

856.152 Practice: Software Development

Documentation

End-of-Term Assignment

WS 2019/20

The documentation is referring to the Java package
“eot_Schlagbauer_Salvans_Servais_Rossboth”

and was elaborated by:

Andreas Schlagbauer andreas.schlagbauer@stud.sbg.ac.at


Gil Salvans Torras gil.salvans-torras@stud.sbg.ac.at

Marius Servais marius.servais@stud.sbg.ac.at

Theresa Roßboth theresa.rossboth@stud.sbg.ac.at

Aim of the work

The goal of the End-of-Term Assignment is the development of Java classes (stored in a Java package), that displays data from two different sources (PostgreSQL database table and OGC WFS data) statically and dynamically by using the `Processing` library.

The PostgreSQL database-table stores  Tweets data, including time, coordinates and keywords related to the sentiment of the message (happiness, anger, disgust). The `WebFeatureService-Connection` allows to access data from physiological measurements of a cyclist including the measured heart-rate as well as the geographic position of the bicycle rider and the time. The desired outcome is a `TimeSeries`-class, that displays a cyclist moving along Boston while showing his heart-rate and twitter data popping up related to the user's feeling.

Design decision

We decided to design our project with three classes, that connect to the different data sources and preprocess the data, and one class, which deals with the preprocessed data and displays it on a map. The last class will be later on available as two versions, a static and a dynamic version (see fig.1).

The **first Connector class**, `WMSConnector`, requests a tile map of Boston from a WMS-Server with specific boundaries and stores the map as an image temporally. The **second Connector**, `WFSConnector`, accesses a WFS-Server of our institute for retrieving the location (geometry), time (attribute) data and physio-measurements (attribute) of a bike trip through Boston. Due to the fact that the visualizer class (last class) has a pixel based coordinate system we need to implement a coordinate converter in the `WFSConnector`. Our solution is to use the extends of the WMS Tile as a predefined bounding box in order to determine the relative position of the bike rider and to be able to convert it in a pixel based coordinate system. The extracted and converted coordinates as well as the attributes physio-measurements and time are stored in several array lists to be accessible from the visualizer class. The **third Connector class**, `PGConnector`, connects to a PostGIS database of our institute and extracts twitter data from the DB with a prefiltered SQL-Statement. The retrieved data consists of the position as geometry and emotion of the user as an attribute. As with `WFSConnector`-class, a coordinate converter with the same functionality is implemented in the `PGConnector` class and the extracted data is stored in array lists.

The **Visualiser class** as a static version, `StaticPointVisualiser`, uses the temporal image of the `WMSConnector`-class and draws it as background, before accessing the lists of the other Connector classes and displaying the location of the bike users and the twitter data on the map as a circle. The colour of the circle will be defined by the physio-measurements (heart-rate) or the emotions of the twitter user. The `TimeSeriesVisualiser` as the dynamic version of the `StaticPointVisualiser`-class has the add-on to visualize the location of the bike rider and the twitter data dynamically by iterating through the extracted time variable of the `WFSConnector` class (data of the bike trip). Due to mismatching time periods within the bike trip data and the twitter data (1.5h vs 2 weeks, differing starting times) we decided to synchronize the twitter data with the bike trip. That means, both datasets have the same starting time and the twitter data will be compressed in such a way that when the bike trip ends the whole twitter dataset was visualised once (see Implementation).

The decision to take only four classes for implementing the task and not more (for example: for getting a higher modularity) is based on the size of the classes. We could create for example a class that defines the bounding box of the WMS or a class that has the formula of the coordinate conversion, but those classes would only consist of 4 or 5 lines of code. Therefore, we decided to implement those smaller tasks within the Connector classes for getting the project more compact.

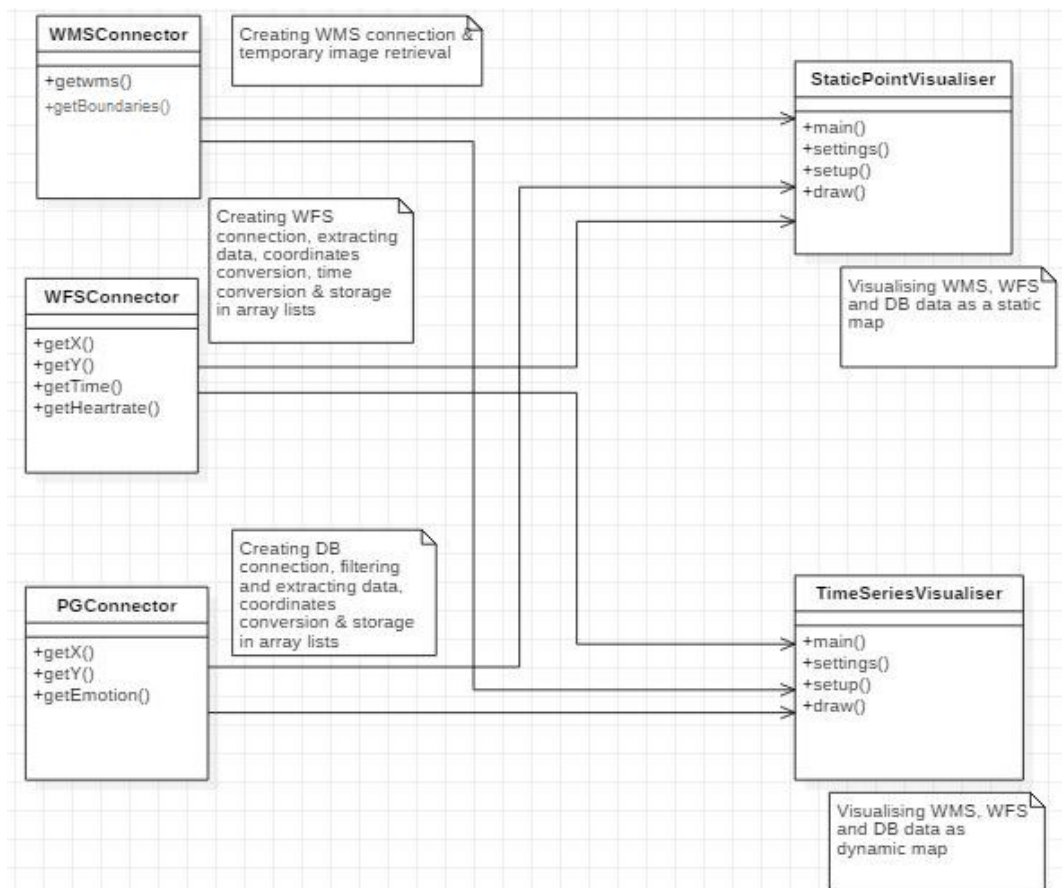


Figure 1: UML design of "eot_Schlagbauer_Salvans_Servais_Rosboth".

Implementation details

As part of the practical lecture for the module "Software Development", we were already introduced to the basic concepts of connecting to a Web Feature Service and to a SQL database as well as visualising points with the processing library. In the process of writing new classes for this project, we re-used some parts of these previous classes, added additional functionalities and adapted them to our needs. For the realisation of the project we used the following external JAR's from the three different libraries (GeoTools 21.2, Processing 3.5.4 and Java Database Connectivity 42.2.10):

- commons-codec-1.13.jar (GeoTools dependencies)
- core.jar (Processing)
- ejml-core-0.38.jar (GeoTools dependencies)
- gt-epsg-hsql-21.2.jar (GeoTools 21.2)
- gt-main-21.2.jar (GeoTools 21.2)
- gt-wfs-ng-21.2.jar (GeoTools 21.2)
- gt-wms-21.2.jar (GeoTools 21.2)
- indriya-1.0.jar (GeoTools dependencies)
- si-quantity-0.7.1.jar (GeoTools dependencies)
- si-units-java8-0.7.1.jar (GeoTools dependencies)
- system-common-java8-0.7.2.jar (GeoTools dependencies)
- unit-api-1.0.jar (GeoTools dependencies)
- uom-lib-common-1.0.2.jar (GeoTools dependencies)
- uom-se-1.0.8.jar (GeoTools dependencies)
- postgresql-42.2.8 (JDBC library)

Due to missing dependencies within the GeoTools library, we used the fixed GeoTools JARs from the lecture. These are referenced as GeoTools dependencies. Hereafter, we describe and explain the used classes.

WMSConnector

The WMSConnector-class is the starting point for our project. `getwms()` produces the output that is required for the visualisation done by the classes `StaticPointVisualiser` and `TimeSeriesVisualiser`. `getBoundaries()` defines the bounding box of our visualisation and is called upon in the class `PGConnector` to filter out the data which is not within the image of the Web Map Service. First off, the class WMSConnector defines the outputs "wms_image" as `PImage` as well as the bounding box parameters as global variables. The parameters of the bounding box (the maximal and minimal values of the x- and y-coordinates) are derived by hand after defining the focus and view of our visualisation. These are stored additionally within a double array to easily import them into the `PGConnector`. After defining the needed parameters and output variables, we create a connection to the Server and request the wanted map using a `GetMapRequest`. Besides the boundaries, we define additional parameters (Coordinate System, Dimensions, Format etc.) for the background map and execute this `GetMapRequest` within a "try catch" statement in order to handle eventual exceptions. The response of the Server is then temporarily stored as a `Buffered Image` and then converted to a `PImage`. This is the corresponding data type for the processing library which is needed to visualise images.

WFSCConnector

The WFSCConnector-class is structured similar to the WMSConnector. The WFSCConnector has four methods: `getX()`, `getY()`, `getAttribute()` and `getHeartRate()`. These functions produce four different array lists containing the x- and y-coordinate as well as the time and heart rate for each point. These array lists are interlinked by their index and can later be used within the classes `StaticPointVisualiser` and `TimeSeriesVisualiser` to display the data from the Web Feature Service.

Therefore, we define the four array lists in the beginning of the class as empty global variables and execute the WMSConnector-class in order to retrieve the boundaries. As a next step, we send a request for the feature "physioMeasurement" and parse the feature for each object. The needed attributes of each object are stored as geometry (consisting of x- and y-coordinate) and attribute (heart rate and time). To visualise the data in the end with the processing library, we need to convert the coordinates of each object from exact coordinates to relative values which represent the position of the objects not on earth but within our visualisation boundary (1000 x 1000 pixels). Therefore, we used the boundary values of each axis corresponding to the pixel sorting of the Processing library. After converting each coordinate individually, they are stored in the respective array-lists. Both attributes, time and heart rate, were stored as string and therefore needed additional parsing to convert them into the needed data type (integer and `LocalDateTime`). Due to the formatting of the time in the Web Feature Service according to ISO standards (yyyy-MM-dd'T'HH:mm:ss"), we can simply parse the time using the `DateTimeFormatter`. Storing the time as a `LocalDateTime` data type allows us to easily extract only the time as `LocalTime` which we need for the `TimeSeriesVisualiser`. These are all added to the empty global array lists within the while loop, ensuring us that related data are stored with the same index.

PGConnector

The class `PGConnector` is counterpart of the `WFSCConnector`. It has also corresponding methods to get the coordinates `getX()` and `getY()` as well as the attribute we are interested `getEmotion()`. The data is stored also in array lists just as the `WFSCConnector` data and later used for the visualisation.

Just as the `WFSCConnector`, we first define our global array lists and get the boundaries by executing the `WMSConnector`-class. After that, we connect to the designated Postgres database with the corresponding access parameters. Similar to the `WFSCConnector`, we also iterate through the tweet data one object at a time. In contrast to the `physioMeasurement` data, the tweet is spread out over a wider area than our area of interest. Therefore after converting the coordinates to relative values within our visualisation boundary of 1000 x 1000 pixels, we have to limit the number of objects depending on their position in relation to the bounding box. This is done by the if condition. It is important that the objects are only added within this if condition to ensure that the related data is stored with the same index.

Static Point Visualiser

This class has been created with the aim of creating a static visualization with the Processing library from the extracted data of the WFS, DB and WMS as the base map.

First of all, as our visualization classes use the Processing library, they must inherit the properties of the Processing Applet (by using "extends PApplet" in the class declaration) in order for us to use the specific methods and commands of the library (settings, setup, draw, background...). Then, the data can be retrieved by creating objects of the classes which connect and filter the data needed (from WFS and DB). As these classes are instantiated as objects, we can call their methods to return the different array lists and therefore, get the filtered data. This part is the part where the processing library and the visualization process starts taking place. First, the methods named settings and setup are used to set the parameters of the desired output such as size or background. Secondly, the method draw will be the one in charge of the visualisation of our desired features. The method draw() works like a loop, therefore it does not stop running. This feature will be useful for the visualisation of dynamic content. Inside this drawing method the base map (as an Image since we saved it as a PImage object) and two for-loops are defined: one for the database data (twitter) and the other one for the WFS data (heart rates). In each for-loop, the programme goes over the data element by element (i) and prints out an ellipse per every point with the specific coordinates (x,y) and categorized with a specific style. If it is a tweet, it is classified according to the emotions (color), and if it is a record from the WFS, it is classified according to heart rate value (size and color). Finally, a main method must be specified in order for the class to become executable and consequently, provide the user with the desired output: the static visualization of the data.

TimeSeriesVisualiser

This class is the one in charge of generating a dynamic visualization of the data as a time series interaction. Like the previous visualization class (Static), it uses the data from the WFS, DB and WMS as base map. As it is a visualization class, it extends from the Processing Applet as explained previously.

In functional terms, it retrieves the data from the connector classes with the methods to get the specific array lists (x- and y-coordinates for both data sources, wfs heart rate, wfs time and twitter emotions). However, why are we just taking these attributes? Basically, because we need latitude and longitude for both cases in order to represent the points on the screen map (WMS output), the heart rate and emotions to categorize the data, and the WFS time to carry out the time series. Therefore, we call the objects of the connector classes and the methods to retrieve the data (as array lists) as global variables (so they can be used in whichever method within the class). In the global environment, we also define the variables j and i (as integers), which are set at value 0 in order to be the argument for the iterations of physiological measurements (WFS) and twitter data (DB) respectively. After that, a settings() and a setup() method must be defined in order to specify background (wms image), frame rate (for the visualization), size or stroke (of the features). In this case, we defined the image background here (in the setup method), because we want to create a dynamic visualization in which the elements drawn must stay there and not be deleted in each iteration (the map is just print once in the beginning). This is necessary because the draw method acts as a loop and keeps running over and over. According to that, there is no need to use loops for each dynamic time frame, because in each iteration the new records (the ones according to the specific time) are printed. These records are categorized and styled in the same way as the static visualization class (see Table 1).

In terms of time, we are only taking the WFS time period into consideration because it is the one which makes the most sense in order to be visualized in a time series interaction: the user can clearly see the route which was carried out (with the heart-rates) and while that, the tweets pop-up in different areas of the city. This synchronisation of data has been carried out with the criteria of displaying 20 physio-measurement records per 1 tweet. In other words, every 20 heart-rate measurements displayed, one tweet from the DB is displayed. With this, the time series can put together data with completely different temporal characteristics (the tweets take 2 weeks of time, while the physio measurements take only 1,5 hours). In visualization terms, the time (from the wfs) is displayed on the top-left of the screen after having been subject to a formatting process ("HH:mm:ss"). Finally, a main method must be created in order for the class to become executable.

color	heartrate	reflecting	color	emotion
●	less than 120	light activity	●	happiness
●	between 120 and 130	light training activity	●	anger_disgust
●	between 130 and 140	medium training activity	●	sadness
●	between 140 and 150	increased training activity	●	fear
●	higher than 150	hard training intensity		

Table 1: Implemented color schemes for the physiological measurement-data (left side) and the tweets-data (right side).

Visualisation

The executable class `StaticPointVisualiser` as described above in the Implementation is fulfilling the *minimum* requirements for visualisation. Herein, the physiological measurement data and the Tweets data are statically drawn like shown in Figure 2.

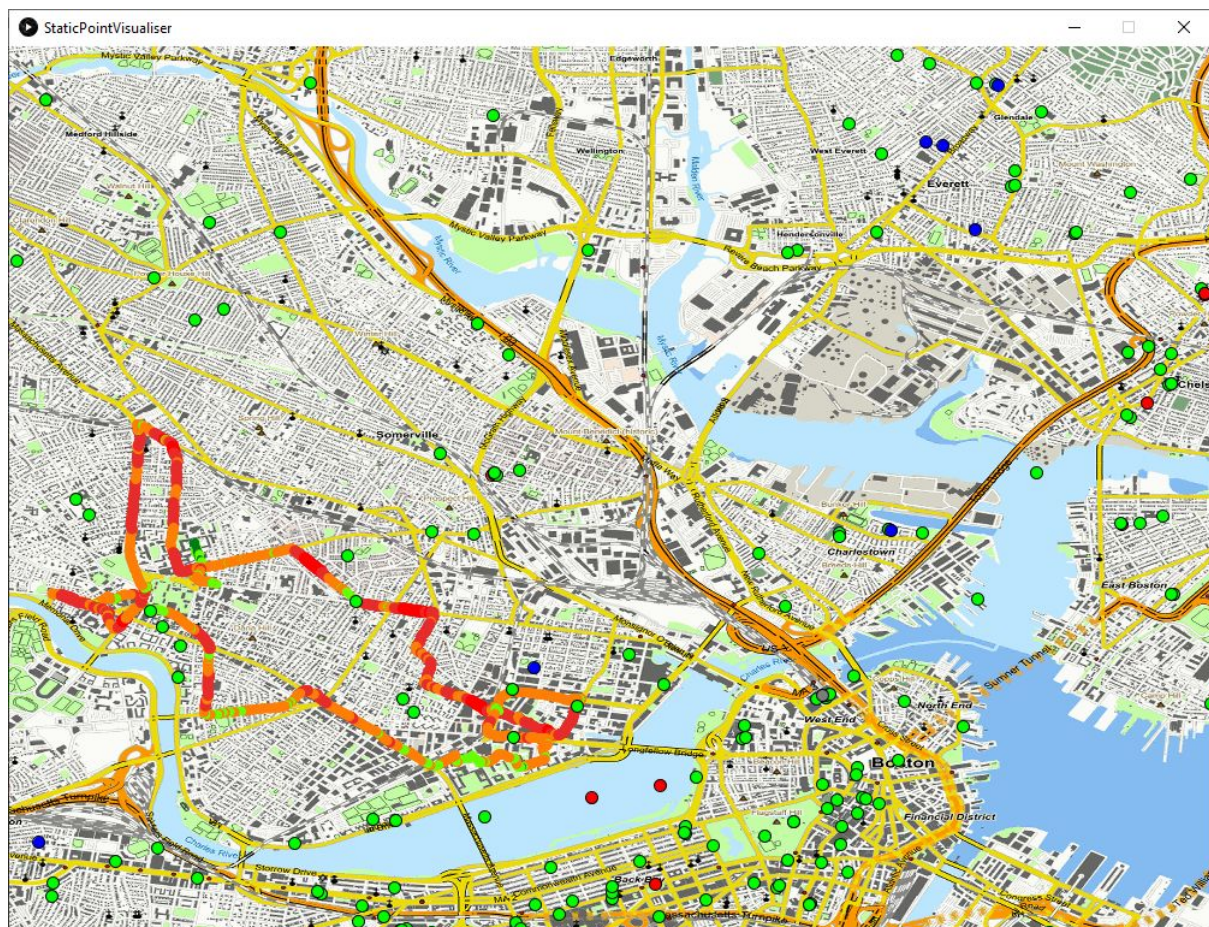


Figure 2: Static visualisation of the heart-rate- and tweets-data.

The executable class `TimeSeriesVisualiser` is achieving an advanced visualisation ("the *optimum*"), in which the "heartrate" values and the tweets are both drawn as a time series. Due to the chosen implementation, the drawing at the stopping point of the time-series visualisation it is equal to the static output.

Work distribution

The present End-of-Term Assignment is a piece of group work. All four students involved in the development contributed fully equally according to their skills and the particular demand during the working process. Regular team meetings were used to discuss the general design, the implementation decision, the coding progress and the final presentation of the result. The chosen work arrangement was rather time consuming and could have been more efficient by explicitly splitting the work flow. However, it turned out as very useful in terms of the individual learning success; whenever something did not work out for one, there was always one of the group to help out. The collective approach is reflected in the work distribution table (see Table 2).

	Andreas Schlagbauer	Gil Salvans Torras	Marius Servais	Theresa Roßboth
Design decision	40%	20%	20%	20%
Modelling (UML)		50%	50%	
Coding	variable:	variable:	variable:	variable:
WMSCconnector	40%	60%		
WFSCconnector	15%	5%	40%	40%
PGConnector			80%	20%
StaticPointVisualiser			50%	50%
TimeSeriesVisualiser	20%	50%		30%
Improvement of the inline documentation	40%	20%	20%	20%
Report	25%	25%	25%	25%

Table 2: Work distribution of the group members.

Obstacles and observations

- Time out problems!!!!!!! (that is a reason why we used Boqin's server for testing)
- No access from home (that is another the reason why we used Boqin's server for testing)
- Data with completely different time frames
- Good availability of documentation, especially for the Processing library.
- The organizational process of definition of objectives, planning and design, is extremely important to carry out whatever kind of project in the software development area ("the house needs to be built bottom-up and not top-down).

References

- Practical lecture documentation
- <https://www.tutorialspoint.com>
- <https://processing.org/>
- <http://docs.geotools.org/>
- <https://docs.oracle.com/javase/tutorial/java/data/manipstrings.html>
- <https://stackoverflow.com/>
- <https://gis.stackexchange.com/>