# **Technical Manual for Distance and Directions with Google Maps**

# By Greg Bayne, Kari Franklin, Michael Booser

**1. Overview**

This document describes how a system must be setup in order to be able to run the Google Maps program.

**2. System Requirements**

* Java version 1.4.0
* Memory 512 MB
* Free disk space 300 MB
* Processor speed 800 Mhz

**3. Assumptions**

* It is assumed the user has met all the minimum system requirements
* It is assumed the user has an active internet connection.

**4. Persistence**

* This will explain everything there is to know about the persistence pertaining to how data is being store for this project.

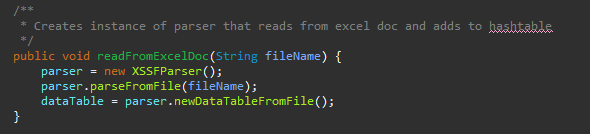
DataController.java

This class is the main class responsible for loading and saving data for the Distance and Directions application. It contains instances of all other classes in the package edu.sru.franklin. The class reads and writes the information to a file called ‘data.txt’. It does this using Java.IO standard classes, which allow you to write a serializable object straight to a file, and load the entire object the same way. This makes it easy to read and write the hashtable to the hard drive, because rather than read and write individual entries in the hashtable, the entire object is saved and loaded at the same time, meaning there is no need for loops.

The DataController uses the Table class to store the data in a hashTable.

DataController.readFromExcelDoc()

This method invokes the XSSFParser class, which can take a String path to a file as an argument. The XSSFParser loads the file, and the loaded data is put into the DataController’s Table. A detailed explanation of the XSSFParser can be found below.



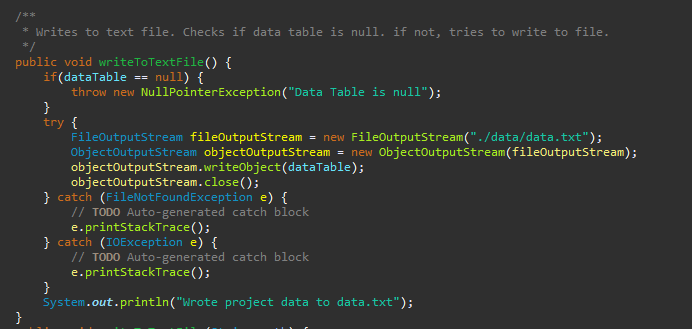
DataController.readFromTextFile()

This method tries to read a file into the program from the specified path. It does this by opening an objectStream, which is a Java IO standard class used to import Java.IO.Serialization objects into the code. Should the objectStream successfully find and open the specified file, in this case ‘data.txt’, it will load the contents of the file into a new Table object, which will then be populated with the contents of data.txt.



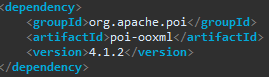
DataController.writeToTextFile()

This method also tries to open a objectStream to the file ‘data.txt’. Because the Table object is serializable, once the stream is open the program can write the entire Table object to the ‘data.txt’ file. This makes it unnecessary to read in individual entries from the save data, rather opting to load the whole object at once.



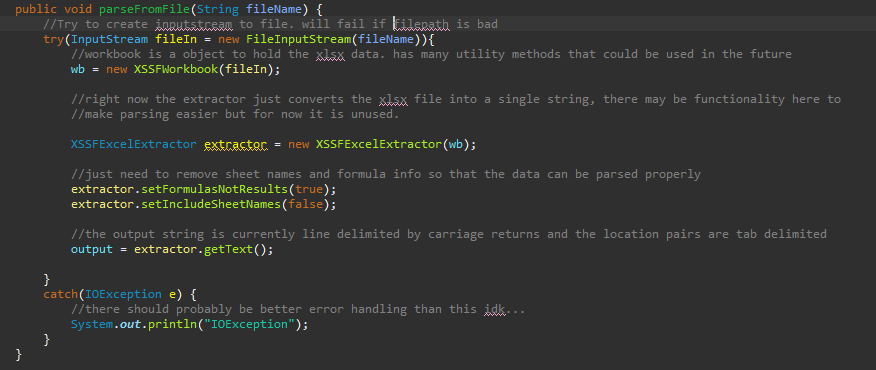
XSSFParser.java

The parser class is a implementation of the 3rd party package ‘Apache POI’. The Apache POI package includes many classes useful in manipulating Microsoft Excel documents, but only a few of them are used in the parse to load .xlsx files. You can find the implementation of Apachi POI in the pom file.



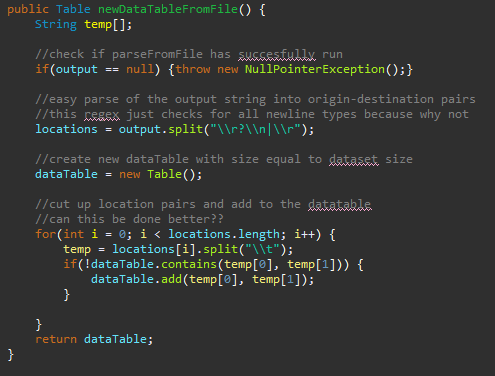
XSSFParser.parseFromFile()

This is one of two methods in the parser class, and is responsible for establishing a file connection to the specified path. It uses Apache POI’s custom data structure XSSFWorkbook to load and store the data from the file. Since we know the exact format of the file we are loading, ‘Addresses.xlsx’, beforehand, it is very easy to use another one of Apachi POI’s classes, XSSFExcelExtractor to extract the individual lines of text from the document and turn it into code-usable strings.



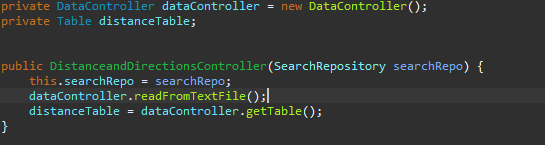
NewDataTableFromFile()

This method takes the strings generated by the parseFromFile() method and runs them through a simple regex. Since we know that the output from the Apachi POI is delimited by tabs and returns, we can use a regex to split it down further, into individual addresses. These addresses are then added to a new hashTable, which is returned by the method.



Code Implementation in DistanceAndDirectionsController()

In the application’s primary class, an instance of the DataController class is used for its save load functionality. For every new query, the DataController instance is polled to see if its internal hashTable already contains the query. If it is not, the DistanceAndDirectionsController.newQuery() method will ask the Google API for the information, add the new query to the DataController’s hash table, then instruct the Datacontroller to save to the ‘data.txt’ file. This means that every time a new query is added, it is immediately saved to the disc for persistence even after the program is closed.





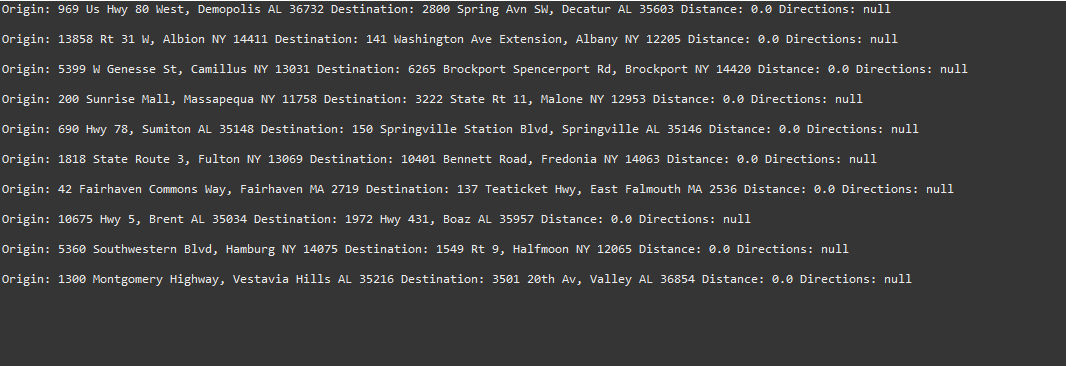


Test Classes

The edu.sru.franklin package has two test classes, SaveTests and FranklinTests.

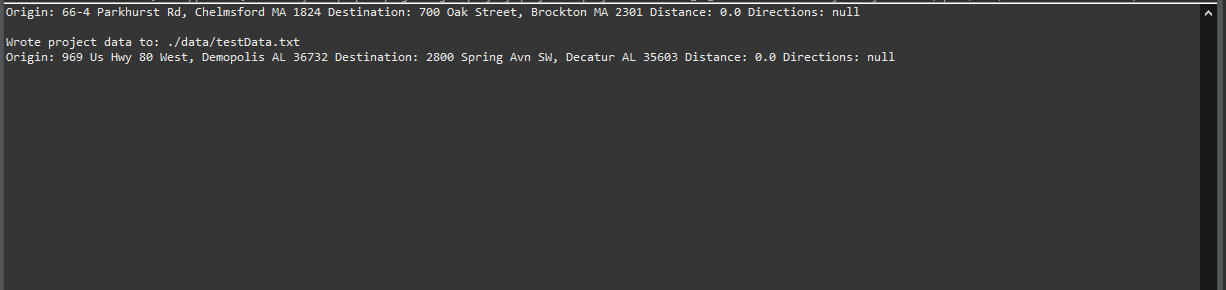
Savetests tests the save/load functionality of the DataController by preforming a save and a load operation, each time printing out a portion of the hashtable to prove that it has loaded and saved.

Sample output:



FranklinTests tests both the functionality of the XSSFParser and the load/save to data.txt

Sample output:



//More about other code such as gregs or michaels??

4. Running Google Maps //Do I even need this??

**5. Google Maps Classes**

**CreateMap.java**- This class builds URL with given address data (Origin and Destination) to create static map with google API. The URL includes markers for start and end locations.

**DatabaseQuery.java**- Queries coordinate data from provided ID.

**DistanceAndDirectionsApplications.java**- Application class to launch website on localhost:8080

**GeoGrabber.java**- Handles geocoding and reverse-geocoding events.

**DistanceAndDirectionsController.java**- The controller class for handling interactions between the SeachRepository (CRUD Repository) and corresponding “search” entities.

**PolyCluster.java**- Used to organize each set of long/lat coordinates into clusters.

**Search.java**- Depicting object model fro query data stored in the H2 database.

**SearchRepository.java**- Repository implementation extending from CRUD repository.

**DataStore.java**- Holds data for map queries, stores the starting and ending locations as well as the distance and DirectionsHolder.

**DirectionsAPI.java**- Generates a request for the Google Directions API and parses the response.

**DirectionsHolder.java**- A class that holds parsed information from Google Directions API including the directions summary.

**DistanceMatrixAPI.java**- Used for obtaining distance using Google Maps API.

**API\_Direcitons.java**- Generates a request for the google directions API and parses the response.

**API\_DistanceMatrix.java**- Used for obtaining distance using Google Maps API.

**API\_GeoCode.java**- This class handles geocoding and reverse-geocoding events.

**DataStore.java (GoogleAPIRoadsData)**- Holds data for map queries, stores the staring the ending locations as well as the distance and DirectionsHolder.

**DirectionsHolder.java**- Holds parsed information from Google Directions API including the directions summary.

**KEYS.java**- Used for accessing and editing API keys in GoogleAPIRoadsData.

**DataController.java**- Runs the xlsx parser and adds its data to a Table object.

**DataObject.java**- It is a container for all the data to be stored.

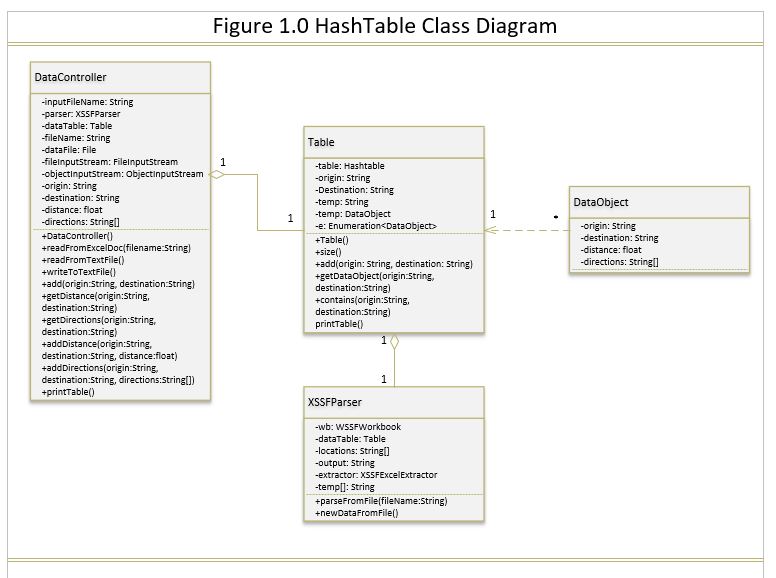
**Table.java**- Used to make an instance of a Hashtable. This holds the data structure that we are using to store data.

**XSSFParser.java**- Uses a third party library called Apache POI that is used to manipulate .xlsx files.

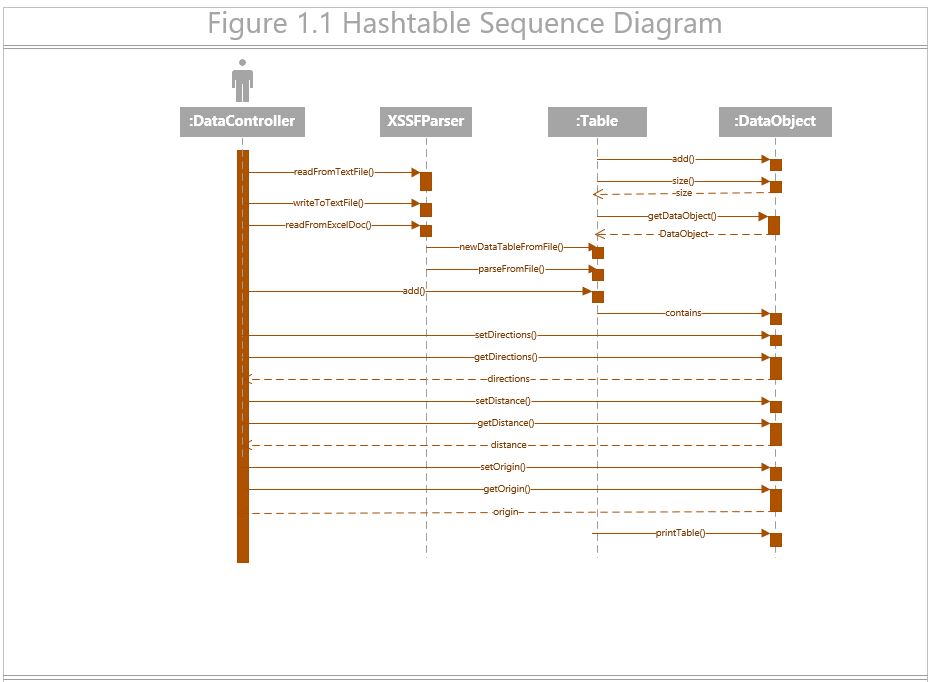
**6. Interactions between Google Maps Classes**

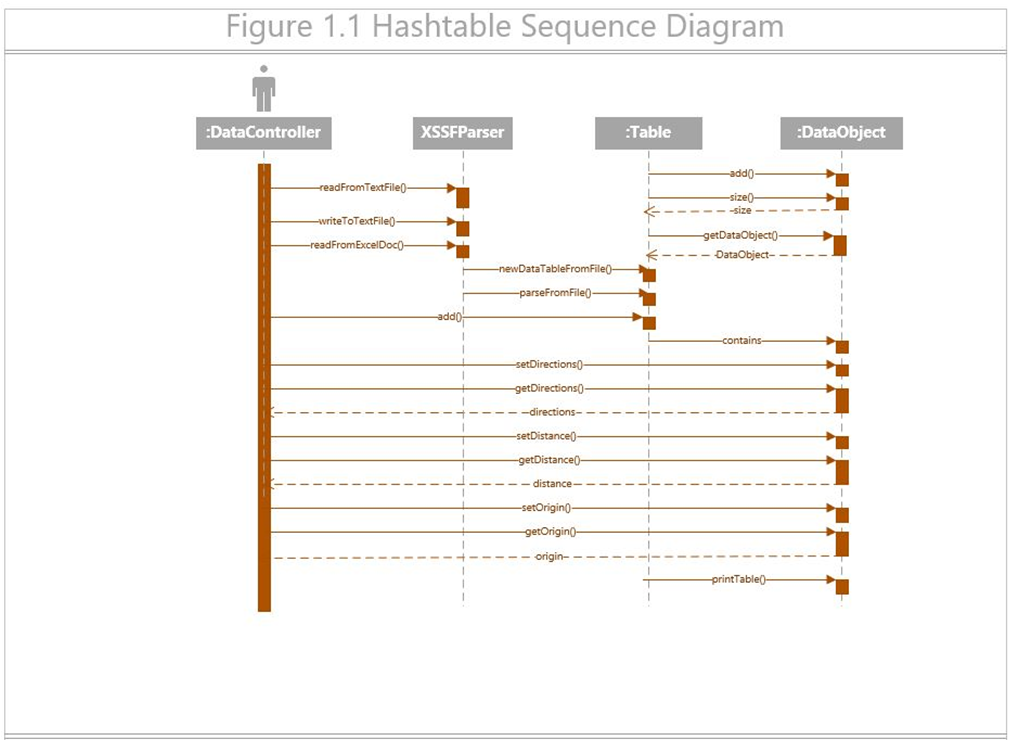
7. UML Diagrams on functionality of Google Maps

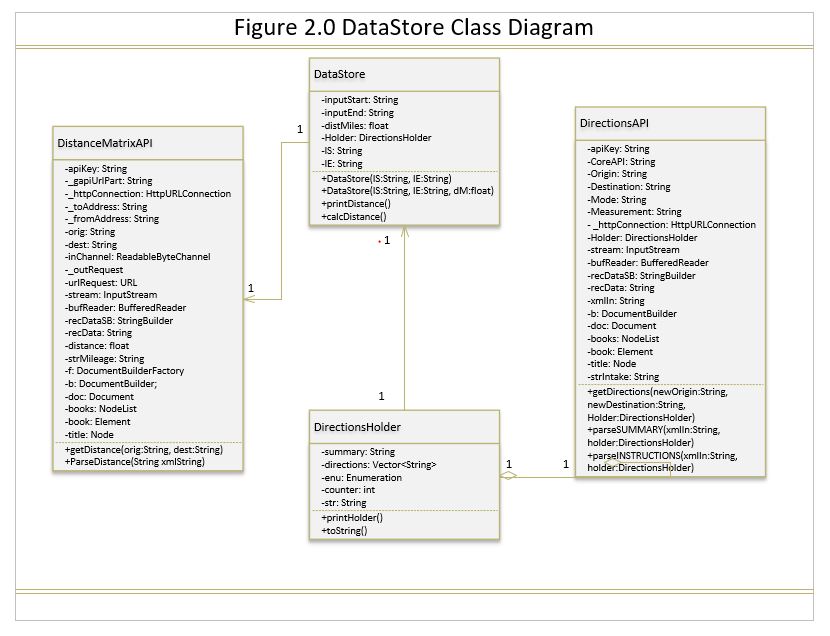
The following UML diagrams will show the interaction between different classes, how they are being used, and their functionality together as a whole.

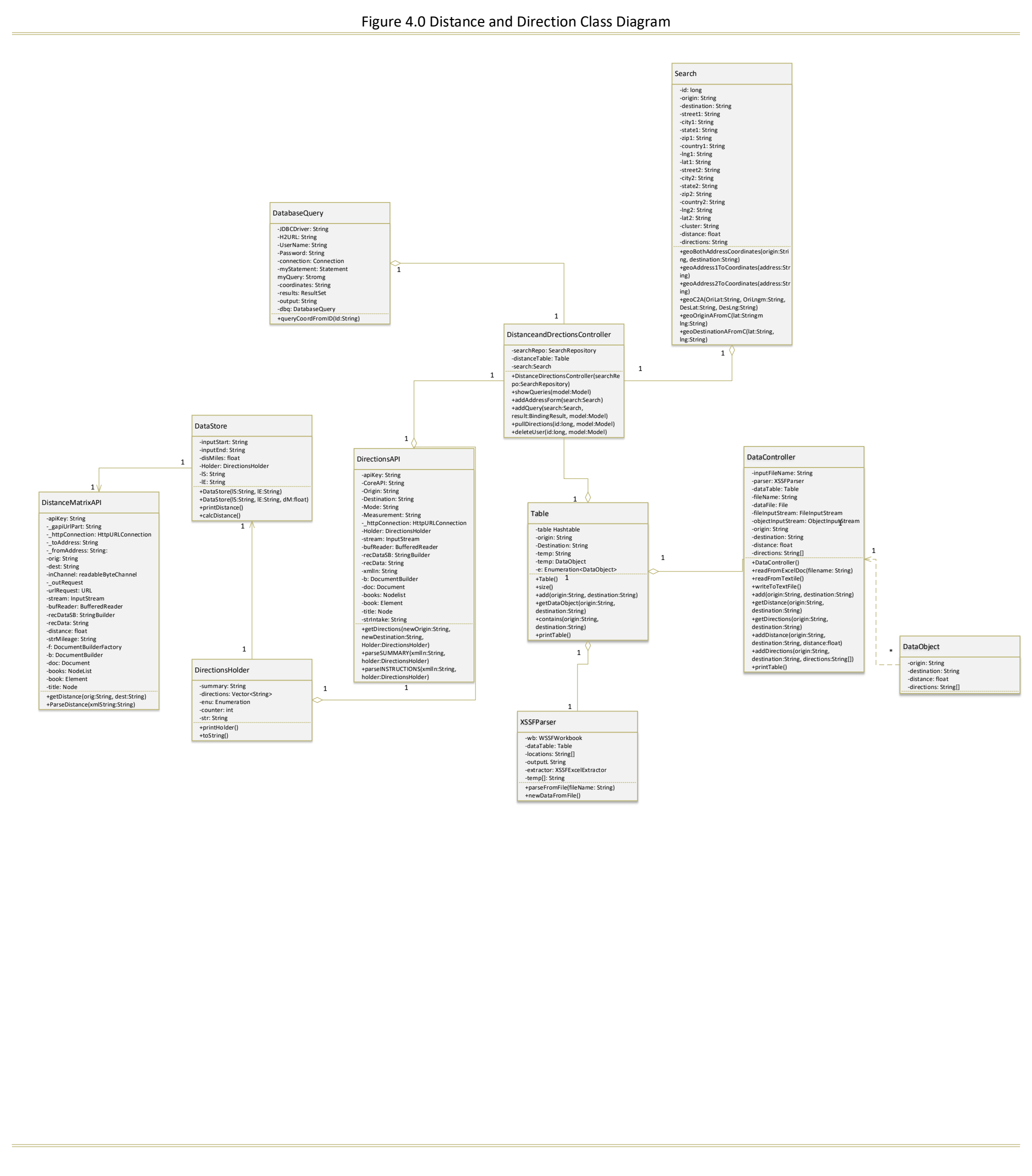


This UML class diagram shows the interactions of classes that are used to create and use the HashTable data structure. The Hashtable is what is used to load and save data for the project. There is a Table class that sets up the table, an XSSFParser that allows reading and writing to a text file. A DataObject that makes getters and setters for the Hashtable info and finally a DataController class that is used to implement all these classes together. Lastly, this shows the associations between these classes.

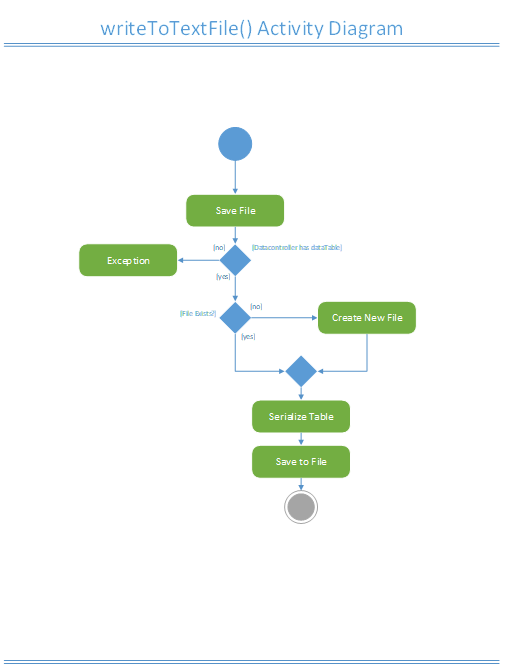


This UML sequence diagram shows the sequence of events existing between these four classes. This displays all the methods that are being used in each class as well as where they are being used and how they’re being used based on each class interaction. This diagram shows the sequence of information being used for the HashTable.

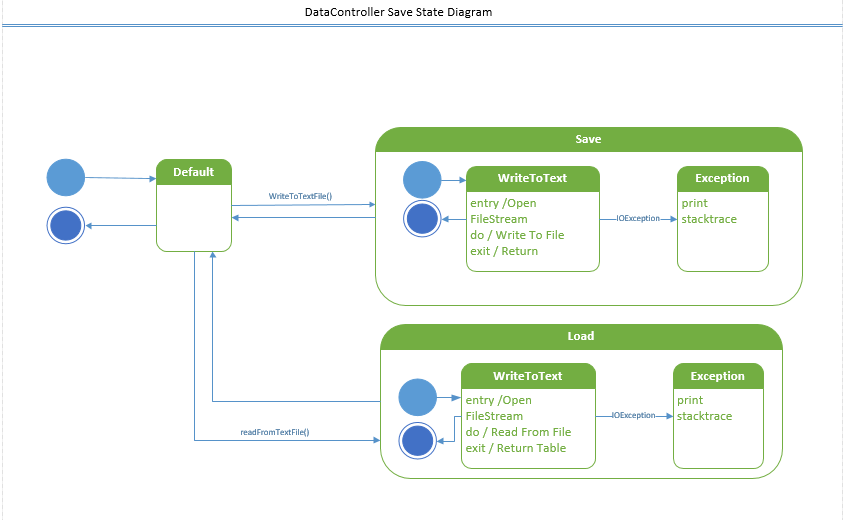




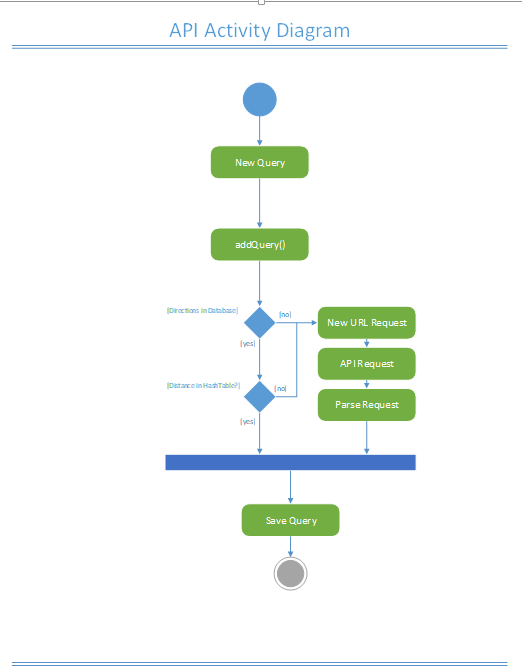
This UML class diagram shows the interactions of all the important classes in the program.



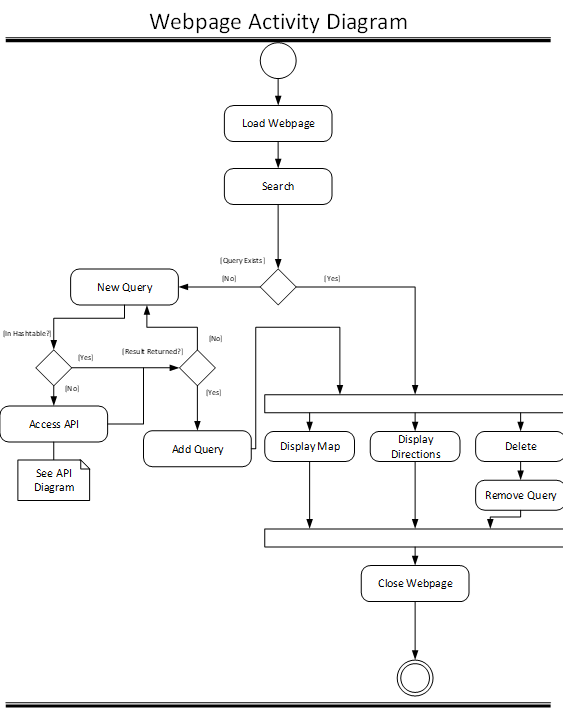
This is an activity UML diagrams that shows the different states involved in writing to a text file as well as the different steps towards saving it.



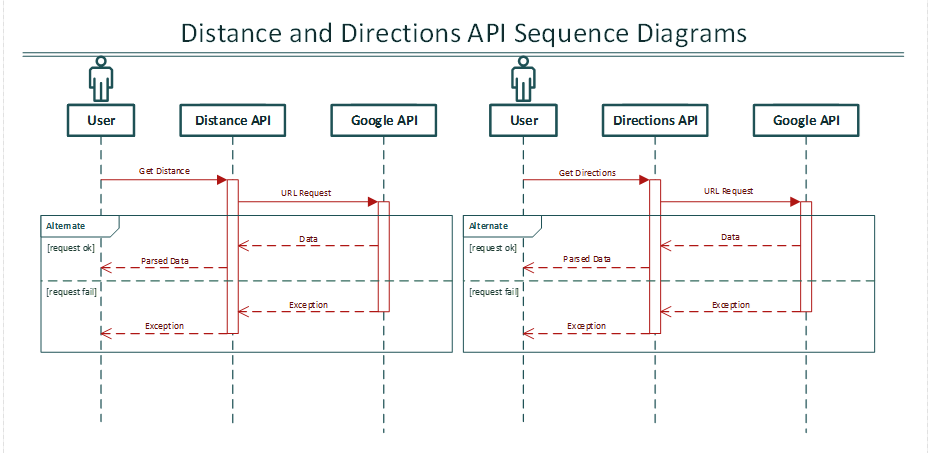
This is a state UML diagram that shows how the loading and saving works in the DataController class for the program.

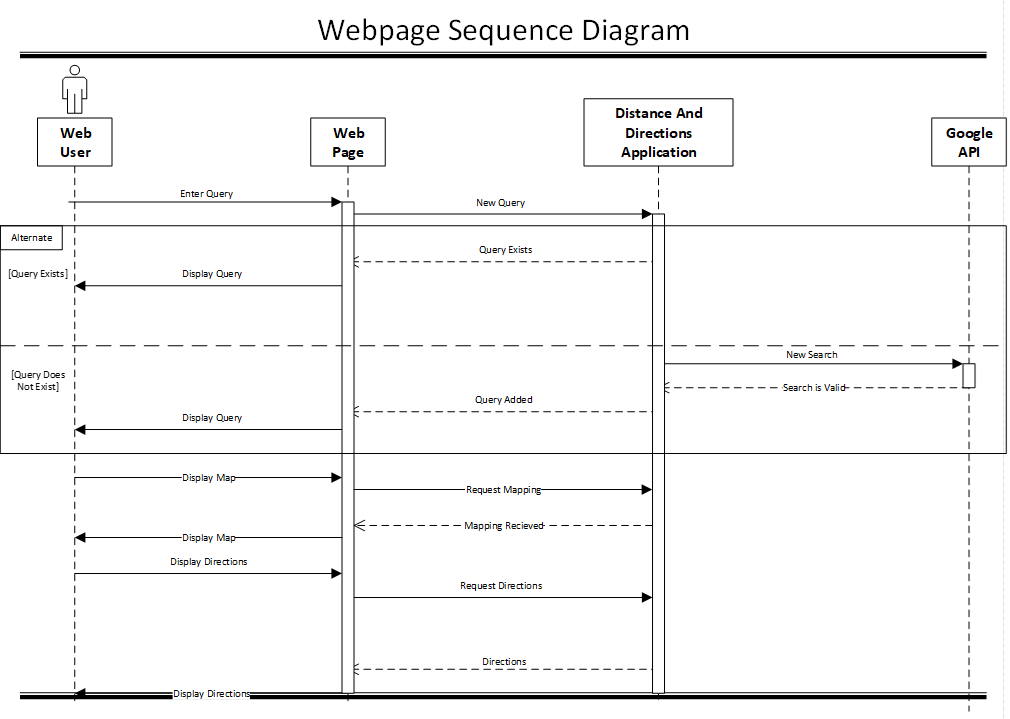


This is an activity UML diagram that shows the steps towards creating a new query and saving the query.

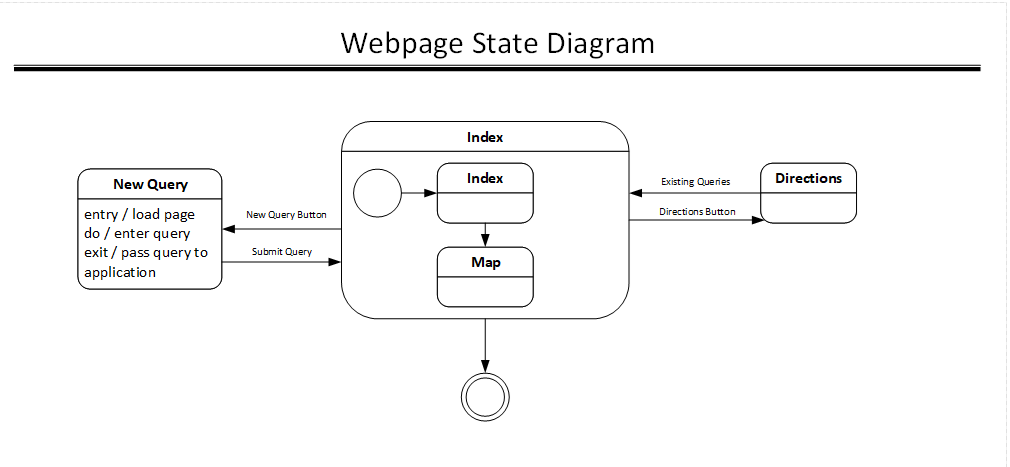


This is an activity UML diagram that shows the different flow of events for the webpage.

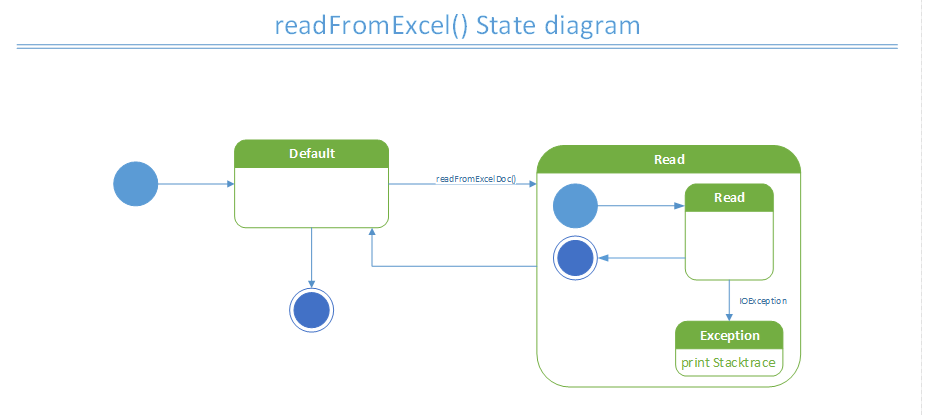
This is a UML sequence diagram that shows the parsing of data methods involved with the DirectionsAPI and the DistanceAPI classes.



This is a sequence UML diagram depicting the sequence of events involved with the operation of the webpage showing the different methods used.



This is a State UML diagram that shows the different states of the webpage and how it’s working.



This is a state UML diagram that is depicting how the readFromExcel() function is working.