Christina Carty, Nikola Vracevic
Practice Software Development
Professor Resch
17 February 2023

SWE Final Project Documentation

1. Project Goals

This program aimed to visualize both a set of tweets and their associated geospatial data, as well as a map layer from a WMS, all using object-oriented programming methods in Java. The goals of this project centered around a set of processes, with the result of each process serving as a requirement for the success of the subsequent. In this project, we have defined these processes as; 1. Connecting to a WMS Server and requesting map image, 2. Parsing a CSV containing the relevant tweet data, and 3. Creating the associated KML file. The task of creating a KML file containing the CSV and Map data was extended to include, stylized time series visualization of the CSV data.

In this documentation, we outline the preliminary setup and software requirements for our program, explain some of the design choices undertaken, dive into more detail about the programmatic implementation of each task, and summarize our results and observations about the program performance.

2. Setup and Software

The entirety of the project was executed in Java via the Eclipse IDE, version 2020-12. Only one externally created file was used, the *twitter.csv* file containing the semicolon delineated values regarding tweet data. Otherwise, all resulting files, namely the png representing the WMS map image and the resulting KML file, were created programmatically, in code. Aside from Java and the Eclipse IDE, installation of Google Earth was also a required prerequisite for this project.

A handful of libraries and modules were used to assist in the goals of the project. The OpenCSV library (and the associated *opencsv-5.7.1.jar* dependency file) was used for parsing the *twitter.csv* data. The JDOM library (dependency *jdom-2.0.6.1.jar*) allows for efficient and clean XML document creation, and was the primary means of programmatically crafting the KML document in this project. The GeoTools library for Java provides a standards-compliant way to work with GIS data, and for our purposes was used to pull map data from a WMS host. Since our geotools needs were specifically and exclusively related to the mapping service, the only dependency used for this library was *gt-wms-21.2.jar*

3. Design Decisions

As mentioned, the project goals were visualized by three distinct yet interconnected processes, and the goal was for this to be reflected in the code. Thus, the package is divided into three classes; 1. *WMSConnector* 2. *CSVParser* and 3. *KMLWriter.* Design decisions specific to each class are discussed below.

3.1 *WMSConnector*

Creativity in this class was limited, as the GeoTools documentation already offers a straightforward way of implementing and executing WMS requests in Java. Perhaps the only choice worth noting was that to pull the two layers from the MassGIS host rather than the single layer from the Heigit map service. This was primarily due to the fact that the Hegit map service returned an error regarding the XML file and extending the WMS request to include the two MassGIS layers proved an easier task than addressing the Heigit error.

3.2 *CSVPulling*

As mentioned, we chose to use a designated library (Open CSV) to tackle the twitter.csv file rather than “manually” manipulating the file as rows of strings. The ease, efficiency, and flexibility offered by the OpenCSV library was simply a more attractive and time-efficient option. While there certainly existed even further 3rd-party based “abstractions” that could have aided in CSV parsing (for example, at one moment we considered creating a bean to allow for ‘column-by-column’ extraction from the CSV), we found that simply the features provided by the OpenCSV library and some well-designed java was enough to parse the CSV in a way that suited the program needs.

Additionally, in this class we chose to not only define the method by which data could be extracted from the CSV, but to go ahead and instantiate those data extractions as separate lists holding the desired data values. By ensuring that these lists were non-static, we could then simply call them from the CSVParser class into a desired class as needed. The alternative method considered was having the CSVParser class be a generic method that could pull any to-be-defined set of values into a to-be-defined list. The specific lists could then have been instantiated in a different class by calling the CSVParser as a function multiple times and changing the “to-be-defined” values each time to create the lists. Ultimately, for clarity and maximum separation of tasks, we decided that it was cleaner to create and fill the lists entirely in the CSVParser method, rather than have part of the “parsing” task spill over into other classes.

3.3 *KMLWriter*

Again, for this class we chose to rely on a library to do the heavy-lifting of the class objective. In previous explorations of KML within the scope of the Software Development course, we had created a KML document by manually writing out the document content in Strings. The use of the JDOM library allowed us to create the KML document in a more programmatic way, that is, where each element was more akin to a variable that could be assigned, filled, and– importantly, within the scope of XML document creation–, nested. Given the increased complexity of the desired KML document, it seemed

cleaner and less error-prone to use JDOM rather than manually write out each section of the KML document.

For visualization of the tweets, we chose to visualize the timestamp of each tweet as a uniquely colored, protruding line. The choice of an extruding line over, for example, a polygon, was due to ease. The format of the data was single point coordinates and so introducing an extruding line at the pre-existing coordinate was simpler than creating a new polygon.

4. Implementation Details

Here we will discuss the programmatic specifics of the project implementation. For clarity, details will once again be discussed class by class.

4.1 *WMSConnector*

As mentioned before, the bulk of the WMSConnector class code was taken from the Geotools WMS documentation. Per this documentation, once the "client" has been defined by storing the server URL in a variable, we instantiate a GetMapRequest object (i.e., *request*) that is then configured by various parameters pertaining to our map request. Of note here is the two instances of *request.addLayer*, to reflect the fact that with the MassGIS client we need to ask for both the "GISDATA.PARCEL_STATUS" and "GISDATA.CENSUS2000TRACTS_POLY" layer.

Once the GetMap request goes through and the image is pulled it then has to be read and processed in a way where it can be downloaded into an actual visual representation, which is what happens through the nested *ImageIO.read* and *response.GetInputStream* methods.

Once the image is saved to our designated output file path the WMSConnector class has served its purpose, and all we need to do is refer back to that file path whenever we want to access this class' output.

4.2 *CSVPulling*

The *CSVPulling* class was the only class that needed to communicate with other classes, so the visibility on the associated variables and methods was important. Notably, *CSVPulling* does not contain the classic **public static void** "main" method, so it is non-executable. That is, no part of the CSVPulling class outputs anything onto the console. This is fine for our purposes– we don't want *CSVPulling* to *output* something, as much as we want it to *create* something, especially something that can be grabbed later on.

Specifically, within the CSVPulling class we define four lists representing the four values we want to pull; latitude, longitude, the tweet, and date-time stamp of the tweet. Then, in a separate *non-static* method, we build and execute the Open CSV parsing mechanisms. Setting the variables outside of this method, and setting this method to be non-static allows us to manipulate the variables (i.e., add to the lists) "in-place" without having to recreate them in the *CSVPulling* method, and without having to lose our in-method changes to them outside of the *CSVPulling* method.

The OpenCSV approach involves using a "CSVreader '' and a nested "CSVparser '' object to handle the CSV file. The CSVParser object allows us to specify that this specific CSV is semi-colon

separated, rather than comma separated. The CSVreader object has the indexable “nextline” method that moves line by line through the CSV. Since nextline is indexable where each space between semicolons represents an index value starting at 0, pulling the desired values requires knowing only what index they sit on in the document, indexing next-line at that point for each line in the CSV, and storing each value in our aforementioned lists.

Finally, in order for the newly created lists to be accessible by the other classes, we added four more methods that acted as “getter” methods. That is, when called by another class, their sole purpose is to pass the specific list they are “getting” into the variable where the method is being called.

4.3 *KMLWriter*

With the PNG image from *WMSConnector* and the pulled values from *CSVPulling*, we are now ready to create the KML file that puts it all together. The entirety of this class takes place in the main method, and begins by instantiating the pre-filled lists containing the desired values, using our “getters” from the *CSVPulling* class. At this point we also take a moment to define some important file paths as string variables that we will need to use later on.

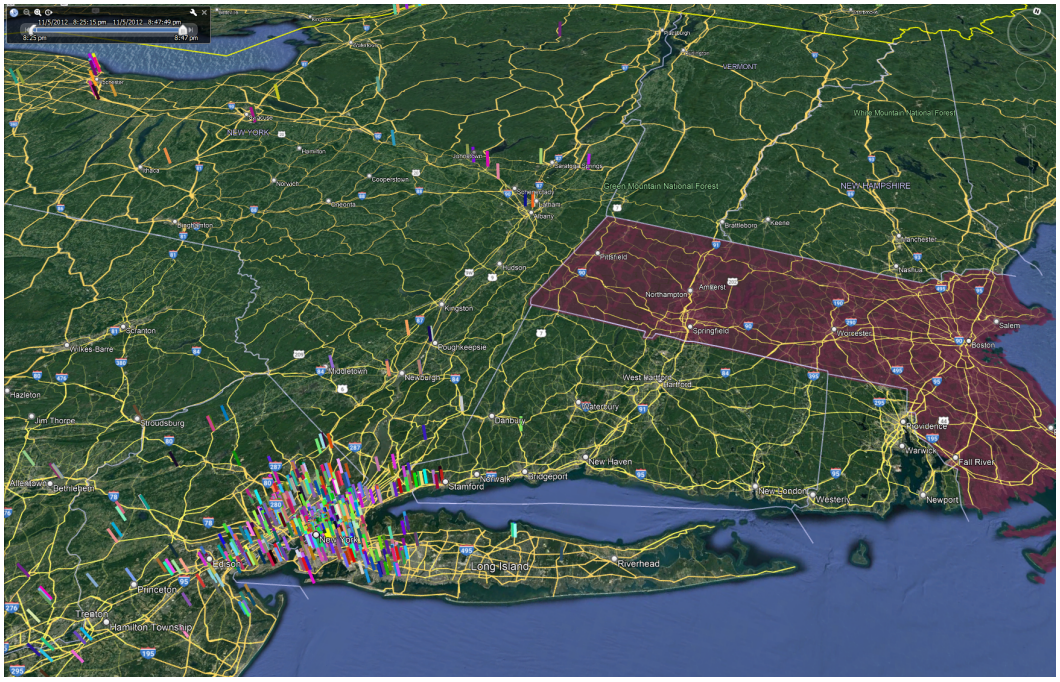
The KML writing officially starts with instantiation of a try/catch block, where the rest of the code will take place. As mentioned, JDOM allows us to turn the task of XML document writing into variable naming and manipulation. Each element is assigned to a named *Element* variable. Elements with text can be filled via the *addText* method, where we usually assign the desired text to a string variable first, and then pass the variable into the *addText* method. Element nesting happens chronologically via the *.addContent* method. Thus, creating the KML required only understanding how and where elements should be nested, and following formatting guidelines.

Some elements, like the overarching *document* element, as well as the *groundOverlay* element and its children, only occur once, and we defined those first. The bounding box of the GroundOverlay was taken from the MassGIS layer metadata, the color was set to a blue-gray with the prefix “B3” to indicate 70% transparency, and the PNG file location was stored in a nested Icon element.

Since we needed a placemark for each line in the *twitter.csv* file, those elements were created in a for loop with as many iterations as the length of the one of the values in our lists (we arbitrarily chose the latitude value).The assignment of each element was as simple as following the KML documentation, but there were some not worthy adjustments. First, for the “coordinate” element, we concatenate the strings of the *latitude* and *longitude* lists at a given iteration, since KML needs to read them as a pair rather than two separate values. Secondly, the dateTime values were stored within a *TimeStamp* element, which enables time series visualization in google earth. Otherwise, the tweet itself was stored in the “description” element of the placemark.

The time-series visualization was achieved by setting a unique style via the StylURL element for the line geometry in each placemark. This was done in the same for loop as the general placemark element adjustments. The styles for these placemarks (namely the colors) were stored in a separate array that was called within the for loop so that each placemark could have a unique style.

5. Results & Reflections



The result is a series of extruding lines that can be visualized by their time of creation via the slider in the top left corner. The WMS Image item is additionally seen on the right.

Despite the relative success of our program in terms of achieving the task goals, there are several areas of improvement. Firstly, we believe that higher levels of modularity and general object-oriented programming ideals could have been achieved. For example, in the KMLWriter class, the entirety of the program exists in the main method, when really the only thing that needs to be "executed" is the google earth launcher. It likely would have made more sense to separate the KML file writing into separate methods or even a different class. Secondly, our work and research on this code came to show that manually adding dependencies into project build paths is generally bad practice, and it is *much* cleaner and more efficient to use a dependency handler like Maven, especially with libraries like GeoTools which are so huge and can often cause problems when are manually implemented. Finally, KML is a quite powerful and flexible way to visualize geospatial data, and we recognize that it was not used to its full potential even within the scope of its program. A more ideal outcome would have taken fuller advantage of the different geometry styling options and visualization methods that KML has to offer.

References

- https://docs.geotools.org/latest/userguide/extension/wms/wms.html
- https://opencsv.sourceforge.net/
- https://developers.google.com/kml/documentation