Course Title: 14:332:452 Software Engineering

Group 4

Onward (Traffic Monitoring)

Report 3

May 1, 2017

GitHub (e-archive): https://github.com/solejar/traffic_ru_ece
Website: http://www.onwardtraffic.com/

Team Members

Name	Email
Ridwan Khan	ridwankhan101@gmail.com
Brian Monticello	b.monticello23@gmail.com
Mhammed Alhayek	almoalhayek@gmail.com
Sean Olejar	solejar236@gmail.com
Lauren Williams	laurenwilliams517@gmail.com
Shubhra Paradkar	shubhra.paradkar@gmail.com

Individual Contributions Breakdown

All team members contributed equally.

Table of Contents

Summary of Changes

1. Updated Customer Statement to better reflect what Onward is.
2. Updated Glossary of Terms for additional terms since Report 1 submission.
3. Our system will now focus on zip code input for Heat Map Feature and not city because a city is not a granular enough input for our system.
4. Revised requirement analysis and priority weights.
5. Updated stakeholders to better represent to the stakeholders in our system.
6. Provided “sub use case” diagrams to break up the Use Case Diagram for better readability.
7. Added the fully-dressed use case descriptions for UC-4, ViewAlternateRoute, and UC-5, ViewGasCost, as these are the use cases to be implemented for demo #2.
8. Added system sequence diagrams for UC-4, ViewAlternateRoute, and UC-5, ViewGasCost.
9. Added effort estimation using use case points.
10. Revised and updated domain model and domain analysis to include UC-4, ViewAlternateRoute, and UC-5, ViewGasCost, such as adding the Gas Calculator conceptual object, and to better reflect the current state of our system, such as the removal of Parameter Storage conceptual object.
11. Revised and updated interaction diagrams to better reflect the current state of our system, the added design patterns, the current classes being used, and to reflect the addition of UC-4, ViewAlternateRoute, and UC-5, ViewGasCost to our system.
12. Revised and updated class diagram, data types and operation signatures to better reflect the current state of our system, the added design patterns, the current classes and methods being used, and to reflect the addition of UC-4, ViewAlternateRoute, and UC-5, ViewGasCost to our system.
13. Updated the system architecture, of identify subsystems, to explicitly show the services being used as well as adding the concept of Gas Calculator to the logic tier. Also provide the “bounds” table description for the new table in our database.
14. Updated user interface and user effort estimation.

Section 1: Customer Statement of Requirements

Modern navigation applications allow the user to enter a starting location A and a destination B and then calculate the best route between both locations, all the while giving an estimation of how long the trip may take. These applications do not, however, allow for users to see the future conditions of traffic and get an idea of how long their trip may take beforehand. As a result, users' ability to effectively plan ahead and adjust for traffic related issues is limited. Being able to more accurately measure trip time is an incredibly valuable asset for commercial users. As a result, an application that allows for a user to view traffic conditions in an area or along a route days before their travel would be meaningful and effective for a large group of users. Although there are several applications that individually display current traffic and weather conditions, they do not take into account historic traffic and weather, and will not calculate the cost of a particular trip. Most importantly, these measures cannot be taken days in advance, which is a necessity for most users. As the customer, we envision an application that takes into account all of these factors simultaneously will create the most appropriate solution for every-day users.

Currently, most mapping applications are mobile oriented and often restrict the user from entering their lengthy travel plans in favor of ease-of-use for immediate, simple travel plans. A web-based application that focuses on helping accommodate users' longer plans would be more helpful in specifying the entirety of the user's trip. This way each location, date, time constraint, and weather condition could be thoroughly described, and a route could be planned that would take each of these factors into account. This application would be more fitting for a traveler that is in constant need of adjusting to different traffic patterns and is inconvenienced by limiting mobile-only software that only allows for basic itinerary information. With this in mind, however, as traveling users often do also spend time away from a computer, it is important that the web-application be responsive for mobile users.

In order to select the fastest **route** between two locations ahead of time, it is necessary to predict the traffic along all possible routes between those locations. But how can such a prediction be made? Historic traffic and weather records can be an effective indicator of future traffic in any given area. By using this historic information, this application would allow the user to see predicted traffic along the three fastest routes connecting two locations once they enter the time, day, and expected weather. Allowing the user to see predicted traffic along these routes will allow them to select the route that will likely minimize time wasted in traffic. A system that uses data from over a large period of time to identify patterns of traffic would likely be accurate enough to consistently help users do this. This would be useful if the user were moving to a new area and wanted to see what their commute would look like in the morning. They could enter their home address and work location, and would be able to see what their average daily commute may look like based on past traffic data. They could then adjust their route accordingly, depending on which routes have the least traffic during rush hour. In addition to

having a starting location and destination and intended time of travel, the user may have a weather they expect to travel in. Furthermore, if they have a specific day within 10 days in mind, then the application could use a weather forecast instead of user input for the weather parameter. In either case, using the historic traffic and weather records, an accurate representation of traffic conditions along a route can be shown. Roads along the route can be highlighted based on traffic severity. And if a route contains excessive traffic, an alternate route can be suggested along with traffic conditions on that route as well. A user would also be in need of this type of application if they wanted to plan a road trip. The user could figure out the optimal departure times on each day of their trip that would minimize time spent in traffic. This application would give meaningful information to users looking to plan a trip ahead of time, or investigate general traffic conditions along a certain route.

Along with predicting the future traffic along a specific route, it would also be useful for the user to see the general traffic trends in a larger area. We would like to have the ability to enter a zip code, a radius around that zip code, a date and/or time, and weather conditions in order to see the predicted traffic for the designated area. A traffic **heat map** could effectively accomplish this, allowing the user to see how much traffic is expected in a particular area at a particular time. By color-coding the traffic severity, users can easily get a visual overview of how congested certain roads are at a certain time. Particularly congested area would be identified as traffic **hot spots**. These would allow the user to identify and avoid routes that are expected to have heavy traffic, if possible. If the user doesn't want to manually enter the weather conditions, the application should also be able to use the weather forecast if the date of travel is less than 10 days away. The traffic will only be shown on major roads, in order to avoid visual clutter. A major road is defined as any interstate, highway, or county road. If the user-specified area does not contain any such roads, the heat map will simply select the roads with the highest percentile of average traffic. The user would also benefit from the ability to show or hide the traffic on this map by the severity of the predicted traffic. That way, the user can more effectively exclude lighter traffic from the visualisation, and focus on the areas with heavier traffic. Combining all of these tools, the user would then be able to use this **heat map** to make educated decisions about where and when they would like to plan their travels.

Another useful extension upon the basic route feature would be the ability to accommodate multiple stops in a user's travel plans. In other words, many users will not only need the ability to plan their route out from point A to point B, but also to plan out their **agenda** if they have multiple stops throughout their day or week. A user may have to run to the grocery store, go to the gym, and go to the mall all in one day. The user would like the ability to enter in all of these locations and find the route with the least amount of traffic between them. The user would benefit from being able to include time constraints to describe this trip. Examples of these constraints are the time frame in which they would like to accomplish all of their tasks, and the time at which they want to arrive at one of their locations. Considering these constraints, the user would like an application that can generate an agenda that minimizes total time in traffic. They would like their destinations displayed in order along with the optimal routes and suggest time of departure. This agenda feature would also allow the user to detail weather conditions

for a particular journey so that they can receive as accurate a trip agenda as possible. This feature would be effective for users who generally travel with multiple destinations in mind and do not have the ability to plan ahead without knowing how much traffic they will face. These users are also generally unaware of the best time to depart for each destination that would allow them to avoid traffic. Currently applications exist where the user is only asked to specify one destination and there is no intuitive way to add multiple destinations. An application that can create an agenda with multiple waypoints would be a huge convenience to a user over having to plan each route one at a time. Looking at the overall trip, rather than the individual routes, allows the application to more effectively minimize total time in traffic.

Currently available applications provide a good estimation of trip mileage, but they do not go any further in turning this information into something usable for the user. The user wants to know not only the mileage of their trip, but also details regarding gas consumption for their trip and therefore the costs associated with this gas consumption. The user can also use the cost of gas as a heuristic for Onward's best route/agenda recommendation. By calculating the cost along with the other individual features mentioned, the user would have a very direct plan of how to travel, when to travel, and the cost of travel. This application would provide that, allowing the users to see a route for their trip, the predicted cost of gas based on the route, and the predicted traffic based on the historic traffic records. Furthermore, if alternate routes are suggested due to excessive traffic on primary route, the user can compare the cost of travel between suggested routes.

The above paragraphs demonstrate the need for an application that can fill in the gaps where traditional navigation applications are lacking. While identification of routes and traffic at the current point in time is meaningful, users also have a need to understand and predict traffic ahead of time. Being able to understand future traffic trends on both specific routes and general areas would allow users to be more effective travellers. In addition, an application that gives users the ability to create, plan, and analyze their multi-stop trips and the traffic they will encounter ahead of time would be exceptionally beneficial to have. Many modern users strive to use technology to make conscientious, well-informed decisions about their day-to-day activities. Onward facilitates this part of the user's life, allowing them to make an accurate and efficient decision about their travel plans ahead of time. We intend for this product to be easily available to everyday users. We do not want the users to be overwhelmed and bothered with formalities such as login and registration. Login and registration would require user time and effort that we feel is unnecessary for our specific system. As the customer, we request that Onward be a product that is a pleasant and informative experience such that a user can be more informed about their travels.

Section 2: Glossary of Terms

Incident - Any event which contributes to an increase in traffic severity which can be reported. Examples include, but are not limited to, road closures, accidents, and detours. Due to the limitations of the incoming traffic data, congestion cannot be taken into consideration

Location - A user inputted point on the map. It is allowed to be defined on any level of granularity from as specific as an address, to as general as a zipcode.

Heat Map - A heat map refers to a zipcode scale map which contains color indicators along the major roads of the designated area. The color indicators vary along a spectrum, and serve to indicate the average traffic conditions on the roads.

Major Roads - Where applicable, defined as all interstates, highways, and county-roads. Where not applicable (i.e. few relevant roads of the above categories), Onward may also consider roads with the highest amount of traffic in the area as major roads.

Hot Spot - Either a subsection of a road or an intersection which is known to regularly exhibit very congested traffic.

Route - A path that connects any two locations together.

Primary Route - A route that is recommended to the user based on the user-determined heuristics (i.e. total time in traffic, gas cost).

Agenda - An element which consists of several, individual, contiguous routes which are combined together. While an agenda will typically end at the same location it starts at, this is not required by definition.

Heuristic - A tool that allows Onward to make decisions regarding how to predict future traffic.

Highlight - A continuous-color marking, along a road segment. Indicates traffic severity.

Landing Page - The homepage of the Onward website that the User first encounters.

Grid Box - An approximately 5 mile by 5 mile part of a map that is granular enough to segment roads for data collection.

Legs - The individual steps of a route.

Section 3: System Requirements

Functional Requirements:

Table 1: Enumerated Functional Requirements

Identifier	Priority	Requirement
REQ1	5	The system shall take user input for a zip code in the tri-state area, time of interest and a value for the range of desired traffic information.
REQ2	3	The system shall provide an option to take user input for day of week to travel and user expected weather.
REQ3	5	The system shall provide the option to automatically select weather conditions based on a 10 day forecast.
REQ4	4	The system shall allow a user to choose to show/hide information corresponding to one of 4 different levels of traffic severity.
REQ5	5	The system shall generate a map that shows traffic trends for the area specified by the user. It should highlight all major roads, and use different colors for different levels of traffic severity.
REQ6	2	The system shall generate statistical data about traffic patterns in a large scale area of interest.
REQ7	5	The system shall take input for a starting point and destination address as well as desired time of travel.
REQ8	4	The system allow a user to choose to show/hide information corresponding to the 4 different levels of traffic severity along their route.
REQ9	5	The system shall suggest a route which connects starting point to destination and shall display this route on a map.
REQ10	4	The system should offer an alternative route for the user to choose in addition to the primary route.
REQ11	2	The system shall display statistical data about traffic patterns along a route in the form of a graph.
REQ12	2	The system should provide an estimated time to get from starting point to destination.

REQ13	3	The system shall take as input multiple locations, and any time constraints for the travel.
REQ14	2	The system should allow users to input specified times they must get to certain destinations and constraints on the order to reach each destination.
REQ15	2	The system shall display the optimal route to reach the specified destination(s) while stopping at all the requested stops based on inputted constraints.
REQ16	2	The system shall suggest the optimal time to depart in order to encounter the least traffic based on inputted constraints.
REQ17	4	The system shall offer users the option to input vehicle gas economy information in order to receive information about the estimated cost of all suggested routes.
REQ18	2	The system shall provide current traffic conditions from starting point to destination.

Nonfunctional Requirements

Table 2: Enumerated nonfunctional requirements

Identifier	Priority	Requirement
REQ19	3	The system shall provide a mobile-responsive interface.
REQ20	5	The system shall collect traffic and weather data every hour and add any new incident information to the database.
REQ21	4	The system should be able to suggest routes or produce heat maps within 20 seconds of user inputted parameters.

On-Screen Appearance Requirements

Table 3: Enumerated functional requirements

Identifier	Priority	Requirement
REQ22	5	The system shall provide a graphical representation of the user's route or area of interest in map form.
REQ23	5	The system shall provide a form for the user to input his/her information about the route or area of interest.
REQ24	4	The system shall provide graph(s) of statistical information about the user's trip or area of interest based on the specified inputs alongside the map interface.

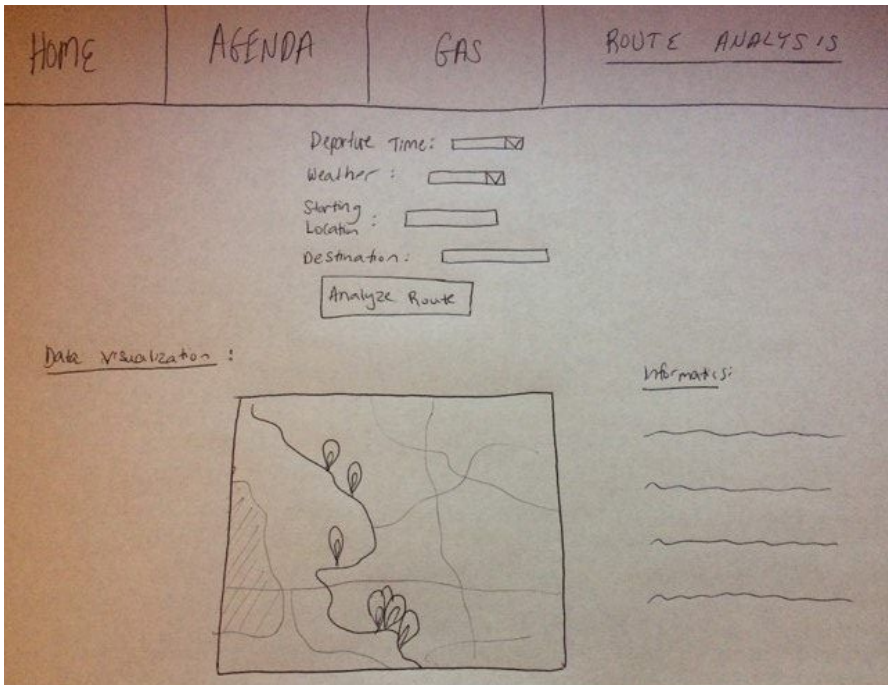

Figure 1: Preliminary Web Application Main Page

This above image was our very preliminary representation of the user interface for basic route analysis. This is what can be considered the conception of our user interface, and it evolved into the elegant user interface we show later on in the report. However to get to the final elegant and detailed user interface, we needed to have this preliminary rough sketch to have a clear communication. This shows the basic user inputs for a route, and an embedded Maps with a route displayed with some indications for traffic. From this simple, very rough and undetailed sketch, we began the planning of our user interface.

Section 4: Functional Requirements Specifications

Stakeholders

- **Commercial Users** - The regular day-to-day user of our application. They use it to plan their trip and commutes in the future.

- **Administrators** - The individuals responsible for developing and maintaining the application.

Actors and Goals

Table 4: Actor and goal information

Actor	Role	Goal
User	Initiating	To obtain some information about traffic either in a general area or along a specific route.
Database	Participating	To hold historical records of traffic and weather in the tri-state area and to provide those records upon request.
Mapping Service	Participating	To return information about areas and routes connecting two points, and to provide a map image which corresponds to a set of coordinates.
Weather Service	Participating	To return information about the forecasted weather in a certain zip code area and along routes.
Gas Feed Service	Participating	To return information about the current gas prices for cost estimation.

We have given generic names for the actors above that involve services our system uses. We would like to now name the specifics of each of these services.
The “Mapping Service” actor consists of the Google Maps API’s, such as the Google Maps Geocoding API[1] as well as the Google Maps Directions API[2], and also the Bing Maps Traffic API[3]. We have coupled these API’s for the simple fact that these are all mapping API’s and as a result are all mapping services.
The “Weather Service” actor is the Weather Underground API[4] (often times referred to as wunderground).
The “Gas Feed Service” actor is the Fuel Economy Web Services API[5].
Throughout the report we will refer to these “Services” in both text and in diagrams, as these are our actor names, and not explicitly by the specific API name. This is for the sake of consistency of using actor names and better readability. However we provided this mapping of “Service” to API’s so that the reader has a clear idea of what exact services are being used.

Use Cases

Casual Descriptions

For the following use cases, we derive from the functional requirements.

- ***UC-1: ViewHeatMap*** - Allows a user to enter a zip code and day of the week/time and weather condition to allow the user to view a heat map of traffic for a specific day of the week and time. The user can also view statistical data about traffic in the area of interest.
    - Associated Actors: User, Mapping Service, Database
    - *Extension:* If the date/time of travel is within the next 10 days, the weather can be forecasted.
    - Derived from requirement REQ1, REQ2, REQ4 - REQ6

- ***UC-2: ViewRoute*** - Allows a user to enter the details of their route on a day of week and time, and allows the user to view route recommendations using the day of the week/time and weather condition provided by the user. The user can also view statistical data about traffic on the route of interest.
    - Associated Actors: User, Mapping Service, Database
    - *Extension*: The user can view alternate routes.
    - *Extension*: The user can view gas prices along their routes.
    - *Extension:* If the date/time of travel is within the next 10 days, the weather can be forecasted.
    - *Extension*: The user can view current traffic conditions along route.
    - Derived from requirement REQ2, REQ7 - REQ9, REQ11, REQ12

- ***UC-3: CreateAgenda*** - Allows a user to enter three or more destinations and a time frame to suggest the best time/order in which to visit each destination using the day of week/time and weather condition provided by the user.
    - Associated Actors: User, Mapping Service, Database
    - *Extension*: The user can view gas prices along their routes.
    - *Extension:* If the date/time of travel is within the next 10 days, the weather can be forecasted.
    - Derived from requirement REQ2, REQ13-REQ16

- ***UC-4: ViewAlternateRoute*** - Allows a user to view a suggested alternate route. (optional sub use case, «extends» UC-2: ViewRoute).
    - Associated Actors: User, Mapping Service
    - Derived from requirement REQ10

- ***UC-5: ViewGasCost*** - Allows a user to view the projected gas cost of his/her trip . (optional sub use case, «extends» UC-2: ViewRoute, «extends» UC-4: *CreateAgenda*).
    - Associated Actors: User, Gas Feed Service, Mapping Service
    - Derived from requirement REQ17

- ***UC-6: ViewForecastInfo*** - Allows a user to take into account the forecasted weather over the next ten days instead of the user inputting these conditions. (optional sub use case, «extends» UC-1: ViewHeatMap, «extends» UC-2: ViewRoute, «extends» UC-3: *CreateAgenda*).
    - Associated Actors: User, Weather Service, Database
    - Derived from requirement REQ3

- ***UC-7: ViewCurrentRoute*** - Allows a user to view the current traffic conditions along their specified route for immediate travel (optional sub use case, «extends» UC-2: ViewRoute).
    - Associated Actors: User, Mapping Service
    - Derived from requirement REQ18

Use Case Diagram

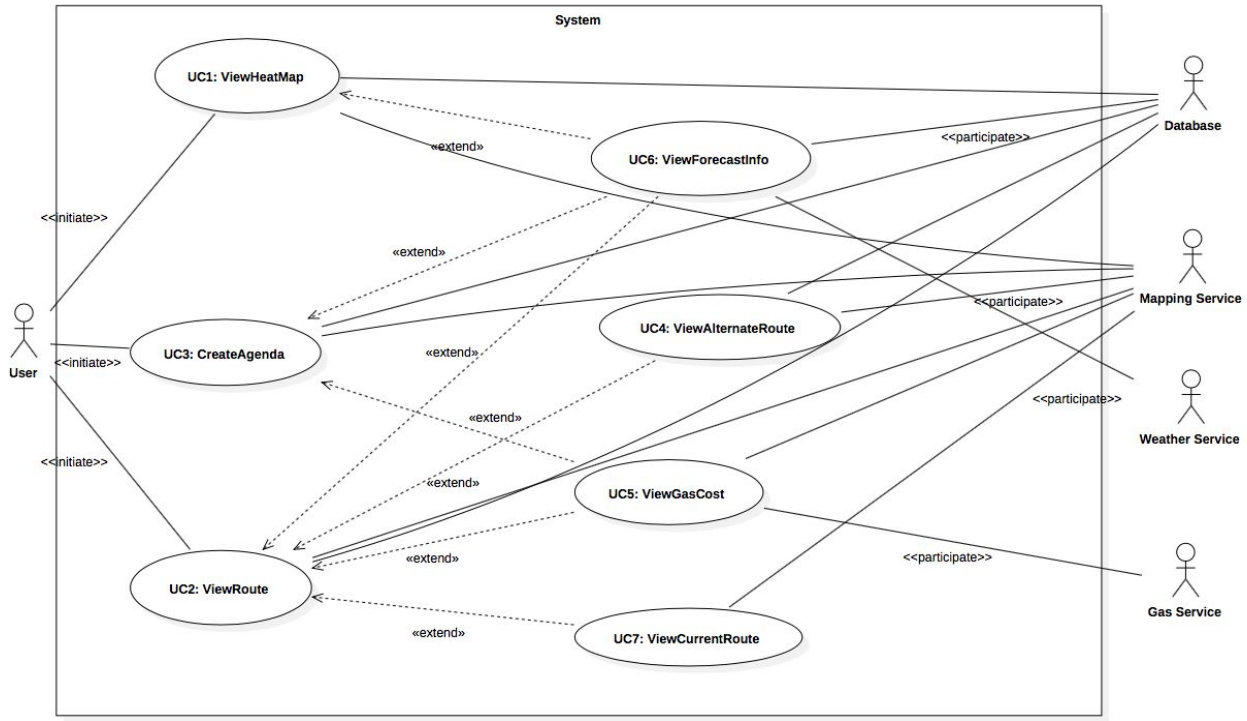

Figure 2: Use Case Diagram

The use case diagram in Figure 2 depicts the interaction of the user with the use-cases as well as the supporting actors: Database, Mapping Service, Weather Service, and Gas Service. UC1, UC2 and UC3 have multiple extensions, UC4-UC7. At first, we considered making a use case for every possible combination of inputs and scenarios. For example, a use case could have been viewing the heat map without forecasted weather information, and another use case could have been viewing the heat map with forecasted weather information. This approach however seemed to be clunky and redundant. Instead we made certain features extended use cases of the base cases for greater simplicity and understanding. As we have previously defined, the Mapping Service is the Google Maps API's, the Weather Service is the Weather Underground API and the Gas Service is the Fuel Economy Web Services API.

For the sake of clearer visibility, we have created 3 “Sub Use Case” Diagrams to better illustrate the diagram. We have split Figure 2 into 3 parts, where each part includes either UC1, UC2, or UC3 and their respective extensions.

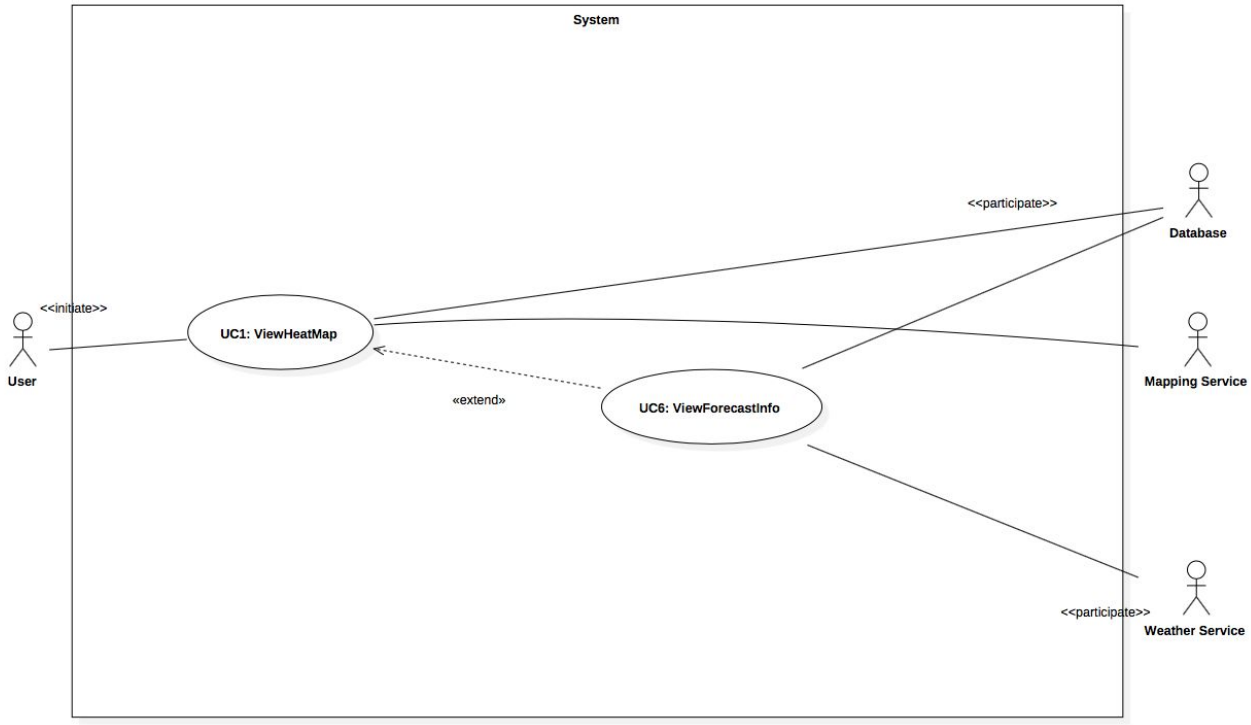

Figure 3: Sub Use Case Diagram - UC1 With Extension

Figure 3 is a Sub Use Case Diagram with UC1- ViewHeatMap, and its extension UC6-ViewForecastInfo.

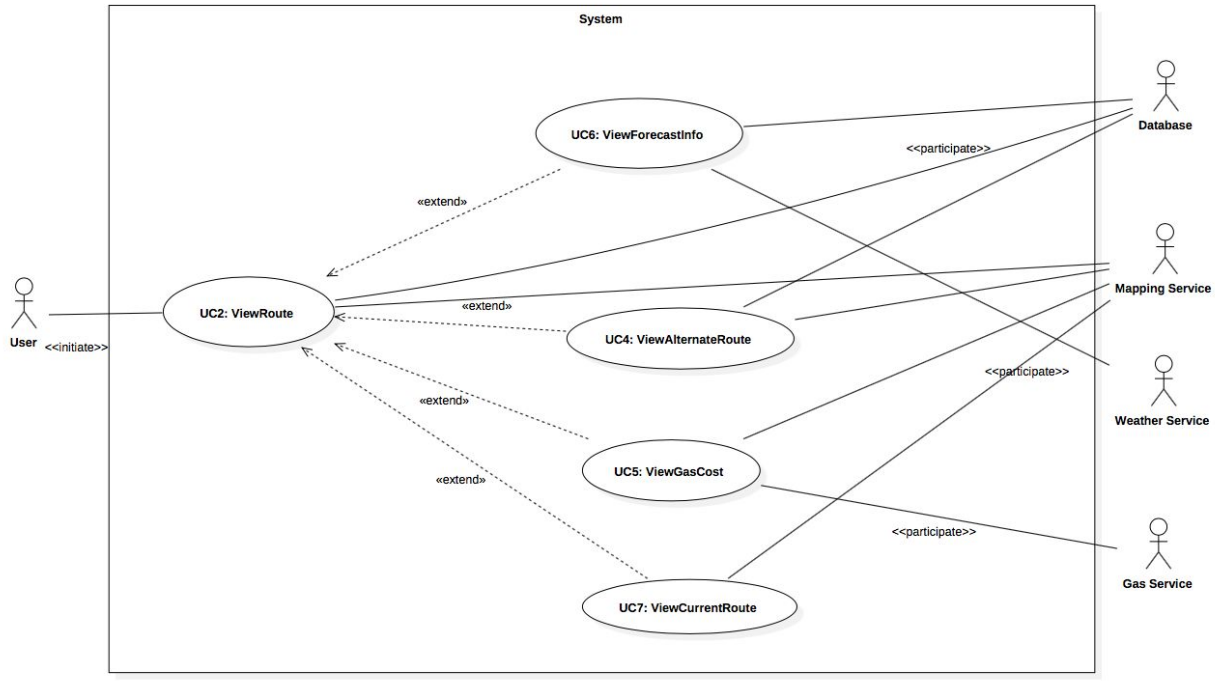

Figure 4: Sub Use Case Diagram - UC2 With Extensions

Figure 4 is a Sub Use Case Diagram with UC2 - ViewRoute, and its extensions UC6-ViewForecastInfo, UC4-ViewAlteranteRoute, UC5-ViewGasCost, and UC7-ViewCurrentRoute.

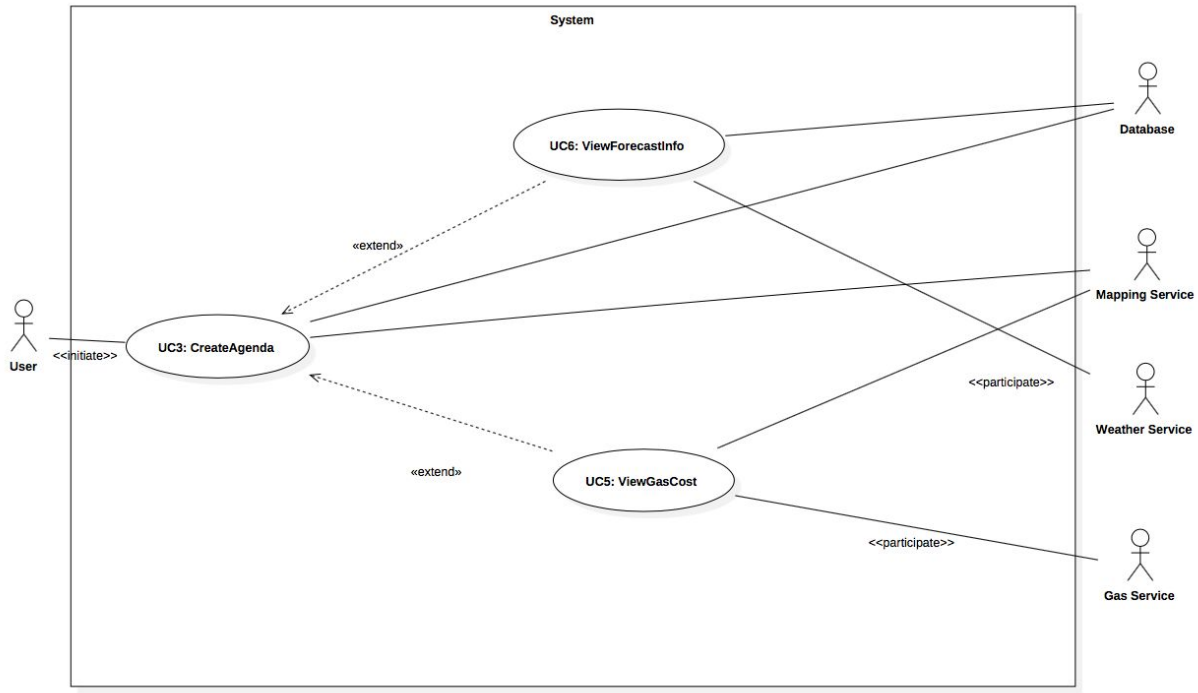

Figure 5: Sub Use Case Diagram - UC3 With Extensions

Figure 5 is a Sub Use Case Diagram with UC3 - ViewAgenda, and its extensions UC6-ViewForecastInfo, and UC5-ViewGasCost.

Traceability Matrix

REQ	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7
REQ1	5	X						
REQ2	3	X	X	X				
REQ3	5						X	
REQ4	4	X						
REQ5	5	X						
REQ6	2	X						
REQ7	5		X					
REQ8	4		X					
REQ9	5		X					
REQ10	4				X			
REQ11	2		X					
REQ12	2		X					
REQ13	3			X				
REQ14	2			X				
REQ15	2			X				
REQ16	2			X				
REQ17	4					X		
REQ18	2							X
Max PW		5	5	3	4	4	5	2
Total PW		19	21	12	4	4	5	2

Figure 6: Use Case Traceability Matrix

Note: Nonfunctional requirements are not included as they apply to all use cases and we did not want to complicate the traceability matrix with extra information. UC-4 - UC-7 are extended use cases. They extended upon UC-1 - UC-3, which is why the PW points of UC-4 to UC-7 are

lower. However since they have to extend Use cases, they will add great benefits to the quality of our system but also great complexity in the development, which the PW points do not reflect.

Use Cases Fully-Dressed Descriptions

Use Case UC-1:		**ViewHeatMap**
Related Requirements:		REQ1, REQ2, REQ4, REQ5, and REQ6
Initiating Actor:		User
Actor's Goal:		To view a heat map and statistical data about traffic around a specific location and specific time.
Participating Actors:		Database, Mapping Service
Preconditions:		- The system displays a menu of available functions: Heat Map, Route, and Agenda. - The traffic and weather data in the database is not empty for the user-specified location.
Postconditions:		- The system displays a map with traffic highlighted with selected severities. - The system displays a graph of statistical traffic data.
Flow of Events for Main Success Scenario		
→	1	**User** arrives on landing page and selects the “Heat Map” option.
←	2	**System** displays a map and input fields for zip code or city, range of miles, day of the week, time, weather conditions, and severity levels.
→	3	**User** inputs zip code or city, range of miles, day of week, time and a general weather condition and severity level.
←	4	**System** signals (a) **mapping service** to get mapping data and (b) **database** to fetch necessary traffic and weather data to analyze conditions of the area for the heat map.
←	5	**System** displays a map with highlighted traffic according to severity and a graph of statistical traffic data.
Flow of Events for Extensions (Alternate Scenarios):		
3a.**User** enters invalid information		
←	1	**System** resets the input field and asks for appropriate input.
3.b **User** inputs date/time within 10 days of the current date. Extends UC-6 ViewForecastInfo.		
←	1.	**System** displays map with highlighted traffic according to the severity and a graph of statistical traffic data based on the forecasted weather data for the user’s specified date/time.

Note: The alternate scenario 3.a is for invalid information input; this can only apply to an invalid zip code/city entered. All other user inputs will be limited, for example with dropdowns and checkboxes, so that the user cannot enter an invalid input. Scenario 3.b is of the extended use case where the user chooses an option to use forecasted weather if their travel is in the next 10 days. This scenario is further described in UC-6 description.

Use Case UC-2:		**ViewRoute**
Related Requirements:		REQ2, REQ7, REQ8, REQ9, REQ11, REQ12
Initiating Actor:		User
Actor's Goal:		To view a route from starting point to destination with traffic information and statistical data along the route.
Participating Actors:		Database, Mapping Service
Preconditions:		- The system displays a menu of available functions: Heat Map, Route, and Agenda. - The traffic and weather data in the database is not empty for the user-specified locations.
Postconditions:		- The system suggests a route on a map with traffic highlighted with selected severities along the route. - The system displays a graph of statistical traffic data along the route.
Flow of Events for Main Success Scenario		
→	1	**User** arrives on landing page and selects the “Route” option.
←	2	**System** displays a map and input fields for starting point, destination, day of the week, time, weather conditions, and severity levels.
→	3	**User** inputs starting point, destination, day of week, time and a general weather condition and severity level.
←	4	**System** signals (a) **mapping service** to get mapping and route data from starting point to destination and (b) **database** to fetch necessary traffic and weather data to analyze conditions of the route..
←	5	**System** displays a map with highlighted route and highlighted traffic according to checked severity levels and a graph of statistical traffic data.
Flow of Events for Extensions (Alternate Scenarios):		
3a.**User** enters invalid information		
←	1	**System** resets the input field and asks for appropriate input.
3.b **User** inputs date/time within 10 days of the current date. Extends UC-6 ViewForecastInfo.		

←	1	**System** displays map with highlighted traffic according to the severity and a graph of statistical traffic data based on the forecasted weather data for the user's specified date/time.
6. **User** chooses to view an alternate route. Extends UC-4 ViewAlternateRoute.		
→	1	**System** suggests an alternative route to take that has less traffic.
7. **User** chooses to show gas information along the suggested route. Extends UC-5 ViewGasCost.		
←	1	**System** displays projected gas costs associated with the route (and alternate routes).

Note: The alternate scenario 3.a is for invalid information input; this can only apply to an invalid addresses entered. All other user inputs will be limited, for example with dropdowns and checkboxes, so that the user cannot enter an invalid input. Scenario 3.b,6 and 7 is of the extended use cases that are further explained in their own use case descriptions. Use cases UC-4 and UC-5 are not fully dressed in this report.

Use Case UC-6:		**ViewForecastInfo** (sub-use case)
Related Requirements:		REQ3
Initiating Actor:		User
Actor's Goal:		To use forecasted weather for their location/route for a date/time in the next 10 days as a parameter for the heatmap, route, and agenda features.
Participating Actors:		Database, Weather Service
Preconditions:		- A user is at either "heat map", "route" or "agenda" page. - The weather service has a forecast available for a date within the next 10 days for the location specified.
Postconditions:		- The system uses forecasted weather for the traffic analysis.
Flow of Events for Main Success Scenario		
→	1	**User** selects option to use weather that is forecasted for next 10 days.
→	2	**User** inputs date and time that is the next 10 day along with other input parameters.
←	3	**System** fetches the weather data from **weather service** for the date/time in the location specified.

←	4	**System** goes back to Step 4 of UC-1 or UC-2, and uses this information for traffic analysis. Date, time and weather condition is printed below map.

Note: This use case extends the three base cases UC-1, UC-2 and UC-3. This option exists for all 3 base cases.

Use Case UC-4:		**ViewAlternateRoute** (sub-use case)
Related Requirements:		REQ10
Initiating Actor:		User
Actor's Goal:		To view an alternative route along with the main route for comparison. If they choose to do so, they can also view the traffic information along the alternate route.
Participating Actors:		Database, Mapping Service
Preconditions:		- A user is already at “route” page - The traffic and weather data in the database is not empty for the user-specified locations. -More than one route exists between starting and ending location.
Postconditions:		- The system suggests an alternate route on a map with traffic highlighted with selected severities along the route. - The system displays a graph of statistical traffic data along the route.
Flow of Events for Main Success Scenario		
→	1	**User** inputs starting point, destination, day of week, time and a general weather condition and severity level.
→	2	**User** selects option to view alternate route.
←	3	**System** signals (a) **mapping service** to get mapping and route data from starting point to destination with flag for alternate route and (b) **database** to fetch necessary traffic and weather data to analyze conditions of both routes.
←	4	**System** displays a map with highlighted routes and highlighted traffic according to checked severity levels on primary route and a graph of statistical traffic data for both routes.
→	5	**User** opts for seeing traffic condition along alternate route.
←	6	**System** displays highlighted traffic according to checked severity levels on alternate route and a graph of statistical traffic data for both routes.

Note: Restrictions on Google Maps API allow us to show highlighted traffic only on one route at a time.

Use Case UC-5:		**ViewGasCost** (sub-use case)
Related Requirements:		REQ17
Initiating Actor:		User
Actor's Goal:		To view cost of gas of a trip along a selected route.
Participating Actors:		Gas Feed Service, Mapping Service
Preconditions:		- A user is at “route” page.
Postconditions:		- The system prints the cost of gas along the specific route.
Flow of Events for Main Success Scenario		
→	1	**User** selects option to view the gas cost of the trip along the route.
→	2	**User** inputs the combined MPG as well as the fuel type of their vehicle of choice.
←	3	**System** signals (a) **gas feed service** to fetch gas prices and (b) and **mapping service** to get miles of travel based on user inputs and perform a calculation to get the cost of gas for the route
←	4	**System** prints the cost of gas below the map upon displaying the route(s) on the map.

Here we have fully-dressed the use cases UC-1,UC-2, UC-6, UC-4 and UC-5. UC-1,UC-2 and UC-6 were for the most part implemented for demo #1. Some additional parts of these use cases, such as generating graphs, are being added for demo #2. UC-4 and UC-5 are also being implemented for demo #2 to make the features of Heat Map and Route more robust.

After demo #1 we evaluated how much time we had left until the next demo and did a feasibility check on what we can get done. In doing this feasibility check, we realized there is not enough time left in the semester to implement the Agenda feature (UC-3) and the current conditions feature (UC-7). For this reason we have not fully dressed these use cases, although we have provided casual descriptions for these use cases in the “Casual Descriptions” section. As a result, we will not address UC-3 and UC-7 any further in this report, aside from Section 5, as they will not be developed for demo #2. These would be use cases that we could have as future work. We will make all the use cases that are being developed for the demo #2 as refined as possible and focus our energies on making these well implemented.

The remainder of this report, aside from Section 5, will focus only on UC-1,UC-2, UC-6, UC-4 and UC-5.

System Sequences Diagrams

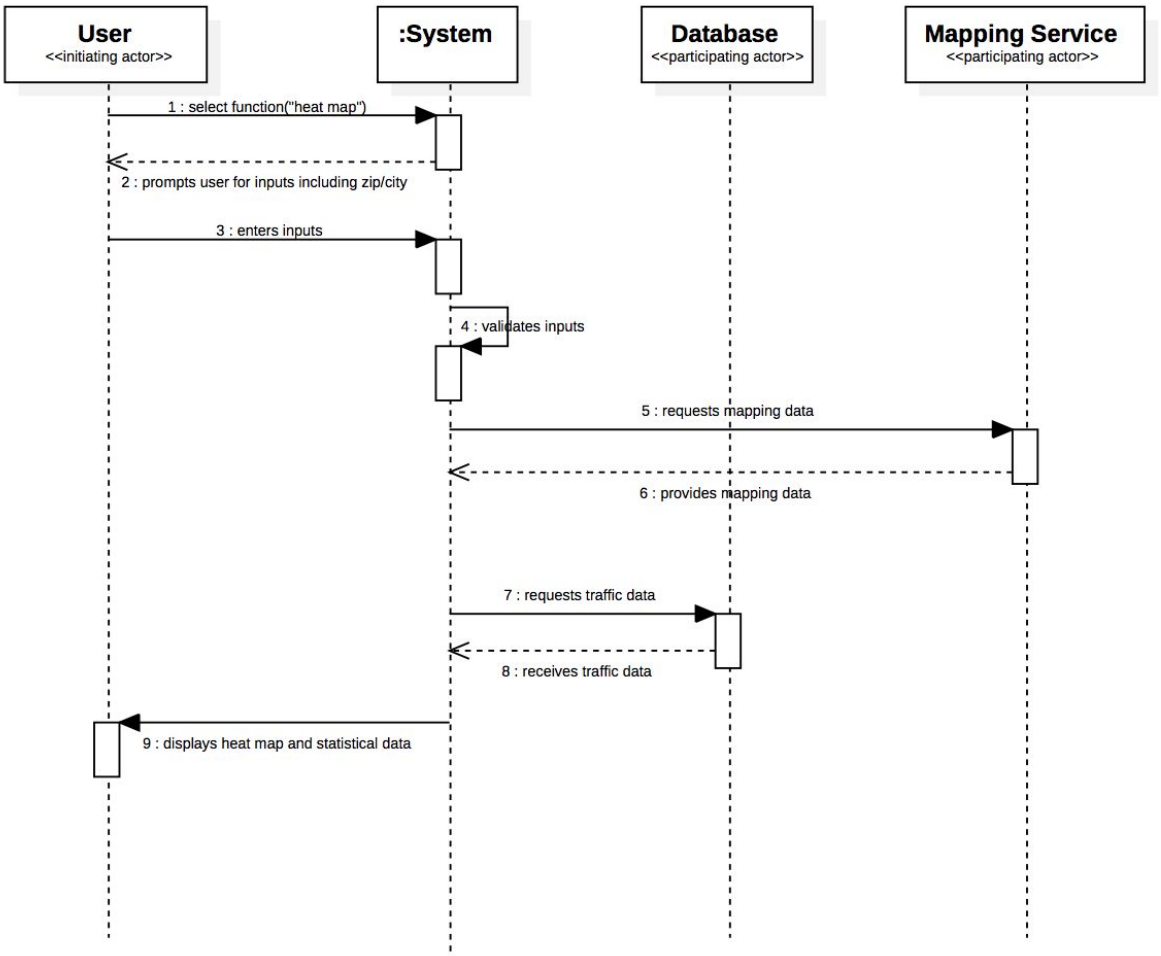

Figure 7: UC-1 Main Success Scenario Sequence Diagram

Figure 7 is a sequence diagram of the main success scenario involving the heat map feature base case, UC-1. The inputs prompted to user and inputted in steps 2 and 3 include zip code or city, range of miles, day of the week, time, weather conditions, and severity levels. This sequence diagram illustrates when the user does not use forecasted weather. The numbering of the sequence diagram is not intended to match the fully dressed description; the numbering here was done by the UML tool used.

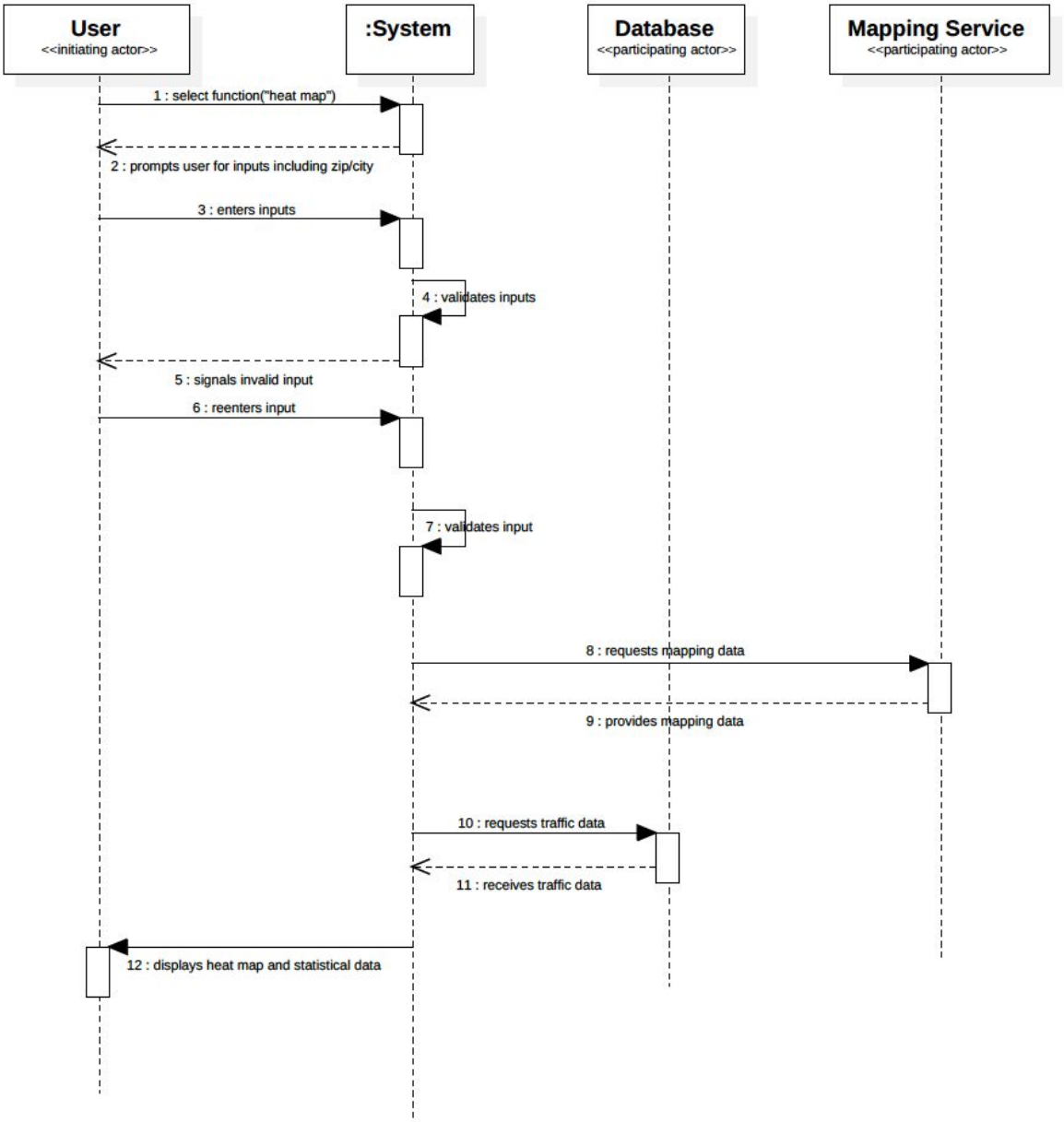

Figure 8: UC-1 Alternate Scenario Sequence Diagram

Figure 8 is a sequence diagram of the alternate scenario involving the heat map feature base case, UC-1, where the user enters an invalid input. The only invalid input the user could enter is city or zip code because the remainder of the inputs will be limited by the user interface. The inputs prompted to the user and inputted in steps 2, 3 and 6 include zip code or city, range of miles, day of the week, time, weather conditions, and severity levels.

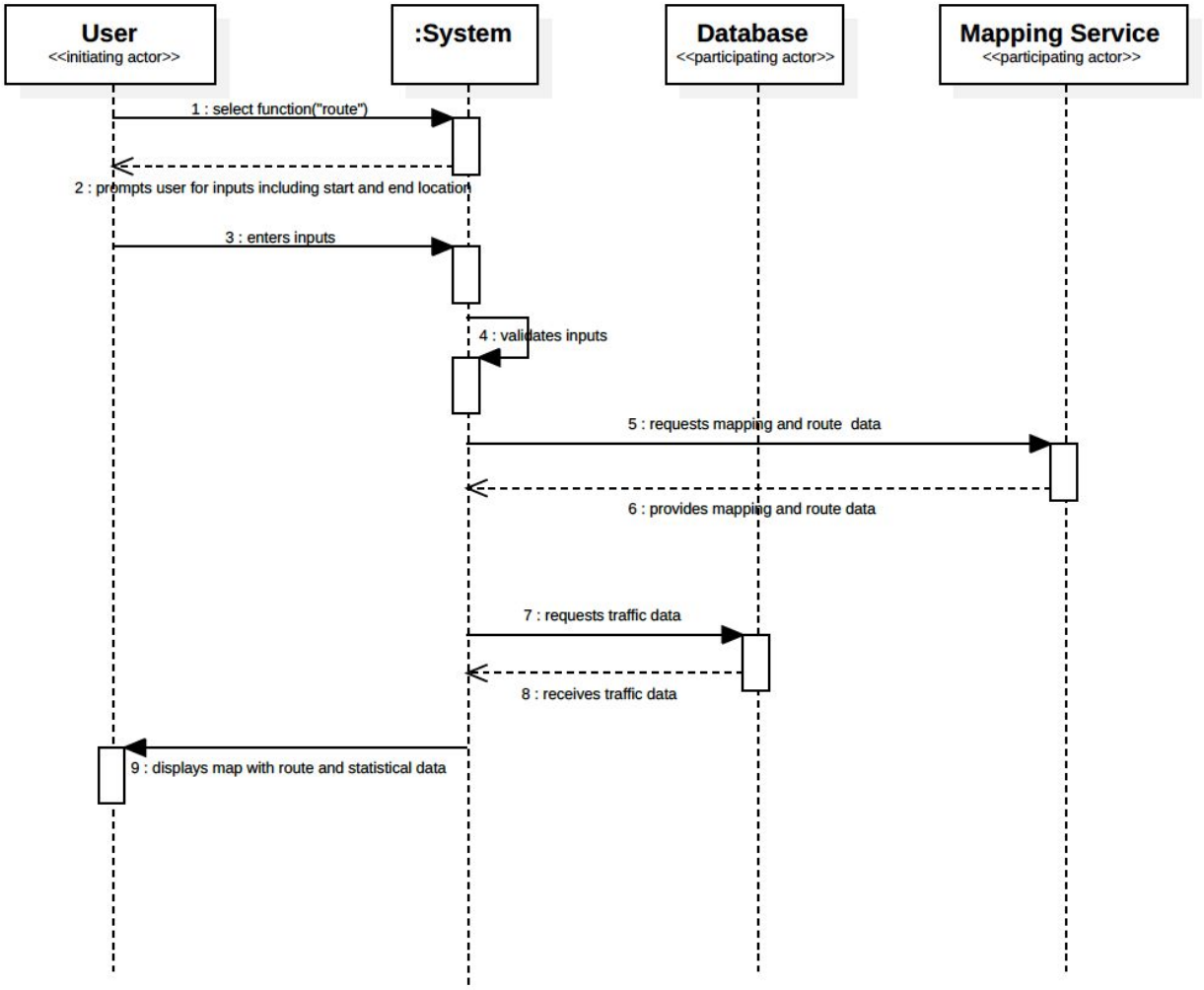

Figure 9: UC-2 Main Success Scenario Sequence Diagram

Figure 9 is a sequence diagram of the main success scenario involving the route feature base case, UC-2. The inputs prompted to user and inputted in steps 2 and 3 include starting point, destination, day of the week, time, weather conditions, and severity levels. This sequence diagram illustrates when the user does not use forecasted weather.

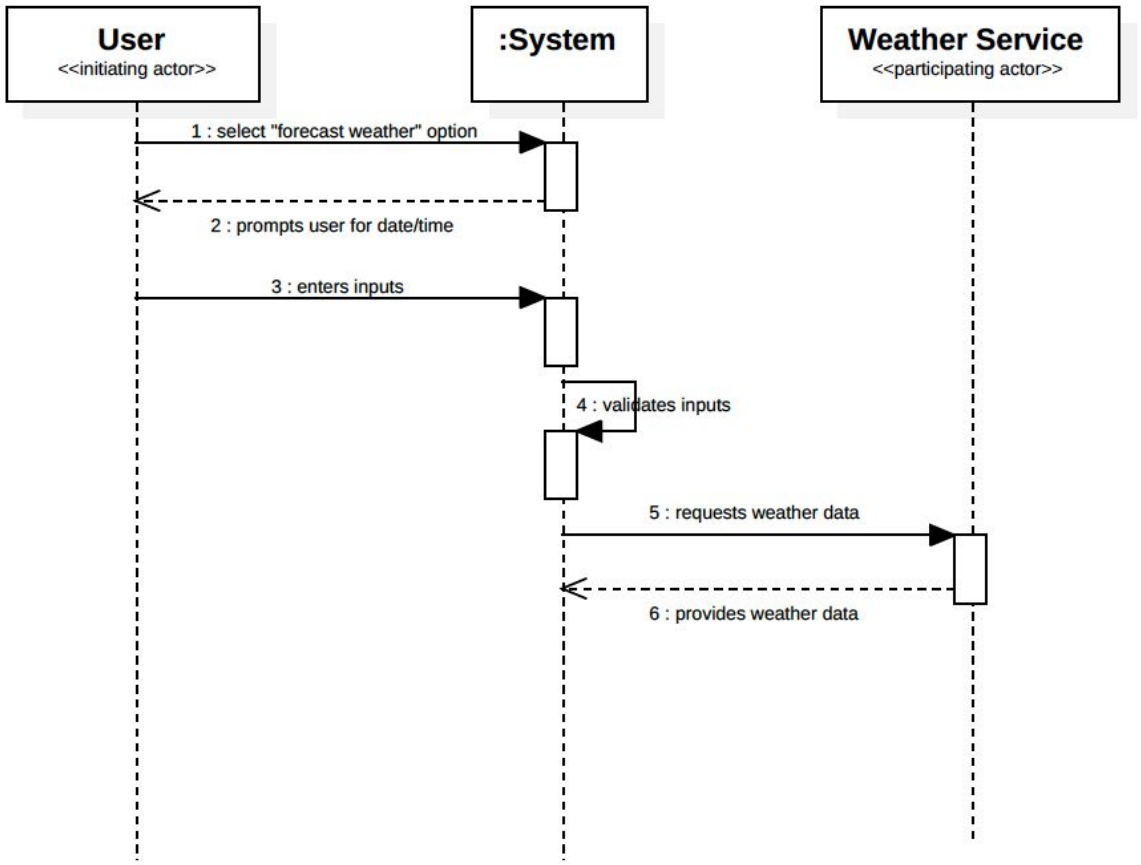

Figure 10: UC-6 Main Success Scenario Sequence Diagram

Figure 10 is the main success scenario sequence diagram for UC-6, where forecasted weather is used. This sequence diagram can be used for UC-1, UC-2 or UC-3 where the user may want to use forecasted weather instead of the inputting weather as this is an extension or alternative scenario of all the base cases. The inputs in steps 2 and 3 are of a date that is in the next 10 days and a time. The option “forecast weather” is a generic phrasing of the option. It is phrased as “Forecast” in the UI.

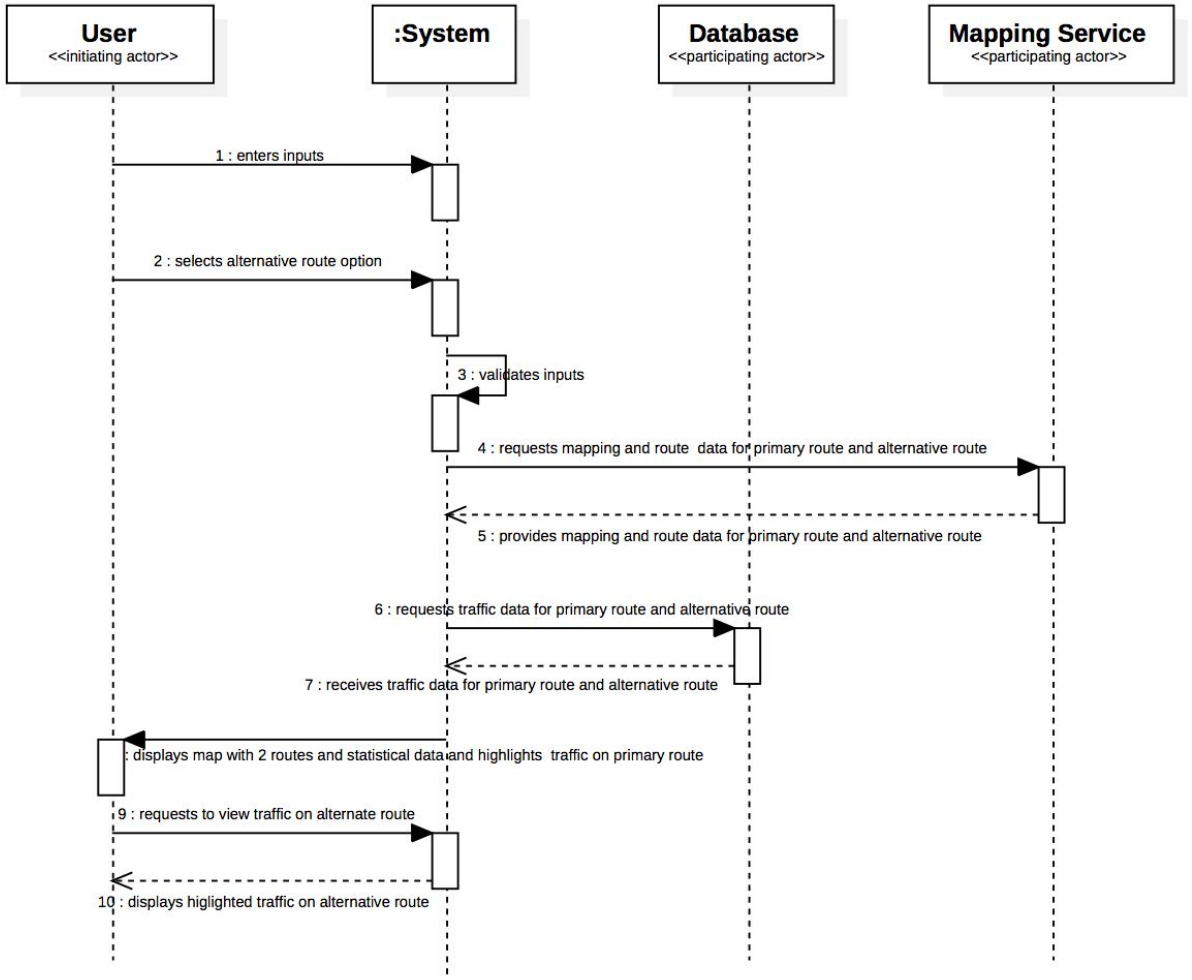

Figure 11: UC-4 Main Success Scenario Sequence Diagram

Figure 11 is the system sequence diagram for the main success scenario of UC-4, where the user is interested in alternative routes. It is an extension of UC-2 and this diagram is initiated after the user has already visited the “Route” page.

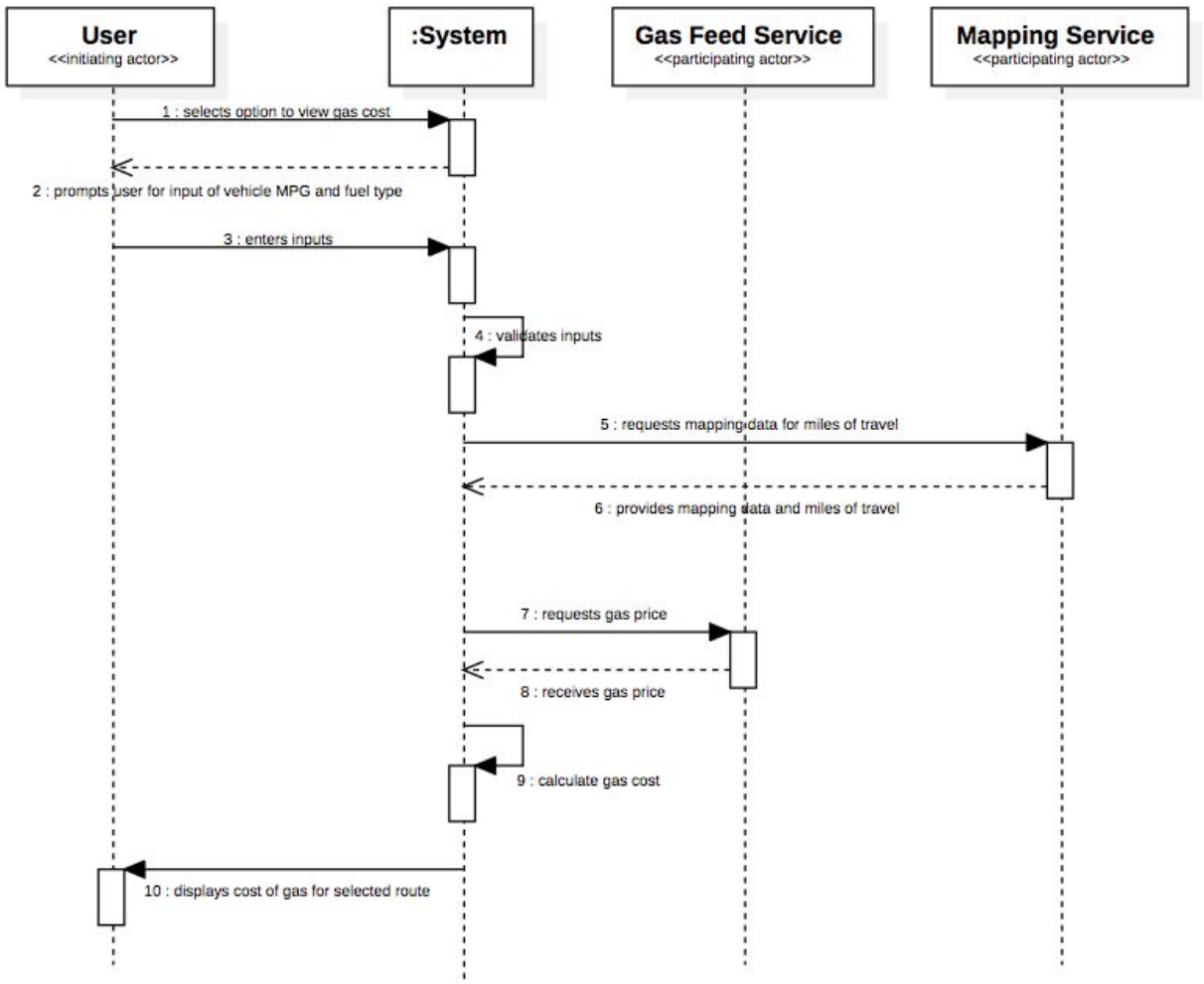

Figure 12: UC-5 Main Success Scenario Sequence Diagram

Figure 12 is the system sequence diagram for the main success scenario of UC-5, where the user is interested in viewing the cost of gas for a selected route. It is an extension of UC-2 and this diagram is initiated after the user has already visited the “Route” page and entered other pertinent inputs for the Route feature. These actions shown will occur concurrently with the other necessary actions needed for UC-2.

Section 5: Effort Estimation Using Use Case Points

This part is calculated based on class slides:
http://www.ece.rutgers.edu/~marsic/books/SE/instructor/slides/lec-14%20Metrics-Intro.ppt

Table 5: Actor Classification And Weights

Actor Name	Description of relevant Characteristics	Complexity	Weight
User	User interacting with system via a graphical user interface	Complex	3
Database	Database is a system interacting through a network communication protocol.	Average	2
Mapping Service	Mapping Service interacts with our system via defined APIs.	Simple	1
Weather Service	Weather Service interacts with our system via a defined API.	Simple	1
Gas Feed Service	Gas Feed Service interacts with our system via a defined API.	Simple	1

UAW = 3 x simple + 1x Average + 1x Complex = 3x1 +1x2 + 1x3 = 8

Table 6: Use Case Weights

Use Case	Description	Category	Weight
ViewHeatMap (UC-1)	Complex user interface. Three actors involved (User, Database, Mapping Service). Greater than 7 steps for main success scenario.	Complex	15
ViewRoute (UC-2)	Complex user interface. Three actors involved (User, Database, Mapping Service). Greater than 7 steps for main success scenario.	Complex	15
CreateAgenda (UC-3)	Complex user interface. Three actors involved (User, Database, Mapping Service). Greater than 7 steps for main success scenario.	Complex	15
ViewAlternateRoute	Average user interface. Three actors	Complex	15

(UC-4)	involved (User, Database, Mapping Service). Greater than 7 steps for main success scenario.		
ViewGasCost (UC-5)	Average user interface. Three actors involved (User, Mapping Service, Gas Feed Service). About 6 steps for main success scenario.	Average	10
ViewForecastInfo (UC-6)	Simple user interface. Three actors involved (User, Weather Service, Database). Less than 6 steps for main success scenario.	Average	10
ViewCurrentRoute (UC-7)	Simple user interface. Two actors involved (User, Mapping Service). Less than 4 steps for main success scenario.	Simple	5

UUCW = 4 x Complex + 2x Average + 1 x Simple = 4x15 + 2x10 + 1x5 = 85

Table 7: Technicality Complexity Factors

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight x Perceived Complexity)
T1	Distributed Web based system, using a database and multiple API's.	2	5	2x5 = 10
T2	Good performance is important for system, but nothing exceptional.	1	3	1x3 =3
T3	End-user efficiency is a very important factor for our system.	1	5	1x5=5
T4	Fairly complex internal processing.	1	4	1x4 =4
T5	Having reusable design and code is important for us to add features on top of current features.	1	5	1x5 = 5
T6	Installation not a factor.	0.5	0	0x0.5 =0

	Simply go to our website.			
T7	Ease of use is crucial.	0.5	5	0.5*5 = 2.5
T8	Great efforts made to make this mobile responsive and portable, but main use is on desktop.	2	4	2x4 = 8
T9	Easy to change is required to an extent.	1	3	1x3 =3
T10	Concurrent use is possible.	1	3	1x3 =3
T11	Security is not a concern. However, invalid inputs are handled.	1	2	1x2 = 2
T12	No direct access for third parties.	1	0	1x0 = 0
T13	No unique training needs.	1	0	1x0 = 0
			Technicality Factor Total:	45.5

Table 8: Environmental Complexity Factors

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor (Weight x Perceived Impact)
E1	Beginner familiarity with UML-based development.	1.5	1	1.5x1 =1.5
E2	Beginner familiarity with application problem.	0.5	1	0.5x1 = 0.5
E3	Some knowledge of object oriented approach.	1	2	1x2 = 2

E4	Novice experienced lead analyst.	0.5	2	0.5x2 = 1
E5	High motivation, but all members with very busy schedules and different priorities.	1	3	1x3 =3
E6	Semi -stable requirements expected.	2	3	2x3 =6
E7	No part-time staff will be involved.	-1	0	-1x0 = 0
E8	Multiple programming languages of average difficulty will be used.	-1	3	-1x3 = -3
			Environmental Factor Total:	11

UUCP = UAW + UUCW = 8 + 85 = 93

TCF = Constant-1 + Constant-2 xTechnical Factor Total; Constant-1 (C1) = 0.6, Constant-2 C(2)= 0.1

TCF = 0.6 + 0.01x45.5 = 1.055

ECF = Constant-1 + Constant-2 xEnviromental Factor Total; Constant-1 (C1) = 1.6, Constant-2 C(2)= -0.03

ECF = 1.4 + (-.03x11) = 1.07

UCP = 93 x 1.055x1.07 = 104.98, or **105 use case points**

Duration = UCP x PF; PF= 28
Duration = 105 use case points x 28hours/use case point = **2940 hours**

Section 6: Domain Analysis

Our domain analysis for the use cases UC-1,UC-2, UC-4, UC-5 and UC-6. We will provide the domain analysis for these only as they are our fully dressed use cases.

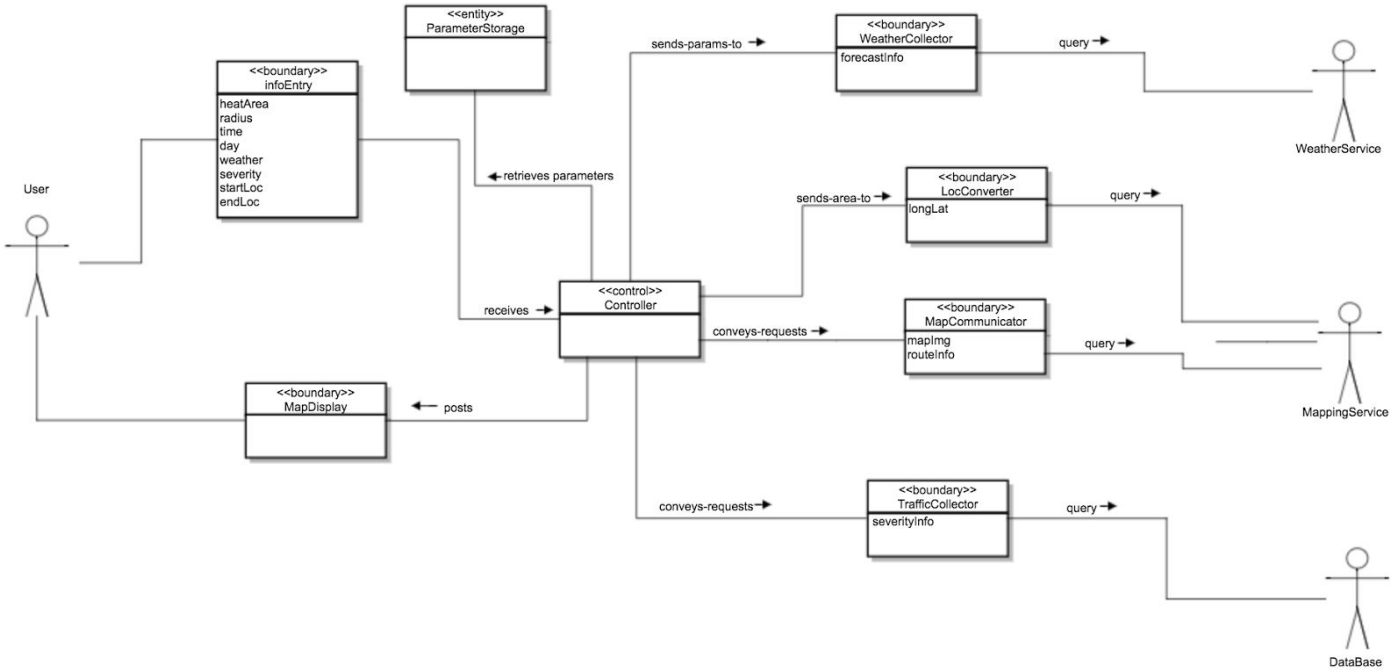

Figure 13: Previous Domain Model for UC-1, UC-2, UC-6

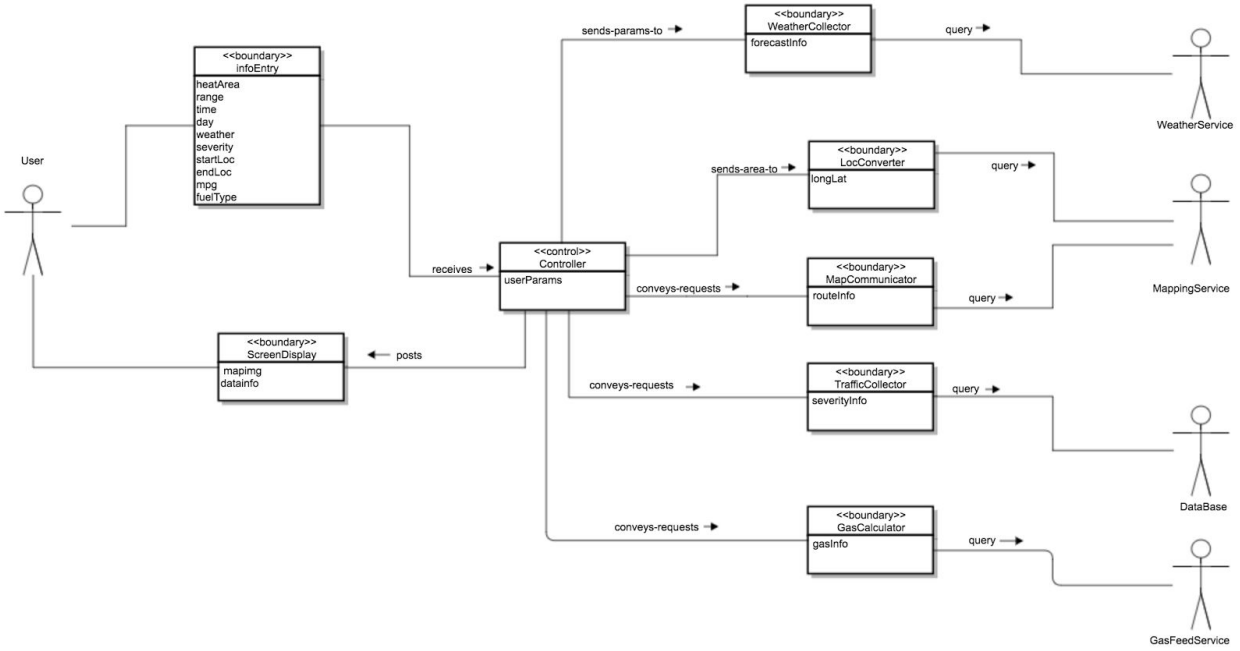

Figure 14: New Domain Model for UC-1, UC-2, UC-4, UC-5 UC-6

Figure -13 shows our original domain model, and Figure-14 is our revised domain model with additions for our added implemented use cases. To construct our domain model, we began by adding the Controller and the InfoEntry. We started here because the user's inputs are the natural starting point of all the use cases. From there, we developed the other concepts as they were needed to fulfill the use cases. The Controller receives these inputs from InfoEntry and, in our original design stores them in the entity ParameterStorage. We later decided that because the user parameters are only used in the exact instant that their request is being serviced, there was no need to create an object in order to store them. Instead, Controller maintains knowledge of user parameters. The Controller then passes the address-formatted locations (zip codes or addresses) to the LocationConverter, which uses the Mapping Service to convert the locations to longitude and latitude pairs. The longitude and latitude pairs are for multiple purposes throughout our system, such as to query our Database via the TrafficCollector conceptual object that gathers traffic incidents in the desired area, and to get weather information from the WeatherCollector for a specific location using this latitude longitude pair. The MapCommunicator conceptual object is used to communicate with the Mapping Service to get route information, and this route information is used to find traffic along the route from the TrafficCollector, which again queries the Database. In our original domain model, we had the concept of MapDisplay, which displays the map with routes and roads highlighted according to severities received in the TrafficCollector. However we found that we require more than just a map to report information to the user, such as graphs and printed statements. Therefore we renamed the concept MapDisplay to ScreenDisplay. This is a far better name and gives it more responsibility. Thus, ScreenDisplay also gained attributes such as mapimg and datainfo. Also, in the new domain model we added the concept of GasCalculator. We needed to create this concept as we need a conceptual object to handle receiving gas price information from a Gas

Feed Service and to calculate the gas cost of the trip for a selected route. As you may have noticed, most of our conceptual objects have remained and not too many had to be added in this second iteration. This is due to forward thinking and planning that caused us to create conceptual objects that would cover most of the responsibilities of our system.

The reader may now see that the domain model is abstract and does not have too many details. We would like to inform the reader that this is the point of the domain model. It is meant to be a simple demonstration of conceptual organization that can be easily and quickly understood. The details that require longer examination will be shown soon in our class diagrams.

Concept Definitions

Table 9: Concept Definitions for UC-1, UC-2, UC-4, UC-5 UC-6

Responsibility	Type	Concept
R1: Coordinate the receiving and sending of info to and from the different concepts	D	Controller
R2: Accept user input for all of the various parameters.	D	Info Entry
R3: Gather the information about gas prices from Gas Feed Service and use it to calculate the gas cost, using user inputted vehicle information.	D	Gas Collector
R4: Take locations in address format and convert them to latitudinal-longitudinal format using a Mapping Service.	D	Location Converter
R5: Take latitudinal-longitudinal format locations and route segments to query the Database for all traffic severities that match the locations/route segments.	D	Traffic Collector
R6: Take a starting and ending location and use those to obtain a route from the Mapping Service.	D	Map Communicator
R7: Take date, time, and location as a longitude pair, and use those to obtain a weather forecast from the Weather Service.	D	Weather Collector
R8: Produce map image and display it to the user with relevant traffic information. Present to them data and statements about weather and cost of gas.	D	Screen Display
R9: Store user inputted parameters as well as system parameters.	K	Controller

By design, we have mostly “Doer” concepts rather than “Knower” concepts. This is a conscientious design decision which is further explained in the Class Diagrams section.

Association Definitions

Table 10: Association Definitions for UC-1, UC-2, UC-4, UC-5 UC-6

Concept Pair	Association Description	Association Name
Info Entry ↔ Controller	User enters data relevant to their needs. The data is sent to the controller.	Conveys Requests
Location Converter ↔ Controller	The controller sends location request from the user to the Location Converter. The Location Converter returns latitude longitude pair(s) based on location request.	Provides Data
Map Communicator ↔ Controller	Controller sends route parameters to Map Communicator. Map Communicator returns detailed route data from the Mapping Service, now parsed, to the Controller.	Provides Data
Traffic Collector ↔ Controller	Controller sends time/date/location information needed from database and Traffic Collector queries the database. Traffic Collector returns the subset of data needed from the database, including severities.	Provides Data
Screen Display ↔ Controller	Controller sends data compiled from the the other concepts and Screen Display uses the data to display a map with the data to the user, as well as other information.	Displays Data
Controller ↔ Gas Collector	The Controller sends information received from user inputs to Gas Collector to calculate gas cost and return to Controller.	Generate
Weather Collector ↔ Controller	If the user chooses to use forecasted weather data, the Controller sends the information needed to the Weather Collector. The Weather Collector returns the weather forecast from the Weather Service.	Provides Data

Attribute Definitions

Table 11: Attribute Definitions for UC-1, UC-2, and UC-6

Responsibility	Attribute	Concept
R10: User inputted location of the area that the heat map will be centered around	heatArea	infoEntry
R10: User inputted range of the area that the heat map will display	range	
R11: User inputted time of day that heat map statistics will be based off of	time	
R12: User inputted day of week (or date) that heat map statistics will be based off of	day	
R13: Weather type that heat map and route data will be based off of (snow, clear, etc.)	weather	
R14: User inputted severity of traffic that should be displayed on the heat map or route	severity	
R15: User's starting location for route feature	startLoc	
R16: User's ending location for route feature	endLoc	
R17: User's vehicle combined mpg information	mpg	
R18: User's vehicle fuel type (Regular, Premium, Midgrade, Premium, Diesel)	fuelType	
R19: Weather forecast for date and time specified by user	forecastInfo	WeatherCollector
R20: Acquired longitude and latitude converted from the inputted zip code	longLat	LocConverter
R21: Map with highlighted routes and traffic information to be displayed to the user	mapImg	ScreenDisplay
R22: Data graphs as well necessary print statements.	datainfo	
R23: Suggested route to get from starting point to destination specified by user	routeInfo	MapCommunicator

R24: Traffic information acquired from the database to be displayed on the map	severityInfo	TrafficCollector
R25: Gas prices received from Gas Feed Service used to calculate the cost of gas	gasinfo	GasCalculator
R26: Parameters used and passed during system execution.	userParams	Controller

Traceability Matrix

	Use Case	UC-1	UC-2	UC-4	UC-5	UC-6
DOMAIN CONCEPTS	**PW**	19	21	4	4	5
	Controller	X	X	X	X	X
	Info Entry	X	X	X	X	X
	Location Converter	X	X	X		
	Traffic Collector	X	X	X		
	Map Communicator	X	X	X		
	Weather Collector					X
	Gas Calculator				X	
	Screen Display	X	X	X		

Figure 15: Traceability Matrix of Domain Model

In Figure 15 we have shown the traceability matrix for our fully dressed use cases.

System Operation Contracts

Table 12: Operation Contract for UC-1

Operation	viewHeatMap
Preconditions	● *heatArea* must be specified by the user. Must be a valid input. ● *range* of interest must be specified by the user. ● *severity* must be specified by the user. ● *time* must be specified by the user. ● *day* of travel must be specified by the user. ● *weather* must be specified by the user. ● Incident data must exist in the database.
Postconditions	● *mapImg* must be displayed for the user. ● *severityInfo* must be displayed on the map on the *heatArea*. ● *severityInfo* analysis must be displayed on a graph.

Table 13: Operation Contract for UC-2

Operation	viewRoute
Preconditions	● *time* must be specified by the user. ● *day* of travel must be specified by the user. ● *weather* must be specified by the user. ● *startLoc* must be specified by the user. ● *endLoc* must be specified by the user. ● *severity* must be specified by the user. ● Route data must exist in the database.
Postconditions	● *routeInfo* must be displayed for the user. ● *severityInfo* must be displayed on the map along the route. ● *severityInfo* analysis must be displayed on a graph.

Table 14: Operation Contract for UC-4

Operation	viewAlternateRoute
Preconditions	● View alternative routes option must be selected by the user. ● Route data must exist in the database for the alternative route.
Postconditions	● *routeInfo* must be displayed for the user for both the primary and alternative route. ● *severityInfo* must be displayed on the map along the route for the selected route. ● *severityInfo* analysis must be displayed on a graph for both routes.

Table 15: Operation Contract for UC-5

Operation	viewGasCost
Preconditions	● View estimated gas cost option must be selected by the user. ● *mpg* must be specified by the user ● *fuelType* must be specified by the user. ● *gasInfo* must be available from the Gas API.
Postconditions	● Estimated gas cost for all routes must be displayed to the user.

Table 16: Operation Contract for UC-6

Operation	viewForecastInfo
Preconditions	● Weather API must have the weather forecast for the next ten days available. ● Weather info must exist in the database.
Postconditions	● *forecastInfo* must be incorporated with the user info to display the proper output.

Mathematical Model

The most pressing question we had to deal with regarding mathematical models was: how do we predict the severity of a road based on historical data? In order to do this, we look at all the incidents recorded along a given road, time, and set of weather conditions. In order to average our results, we take the sum of the severities of these incidents and divide it by the total amount of possible incidents.

Informally, consider the following example: Over 5 months of data collection, it has been 5:00PM on a Tuesday with rain a total of 5 times. Of these 5 rainy Tuesday's at 5:00pm, 3 traffic incidents of severities 4, 2, and 4 were recorded on road A. The average severity equals (4+2+4)/5 = 2. That means that a user will see traffic of severity 2 predicted for them on road A at 5:00PM on a Tuesday with rain. Below is a formalized representation of the algorithm that we will be employing to calculate the average severity for a road at a certain datetime.

$$A_{x,t} = \frac{\sum_{i=0}^{N} \tau(x,t)}{M} \quad (1)$$

Where:

- $A_{x,t}$ is the average severity along a given road *x* at datetime *t* (a specific hour on a specific day of the week).
- N is the total number of traffic incidents for a given road *x* at datetime *t* (a specific hour on a specific day of the week).
- M is the total number of times the cron[6] job has ran since data collection has commenced for a given road *x* at datetime *t* (a specific hour on a specific day of the week).
- $\tau(x, t)$ is a function that fetches the severity for each traffic incident for a given road *x* at datetime *t* (a specific hour on a specific day of the week).

All traffic incidents in our database are assigned a corresponding severity level from 1-4. In equation (1), all the severities of traffic incidents that occurred on road *x*, at a specific time *t* are summed. This total is then divided by the number of times the cron job has ran for that specific scenario (road and datetime) since data collection commenced. This will ensure that the result is scaled with consideration for the null case where no traffic is recorded at a given time.

Section 7: Interaction Diagrams

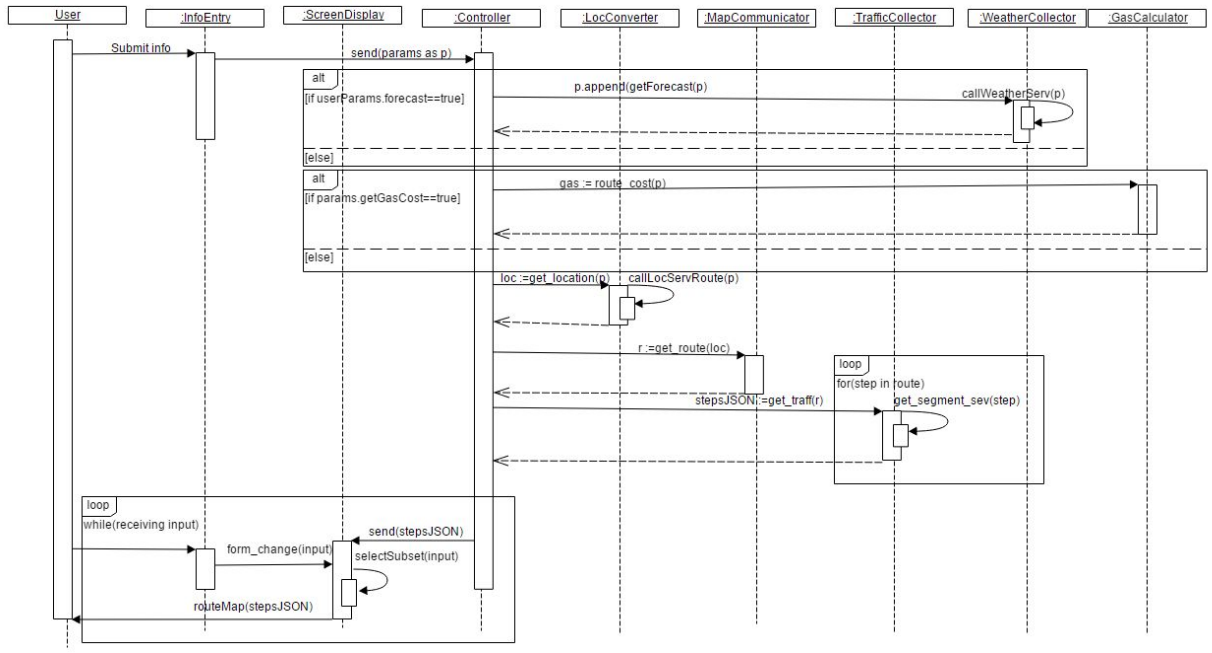

Figure 16: UC-2 “View Route” success sequence diagram, with UC-5 and UC-6 extension

This diagram incorporates both UC-2, the “Route” feature, and its extensions UC-5 “Gas Cost” and UC-6 “Forecast Weather”. This diagram is initiated by the user choosing the “Route” feature on the landing page and beginning to input parameters.

Just a quick note before diving into the interaction diagram’s narrative. InfoEntry and ScreenDisplay are user-facing JavaScript objects, and every other object is implemented as server-side PHP. Because of the paradigm of client-side vs. server-side interaction, the exchange of info between the back-end PHP and front-end JavaScript cannot be very adequately described. “send(params as p)” is an actuality accomplished with a POST request to the server. “send(stepsJSON)” in actuality never occurs, because the PHP variables are globally available to the JavaScript after the page refreshes.

We designed all of our concepts with high cohesion and modularity in mind. Having all communication mediated by the Controller entity allows us to decouple the individual components of our system. This helps keep our system modular, since we can add and remove components without risk of disturbing the functioning of other components. Additionally, each component has a focused, cohesive purpose.

This sequence diagram is directly derived from our domain model.

This version of our design also shows our refactored Publisher-Subscriber design pattern. We decided that rather than hardcoding a response for every time the user interacts with the form entry on the route page, we could turn it into a “form_change(form.id)” alert, which is then sent to all subscribers that care about it. Currently, the only subscriber is ScreenDisplay. Despite the fact that we only have one subscriber, should we develop modules in the future which care about user-interaction, our design will be easily scalable to include them. In addition, use of this design pattern allows all of the knowledge of the associated logic to be abstracted away from infoEntry itself. Though this capability exists for all other features, we decided to not re-include it in the other interaction diagrams in the interest of focusing on content that is different between features.

One last thing to note, although the refactoring seems like a small change, it actually involved changing a substantial amount of logic that originally was hard-coded into controller, and moving it to front-end “form_change” version. This effort was definitely worth it, however, as our code is not only more scalable and maintainable, but it’s also much faster, as user inputs can now be handled dynamically, as opposed to requiring a page-refresh every time the inputs change.

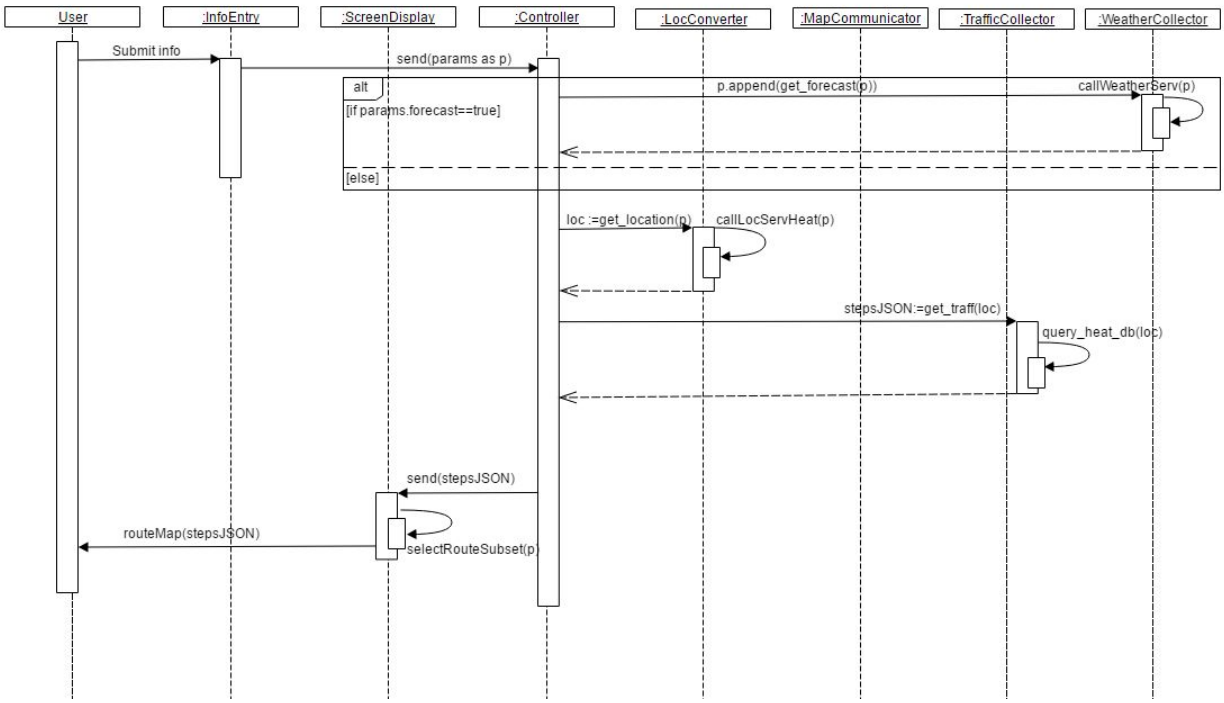

Figure 17: UC-1 “View HeatMap” success sequence diagram, with UC-6 extension

This diagram shows the success scenario for both UC-1, the “Heat Map” feature, and its extension UC-6. UC-6 is the extension in which the user decides to use forecasted weather as opposed to entering it in manually, therefore UC-6 does not have its own sequence diagram. This diagram is initiated by the user choosing the “Heat Map” feature on the landing page and beginning to input parameters.

This diagram is similar to Figure 16 above. The components used are all the same, though the functions called are specific to the heatmap feature. As a result, the design principles mentioned in the description for Figure 16 still apply here.

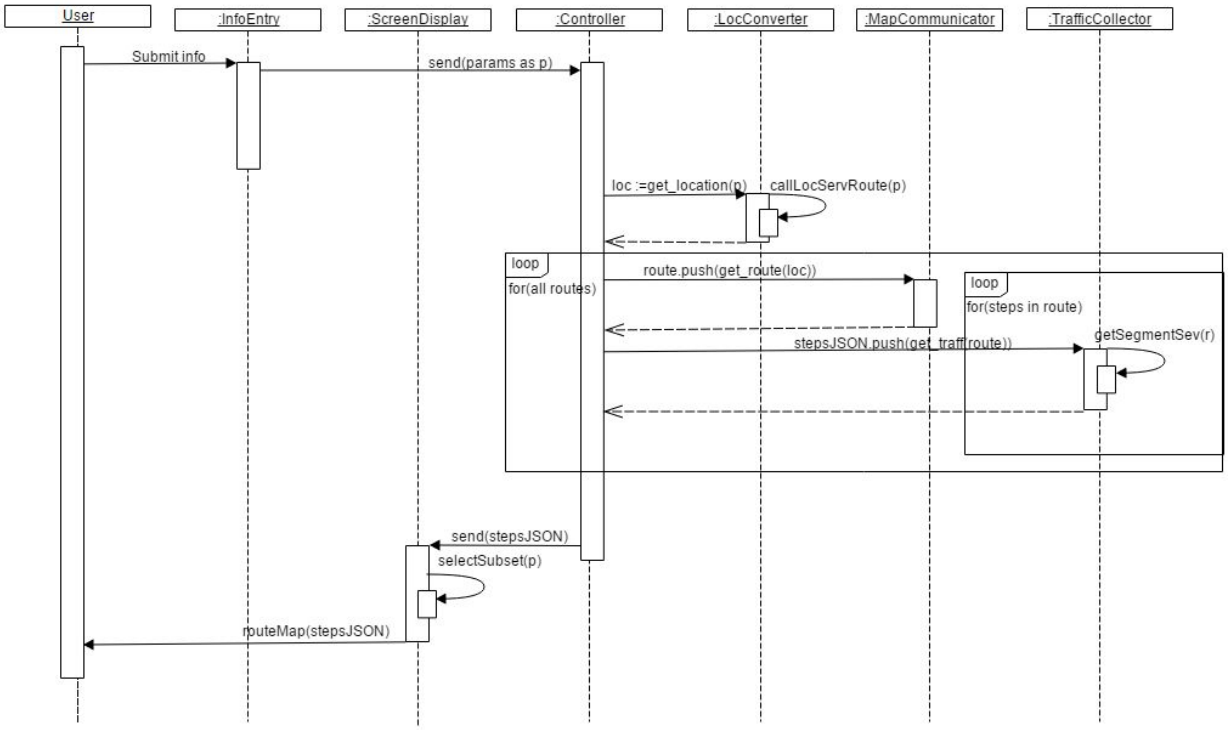

Figure 18: UC-4 “View Alternate Route” shown as extension of UC-2 “View Route”

This diagram shows our UC-4 “View Alternate Route”. In terms of narrative, it doesn’t vary from the sequence diagram of UC-1. The only difference is that rather than returning one route from the Mapping Service, we now return two. We then collect traffic for both of the routes. Only traffic data for one route at a time is displayed, as selected by the user.

Alternate Design

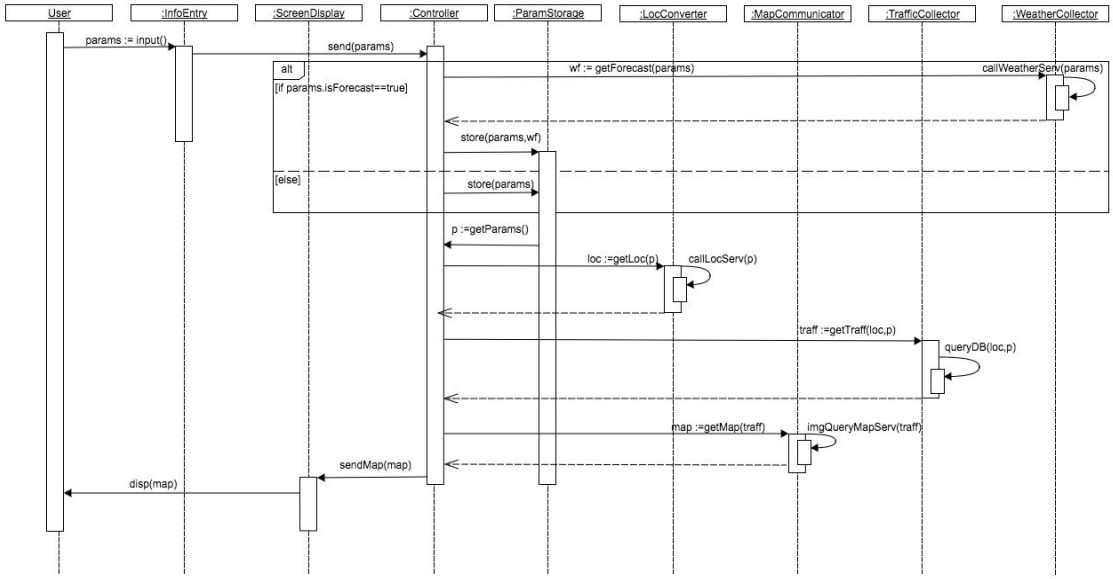

Figure 19: Alternate Design Interaction Diagram

This is a sequence diagram from our original iteration of the software. Notable changes include the lack gas cost feature and the lack of dynamic form resubmission, which were included in later iterations of the design. In addition, a conscientious decision was made to remove ParamStorage from the sequence diagram. This is because there is no need to associate an object to the user's parameters. A user's parameters only need to live in the instant that they are used to service the user's request. Because of this, the controller stores the parameters during its lifetime.

Section 8: Class Diagram and Interface Specification

Class Diagram

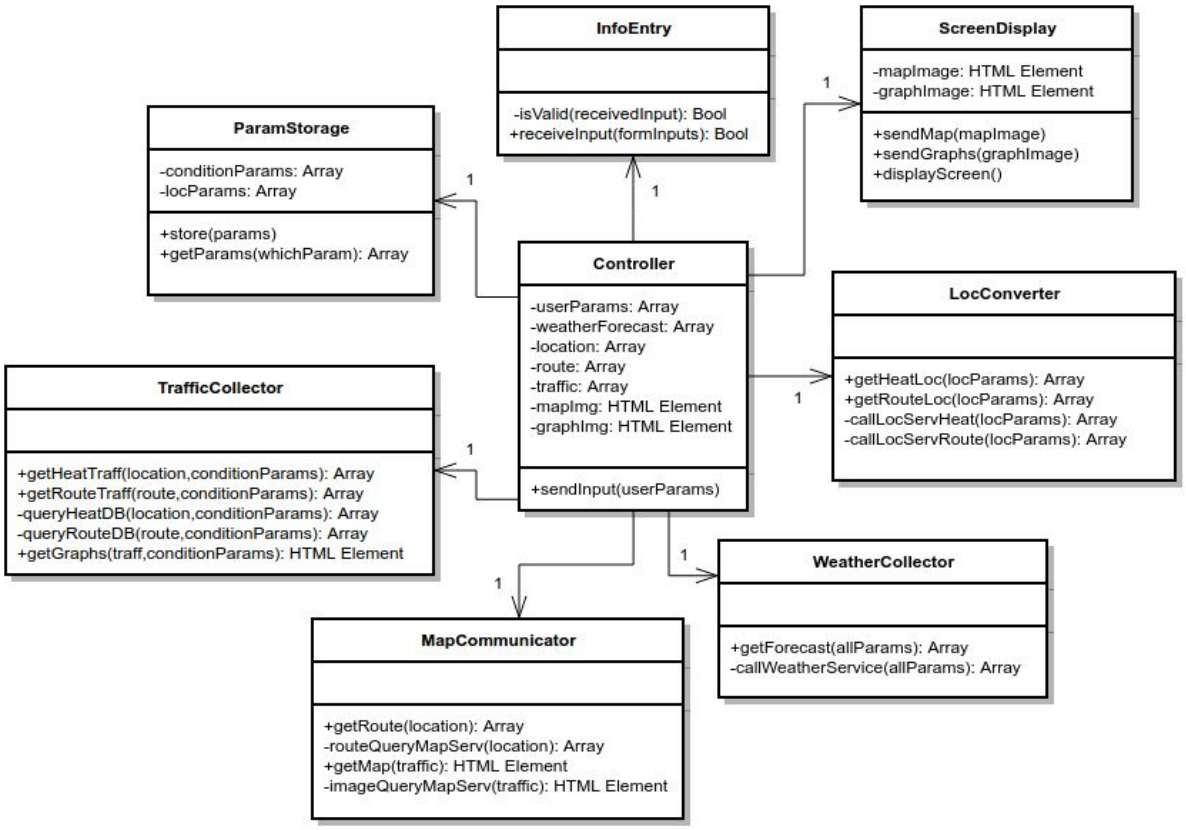

Figure 20: Previous Class Diagram

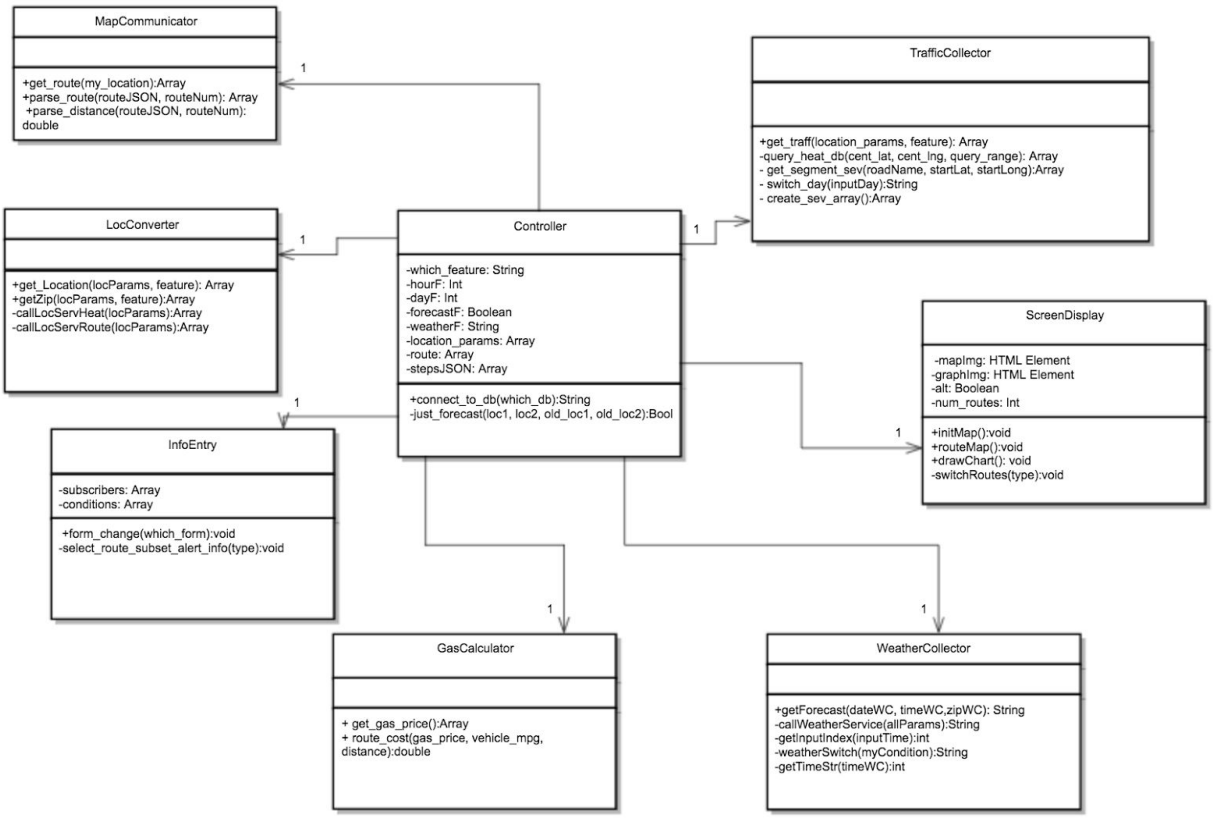

Figure 21: Updated Class Diagram

Figure 20 is the previous Class Diagram from Report 2 and Figure 21 is the newly updated Class Diagram. The updated class diagram reflects changes such as the removal of ParamStorage, the addition of GasCalculator. There are many changes to the attributes and operations of these classes to reflect changes made during actual implementation. Such changes occurred due to the fact that we learned more about the required operations for each class . However, the responsibility of the classes that remained are still the same.

Design Patterns

As you can see, all classes' interactions are mediated entirely by the Controller class. This was done intentionally to make our system modular. Eliminating visibility between the classes makes it easy to add and remove modules as required, which proved useful when adding the Gas Calculator. When a specific class needs information from another class, it will simply communicate that to the controller, which will then request the information from the appropriate class and then return it back to the class that originally asked for it. For example if the MapCommunicator class needs traffic information from the TrafficCollector class, it will receive that information from the Controller. The Controller would use the getter functions of the TrafficCollector to retrieve this information and pass it back to the MapCommunicator class. It may be noticed that there are minimal attributes throughout our classes. This is because our system only requires the parameters in the instant that the user's request is serviced. There is no need for object permanence in this regard. Our original attempts to implement with storing the transient parameters as object attributes proved to be an extra complication on development, with little benefit received in terms of usability or maintainability of the code.

In addition, after the first demo, we refactored our design to include the Publisher-Subscriber design pattern. This was motivated by the fact that our original design required there to be hardcoded responses for any kind of change in input-parameters. Now, if any input parameter is changed, a generic "form_change(id)" alert function is called, which alerts any entities that care about new input parameters. Currently, the only subscriber to this function is screenDisplay. Going forward, however, we can envision that other entities might care about this info as well. This design pattern would allow us to easily scale to that situation.

Data Types and Operation Signatures

Controller: Responsible for mediating interactions between all other objects (keeping them loosely coupled), and limiting inter-object visibility.

Attributes:

- **-which_feature:** String
  - A string which holds the name of the current feature being used
- **-hourF:** Int
  - An int which holds the hour
- **-dayF:** Int
  - An int which represents the day of the week (0 for Sun, 1 for Mon, etc.)
- **-forecastF:** boolean
  - A boolean which represents whether or not the weather needs to be forecasted.
- **-weatherF**: String
  - A string representing a weather condition
- **-location_params**: Array
  - The user entered parameters for addresses or a zip code and range.
- **-route**: Array
  - An array of steps that connect two locations.
- **-stepsJSON**: Array
  - An array of traffic severities in a JSON format.

Operations:

- **+connect_to_db(which_db:String):** String
  - Makes a connection to the specified database
- **-just_forecast(loc1, loc2, old_loc1, old_loc2):** Boolean
  - Compares new location with old location and returns a true if the location has not been changed. This helper function when the user changes the date or time and the weather needs to be re-forecasted

infoEntry: Responsible for alerting all of the ‘subscribers’ whenever the user changes any info. The subscribers are simply the modules which care about the user info.

Attributes:

- **-subscribers:** Array
  - An array of all of the entities which care about a change in info
- **-conditions:** Array
  - An array of condition parameters, for parsing the json to display into the map.

Operations:

- **+form_change(***id: String***)**
  - A function that alerts all subscribers whenever a form is changed. The subscribers will use the information however they see fit.

- **+select_route_subset_alert_info**(*id:String*)
    - A function which parses the JSON that displays onto the map

ScreenDisplay: Responsible for accepting information about what to display from the controller and then using that information to create a map with highlighted traffic data. Additionally, it uses that information to create a graph of traffic severities at different times of the day. It uses Google Maps Javascript API to display the map and Google Charts to display the graph.

Attributes:

- **-mapImg**: HTML Element
    - Google Maps GUI element, with traffic severities highlighted
- **-graphImg**: HTML Element[9]
    - Statistical data about the traffic, represented as a graph.
- **-alt:** bool
    - This is a boolean that tells functions whether or not there are alternate routes present.
- **-num_routes:** int
    - This is an int with the number of routes. Our current implementation only handles up to 2 routes, but this could be extended easily in the future.

Operations:

- **+initMap():void**
    - A function that displays either the default map, or calls route map if info exists to print. The default map is a map centered on New Jersey.
- **+routeMap():void**
    - A function that highlights traffic data on the map by highlighting the road segment which has a traffic severity associated with it.
- **+drawChart():void**
    - A function that uses traffic severity information to create a line graph and displays it for the user. The graph created shows traffic severities for a route at different times of the day.
- **-switchRoutes(type):void**
    - A function that swaps which route (primary or alternate) has its traffic severities highlighted based on the parameter *type*. If type is “prim”, it highlights traffic on the primary route. If type is “alt”, it highlights traffic on the alternate route.

LocConverter: Responsible for taking a user’s input parameters for location, such as a zip code or city name, and converting it to a lat/long format. It uses Google Maps Geocoding API to get back a JSON response that is parsed to get zip codes and latitudes and longitudes.

Operations:

- **+get_location**(locParams: Array, feature: String): Array
    - Getter function, returns lat/long pair(s) in an array.
    - Calls callLocServHeat or callLocServRoute depending on the feauture
- **+getZip**(locParams: Array, feature: String): Array

    - Getter function that returns zip code based on user inputs. Mainly used for returning the zip code of starting location to be used for WeatherCollector.
- **-callLocServHeat**(locParam*s: Array*): Array
    - Helper function for get_location(). Takes the user-inputted zip code, and uses it to make an API call to the Google Maps Geocoding API. Returns a lat/long pair in an array.
- **-callLocServRoute**(locParams*: Array*): Array
    - Helper function for get_location(). Takes the user-inputted start and end coordinates, and uses them to make an API call to the Google Maps Geocoding API. Returns lat/long formatted coordinates in an array.

MapCommunicator: Responsible for acquiring routes information from the Mapping Service, i.e the Google Maps Directions API.

Operations:

- **+get_route**(my_*location: Array*): JSON object
    - Getter function, takes an array of 3, with starting and ending location and a flag of whether the user wants an alternative route, returns JSON response with route information.
- **+parse_route**(routeJSON: JSON object, routeNum: int): Array
    - Helper function that uses the JSON to parse the legs for the directions of a route and returns an array of necessary information on each leg of the route. This information includes starting and ending coordinates, road names, and severities for each leg. The routeNum is to flag whether the primary or alternate route is being parsed.
- **+parse_distance**(routeJSON: JSON object, routeNum: int): double
    - Function that uses the JSON, routeJSON, to parse the distance from the route to be used in the GasCalculator.
    - The routeNum is to flag whether the primary or alternate route is being parsed.

GasCalculator: Responsible for acquiring the cost of gas from the Gas Feed Service, Fuel Economy Web Services API, and calculating the cost of gas for route(s).

Operations:

- **+get_gas_price**(): Array
    - Getter function that calls the Fuel Economy Web Services API for the current gas prices and gets back an XML response. The XML response is parsed and then an array of 4 gas prices is returned, corresponding to Regular, Midgrade, Premium, and Diesel gas prices.
- **+route_cost**(gas_price, vehicle_mpg, distance): double
    - Function that takes in the gas price, that is based on returned array of gas prices and the user inputted fuel type, the vehicle combined mpg, and the

distance that was returned from Map Communicator. Using these parameters, the cost of gas is calculated for the route and is returned.

WeatherCollector: Responsible for acquiring the weather forecast from the Weather Service, the Wunderground API using hourly 10 day forecast. It receives a JSON response and parses it get a forecast.

Operations:

- **+getForecast**(*dateWC: String, timeWC: time, zipWC: int*): String
  - Getter function, takes in date, time and zip code as parameters, and calls callWeatherService. Returns a String with the forecast as either “Clear”, “Snow”, “Rain”, “Cloudy”, or “Fog”.
- **-callWeatherService**(*allParams: Array*): String
  - Helper function used by getForecast(). Takes an array of parameters that contains the date, time and zip from getForecast(). Makes an API call to the Wunderground hourly 10 day forecast API using the zip code (a 240 element array [10 days x 24 hours] is returned), and gets the weather condition for the exact day and hour and location by getting the index of this array by calling getInputIndex. Calls the weatherSwitch function to cast the condition to either r “Clear”, “Snow”, “Rain”, “Cloudy”, or “Fog” and returns one of these as a string.
- **-getInputIndex**(inputTime: String): int
  - Helper function called by callWeatherService that returns the index corresponding to the array in the JSON object which is the difference in time, and days from current system time to user input time.
- **-weatherSwitch(myCondition: String) : String**
  - Helper function that uses a switch case to identify the forecasted weather conditions into a registered condition in our system and returns this as one of the 5 conditions.
- **-getTimeStr(timeWC: time) : int**
  - Helper function that contains a switch statement that allows for form input entry time to be converted to an integer 0-23 and returns this integer.

TrafficCollector: Responsible for accessing our database to get traffic severities associated with locations or routes.

Operations:

- **+get_traff**(*location_params: Array, feature:String*): Array
  - Getter function, takes in a location parameters and input conditions and the feature being used as input, returns an array of average traffic severities of roads inside that bounding box. Calls query_heat_db() and get_segment_sev() depending on the feature parameter.
- **-query_heat_db**(*cent_lat:double, centlLng: double, query_range:double, input_conditions:String*): Array

    - Takes a center latitude and center longitude pair for the center of the zip code (originally in location_params) and input conditions that contain day of week, time and weather condition. Queries the Database for all traffic severities that match those params, returns those severities as an array.
- **-get_segment_sev**(*roadName: String, startLat: double, startLong: double, inputConditions:Array*): Array
    - Takes the roadName, starting latitude and longitude (originally in location_params) and the input conditions and queries the DB for all traffic severities that match those params, returns the average severities.
- **-create_sev_array**(): Array
    - Creates an Array of day, weather, and time options to use as a template for the other functions to use.
- **-switch_day**(inputDay: int):String
    - Helper function that uses a switch case change the day of the week from a number to a String day of the week and returns the string as a day of the week.
- **-switch_weather**(inputWeather: String):String
    - Helper function that uses a switch case to identify the average weather conditions and change them into a registered condition in our system and returns this as one of the 5 conditions.

Traceability Matrix

Classes:	Controller	InfoEntry	GasCalculator	LocConverter	TrafficCollector	MapComm	WeatherCollector	ScreenDisplay
Domain Concepts								
Controller	x							
Info Entry		x						
Gas Calculator			x					
Location Converter				x				
Traffic Collector					x			
Map Communicator						x		
Weather Collector							x	
Screen Display								x

Figure 22: Class Diagram Traceability Matrix

As you can see from the traceability matrix, there is a one-to-one relationship between the classes and domain concepts. For each domain concept we had, we created one class. We constructed our domain concepts with object-oriented design principles in mind. Each concept's role was cohesive and focused. As a result, when we designed our classes, there was little that needed to be changed. This is also a byproduct of the fact that our system contains relatively simple interactions, which do not necessitate complicated class design. A difference from domain model that can be made is that "attributes" in domain model do not carry directly to the class diagram. The "attributes" in domain model are there to illustrate what that concept is responsible for, such as "severityinfo" and "routeinfo", and responsible for returning to the Controller. The class diagram reflects the attributes of the actual implementation and follows the description given in the "Design Patterns" section above.

Object Constraint Language

LocConverter

getlocation:

Context loc_converter :: get_location(location_parameters:Array, feature:String)
inv: feature != NULL
Context loc_converter:: get_location(location_parameters: Array, feature:String)
pre: location_parameters != NULL
Context loc_converter:: get_location(location_parameters:Array, feature:String)
post: self.latLongLC != NULL

getZip:

loc_converter::getZip(location_parameters(location_parameters:Array, feature:String) inv:
self.location_parameters!= NULL
Context loc_converter::getZip(location_parameters:Array, feature:String)
pre: self.getZip() != false
Context loc_converter::getZip(location_parameters: Array, feature:String)
post: self.location_parameters or self.startZip != NULL

callLocServHeat:

Context loc_converter inv::callLocServHeat(location_paramters: Array)
self.zipCodeLC != NULL
Context loc_converter::callLocServHeat(location_parameters: Array)
pre: callLocServHeat() != false
Context loc_converter::callLocServHeat(location_para: Array)
post: self.heatArrayLC != NULL

callLocServRoute:

Context loc_converter::callLocServRoute(location_parameters: Array)
inv: location_parameters != NULL
Context loc_converter::callLocServRoute(location_parameters: Array)
pre: callLocServRoute() != false
Context loc_converter::callLocServRoute(location_parameters: Array)
Post: self.routeArrayLC != NULL

MapCommunicator

getRoute:

Context MapCommunicator::getRoute(my_location:Array)
inv: self.route_start != NULL or self.route_end !=NULL
Context MapCommunicator::getRoute(my_location:Array)
pre: self.loc_api_key != NULL
Context MapCommunicator:: getRoute(my_location:Array)
post: self.getRoute() != false

GasCalculator

get_gas_price:

Context gas_calculator::get_gas_price()
inv:self.XMLGas != NULL
Context gas_calculator::get_gas_price()
pre: self.gas_prices != NULL
Context gas_calculator::get_gas_price()
Post: self.gas_prices != NULL

WeatherCollector

callWeatherService:

Context WeatherCollector :: callWeatherService(allParameters: Array) inv: allParameters!=NULL
Context WeatherCollector :: callWeatherService(allParameters: Array) pre: self.callWeatherService()!=false
Context WeatherCollector :: callWeatherService(allParameters: Array) post: self.myCondition!=false

getInputIndex:

Context WeatherCollector :: getInputIndex(inputTime: String) inv: inputTime!=NULL
Context WeatherCollector :: getInputIndex(inputTime: String) pre: self.getInputIndex()->!=NULL
Context WeatherCollector :: getInputIndex(inputTime: String) post: self. input_Index!=NULL

getForecast:

Context WeatherCollector :: getForecast(date: String, time: time, zip: int) inv: date!=NULL and zip!=NULL
Context WeatherCollector :: getForecast(date: String, time: time, zip: int) pre: time!=NULL
Context WeatherCollector :: getForecast(date: String, time: time, zip: int) post: self. getForecast()!=false

TrafficCollector

get_traff:

Context TrafficCollector :: get_traff(locationParameters:Array, feature:String) inv: feature!=NULL
Context TrafficCollector :: get_traff(locationParameters:Array, feature:String) pre: ocationParameters!=NULL
Context TrafficCollector :: get_traff(locationParameters:Array, feature:String) post: self. query_heat_db()!=false or self. get_segment_sev()!=false

query_heat_db:

Context TrafficCollector :: query_heat_db(centerLatitude:double, centerLongitude:double, query_range:double) inv: query_range!=NULL
Context TrafficCollector :: query_heat_db (centerLatitude:double, centerLongitude:double, query_range:double) pre: self.query_heat_db()!=false
Context TrafficCollector :: query_heat_db (centerLatitude:double, centerLongitude:double, query_range:double) post: self.steps_array!=NULL

get_segment

Context TrafficCollector :: get_segment_sev(roadName:String, startLatitude:double, startLongitude:double) inv: startLatitude!=NULL and startLongitude!=NULL
Context TrafficCollector :: get_segment_sev(roadName:String, startLatitude:double, startLongitude:double) pre: self.get_segment_sev()!=false
Context TrafficCollector :: get_segment_sev(roadName:String, startLatitude:double, startLongitude:double) post: self.sev_array!=NULL

Controller

connect_to_db:

Context controller::connect_to_db(which_db:String)
inv:self. != NULL
Context controller::connect_to_db(which_db: String)
pre: self.mysqli_connect_errno() != true
Context controller::connect_to_db(which_db:String)
post: self.conn != NULL

Section 9: System Architecture and System Design

Architectural Style

Onward is a web-based application, and the architectural style that most aligns with our system is a three tier architecture. Onward consists of a user interface presentation tier, functional process logic with business rules tier, and a data tier. The three tier architecture is appropriate for our system because the user interface, functional logic, and data are all independent modules of our system. The user interface, which is our presentation tier, can be visually modified in any way without altering the functional process logic and database. This allows the tier to have a certain level of independence from the other tiers. In addition, we have a logic tier that consists of entities that handle the application's logic and information processing. Furthermore, we also have persistent data storage that is explored further in the "Persistent Data Storage" section below. The data tier consists of any data that the logic tier may need for processing. The logical tier interacts with the user interface and the data tier, but the user interface and data have no direct interactions, allowing for modularity and effective decoupling of each tier. The architectural style and the subsystems are further explained in the section below.

Identifying Subsystems

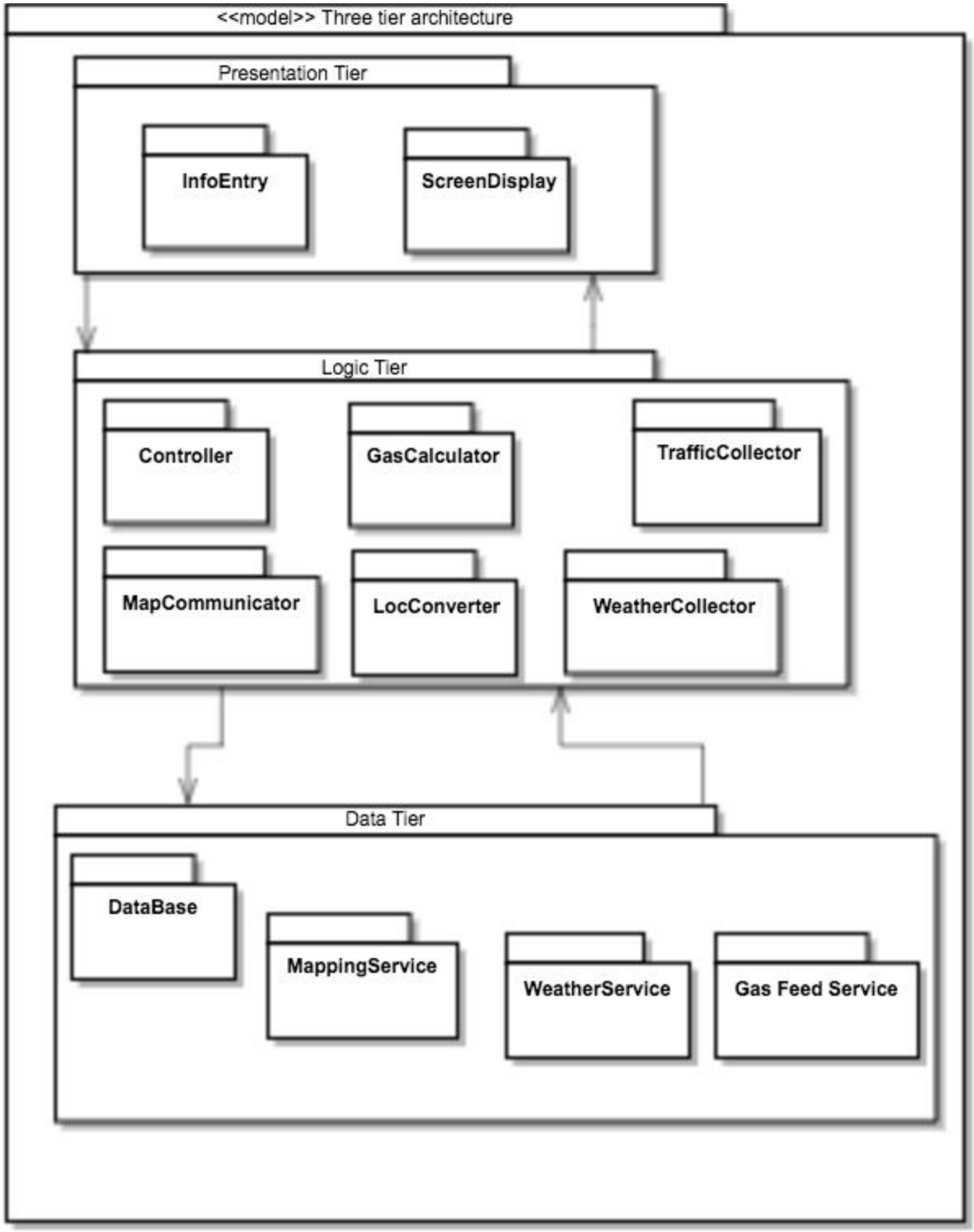

Figure 23: Package Diagram

The subsystems for Onward can be broken into the three different tiers of the system architecture: Presentation, Logic, and Data, as can be seen in the above figure. These are our subsystems because each tier is a large enough subsystem to contain its own set of packages that are independent of each other. The Presentation Tier consists of the InfoEntry and ScreenDisplay concepts from our domain model as these concepts are responsible for the user interface of our system. The Logic Tier consists of these concepts from our domain model: Controller, ParamStorage, LocConverter, MapCommunicator, WeatherCollector, GasCalculator and TrafficCollector. These domain concepts handle the application's logic and information processing. Finally, in the Data Tier subsystem there is the Database, and the services we use - that are based on the participating actors, where the Database is the participating actor itself and the services included are Mapping Service, Weather Service and Gas Feed Service. As can be seen from our interaction diagram, the concepts in the logical tier query both the database, using SQL, and the data services, and then process this data. After processing the data, the logical tier sends the results to be presented in the user interface. All of these packages are unique to each subsystem; however, the different tiers themselves are communicate with each other in a limited way, as can be seen by the various arrows pointing between the different tiers.

Mapping Subsystem to Hardware

The above subsystems are all not housed in the same hardware. The presentation tier is client-side, and involves either the user's computer or mobile device and their respective web browser. The logic tier houses the controller (among other entities) and maps the input from the presentation to the data tier. This tier and all of its modules are located on a remote server. The data tier works to retrieve the necessary information and supplies it back to the logic tier. This tier, which includes the database of collected data for our application, is located on the same remote server. The server we are using to house the data and logic tier is a webhosting service, 1and1.com.

Persistent Data Storage

Below are the relational database tables that Onward uses to perform the various tasks needed to collect traffic data, calculate average road/area severity, and collect weather frequency information. Our database is a crucial part of our system and has been designed with great detail and thought. Along with a graphic of each table is an accompanied description. The algorithms for the database are further described in the "Algorithms" portion of this report. We were unable to retrieve the schema using the "description" command but we have attached screenshots of the schema.

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	incidentId	bigint(20)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	startLat	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	endLat	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	startLong	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	endLong	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	zipCode	int(5)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	description	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	startTime	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	endTime	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
11	severity	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
12	type	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
13	roadClosed	tinyint(1)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
14	lastModified	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
15	weather	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
16	temp	float(3,1)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
17	precip	float(3,2)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 24: Traffic Incident Table

Above is the table used to collect the traffic incident data. The Bing Traffic API is used to collect information about the specific traffic incidents that occur by specifying a "bounding box" of coordinates that represents the region over which we have chosen to collect traffic data. On the top of every hour, our server runs a cron job[6] that calls the Bing Traffic API, parses the JSON result, and stores the relevant data in this table. It is also necessary for Onward to have the specific weather condition associated with the incident in order to accurately present to users future traffic under certain weather. In regards to this, we took the starting latitude and longitude

collected from the Bing Traffic API for each incident, used a geocoding API to convert it to a zip code, and then used that zip code to call the Weather Underground API to get the corresponding weather condition, temperature and precipitation level. Additionally, in this table, certain traffic incidents can be modified by Bing at any time, so each cron job run checks to see if the last modified date of each incident has changed, and if it has, updates the entire entry (neglecting weather condition). We keep all this data for each entry, even if it may be extraneous, so that as we move forward in development and we feel we need the data it is there.

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	mediumint(9)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	gridId	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	zipRegion	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	roadName	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	day	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	hour	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	sev_clear	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	sev_snow	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	sev_cloudy	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	sev_rain	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
11	sev_fog	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
12	avg_clear	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
13	avg_snow	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
14	avg_cloudy	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
15	avg_rain	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
16	avg_fog	float			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 25: Severity Table

The severity table above is responsible for storing the sum and average of traffic severities for each road in a given Grid-Box (explained below) given a certain weather condition, hour, and day of the week. The purpose of having this information is to be able to calculate (including the null case previously mentioned in the mathematical model) the average traffic severity for a specific road under any weather condition, time of day, and day of the week. Additionally, each entry in this table is also assigned a gridId (explained in the next table), which specifies which segment of the road is being depicted.

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	latN	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	latS	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	longE	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	longW	float(10,6)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	zipCode	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	latcenter	float(10,6)			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	lngcenter	float(10,6)			Yes	NULL		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 26: Grid-box Table

An interesting problem that arose as this table was forming was the fact that assigning a general severity to a whole road wouldn't accurately be able to depict where on the road traffic specifically occurs. To solve this problem, we decided to divide our entire "bounding box" into a

grid of grid-boxes. Each grid-box is 0.085° in length and width, which corresponds to approximately 5x5 miles per box. We chose these dimensions in order for our grid-boxes to be granular enough to display precise data for specific segments of roads, but also to be large enough to on average cover the entire length of a single incident. Therefore, for each road in the severity table that has had a traffic incident occur on it, a grid-box is assigned based on the specific location of the incident (the incident that occurred is contained within the assigned grid-box). Each grid-box is assigned a zip code by using the latitude and longitude of the center of the box, then using a geocoding API to find the zip code of that coordinate point. By doing this, we now have the ability to sum total severities for a specific segment of a specific road at a certain day and time.

#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(11)			No	None	AUTO_INCREMENT	Change Drop Primary Unique Index Spatial Fulltext Distinct values
2	freq_clear	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	freq_snow	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	freq_cloudy	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	freq_rain	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	freq_fog	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	zipRegion	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	hour	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	day	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	observedAt	datetime			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 27: Frequency Table

The above frequency table is responsible for storing the frequency of all weather conditions for each grid-box for every hour, each day of the week. In order to query the Weather Underground API to get weather information, a zip code is needed. To make sure that the API query limit was not hit, we developed a strategy to efficiently collect weather. Based on an assumption that most neighboring zip codes will have similar weather, we decided to only collect weather once for a region of zip codes. For example, instead of checking the weather for 07853, 07854, and 07867, we simply collect the weather for 07853, then generalize that weather to all zip codes starting with ‘078’. This lets us collect weather data for a much larger range without exhausting our API key. Ultimately, for each hour of each day of the week, a running total of the weather conditions are stored in this table. By collecting data each hour instead of only when an incident occurs, we are able to include the null case in our average severity calculation.

1	id	mediumint(9)		No	None	AUTO_INCREMENT
2	start_lat	float		Yes	NULL	
3	start_lng	float		Yes	NULL	
4	end_lat	float		Yes	NULL	
5	end_lng	float		Yes	NULL	
6	grid_id	int(5)		Yes	NULL	
7	road_name	varchar(25)	latin1_general_ci	Yes	NULL	
8	pt_count	int(3)		Yes	NULL	

Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 28: Bounds Table

Although we have segmented roads, as seen in the severity table, we still require the starting and ending latitude longitude pairs to place on the map. We place these 4 parameters in the “bounds” table, along with grid_id for the grid this segment is, with the road name as well as the pt_count, which keeps track of whether we have both the starting and ending pairs for this road segment. If We have only the starting, pt_count is 1 and end_lat and end_lng are NULL, and if pt_count is 2 that means we have all the necessary information for this road segment.

Network Protocol

A HTTP request will be used to access “Google Maps” for the map images displayed on the website. The HTTP request was suggested because it is the most commonly used protocol and it is a standard protocol used in any browser space. We also require multiple API calls throughout our system’s functions, that also require HTTP requests. We contact the Weather Underground API for weather information, the Google Maps API for route information and displaying the map on our web page and the Bing Maps Traffic API for traffic information, as well as the Fuel Economy Web Services API to get gas prices to calculate the gas cost. From these API’s, the responses are in JSON or XML that we parse in our php files to extract the pertinent data.

In addition, HTTP POST requests are used to communicate information from the user-end JavaScript forms to the back-end PHP scripts.

Global Control Flow

Execution order:

Our system is mostly procedure-driven. The navigation of the website as well as inputting information are the only ‘asynchronous’ events which occur. The system is mostly procedure-driven because each object does its job in a chronological order. That is to say that each object’s job begins after the previous object’s job ends. For example, once the TrafficCollector finishes querying the DB, the controller signals to the MapCommunicator to collect the route information.

Time Dependency:

This traffic monitoring system will have parts that will operate in real time, and those that are not time sensitive. Many pieces of data are collected in real-time from API calls. Other pieces of information are calculated and stored ahead of time, so they are not time sensitive. The map will not dynamically update if the user changes the parameters. Instead, they will have to click “View HeatMap/Route” again, and the new image will be calculated with the new parameters.

It should be noted that there will be a small time delay between when the user request is entered and when the final processed images are displayed. We do, however, expect to return our map to the user within 2 minutes. The 2 minutes is an upperbound, not a target time.

Concurrency:

The system is not concurrent, and does not use multiple threads in the user interface or the data collection systems. We assume single threads for each system.

Hardware Requirements

Client:
-We recommend user to have any of the following:

- The current version of Microsoft Edge (Windows)
- Internet Explorer 10 and 11 (Windows)
- The current and previous version of Firefox (Windows, macOS, Linux)
- The current and previous version of Chrome (Windows, macOS, Linux)
- The current and previous version of Safari (macOS)

These are the web browsers supported by Google Maps, which is what will be used for the map images on the platform.

- Screen Resolution of at least 1024x768 is required
- Internet connection with minimum bandwidth of 56Kbps
- Javascript functionality must be enabled in the web browser

Server:

- Internet connection with minimum bandwidth of 56Kbps per concurrent client.
- 2 GB of free disc space per month for storing historical data
- The server must provide the following services: PHP, MYSQL, and Apache HTTP server.

Section 10: Algorithms and Data Structures

Algorithms

The algorithms for our database are very important to our project, and we explain the algorithms in great detail below. We believe that it should be distributed more than the 4 points currently allotted to this section.

There were many algorithms that needed be to implemented in order to realize the mathematical models previously described for our database collection. Recall that the mathematical model (equation 1) was designed to determine the average traffic severity of each road segment for a given time, day of week, and weather condition. Below is the mathematical model previously described:

$$A_{x,t} = \frac{\sum_{i=0}^{N} \tau(x,t)}{M} \qquad (1)$$

In this model, the total sum of severities for each road segment at a given time, day of week, and weather condition, was divided by M, the total frequency for the set of time, day, and weather conditions. For example, if the database shows 5 instances of snowy weather on a Tuesday at 5:00pm, and the total sum of traffic severities along a given stretch of US-206 that matches those conditions is 12, then the average severity is 12/5 = 2.4.

There were three different algorithms that needed to be implemented in order to realize this equation. (A) First, an algorithm was needed to collect the traffic data. (B) Second, an algorithm was needed to record the frequency of each weather condition for all the zip codes in our bounding box every hour. And finally, (C) there had to be an algorithm that sums the severities of each road segment and calculates the average severity for said road segment for each weather condition. Below are activity diagrams for A, B, and C, that will be used to describe each of these algorithms.

A. Traffic Collection:

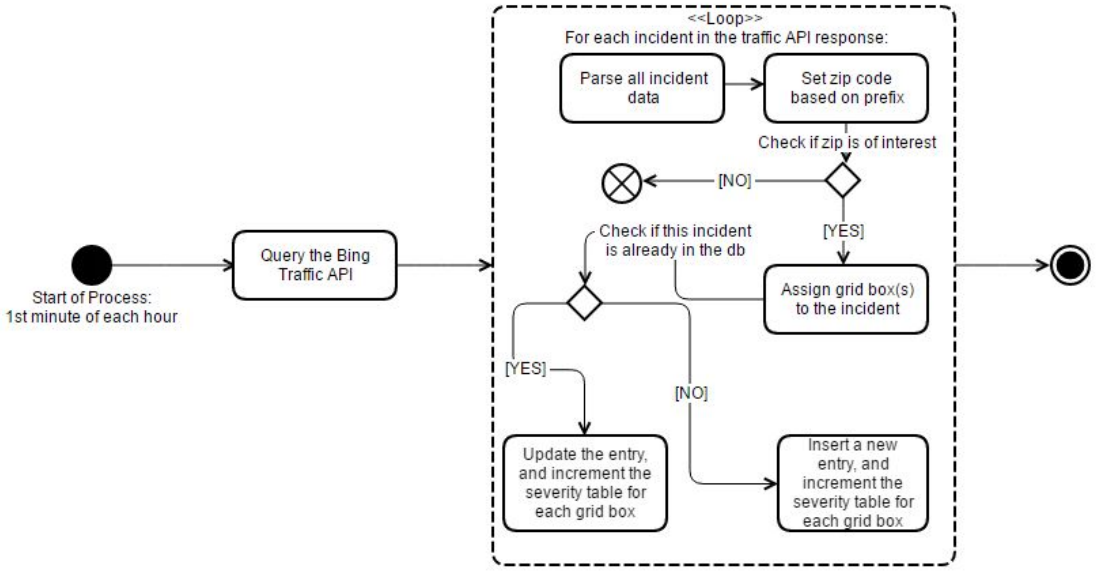

Figure 29: Traffic Collection Algorithm

In this algorithm, each traffic incident from the Bing Traffic API is parsed for its data. Then, if the zip code is within our service coverage area, we check and see if it's in the traffic collection database already. If it is, the incident info gets updated. If not, it gets inserted into the database. Additionally, for every hour the incident lasts, the severity of traffic for the specific road segment (contained in each grid box) is incremented for the current hour and day. This information is also used to insert or update the the starting and ending latitude/longitude pairs of road segments in certain grids.

B. *Weather Frequency Collection:*

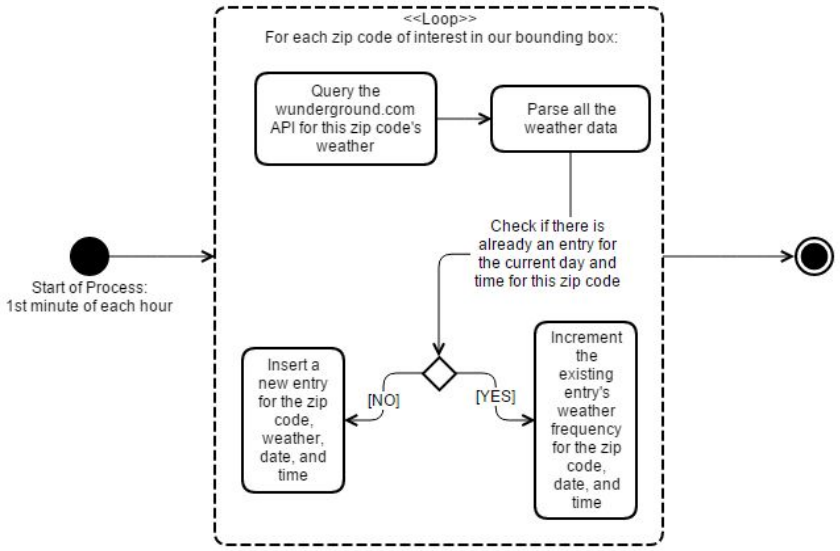

Figure 30: Weather Collection Algorithm

In this algorithm, the weather of each zip code of interest for our bounding box is collected from the wunderground.com API. The weather data for each zip code is parsed and inserted into the weather frequency table if there currently does not exist an entry for the current day, hour, and zip code, or updates by incrementing the weather frequency if there is already an entry that matches this criteria. The wunderground API provides a multitude of weather conditions, but to simplify the number of conditions in our database we add a switch case that assigns any weather condition either to “Clear”, “Snow”, Cloudy”, “Rain” or “Fog”. This also benefits the user since these 5 conditions are the conditions the user can choose from, so that they are not overwhelmed with multiple overly similar conditions.

C. *Total Severity Calculation:*

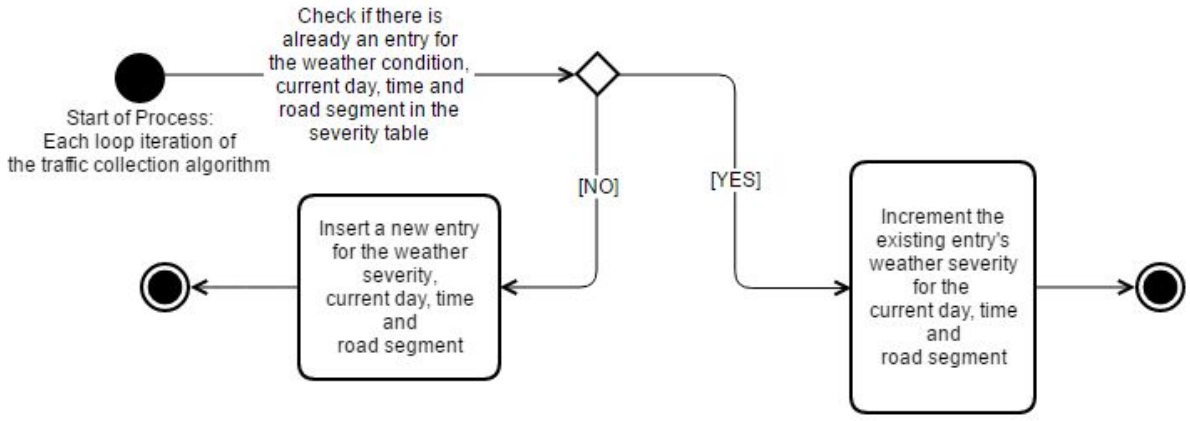

Figure 31: Total Severity Calculation Algorithm

For each incident in the traffic API response, this algorithm is run to increment the total severity for the road segment of the incident for the day and hour if there already exists an entry in the table, or inserts one.

Finally, after the severities and weather frequencies have all been updated/inserted for each road segment at the specific hour and day, the averages for each can be calculated simply using the equation (1). Moreover, in order to realize the persistent data storage described in the *Persistent Data Storage* section of this report, a cron job was enabled on our server that calls all these algorithms on the top of every hour to ensure that data is collected continuously.

In this section we explained the algorithms associated with our Data Tier. The Logic Tier and Presentation Tier do not have complex algorithms and therefore there is no description of algorithms associated with those tiers.

Data Structures

Our system does not rely on complex data structures. Most of our data needs are handled by our relational databases. Though many objects interface with API's, all XML or JSON responses are parsed into arrays before being returned, so as to make sure that formatting is consistent. This also helps abstract away the API interfacing from the logic tier.

Section 11: User Interface Design and Implementation

Mockups

We first would like to show you the original mockups we had for user interfaces before they were implemented to show the evolution of our user interface.

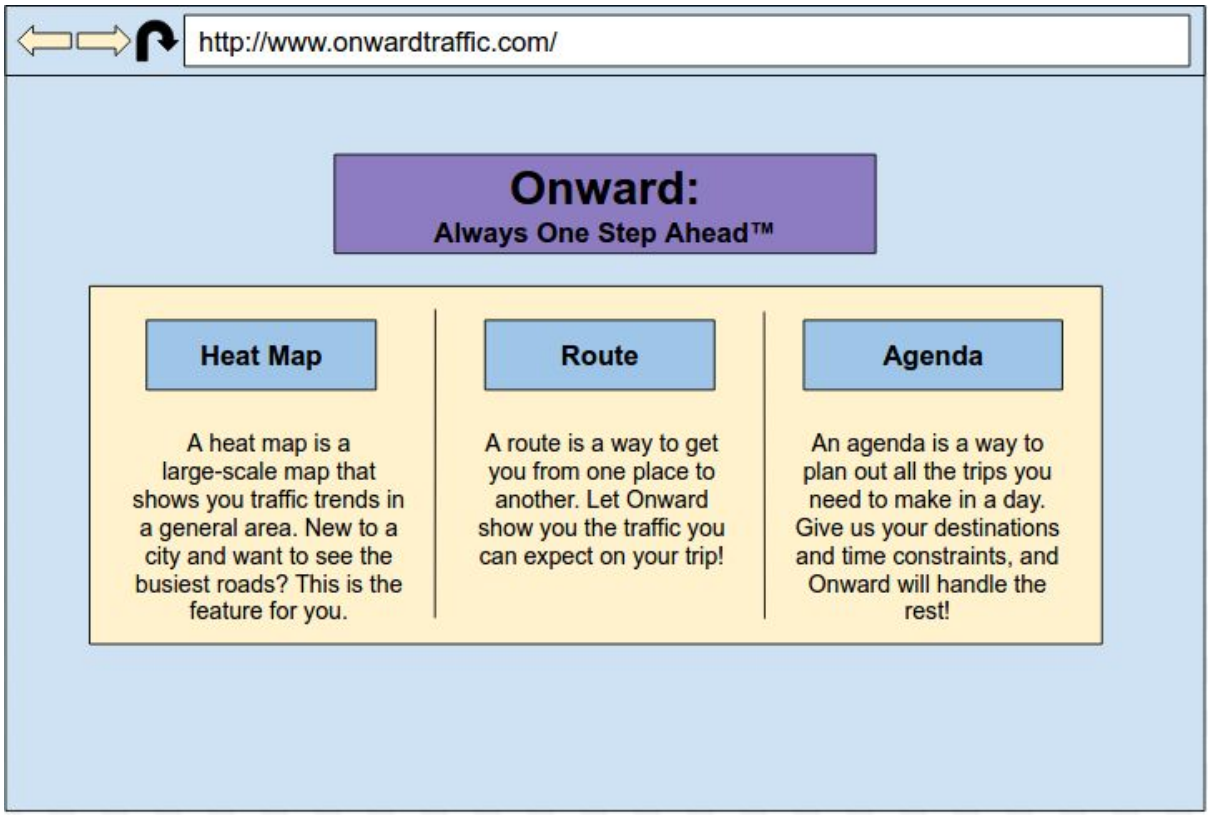

Figure 32: Landing Page Mockup

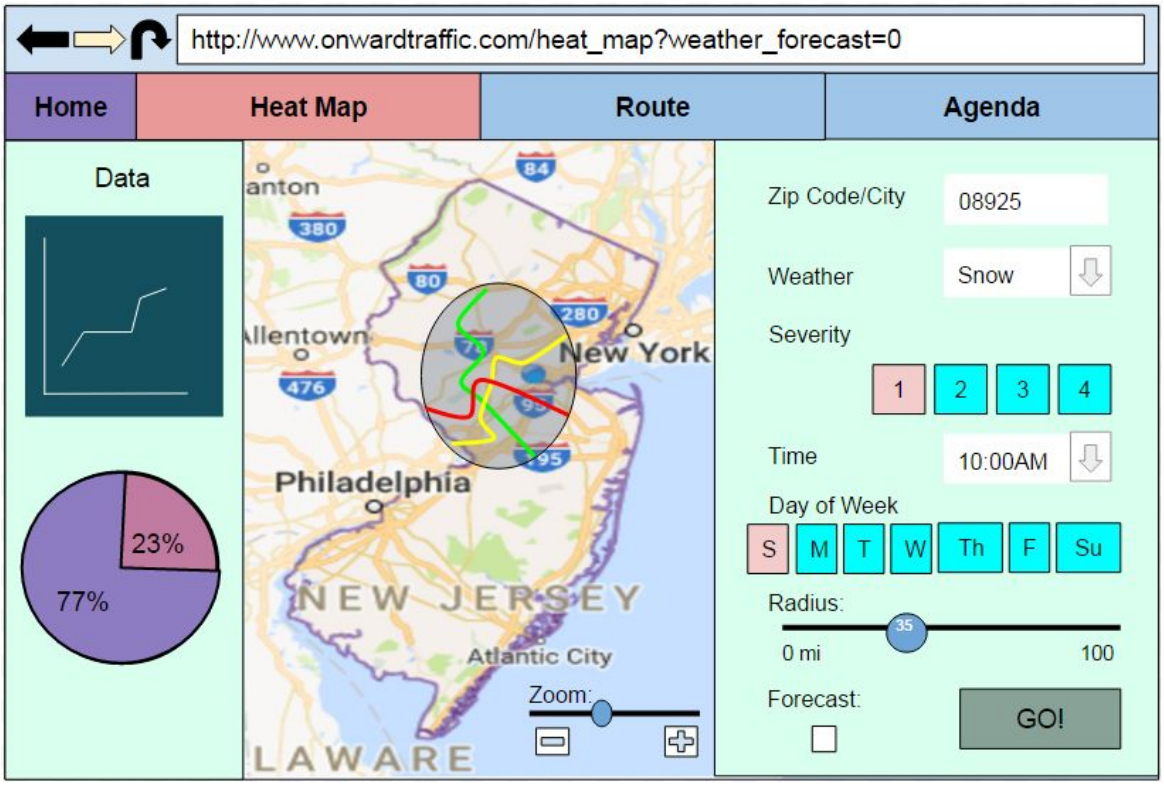

Figure 33: Heat Map Mockup

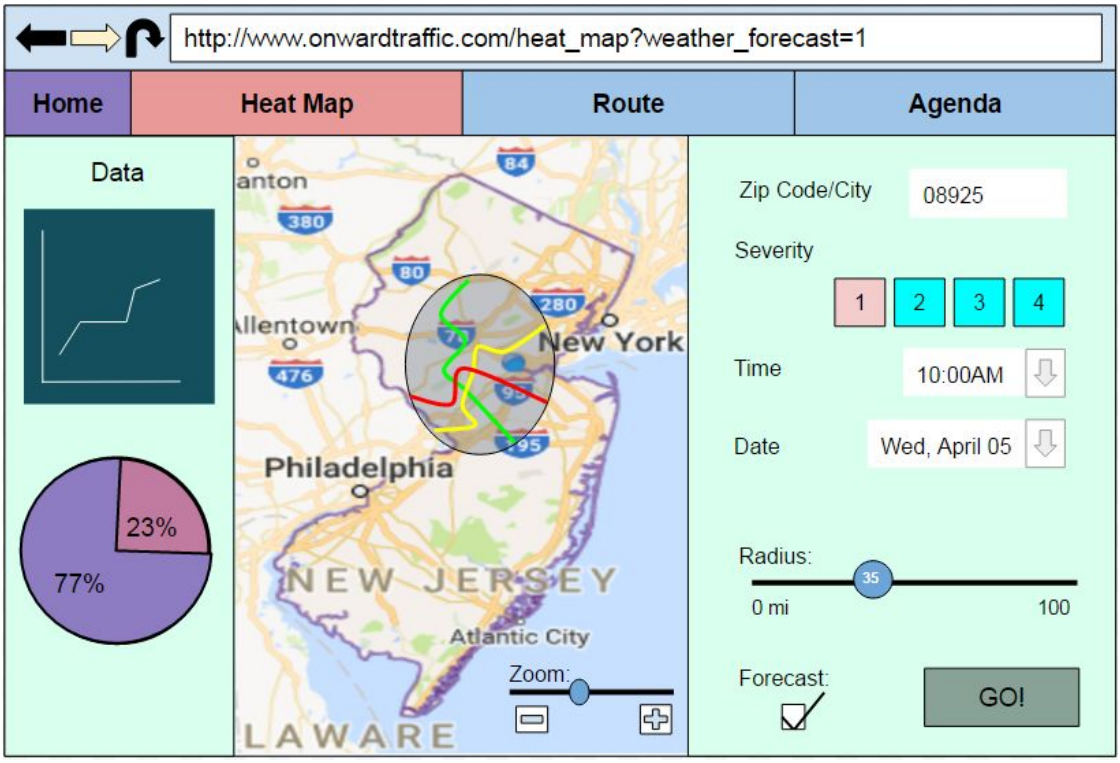

Figure 34: Heat Map with Weather Forecast Mockup

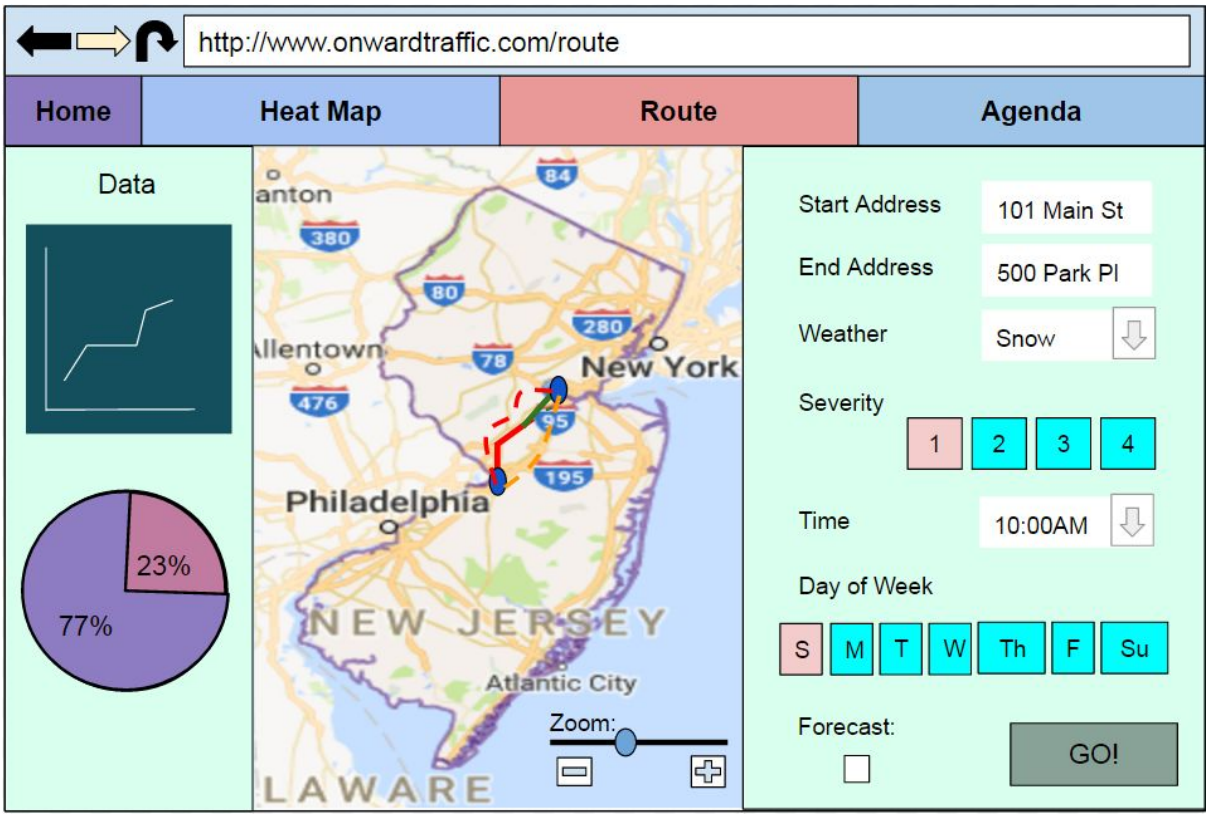

Figure 35: Route Mockup

These mockups were transitioned into implementation, better refined and made appropriate for actual implementation and to address the current developed use cases. There are flaws in these mockups, but that is because these are very early in the stages of this project. Those flaws have been addressed and fixed in the implemented user interfaces below. We have only placed the above mock ups in the report for the reader to see the evolution of the software.

Implemented

Our user interfaces from Report 2 have remained relatively the same with some new improvements for involving the use cases we have developed as well as having a better, user friendly, dynamic interface. We show the parts of the user interface that have been implemented for UC-1, UC-2, UC-4,UC-5 and UC-6 as well as now using dynamic forms for a better user experience.

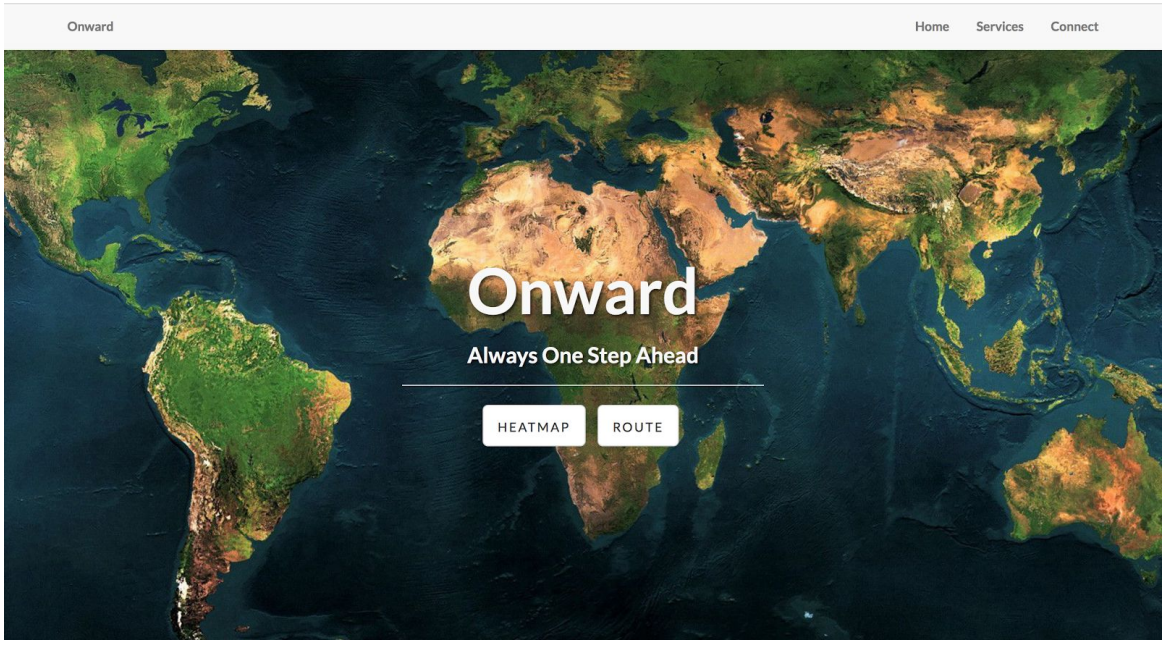

Figure 36: Landing Page (1)

Figure 36 is our current Landing Page, where a user will be directed when they go to http://www.onwardtraffic.com/. We have 2 feature buttons,”HEATMAP” and “ROUTE” that direct to their respective pages. We also place feature descriptions below the figure you see above, as shown in Figure 37.

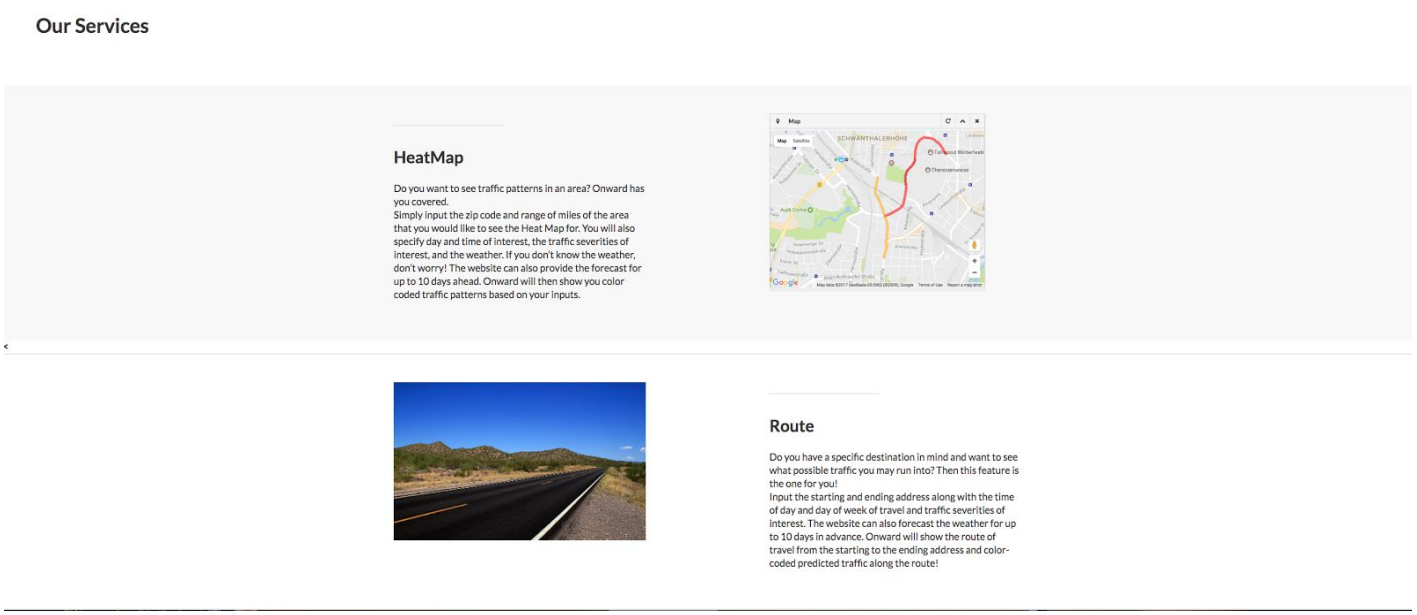

Figure 37: Landing Page (2)

On the same Landing Page, if the user scrolls down, they will see Figure 37. We decided to describe the features below the Figure 36 screen, as opposed to on it, for a few reasons. The first is that having the descriptions in the feature buttons would be cluttered and might overload a new user with new information. The other being that a returning user would not want to see the overview of the features every time they visit our site, but rather just select one of the features right away. This does not increase user effort, but rather declutters and better organizes our Landing Page.

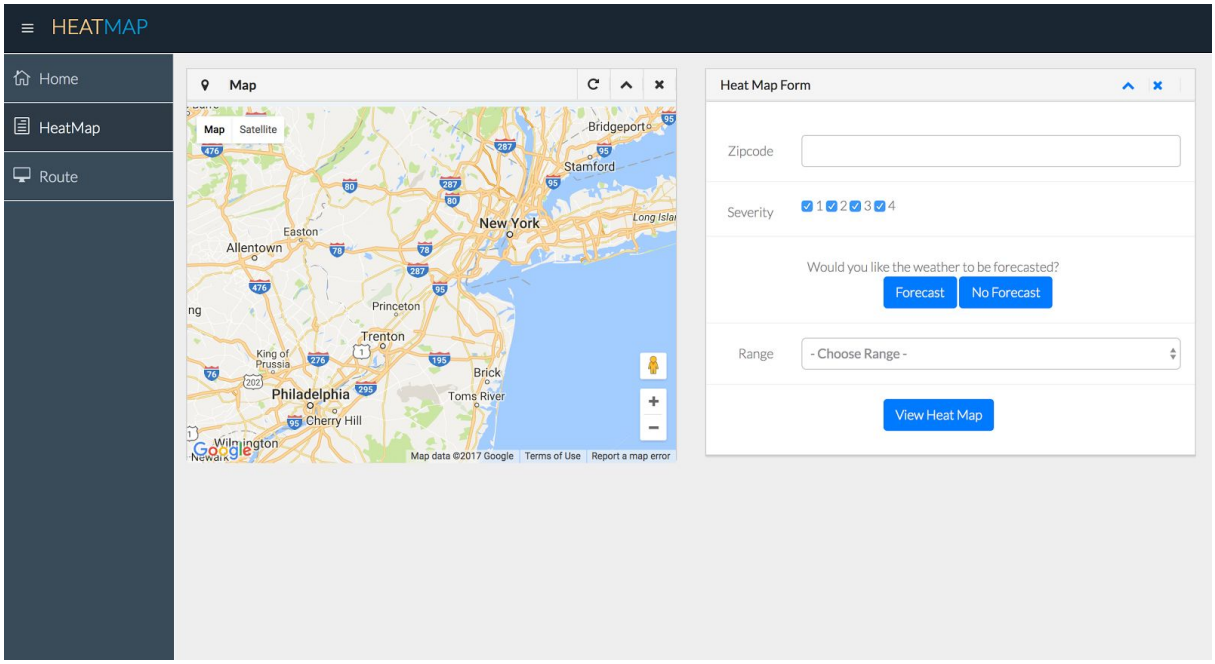

Figure 38: Heat Map Feature Page

Figure 38 shows the user interface of the Heat Map feature. The user would navigate to this page by clicking on the “Heat Map” button shown in the center of Figure 36. The user can switch between features using the menu on the left. There is an embedded Google Maps Javascript Map that is interactive as well as an input form for the user to use. We see that there is “Forecast” and “No Forecast” button on the page. This to allow the user to choose whether they want us to forecast a weather for them or they want to input a weather condition themselves.

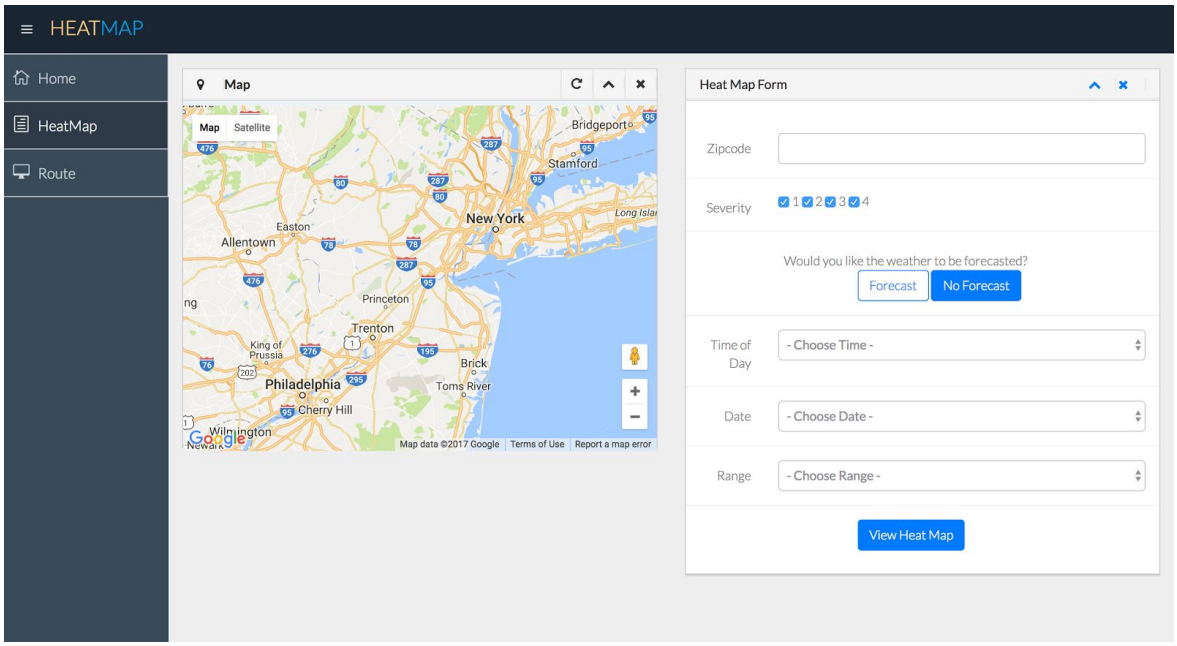

Figure 39: Heat Map Feature Page - Forecast

Figure 39 is the user interface on the Heat Map page when the user wants to use forecasted weather. They first enter a 5 digit zip code. They can select a “Time of Day” from the “Choose Time” dropdown that will provide the options of 12:00 AM to 11:00 PM, select “Date” from the “Date” drop down to choose a date that is within the next 10 days from the current date, and select a “Range” from the “Choose Range” drop down with options are “5” , “10”, “15, “20” miles. They also select the severities they are interested by checking and unchecking the check boxes.

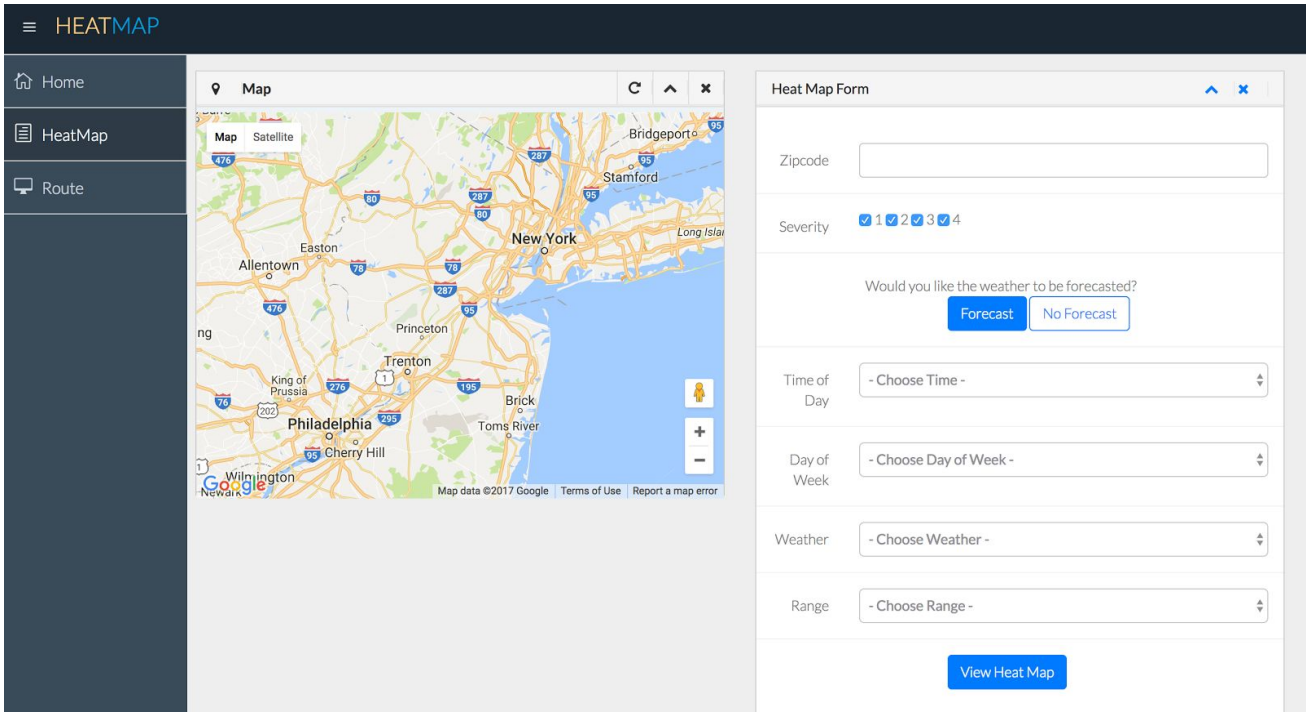

Figure 40: Heat Map Feature Page - No Forecast

Figure 40 is the user interface on the Heat Map page when the user wants to manually input weather. They first enter a 5 digit zip code. They can select a “Time of Day” from the “Choose Time” dropdown that will provide the options of 12:00 AM to 11:00 PM, select “Day of Week” from the “Choose Day of Week” drop down to choose either “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday” or “Sunday”, select “Weather” from the drop down “Choose Weather” that has the options “Any Weather”, “Clear”, “Snow”, “Cloudy”, “Rain”, and “Fog”, and select a “Range” from the “Choose Range” drop down with options are “5” , “10”, “15, “20” miles. They also select the severities they are interested by checking and unchecking the check boxes.

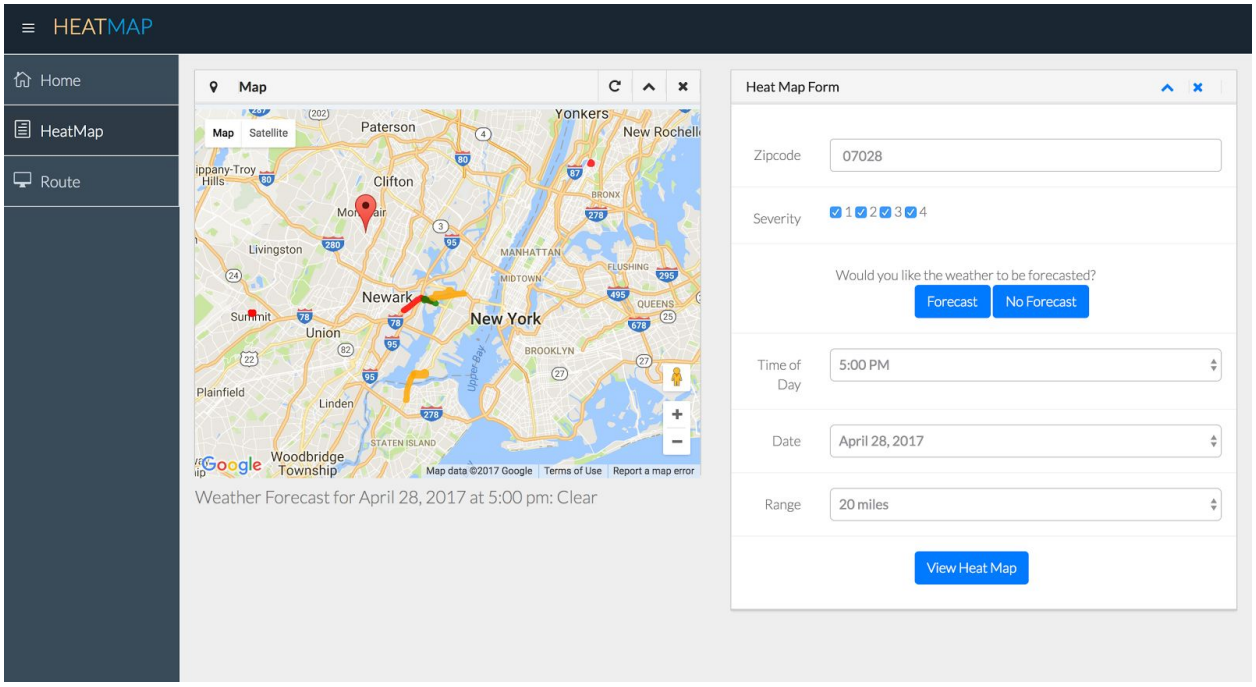

Figure 41: Heat Map Feature Page - Sample Output

Figure 41 shows a sample input and output on the Heat Map Feature, using the weather being forecasted. The map has marker in the center of the zip code entered, and roads highlighted based on traffic (severity 4 is red, severity 3 is orange and severity 1 is green). We also print the weather forecast condition below the map.

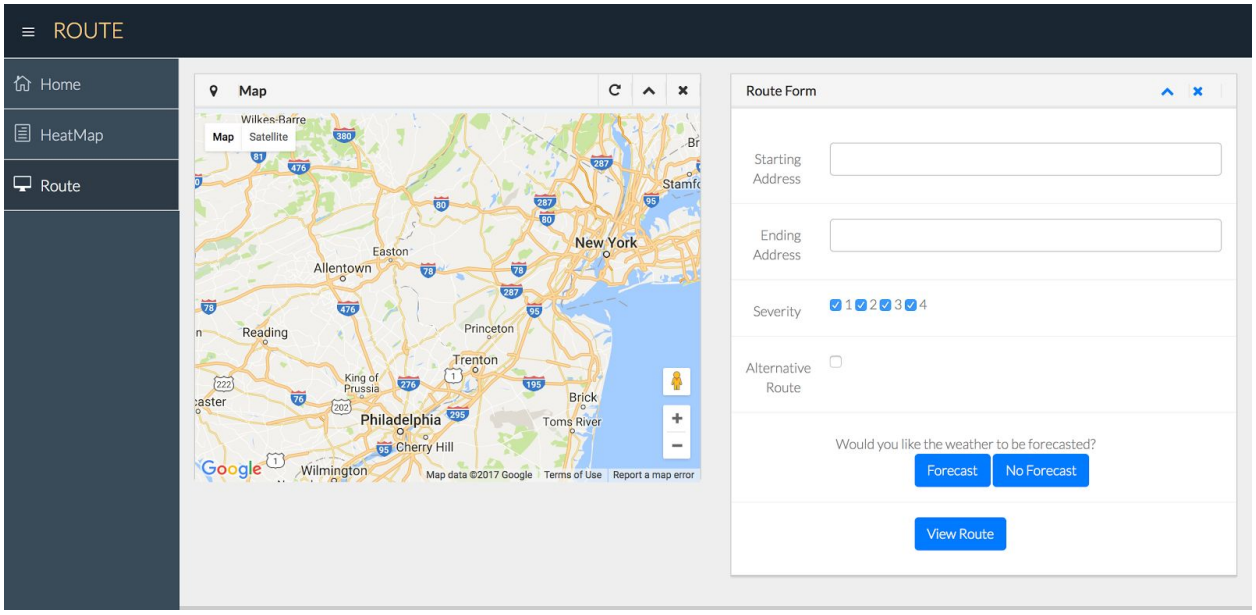

Figure 42: Route Feature Page

Figure 42 is the user interface of the Route Feature. The user can navigate to this page by clicking on the “Route” button shown in the center of Figure 36. Similar to the Heat Map, there is a Google Maps Javascript Map and a user input form.

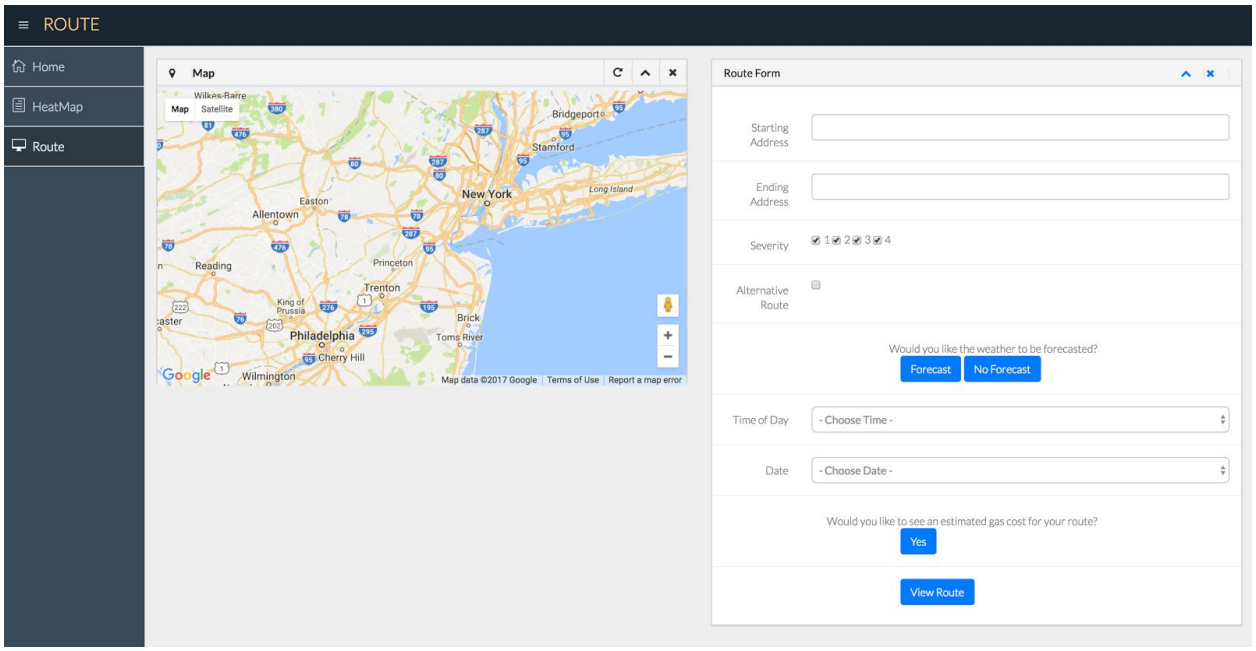

Figure 43: Route Feature Page - Forecast

Figure 43 is the user interface on the Route page when the user wants to use forecasted weather. The first enter a valid starting and ending address.They can select a “Time of Day” from the “Choose Time” dropdown that will provide the options of 12:00 AM to 11:00 PM and

select “Date” from the “Date” drop down to choose a date that is within the next 10 days from the current date.They also select the severities they are interested by checking and unchecking the check boxes. They have the option to view an alternate route by clicking the “Alternative Route” check box.

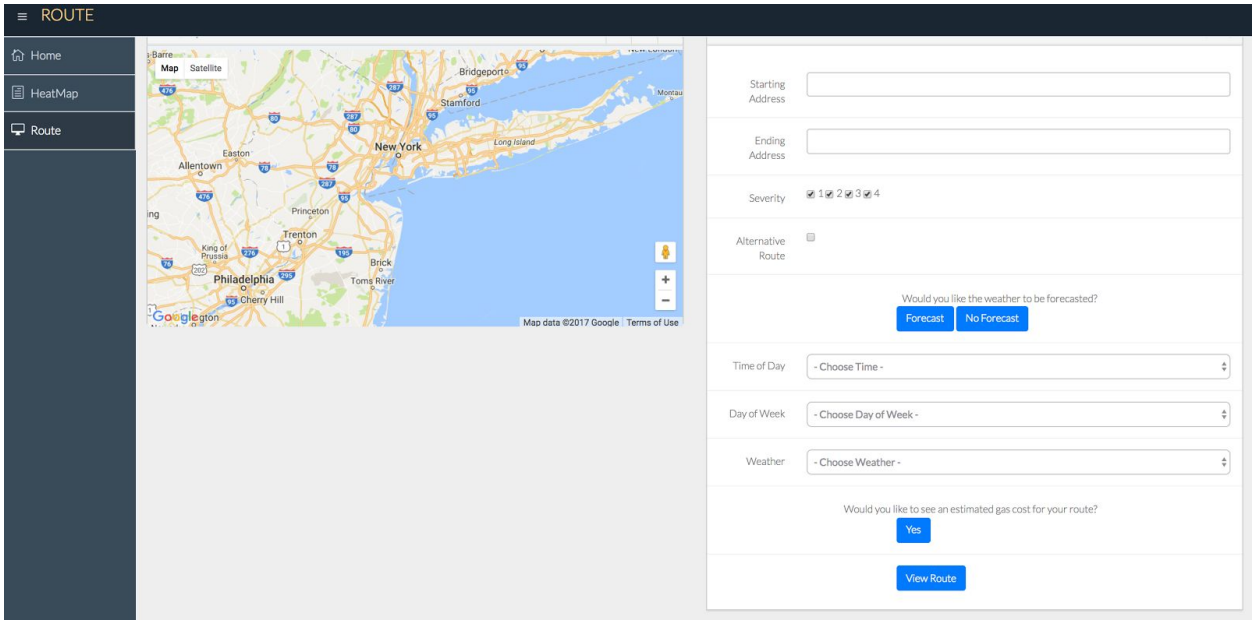

Figure 44: Route Feature Page - No Forecast

Figure 44 is the user interface on the Route page when the user wants to manually input weather. The first enter a valid starting and ending address.They can select a “Time of Day” from the “Choose Time” dropdown that will provide the options of 12:00 AM to 11:00 PM, select “Day of Week” from the “Choose Day of Week” drop down to choose either “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday” or “Sunday”, select “Weather” from the drop down “Choose Weather” that has the options “Any Weather”, “Clear”, “Snow”, “Cloudy”, “Rain”, and “Fog”. They also select the severities they are interested by checking and unchecking the check boxes.

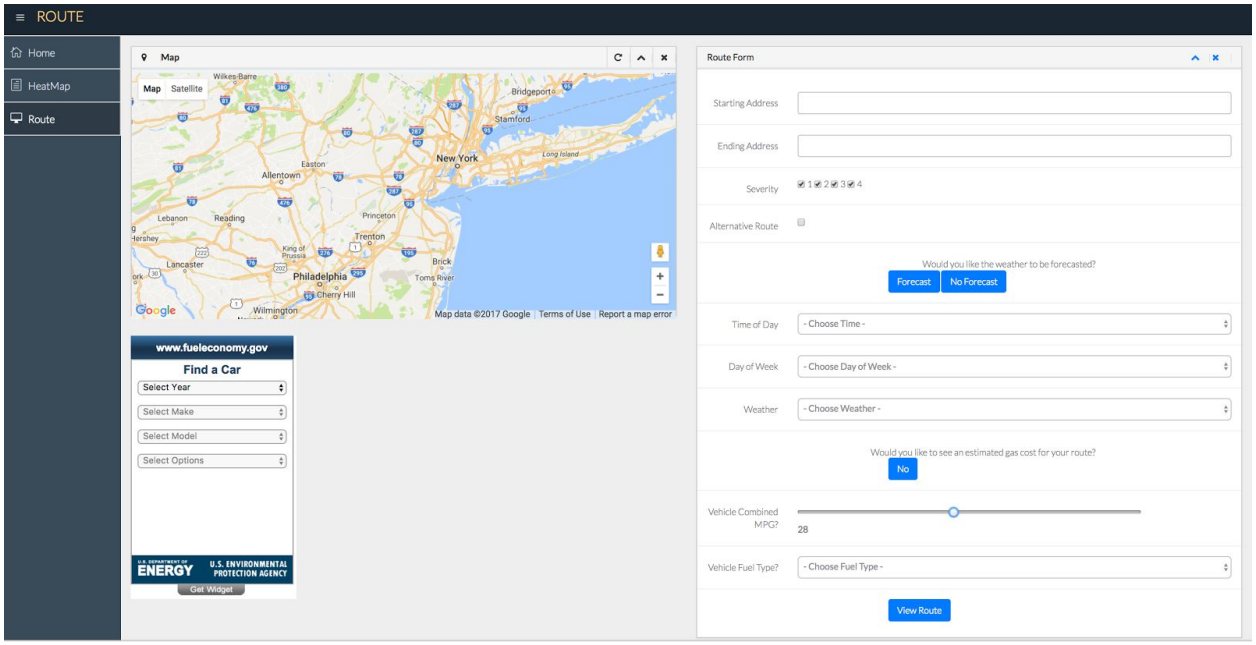

Figure 45: Route Feature Page - No Forecast and View Gas Cost

Regardless of whether the chooses forecast or no forecast, they are also given the question if “Would you like to see an estimated gas cost for your route?”. If they select “Yes” , they are presented with a slider to input the combined MPG of their vehicle and enter the fuel type of their vehicle using the drop down “Choose Fuel Type”, where the options are “Regular”, “Midgrade”, “Premium” and “Diesel. As a tool, we provide the fueleconomy.gov widget where a user enters their year, make, model and options (engine type) and get back the combined mpg and fuel type. Then, after submission pressing “View Route” they will be presented with the a printed statement for the cost of gas along the route as well as an alternate route if they had chosen that option.

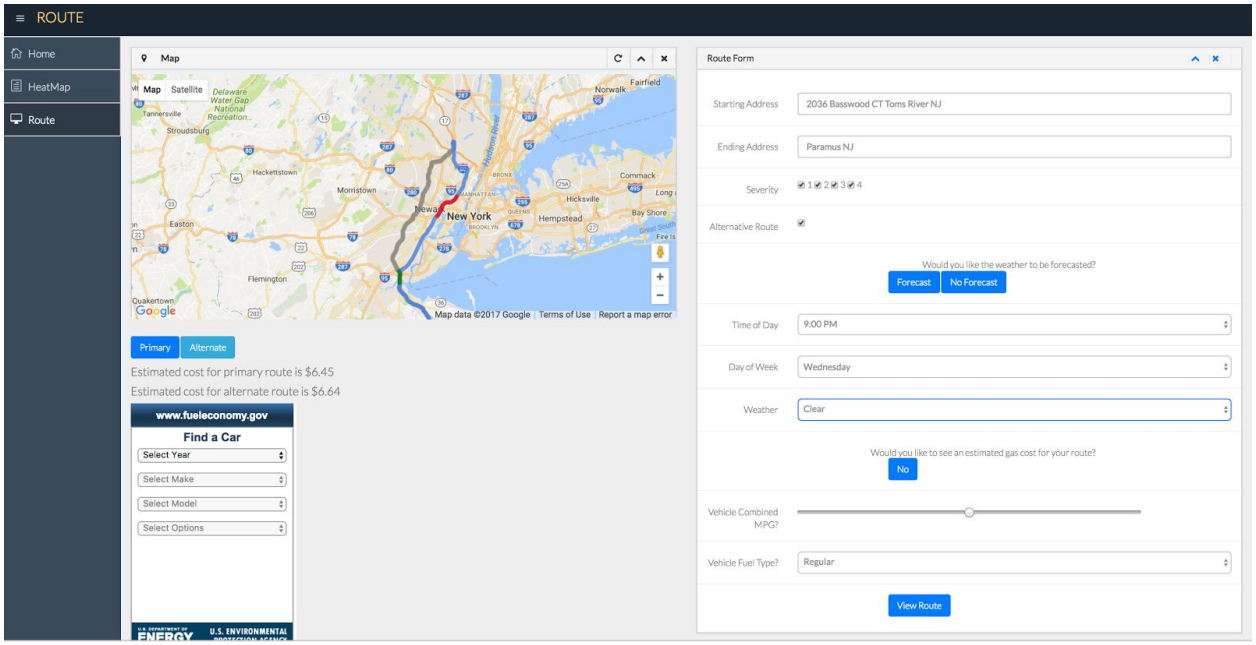

Figure 46: Route Feature Page - Sample Input and Output

Figure 46 shows the system showing both alternate routes and the gas costs. One route is highlighted with traffic and the other is grayed out. Clicking the blue and light blue “Primary” and “Alternate” buttons will switch which route highlighted with traffic and which is greyed out. The gas prices of the primary and alternate route are also printed.

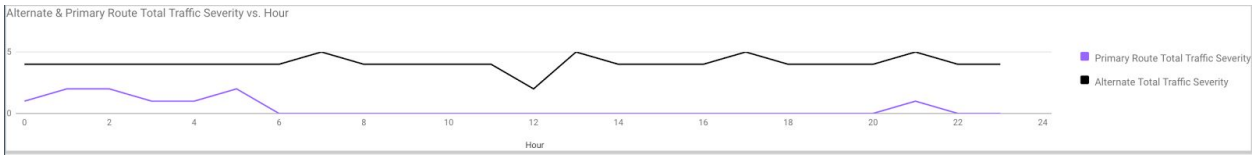

Figure 47: Graphs

If the user scrolls down on the same page they can view graphical representation of the traffic. What this graph shows is the 24 hours along the x axis and an aggregate of the severities along a route on the y axis (if we have a severity 4 road segment and a severity 2 road segment along a route then a 6 is on the y axis). Here we show it to compare a primary and alternate route, but a graph can also be shown for just one route when alternate route option is not chosen. We create these graphs using Google Charts[10].

To implement the web pages, Bootstrap[7] templates were used as a starting point, and modified to match our desired design. We used the template “Landing Page”[8] for the Landing Page and the “Nice Admin”[9] template for the other feature pages.

User Effort Estimations

We show here certain ways the user may interact with our system, showing a few of the permutations, and covering all the use cases.

UC-1 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click “Heat Map”
    --*after completing data entry as shown below*--
    b. Click “View Heat Map”
2. Data Entry: Total 11 mouse clicks and up to 5 keystrokes as follows:
    a. Click cursor to “Zip Code” text field
    b. Press five keys which correspond to the zip code i.e. “07652”
    c. Click on the severity of the traffic incidents don’t want displayed i.e. “1”(severities are all checked by default)
    d. Click “No Forecast”
    e. Click the dropdown menu next to “Time of Day”
    f. Click the desired time of interest to display the Heat Map i.e. 2:00 PM
    g. Click the dropdown button next to Day of Week to choose day.
    h. Click the desired day of the week from dropdown i.e “Sunday”
    i. Click the dropdown button next to “Weather”.
    j. Click the desired weather option i.e. “Clear”
    k. Click the dropdown button next to “Range” to choose range
    l. Click the desired range in miles around zip code i.e 10

UC-6 with UC-1 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click “Heat Map”
    --*after completing data entry as shown below*--
    b. Click “View Heat Map”
2. Data Entry: Total 9 mouse clicks and 5 keystrokes as follows:
    a. Click cursor to “Zip Code” text field
    b. Press five keys which correspond to the zip code i.e. “07652”
    c. Click on the severity of the traffic incidents don’t want displayed i.e. “1”(severities are all checked by default)
    d. Click “Forecast”
    e. Click the dropdown menu next to “Time of Day”
    f. Click the desired time of interest to display the Heat Map i.e. 2:00 PM
    g. Click the dropdown button next to “Date” to choose day
    h. Click the desired date to display i.e. “April 27, 2017”
    i. Click the dropdown button next to “Range” to choose range

j. Click the desired range in miles around zip code i.e 10

Note: Note here that using Forecast removes the need to select a weather. This has the same effect for the "Route feature". For "Route" User Effort Estimation we will continue to show with "No Forecast".

UC-2 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 2 mouse clicks as follows:
    a. Click "Route"
    --*after completing data entry as shown below*--
    b. Click "View Route"
2. Data Entry: Total 10 mouse clicks and 22 keystrokes as follows:
    a. Click cursor to "Starting Address" text field
    b. Press the keys which correspond to the address i.e. 101 Main St
    c. Click cursor to "Ending Address" text field
    d. Press the keys which correspond to the address i.e. 500 Park Pl
    e. Click on the severity of the traffic incidents don't want displayed i.e. "1"(severities are all checked by default).
    f. Click "No Forecast".
    g. Click the dropdown menu next to "Time of Day".
    h. Click the desired time of interest to display the Route i.e. 2:00 PM
    i. Click the dropdown button next to "Day of Week" to choose day.
    j. Click the desired day of the week from dropdown i.e "Sunday"
    k. Click the dropdown button next to "Weather".
    l. Click the desired weather option i.e. "Clear"

Note: Keystroke count depends on the address inputted. Click count will be different if they choose to un-see multiple severities.

Next we will show UC-2 with UC-4 and UC-5, so the Route feature using an alternate as well as Gas Cost. This will be the maximum interaction of the user using the system.

UC-2 with UC-4 and UC-5 (Assuming that the user has successfully navigated to the home page of the website)

1. Navigation: Total 3 mouse clicks as follows:
    a. Click "Route"
    --*after completing data entry as shown below*--
    b. Click "View Route"
    c. Click "Alternate" to view traffic on alternate route
2. Data Entry: Total 15 (20 with widget) mouse clicks and 22 keystrokes as follows:
    a. Click cursor to "Starting Address" text field
    b. Press the keys which correspond to the address i.e. 101 Main St
    c. Click cursor to "Ending Address" text field

d. Press the keys which correspond to the address i.e. 500 Park Pl
e. Click on the severity of the traffic incidents don't want displayed i.e. "1"(severities are all checked by default).
f. Click checkbox for "Alternate Route"
g. Click "No Forecast".
h. Click the dropdown menu next to "Time of Day".
i. Click the desired time of interest to display the Route i.e. 2:00 PM
j. Click the dropdown button next to "Day of Week" to choose day.
k. Click the desired day of the week from dropdown i.e "Sunday"
l. Click the dropdown button next to "Weather".
m. Click the desired weather option i.e. "Clear"
n. Click "Yes" to "Would you like to see an estimated gas cost for your route?"
o. Optional: Additional 5 Clicks to use Fuel Economy Widget to see combined MPG and Fuel type
p. Click and drag slider to enter Vehicle Combined MPG
q. Click dropdown menu "Choose Fuel Type"
r. Click the desired Fuel Type i.e "Regular"

For the effort estimations, the worst case scenario is that the user would re-enter their zip code or addresses which would increase keystrokes.

There are many permutations for the user using the system, depending on whether they use Heat Map or Route, Forecast or No Forecast, want or don't want Alternative Route, wants to view or doesn't want to view cost of gas. The effort will vary user to user, depending on the permutation they choose.

We put great efforts into making our forms be dynamically changed as to improve the user experience. Before, we had it so that if the user were to suppose choose "Forecast" the page would have to reload to a different page, and as a result a different index file. To better streamline this for the user, we made it so the form dynamically changes so when user clicks "Forecast" or "Yes" to viewing gas the input form is modified and adjusted instantly. In addition to that, to improve our user interface for demo 2, we have reduced the number of times the user would need to click "View Heat Map" or "View Route". If the user changes simple parameters by change the time or date or severities to view, the interface can immediately change such that the map and traffic shown is properly changed. This took a great amount of development time, as we now query a much larger subset of our database based on locations and show the user a certain subset based on the current inputs, and as they change time, day or severities we show different subsets.

Section 12: Design of Tests

We designed test cases that would reflect possible expected and unexpected behavior on the part of the user and participating actors. This is to ensure that our system can handle, for example, bad user-inputs or an HTTP request to an API returning an error.

We separated these test cases by the object that they apply to.

Unit Test Cases

Note: As instructed in class, in unit tests, functions should be selectively tested as to have our testing be systematic and informative for developers, thus, not all functions in every class have been unit tested. For those functions that aren't unit tested, an explanation is provided below.

InfoEntry:
There are currently no unit tests for the InfoEntry class because it is impossible for it to fail. The info entry class is simply for sending the variables entered in the heatmap/route form to the controller. All form entries are using error-proof entry fields such as checkboxes and dropdowns except for the location entry. For the route feature, the text entered in location can not fail because the google maps api can interpret any text as a location. If the text is not interpreted as a location, this would be caught by the Map Communicator class, which we have tests for. For the heatmap feature, the text entered for a zip code cannot fail because we validate to make sure the entered text is a 5 digit number. The variables get sent to the controller through a form post. This class also has two functions, form_change() and select_route_subset_alert_info() which do not get unit tested. The function form_change() simply refreshes the map when a user changes his form inputs and has a deterministic output every time. The function select_route_subset_alert_info() simply implements a switch case. Therefore, it is unneccessary and non-systematic to unit test these functions. Although this class is not unit tested, it is covered in the integration test to assure that it works with the rest of the system.

ScreenDisplay:

Test-Case Identifier: TC-1

Function Tested: initMap(): HTML Element

Pass/Fail Criteria: The test passes if the default map is displayed on the screen.

Test Procedure	Expected Results
Call function with valid API key. (Pass)	The API key has not reached its quota. An HTML element for a map centered on NJ is displayed on the screen.

Test-Case Identifier: TC-2

Function Tested: routeMap(): HTML Element

Pass/Fail Criteria: The test passes if a map with a polyline highlighting a route is displayed.

Test Procedure	Expected Results
Call function with valid API key. (Pass)	The API key has not reached its quota. An HTML element for a map with a highlighted route is displayed on the screen.

The screen display class contains 4 functions: initMap(), routeMap(), drawChart(), and switchRoutes(type). None of these functions take any parameters except switchRoutes(). The functions initMap() and routeMap() are tested by simply calling them and checking to see if a map is displayed on a web page. This is done to verify that the API key has not reached its quota and is still valid. This is the only way this function could fail. The function drawChart() is not unit tested because it does not require an API key and thus cannot fail. The function switchRoutes() is not tested because it is also impossible for this function to fail on its own. It gets its input from a button on the user interface and has a straightforward execution every time. The precondition for these functions is the successful execution of the modules that ran before it. As long as that happens, these functions will not fail. For this reason, we do not have a strict need to test these functions by themselves.

WeatherCollector:

Test-Case Identifier: TC-3

Function Tested: getForecast(dateWC, timeWC, zipWC): String

Pass/Fail Criteria: The test passes if the function returns a weather forecast for the time, day, and zip code entered by the user.

Test Procedure	Expected Results
Call function with day and time within the 10-day forecast window. (Pass)	The function will call the callWeatherService() method to obtain a weather forecast for the date and time specified. It will return the forecasted weather.
Call function with invalid zip code. (Fail)	The function will not be able to forecast the weather and will do nothing.

The following functions in the WeatherCollector class do not get unit tested: callWeatherServiceLoc(), getInputIndex(), weatherSwitch(), and getTimeStr(). This is because these four functions are simply private helper functions that only get called when the getForecast() function gets called. It is impossible for them to fail on their own based off any external events. For example, weatherSwitch() and getTimeStr() simply implement switch cases on a string passed to them by the getForecast() function.

MapCommunicator:

Test-Case Identifier: TC-4

Function Tested: get_route(my_location): JSON

Pass/Fail Criteria: The test passes if a route to get from starting point to destination is returned.

Test Procedure	Expected Results
Call function with valid location for starting and ending points. (Pass)	A JSON with a route to get from starting point to ending point is returned.
Call function with invalid location for starting and ending points. Invalid point could be a point on an ocean. (Fail)	The function will not be able to find a route and return false.

Test-Case Identifier: TC-5

Function Tested: parse_route(routeJSON, routeNum): Array

Pass/Fail Criteria: The test passes if an array of the steps to get from starting point to destination is returned.

Test Procedure	Expected Results
Call function with valid routeJSON. (Pass)	An array of the steps in the routeJSON is returned.
Call function with invalid routeJSON. (Fail)	The function will not be able parse the route and return false.

The following function in the MapCommunicator class does not get unit tested: parse_distance(). This is because this function simply parses the JSON response received by the get_route() function and is impossible to fail unless the get_route() functions fails.

TrafficCollector:

Test-Case Identifier: TC-6

Function Tested: query_heat_db(cent_lat, cent_lng, query_range): Array

Pass/Fail Criteria: The test passes if there is traffic information for the given location and conditions in the database and it is successfully queried.

Test Procedure	Expected Results
Call function with valid location. Database contains traffic information for this input. (Pass)	Function queries the traffic table database to obtain all traffic information for desired location and conditions. Returns information to calling method.
Call function with invalid location. Database does not contain traffic information for this input. (Fail)	The function will not be able to find traffic information and return false.

Test-Case Identifier: TC-7

Function Tested: get_segment_sev(roadName, startLat, startLong): Array

Pass/Fail Criteria: The test passes if traffic information for the segment of the road provided is returned.

Test Procedure	Expected Results
Call function with valid road name and coordinates that correspond to this road. (Pass)	Traffic information along the segment of the road provided is obtained and returned.
Call function with valid road name but coordinates that don't correspond with this road. (Fail)	The function will not be able to find traffic information and return false.

The following functions in the TrafficCollector class do not get unit tested: create_sev_array(), switch_day(), switch_weather(), and get_traff(). This is because these four functions are simply helper functions. The function get_traff() simply calls get_segment_sev() or query_heat_db() depending on the feature selected by the user and thus by testing get_segment_sev() and query_heat_db(), we effectively cover this function. The function create_sev_array() is used to create a template for an array and is impossible to fail. The functions switch_day() and switch_weather() simply implement a switch case statement, therefore, it is impossible for them to fail as well.

LocConverter:

Test-Case Identifier: TC-8

Function Tested: getZip(locParams, feature): int

Pass/Fail Criteria: The test passes if the function successfully converts location parameters into a valid zip code and returns the zip code to the caller.

Test Procedure	Expected Results
Call function with valid locationParams (zip code). Set feature to "heatmap". (Pass)	Function verifies that the zip code entered is valid and returns it to caller.
Call function with invalid locationParams. Set feature to "heatmap". (Fail)	The function will detect an invalid zip code and return false.
Call function with valid locationParams (starting address). Set feature to "route". (Pass)	The function converts the address into a valid zip code and returns that zip code to the caller.
Call function with invalid locationParams (starting address). Set feature to "route". (Fail)	The function will not be able to convert the address to a zip code and return false.

Test-Case Identifier: TC-9

Function Tested: callLocServHeat(locParams): Array

Pass/Fail Criteria: The test passes if the function successfully converts location parameters for zip code into a set of coordinates using a geocoding API and returns the coordinates to the caller.

Test Procedure	Expected Results
Call function with valid locationParams (zip code). API key has not passed its usage limit. (Pass)	Function uses geocoding API to convert the location into a set of coordinates and returns them to caller.

Call function with invalid locationParams. API key has not passed its usage limit. (Fail)	The function will not be able to do the geocoding and will return false.

Test-Case Identifier: TC-10

Function Tested: callLocServRoute(locParams): Array

Pass/Fail Criteria: The test passes if the function successfully converts location parameters for starting and ending addresses into two sets of coordinates using a geocoding API.

Test Procedure	Expected Results
Call function with valid locationParams (starting and ending address). API key has not passed its usage limit. (Pass)	Function uses geocoding API to convert the starting and ending points into two sets of coordinates and returns them to calling method.
Call function with invalid locationParams. API key has not passed its usage limit. (Fail)	The function will not be able to do the geocoding and will return false.

The function get_location() in the Location Converter class does not get tested because its job is simply to call the CallLocServ() functions. By testing the CallLocServ() functions, we effectively test the get_location() function.

Controller:
The Controller only has two functions, connect_to_db() and just_forecast(). Both of these are short helper functions that replaced blocks of codes that were in constant use. It makes no sense to unit test them because they just implement a short and deterministic if-else statement that will run the same way every time it's called.

Test Coverage

Although user interaction in Onward is limited, the system does not necessarily behave in a deterministic way. That is because our system frequently interacts with participating actors through the use of API calls and SQL queries. There are several reasons why these interactions may fail, such as a server outage, or the expiration of an API key. For this reason, we decided to develop a state-based coverage which outlines all the possible transitions and events the system may encounter while attempting to process a user request. We initially didn't consider these failure edge cases as meaningful enough to include. After experiencing a server outage which cut off access to our SQL database, we ended up realizing that they were, in fact, important enough to consider. By testing to ensure that all transitions and states are encountered, we effectively cover all requirements of this project. Furthermore, by unit testing each function with good and bad inputs, we fully cover each individual module.

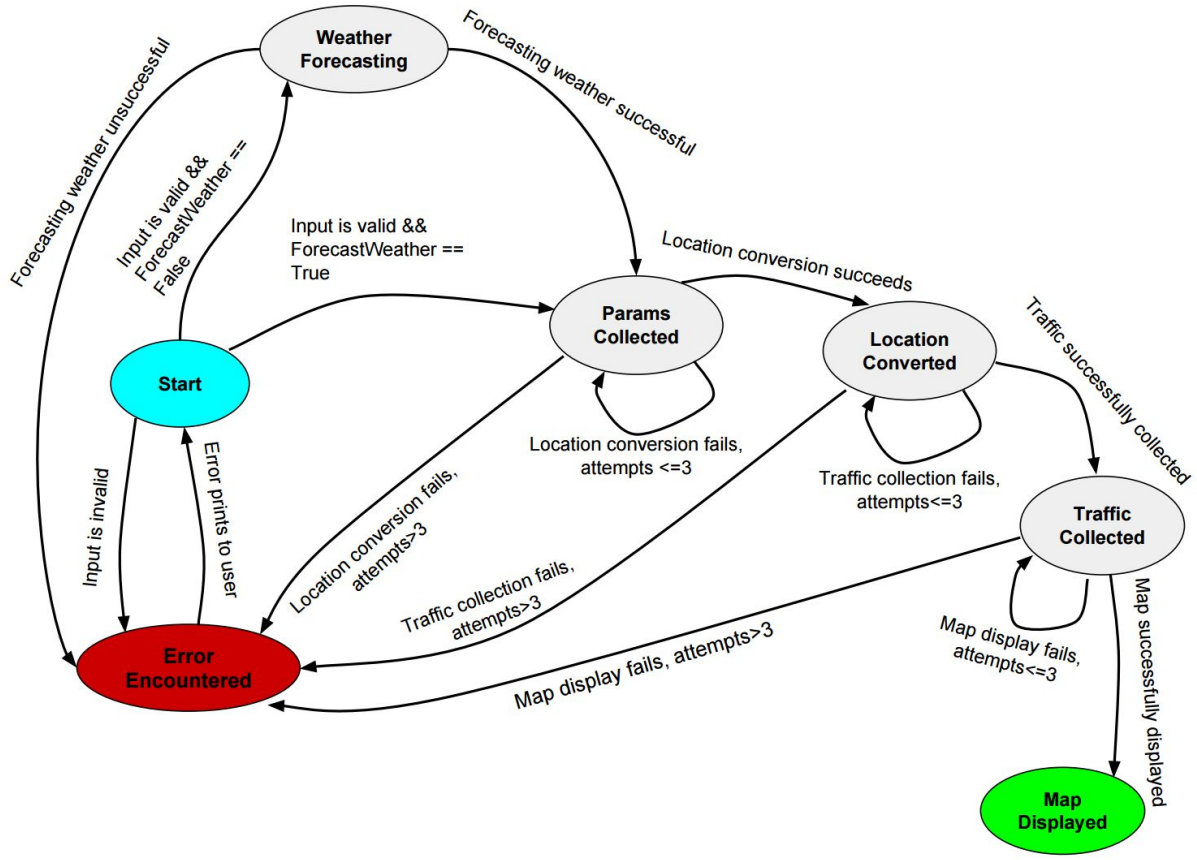

Figure 48: State Diagram for UC-1 and UC-6

Figure 48 shows the state diagram for UC-1, View HeatMap and UC-6, ForecastWeather. Successful execution is represented by the path from ‘Start’ to ‘Map Displayed’. Our integration test will ensure that all of these systems states and transitions are reached at least once.

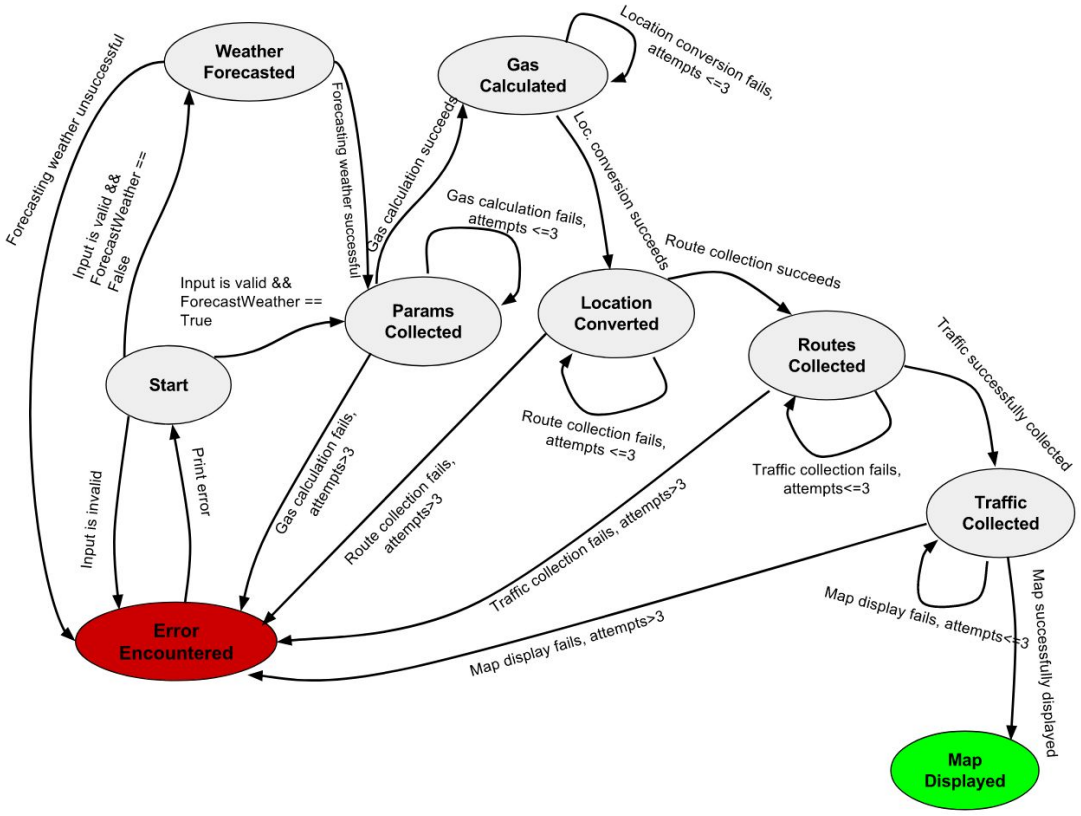

Figure 49: State Diagram for UC-2, UC-4, UC-5, & UC-6

Figure 49 shows the state diagram for UC-2, ViewRoute, UC-4, ViewAlternateRoute, UC-5, ViewGas, and UC-6, ForecastWeather. This diagram differs from Figure 48 in that this time the “Routes Collected” stage must be entered before going to “Traffic Collected”. Also note that the ViewRoute feature has the added step of collecting gas cost for the route.

Integration Testing

After unit testing is complete, we will perform our integration testing based around the state-based diagrams above. We will make sure that every state is reached at least once, and that every transition, both valid and invalid, will be encountered at least once. This will ensure that our units can function together for both expected and unexpected behavior. This also ensures that our testing is systematic.

Acceptance Testing

Once integration testing is complete, we will move onto acceptance testing. The goal of our acceptance tests is more holistic, and is to basically ensure that the system performs as expected when put into the hands of a regular user. The user will then report whether or not the desired functionality was achieved. The user's feedback will also help us to determine if the non-functional and UI requirements have been met. The UI requirements are binary, such as "will display a map" (it either does display a map, or it does not), so it is fairly easy to determine if they have been met.

Additionally, these user acceptance tests will allow us to gather qualitative feedback from an end user on how our system can improve. This feedback could involve a desired UI change to improve ease-of-use, a suggestion for a future feature, etc. Our design process is iterative, so we would be able to incorporate feedback into future releases if so desired.

Section 13: History of Work, Current Status, and Future Work

January 17, 2017 - January 29, 2017
In this time frame we formed our team of 6. We began with discussing each other's strengths and experience and to plan out a way of how we would all work together for the semester. We evaluated the project ideas on the website and found that the "Traffic Monitoring" project would be the most challenging and most rewarding project. We used this time to think more about our project and what features we wanted to create for our project. This was also the time frame we wrote our proposal and as a result explored ideas of what product we wanted to create, and submitted this proposal on January 29th.. We also began to look into what kind of data we would need and looked into API's we would use.

January 30, 2017 - March 5, 2017
Much of this time frame was used for setting up our database as well as doing the documentation for the first report, and some of the second report. With the first report, we were able to do in depth analysis of our requirements, of what our use cases were, and got a clear idea of what our system would be. We submitted the first report on February 18th. Setting up our database was a large deal of work for this time frame, as it required a lot of learning as well as it took as a lot of time to figure out what specific data we needed, as well as create the complex algorithms we explained above to segmentate roads into grids. The database needed many updates throughout the semester, as with more and more fine tuning, it became a robust database that greatly helped our system. We spent some of this time setting up the basics of our user interface as well as other parts of the basic infrastructure. With the start of report 2, we analyzed our interaction among classes in our system and what our classes would do to allow us to evaluate how we would make our system functional.

March 6 2017 - March 28, 2017
In this time frame we worked on the documentation for the Report 2 as well as preparation for our first demo. From doing the documentation, we had a good idea of what our classes would be and how classes would interact, which helped us transition into the heavy development phase. We submitted the second report on March 12th. We worked very diligently and efficiently to get the basics of the Heat Map and Route feature for our first demo ready. By the time the demo, which was March 28, we had the functional basics of the Heat Map and Route feature ready. We had also in this time created integration and unit tests for our system, which proved useful in evaluating our software and in helping us debug issues.

March 29, 2017 - May 1, 2017
After the first demo, we took some time to evaluate where we stood with the project and what parts remaining in the project we could accomplish. We realized that with a month remaining, the Agenda feature we had originally thought of was overly ambitious for the time frame of the

semester and that it could not be completed in this time frame. Therefore we decided to focus on the other use cases, which were providing the user with alternative route options as well estimated gas costs for the routes they view. In addition, we implemented showing the user analytical graphs to get more information regarding the traffic they are interested in. We also spent a great deal of time improving our user interface so that is more intuitive, quicker to use, and to provide an overall better experience to end users. We had also been working on the third report of the semester, where elaborated the other use cases/features we were adding to our project as well as the new design patterns we implemented into our project.

We worked to the deadlines and milestones planned in our previous reports as much as we could, but at many times the work was delayed. This was due in part to the fact that many times the documentation and reports took a lot more of our time, delaying development, as well as the fact that setting up the database, debugging and refining the database, took much of our time. However we see this time taken with the reports and database, although delaying development, well used. By taking time with reports and properly analyzing our problem domain, development was far more clear and we had a better understanding of our system. As with the database, setting up this robust database that took many weeks in preparation, allowed us to show great results once we implemented the rest of our system.

Current Status:

- Robust and functional database.
- Clean, sleek, and sophisticated user interface.
- Working feature providing users with a Heat Map of traffic in the vicinity of a zip code based on historic traffic data
- Working feature providing users to be able to view traffic conditions along a Route based on historic traffic data
- Allow users to view analytical graphs regarding traffic on routes of interest.
- Allow users to view traffic conditions along an alternate route based on historical traffic data.
- Allow users to view the cost of gas along a route, and also compare the cost of gas among a primary and alternate route.

Future Work:

For future work on this project, we would want to implement the Agenda feature. The agenda feature, as described previously in this report, would have allowed users to enter multiple destinations they would want to reach in a certain time frame and we would provide the optimal time to go to these destinations as well as the order to go to these destinations so that they are met with the least amount of traffic.

Section 14: References

1. Google GeoCoding API - *Google. 2017.* <https://developers.google.com/maps/documentation/geocoding/start>
2. Google Directions API - *Google. 2017.* <https://developers.google.com/maps/documentation/directions/>
3. Bing Traffic API - *Bing Developer Network.* 2017*.* <https://msdn.microsoft.com/en-us/library/hh441725.aspx>
4. Current Weather Data - *Weather Underground API*. 2017. <https://www.wunderground.com/weather/api>
5. Fuel Prices API - *FuelEconomy.gov Web Services.* <http://www.fueleconomy.gov/feg/ws/index.shtml#ft9>
6. Cron Job - *Wikipedia. 2017.* <http://www.adminschoice.com/crontab-quick-reference>
7. Bootstrap - *Start Bootstrap. 2017.* <http://getbootstrap.com>
8. Bootstrap Landing Page - *Start Bootstrap. 2017.* <https://startbootstrap.com/template-overviews/landing-page>
9. Bootstrap Nice Admin Template- *Start Bootstrap. 2017.* <https://bootstraptaste.com/nice-admin-bootstrap-admin-html-template>
10. Google Charts - *Google. 2017.* <https://developers.google.com/chart/>
11. Gliffy - *Gliffy Inc. 2017.* <https://www.gliffy.com>

The references 1-10 are used throughout this report, and there is a superscript matching the number of the reference the first time any of these references are mentioned. Reference 11 is the tool used for UML diagrams.