MAPMAP.JS: A DATA-DRIVEN WEB MAPPING API FOR THEMATIC CARTOGRAPHY

MAPMAP.JS: uma Aplicação Web Orientada a Objetos para Cartografia Temática

Florian Ledermann & Georg Gartner

Technische Universität Wien
Department of Geodesy and Geoinformation
GuÃŸhausstr. 28-30 / E120 (FB0110), Wien, 1040
florian.ledermann@tuwien.ac.at
georg.gartner@tuwien.ac.at

ABSTRACT

Creating interactive maps for the web is a complex task. In this paper, we present mapmap.js, our approach to designing a high-level API that supports the creation of interactive thematic maps. We discuss the attributes high-level, transparent, data-driven and horizontal with respect to mapping APIs and argue why these are desirable qualities, and give an overview of some aspects of the design of the API. Examples implemented using mapmap.js are presented and discussed in the concluding section.

Keywords: Web Cartography, Thematic Cartography, Mapping APIs.

RESUMO

A criação de mapas interativos para plataformas Web é uma tarefa complexa. Neste artigo, apresenta-se o sistema "mapmap.js" como uma possível abordagem à concepção de um aplicativo de "alto-nível" que dá suporte à tarefa de criação de mapas temáticos interativos. Discute-se conceitos como atributos de alto nível, transparência, orientação a objetos e horizontalidade, todos relacionados ao tema de desenvolvimento de aplicativos para criação de mapas; indica-se, também, porque essas são qualidades "desejáveis" de um projeto como este. Adicionalmente, este trabalho provê uma visão geral sobre aspectos relacionados à construção do aplicativo apresentado neste trabalho. Apresenta-se e discute-se alguns exemplos implementados utilizando o "mapmap.js", no item conclusão.

Palavras chaves: Cartografia Web, Cartografia Temática, APIs de Mapeamento.

1. INTRODUCTION

In the fields of online and interactive cartography, programming is often a major part of the task of creating a map. Such maps are rarely programmed "from scratch" using only low-level functions provided by the execution environment (e.g. operating system or web browser); an API (application programming interface) can provide re-usable parts of code, exposed as a set of high-level abstractions and methods, to the map maker.

Using an appropriate API, the configuration of technologically complex processes can be reduced to single lines of code, and this leverage can be used to implement interactive maps with resources or knowledge that would not be sufficient to create an identical solution with low-level code. The widespread use of APIs related to mapmaking – like the Google Maps API, Leaflet or D3.js – shows the appeal of such tools to mapmakers.

The abstractions and methods provided by an API impose a certain view onto the task of creating

a map – a *mental model* – and therefore shape to a large extent the space of solutions that can be implemented – or that are even considered implementing – using a given technology. Cartographers have often found themselves on the user side of web mapping technologies, with mental models defined by other parties (e.g. software and service vendors, open source developers) and potentially only partially matching the ones of cartographers. Recently, interest in investigating the programming tools used in cartographic production and teaching and how they match the requirements of cartographers has increased.
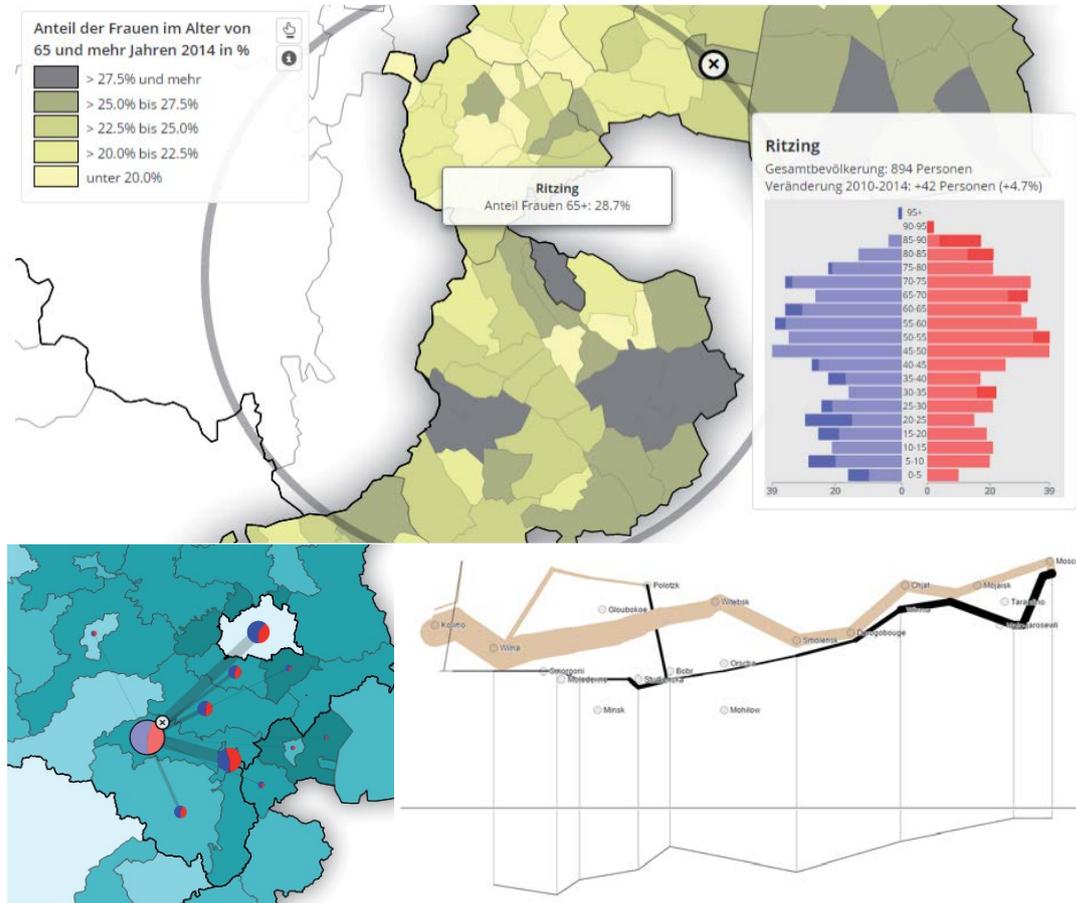

Fig. 1 - Interactive thematic maps created with mapmap.js. (a) A map from the genderATlas project, integrating a zoomable choropleth map with D3.js based data visualizations. (b) Detail of a thematic map showing commuting patterns. (c) Thematic map inspired by Minard's work, linking map and diagram through *anchors*.

The aspect of mental models is of special relevance for teaching. If cartographers do not confine themselves to a consulting role in such technological contexts (and leave the implementation to programmers, computer scientists etc.), the technological foundations for creating maps through code need to be taught in cartography curricula. The concepts employed by the APIs and tools we use for implementing these maps need to be reflected in what we teach – and in order to not let the "tail wag the dog", i.e. let the abstractions of the available technology determine the abstractions we use for reasoning about the problems of our field, it is therefore important to align the technological concepts with the concepts we deem important for advancing the field of cartography.

The mapmap API is our attempt to design an API with an appropriate mental model for interactive thematic cartography. Mapmap was originally developed in the context of an interactive online atlas project, the Austrian genderATlas[1]. The project required a range of different interactive maps and visualizations to illustrate various topics, and a flexible solution was desired to support experimentation and

[1]http://genderatlas.at

iterative development. Initial experiments using D3.js to create the visualizations for the atlas were promising, but even simple interactive maps required a lot of code that had to be written and, eventually, maintained. After a more structured investigation of available technologies and literature on the topic of API design, we decided to create mapmap.js[2], our version of a contemporary cartographic API for creating interactive maps in the browser.

In this paper, we present our approach to a structured design process of a cartographic API. The importance of understanding how we interact with maps through code, and the didactic implications of the increased relevance of creating maps with code have been recognized by the cartographic research community recently. We will look at some previous contributions towards a better understanding of these aspects in section 2. An attempt to condense these findings into general concepts and properties of APIs, and mapping/cartographic APIs in particular, will be presented in section 2.1. Section 3 will discuss how we applied these general principles to the design of mapmap.js, an open source library that provides a cartographic API for creating thematic maps. We will conclude by presenting some examples created with mapmap.js and discussing future plans and challenges in sections 4 and 5, respectively.

The goal of this paper is to provide some input to stimulate a broader discussion of how cartography relates to computer code used to implement interactive geographical visualizations and maps, and how the evolving field of interactive cartography may require new or extended conceptual models. We hope that by presenting our approach to the design of a cartographic API and by making the result of this design process available to other researchers, we can contribute towards a better understanding of, and provide a tool that may be of use in, interactive cartography projects, teaching or research.

2. RELATED WORK

Interest in discussing qualities of web mapping APIs has increased in the cartographic community in recent years. In a recent paper, Peterson compares different "slippy map" APIs (Google Maps API, Leaflet, Mapstraction and others), applying quantitative criteria (execution speed, length of code, cost) to example programs (PETERSON, 2015). Although "cartographic functionality" is listed as an evaluation criterion, no systematic exploration of the functionality of the different APIs and how they relate to cartographical concepts is given. Peterson concludes with a pragmatic position, mentioning cost and long-term viability as possibly the most important evaluation criteria for production contexts – both of which are not related to the functionality provided by the API.

Empirically-oriented researchers would desire quantitative measures like length of code, cost or execution speed to determine the best technology among a number of candidate solutions. However, like cartography itself, programming is a complex creative task that cannot be measured by simple metrics to gauge the success or power of different approaches. To illustrate the argument, imagine a hypothetical "API" that would create a map exactly as needed in a single line of code (Figure 2) – it is easy to see that while it is realistic to create such an API, the result is not something that would be considered a good API, as it would be tailored to a single use case. It is, however, also not generally valid to claim that flexibility is more important than brevity, as using the underlying technologies directly, without the abstraction of an API, will ultimately be the most flexible option (i.e. the option that guarantees to allow all technically possible solutions); There seems to be a more complex payoff between brevity and flexibility, and we need to look to other sources for concepts allowing us to reason about API quality[3].

---

[2] mapmap.js is available online at https://github.com/floledermann/mapmap.js
[3] A similar argument applies to other criteria such as "execution speed" – arguably a faster implementation would be preferred over a slower one. But if the results differ in any aspect, the cartographic quality of the result may be of greater importance than minor differences in execution speed. This payoff cannot be expressed in quantitative terms.

```
var map = createExactlyTheMapThatIWant();
```

Fig. 2 - A hypothetical "perfect" mapping API. This shows that *length of code* is hardly a criterion that facilitates discussions about the quality of mapping APIs.

In their extensive study, Roth et al. (2015) acknowledge the need to look into API usability issues, especially in the context of teaching cartography and web mapping and in the face of rapid ongoing innovation in the field.

Although JavaScript-based web mapping APIs constitute only a small fraction of technologies selected for their initial competitive analysis and needs assessment surveys, all the technologies selected for the final stage of analysis, a qualitative diary study of students implementing web mapping projects, are from this category. In addition to APIs implementing the well-known "slippy map" paradigm, the authors include D3.js (BOSTOCK et al.,2011), a data visualization library that has gained popularity in recent years. These choices can be read as indicating a rising awareness that the "slippy map" paradigm is not sufficient as a universal mental model for the wide range of visualizations cartographers are expected to produce online today. The choice of a qualitative instrument – the diary study – to compare the feasibility of different technologies can be read as acknowledging that a comparison of APIs cannot be done on quantitative factors alone.

Another contribution of Roth et al. is a list of techniques that web mapping solutions can be checked against. The authors group these techniques into the domains of *representation* and *interaction*, drawing from the literature the basic taxonomies of such techniques (Roth, 2013; Slocum et al., 2009). This list of techniques – intended to be extended as our understanding of the field evolves – can be used as a basic framework for initial evaluation of new technologies not included in their study.

The findings of Roth et al. aim to provide reproducible methods for *choosing* suitable candidates among a number of technologies; they do not provide much guideline for *creating* new base technologies to support interactive cartography. For an informed approach to designing a novel web mapping API we looked into the general software engineering literature for advice and guidelines.

Tulach (2012) puts the concept of "selective cluelessness" at the center of his guidelines for API design. For Tulach, an API is a "tool to help us maximize cluelessness", meaning to hide away all implementation details and technological necessities that the API user does not want to deal with or does not have the desire to control in detail. The attribute *selective* must be stressed however – at any time, the user should be able to break through the cluelessness, and adapt or replace the built in methods if she chooses so.

Tulach (2012) also expresses the idea that APIs are not tailored to the needs of a computer, but to a human user:

> *"APIs are verbose, documented, and, in fact, very different from what a computer needs. […] Designing APIs is different. I hadn't been taught to do it and I have a feeling that other programmers haven't been taught to do it either." (TULACH, 2012)*

While the concept of selective cluelessness can be an important and powerful guideline for the task of designing an API, it doesn't provide a lot of analytical depth to reflect about design decisions. In other software engineering publications we find a more differentiated set of concepts to help us reason about how to design a cartographic API.

Stylos and Myers (2007) analyze the conceptual space of API design decisions and discuss qualities that a good API should have. On the highest level, they define the two basic qualities of an API as *usability* ("the qualities of an API that affect its use when creating and debugging code") and *power* (referring to "limits of the code that can be created"). They further differentiate these two basic qualities:

> *Usability includes such attributes as how easy an API is to learn; how productive programmers are using it; how an API prevents errors; how simple it is; how consistent; and how well it matches its users' mental models.*
>
> *Power includes an API's expressiveness (the sorts of programs it can create); its extensibility (how users can extend the API to create convenient user-specific components);*

its evolvability for the API designers who will update the API and create new versions; its performance (in terms of speed, memory and other resource consumption); and the robustness […] of the API implementation. (STYLOS & MYERS, 2007)

It must be noted that all of the individual qualities listed in the category of usability are challenging to verify using scientific methods. In our analysis we decided to focus on the aspect of the supported *mental models* for usability questions, and the aspects of *expressiveness* and *extensibility* for aspects of power. We need to leave detailed investigations of the other aspects to future work.

2.1 Mapping the space of mapmaking APIs

In contrast to evaluating individual algorithms and visualization methods, where objective metrics can be devised that different algorithms or implementations can be evaluated against, we already argued above why such an approach is not available for the question of how code is structuring the overall map creation task. We therefore cannot derive good code – or a good API design – from empiric results alone. Therefore, for initial investigation, we have to turn towards rational argumentation to develop some hypotheses which can later be tested in experimental settings.

Stylos and Myers (2007) point out the importance of the mental model supported by an API. For the core mental model of the popular Google Maps API – show a map of an area, add markers and content to the map – it could be argued that this view does not reflect the mental model of a cartographer, but rather of a map user. For a cartographer, the granularity of the exposed mental model – even if we take into account recently added support for styling the base map through the API – will be frustratingly coarse, and the map at its core has been created by a third party, not by the API user.

So if we label the Google Maps API and other "slippy map" APIs as a *map use* API, what would a *cartographic* API look like? Without claiming to be able to come up with a single unified mental model of cartography, we believe that a cartographic API should represent and give users detailed control over the complete map creation process – accessing raw data and geometry, processing that data and geometry, creating visual representations of geometry and assigning visual variables, and defining methods for users to interact with the map.

In contrast to slippy map APIs, which impose a certain model of what a map is and how it is created, the D3 library allows for detailed control of all steps in the process. One appeal of the D3 is its use of a *data-driven* programming paradigm (Raymond, 2003). Data is not just auxiliary input to a process, but fundamentally affects and drives the logic of the program. This is a concept that aligns well with our understanding of digital cartography.

While some aspects of D3 can be considered to support cartographic mental models (e.g. its built-in support for a wide range of map projections and its data-driven programming paradigm), creating interactive maps with D3 requires detailed knowledge of web technologies in order to accomplish the envisioned result. Tulach's principle of selective cluelessness is not realized in D3 with respect to cartography – all the details of how the map is represented in the browser's document object model (DOM) need to be taken care of by the author, and some of D3's core concepts are not related to cartography but rather to the technological platform of the web browser. A concept that expresses a technology's support for selective cluelessness is its *level of abstraction*. The Google Maps API would be a map-use API with a high level of abstraction; D3 can be considered a low-level API with partial support for cartographic tasks.

Can selective cluelessness be achieved, requiring a high level of abstraction, while at the same time allowing detailed control over all aspects of the mapmaking process if demanded? This apparent contradiction can be resolved by introducing the concept of *transparency*. A transparent API allows for detailed control over the inner workings of its methods, if required. There are several techniques to support transparency in code – e.g. configuration objects, method parameters, callback functions, inheritance – but all allow us to work on a high level of abstraction by default and override the built-in behavior on demand.

A transparent API supporting selective cluelessness would need to provide *sensible*

defaults - even if the user remains clueless, the API should produce a good map with respect to the state of the art in cartography, as far as the information provided allows it. If these default behaviors go beyond simple attributes, but encompass complex behavior, sometimes the term "*magic*" is used in software engineering contexts[4], bearing the negative connotation of users not knowing what is going on behind the scenes.

Another aspect to categorize an API is its scope in the context of a given process. We say an API is *horizontal* if it covers a wide range of aspects of the overall process (like, for example, the steps of an envisioned cartographic process mentioned above), or *vertical* if it is concerned with the details of only a single aspect (like data loading and transformation).

3. DESIGNING THE MAPMAP.JS API

From the related literature in web mapping and software engineering, we took away some important building blocks to inform the design of our cartographic API: The attractiveness of a data-driven approach from D3.js, a list of cartographic features for representation and interaction from Roth et al., the importance of selective cluelessness and, as a corollary, separation of concerns and sensible defaults from Tulach,(2012) and a conceptually structured view on the API design process and possible design decisions from Stylos and Myers' work.

Using the terminology developed in section 2.1, we distilled a list of desirable characteristics of an envisioned API:

Its *mental model* should represent a cartographic process (it should be a *cartographic* API)

It should be *high level* to support "selective cluelessness" and the easy creation of simple maps. Technological details of the platform that are of peripheral relevance to the cartographic design (e.g. asynchronous resource loading, event callbacks, DOM manipulation etc.) should be taken care of by the API internally, even if some "magic" is involved.

At the same time the API should be *transparent*, allowing for control of every detail in the process if so desired by the author.

The process to create the map should be *data driven* and the API should be able to work with real-world data. It should not be necessary to convert data into a special format or representation.

It should be a *horizontal* API, encompassing the complete process of rendering an interactive map from raw data, which comes with sensible defaults and "batteries included".

Overall, code using the API should meet the criteria expressed in a quote attributed to Alan Kay: "Simple things should be simple, complex things should be possible".

3.1 General Principles

Mapmap.js assumes a mental model of the overall cartographic process as a sequence of inputs and transformations. At the core of the process is the transformation of geographic features, which can be loaded from a data file or generated algorithmically, into visual representations. Features may be augmented with auxiliary data, such as data loaded from a CSV file. Metadata can be added to specify the properties of the data. Visual variables are derived from the geometry, data and metadata, and the visual representations (SVG elements) of features are created. Finally, interaction methods can be added to representation elements.

The smallest units of code in mapmap.js are *functions*. With its powerful and concise syntax for function literals and closures, JavaScript affords a functional programming style in which functions are not only used as procedures and methods, but also as parameter values for other functions. Cartographic tasks often follow a model of output deduced from a set of input data and parameters, making a function a suitable abstraction to serve as basic building block for map composition. As it is not always feasible to construct inline functions for trivial tasks, e.g. property lookup, most parameters support polymorphism in a standardized way across the API: a parameter can either be a string (to access a field of the given name), an Array (to perform the operation for multiple properties) or a function (for completely customizable lookups, calculations and conversions). In addition, mapmap.js offers a library of helper

---

[4]See http://en.wikipedia.org/wiki/Magic_(programming)

functions for convenient access to frequently used functionality.

3.2 Geometry, Data and Metadata

Data to render digital maps is usually available in one of various geodata formats (e.g. GeoJSON or TopoJSON), or as auxiliary data (e.g. CSV files or Excel sheets) that can be linked to geodata entities[5]. The fundamental method of the mapmap API to add geodata is *.geometry(source, options)*. Auxiliary data can be associated with the geometry with subsequent calls to *.data(source, options)*. Both methods accept a source URL or an array of data as their first parameter, and an optional parameter specifying the options for processing and joining the data.

A powerful paradigm for data processing is the MapReduce programming model (DEAN & GHEMAWAT, 2008). Mapmap.js supports applying MapReduce-based transformations to the data or geometry as an option. We found the MapReduce model to be a powerful paradigm allowing for various transformation and aggregation tasks necessary for working with real-world datasets.

To support rendering the data using appropriate symbology, or to create legends or interactive explanations on the map, we need information about the data fields associated with features, or *metadata*. Metadata is specified using *.meta(spec)*, passing an associative array to map field names to metadata descriptors. To assign identical metadata to multiple fields, wildcards can be used to match field names.

Metadata is used by symbolizers (see section 3.3) to assign visual variables – for example, the *domain* and *colors* metadata properties are used by the choropleth symbolizer to assign color values to geometries. Other fields in the metadata descriptor like *label* or *numberFormat* are used whenever human-readable representations of the data need to be displayed.

3.3 Representing Map Objects: Visual Variables, Symbolization and Selections

The task of visualizing data and geometry is modeled as a two-step process in mapmap:

assign visual attributes using geometry, data and metadata, and create a visual representation using the geometry and visual attributes. This two step process allows for the simple representation of simple tasks (e.g. assign a fill color attribute from a data value to create a choropleth map using the default symbolizer) as well as the implementation of more complex tasks that involve modifying the geometry, like generalization or the creation of abstract visualizations.

For operations that should affect only parts of the map, mapmap.js provides two ways to express selections: either as a filter function returning a Boolean result for each feature, indicating whether the object should be selected or not, or as a string, that is used as input to a global identify function, selecting objects based on matching field values. The selection of objects to operate on can be defined globally or in the options of each method call.

Additional methods are available for adjusting the maps position and size in relation to the viewport (the area on screen containing the map) and adding user interaction methods to elements of the map.

3.4 Connecting the Perimap: Legends & Anchors

Maps do not exist in isolation – they are embedded in the *perimap*, the information surrounding & accompanying the map, like legends and supplementary diagrams (WOOD et al., 2009). The creation of individual elements of the perimap is outside the scope of the mapmap API; however, it would be desirable for a cartographic API to provide means to express a map's relation to perimap elements where necessary.

Two standard elements of the perimap are legends and annotations. Legends can take many shape and forms, and detailed approaches exist to render legends or legend components from map data (DYKES et al., 2010; JENNY et al., 2009). It would be therefore out of scope for a mapping API to fully support creating all possible kinds of legends. However, as the legend relates to the map and its underlying data, it would be desirable to be able to express this relationship and extract the information necessary for rendering map

---

[5] Note that in applications dealing with geometry as first-class data, such auxiliary data is sometimes called "metadata". In mapmap.js, the term metadata is used for data describing auxiliary data.

legends through the API, and to keep map and legend in sync in case the map changes.

In mapmap.js, *legends* are conceptualized as visualizations of the map's metadata. A legend or part of a legend for a given data field is defined as a function, taking the field's metadata as its main argument. The metadata object provides methods to access statistical data (such as minimum and maximum values, or a histogram of the value distribution). Two built-in legend generation functions are provided for rendering simple legends in HTML or SVG – for rendering application specific and more advanced legends, a user-created function can be supplied.

Annotation overlays and links to external diagrams are supported through the concept of *anchors*. Anchors are functions mapping arbitrary data objects to pixel locations on the map. The default anchoring function provided by mapmap takes a key value as input and returns the centroid of the visual representation of the feature matching the key, if present. Again, this can be changed to a user-supplied anchoring function, potentially performing more complex geometry calculations or application dependent anchoring of data objects. A built-in method that makes use of anchors is *.hoverInfo( )*, that sets up the dynamic display of object information in a "popup" box at the anchor location upon mouse interaction.

4. RESULTS

Stylos and Myers (2007) sketch out a general framework of evaluating APIs establishing two main qualities of APIs, *usability* and *power*. Regarding the usability aspects, we have sketched out the mental model the API supports and argued why it is a mental model suitable for cartographers. Regarding the APIs power, its expressiveness and extensibility is demonstrated by selected examples of interactive maps that have been created with mapmap.js[6].

In the genderATlas project, we successfully used mapmap.js to implement a range of different customized interactive statistical maps of Austria, using only a few lines of code to specify the basic map setup and interaction for each thematic map used in the atlas. Depending on the story, specific functionality had to be implemented for visualization or interaction; this code is related to the individual map, and mapmap.js provides a framework for how special-purpose functionality is integrated in a modular fashion into the overall map implementation, avoiding "spaghetti code". Also, some of the built-in functionalities have helped to write code on a higher level of semantics – for example, to zoom in to one of Austria's counties, the name of the county can be specified instead of an id code, making the code more readable to outsiders.

Besides the visualizations developed for the genderATlas project, we are currently looking at "canonical" thematic maps and geovisualizations and try to recreate them using mapmap.js to verify its power to express the state of the art in interactive cartography. One classic – historic – example is Minard's map of Napoleon's Russian campaign, a widely known thematic map that has been extensively analyzed in the cartographic literature (For a recent overview, see Kraak (2014)). Interactive and non-interactive versions of Minard's map have been implemented by various authors using different programming languages and APIs, making it a de-facto benchmark for verifying and comparing mapmaking APIs (FRIENDLY, 2002). We created a simple version of Minard's map (see Figure 1.c) in 20 lines of code, excluding the temperature diagram which has been implemented using D3.js using 43 lines of code. A key part of our implementation is the connection between the map and the diagram through an *anchoring* function mapping longitude values of the temperature data to pixel coordinates on the map, which adds another 10 lines of code.

As argued in section 2, counting line numbers cannot be a valid quality metrics by itself – however, the mental model of the mapmap API allows authors to express the *necessary structure* of a map in a concise way, while at the same time allowing them to improve each individual aspect of the map incrementally by exploiting the transparency of the API.

---

[6]You can find live examples linked online at https://github.com/floledermann/mapmap.js

```
var map = mapmap('mapEl')
    .geometry('../data/austria.topojson', 'iso')
    .data('income_AT.csv', 'code')
    .meta({
        'income_med': {
            label: "Median Income, 2012",
            valueLabel: "Median Income"
        }
    })
    .select('bezirke')
    .choropleth('income_med')
    .hoverInfo(['name','income_med'])
    .applyBehaviour(mapmap.interactions.zoom())
    .legend(mapmap.legend.html());
```

```
function isInBurgenland(d) {
    return d.iso && d.iso[0] == '1';
}

var map = mapmap('mapEl')
    .geometry('../data/austria.topojson', 'iso')
    .data('income_AT.csv', 'code')
    .meta({
        'income_med': {
            label: "Median Income, 2012",
            valueLabel: "Median Income",
            domain: [1200,3200],
            colors: colorbrewer.YlOrRd[5],
            numberFormat:'0,000'
        }
    })
    .extent(isInBurgenland)
    .highlight(isInBurgenland)
    .choropleth('income_med')
    .hoverInfo(['name','income_med'])
    .legend(mapmap.legend.html());
```
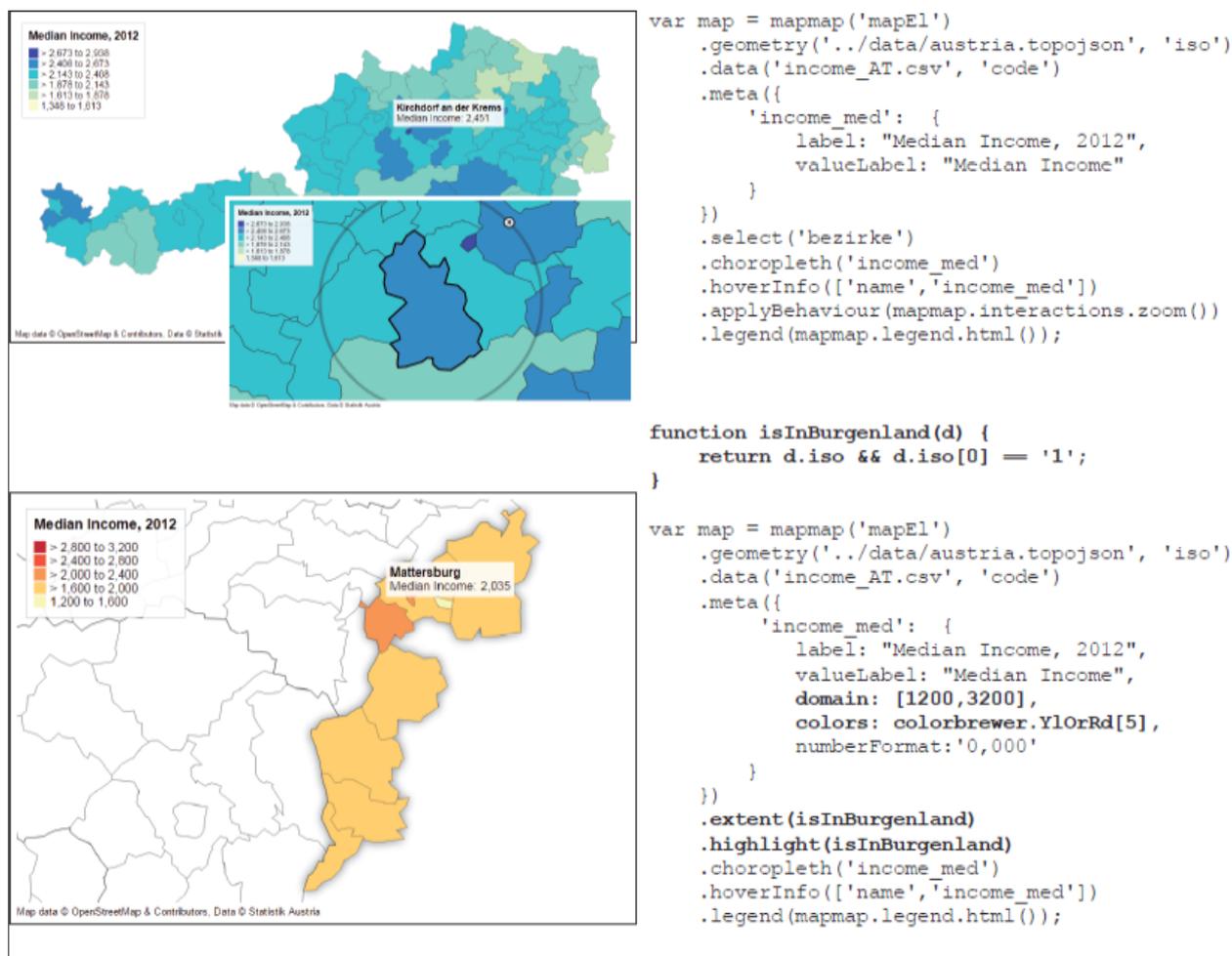
Fig. 3 - Two simple maps with their full source code using the mapmap API. Both maps use the same geometry and data files; the bottom map is configured to show only districts within the county of "Burgenland" (ISO codes starting with "1").

5. CONCLUSIONS

In this paper, we have presented a structured approach to designing and analyzing mapmaking APIs. Through the concepts of *mental models*, *level of abstraction*, *transparency* and *horizontal vs. vertical* APIs we hope to have offered some terminology that can help researchers structure the space of mapmaking APIs, and we have presented our design of mapmap.js, our attempt to create a high-level, transparent, horizontal cartographic API.

On a pragmatic level, we hope that by releasing mapmap.js as an open source API we can provide a tool that may be helpful to other map makers, and that can serve as a technological artifact stimulating further discussion about the implementation of cartographical concepts in code.

5.1 Future work

The work on mapmap.js has spurred ideas

for further investigations on different levels. On the level of cartographic theory, we believe there is potential for improving our understanding of how program code relates to theoretical concepts of cartography, and how implementations of such concepts as *mental models* can potentially help to deepen our understanding of cartography. For example, the issue of how elements of the perimap are anchored to and by the map would need further investigation both in theory and in the practices of mapmaking and programming.

On a more pragmatic level, we are currently working on a more formal evaluation of the mapmap API, both in terms of comparing it to other technological approaches and in terms of objective measures for issues of API power and API usability. Performing such evaluations is methodologically challenging, since, as argued in section 2, we cannot rely on quantitative measures to judge the quality of code. We are currently investigating methods to verify some

of the aspects of API usability (ease of learning, programmer productiveness, error prevention, simplicity, consistency), and hope to be able to gain insights by using mapmap.js in teaching interactive cartography courses, comparing it with other APIs in a controlled process informed by Roth et al.

On the engineering level, there are many ideas for improving the mapmap API and its implementation, including features like "pluggable" rendering engines, to separate the technology used for representing the map output (currently SVG) from the API and to be able to support different output technologies (e.g. canvas-based rendering or 3D maps through WebGL). Other plans include improved viewport management and map layout, the integration of tile-based raster backgrounds or an improved model for modular interaction.

ACKNOWLEDGEMENTS

REFERENCES

BOSTOCK, M., OGIEVETSKY, V., HEER, J. D3 : Data- Driven Documents. **IEEE Transactions on Visualization and Computer Graphics** 17, 2301–2309pp. 2011. doi:10.1109/TVCG.2011.185

DEAN, J.; GHEMAWAT, S., 2008. MapReduce: Simplified Data Processing on Large Clusters. Commun. **ACM** 51 , 107– 113 pp. 2008. doi:10.1145/1327452.1327492

DUBOCHET, G. Computer Code as a Medium for Human Communication: Are Programming Languages Improving? **Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group.** 174–187pp 2009.

DYKES, J., WOOD, J., SLINGSBY, A. Rethinking Map Legends with Visualization. **IEEE Transactions on Visualization and Computer Graphics** 16, 890–899pp. 2010 doi:10.1109/TVCG.2010.191

FRIENDLY, M. Visions and Re-Visions of Charles Joseph Minard. **Journal of Educational and Behavioral Statistics** 27, 31–51p. 2002.

JENNY, B., HUTZLER, E., HURNI, L. Self-Adjusting Legends for Proportional Symbol Maps. **Cartographica: The International Journal for Geographic Information and Geovisualization** 44, 301–304pp. 2009. doi:10.3138/carto.44.4.301

KRAAK, M.-J., 2014. **Mapping Time: Illustrated by Minard's Map of Napoleon's Russian Campaign of 1812**, Redlands: 1 edition. ed. Esri Press. 2014.

PETERSON, M.P. Evaluating Mapping APIs, in: BRUS, J.; VONDRAKOVA, A.; VOZENILEK, V. (Eds.), **Modern Trends in Cartography, Lecture Notes in Geoinformation and Cartography**. Springer International Publishing. 2015. pp. 183–197.

RAYMOND, E. S. **The Art of UNIX Programming**. Boston: Addison-Wesley. 2013. 560p.

REEVES, J.W. What is software design. **C++ Journal** 2, 14–12pp. 1992.

ROTH, R.E. An empirically-derived taxonomy of interaction primitives for interactive cartography and geovisualization. **IEEE Transactions on Visualization and Computer Graphics** 19, 2356–2365pp. 2013. doi:10.1109/TVCG.2013.130

ROTH, R.E.; DONOHUE, R.G.; SACK, C.M.; WALLACE, T.R.; BUCKINGHAM, T.M.A. A Process for Keeping Pace with Evolving Web Mapping Technologies. **Cartographic Perspectives**, 78. 25-52pp. 2015. doi:10.14714/CP78.1273

SAMPLE, J.T.; IOUP, E. **Tile-Based Geospatial Information Systems**. New york: Springer US. 2010. 237p.

SLOCUM, T.A.; MCMASTER, R.B.; KESSLER, F.C.; HOWARD, H.H., 2009. **Thematic Cartography and Geovisualization**. New Jersey: Prentice Hall. 2009. 576p.

STYLOS, J.; MYERS, B. Mapping the Space of API Design Decisions, in: **IEEE Symposium on Visual Languages and Human-Centric Computing**, 2007. VL/HCC 2007. Presented at the IEEE Symposium on Visual Languages

and Human-Centric Computing, 2007. VL/HCC 2007, pp. 50–60. doi:10.1109/VLHCC.2007.44

TULACH, J. **Practical API Design: Confessions of a Java Framework Architect**. New York: ed. Apress, New York. 2012. 416p.

WOOD, D., FELS, J., PICKLES, J. **The Natures of Maps: Cartographic Constructions of the Natural World**. First Edition edition. Chicago: ed. University Of Chicago Press. 2009. 231p.

WOOD, J., DYKES, J. Spatially Ordered Treemaps. **IEEE Transactions on Visualization and Computer Graphics** 14, 1348–1355pp. 2008. doi:10.1109/TVCG.2008.165