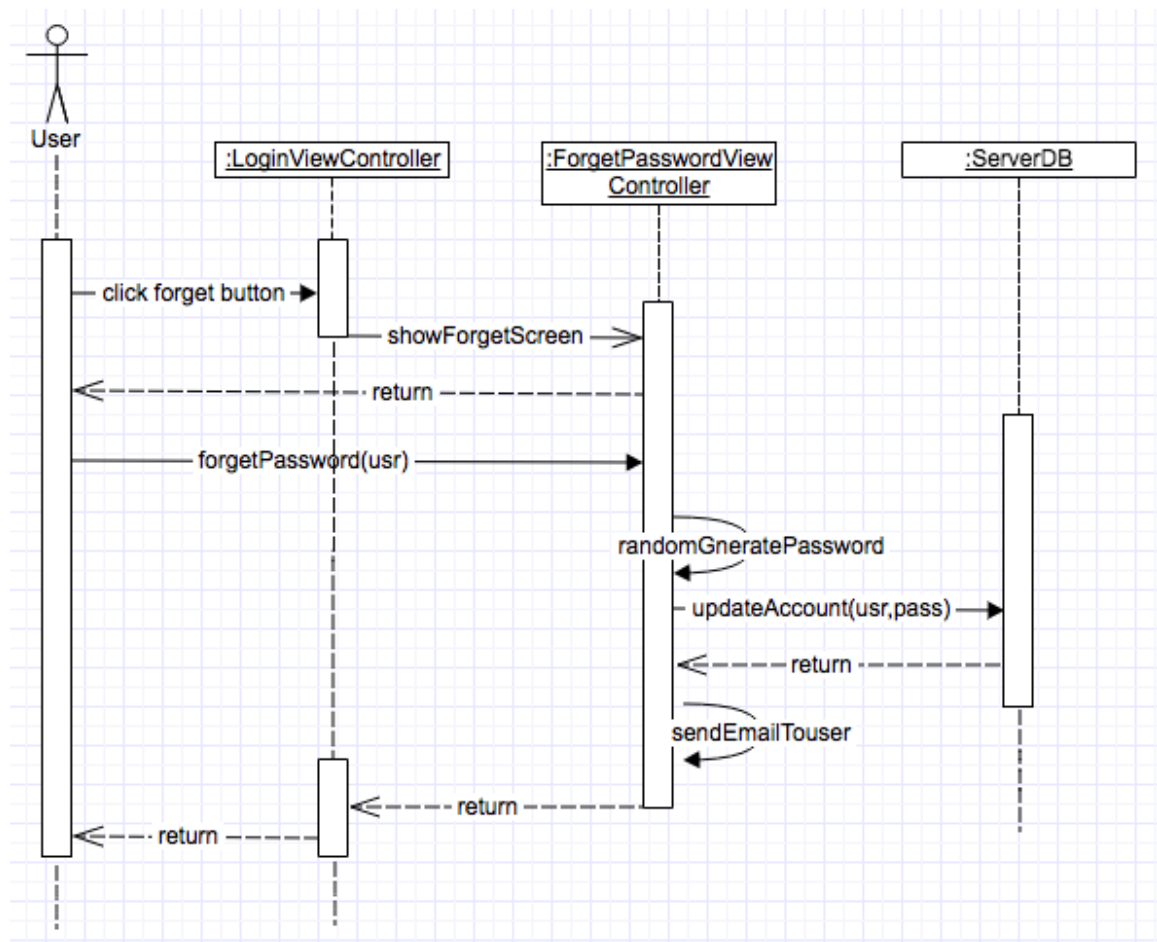


Figure 10. Sequence diagram of resetting password

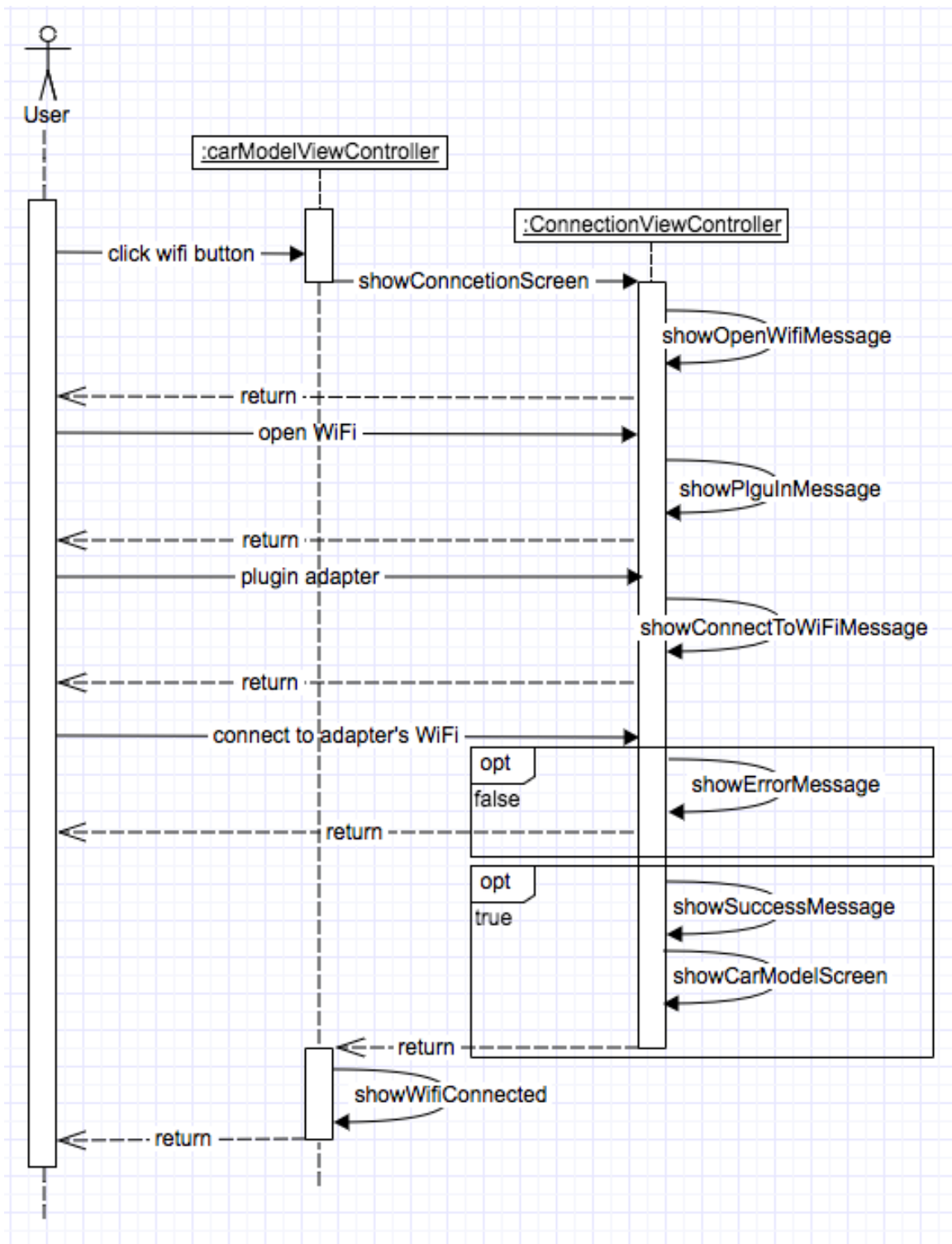


Figure 11. Sequence diagram of connection

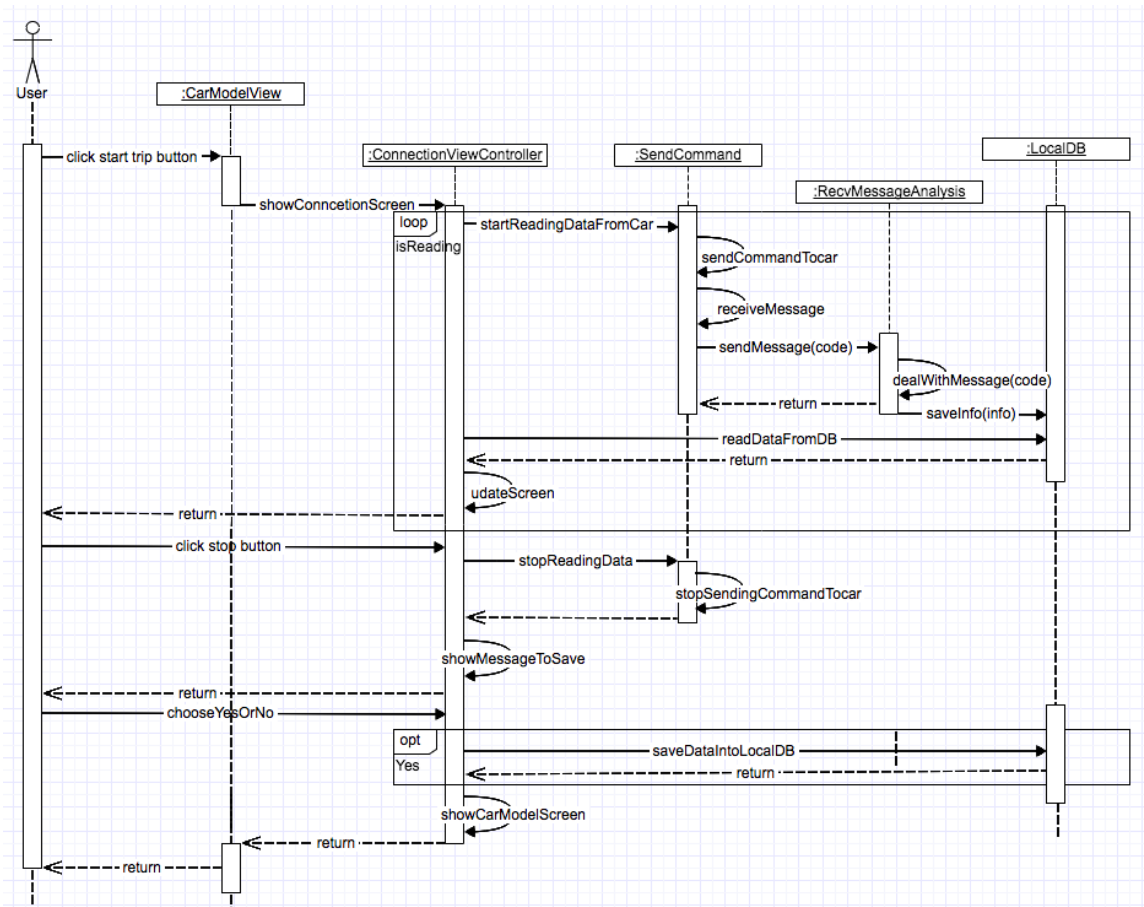


Figure 12. Sequence diagram of reading data

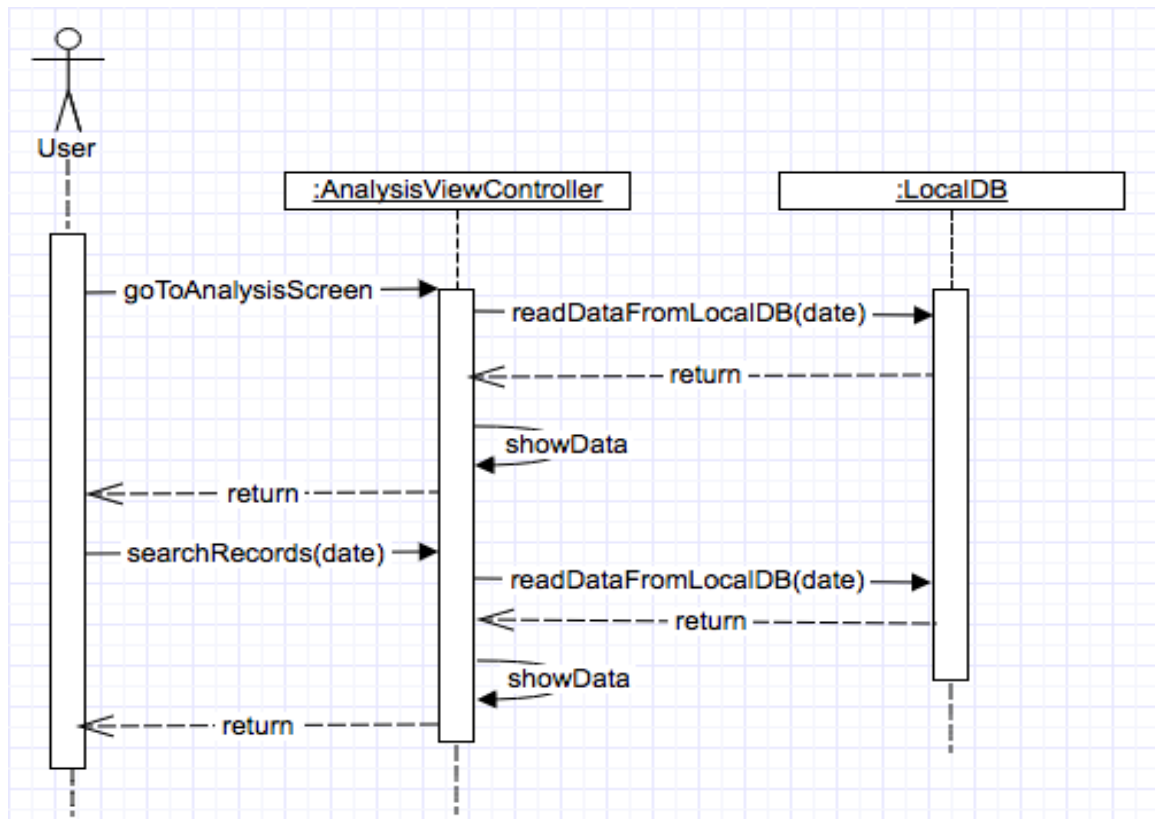


Figure 13. Sequence diagram of searching a trip

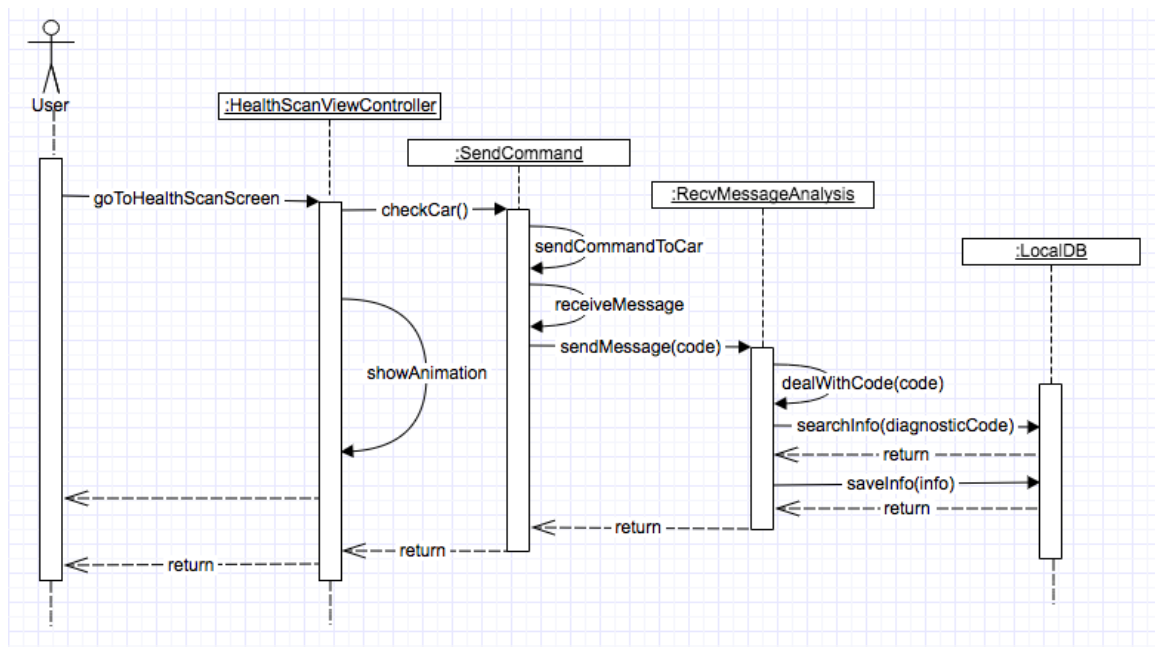


Figure 14. Sequence diagram of checking a car

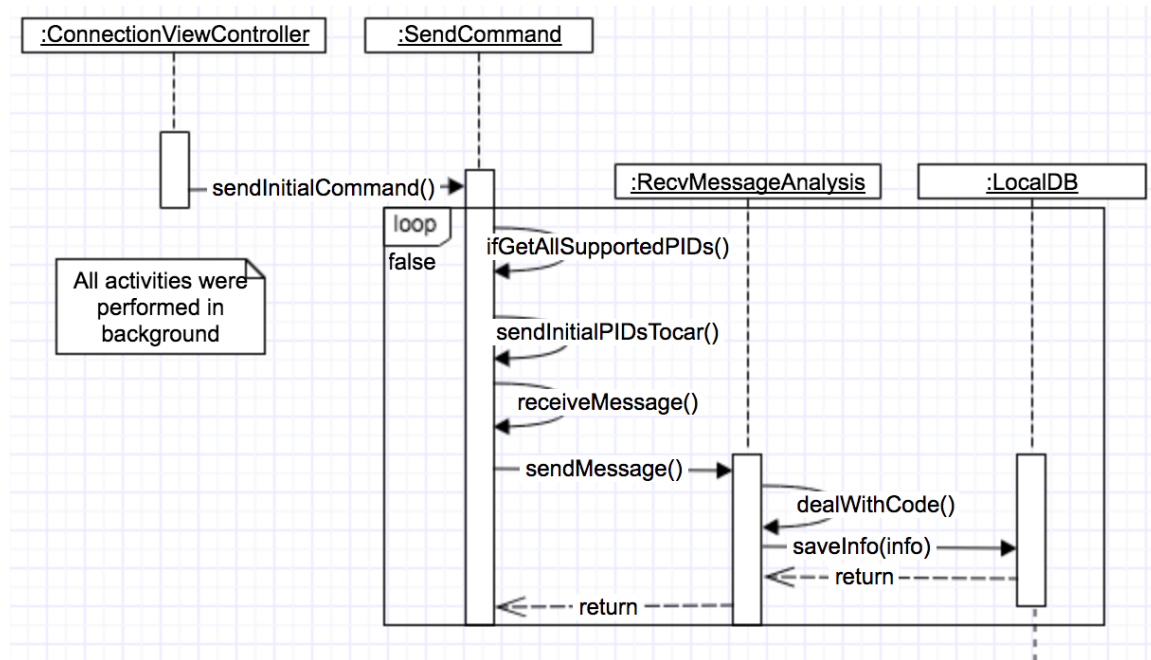


Figure 15. Sequence diagram of sending initial commands

4.5 How driving distance was calculated

As mentioned above, some information of a journey could not be read directly from the vehicle and that kind of data would be worked out by the system. This part would describe the process of calculating the driving distance. Physics formulas had taught us distance was equal to elapsed time plus average velocity. We could apply it into calculating driving distance. The elapsed time could be worked out by just counting elapsed time. The system could also read the current speed from the vehicle every second. By using the theory of differential calculus, the error of the distance would be reduced by using smaller time interval and the smallest time interval for reading current speed was one second. So, the first step for calculating distance was to get average speed in one second. For example, the speed of the third second was ten miles per hour and the speed of the fourth second was twenty miles per hour, so the average speed between the third second and the fourth second was fifteen miles per hour. Then, the second step was

to calculate the distance in one second by using the physics formula ($s=v*t$). The last step was to add every second's distance together to get total distance.

4.6 User interface design

As we know, user interface design is one of the most important parts in software development. The quality of user interface has huge influence on user experience of the product. A good GUI (graphical user interface) not only catches a first-time user's eye but also motivates the user to use the product again. I would introduce four principles of UI design which come from a paper [10] that I had read. The first one is reading gravity. The second one is minimization of cognitive load. The third one is using pre-existing mental models. The last one is use of color to direct attention. In addition, I also have my own GUI designing experience which come from Professor Kasi's class. The first one is that GUI should be usable to as many as possible. The second one is that GUI must report appropriate error messages at the right time. The third one is that providing appropriate help at the right time and motivate the user to continue. The last one is that try to protect the users from introducing errors into the system (error prevention). Those UI design principles will be used as references to design UI of this product.

I would first introduce some basic patterns for mobile navigation. Tab bar pattern was one of the most frequently used patterns in mobile applications. Many popular applications, like Facebook, Instagram and Twitter, using tab bar pattern in their design. It was originally from desktop design. The tab bar usually put navigation icons at the bottom of a screen to fit finger touch. Users could navigate directly to the screen associated with the item. By using tab bar navigation pattern, users could easily have access to all main functionalities without multiple touches; In addition, users could know where they were and what to do when they are using the application. In this product, all main functions were represented by bottom icons in tab bar. Figure 16 showed an example.

Another popular UI design pattern was navigation bar. Almost all popular mobile applications had navigation bars. In general, a navigation appeared between the status bar

and an app screen. Users could view the status and application screens without having any troubles. A navigation bar usually had title, back button and control button. Back button was always associated with previous screen and appears on the left side of the bar. Title always showed information of current screen and appears on the middle of the bar. As for control button, it was related to a Cancel or Done button and mainly used to manage users' actions. This product had used lot of navigation bars. Figure 17 showed an example.

In addition to UI design patterns mentioned above, some popular UI elements were also frequently used in famous mobile applications. For example, Messenger used table view to show contract list. The table view was used to manage data which had similar data structure or belongs to similar category. Table view could help us to show additional contents by scrolling with it. Some table views also had headers and footers which could make it easy to tract a long table. Figure 18 showed an example in this product.

Alert is another important UI element in mobile application design. Many applications used alerts to remind users of something important. An alert usually contains a title and optional information. The title indicated what that the alert's meaning and optional information told users what they could do. A widely-used example for using alert was AMBER Alert. Almost all people could view that information immediately once an AMBER Alert occurred. From this example, we could easily find out the advantages of using alerts in mobile application. Figure 19 showed an example in this product.

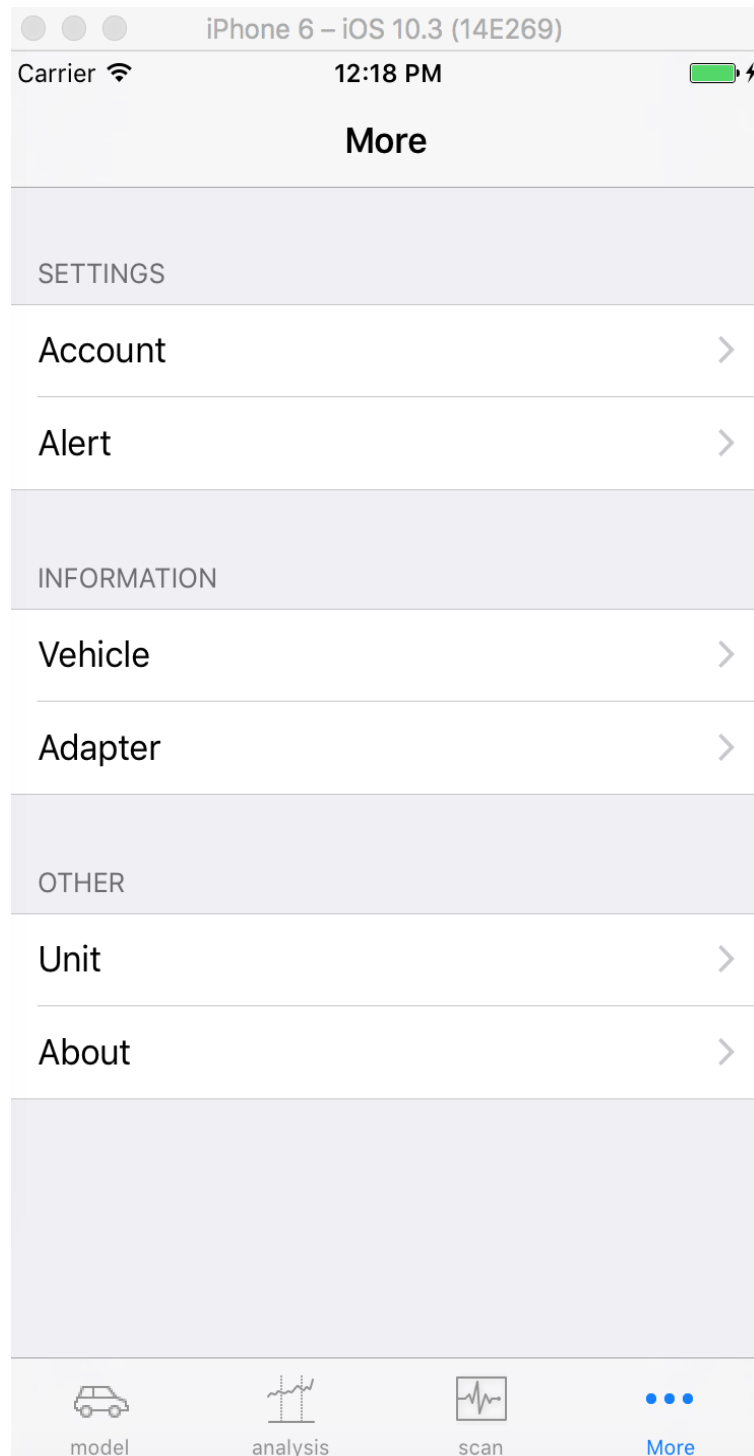


Figure 16. Example of using a tab bar

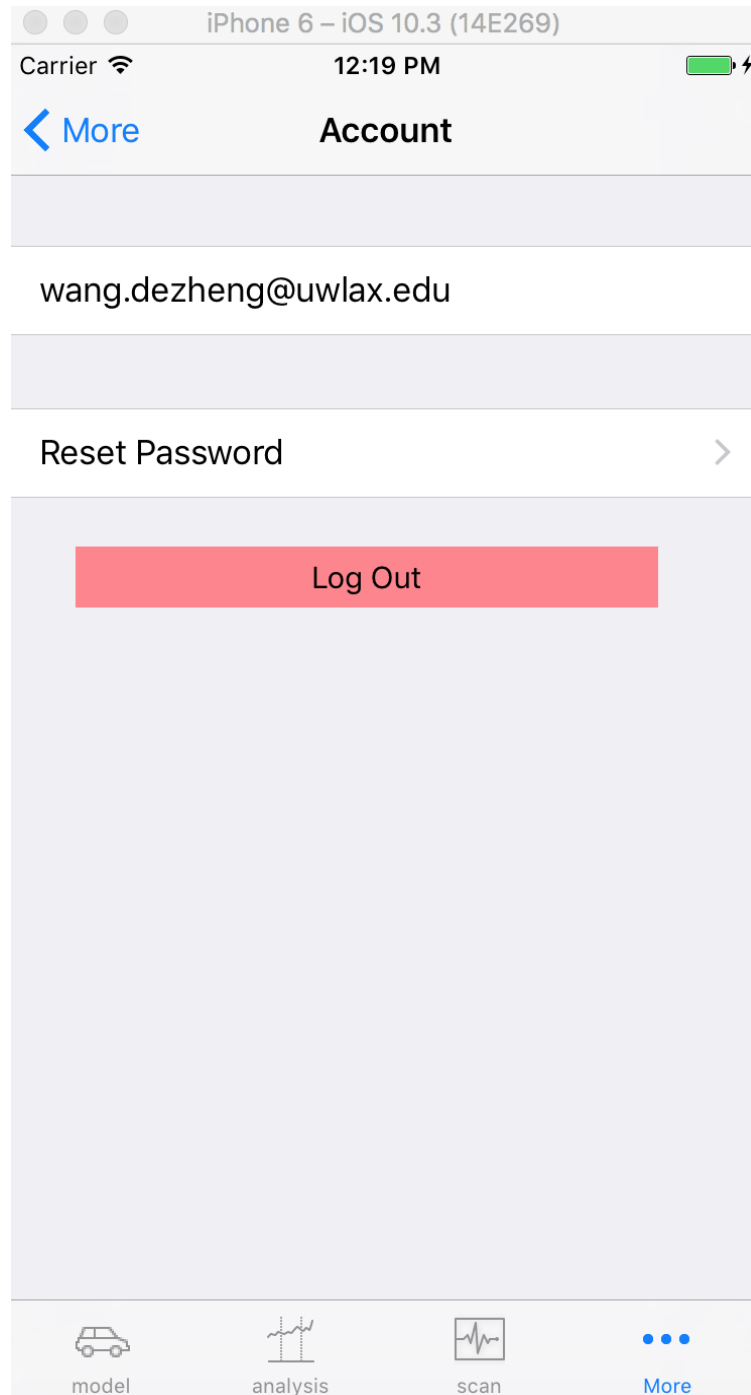


Figure 17. Example of using navigation bar



iPhone 6 – iOS 10.3 (14E269)		
Carrier 	12:37 PM	
Trip Information		Stop
Driving Time	6	s
Driving Distance	0.00	km
Speed	0.0	km/h
Average Speed	0.0	km/h
RPM	0	rmp
Realtime MPG	0.00	mpg
Total Fuel Consumption	0.00	gal
AverageMPG	0.00	mpg
Fuel Cost	0.00	gal
Engine Coolant Temperature	0	°C
Control Module Voltage	0	V
Sharp Acceleration	0	time
Sharp Braking	0	time

Figure 18. Example of using table view

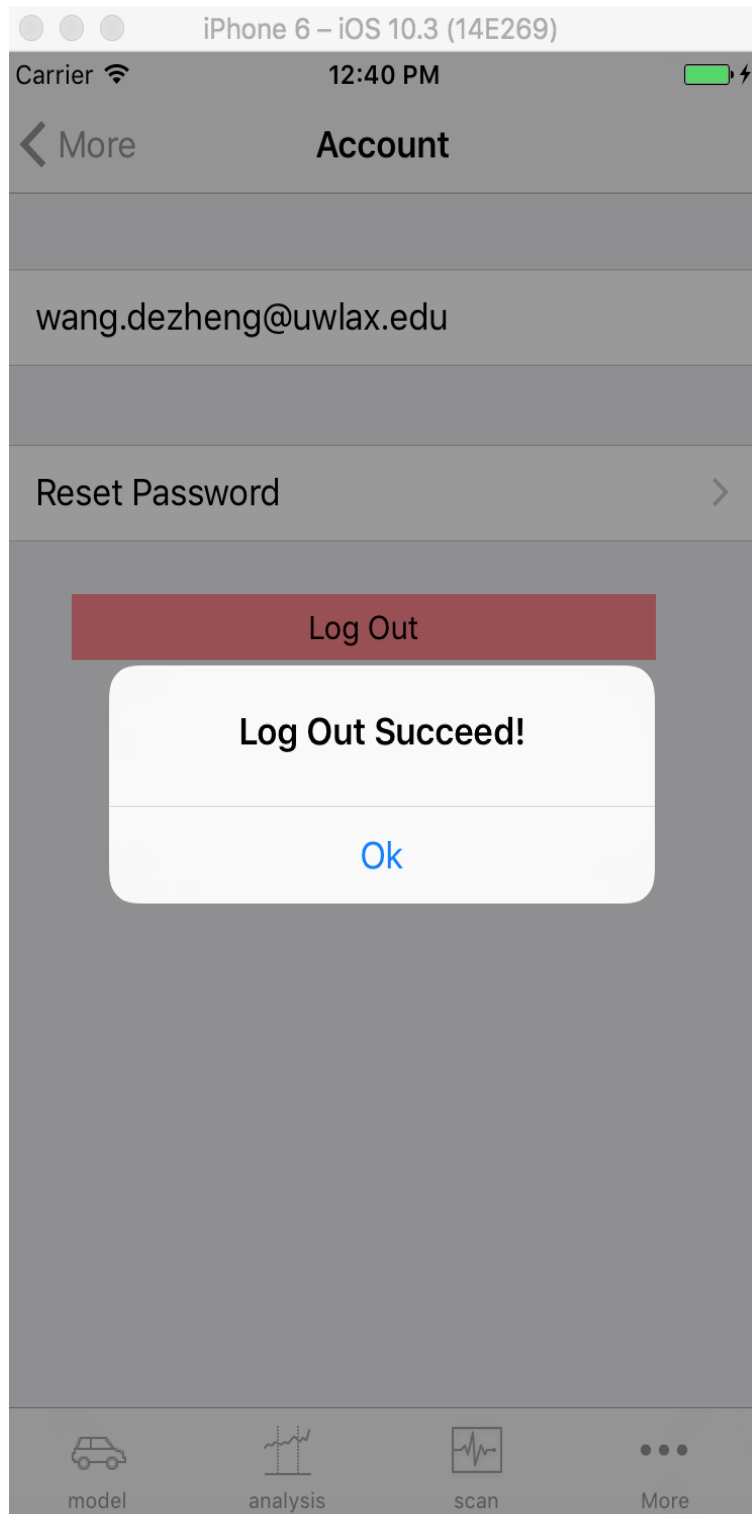


Figure 19. Example of using an alert

5. System implementation

Instead of using objective-c codes, this section will use pseudocode to talk about the main algorithm and logic in this application. Pseudocode can be understood by most people.

5.1 Login module

This part used pseudocode to describe the login procedure. It involved following functions: Login (), SetInitialStatus (username), SetUser(username), SetTrip (userID).

If username and password had been memorized

 Go to main screen

Else if users click login button

 Login () //User click login button will call this function

Else if users click register button

 Go to register screen

Else

 Go to forget password screen

Login () {

 Set username to username text field value

 Set password to password text field value

 If username or password is empty

 Alert "Username or password should not be empty"

 Else

 Send get request to the server //request body contains username

 Query password based on username in user table

 If return value is empty

 Alert "Incorrect username or password"

 Else if return value is not equal to password

 Alert "Incorrect username or password"

 Else if return value is equal to password

```

        Store username, password, userID in local database
        SetInitialStatus (username) // background thread
        Go to main screen
    }
}

SetInitialStatus (username) {
    SetUser(username) // upload local data (update local database if needed)
    SetTrip (userID) // upload local data (update local database if needed)
}

SetUser (username) {
    If user table is empty in local database
        Send get request to the server //request body contains username
        Query all related information based on username in user table in server
        Set a delay // need time to download data
        Set userInfo to return value
        Update user table in local database based on userInfo
    Else
        Query all related information based on username in local database
        Set userInfo to return value
        Send put request to the server // request body contains userInfo
        Update user table in server database based on userInfo
    }
}

SetTrip (userID) {
    If new trip table is empty in local database
        Send get request to the server //request body contains userID
        Query all related information based on userID in trip table in server
        Set a delay // need time to download data
        Set tripInfo to return value
        Update trip table in local database based on tripInfo
    Else

```

```

        Query all related information based on userID in new trip table in local
        Set newTripInfo to return value
        Send post request to the server // request body contains newTripInfo
        Add newTripInfo to trip table in server database
    }

```

5.2 Registration module

This part used pseudocode to describe the registration procedure. It involved following functions: Verify ().

If users click submit button in register screen

Verify () //User click submit button will call this function

Else users choose to go back login screen

Go to login screen

Verify () {

Set username to username text field value

Set password to password text field value

Set confirm to confirm password field value

If username is not in email format

Alert "Username should in email format"

Else if password or confirm is empty

Alert "Password or confirm password should not be empty"

Else if password is not equal to confirm

Alert "Passwords must match"

Else

Send post request to the server // request contains username and password

Insert username and password into user table

If return value is false

Alert "Sorry, this username had been used!"

Else

```
        Alert "Register Succeed"  
        Go to login screen  
    }
```

5.3 Forget password module

This part used pseudocode to describe how to reset password when a user forgot his password.

```
Set username to username text field value  
If username is empty  
    Alert "You must enter username"  
Else  
    Send get request to the server //request body contains username  
    Query that username in user table in server database  
    If return value does not have content  
        Alert "Username does not exist"  
    Else  
        Set new Password to random six-digit number  
        Send email to username // username is in email format  
        Send put request to the server  
        Update user table in server database based on username and newPassword  
        Go to login screen
```

5.4 Device connecting module

This part used pseudocode to describe the procedure of connecting to an OBD-II Wi-Fi scanner. It involved flowing functions: SendInitialCommand (), ReceiveMessage ().

```
while Wi-Fi is open  
Do  
    Show plug in button  
    Users click plug in button
```

```

Show connecting to Wi-Fi button
If device is connected to adapter's Wi-Fi
    Set wifiStatus to connected
    Store wifiStatus in local database
    Use SendCommand class
    SendInitialCommand () // this method comes from SendCommand class
    Go back main screen
Else
    Alert "Connection failure"

```

```

SendInitialCommand () {

```

```

    Create timer
    Set timer priority to background

```

```

    Timer:

```

```

        If last command had not been sent

```

```

            Use Session class
            Session send commands
            ReveiveMessage ()

```

```

        Else

```

```

            Stop timer

```

```

    }

```

```

ReveiveMessage () {

```

```

    Set message to receive message
    Use regular expression to deal with message
    Analyze message and get value// value represents vehicle supported command
    Store that value in local database

```

```

}

```

5.5 Trip information module

This part used pseudocode to describe how to read real-time information from a vehicle. It involved following functions: UpdateDataInTable (), ReveiveMessage ().

while Wi-Fi status is connected && users don not click stop button

Do

UpdateDataInTable () //this method comes from SendCommand class

Create timer

Set timer to priority to main

Timer:

Read data from local database

Analyze and compute data to get other data

Show data in table view

Reload table

If users click stop button

Alert (“Save trip information? Yes: No”)

If users select yes

Save those data and date in new trip table in local database

Go back main screen

UpdateDataInTable () {

Create timer

Set timer priority to background

Use Session class

Timer:

Get supported commands from local database

Session send supported commands

ReveiveMessage ()

}

ReveiveMessage () {

Set message to receive message

Use regular expression to deal with message

```
Analyze message and get corresponding value // value represents vehicle data
Store that value in local database
}
```

5.6 Health scan module

This part used pseudocode to describe how to read fault codes from a vehicle and show corresponding diagnostic information to users. It involved following functions:
SenDiagnosticCode (), ReveiveMessage ().

// this method will be executed automatically when users go to health scan screen

```
StartChecking () {
    Use SendCommand class
    SenDiagnosticCode ()

    Set timer to priority to main
    Timer:
        If timer ends
            Read diagnostic information and code from local database
            Show diagnostic information and code in table view
            Show rediagnostic button
            If users click rediagnostic button
                Set rediagnostic button and table view to hidden
                StartChecking ()
        Else
            Update UI to let users wait for scanning
```

```
SenDiagnosticCode () {
    Create timer
    Set backgroundTimer to priority to background
    backgroundTimer:
        Use Session class
        Session send diagnostic commands
```

```
ReveiveMessage ()  
}  


---

ReveiveMessage () {  
    Set message to receive message  
    Use regular expression to deal with message  
    Analyze message and get corresponding value // value represents diagnostic code  
    Search diagnostic information in fault code table based on diagnostic code  
    Store diagnostic information and diagnostic code in local database  


---


```


6. Testing

Software development always involves software testing which is the process of evaluating correctness, integrity and quality of the software. In the other word, software testing is the process of fixing bugs to satisfy user requirements. Software is usually planned earlier and it is performed only after coding. This section would talk about each phrase in software testing in detail.

6.1 Testing plan

In unit testing, testing strategy is to test server side and client side separately. For server side, it will use Postman to send different kinds of requests and check response. After bugs fixed in server side, it will use UI testing in Xcode to test each individual user interface. Compared to other UI testing framework, the major benefit of UI testing in Xcode is that it can record developers' operations to the system. In the other words, developers can perform all kinds of operations in each user interface to test all kinds of situations and then UI testing will memorize those operations and transfer them into testing codes which can be executed automatically. In a word, UI testing in Xcode is a kind of scene simulation and can bring lots of convenience to testers. Integration testing will also use UI testing in Xcode to testing the whole system including server side and client side. As for system testing, it will mainly test data accuracy. A tester will use this application to record plenty of data. Another tester will use recording functions in the car to get plenty of data. Then a tester will use graphs to make comparisons and then get conclusions.

6.2 Unit testing

6.2.1 Testing for server module

This part was responsible for testing different requests which were sent from the client side to the server side. Test cases were generated based on requests which had been used in the application. Other types of requests would not be taken into consideration. Table 7 showed details of each test case. Scenario indicated the testing environment. URL

represented all kinds of requests occurred in the application. Return value was message from the server side. Status indicated whether the test case passed or not.

Test case #	Scenario	URL	Return value	Status
1	User table has this account	138.49.101.87:9000/userSettings/wang.dezheng@uwlax.edu Method: GET	All user information related to username 'wang.dezheng@uwlax.edu'	Passed
2	User table doesn't have this account	138.49.101.87:9000/userSettings/wang.dezheng@uwlax.edu Method: GET	[]	Passed
3	Trip table has this trip	138.49.101.87:9000/tripInfo/1 Method: GET	All trip information related to userID '1'	Passed
4	Trip table doesn't have this trip	138.49.101.87:9000/tripInfo/1 Method: GET	[]	Passed
5	Add a new user to user table	138.49.101.87:9000/userInfo requestBody: {username: yuan.zhihao@uwlax.edu , password :123} Method: POST	'true'	Passed
6	Add an existed user to user table	138.49.101.87:9000/userInfo requestBody: {username: yuan.zhihao@uwlax.edu , password :123} Method: POST	'false'	Passed

7	Add a new trip to trip table	138.49.101.87:9000/tripInfo requestBody: {trip information} Method: POST	'true'	Passed
8	Update a user's information	138.49.101.87:9000/userInfo requestBody: {updated user information} Method: PUT	'true'	Passed

Table 7. Test cases for the server side

6.2.2 Testing for client module

Testing for client module would be divided into five parts. The first part was to test the session module which was responsible for sending commands and receiving messages from the car. The second part was to test login module. The third part was to test registration module. The fourth part was to test resetting password function. The last part was to test searching history trips module.

Table 8 showed all test cases for testing session module. Test cases were generated based on those OBD-PIDs which had been used in this application. Each test case had three kinds of return messages and all three were right. The reason why one OBD-II PID had three return values was that the frequency of the system receiving messages would have influence on message sent by the OBD-II system. Some message could not be sent during that time interval and it would be sent during next time interval.

Test case #	Scenario	Command	Return message	Status
9	To get supported PIDs from 0101-0120	0100	"0100\r"	Passed
		0100	"41 00 BF 9F A8 93 \r\r>"	Passed
		0100	"0100\r 41 00 BF 9F A8 93 \r\r>"	Passed
10		0120	"0120\r"	Passed

	To get supported PIDs from 0121-0140	0120	“41 20 80 07 F1 19 \r\r>”	Passed
		0120	“0120\r 41 20 80 07 F1 19 \r\r>”	Passed
11	To get supported PIDs from 0141-0160	0140	“0140\r”	Passed
		0140	“41 40 FE F0 00 00 \r\r>”	Passed
		0140	“0140\r 41 40 FE F0 00 00 \r\r>”	Passed
12	To get engine coolant temperature	0105	“0105\r”	Passed
		0105	“41 05 88 \r\r>”	Passed
		0105	“0105\r 41 05 88 \r\r”	Passed
13	To get engine RPM	010C	“010C\r”	Passed
		010C	“41 0C 09 92 \r\r>”	Passed
		010C	“010C\r 41 0C 09 92 \r\r>”	Passed
14	To get vehicle speed	010D	“010D\r”	Passed
		010D	“41 0D 00 \r\r>”	Passed
		010D	“010D\r 41 0D 00 \r\r>”	Passed
15	To get MAF air flow rate	0110	“0110\r”	Passed
		0110	“41 10 01 1B \r\r>”	Passed
		0110	“0110\r 41 10 01 1B \r\r>”	Passed
16	To get control module voltage	0142	“0142\r”	Passed
		0142	“41 42 36 41 \r\r>”	Passed
		0142	“0142\r 41 42 36 41 \r\r>”	Passed

Table 8. Test cases for session module

Testing for registration module would use UI testing form Xcode. Firstly, it would try to register for an existed account to see if the system would give error message. Then it would use a new username. It would try to have inconsistent passwords in password field

and confirm password field to see if the system would give error message. Then it would use consistent password to register for a new account. Following codes were testing codes for registration module.

```
1. -(void)testRegisterFunction {
2.     XCUIApplication *app = [[XCUIApplication alloc] init];
3.     [app.buttons[@"Register"] tap];
4.     XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
5.     [emailAddressTextField tap];
6.     [emailAddressTextField typeText:@"793013053@qq.com"];
7.     XCTAssert(app.staticTexts[@"Sorry, this email had been used!"].exists);
8.
9.     [app.buttons[@"Clear text"] tap];
10.    [emailAddressTextField typeText:@"w793013053@gmail.com"];
11.    XCTAssert(!app.staticTexts[@"Sorry, this email had been used!"].exists);
12.
13.    XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
14.    [passwordSecureTextField tap];
15.    [passwordSecureTextField typeText:@"123"];
16.    XCUIElement *confirmPasswordSecureTextField = app.secureTextFields[@"Confirm
password"];
17.    [confirmPasswordSecureTextField tap];
18.    [confirmPasswordSecureTextField typeText:@"12"];
19.    XCUIElement *submitButton = app.buttons[@"Submit"];
20.    [submitButton tap];
21.    XCTAssert(app.alerts[@"Error!"].exists);
22.
23.    [app.alerts[@"Error!"].buttons[@"OK"] tap];
24.    [confirmPasswordSecureTextField tap];
25.    [confirmPasswordSecureTextField typeText:@"123"];
26.    [submitButton tap];
27.    XCTAssert(app.alerts[@"Register Succeed"].exists);
28.    [app.alerts[@"Register Succeed"].buttons[@"OK"] tap];
29. }
```

Testing for login module would also use UI testing. Firstly, it would try to login with correct username and incorrect password to see if the system would give error message. Then it would incorrect username and correct password to see if the system would give error message. Finally, it would use correct username and password to login.

```
1. -(void) testLoginFunction{
2.     XCUIApplication *app = [[XCUIApplication alloc] init];
3.     XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
4.     [emailAddressTextField tap];
5.     [emailAddressTextField typeText:@"w793013053@gmail.com"];
6.
7.     XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
8.     [passwordSecureTextField tap];
9.     [passwordSecureTextField typeText:@"123456"];
10.    XCUIElement *loginButton = app.buttons[@"Login"];
```

```

11.     [loginButton tap];
12.     XCTAssert(app.staticTexts[@"Incorrect username or password"].exists);
13.
14.     [emailAddressTextField tap];
15.     [app.buttons[@"Clear text"] tap];
16.     [passwordSecureTextField tap];
17.     [app.buttons[@"Clear text"] tap];
18.     [loginButton tap];
19.     XCTAssert(app.staticTexts[@"Incorrect username or password"].exists);
20.
21.     [emailAddressTextField tap];
22.     [emailAddressTextField typeText:@"w7930130533@gmail.com"];
23.     [passwordSecureTextField tap];
24.     [passwordSecureTextField typeText:@"123"];
25.     [loginButton tap];
26.     XCTAssert(app.staticTexts[@"Incorrect username or password"].exists);
27.
28.     [emailAddressTextField tap];
29.     [app.buttons[@"Clear text"] tap];
30.     [emailAddressTextField typeText:@"w793013053@gmail.com"];
31.     [passwordSecureTextField tap];
32.     [app.buttons[@"Clear text"] tap];
33.     [passwordSecureTextField typeText:@"123"];
34.     [loginButton tap];
35.
36.     XCTAssert(!emailAddressTextField.exists);
37.     XCTAssert(!passwordSecureTextField.exists);
38.     XCTAssert(!loginButton.exists);
39.     XCTAssert(!app.buttons[@"Register"].exists);
40.     XCTAssert(!app.staticTexts[@"Incorrect username or password"].exists);
41. }

```

Testing for resetting password module would also use UI testing in. Firstly, it would try to enter wrong username to see if the system would give error message. Then it would click reset button without having any username to see if the system would crash. Finally, it would enter correct username and click reset button. The tester should also check the email to ensure new password had been sent. The new password should be same with password value stored in database.

```

1. -(void)testForgetPasswordFunction{
2.     XCUIApplication *app = [[XCUIApplication alloc] init];
3.     [app.buttons[@"Help"] tap];
4.     XCUIElement *emailTextField = app.textFields[@"Email address"];
5.     XCTAssert(emailTextField.exists);
6.
7.     [emailTextField tap];
8.     [emailTextField typeText:@"w7930130533@gmail.com"];
9.     XCUIElement *resetButton = app.buttons[@"Reset"];
10.    [resetButton tap];
11.    XCTAssert(app.alerts[@"Error!"].exists);
12.

```

```

13.     XCUIElement *okButton = app.alerts[@"Error!"].buttons[@"OK"];
14.     [okButton tap];
15.     XCUIElement *clearTextButton = app.buttons[@"Clear text"];
16.     [clearTextButton tap];
17.     [resetButton tap];
18.
19.     XCTAssert(app.alerts[@"Error!"].exists);
20.     [okButton tap];
21.     [emailTextField typeText:@"w793013053@gmail.com"];
22.     [resetButton tap];
23.
24.     XCTAssert(!resetButton.exists);
25. }

```

Testing for resetting connection module would also use UI testing in. Firstly, it would login. Secondly, it will go the connection screen. Thirdly, it would plug the OBD-II scanner in. Finally, it would connect to scanner's Wi-Fi and users could start recording trip information.

```

1.  XCUIApplication *app = [[XCUIApplication alloc] init];
2.  XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
3.  [emailAddressTextField tap];
4.  [emailAddressTextField typeText:@"w793013053@gmail.com"];
5.
6.  XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
7.  [passwordSecureTextField tap];
8.  [passwordSecureTextField typeText:@"123"];
9.  [app.buttons[@"Login"] tap];
10. [app.navigationBars[@"Riding Mechanic"].buttons[@"Wi Fi"] tap];
11. [app.buttons[@"Device plugged please scan now"] tap];
12. [app.buttons[@"Yes, Wifi conneted"] tap];
13.
14. XCTAssert(app.buttons[@"Start Trip"].exists);

```

Testing for starting trip module would also use UI testing in. Firstly, it would connect login and connect to OBD-II scanner's Wi-Fi. Secondly, it would use star trip function to read data from the car and show vehicle information. Finally, it would stop recording and choose to save trip information.

```

1.  XCUIApplication *app = [[XCUIApplication alloc] init];
2.  XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
3.  [emailAddressTextField tap];
4.  [emailAddressTextField typeText:@"w793013053@gmail.com"];
5.
6.  XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
7.  [passwordSecureTextField tap];
8.  [passwordSecureTextField typeText:@"123"];
9.  [app.buttons[@"Login"] tap];
10. [app.navigationBars[@"Riding Mechanic"].buttons[@"Wi Fi"] tap];

```

```

11. [app.buttons[@"Device plugged please scan now"] tap];
12. [app.buttons[@"Yes, Wifi conneted"] tap];
13. [app.buttons[@"Start Trip"] tap];
14.
15. XCUIElementQuery *tablesQuery = app.tables;
16. [tablesQuery.staticTexts[@"Driving Distance"] tap];
17. [tablesQuery.staticTexts[@"Realtime MPG"] tap];
18. [tablesQuery.staticTexts[@"Total Fuel Consumption"] tap];
19. [tablesQuery.staticTexts[@"Fuel Cost"] tap];
20. [tablesQuery.staticTexts[@"Sharp Braking"] tap];
21. [app.navigationBars[@"Trip Information"].buttons[@"Stop"] tap];
22. [app.alerts[@"Do you want to store trip information?"].buttons[@"YES"] tap];

```

Testing for trip analysis module would also use UI testing in. Firstly, it would connect login and go to analysis screen. Then, it would show trip information based on current date. Finally, it would select different date to search trip information.

```

1. XCUIApplication *app = [[XCUIApplication alloc] init];
2. XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
3. [emailAddressTextField tap];
4. [emailAddressTextField typeText:@"w793013053@gmail.com"];
5.
6. XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
7. [passwordSecureTextField tap];
8. [passwordSecureTextField typeText:@"123"];
9. [app.buttons[@"Login"] tap];
10.
11. XCUIElementQuery *tabBarsQuery = app.tabBars;
12. [tabBarsQuery.buttons[@"analysis"] tap];
13.
14. XCUIApplication *app2 = app;
15. [app2.buttons[@"8"] tap];
16. [app2.buttons[@"10"] swipeRight];
17. [app2.buttons[@"7"] tap];
18. [app2.buttons[@"6"] tap];
19. [app2.buttons[@"5"] tap];
20. [app2.buttons[@"4"] tap];
21. [app2.buttons[@"3"] tap];
22. [app2.buttons[@"2"] tap];
23. [app2.buttons[@"1"] tap];
24. [tabBarsQuery.buttons[@"More"] tap];
25. [app2.tables.staticTexts[@"Account"] tap];
26. [app2.tables.buttons[@"Log Out"] tap];
27. [app.alerts[@"Log Out Succeed!"].buttons[@"Ok"] tap];

```

Testing for more module would also use UI testing. Firstly, it would connect login and go to more screen. Secondly, it would go to speed alert screen to change speed limit. Thirdly, it would go to tire driving screen to change tired driving limit. Fourthly, it would go to engine coolant temperature screen to change coolant temperature limit. Fifthly, it would go to vehicle screen to change fuel price. Sixthly, it would go to

adapter screen and about screen to view detailed information. Finally, it would use function of changing password and function of logout.

```
1. XCUIApplication *app = [[XCUIApplication alloc] init];
2. XCUIElement *emailAddressTextField = app.textFields[@"Email address"];
3. [emailAddressTextField tap];
4. [emailAddressTextField typeText:@"w793013053@gmail.com"];
5.
6. XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
7. [passwordSecureTextField tap];
8. [passwordSecureTextField typeText:@"123"];
9. [app.buttons[@"Login"] tap];
10. [app.tabBars.buttons[@"More"] tap];
11.
12. XCUIElementQuery *tablesQuery2 = app.tables;
13. XCUIElementQuery *tablesQuery = tablesQuery2;
14. [tablesQuery.staticTexts[@"Alert"] tap];
15. [tablesQuery.switches[@"Alert"] tap];
16. [tablesQuery.switches[@"Alert"] tap];
17. [tablesQuery.staticTexts[@"Speed Alert"] tap];
18. [tablesQuery.switches[@"Speed Alert"] tap];
19. [tablesQuery.switches[@"Speed Alert"] tap];
20.
21. XCUIElement *textField = [[tablesQuery2.cells containingType:XCUIElementTypeStaticText identifier:@"Speed Limit"] childrenMatchingType:XCUIElementTypeTextField].element;
22. [textField tap];
23. [textField typeText:@"80"];
24. [app.navigationBars[@"Speed Alert"].buttons[@"Alert"] tap];
25.
26. XCUIElement *tiredDrivingAlertStaticText = tablesQuery.staticTexts[@"Tired Driving Alert"];
27. [tiredDrivingAlertStaticText tap];
28. [tablesQuery.switches[@"Tired Driving Alert"] tap];
29. [tablesQuery.switches[@"Tired Driving Alert"] tap];
30.
31. XCUIElement *textField2 = [[tablesQuery2.cells containingType:XCUIElementTypeStaticText identifier:@"Tired Driving Hour Limit"] childrenMatchingType:XCUIElementTypeTextField].element;
32. [textField2 tap];
33. [textField2 typeText:@"4"];
34.
35. XCUIElement *alertButton = app.navigationBars[@"Tired Driving Alert"].buttons[@"Alert"];
36. [alertButton tap];
37. [tiredDrivingAlertStaticText tap];
38. [alertButton tap];
39. [tablesQuery.staticTexts[@"Water Temperature Alert"] tap];
40. [tablesQuery.switches[@"Water Temperature Alert"] tap];
41. [tablesQuery.switches[@"Water Temperature Alert"] tap];
42.
43. XCUIElement *textField3 = [[tablesQuery2.cells containingType:XCUIElementTypeStaticText identifier:@"Water Temperature Limit"] childrenMatchingType:XCUIElementTypeTextField].element;
44. [textField3 tap];
```

```

45. [textField3 typeText:@"220"];
46. [app.navigationBar[@"Water Temperature Alert"].buttons[@"Alert"] tap];
47. [app.navigationBar[@"Alert"].buttons[@"More"] tap];
48.
49. XCUIElement *vehicleStaticText = tablesQuery.staticTexts[@"Vehicle"];
50. [vehicleStaticText tap];
51.
52. XCUIElement *textField4 = [tablesQuery2.cells childrenMatchingType:XCUIElementTypeTextField].element;
53. [textField4 tap];
54. [textField4 typeText:@"2.35"];
55.
56. XCUIElement *moreButton = app.navigationBar[@"Vehicle"].buttons[@"More"];
57. [moreButton tap];
58. [vehicleStaticText tap];
59. [moreButton tap];
60. [tablesQuery.staticTexts[@"Adapter"] tap];
61. [app.navigationBar[@"Connectivity"].buttons[@"More"] tap];
62. [tablesQuery.staticTexts[@"Unit"] tap];
63. [app.navigationBar[@"Unit"].buttons[@"More"] tap];
64. [tablesQuery.staticTexts[@"About"] tap];
65. [app.navigationBar[@"About"].buttons[@"More"] tap];
66. [tablesQuery.staticTexts[@"Account"] tap];
67. [tablesQuery.staticTexts[@"Reset Password"] tap];
68.
69. XCUIElement *currentPasswordSecureTextField = app.secureTextFields[@"Current password"];
70. [currentPasswordSecureTextField tap];
71. [currentPasswordSecureTextField typeText:@"123"];
72.
73. XCUIElement *newPasswordSecureTextField = app.secureTextFields[@"New password"];
74. [newPasswordSecureTextField tap];
75. [newPasswordSecureTextField tap];
76. [newPasswordSecureTextField typeText:@"111"];
77.
78. XCUIElement *confirmNewPasswordSecureTextField = app.secureTextFields[@"Confirm new password"];
79. [confirmNewPasswordSecureTextField tap];
80. [confirmNewPasswordSecureTextField tap];
81. [confirmNewPasswordSecureTextField typeText:@"111"];
82. [app.buttons[@"Submit"] tap];
83. [app.alerts[@"Password change succeed"].buttons[@"OK"] tap];
84. [tablesQuery.buttons[@"Log Out"] tap];
85. [app.alerts[@"Log Out Succeed!"].buttons[@"Ok"] tap];

```

Testing for health scan module would also use UI testing. Firstly, it would connect login and go to health scan screen. The system would show diagnostic information when the screen was loaded. Finally, it would use re-diagnosis function to get newest diagnostic information from the vehicle.

```

1. XCUIApplication *app = [[XCUIApplication alloc] init];
2. XCUIElement *emailAddressTextField = app.textFields[@"Email address"];

```

```
3. [emailAddressTextField tap];
4. [emailAddressTextField typeText:@"w793013053@gmail.com"];
5.
6. XCUIElement *passwordSecureTextField = app.secureTextFields[@"Password"];
7. [passwordSecureTextField tap];
8. [passwordSecureTextField typeText:@"123"];
9. [app.buttons[@"Login"] tap];
10. [app.tabBars.buttons[@"scan"] tap];
11. [app.buttons[@"Rediagnosis"] tap];
```

6.2.3 Code coverage

As we had used UI testing from Xcode, it also provided users with testing code coverage. Figure 20 had showed the testing code coverage from Xcode. As it indicated in the figure, some classes were not covered. The reason why they were had little testing code coverage was that some classes were external classes or from third-party library which had lots of functions. But only one or two functions would be used for this application among those functions.






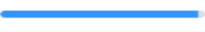

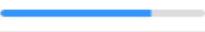

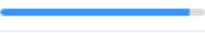

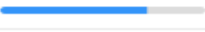

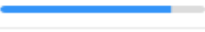



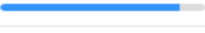

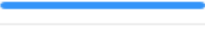

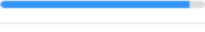

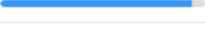

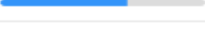

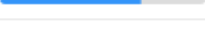

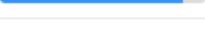

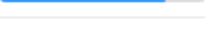





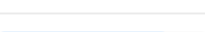

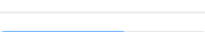

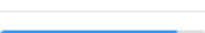

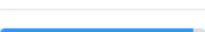

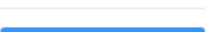

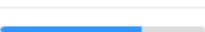

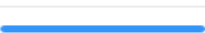

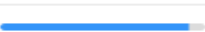

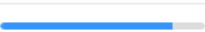

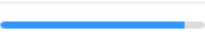

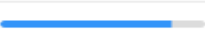
















Name	Coverage
▼  RidingMechanic.app	
▶  WaterTemperatureAlertTableViewCell.m	
▶  LoginViewController.m	
▶  ResetPasswordViewController.m	
▶  RestAPI.m	
▶  SetInitialStatus.m	
▶  CarModelViewController.m	
▶  AboutViewController.m	
▶  ConnectivityTableViewCell.m	
▶  AlertTableViewCell.m	
▶  AccountTableViewCell.m	
▶  ForgetPasswordViewController.m	
▶  Reachability.m	
▶  UnitTableViewCell.m	
▶  AlertTableViewController.m	
▶  SpeedAlertTableViewCell.m	
▶  AppDelegate.m	
▶  ASDayPicker.m	
▶  AnalysisViewController.m	
▶  TiredDrivingAlertTableViewCell.m	
▶  HealthScanViewController.m	
▶  SendCommand.m	
▶  SettingsTableViewCell.m	
▶  ShowTripInfoTableViewCell.m	
▶  ConnectionViewController.m	
▶  main.m	
▶  RegisterViewController.m	
▶  VehicleTableViewCell.m	
▶  Session.m	
▶  MenuViewController.m	
▶  RecvMessageAnalysis.m	
▶  HcdProcessView.m	
▶  Base64Transcoder.m	
▶  HSK_CFUtilities.m	
▶  NSData+Base64Additions.m	
▶  NSStream+SKPSMTPExtensions.m	
▶  SKPSMTPMessage.m	

Figure 20. Code coverage

6.3 Integration testing

As mentioned above, this application should be used along with an OBD-II W-Fi scanner connected to the OBD-II port on the car. So, integration testing was performed while the tester was driving. As previously mentioned, this application could read real-time data from a car. Reading real-time data was the core part of this project and it would involve most modules of this application. So, integration testing would focus on data comparison between data from the application and data from the car. Table 9 used those recorded data to compute the margin of error.

Driving distance recorded by the app (mile)	Driving distance recorded by the car (mile)	Margin of error (%)
0.00	0.0	0%
0.20	0.2	0%
0.82	0.8	2.5%
1.42	1.4	1.4%
1.67	1.7	1.7%
1.91	1.9	0.5%
2.39	2.4	0.4%
3.34	3.3	1.2%
3.78	3.8	0.5%
4.22	4.2	0.4%
5.07	5.1	0.5%
6.54	6.5	0.6%
11.56	12.0	0.3%
25.44	25.5	0.2%
28.64	28.7	0.2%
Geometric Mean:3.39, 3.4		0.9%

Table 9. Data comparison and margin of error

Figure 20 showed those data in coordinate system. Green points were consisted of those two kinds of data. X value was data recorded by the app and Y value represented data recorded by the car. The orange line represented $Y=X$.

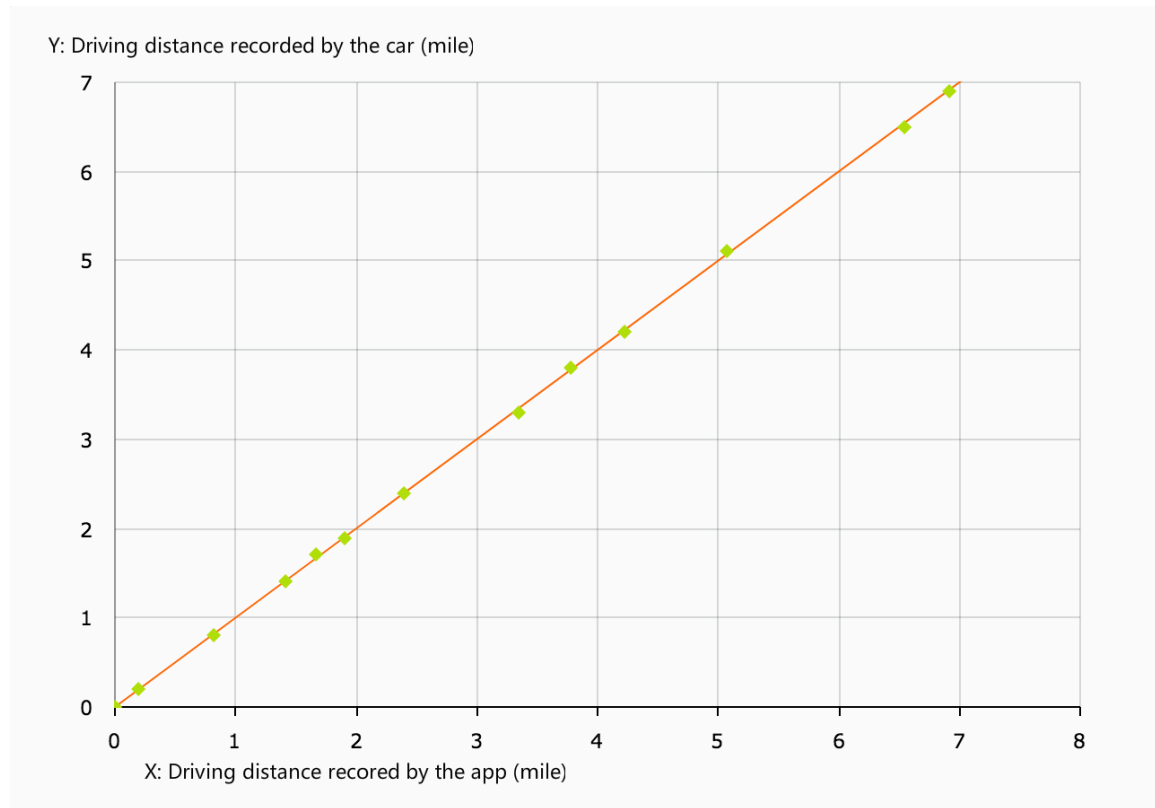


Figure 21. data comparison in coordinate system

6.4 System testing

System testing would focus on if the application satisfied users' requirements. Table 10 gave comparisons between required functions from users and functions in this application.

A user want to do (during a journey)	Application	Status
A user want to use an application to connect to a car	The app can help to build connection between the user and the car	Completed
A user want to get how long he had been driving	The app can provide the user with driving time	Completed

A user want to get specific speed	The app can provide the user with specific speed value	Completed
A user want to get specific RPM	The app can provide the user with specific RPM value	Completed
A user want to get average speed	The app can provide the user with average speed value	Completed
A user want to get fuel consumption	The app can provide the user with fuel consumption	Completed
A user want to get fuel cost	The app can provide the user with fuel cost	Completed
A user want to get engine coolant temperature	The app can provide the user with engine coolant temperature	Completed
A user want to get control module voltage	The app can provide the user with control module voltage	Completed
A user want to get sharp acceleration times	The app can provide the user with sharp acceleration times	Completed
A user want to get sharp braking times	The app can provide the user with sharp braking times	Completed
A user want to save related information of a journey	The app allows the user to register for an account to save related information of a journey	Completed
A user want to search journey information of specific date	The app allows the user to search journey information of that date	Completed
A user want to get diagnostic information of the car	The app can provide users with the car's diagnostic information	Completed
A user want to set an alert for over speed	The app allows the user to set up speed limit	Completed
A user want to set an alert for engine being overheated	The app allows the user to set up engine coolant temperature limit	Completed

A user want to set an alert for control module voltage being abnormal	The app allows the user to set up control module voltage limit	Completed
A user want to change measurement of unit	The app allows the user to change measurement of unit	Completed

Table 10. Achievements of requirements

7. Conclusion and Future Work

7.1 Conclusions

The overriding achievement of this thesis was use of software engineering in mobile application design. It had showed the process of analyzing user requirements, designing system architecture, constructing the whole system, implementation and testing final product which will be used by end users.

- Chapter 2 showed how to analyze users' requirements from different perspective. Functions that could be performed by users were described in detail in functional requirements. Non-functional requirements were related to system qualities.
- Chapter 3 described both goals and constrains of system architecture design. Then it gave an overview of what the application would look like when it was completed. It also pointed out some key issues and its solutions.
- Chapter 4 presented detailed system design including database table design and use of UML diagrams to analyze user scenarios, relationship between classes and interactions between each component in the system.
- Chapter 6 showed implementation of main functions. It used advantages of pseudocode to make logic of each process clear.
- Chapter 7 presented testing strategies and how they were used in testing. It also showed the process of testing and testing data included in each kind of testing.

Although this product had been tested for many times, there were some defects in it. User interfaces did not look very nice and did not have any appropriate message while loading data from server database. The screen did not look very fluent while logging out. Users could no select their own vehicles and find out some manufacture-defined fault codes.

7.2 Future Work

As mentioned above, there existed some weakness in the final product. The first thing of future work was to overcome those shortcomings. In addition, user interface could be

resigned and colored to give better user experience. More data could be worked out to provide more vehicle information for users. For example, giving limit speed while driving. Because this product only provided universal diagnostic information, some manufacturer-specific codes could not be recognized. The database could gather more information for both universal diagnostic codes and manufacturer-specific codes. It should also classify those codes. Users should have choices to select model type of their vehicles and then database could provide corresponding diagnostic codes. In a word, this application still had lots of work to do to become a real product.

8. Bibliography

- [1] Wikipedia, “ELM327”, July 2013.
- [2] United States Environmental Protection Agency, “On-Board Diagnostic (OBD) Regulations and Requirements: Questions and Answer”, 2003.
- [3] AA1Car Auto Diagnosis Repair Help, “Trouble Codes”, 2008. Available: <http://www.trouble-codes.com/>
- [4] Alex (Im) E, “Complete List of OBD Codes: Generic OBD2 (OBDII) & Manufacturer”, January 29, 2013. Available: <http://www.totalcardiagnostics.com/support/Knowledgebase/Article/View/21/0/genericmanufacturer-obd2-codes-and-their-meanings>
- [5] Elm Electronics Inc., “ELM327 AT Commands”, October 2010. Available: https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf
- [6] Microsoft Patterns & Practices Team, “Microsoft® Application Architecture Guide, 2nd Edition”, Nov 2009.
- [7] Craig Larman, “Object-Oriented Analysis and Design”, Jan 21, 2005.
- [8] Grady Booch, “Object-Oriented Analysis and Design with Applications (3rd Edition)”, 2004.
- [9] Ahmad Shuja, Jochen Krebs, “RUP Reference and Certification Guide”, 2007.
- [10] Scott W. Ambler, “The Elements of UML™ 2.0 Style”, May 2005.
- [11] Lucid Software Inc., “Database Structure and Design Tutorial”, last updated 2017. Available: <https://www.lucidchart.com/pages/database-diagram/database-design>
- [12] Noah Stupak, Nicholas DiFonzo *, Andrew J. Younge, Christopher Homan, “SOCIALSENSE: Graphical user interface design considerations for social network experiment software”, Available online 11 December 2009.

Appendices