Initial Findings from Close Reading of Cartographic Programs

Florian Ledermann
Technische Universität Wien, Department of Geodesy and Geoinformation
Erzherzog-Johann-Platz 1
Vienna, Austria
florian.ledermann@tuwien.ac.at

Abstract

A novel method for analyzing cartographic program code on a detailed level is presented in this paper. The method, loosely based on the *grounded theory* approach, has been applied to a corpus of cartographic programs to yield an empirically-derived taxonomy of cartographic operations, an initial analysis of structural aspects of cartographic programs, and visualizations of cartographic program code.
Keywords: Code, Programming, Taxonomy, Cartographic Transformation, Grounded Theory.

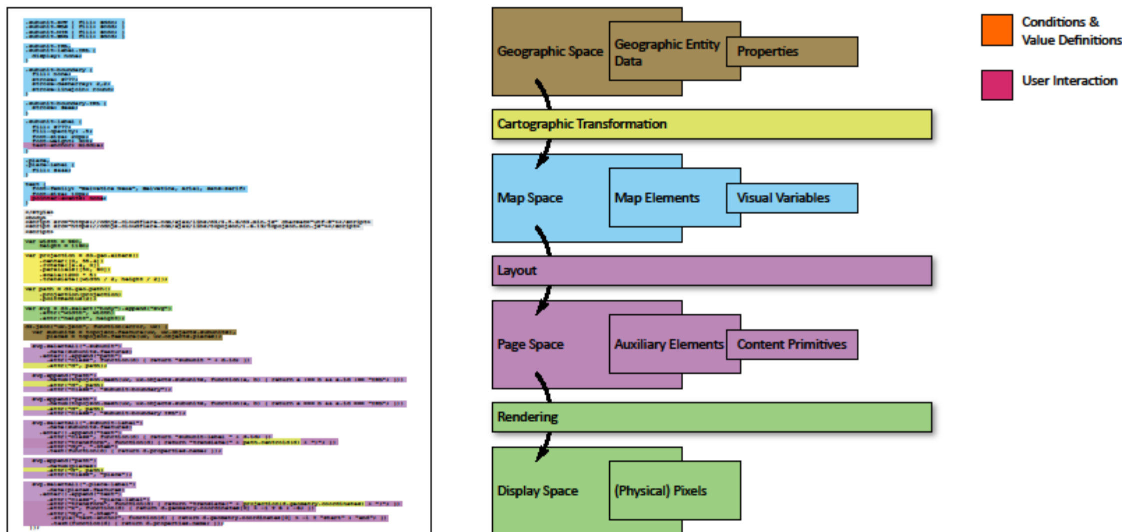

Figure 1: Left: "Let's make a map" tutorial by Mike Bostock, visualized as a "document portrait" using the taxonomy constructed from this research. A schematized representation of the program code is color-coded using the derived taxonomy. Right: Overall conceptual structure of the taxonomy of cartographic elements and operations. Colors match the annotated code to the left.

1 Introduction

Cartographic programming is usually seen as means to an end for creating online and interactive maps, and rarely the main focus of scientific investigation in itself. With the presented research, the focus of attention is shifted to the analysis of cartographic program code. A cartographic program and the required data contain all the technical information to unambiguously (re)create a specific map, and therefore can be seen as an alternative representation of that map in text form. Shifting the attention to the textual representation of the map may open new paths of analysis of cartographic ideas and processes, and ultimately an improved understanding of (digital) cartography.

2 Research method

If cartographic program code is to be made the subject of scientific investigation, research methods are required that support the structured analysis of program code within a conceptual framework of cartography. Computer science has an established tradition of analyzing program code in mostly quantitative ways with the field of "source code metrics" [1], [2] – however, these generic and mostly quantitative measures do not offer any domain-specific insight into cartographic programs. Rather than in the strictly technical structure of the code, we would be interested in the *relevant* parts of the program, the decisions taken, rules implemented and techniques applied from the point of view of cartography. The program code represents a cartographic process, but is sometimes obscured by technical necessities and details – highlighting those purely technical structures will not help us find out about the cartographic process and the core ideas implemented in the program.

The *meaning* of a program's code can only be found out by human interpretation, as for the computer all programs are just abstract manipulation of data. Therefore we need to look towards structured methods of interpretation; in social sciences, such methods have a long tradition when applied to human communication artefacts (written text, interview transcriptions). Specifically, we found the methods provided by grounded theory [3] helpful for analyzing textual artefacts with no clearly established and detailed ex-ante understanding of what will be found. Grounded theory builds on an incremental process of "*coding*"[1], i.e. assigning keywords to

[1] The "coding" activity of grounded theory is not to be confused with programming which is sometimes causally referred to by the same verb

parts of the text, line by line, in close reading sessions, "*memoing*", i.e. taking notes and conceptualizing the findings, and "*sorting*", rearranging the collected codes and memos continuously to form a consistent theory.

In contrast to traditional grounded theory, we could not strictly stick to the rule that no pre-research literature review should be undertaken in order to start with as little preconceptions as possible. This is only practical for studying a field previously unknown to the researcher – being a cartographer and/or programmer, one cannot help but to be aware of the fundamental theoretical concepts in both fields. A rough idea of a cartographic process inspired by a conceptualization of cartography as a sequence of transformations, as already sketched out by Waldo Tobler's idea of a "transformational view of cartography" [4], served as initial scaffolding for the coding phase.

In addition to methodological considerations, a key question is *which* code to analyze. Even if the corpus of programs to choose from is restricted to open source software for practical reasons, the specific choice of code – its functionality, but also the individual style and competence of its creator – would have an overwhelming effect on the results of the analysis. We argue that a careful and conscious choice of example programs can mitigate that methodological problem to some extent; But we remain aware that the choice of example programs has a large influence on the results of the analysis until a large enough body of programs has been analyzed.

Three base technologies have been selected for an initial analysis: The Google Maps API[2], the D3.js visualization API [5] and the Kartograph API[3]. Two strategies have been applied for selecting exemplary programs. A first set of programs has been chosen from the collection of "official" examples found on the APIs' web pages; while these programs may differ greatly in functionality between the individual technologies, we assume that their creators were competent in programming using the respective technology, and that these examples "showcase" the specific strengths of the technology. A second set of programs is currently being compiled to represent "canonical" examples of thematic cartography. A simple choropleth map was the first example we could include in the analysis; in the future, well-known cartographic "standards" will be incuded in the analysis. Where available, implementations by the APIs creators were preferred, in order to assure competence with the technology.

3 Multi-level coding

For annotating the program code of the selected examples, the program MaxQDA[4] was used. MaxQDA is a software for the coding and qualitative analysis of human language (text, interview / video transcriptions); to our knowledge our investigation is the first one – certainly in the domain of cartography – to apply methods from qualitative text analysis to program code.

Programs express a lot of meaning in a very condensed space. When applying the methods discussed in Section 2, we discovered that even a single line of program code can contain functionality on many different levels. For example, the line

```
topojson.mesh(uk, uk.objects.subunits, function(a, b) {
return a !== b && a.id !== "IRL"; })
```

(taken from D3's "Let's make a map" tutorial) performs a geometry conversion on a subset of geometry and filters the geometry to contain only internal borders of the UK – at least three hierarchically nested codings are needed to annotate this single line in detail.

Initially, all of the code was annotated in as much detail as possible; this led to severe cluttering of the interface of the analysis software which had not been designed with such usage in mind, and to an overwhelming amount of detail in the annotation.

We thus followed a multi-level approach: in a first pass, each line or block of consecutive lines is annotated by the "main theme" of its functionality on "level 1". For above example line, this would be "geometry conversion". The main theme of a line of code often corresponds with the outermost language construct (in above example, the function `topojson.mesh()`). By annotating on this level, we get a broad overview of the sequence of operations expressed in the code, without risking "missing the forest for the trees" by a cluttered and overloaded annotation. Figure 1 shows an entire program annotated on the first level; we can clearly distinguish different blocks of code representing different aspects of the program (for example the light-blue block of code assigning visual variables through CSS at the start of the program), without seeing the detailed structure of every line of code.

In a second pass, each line is annotated in as much detail as possible on "level 2". This level provides a detailed analysis of each line of the program, down to individual characters.

```
svg.selectAll(".place-label")
    .data(places.features)
  .enter().append("text")
    .attr("class", "place-label")
    .attr("transform", function(d) { return "translate(" + projection(d.geometry.coordinates)
    .attr("x", function(d) { return d.geometry.coordinates[0] > -1 ? 6 : -6; })
    .attr("dy", ".35em")
    .style("text-anchor", function(d) { return d.geometry.coordinates[0] > -1 ? "start" : "en
    .text(function(d) { return d.properties.name; });
```

Figure 2: Detail of annotated source code on "level 2", showing identified aspects of visual element creation (purple), iteration (orange), cartographic projection (yellow), geodata access (brown), conditions (red) and assigning visual variables (blue)

MaxQDA allows the dynamic filtering of the codes to display, so the level of analysis can be changed in real time once the coding on multiple levels has been completed. In addition, complex queries can be used on the annotated code, for example one could retrieve all lines where a visual variable is assigned (level 1) overlapping with a conditional expression (level 2). This allows for very flexible analysis once the corpus of code is fully annotated.

4 Preliminary results

The close reading and detailed coding of the initial set of programs resulted in a taxonomy of operations and techniques found in these cartographic programs – the codes used to annotate the programs, sorted into a hierarchical structure. The taxonomy currently contains 90 concepts – 66 Leaf nodes and 24 groups of related concepts. In addition to this

[2] https://developers.google.com/maps/documentation/javascript/

[3] http://kartograph.org/

[4] http://www.maxqda.com/

taxonomy of cartographic operations, two smaller taxonomies of technology-specific operations (containing 9 concepts) and code structure (containing 15 concepts) have been extracted from the corpus.

Geodata	81
Cartographic Transformation	27
Visual Elements	261
Auxiliary Elements	4
Viewport Setup	19
Rendering / Pixelization	11
Page Layout	7
Interaction	16
Technology Specific	47
Code Structure	128

Table 1: Highest level of the taxonomy, including the number of occurences summed up across the entire corpus.

In contrast to other taxonomies and ontologies, these concepts have not been extracted by theoretical reasoning or literature research, but are directly derived from empirical evidence in the underlying corpus of annotated programs. For each concept, the number of occurrences in the entire corpus as well as in individual programs can be retrieved. The most frequent concept over the whole corpus is "*selection of visual elements by class or group*" (49 occurences), while for example the concept "*generalization*" occurs only a single time in the current collection.

Visual Elements	
Creation	
From Data	
From Entity Geometry	16
Loading	2
Synthetization	2
Specification	6
Visual Variables	
Size	9
Fill Color	34
Visibility	7
Style	
Text Style	5
Line Style	18
Image / Icon	3
Texture	2
Border	3
Effects	5

Table 2: Part of the "Visual Elements" section of the extracted taxonomy, with frequencies for each concept for the entire corpus of example programs.

In addition to quantitative aggregation, a structural analysis can also be performed on the annotated programs. Structural aspects that were of interest in our analysis would include the sequence of concepts following each other (we hypothesized that programs would follow a directed cartographic process from geodata to projection and transformation to rendering on the screen) or the overlapping or nesting of codes (for example, all parts where a "*condition*" occurs within the assignment of a "*visual variable*", indicating a dynamic assignment of that variable by program logic).

A third way of analysis is available through visualization of the coded programs. A color scheme has been developed to visually link coded parts of the program to distinct steps in the cartographic process described by the taxonomy. Using the color scheme, "document portraits" can be created to give an overview of the structure of cartographic programs. Figure 1 shows the result of such a visualization for a single example program, using the color scheme to identify high-level concepts in the taxonomy.

Besides looking at the entire corpus or individual programs, we would be interested in the comparison of different technologies – for example, do maps implemented in D3 tend to employ different concepts than those realized with tile-based mapping APIs like Google Maps? For an initial investigation, we compared implementations of choropleth maps found in the official examples collection of the three selected technologies. The code of the examples was simplified by removing functionality not present in the other examples, to provide an equal base for comparison.

On the quantitative level, we found large differences between the technologies. The Google Maps implementation was the longest one with 99 lines; 68% more than the 59 lines of the shortest implementation in Kartograph. The number of coded segments (identified concepts mixed in the code) was by far the highest in D3 (101 segments), 80% more than Google Maps (56) and 98% more than the Kartograph sample.

If we set these numbers in relation, we can calculate the average coded segments per line, giving an indication of the density of different aspects of the cartographic pipeline in the code. The D3 example stands out with 1.68 codes per line, nearly 3 times as much as in the Google Maps example.

	Google Maps	Karto-graph	D3
Lines of code	**99 (168%)**	59 (100%)	60 (102%)
Number of coded segments	56 (110%)	51 (100%)	**101 (198%)**
Codes per line	0.57 (100%)	0.86 (153%)	**1.68 (298%)**

Table 3: Quantitative code metrics of the choropleth maps examples in Google Maps, Kartograph and D3.

On the qualitative level we want to try to find out *what* is going on in the program, which aspects of the cartographic pipeline are covered and if there are differences between the technologies. Currently, we have analyzed the sources with regards to the first level concepts of the taxonomy. Only 6 of the high-level concepts are used in the Google Maps example (geodata loading, -transformation, assignment of visual variables, viewport setup, page layout and interaction), while the D3 code uses 8 concepts and the Kartograph-based example 9. Figure 3 shows an overview of the high-level concepts in use in the three different examples.

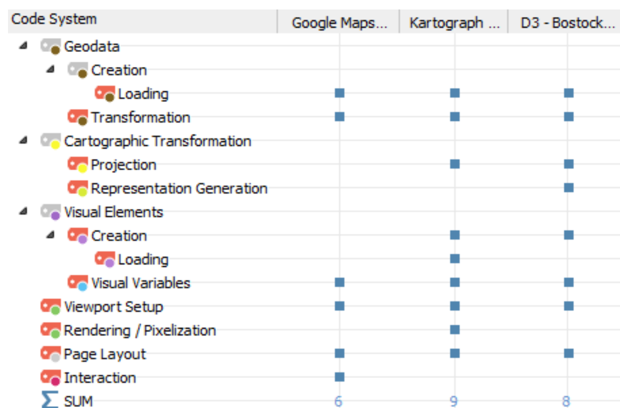

Figure 3: High-level concepts of the taxonomy and their occurrence in examples creating a choropleth map using the selected web cartography APIs.

5 Discussion & Outlook

The work presented here has the potential to improve our understanding of the relationship between program code and cartography on multiple levels. Firstly, a *new method* based on the established method of grounded theory was presented that allows us to analyze the cartographic functionality represented in source code, without having an *a priori* conception about what functionality we are expecting to find and how it is structured. Insights from the presented initial study should be used to further refine and formalize the method – for example, at the moment it is not clear how much the collection of identified concepts is dependent on the person doing the coding; investigating the agreement between different subjects annotating cartographic programs would be a next step to verify and possibly improve the method. A *coding handbook* should be produced to clarify the rules and criteria by which to perform the annotation.

The presented method was used to extract a *taxonomy* of cartographic operations found in a corpus of cartographic programs. In a next step, we would like to relate this empirically-derived taxonomy with other taxonomies from the cartographic literature (e.g. [6], [7]) – are there concepts in our taxonomy that we cannot find the literature (which would indicate that things are *done* in code that we haven't acknowledged in the theory yet)? Which concepts from the literature did we not find in code (which would indicate either an incomplete corpus of examples, lower practical relevance of these theoretical concepts or difficulties putting the theory into practice)?

On a third level, we presented initial insights into the structure of cartographic functionality of the *analyzed code examples*. We found considerable differences in length and number of concepts used between programs implementing a trivial non-interactive choropleth map. These differences seem to support or express the character of the different technologies; for example, D3 has a much higher density of concepts expressed per line of code, supporting the claim that D3 is an "elegant" API that allows the detailed control of all aspects of the output in a very concise way, but also indicating that this may be somewhat overwhelming to novices. More work will be needed to analyze in detail the difference of how cartographic concepts are applied in the different APIs, and to understand the human implications of this, for example in the context of teaching.

The analysis is of course highly dependent on the *corpus* of selected programs. After this initial analysis, a strategy should be developed to build a representative corpus of cartographic programs for further analysis. Since the analysis of programs in such detail takes considerable time, automatic methods for the annotation of at least a part of the taxonomy (e.g. structural aspects like loops, conditions etc.) need to be investigated.

For performing the initial analysis, it was helpful to be able to use an existing mature software package for qualitative data analysis – however, due to the specific structure of program code, which, compared to human language, has a much higher density of concepts and syntactic elements per character, we found the interface of a tool geared towards natural-language analysis partly unsuitable. Also the provided visualization and analysis techniques were only partly applicable to our research. Further investigation in suitable tools for conducting the presented kind of research will be required.

5.1 Acknowledgements

The author wants to thank Georg Gartner and Silvia Klettner for fruitful discussions and helpful suggestions in early stages of this research.

References

[1] J. K. Navlakha, "A Survey of System Complexity Metrics," *The Computer Journal*, vol. 30, no. 3, pp. 233–238, Jan. 1987.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[3] A. Strauss and J. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Second Edition edition. Newbury Park, Calif.: SAGE Publications, Inc, 1990.

[4] W. R. Tobler, "A Transformational View of Cartography," *The American Cartographer*, vol. 6, no. 2, pp. 101–106, 1979.

[5] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[6] R. E. Roth, "An empirically-derived taxonomy of interaction primitives for interactive cartography and geovisualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2356–2365, Dec. 2013.

[7] T. Penaz, R. Dostal, I. Yilmaz, and M. Marschalko, "Design and Construction of Knowledge Ontology for Thematic Cartography Domain," *Episodes*, vol. 37, no. 1, pp. 48–58, Mar. 2014.