Course Title: 14:332:452 Software Engineering

Group 4

Onward (Traffic Monitoring)

Report 2 Part 2

March 5, 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

Interaction Diagrams

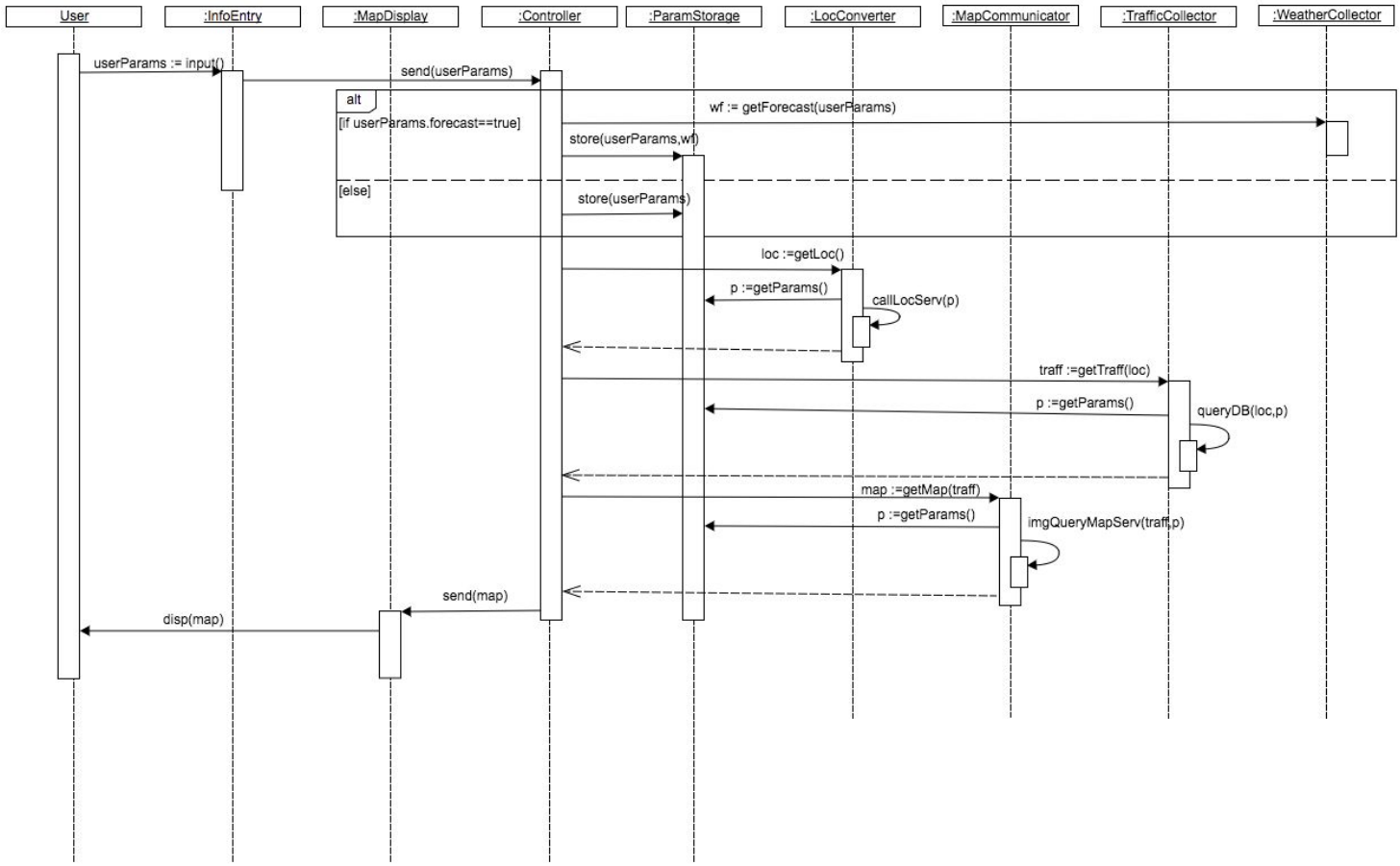

Figure 1: Sequence Diagram of UC-1 with extension UC-6

This diagram incorporates both UC-1, the “Heat Map” feature, and its extension UC-6. UC-6 is simply the situation 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 from after the point where the user chooses the “Heat Map” feature on the landing page and begins to input parameters. We designed all of our concepts with cohesion and modularity in mind. Having all communication mediated by the Controller entity allows us to mostly decouple the individual components of our system. This helps keep our system modular, since we can add and remove components at will 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.

As described in our alternate design decisions, we originally had an issue distinguishing in our system between the different parameters of UC-1 and UC-6. In order to clarify the origin of the parameters, we added the entity ParamStorage, which handles the job of distinguishing and storing the different parameters for the different use cases. Also, we decided against explicitly including the participating actors in the diagram to improve the simplicity and readability of the sequence diagram. Our participating actors are the Database, Location Service, Mapping Service and Weather Service. The system is client-side only when the user is entering information and viewing the map. Everything else happens on the server-side.

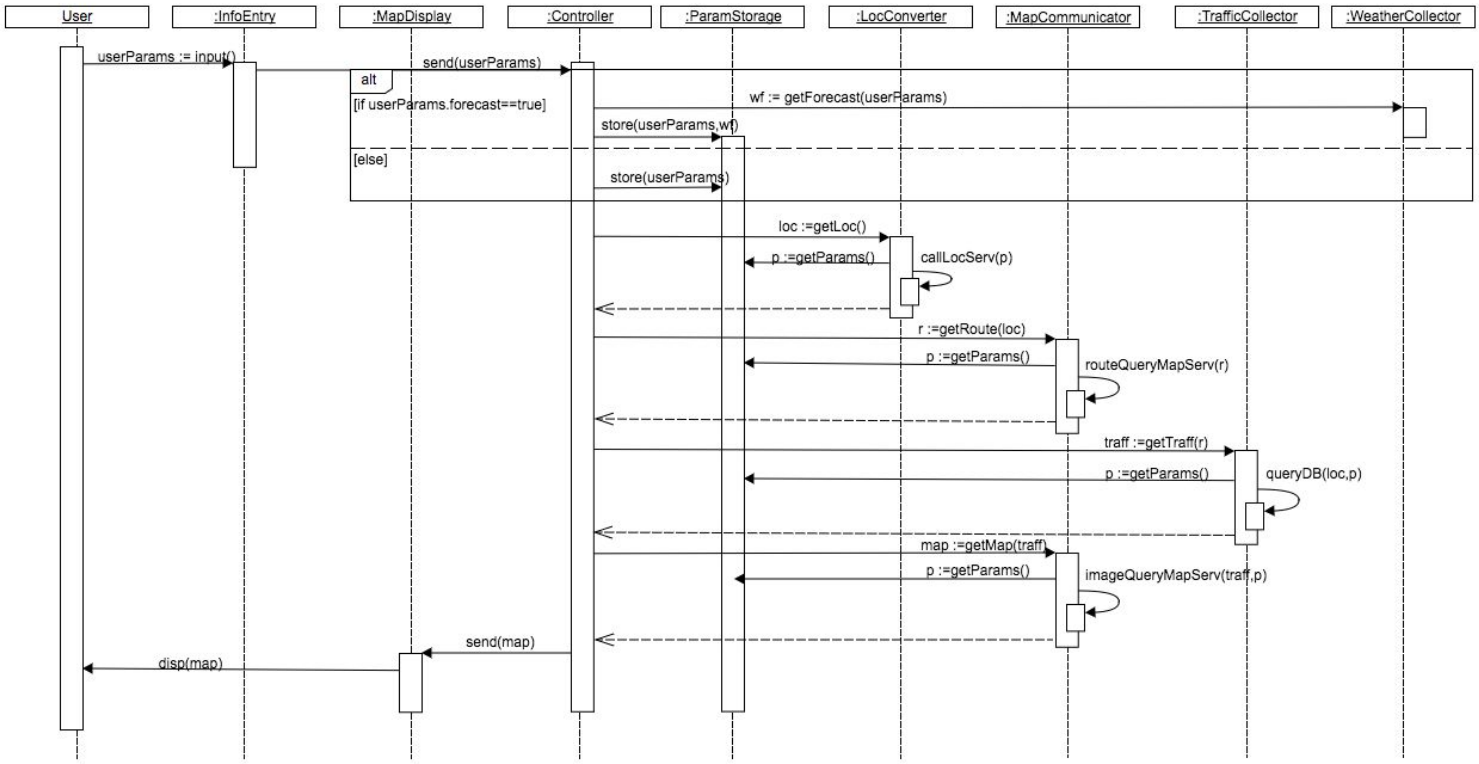

Figure 2: Sequence Diagram of UC-2 with extension UC-6

This diagram incorporates both UC-2, the “Route” feature, and its extension UC-6. This diagram is similar to Figure 1 above. The components used are all the same, though the order in which they are accessed is different as well as the function calls that are required to implement the route feature. As a result, the design principles mentioned in the description for Figure 1 still apply here. This diagram is initiated from after the point the user chooses the “Route” feature on the landing page and begins to input parameters.Both use cases use the same conceptual objects, which is by design to make our concepts reusable since both features have similar functions.

Alternative Design

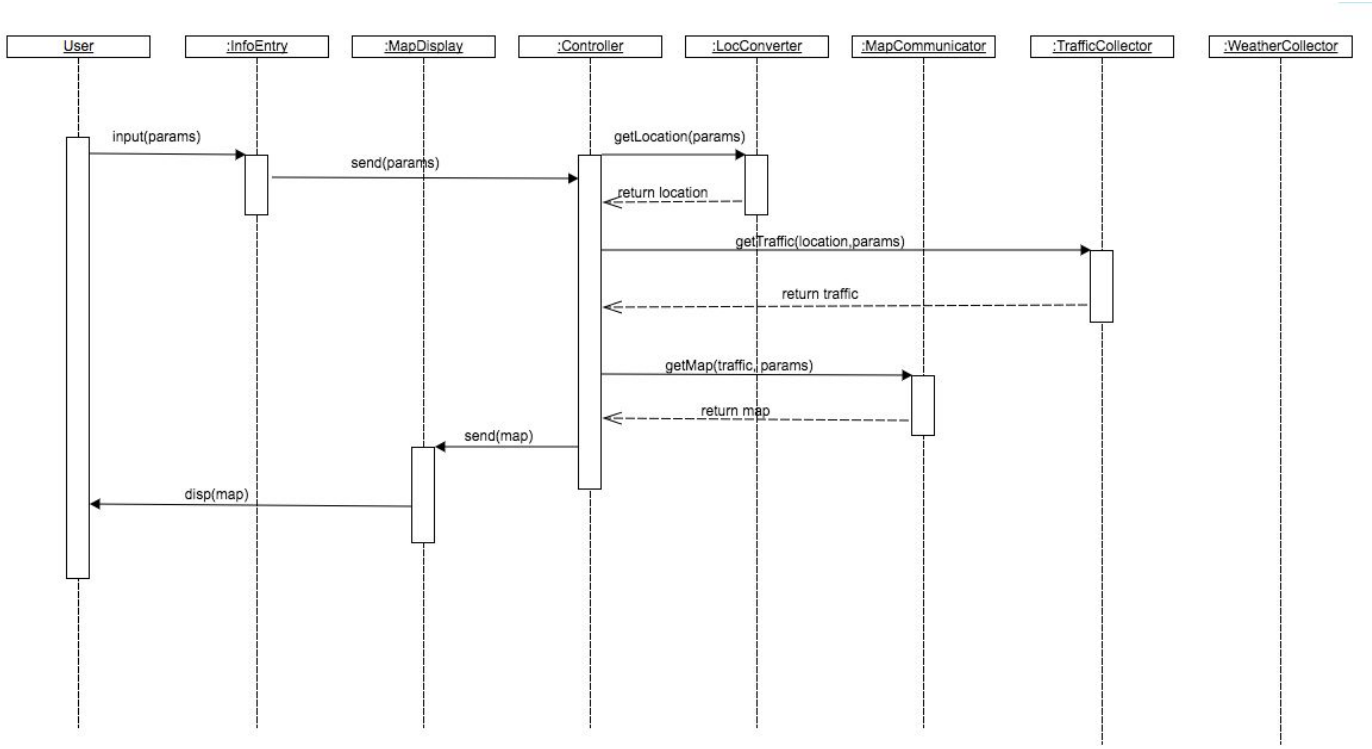

Figure 3: Alternate Design Diagram

This diagram was the first design we came up with for the UC-1 interaction diagram. In this diagram, InfoEntry sends the user inputted parameters to the controller, which stores these parameters for use by the other components. We decided against this because we felt that it was non-cohesive to have the Controller concept also in charge of info storage. This is shown in our final design for the interaction diagram (Figure 1), where the concept ParamStorage receives the user inputs and stores them. We also found this necessary because not all of the parameters that are used come from the user, such as in UC-6. The addition of the ParamStorage concept allows us to resolve parameters of different origin into one easy-to-access location. The final design is, as a result, more cohesive because the Controller is now no longer responsible for keeping track of parameter info.

Class Diagram and Interface Specification

Class Diagram

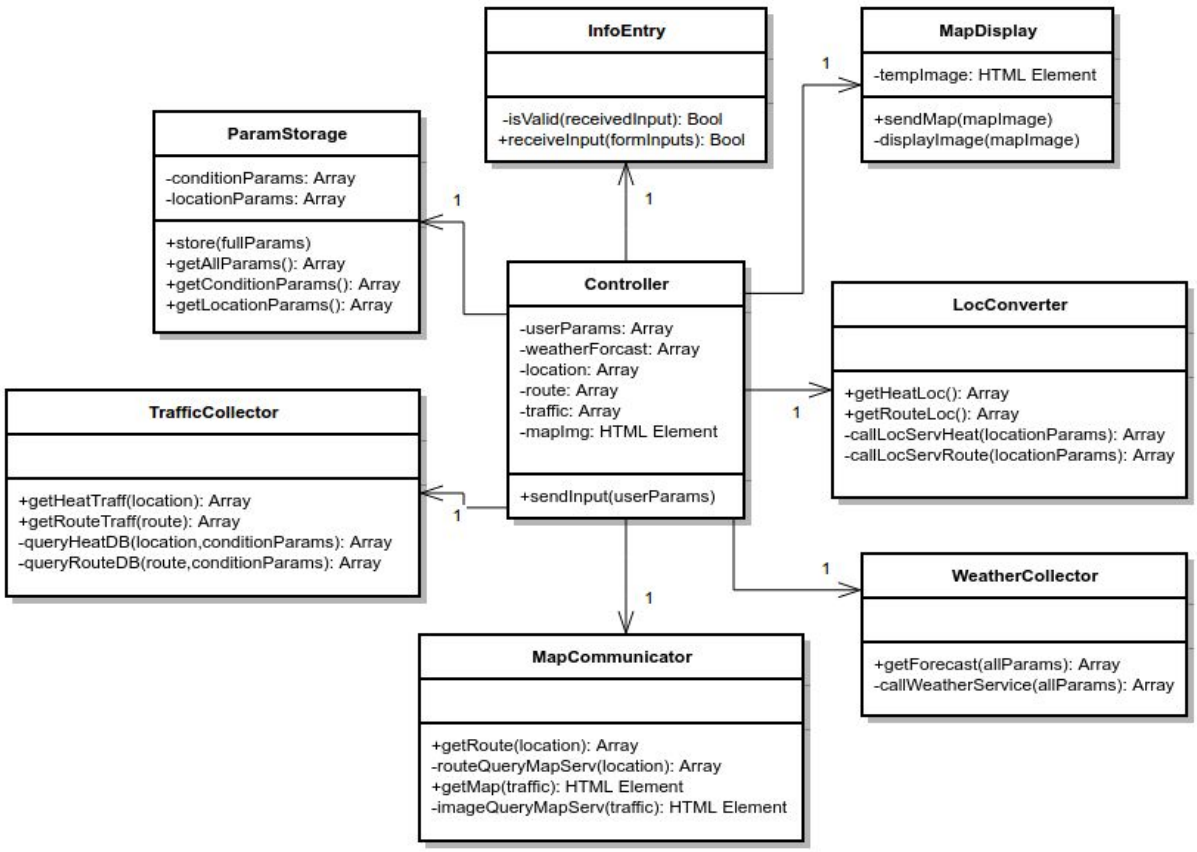

Figure 4: Class Diagram

This is the class diagram for our application. 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 (perhaps for yet-to-be-implemented use cases). 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 get functions of the TrafficCollector to retrieve this information and pass it back to the Map Communicator class.

Data Types and Operation Signatures

InfoEntry: Class that is responsible for handling parameter entry for the heat map or route feature. Responsible for validating inputs before sending them to the controller.

Operations:

- **-isValid**(*receivedInput: Array*): Bool
  - Checks the user's input for validity. If the entries are valid, it returns true, otherwise it returns false. Parameter receivedInput is an array that contains all the user's inputs.
- **+receiveInput**(*formInputs: Array*): Bool
  - Returns true if the user successfully completed the form to generate a heat map or route, otherwise it returns false. Parameter formInputs is an array that contains all the user's inputs.

MapDisplay: Class that is responsible for generating the map with proper highlights and markers and displaying it to the user.

Attributes:

- **-tempImage**: HTML Element
  - Element provided by the Google Maps API that allows the application to embed the map GUI.

Operations:

- **+sendMap**(*mapImage: HTML Element*)
  - Takes in the HTML Element for the map as a parameter and uses that to set up the map that will be displayed to the user.
- **-displayImage**(*mapImage: HTML Element*)
  - This function gets called by the sendMap() function and actually displays the map to the user. Its input parameter is the HTML Element for the map.

ParamStorage: Class that stores all parameters obtained from the user and from API calls and provides them to functions that need them.

Attributes:

- **-conditionParams**: Array
  - Array of parameters about the traffic and weather conditions for the required locations.
- **-locationParams**: Array
  - Array of parameters about the location that the user needs data for. For the Route feature, this could be starting and ending coordinates. For the Heat Map feature, this could be a zip code and bounding box.

Operations:

- **+store**(*fullParams: Array*)
  - Function that takes all the parameters and divides them up into location parameters and condition parameters then sets the appropriate attributes. The parameter fullParams is an array of all the parameters obtained from the user and API calls.
- **+getAllParams**(): Array
  - Returns an array of all parameters as an array.
- **+getConditionParams**(): Array
  - Returns only the parameters associated with date, time, and optionally weather as an array.
- **+getLocationParams**(): Array
  - Returns only the parameters associated with desired locations.

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

Attributes:

- **-userParams**: Array
  - The portion of parameters specified by the user. In some use cases, these are all parameters used. In other use cases, these are combined with parameters generated by participating actors.
- **-weatherForecast**: Array
  - An array of forecast info generated by the WeatherCollector.
- **-location**: Array
  - A bounding box of lat/long values stored in an array
- **-route**: Array
  - An array of routes that connect two locations.
- **-traffic**: Array
  - Traffic severities in an array
- **-mapImg**: HTML Element
  - Google Maps GUI element, with traffic severities highlighted

Operations:

- **+sendInput**(*userParams: Array*)
  - Acquires user parameters from the client side.

LocConverter: Responsible for taking a user's input parameters for location such as a zip code

Operations:

- **+getHeatLoc**(): Array
  - Getter function, returns a lat/long bounding box in an array.
- **+getRouteLoc**(): Array
  - Getter function, returns an array of latitudes and longitudes corresponding to the user-inputted start and end coordinates
- **-callLocServHeat**(*locationParams: Array*): Array
  - Helper function for getHeatLoc(). Takes the user-inputted zip code, and uses it to make an API call to the Location Service. Returns a lat/long bounding box in an array.
- **-callLocServRoute**(*locationParams: Array*): Array
  - Helper function for getRouteLoc(). Takes the user-inputted start and end coordinates, and uses them to make an API call to the Location Service. Returns lat/long formatted coordinates.

MapCommunicator: Responsible for acquiring routes and map images from the Mapping Service actor

Operations:

- **+getRoute**(*location: Array*): Array
  - Getter function, takes an array of 2 lat/long coordinates, returns an array of 3 routes that connect those coordinates.
- **-routeQueryMapServ**(*location: Array*): Array
  - Helper function for getRoute(). Makes API call to Mapping Service with input coordinates, returns array of routes connecting them.
- **+getMap**(*traffic: Array*): HTML Element
  - Getter function, takes an array of traffic severities, returns an HTML Element representing the Google Maps GUI displayed to the user.
- **-imageQueryMapServ**(*traffic: Array*): HTML Element
  - Helper function for getMap(). Makes API call to Mapping Service, returns GUI element with all the traffic severities highlighted

WeatherCollector: Responsible for acquiring the weather forecast from the Weather Service.

Operations:

- **+getForecast**(*allParams: Array*): Array
  - Getter function, takes an array of date/time info as parameters, outputs array of strings that detail weather forecast
- **-callWeatherService**(*allParams: Array*): Array
  - Helper function used by getForecast(). Makes an API call to the weather service, and formats the API's JSON response into an array of forecast info

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

Operations:

- **+getHeatTraff**(*location: Array*): Array
  - Getter function, takes in a location bounding box as input, returns an array of traffic severities of roads inside that bounding box.
- **+getRouteTraff**(*route: Array*): Array
  - Getter function, takes in an array of routes, returns an array of traffic severities for all roads in the routes.
- **-queryHeatDB**(*location: Array,conditionParams: Array*): Array
  - Helper function for getHeatTraff(). Takes a location bounding box and array of weather, date, and time conditions as parameters. Queries the DB for all traffic severities that match those params, returns those severities as array.
- **-queryRouteDB**(*route: Array,conditionParams: Array*): Array
  - Helper function for getRouteTraff(). Takes an array of routes and array of weather, date, and time conditions as parameters. Queries the DB for all traffic severities that match those params, returns those severities as array.

Traceability Matrix

Classes:	Controller	InfoEntry	ParamStorage	LocConverter	TrafficCollector	MapCommunicator	WeatherCollector	MapDisplay
Domain Concepts								
Controller	x							
Info Entry		x						
Parameter Storage			x					
Location Converter				x				
Traffic Collector					x			
Map Communicator						x		
Weather Collector							x	
Map Display								x

Figure 5: 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.

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

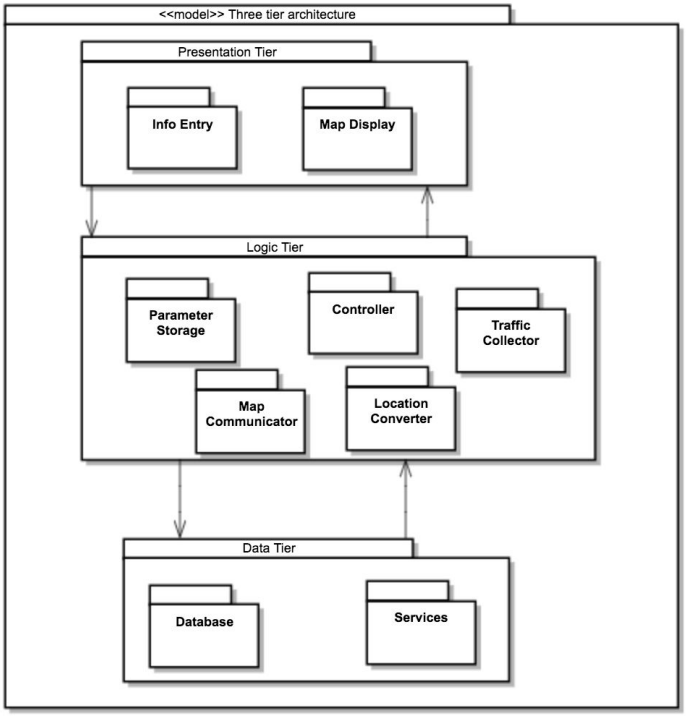

Figure 6: Package Diagram

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 Figure 6. 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 Info Entry and Map Display 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, Parameter Storage, Location Converter, Map Communicator and Traffic Collector. These domain concepts handle the application's logic and information processing. Finally, in the Data Tier subsystem, two generalized packages - Database, and Services - are depicted to represent the specific functionalities of the Data Tier that are based on the participating actors, where the Database is the participating actor itself and the Services package includes Mapping Service and Weather 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.

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. Along with a graphic of each table is an accompanied description. 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 7: 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 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.com 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).

#	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	gridId	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
3	roadName	varchar(255)	latin1_general_ci		No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
4	day	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
5	hour	time			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
6	sev_clear	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
7	sev_snow	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
8	sev_cloudy	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
9	sev_rain	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values
10	sev_fog	int(11)			No	None		Change Drop Primary Unique Index Spatial Fulltext Distinct values

Figure 8: Severity Table

The severity table above is responsible for storing the sum 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

Figure 9: 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 to a zip code by using the latitude and longitude of the center of the box use the geocoding API to convert it to a zip code 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 10: Frequency Table

The above frequency table is responsible for storing the the frequency of all weather conditions for each Grid-Box for every hour each day of the week. In order to query the weather.com API to get weather information, a zip code is needed. To make sure that the API query limit was not hit, we decided to only collect weather for certain zip codes. The zip codes of each state all begin with a 3 digit prefix that specifies a general region that contains all the zip codes beginning with said prefix. An assumption that we made was that the weather for each pre-fixed region will be roughly the same. Therefore, we decided to collect the weather of one zip code per prefix. 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.

Network Protocol

A HTTP protocol will be used to access “Google Maps” for the map images displayed on the website. The HTTP protocol was suggested because it is the most commonly used protocol and it is a standard protocol used in any browser space. An HTTP request to read (GET) and one to write (POST) to the web domain will be necessary for acquiring images for the user’s inputs. However, we are still in the early stages of implementation and will decide on the most appropriate method for acquiring the map images at a later time.

Global Control Flow

Execution order:

Our system is both procedure-driven and event-driven. The system is procedure-driven because the user can request to use any of the features of the application at any time. The user navigates to the website and follows a similar procedure in terms of how to use any of the features presented on the website. The navigation of the website as well as inputting information, therefore, follows a linear fashion. The system is also event driven because once the event of the user entering relevant information into a specific feature takes place, the Presentation Tier will communicate with the Logic Tier which will then communicate with the

Data Tier and finally the user will receive their information. The weather and traffic receivers have therefore taken available data from websites and stored it so that the program can properly present the requested information upon the *event* of the user pressing “go” (with the appropriate inputs). While the user follows a similar procedure each time the use Onward, the procedure will produce no output until the user “presses go”. This *event* of “pressing go” is crucial for the system to display any outputs and for the user’s procedure to interact with the system. Since this event of “pressing go” is key in producing output, we can say the system is also event-driven.

Time Dependency:
This traffic monitoring system will have parts that will operate in real time, and those that will operate based on event-response time. The program contains features that are real-time in that they compute and process information when the user selects options to view statistics along a route or statistics for a region. The way in which our database is set up, a lot of these calculations are done at the time of data collection, so many of these real-time calculations are compiling what is already there and will help to create a quick and easy experience for the user. These features will include any user interaction with the map. This includes zooming in and out, or moving the map left and right. The sliding bar for selecting the radius also operates in real time, as the user slides it to select different radii, the map will accurately represent this change. The remaining features ie. heat map, route, and agenda features will be computed/displayed once the user continues with the “event” of pressing the “go” button.
It should be noted that there will be a small time delay between when the user request is entered and when the final processed display is presented. The timers in the system are implemented for whether and traffic receivers, which take available data from websites and store it into the database. However, this is not a constraint on the system.

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, Python with standard libraries and Apache HTTP server.

Project Management

Merging the Contributions from Individual Team Members

While collaborating on the report it sometimes was difficult to blend writing styles of all six group members. Often times several different ways of writing would conflict and interrupt the cohesiveness and coherence of the section or sections of the report. In order to cooperate all the ideas of the group, it was often more beneficial to let all members write their ideas and then later work to edit the document so that it would smoothly transition from one response to the next. The group took turns editing the document at different moments so that each member could check the coherency of the document as it came together. Another problem that we encountered was the different conventions for labeling figures and diagrams and its respective elements. We dealt with this issue by coming to common conventions of detailing figures and diagrams that would most effectively work for the group in both the reports and in the implementation phase. By doing this we were able to ensure consistency with how we communicated our ideas to those reading our report.

Project Coordination and Progress Report

Currently the group is implementing UC-1 and UC-2. The group is working on tackling these two use cases are currently being worked on by individual subteams 1 and 2. Each subteam has split the priorities and responsibilities for these two use cases. The database used for data collection has already been created and is functional in terms of properly collecting data for weather and traffic incidents. This database was created by all members of the group and will be later used properly in the use cases. Some relevant project management activities include keeping a schedule that accounts for the report submission dates, the demo dates, and necessary implementation due dates. This schedule allows for every group member to aware of the specific work that needs to be completed by a specific date so that each group member can plan his/her schedule and complete the work by the designated time.

Plan of Work

Our team is working according to the Gantt Chart below.

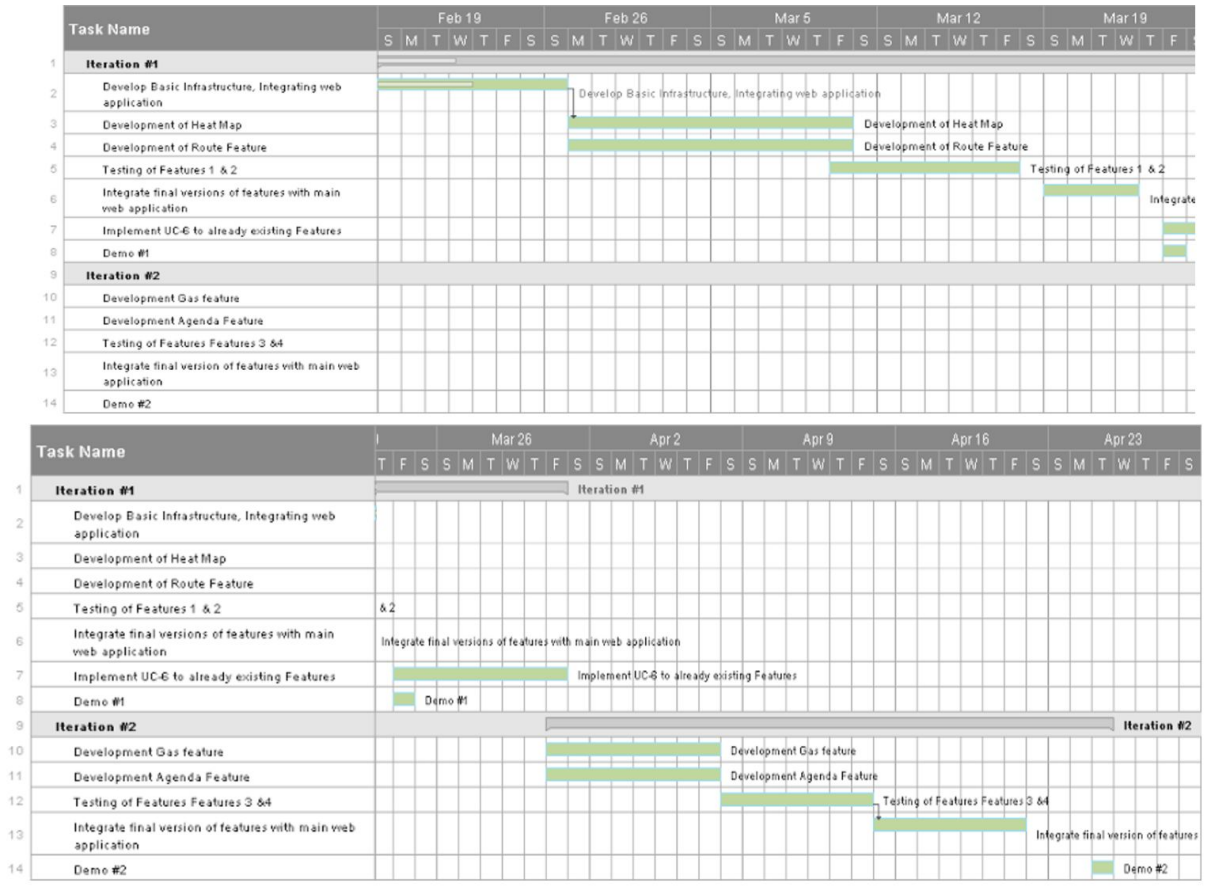

Figure 11: Gantt Chart

Breakdown of Responsibilities

Heat Map Feature:
The Heat map was discussed and the features within it were formulated and thoroughly worked on by all members of the group. The development of this feature will be split up evenly between Brian, Ridwan and Mhammed. They will be equally responsible for coding and testing this feature. It is currently premature to say which specific modules and classes will be worked on by each member.

Route Feature:
The Route Feature was discussed and the features within it were formulated and thoroughly worked on by all members of the group. The development of this feature will be split up evenly

between Lauren, Shubhra, and Sean. They will be equally responsible for coding and testing this feature. It is currently premature to say which specific modules and classes will be worked on by each member.

Sean will be coordinating the integration of the two features. He is the one who approves pull and merge requests from the branches. He will be the one with the ownership of the master branch and make sure that both the Heat Map and Route branches are merged to it.

Each individual team will test to make sure that their feature works in the program as a whole. Once the Heat Map Feature and the Route Feature are fully implemented, it will then be Ridwan's task to test whether the two parts work together as a cohesive unit. Each team will perform the integration tests for the first demonstration, and Ridwan will complete integration tests on the web platform as a whole. These integration tests will also be completed as each team completes the various aspects of their implementation. Since both of these features have similar methods of implementations, and as a result have overlapping concepts, both sub teams will also work to make sure that their contributions to the concepts/classes do not interfere and cause merge conflicts. The sub teams will test how each aspect of the project integrates with the project as a whole ie. with the web platform. However, we would also like to run integration tests on both features once completed and how these features act under user interaction with the web platform.

References

- Bing Traffic API - *Bing Developer Network.* 2017. <https://msdn.microsoft.com/en-us/library/hh441725.aspx>
- Current Weather Data - *Weather Underground API*. 2017. <https://www.wunderground.com/weather/api/>
- myGasFeed API - *JGSolutions.* 2010. <http://www.mygasfeed.com/keys/api>
- GeoCoding API - *Google. 2017.* <https://developers.google.com/maps/documentation/geocoding/start>

The above references are the current API's being used to collect data and that will be used for future implementation.