# Introduction

xxxxxxxxxxxx

# General Structure

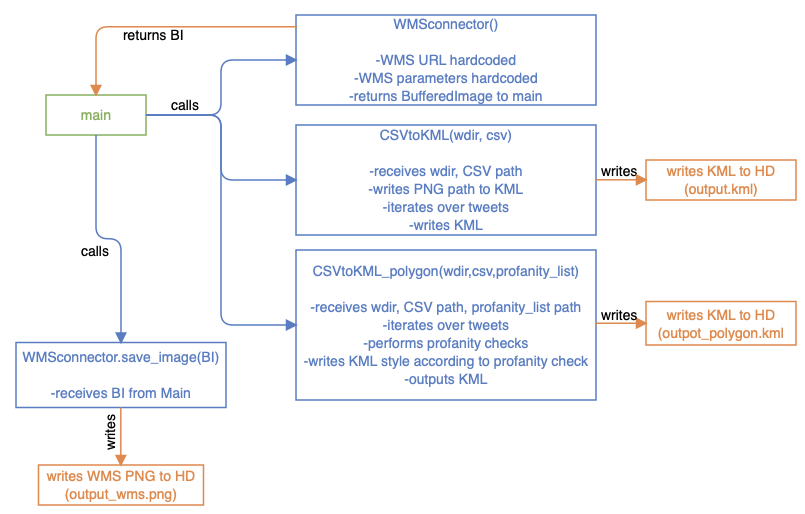
The project submission contains of 3 Java classes:

* The *GoogleEarthTweetMapper.java*, the main class, which calls functions from the other classes and saves the resulting WMS image
* The *WMSconnector.java*, which sends the WMS request and returns a Buffered Image, which is then saved by the main class
* *CSVtoKML.java*, which iterates over the *tweets.csv* file line by line and parses the information into the KML format, which is then saved.
* *CSVtoKML\_polygon.java*, which performs the same iteration and KML creation as the *CSVtoKML* class, but additionally creates an extruded polygon and checks the tweets for profanity, color-coding the polygons to visualize wether or not the tweets contains profanity.

All java files, including the KML outputs as well as the *tweets.csv* file and the profanity list can be found in a public [GitHub Repo](https://github.com/simon-donike/eot_donike_porti).

UML DIAGRAM GOES HERE

# GoogleEarthTweetMapper (main)



As shown in the diagram above, the WMSconnector is called first by instanciating a WMSconnector object. Then, on that object, the getWMSimage method is called, which returns a *Buffered Image*. This is then saved and written to the Hard Drive via a file writer, passing the Buffered Image to the WMSconnector.save\_image() function.

Next, the CSVtoKML class is called, passing the directory and file name of the *twitter.csv* file, which saves the created KML into the working directory.

The same is done with the CSVtoKML\_polygon class, which receives the same arguments. The created KML is saved to the Hard Drive from within the class. I would also be possible to write the finished HTTP WMS request into the KML so that it is dynamically loaded when opening the file, but it was explicitly stated in the task to store the WMS PNG on the hard drive.

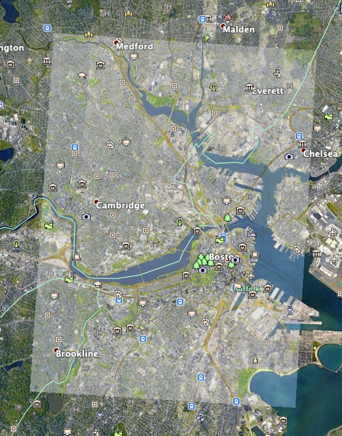
# WMSconnector

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

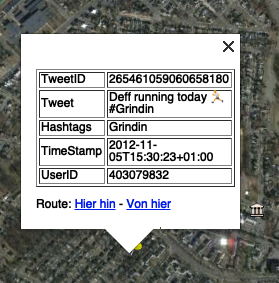
The WMSconnector clas contains two functions. The getWMSimage() function is called from main without any arguments, since all relevant parameters are hard-coded into *GetMapRquest.request*. Before that, a try-catch block validates that the input WMS URL is valid. After sending the request via .issueRequest(wms), the returning image is read via getInputStream() and saved as a *Buffered Image (BI)* variable and returned to main. From main, the BI is then sent to the WMSconnector.save\_image(BI) function, which is passed the BI and then saves it as PNG to the working directory.

# CSVtoKML

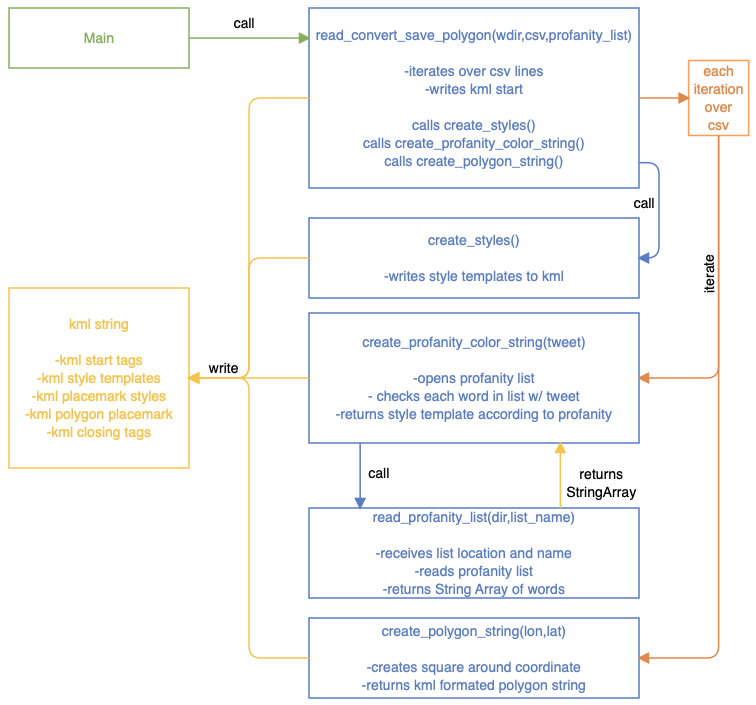


This java class takes the working directory and the csv file name as arguments when it is called from main. It only contains one function, which handles all operations. This was done because the writing of the KML is quite straight forward, the iterating over the csv can easily be done with a while loop and therefore it seems overly complicated to split the steps up in different functions.

Firstly, a ground overlay element is created via pre-defined strings and attached to the KML string. The transparency is set via the color tag, which is explicitly not intended for use with raster data, but it works anyway. The WMS image is inserted by pointing to the correct storage location of the PNG.

  
Then, the csv file is read via *Buffered Reader* and an empty String Array created. Also, the string which will later on contain the whole KML string is created and filled with the KML header information. Iterating over the tweets list (excluding the header) via a while loop, the information for each tweet is read and stored in the array. Using string concatenation, the relevant KML tags are opened and the info from the csv line inserted. The extended data tags are used because they are shown as a nicely formatted table when clicking on the icons in Google Earth Pro. Additionally, the timestamp from the tweet does not conform to the specifications as given by the KML documentation, so the space is replaced with a T and the hour added to the end of the string. The timestamps are saved within “TimeStamp” KML tags so that Google Earth can correctly identify the time series of the tweets. After each iteration, the string holding this line’s tweet information is appended to the KML string. All statements within that KML string are given with formatting statements such as tabs and new lines. At the end, the KML string s saved as a KML file and can be openend with Google Earth Pro.

# CSVtoKML\_polygon



In general, this class shares the same skeleton with the CSVtoKML class, therefore the inner workings of the KML creation and writing will not be detailed again.

Generally, the class performs the same tasks but with two additions:

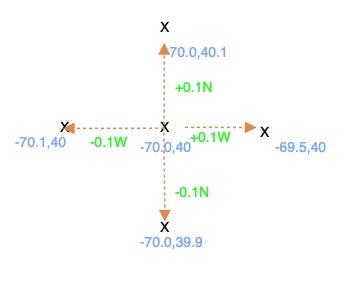
* Extruded Polygons instead of Points are shown on the map
* The tweets are checked if they contain profanity and the polygons colored accordingly.

The list of offensive words is taken from <https://github.com/RobertJGabriel/Google-profanity-words>, which is the list of words considered offensive by the Google Search Engine.

Since the creation and extruding of polygons, as well as the saving of KML style templates and the opening, checking and visualization of the tweets according to their profanity content contains quite a lot more complexity, many of those tasks were outsourced to functions. The functions are all called from within the read\_convert\_save\_polygon(…) function, therefore only one call from main is necessary to start the whole process.

Firstly, after writing the opening KML tags to the string, the create\_styles(…) function is called. This returns style templates for the polygons as a string which is written to the KML string and can later be referenced by their IDs from within the polygon tags.

Then, within every loop over a new line of the tweets.csv file, the create\_profanity\_color\_string(…) function is called. This function is passed the tweet itself aswell as the location and name of the profanity list file. For each tweet, the read\_profanity\_list(…) function is called, which receives the location and name of the profanity list and opens it, returning it as a String Array. Back in the create\_profanity\_color\_string(…) function, this String Array is iterated over and checked for matches with the tweet. If a match is found, the KML color style tag for “red” is received and written into the KML structure of the polygon of said tweet. The detection is not perfect, since checking for the words itself would return many false positives (“I p**ass**ed my exam”). Therefore, a space in front of each offensive word is added to exclude such false positives. Adding a space behind the word would have excluded many common expressions where a punctuation mar is added at the end, such as “you’re a bitch!”. At the moment, the file is opened, read and closed for each tweet and it would be more elegant to store the profanity list as a string array.

After the style information is added according to the profanity content, the polygon around the point is created. The create\_polygon\_string(…) function is called, which receives the coordinates of the tweet. Adhering to the KML polygon specifications, the polygon tags are created as strings. The coordinates are transformed, adding and substracting a certain value to and from the coordinate to create a square around the coordinate. Also, the parameters for extrusion are enabled and set as relative to ground, meaning the polygons are extruded from the surface by 100m as specified with the z coordinate in the coordinate section). The KML string containing the polygon is then returned to the loop.

Since for each tweet, the information of the tweet itself, the polygon and the check for profanity and the definition of the according style is now done, the temporary tweet KML string can be added to the ever-growing KML string. Finally, the KML string is stored as “output\_polygon.kml” in the working directory.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

# Conclusion

xxxxxxx