Final thesis

Geoexplorer

by

Johan Hanssen Seferidis

LIU-IDA/LITH-EX-G--14/007--SE

2013-08-30
Geoexplorer
– A free open-source framework for black-box testing and scraping information from geographic services –
This is a report on the development of a free open-source framework. The framework is meant to be used to mainly black-box test and/or scrape information from a geographic service like Google Places, Facebook Places or Foursquare. In reality any service that is based on geographic coordinates can be used with the framework.

Amongst others, the framework offers functionalities like visualisation on-the-fly and logging of different aspects of the service. There are a few similar tools scattered on the world-wide web, but they usually are hard to find and if they are found, they either are not open-source, free or they lack in functionality. Another major drawback is that the available solutions are very generic, and thus limiting their capabilities. The work described here is an attempt for a concise, easy to use, extendible framework solely focused on geographic services.

In this report, the technologies used are demonstrated, while at the same time the reasons are given as to why a specific technology was selected in each case. Some documentation is also presented and a few references to the actual code-base in case someone wants to extend Geoexplorer or use it at their organization.
# Table of Contents

1 INTRODUCTION .................................................................................................................. 1

2 BACKGROUND ...................................................................................................................... 3
   2.1 THE PROBLEM ................................................................................................................. 3
      2.1.1 An example ............................................................................................................... 4
      2.1.2 Scalability issues ...................................................................................................... 4
      2.1.3 Behaviour Testing .................................................................................................... 5
      2.1.4 Visualisation .......................................................................................................... 6
   2.2 FUNCTIONAL REQUIREMENTS ....................................................................................... 6
   2.3 EXISTING SOLUTIONS .................................................................................................... 6

3 GEOINFORMATICS THEORY .............................................................................................. 8
   3.1 MAPS .............................................................................................................................. 8
      3.1.1 From sphere to flat surface .................................................................................... 8
      3.1.2 Mercator projection ............................................................................................... 10
      3.1.3 Geocoding ........................................................................................................... 12
      3.1.4 Reverse geocoding ............................................................................................... 13

4 METHOD ............................................................................................................................. 14
   4.1 NAME CONVENTIONS .................................................................................................... 14
   4.2 BACK-END STRUCTURE ............................................................................................... 14
      4.2.1 Scanner .................................................................................................................... 16
      4.2.2 GUI .......................................................................................................................... 17
      4.2.3 Grid .......................................................................................................................... 17
      4.2.4 Logger ....................................................................................................................... 18
      4.2.5 services .................................................................................................................... 19
   4.3 MAP COMPUTATIONS ..................................................................................................... 20
      4.3.1 Distance calculation ............................................................................................... 20
   4.4 VISUALISATION ............................................................................................................. 21
      4.4.1 Map .......................................................................................................................... 22
      4.4.2 Websockets ............................................................................................................ 22
      4.4.3 Communication between messengers ...................................................................... 23
      4.4.4 Threading ............................................................................................................... 24
   4.5 LATENCY .......................................................................................................................... 26
   4.6 EXTENDIBILITY ............................................................................................................. 26
      4.6.1 Adding a Service ..................................................................................................... 26
      4.6.2 API .......................................................................................................................... 28
      4.6.3 Extra tools .............................................................................................................. 29
   4.7 OUTPUT DATA ............................................................................................................... 30

5 RESULT ................................................................................................................................. 31

6 CONCLUSIONS .................................................................................................................... 33

7 FUTURE WORK .................................................................................................................... 35
   7.1 BACK-END .................................................................................................................... 35
   7.2 GUI ................................................................................................................................. 36

8 REFERENCES ........................................................................................................................ 39
List of Figures
Figure 1: Definition of geographic services.................................................................3
Figure 2: Search area limitation in Google Places...........................................................5
Figure 3: Earth Globe [17]..............................................................................................9
Figure 4: Google maps: 10x10 degree rectangles.............................................................10
Figure 5: Mercator projection [19]................................................................................11
Figure 6: Module hierarchy.............................................................................................15
Figure 7: Grid...................................................................................................................18
Figure 8: Application - GUI communication.................................................................21
Figure 9: Threads in Geoexplorer....................................................................................25
Figure 10: Screenshot from Geoexplorer's GUI...............................................................32
Figure 11: Screenshot of the Send button.......................................................................37

List of Tables
Table 1: Moving a vertical vector on the latitude axis.....................................................12
Table 2: Moving a horizontal vector on the latitude axis...............................................12
Table 3: Name conventions............................................................................................14
Table 4: Framework modules.........................................................................................15
Table 5: Framework classes...........................................................................................16
Table 6: Messages from client to server.........................................................................23
Table 7: Messages from server to client.........................................................................24
Table 8: API: box object...............................................................................................28
Table 9: API: logger object............................................................................................29
Table 10: API: geotools package....................................................................................30
1 Introduction

Since Google launched Google Maps, there is virtually no person who hasn't used a geographic service at least once in their lifetime. Google Maps, Google Places, Bing Places, Foursquare, Facebook Places are just a few widely-known examples of geographic services.

The popularity of such services is gaining momentum all the time and that is not a strange thing if we consider that a big portion of our life is about finding our ways to specific locations on a map.

While finding your way to the nearest pizzeria is very helpful, there is a bunch of other questions that can be answered by using geographic services as a medium. Such questions can be:

- How many pizzerias exist in New York?
- Which area of Netherlands has the least number of grocery stores?
- In which countries can Burger King be found?
- Is it Google Places, Facebook Places or Foursquare that has the best coverage over London's restaurants?
- I need all the gas-station positions in Sweden to jump-start my app.
- What are the limits of Foursquare or any other service?
- When does Foursquare or any other service start being non-responsive?

These questions highlight three major needs in the software community:

1. The need to use a geographic service in a bigger area than what the service was intended to. So instead of searching for restaurants in a radius of 1 km maybe I want to search for restaurants in a whole country or continent.

2. The need to collect statistical data for a geographic service. For example, seeing how many requests were sent, how much time it has taken for each request to reach its destination, etc.
1. Introduction

3. The need to collect the output of a geographic service. When a service is used, usually the results are being stored temporarily somewhere on the computer, not letting the user to re-use the results in the future.

All these questions and the three major points were the main motivation behind my thesis. By developing the Geoexplorer framework, someone can now answer all these questions and many more by just downloading the framework and using it with the geographic service of their preference.

The three major points described above, might imply a possible use of the framework for black-box testing, but by design Geoexplorer can offer much more. While someone can use it to test a service for its limitations and collect statistical data, someone else can use it as a scraping tool to collect information by using one or more services.
2 Background

2.1 The problem
There are many web services nowadays that offer GPS directions, weather forecasts, information about events and establishments in the proximity of the user's position, jobs, and more. All these services have something in common: they make use of geographic coordinates. More specifically each geographic service can be seen as a service that takes some parameters as input and outputs a set of geographic coordinates.

Some well-known such services is Google Places and Openstreet (http://www.openstreetmap.org), which are being widely used in a plethora of websites and smart-phone applications. There are also service mashups which combine different services to create new ones like Yahoo's YML [11] to name the latest and most popular.

Although all these services are widely being used for different purposes, by both freelance developers and companies, there is a limited number of tools to assist the user/developer in the pre-development step of research and design and the pre/post-development step of testing. The developer is usually only given an API and a few examples, but without any further assistance.

For Google services like Google Places, there is of course a huge documentation database with loads of examples. Still even such a very backed-up service, lacks a dedicated tool for geographic scalability testing, behaviour testing and visualisation.
2. Background

2.1.1 An example

Throughout the thesis, I will use Google Places Search service as a reference point, mostly because of its popularity and its publicly stated limitations [3][4]. For the reader to digest all the information, a brief explanation on how Google Places Search works is provided.

Google Places Search is a service which lets a client make queries to a Google server. Someone who wants to look for all the restaurants in Stockholm, would send the appropriate query to the Google server (in the form of a URL).

The response of a query like that, is a set of geographic points on a map. In this case, when we send a query to find all restaurants in Stockholm, the response would be a set of points, with each point being a restaurant. So essentially Google Places Search service is all about making a query to a huge database with all sorts of establishments (restaurants, groceries, pizzerias, etc.) in a global scale and getting their locations. However this service has a few limitations just like any other service.

2.1.2 Scalability issues

Generally, scalability is the increase in size of users. In this case however the term scalability is used to denote geographic scalability [12]. Simply put, geographic scalability is how good the service responds when we apply the same query to a larger area than before.

Google Places Search gives a response for a maximum radius of 50 km [4] (Figure 2). What that means, is that when we send a query to find all restaurants in Stockholm, the Google server will return the locations of restaurants included in a circle where; the centre is the centre of Stockholm and the radius is 50 km. For a better perspective on how “big” a circle like that is, see below (Figure 2).

Although the reasons for such limitations are not publicly made known, my belief is that in most cases they are due to performance issues.

A geographic limitation like the radius on Google Places Search is a hinder for testing the service on larger regions. What if the demand specifies a search for a whole country or a continent? Just like with everything, there is a work-around to just do that without getting error codes from a service.
2. Background

2.1.3 Behaviour Testing

A service is usually delivered, to a big extent, bug-free. A bug-free service however does not imply a bug-free usage by the user/developer. What this means is that a service might behave differently for different users.

An example is the Foursquare service (https://foursquare.com), a service similar to Google Places Search, which lets a user search for venues (restaurants, groceries, pizzerias, etc.) in a specific area. Foursquare seems to work fluently to the naked eye, but by using the framework (presented in this thesis) to test the service, some idiosyncrasies can be spotted. One such behaviour is the omitting of venues at specific locations and in specific search areas (in this service there is no apparent radius limitation).

In this case the framework acts as an **exploring tool**, for finding unpredicted behaviours. This is essentially black-box-testing.

By definition, black-box-testing is the testing of software without having any knowledge of the source code or inner workings of the software itself [13]. Black-box-testing can be further split into more specific testing domains like boundary testing,
performance testing, security testing and more. Still the term behaviour-testing is preferred as it seems a less deluging term which still keeps the semantics at place.

2.1.4 Visualisation

If a program is going to handle geographic coordinates then it's a big plus if the user can see the coordinates being manipulated in a map. For that reason from the beginning of Geoexplorer, I added google maps to the graphical interface. In that way, the user of Geoexplorer can see directly the results generated by the services.

Further more, the map has to be updated in **real-time** for every small update of a service or reruns of the service. This is mostly for getting a realistic idea of how the service is behaving.

2.2 Functional requirements

What is needed, is a framework which provides:

- Geographic scalability testing
- Behaviour testing
- Visualisation

The requirements for this problem are very specific and there exists no solution that fulfils them all while still following an open domain license (what exists, is presented next).

Besides the ultimate goals above, some additional requirements are needed in order to provide a reliable framework which follows good coding practices and is easy for developers to enhance:

- Logging
- Easy configuration
- Extensibility
- Documentation

2.3 Existing solutions

There are a few existing tools out there that can assist people who want to black-box test services like Google Places, Google Maps, Streetview, Foursquare, etc. All these tools though lack in some ways. One major drawback is that all the available tools are
meant to work with services in general instead of being specialized in geographic services.

The existing tools can be categorized in stand-alone tools and web-driven tools, meaning that the first ones run on a desktop computer while the latter ones run directly from the web.

I will describe first the web-driven tools. These tools are usually solutions scattered on web blog pages or small web sites. They do offer elementary capabilities, are bound to one or two services and not much can be configured. Many of them also come with a fee. An example is http://www.localleadmaximizer.com/blog/affordable-google-places-scraper which states a usage price of $47 (as of 1/04/2014). At the same time this tool does nothing more than just making usage of Google Places.

An other example is SniperBot found at http://sniperbotpro.com, which follows the same steps of the example above. The mare difference is that SniperBot offers more capabilities and supports more services. It still asks for a fee though and it can not be extended by adding custom services.

The second group, the stand-alone tools, have more similarities with Geoexplorer. Still they do lack here and there when they are compared to Geoexplorer.

WebSob [16] is a tool created at North Carolina University with the purpose to black-box test services (notice the general term “services” instead of “geographic services”). This solution seems over-complicated for the common user or programmer. Also the lack of direct linkage to the source-code makes it hard to make direct usage of the tool. At the same time the question of documentation remains on the table as there is no direct access to the source.

Another solution is WebInject [15]. WebInject is a testing framework for testing web services. While WebInject has an easy code-base and is more open than the above mentions, it still lacks in terms of documentation and is a bit hard to make more advanced testing of geographic services as it is based on the HTTP/Web layer and can't go any “deeper”.

There are many more solutions like the more popular Robot Framework (http://robotframework.org) but they all show the same problems. All these solutions are either very generic or specialized in an other field than geographic services. Many of them also do not provide documentation, source-code and might require a fee.

All this makes it apparent that there is a huge lack of free open-source (or closed-sourced) testing tools specialized to work with geographic services.
3 Geoinformatics theory

A few theory points in the field of geoinformatics are presented in this chapter to make it easier to dwell more in the context of the thesis. As the developed framework is based on geographic services, knowledge on how maps are represented and how geographic distances are calculated, can be insightful.

This information is not vital for comprehending the rest of the chapters and thus can be skipped if wished.

3.1 Maps

The visualisation of the framework is making usage of a map (Google Maps) so it is assumed a duty to provide a bit of explanations on how a map actually works, what type of calculations take place, the types of maps that exist and more.

3.1.1 From sphere to flat surface

There are two separate concepts to keep in mind when working with a map:

1. The actual earth globe
2. The projection of the globe (how earth is presented on a flat surface)

When we try to find the coordinates of a place or use other geographical tools involving latitudes and longitudes, it is a good practice to visualize the earth as a sphere. That is approximately the shape of the earth and all mathematical calculations are based on this assumption.

A map is merely the projection of the earth globe. Trying to apply mathematical formulas, comparing lines on the map, etc. are useless on the projection because the projection is not the actual object we started off with, but rather just a visual artefact that is made solely so that the globe can fit into our flat monitor screens (and paper maps of course).

On a side-note, in the programming field, there can usually be found external libraries that provide geographic calculations, conversions between geographic coordinate systems and more. In the framework presented in this thesis, some of these basic calculations had to be developed in the form of a library provided to the user of the framework.
Bellow is a screen shot from a Google map that can illustrate the fact that what we see is distorted (Google maps use a version of the Mercator projection [18] which is discussed later on).

The rectangles in Figure 4, are all rectangles of 10 degrees in longitude and 10 degrees in latitude. It can be seen clearly in this specific type of projection (Mercator projection) that:
1. Latitude and longitude lines are orthogonal to each other
2. Longitude lines are spaced equally from each other
3. Latitude lines are spaced more from each other the further away we are from the equator

From this it is apparent that this is very different from how the coordinate lines look in the actual globe sphere (Figure 3). There the latitude lines are spaced (almost) equally and the longitude lines are less spaced towards the poles. This makes it apparent that the projection and the actual globe sphere are two different entities and therefore should not be mixed in our minds.

3.1.2 Mercator projection

Google Maps is based on the famous Mercator projection [9][18] of the earth, or to be more precise a version of the Mercator projection. The Mercator projection, is the projection of the earth by taking the globe sphere and unwrapping it in to a two-dimensional sheet of paper (Figure 5).
A Mercator map has horizontal and vertical lines called latitude and longitude lines respectively. When we transform the spherical surface into the flat surface, we need to stretch the longitude lines, at the edges. That distorts the actual shape of the areas continents. The closer we are to the poles, the greater that distortion becomes apparent. The latitude is also stretched so that the latitude lines have more space between them the farer we are from the equator (central horizontal line).

Mercator map was developed to be used as a navigational map and thus choosing any arbitrary point A in the map and getting the direction to a point B, the line between the two points correspond to the actual direction from point A to point B.

The Mercator has for that reason, been used in naval navigation [14] and its popularity outside of navigation is a bit controversial because of the distortions found in this map. On the table below some examples are given that make these distortions more apparent with actual numbers.
In table 1 we have a vertical vector (line) of length 10 degrees, that we move up and down. The actual distance of the line remains constant. Furthermore if we cut the line in half (row 3), the actual distance gets divided in half.

In table 2 we have a horizontal vector (line) of length 10 degrees, that we move up and down (as on the previous example). When placed at different latitudes, the distances vary enormously. This is due to the stretch of the longitudes during the conversion of the sphere to a flat surface.

From the above tables we can draw two main conclusions:

1. **Vertical lines** of the same longitude length but on different latitudes on the map, will have the same length in meters.
2. **Horizontal lines** of the same longitude length but on different latitudes on the map, will have different length in meters.

In both conclusions we do not take into account changing the longitude where the line is placed as that has no effect in neither visual or actual distance.

### 3.1.3 Geocoding

As the framework presented in this thesis is based on geographic services, there are some minimum sets of functions that should be available to the user of the framework at any time. Calculating the distance between two points is one, geocoding is the other.
Geocoding is the process of taking a street address or other geographic identifier and from solely that, retrieving the coordinates of the position, which are in latitude and longitude.

Many problems arise when it comes to geocoding and using a two-dimensional visual representation of the earth. One very obvious problem is the fact that the earth is not a perfect sphere but rather an ellipsoid.

In practice this is not a problem as there are many online services which provide geocoding calculations. Such services can be found from big companies like Google but there are even open-source implementations like Open Street View [2].

3.1.4 Reverse geocoding

Reverse geocoding is the process of finding the identifier of a geographic place, given a latitude and longitude pair.

Most of the services that offer geocoding, also offer reverse geocoding.
4 Method

In this chapter, the approach to build the framework is described. Insight on the methods used to tackle the described problems in chapter 2.1 is provided, sometimes giving a bit of a deeper perspective if it is considered necessary.

To easier understand what is being discussed, the actual project can be checked online at [https://github.com/Pithikos/Geoexplorer](https://github.com/Pithikos/Geoexplorer) or even downloaded and run. All information about running it can be found in the README file at the same link.

4.1 Name conventions

For the ease of the framework's development, a few specific words have been used throughout the whole project. The definition of these words can be seen in Table 3 below. It's worth mentioning that these words are also being used for the configuration of the framework.

<table>
<thead>
<tr>
<th>Word</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>The instance from where the application has started until it has exited.</td>
</tr>
<tr>
<td>Scan</td>
<td>Each scan represents a query to the service being used. In the case of Google Places Search, a scan is a single query request sent to the Google server.</td>
</tr>
<tr>
<td>Results</td>
<td>The output information from each scan. In the case of Google Places Search, that is the response fetched back after a request was sent to the Google server.</td>
</tr>
<tr>
<td>Box</td>
<td>A rectangle representing a single scan.</td>
</tr>
</tbody>
</table>

4.2 Back-end structure

The back-end portion of the framework is structured in logical entities called modules (this follows Python conventions [5]). More specifically the modules found in the project are shown in the figure below.

Every module is a Python file, except services which is a folder.
In the figure above, the modules on top have access to the modules bellow them. That is also how the modules are associated with each other.

A synopsis as to what each module does, is provided in table 4. More extensive descriptions for all modules is provided in the sub-chapters 4.2.1 to 4.2.5.

**Table 4: Framework modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanner</td>
<td>Scanner is the main element of the application. Scanner uses a service to scan an area, has control of the GUI and keeps a Logger instance.</td>
</tr>
<tr>
<td>GUI</td>
<td>This is the graphic interface given to the application.</td>
</tr>
<tr>
<td>Grid</td>
<td>This is the whole area where scans are allowed.</td>
</tr>
<tr>
<td>Logger</td>
<td>Logger is just an element for logging events into files.</td>
</tr>
<tr>
<td>services</td>
<td>Service can be any service supported by the framework.</td>
</tr>
</tbody>
</table>

Now as the back-end part of the framework is coded in object-oriented Python, the code-base is essentially a set of classes organized in modules. In the beginning a more Java-like approach was taken by saving each class in its own file. However it became apparent that there would be too many files and the organisation would be not that efficient. Therefore during the development of the framework the project got reorganised so that modules came into play.

Every module contains tightly connected classes. For example the class Grid which is supposed to hold the areas where the scans take place, is made up of many instances of the class Box. Both classes are connected tightly as Grid makes direct usage of the
4. Method

class Box. This logical connection was the motivation behind the arrangement of the different classes in the appropriate modules. Below follows a table with the classes, with the modules they belong to and a brief description of each class.

Table 5: Framework classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td>Grid</td>
<td>A box is a rectangle. A grid holds many instances of Box.</td>
</tr>
<tr>
<td>Grid</td>
<td>Grid</td>
<td>The scanning grid seen on the map.</td>
</tr>
<tr>
<td>Messenger</td>
<td>GUI</td>
<td>A messenger is used to communicate with the map and generally the web GUI.</td>
</tr>
<tr>
<td>GUI</td>
<td>GUI</td>
<td>Graphical User Interface. Essentially an API for the map.</td>
</tr>
<tr>
<td>Logger</td>
<td>Logger</td>
<td>Logger that logs to files.</td>
</tr>
<tr>
<td>*</td>
<td>services</td>
<td>Holds all the services supported by the framework.</td>
</tr>
<tr>
<td>Scanner</td>
<td>Scanner</td>
<td>Scanner is the main element of the backend application.</td>
</tr>
</tbody>
</table>

The module services as stated earlier can have multiple classes. Each supported service is supposed to have its own class.

4.2.1 Scanner

The scanner is the main class that keeps everything glued together. Scanner itself has a singleton, that is an encapsulated instance of all the other modules. The reason for this, is that the scanner is supposed to be the brain of the application and for that reason, it should be the class that can access everything else.

A lot of work was put into making the scanner have a small code-base. The reasoning behind this, is that the scanner is the class that always has to change for every new change in the framework.

The scanner is running for the whole session of the application. The session terminates when the application is terminated. If someone wants to see the logical flow of the application, the Scanner is the place to start.
It is worth mentioning that the Scanner keeps updated variables throughout the session for the state of session, the areas scanned, the number of requests sent, the number of responses sent and other statistical information that can be informative to the end-user.

4.2.2 GUI

The GUI from the back-end perspective, is merely holding all the code that has to do with adding graphical elements.

This is essentially an API for the back-end application to update the map visualisation at the front-end. How this works and the role of the Messenger class in this is explained in much more detail in chapter 4.4 (Visualisation).

What is vital is to digest the idea that the GUI from the back-end prospective is merely an API used by the back-end of the framework and nothing more.

4.2.3 Grid

Grid is the data structure for the areas to be scanned, making it in reality the main data structure. The grid holds other classes like the Box. Box is nothing more than a rectangle on the map. Many rectangles put close to each other make up the so-called Grid (Figure 7).

The Box is also part of the API for the end-user of the framework as seen in chapter 4.6.2 API. While the Box is accessible from the end-user, the Grid is accessible only by the Scanner. The reason for this access pattern, is that the service is going to solely scan each box and thus needs only access to that. Giving very specific access to each object in the project makes sure that the code doesn't get messed up.
The Grid has also direct access to the GUI. When the Grid is being altered in any way, it is the Grid's job to update its representation to the user. Therefore it is the Grid's job to mediate any changes.

### 4.2.4 Logger

The Logger is nothing more than a class devoted to logging to three separate files. The three files correspond to the session, scans and results.

The session log keeps information that has to do with more global information like the total amount of requests sent, the average time it has taken to send the requests, the ping, etc.

The scans log keeps the state for every box scan. So if at some box an error is encountered, the trace-back of the error, will be made in this log file.
Results is a collection of the results from the session. Every service should add a new result.

The Logger has a public API which should be used by every deploying service (more details in chapter 4.6.2). The public API comes in the form of two methods: \texttt{log\_result()} and \texttt{log\_scan()}. The first one is being used by the service for each result in a scan. For Google Places Search, the method \texttt{log\_result} is being called for every single latitude longitude pair. So if there are 140 markers on the map, that means that \texttt{log\_result} has been called 140 times.

The method \texttt{log\_scan} is being used to log the state of the scan. In the case of Google Places Search, \texttt{log\_scan} is being called for every request and every response fetched by the server.

The third logging, the logging of the session, is automatic and thus no API is provided for it.

### 4.2.5 services

\texttt{services} is a folder that keeps a single python file for every service supported by the framework.

Every supported service is a class of its own in its own file (similar to Java conventions). For example Google Places Search which is supported from the beginning of the framework's development, is the class \texttt{GooglePlacesSearch} found in the file \texttt{GooglePlacesSearch.py}. The service Foursquare Explore is the class \texttt{FoursquareSearch} in the file \texttt{FoursquareExplore.py}.

The reasons that this (java-like) approach was taken, instead of having all the services in the same file (which was actually the initial plan) are:

1. Reduced code to go through when adding a new service (essentially none). This is also very helpful for finding errors in the service class as there is much less code to go through.

2. Freedom of developer when adding a new service. The developer is forced to use far less conventions and rules.

3. The user of the framework can have a direct view on which services are supported by just checking the file-names in the folder services.
Essentially each service class is a wrapper for the service, the glue that sticks together the framework with the service. Besides wrapping the service, the class of a service has as responsibility to provide logging of errors for easier debugging.

4.3 Map computations

A big part of the framework's GUI is the map. Here we treat the map solely as a computational problem. For the map as a visual problem, you should read chapter 4.4.1.

4.3.1 Distance calculation

For calculating the distance between two surfaces two main groups of formulas exist.

1. The first one, Great-circle distance, makes the assumption that the earth is a sphere.
2. The second, Great-spheroid distance, makes the assumption that the earth is an oblate spheroid.

In the framework, distance calculations are needed for splitting the grid into smaller squares of equal size and for taking decisions upon a service's area limitations. A concrete example is the 50 km radius limitation found in Google Places Search service (chapter 2.1.2 Scalability issues).

If Google Places Search uses a different formula than the one the framework uses, a 50 km limitation stated by Google, could mean 45 km or 55 km for the framework. This could have devastating effects. Imagine making a query for a box with radius 55 km with a service that outputs results to a maximum of 50 km radius. Everything beyond the 50 km radius would be omitted without the end-user noticing it.

From the above example it is apparent that the decision as to which formula to use is vital. Google Places Search uses an unknown formula with unknown constants and that is just a single service. There are tons of services using different formulas with different constants and the framework has to work for most of them if not all of them.

In the end a rather diplomatic approach was taken. A heuristic method was used (trial-and-error) to find the formula which gives the results closest to Google Places calculations without exceeding at any point the values given by Google Places. Simply put, that means that random vectors with random lengths (on the whole globe) were tested for different formulas and different constants. The output value of each formula
then got compared with the output value from Google Places for the same vector. The formula that was the closest to the Google Places values, without exceeding them at any time, was chosen.

The formula chosen is *Vincenty's formulae*:

$$
\Delta \sigma = \arctan \left( \sqrt{ \frac{ (\cos \phi_2 \sin \Delta \lambda)^2 + (\cos \phi_1 \sin \phi_2 - \sin \phi_1 \cos \phi_2 \cos \Delta \lambda)^2 }{ \sin \phi_1 \sin \phi_2 + \cos \phi_1 \cos \phi_2 \cos \Delta \lambda } } \right)
$$

In practice, the result $\Delta \sigma$ is being multiplied with a constant $R$ which is the radius of the earth. The constant was adjusted to fit Google's results. To find the right constant, different values for $R$ were tested until an $R$ was found that would fit most of the values for arbitrary geographical points below 75° of latitude.

### 4.4 Visualisation

In chapter 4.2.2, we treated the GUI from the back-end perspective as an API for the back-end part of the framework. Here we treat the GUI from the front-end perspective, giving the actual graphical interface. The interconnections between the back-end of the framework with its front-end are also mentioned.

The visualisation comes in the form of a GUI that can be opened on demand. The back-end application can thus run without the GUI opened at all but the user then has no direct feedback on the state of the application.

The GUI is implemented in HTML5 and Javascript. Furthermore Google Maps is being used for showing the map, markers, the grid, and other objects. The map is part of the web-page that is used for the GUI.

The back-end and the GUI communicate by exchanging short messages through agents called *messengers*.

![Application - GUI communication](image)
Both back-end and GUI, have their own messenger as seen in Figure 8.

4.4.1 Map

Google Maps v3 has been used for showing the actual scanning and its results. The reason Google Maps was chosen was mostly because of its deep documentation and ease of use. During the mingling with the map, questions arose regarding geocoding and map topology. These questions are being answered in the theory section of this thesis (chapter 3).

Google Maps v3 offers besides a fully working map, additions like polygons, figures, markers and more. The main objects needed for an initial application is a map, markers and a grid. The grid shown to the user is made up of smaller rectangles provided by the Google Maps suite (this has nothing to do with the internal grid structure at the back-end).

The thorough documentation of the library, makes it very simple to work with and any problems that arise, are easily being answered by a bit of searching on the web.

4.4.2 Websockets

As mentioned in chapter 4.4, the back-end and the GUI are exchanging information via agents, called messengers (Figure 8). The messengers have to communicate with each other somehow. Websockets were used for this intercommunication.

Websockets are similar to sockets in behaviour, with the difference that they were explicitly created for communication between web servers and web browsers (clients) [1]. In our case, the server is the main application and the client is the GUI.

The usage of normal sockets in web applications is not allowed because of security risks. There are a few libraries out there but they make usage of a Flash object, something that is not a usable solution in many cases mostly because of the bad support of the Flash technology in some operating systems.

For the above reasons, an open-source websockets library called Websockify (https://github.com/kanaka/websockify), was used. The library offers, amongst other things, a web server module in Python and a client module in Javascript, both of which were used for the development of the framework.

Some parts of the library had to be slightly modified to fit the framework's needs. The author of the library was kind enough to provide help during the framework's
development for areas regarding websockets and the Websockify library itself. The modifications of the initial library are well documented for future reference.

The way the websocket technology is implemented on the framework is by having the main application's messenger act as the websocket server while the GUI acts as the websocket client. Because websockets offer full-duplex communication, they allow space for direct control of the application from the GUI while the back-end is sending messages (updates) to the front-end. So during a burst of such messages from the back-end to the front-end, the user can pause for example the application without having to wait for the burst of messages to complete.

### 4.4.3 Communication between messengers

The websockets are just used for the communication between the two Messenger entities described above (chapter 4.4.2). A draft protocol had to be created so that both Messengers understand each other. Following are some tables showing the full spectrum of the messages that can be exchanged between server and client and vice versa.

It should be noted that the messages from client to server and from server to client are totally different. The messages are also case-sensitive.

#### Table 6: Messages from client to server

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE</td>
<td>Close the application</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Pause the application</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Continue the application if paused</td>
</tr>
</tbody>
</table>

The messages from the client to the server (Table 6) are essentially only commands to control the back-end application. Remember that the client is the GUI – front-end of the application.

The messages from server to client (Table 7) are a bit more complicated as they support a lot of different things, mostly GUI oriented. They also have a common pattern on the way they are structured. The structure is as this:

\[
<\text{action}>: <\text{object}>, <\text{arg1}>, <\text{arg2}>, <\text{arg3}>, ..
\]
Bellow follows a table with the different parts of such a message.

**Table 7: Messages from server to client**

<table>
<thead>
<tr>
<th>&lt;action&gt;</th>
<th>&lt;object&gt;</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw</td>
<td>box</td>
<td>Adds a rectangle to the map</td>
<td>lat1,lng1,lat2,lng2</td>
</tr>
<tr>
<td></td>
<td>marker</td>
<td>Adds a marker to the map</td>
<td>lat,lng</td>
</tr>
<tr>
<td>remove</td>
<td>box</td>
<td>Removes a rectangle from the map</td>
<td>lat1,lng1,lat2,lng2</td>
</tr>
<tr>
<td></td>
<td>marker</td>
<td>Removes a marker from the map</td>
<td>lat,lng</td>
</tr>
<tr>
<td>change</td>
<td>view</td>
<td>Sets the view of the map to given coordinates</td>
<td>lat1,lng1,lat2,lng2</td>
</tr>
</tbody>
</table>

In the arguments field, *lat* and *lng* stand for latitude and longitude. To further demonstrate the usage of the table in the creation of a message we will take a concrete example. If a marker is to be added in the position -35,23 on the map, the message that should be sent to the GUI from the application should be:

**draw:marker,-35,23**

### 4.4.4 Threading

The GUI of the framework comes in the form of a webpage. The back-end keeps a GUI API which essentially is a wrapper for the back-end's messenger. When something needs to be updated in the GUI, the back-end uses the GUI-API. It's the GUI API's responsibility to use the Messenger correctly to communicate with the GUI(in essence with the GUI's Messenger).

In Figure 9 the associations between the different elements becomes more apparent. Furthermore in the same drawing the threads where each element is running on are being shown.
4. Method

No matter if the application is ran on the same system or if the front-end is ran on a separate host, the Webpage GUI will always run on its own thread as it essentially opened with a browser (a second program) which will run on a separate thread than the framework in any case.

What we can have control over, when it comes to threads, is the threads in the back-end portion of the application. As seen in the diagram above, the main application's messenger runs on a separate thread than the Scanner although they both are part of the back-end.

The reason for dividing the messenger from the rest of the back-end application is to avoid hangup of the back-end due to big loads of messages being transferred between the messengers. The back-end is doing processing all the time and sending updates to the GUI, no matter if the GUI can cope with it or not. For this reason what is needed is to transfer all the update load to second thread which will further send the updates to the GUI on its own pace.

This design of course makes an open space for latencies in the GUI (not corresponding to the real-time processing), but that is a triviality. What is important is that the main application runs fluently.

For the threads in the framework the standard thread library from Python3 was used.

4.5 Latency

The framework makes use of services. In most cases these services are network based. Thus the bottleneck for the whole framework will in most cases be the communication latency for the requests and responses exchanged with the service's server or servers.

Keeping the above in mind, makes it apparent that the general speed of the application is neglect-able compared to the “lag” of the communication between the application and distant hosts. For this reason no further consideration was taken into making the
application running fast but instead readability of the code and the framework's ease of use was prioritized.

4.6 Extendibility

The user of the framework has generally two options:

1. Use an already supported service.
2. Add a new service to the framework.

With extendibility, the second choice is meant, namely adding a new service to the framework.

For this, the user has a limited view of the framework in the form of a simple API. The API was created with a few key points in mind:

- The API should be short and concise.
- The API should be easy to understand.
- The API should offer most of the basic calculations like distance calculating between two points.

More information about the API can be found in chapter 4.6.2 API.

4.6.1 Adding a Service

All services implemented can be found in the folder services. All services found in this folder are considered supported. For the framework to support a new service, the new service has to be added to this folder.

There are specific guidelines on how to properly code the new service and they should be followed to make sure that the service functions correctly.

In practice there are two things that have to be done:

1. Create the class and methods of the service.
2. Define the service's limitations.

The template of a service class looks like this:

```python
class myService :
    def search (self, box, logger):
        # do work here to produce markers
        return markers
```
service is a dictionary data structure which keeps all known limitations of the service. The most common limitations are:

- Authentication credentials
- Max geographic area where the service can be applied
- Costs and limitation in number of requests and responses

For the full range of possible limitation values, please refer to the configuration file of the application at the project source code ([https://github.com/Pithikos/Geoexplorer/blob/master/config.py](https://github.com/Pithikos/Geoexplorer/blob/master/config.py)).

myService is the name of the service. This should be something that describes the service as much as it can. The service Google Radar Search for example is named GoogleRadarSearch. (The class name doesn't need to have the same name as the name of the file where it resides.)

The search () function is going to be run for each box in the map. The box is never going to be larger than the area accepted by the service, as the scanner takes into consideration the service's limitations when creating the boxes.

It is up the creator of the new service, to make sure that this function will return a set of latitude, longitude pairs. Each such pair correspond to a new marker to be set on the map.

For the function search () an object box and an object logger are provided to give access to each box on the grid and access to the logger of the application. More information about these objects can be found on the API documentation. Further documentation is also available at the source code of the project ([https://github.com/Pithikos/Geoexplorer/tree/master/doc](https://github.com/Pithikos/Geoexplorer/tree/master/doc)).

### 4.6.2 API

The API consists essentially of access to two objects: the Box object and the Logger object. Box object is every single box seen on the map. Its interface looks as bellow.
4. Method

Table 8: API: box object

<table>
<thead>
<tr>
<th>box object</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Properties</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>N</td>
<td>Coordinates of center-north point of the box</td>
</tr>
<tr>
<td>E</td>
<td>Coordinates of center-east point of the box</td>
</tr>
<tr>
<td>S</td>
<td>Coordinates of center-south point of the box</td>
</tr>
<tr>
<td>W</td>
<td>Coordinates of center-west point of the box</td>
</tr>
<tr>
<td>center</td>
<td>Coordinates of center point of the box</td>
</tr>
<tr>
<td>WN</td>
<td>Coordinates of west-north corner of the box</td>
</tr>
<tr>
<td>NE</td>
<td>Coordinates of north-east corner of the box</td>
</tr>
<tr>
<td>WS</td>
<td>Coordinates of west-south corner of the box</td>
</tr>
<tr>
<td>SE</td>
<td>Coordinates of south-east corner of the box</td>
</tr>
<tr>
<td>xMeters</td>
<td>Length of the horizontal side of the box in meters</td>
</tr>
<tr>
<td>yMeters</td>
<td>Length of the vertical side of the box in meters</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>bounds ()</td>
<td>Gives the bounds of the box</td>
</tr>
</tbody>
</table>

The Box object holds thus, only coordinate information about specific points in the box itself. So if we assume that the centre of the box is the point in the middle of its x-axis and y-axis, property N (from the interface above) will give the coordinates of the point found between the two north corners.

The second object, the **logger**, has an even more simplistic interface. The reason is that the logger is mainly used just as an access object so that there is access to logging by the service at any point.
### Table 9: API: logger object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_scan (line)</td>
<td>Appends string <code>line</code> to the scan log file</td>
</tr>
<tr>
<td>log_result (line)</td>
<td>Appends string <code>line</code> to the results log file</td>
</tr>
</tbody>
</table>

As mentioned in chapter 4.1 there are specific name conventions. The words *scan* and *results* are two of those.

*Scan* is the scan of a single box, and thus actions like sending requests, receiving responses, reading the responses, and similar can make usage of this logging method.

*Result* is every single marker seen in the map. For every result, `log_result()` should be used to record the result in a desirable manner.

To all logging, only timestamps are being added by default. Line can thus hold any information wished by the creator of the service.

The names of the logging files can be altered by configuring the framework by altering the `config` file found in the project.

#### 4.6.3 Extra tools

Besides the API provided, a small set of external functions can be found in the `lib` directory. Listed below are some of these functionalities and the package in which they belong.
### 4. Method

**Table 10: API: geotools package**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dist(lat1, lon1, lat2, lon2)</td>
<td>Gives the distance between two points on the earth's surface</td>
</tr>
<tr>
<td>middleLat(lat1, lat2)</td>
<td>Gives the middle of two latitudes</td>
</tr>
<tr>
<td>middleLng(lng1, lng2)</td>
<td>Gives the middle of two longitudes</td>
</tr>
<tr>
<td>getCountryCode(lat, lng)</td>
<td>Gives the country code at specific coordinates. Return value can be:</td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN' OR &lt;country_code&gt; OR None</td>
</tr>
</tbody>
</table>

#### 4.7 Output data

The first output of the framework is by using the map in the GUI. However many times, a record needs to be taken for further manipulation or research.

The framework allows the usage of logging to three different log files:

- **results.log**: Keeps track of the results from each scan.
- **scan.log**: Logs the actions taken inside the framework. This log should be used for debugging.
- **session.log**: Keeps track of statistics about the session. Some examples of such statistics is the number of requests sent, the starting and ending time of the session, the number of responses received, the total number of results, etc.

All three different logging files can be configured from the configuration file config.py found in the framework's main folder.

The logging files are created in the same path where the framework is run.
5 Result

The project started with a bit of research in what was available and what could be used to aim for the development of the framework.

Initially it was not apparent if the project should be about making a framework or rather a specific program. During the development however it became more convenient to make a framework out of the prototype than anything else.

The framework is supposed to fill a gap between the existing geographic services and testing tools available. The framework offers a basic suit of functionalities like logging and live-visualization that can be used for testing any geographic service. The only requirement for the geographic service is that it must output geographic locations.

At its current state, the framework succeeds in meeting all three major requirements mentioned in the beginning of this thesis paper:

1. Geographic scalability testing
2. Behaviour testing
3. Visualisation

A service can thus be tested for a larger region than the region it is limited to and tested for its general behaviour in different areas and different situations. On top of that, a visualisation is presented in the form of a GUI map to show what the service is doing at its current state.

There is a lot that can be improved in the framework, like for example the addition of new services. Currently only Google's Radar Search and Foursquare is available as services but they demonstrate in a concise way what the framework can achieve. The end-user is responsible to follow the policies of each service and not divert from them.

The framework uses a service in a geographic location, overriding any initial limitations of the service. The results after a run of the framework can be checked in two ways:

1. The user of the framework can check the log files
2. or/and check the GUI of the framework.
5. Result

The screen-shot above shows the GUI of the framework when running Google Radar Search service in a portion of Scandinavia.

At the current version of the framework, the GUI looks similar for all services. The log and send GUI elements seen in Figure 10 are merely for debugging purposes.
6 Conclusions

The framework is in a fully functional state, making it possible for anyone to start digging into the code and using the framework to his or her needs, irrespective if the user is a company, a hobbyist or a professional programmer.

This thesis had as its main goal to add to the open community a tool that does not exist, a tool which can be used without any constraints. The solution presented tries to follow the guidelines pinpointed in the beginning of this report, as much as possible. The guidelines were derived from rather specific needs of individuals. While fulfilling those individuals' needs, by making the framework open and also free to use, a superset of needs have been met. For example initially the framework was supposed to support only a few specific services. However at its current state the framework can support any service.

More specifically, the framework offers all the main features mentioned below.

**Scalability testing**
A service once added to the framework can be run on an arbitrary geographical area on the globe bypassing any geographical limitations that the service might have.

**Behaviour testing**
The framework provides logging to simple files and live visualisation which can assist for black-box-testing a service.

**Visualisation**
When the framework starts scanning using the chosen service the results of each scan is presented in the form of markers in a map. This visualisation is part of the framework's GUI which comes in the form of a webpage.

Possible scenarios for using the framework could be:

- A user who wants to scrape information by using a service.
- A company which has a service, wants to see how the service would scale for a larger geographic area.
- A user wants to benchmark a service to find out when the service becomes slow or irresponsive.
- A user wants to find errors output by the service for certain areas.
The limitations of the services are only constrained by the limitation of the user's imagination. There is always going to be a new way to use the framework.
7 Future Work

The framework is in a fully functional state. However there is a lot that can be built to make it much more stable and extend its capabilities.

7.1 Back-end

Addition of services
At its current phase, the framework supports a single service, GoogleRadarSearch, and that strictly for demonstration purposes. A whole set of services can be added to the framework, or at least some of the most popular ones. Some possible services that could be added are shown in the list below.

- Arbetsförmedlingen
- Facebook Places
- Openstreet
- Gowalla
- Yelp
- Google Nearby Search
- Craigslist

The addition of more services would make the framework richer, would reveal probably some underlying bugs and produce more examples of services that future users can get ideas from.

Topographic exclusion
Some problems arose during the development of the framework, mainly concerning the addition of more capabilities to the framework. One such problem was the feature of excluding results based on a country criteria. This is sort of a filter which would avoid searches for boxes outside a specified country. The problem is how exactly can a box be specified if it is touching another country or not. What happens if it is partially inside an other country? A suggestion is to have a database of country boundaries ship with the framework. That solution however takes a lot of implementation effort and therefore another route was taken to meet deadlines. An online service has been used, namely Openstreet Nomatim, which can be found in the geotools.py library. This however adds a huge latency to the whole search and should therefore be avoided if latency is a concern.
Extend logging capabilities

The framework could have additional logging capabilities. One such capability would be to automatically save files on the local machine. This would make the framework a scraper which copies raw files in their initial format for further investigation.

Piping services

Something else that would be very interesting would be to be able and pipe services, in the same way that piping commands in Unix happens. This would be accomplished by creating a chain of services, having each service sending its output as input to an other service. In this way a lot of filtering can be achieved and further information can be acquired from the coordinates generated.

Run multiple services

When a service is being used with Geoexplorer, most of the time is being spent waiting for a response from the server where the service is located. This time could be used instead to use one extra or more services. Thus someone would be able to run both Google Places and Foursquare search in the same area at the same time. This would probably need the further tuning of the threads in the back-end of Geoexplorer.

7.2 GUI

Markers

Besides the addition of new services, some more profound changes and additions can be made. One such change would definitely be the enhancement of the Graphical User Interface. At its current state, the GUI just shows a google map with the search area and its search boxes. During a search session the search area gets populated with markers box-by-box.

At the current state, the markers are the basic markers and have as mere functionality to just pinpoint each result on the map. A lot more could be added. First and most importantly it would be beneficial for the user looking at the map to somehow obtain information about the different markers like, address and information about what the marker represents. The markers themselves could look different depending on what the represent (pizzeria, restaurant, club, grocery, etc.).

GUI controls

Besides the map on the GUI, a control panel would be very helpful to Play and Pause the application. The protocol draft for the Messengers already contain messages for pausing and stopping the application. This can be tested by writing the message
“PAUSE” or “STOP” in the field above the google map (see screen-shot below) and send it. This is however for testing and is not implemented in the framework at the moment.

The reasons a user might want to pause the application can be many. One apparent reason is in the case of a service's server going down. In that case it might be desirable by the user to temporarily pause the session at its current state and continue later when the server is up again.

A second reason could be to debug a service or control its output in the log files before they get too messy. This is very helpful during the addition of a new service to the framework.

![Figure 11: Screenshot of the Send button](image)

Such controls would make it easy for the user to have a say during the application run without terminating the application.

Besides basic controls like that, a graphical configuration would be very nice, letting the user to directly edit the behaviour of the framework from inside an HTML page. Practically this would be to alter the options found in the configuration file of the application before starting a scan session.

**Additional map**

Another enhancement that could be possible, would be to detach Google Maps from the GUI and replace it with some other type of map. Further research should be made to decide wherever that would have any benefits to the application, besides the exclusion of the Google logo which by Google Maps policy has to be visible on every google map. Furthermore Google Maps locks the framework to a Mercator projection. This however can give false visual results on certain occasions, namely when the map is observed from a big height. Because of Mercator distortions, some other map could
be used or maybe the framework could roll back to a different map after a specific threshold has been reached while zooming in or out.
8 References


[6] Python Pocket Reference: Python in your pocket, Mark Lutz, 2009, Publisher: O'Reilly Media


8. References


[19] Apple Inc., Displaying Maps, Figure 5-1, Mapping spherical data to a flat surface