Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Pyry Kröger

Visualizing Geographical Data on the Web

Reducing the work needed by eliminating boilerplate

Master's Thesis
Espoo, May 28, 2014

DRAFT! — October 10, 2014 — DRAFT!

Supervisor: Professor Petri Vuorimaa
Instructor: N.N. M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Pyry Kröger

Title:
Visualizing Geographical Data on the Web - Reducing the work needed by eliminating boilerplate

Date:	May 28, 2014	**Pages:**	vii + 42
Professorship:	Media	**Code:**	T-111
Supervisor:	Professor Petri Vuorimaa		
Instructor:	N.N. M.Sc. (Tech.)		

A dissertation or thesis is a document submitted in support of candidature for a degree or professional qualification presenting the author's research and findings. In some countries/universities, the word thesis or a cognate is used as part of a bachelor's or master's course, while dissertation is normally applied to a doctorate, whilst, in others, the reverse is true.

!FIXME **Abstract text goes here (and this is an example how to use fixme).** FIXME! Fixme is a command that helps you identify parts of your thesis that still require some work. When compiled in the custom `mydraft` mode, text parts tagged with fixmes are shown in bold and with fixme tags around them. When compiled in normal mode, the fixme-tagged text is shown normally (without special formatting). The draft mode also causes the "Draft" text to appear on the front page, alongside with the document compilation date. The custom `mydraft` mode is selected by the `mydraft` option given for the package `aalto-thesis`, near the top of the `thesis-example.tex` file.

The thesis example file (`thesis-example.tex`), all the chapter content files (`1introduction.tex` and so on), and the Aalto style file (`aalto-thesis.sty`) are commented with explanations on how the Aalto thesis works. The files also contain some examples on how to customize various details of the thesis layout, and of course the example text works as an example in itself. Please read the comments and the example text; that should get you well on your way!

Keywords: work, in, progress

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Pyry Kröger		
Työn nimi:	Geografisen datan visualisointi webissä - Tarvittavan boilerplate-koodin vähentäminen		
Päiväys:	28. toukokuuta 2014	**Sivumäärä:**	vii + 42
Professuuri:	Media	**Koodi:**	T-111
Valvoja:	Professori Petri Vuorimaa		
Ohjaaja:	Diplomi-insinööri N.N.		

Kivi on materiaali, joka muodostuu mineraaleista ja luokitellaan mineraalisisältönsä mukaan. Kivet luokitellaan yleensä ne muodostaneiden prosessien mukaan magmakiviin, sedimenttikiviin ja metamorfisiin kiviin. Magmakivet ovat muodostuneet kiteytyneestä magmasta, sedimenttikivet vanhempien kivilajien rapautuessa ja muodostaessa iskostuneita yhdisteitä, metamorfiset kivet taas kun magma- ja sedimenttikivet joutuvat syvällä maan kuoressa lämpötilan ja kovan paineen alaiseksi.

Kivi on epäorgaaninen eli elottoman luonnon aine, mikä tarkoittaa ettei se sisällä hiiltä tai muita elollisen orgaanisen luonnon aineita. Niinpä kivestä tehdyt esineet säilyvät maaperässä tuhansien vuosien ajan mätänemättä. Kun orgaaninen materiaali jättää jälkensä kiveen, tulos tunnetaan nimellä fossiili.

Suomen peruskallio on suurimmaksi osaksi graniittia, gneissiä ja Kaakkois-Suomessa rapakiveä.

Kiveä käytetään teollisuudessa moniin eri tarkoituksiin, kuten keittiötasoihin. Kivi on materiaalina kalliimpaa mutta kestävämpää kuin esimerkiksi puu.

Asiasanat:	vähän, vielä, kesken
Kieli:	Englanti

Acknowledgements

So long, and thanks for all the fish.

Espoo, May 28, 2014

Pyry Kröger

Abbreviations and Acronyms

API	Application Programming Interface
POI	Point of Interest; a piece of data with geospatial dimension
GIS	Geographic Information System

Contents

Chapter 1

Introduction

!Fixme **Here maybe something related to why I chose this topic, i.e. why it is beneficial for me/others to have a framework that eases the map visualization process.** Fixme!

Geographical data is data with geospatial dimension, such as POI with location data as coordinates. The most natural method for visualizing geographical data is usually with various maps. In the past, geographical data was predominantly visualized by cartographers, but it has been recognized (Kraak and MacEachren, 1999) that the situation has changed, with people from increasing number of fields having a need for visualizing geographical data. Moreover, the popularity of Google Maps (Google, 2005b) along with its API (Google, 2005a) has proved that in addition to experts of other academic fields, there is a definite demand for web map visualizations within consumers as well.

The web makes publishing and bundling map visualizations extraordinarily straightforward when compared to traditional desktop-based GIS applications, which is especially important when the visualizations are made by non-cartographers who only make visualizations occasionally and lack the needed resources and experience for more complex publishing process. !Fixme **find a reference for this maybe** Fixme!. However, as the web platform is primarily designed for static documents instead of dynamic applications (Berners-Lee, 1989; Berners-Lee et al., 1992), there are some additional concerns to address when making a complex data visualization on the

web.

!FIXME **Why geodata should be visualized? Kraak and Ormeling (2011, chap. 1.1) or Bartz Petchenik (1979) should help.** FIXME!

!FIXME **Needs more flesh** FIXME!

1.1 Problem Statement

Currently, there are several libraries available for displaying maps and simple visualizations !FIXME **add references to the libraries. GMaps, Leaflet, OpenLayers etc.** FIXME!. However, none of the mainstream libraries is of sufficiently high abstraction level for building map visualizations effectively, resulting in the need for writing *boilerplate* code that does not directly contribute to the visualization. Moreover, the libraries are not designed primarily for visualizations and therefore do not encourage or push the visualizer to create visually and cognitively effective visualizations.

In the scope of this thesis, we make the difference between an efficient and an effective process. By an efficient process, we mean a process which requires as little as possible effort and other resources to complete. By an effective process, we mean a process which reaches its target results sufficiently. Therefore, building a visualization efficiently indicates that the building process is as effortless as possible, and building effective visualization indicates that the resulting visualization conveys its intended message appropriately.

We plan to evaluate means to make creating map visualizations for the web more efficient by building a higher abstraction level software framework for map visualizations. This framework should provide the structure for creating the visualization as well as common web application features needed in modern web applications.

In order to find the solution for the problem, it is necessary to study geographical visualizations and software frameworks. The process of making geographical visualizations should be studied to ensure that the framework enables creating *effective* visualizations. In addition, software frameworks and software reuse should be studied in order to be able to create visualizations *efficiently*. Therefore, we select the following research questions for this thesis:

RQ1 How to build effective geographical visualizations in the web?

RQ2 How to build a web software framework which enables creating effective geographical visualizations efficiently?

!FIXME **Would it make sense to drop the RQ1 since RQ2 pretty much covers it? Or rephrase the questions to overlap less?** FIXME!

1.2 Structure of the Thesis

!FIXME **This section is an early draft and will change considerably.** FIXME! Chapter 1 (this introduction) presents the motivation for this thesis as well as the problem statement. Chapter 2 describes the background of the work. In particular, the chapter describes how to visualize geographical data and the essence of web software frameworks. Chapter 3 presents the web technology, standards and other needed material for building a web visualization as well as describing some existing map visualizations.

In chapter 4, we discuss the methods used to examine the problem and evaluate solutions we will propose. In chapter 5, we describe the methods used to solve the problem. In chapter 6, we evaluate the implementation and its results. !FIXME **Chapters 7 and 8 missing. Also, maybe elaborate description about chapters 4-6 a bit** FIXME!

!FIXME **Rewrite. No active passive mixing etc.** FIXME!

In order to build an efficient framework for visualizing geographical data, it is needed to study (a) how to visualize geodata and (b) how to build frameworks. We are going to tackle this problem by first studying the basics of data visualization with an emphasis on geographical data, maps and the visualization process. After visualization, we are going to study the essence of software reuse, focusing on building and evaluating reusable software, also known as software frameworks.

Chapter 2

Data Visualization

2.1 Definition

According to Kosara (2007, chap. 3), there is no universally accepted definition of visualization. He proposes the following for a "minimal set of requirements for any visualization":

- It is based on (non-visual) data
- It produces an image
- The results are readable and recognizable

According to him, while visualizations can also have other properties or qualities, such as interaction or visual efficiency, the requirements above are the ones needed for technical definition of the term. Moreover, it should be emphasized that according to this definition, visualization is the *process* itself, not the result of it.

Kosara (2007, chap. 4) argues that visualization is separated into two types, *pragmatic* and *artistic* visualization. Pragmatic visualization focuses on the analysis of the data in order to show its relevant characteristics as efficiently as possible. Artistic visualization on the other hand concentrates on the communication of the concern behind the data, not the display of the actual data. Kosara states that while these types focus on the opposite sides

of the visualization spectrum, it may be possible to close the gap using e.g. interaction.

The first requirement for visualizations by Kosara (2007) dictates that the visualization is based on data. This is an essential characteristic of *data* visualizations: the visualization is a function which takes data as an input and produces a visual object as an output. In less technical terms, this means that the visualization turns data into visual, effortlessly and efficiently digestible format.

This leads to the fact that the data and visualization are not inherently tied to each other; the visualization "function" can be independent of the data and thus it may be possible to create a visualization framework or platform which is able to function on a potentially wide range of data.

!FIXME **The goal of visualization is (usually) better understanding of the data.** FIXME!

2.2 Principles for Successful Data Visualization

The requirements presented in the previous section are sufficient for the definition of data visualization. However, they do not convey any information about visualization quality. In order to discover the characteristics for successful data visualization, additional principles are needed. Tufte (1986, p. 13) states that excellent graphics (i.e. results of visualizations) consist of "complex ideas communicated with clarity, precision and efficiency". In practice, this means that the graphics should emphasize the actual data and its nuances above everything else, while serving a clear purpose.

In addition to graphics principles presented in the previous paragraph, Tufte (1986, p. 93) presents the concept of *data-ink*. Data-ink represents the ink used for displaying the data in a visualization. He argues that in an excellent visualization, most, if not all, ink used should contribute to display of the data. However, research by Inbar et al. (2007) suggests that maximizing the share of data-ink may not be beneficial to the user experience of the visualization.

The principles presented above are essential, but too abstract in order to be used as a sole basis for defining a good visualization. However, when combined with the data visualization definition stated above, the principles become considerably more useful and concrete. Azzam and Evergreen (2013) propose an adapted version of the definition by Kosara (2007), complementing the second requirement by requiring the produced image to represent the data truthfully. This definition effectively combines the definition by Kosara (2007) with the principle of showing data introduced by Tufte (1986). The adapted definition facilitates the process of creating a successful data visualizations by offering a more concrete version of Tufte's principles. It gives the developer of the visualization a concrete checklist for representing the data: make sure it does not (a) omit or (b) overrepresent any information (Azzam and Evergreen, 2013).

!Fixme **If desperately in need of more background, add human perception in relation to information visualization (from Ware).** Fixme!

2.3 Visualizing Geographical Data

!Fixme **Visualization - Scientific Visualization - Map Visualization - Kraak (1998) has a good overview.** Fixme!

The most natural way of visualizing geographical data is by using a map(Kraak, 1998; Kraak and Ormeling, 2011, chap. 1). This technique is called *thematic mapping* (Slocum and McMaster, 2014, chap. 1). Thematic mapping does not require any specific format of data, except for the geographical dimension (Kraak and Ormeling, 2011, chap. 1). However, the nature of the data has a great effect on the method, or type, of thematic mapping.

2.3.1 Methods for Thematic Mapping

As stated above, there are several types of geographical data, many of which are fundamentally different requiring different visualization methods. Therefore, several different thematic mapping methods have been developed. Slocum

and McMaster (2014, chap. 14-18) list some of the most typical ones: !FIXME **rephrase the descriptions below** FIXME!

- Choropleth map - Shows data aggregated for a set of predefined areas (countries, regions etc)
- Isarithmic map - Maps with areas separated by contour lines.
- Dasymetric map - !FIXME **what's this?** FIXME!
- Proportional symbol map - like dot map, but replaces dots with relevant symbols of sizes proportional to the data
- Dot map - simple maps with dots on relevant locations
- Multivariate mapping - A map that shows data of several dimensions (in addition to location).
- Flow map - a map that shows "flows" from one area to another. Napoleon Russian campaign map.

When designing a software framework for geovisualization, it is not necessary to support every method above. However, as those are some of the most used ones, omitting any must be a conscious decision.

!FIXME **describe the use cases for each type** FIXME!

Even a single thematic map is often used for multiple different purposes (Schlichtmann, 2002, chap. 2). For instance, a single map can be read on the *overall level* ("where are the primary schools located in Helsinki metropolitan area?") and *elementary level* ("is there a primary school in Punavuori?"). Furthermore, some possible uses for a thematic map are "what is the ratio and distribution of Finnish schools compared to Swedish schools in Helsinki" or "what is the spatial distribution of sizes of schools in Helsinki". Therefore, an efficient map visualization should not lock the user to any single perspective. !FIXME **maybe move somewhere? Create a separate (sub)section for interactivity?** FIXME! !FIXME **Interactivity could help here? See (Andrienko and Andrienko, 1999)** FIXME!

!FIXME **Analysis part of geovisualization - "...is considered out of the scope of this work."** FIXME!

!FIXME **To Do** FIXME!

- Geographic visualization (e.g. in relation to scientific visualization)
- Map visualization vs. map (thematic map vs general-reference map) (Bartz Petchenik, 1979)
- How do the principles introduced in the previous subsection apply to geographical data? Is there anything else to consider?
- Thematic map interactivity (Andrienko and Andrienko, 1999) (Where should this go? Is this an env related thing or here somewhere)

Tufte (1986, p. 16) May help here as well.

2.4 How Thematic Maps Are Made

Schlichtmann (2002) describes making thematic maps as a six-step process. The first four steps involve deciding on and obtaining the data which are not relevant when building a software framework for visualization. Therefore, we ignore those steps. The step five consists of selecting the visualization method and using it to produce a meaningful visualization from the data. The step six involves explaining the visualization in legend. !FIXME **Schlichtmann not concerned with the step 6, find some other source of displaying the legend.** FIXME!

In the map visualization process, several identified objectives for the resulting graphic exist (Schlichtmann, 2002). The objectives are presented in the table below.

Name	**Description**
Clarification	Making the map clear and readable. In practice, this means that the topemes (symbols) in a map should be easily detectable and distinguishable from each other
Emphasis	Making topemes and other important characteristics of the visualization to stand out visually

Types of Entries	Having a clearly distinguishable type for each topeme.
Sets of Types	Grouping data points and symbols with similar traits in order to make them belong together visually. Ideally, the visual similarity should be related to the conceptual similarity.
Cross-Relations	Visually indicating the potential relations and similarities between different types or between entries of different types.
Local Syntax	Aligning visual properties of the topemes to prevent unintentional emphasis of single topemes.
Local Ensembles	Supporting topemes with multiple properties (such as the numbers of children and adults in an area) so that the topeme visually reflect both the individual properties and the combination of all properties.
Multilocal Ensembles	Supporting topemes with multiple geographical properties (such as spatial distribution of people)
Addable and Non-Addable Quantities	Differentiating addable and non-addable properties. Typically absolute quantitative properties are addable while relative and qualitative properties are non-addable. Addable properties should be visualized in a way that cognitively supports addition (e.g. with sizes of elements) while non-addable quantities should be visualized without said feature (e.g. with colors.)
The Surface Illusion	Creating an illusion of surface on the map. This can be achieved for example by using illumination and shadowing. These visual traits can convey a meaning themselves and often naturally do so.

Table 2.1: Map visualization objectives as per Schlichtmann (2002)

The objectives above are important when visualizing geographical data on a map. Therefore, it is needed to take those into account when creating a visualization tool or framework in order to enable or even encourage the

visualizers to reach as many of the objectives as possible.

!Fixme **This could use an opinion from some other source as well (Maybe (Slocum and McMaster, 2014, p. 5))** Fixme!

Chapter 3

Software Reuse

In order to create a reusable software framework for visualization, it is necessary to study software reuse along with different reuse techniques and their characteristics, advantages and disadvantages.

Krueger (1992) presents software reuse as a process of reusing existing software code (applications, libraries, functions or single lines) when building new software, while according to Mohagheghi and Conradi (2008), reuse is not restricted to code, but can also refer to other software assets such as design. However, both agree that software reuse combines several different existing pieces of code (and possibly other assets) along with new assets which are specific for the application in question. According to Mcilroy (1969) and Boehm (1999), it is one of the most effective techniques of reducing the development time and cost of complex software products.

3.1 Software Reuse Advantages & Disadvantages

When used appropriately, software reuse has several benefits. In their overview of multiple case studies, Mohagheghi and Conradi (2008) discovered that in most cases, using reused software components resulted in a considerably lower number of software defects and better productivity. Several of the studies implied that reusing software is also beneficial for software complexity and product time-to-market. However, it should be noted that since the overview

only addresses case studies, its results should not be considered universally applicable.

Although reusing software is often said to decrease the effort needed (Mcilroy, 1969; Boehm, 1999; Mohagheghi and Conradi, 2008) !FIXME **there must be many more sources for this available** FIXME!, concrete evidence for this is difficult to find (Mohagheghi and Conradi, 2008).

Given its lucrative advantages, software reuse is definitely beneficial for many software systems. However, according to Krueger (1992), software reuse can be problematic and even disadvantageous. Learning to use a specific piece of reusable software often takes considerable effort. Moreover, finding suitable code fragments may also prove to be a challenge. For uncomplicated software systems and especially reusable components, it may not be worth the effort. Therefore, developer needs to carefully consider all sides of reusing when building a software system; according to Krueger (1992, chap. 1.3), for successful software reuse scenario, the amount of intellectual effort between the concept and implementation of the system must be as low as possible. In practice, this means that the value of the reused component must be as high as possible for the developed system, while the implementation cost (resources needed to take the reusable component into use) should be relatively low.

3.2 Factors for Successful Software Reuse

Frakes and Isoda (1994) state that for successful software reuse, the procedure must be planned beforehand and evaluated with a cost-benefit analysis. !FIXME **They also present several other factors, add them here.** FIXME! !FIXME **This needs a lot more flesh. Add some general principles etc. "How to make reusable software"** FIXME!

3.3 Analyzing Software Reuse

According to Krueger (1992), in order to analyze reusing software, the reuse process to be studied should be separated into four *dimensions*. The dimen-

sions are presented below.

Abstraction is the process of making a piece of software more generic, thus making it applicable to a wider range of software projects. Software reuse is almost always based on abstraction, but according to Krueger (1992), raising the abstraction level has proven to be difficult, thus making building reusable software a nontrivial process.

Selection facilitates finding, comparing and choosing suitable pieces of software. For example, libraries or frameworks aid selection by bundling and structuring the software components.

Specialization is the process of making the abstracted component more specific, usually by parameterizing the software or making it transformable.

Integration facilitates providing the software with reusable components, for example with a mechanism to import relevant modules or functions to the software.

!FIXME **Add advantages and disadvantages in general about reuse, not about different methods as those are described in the reuse methods chapter. Software reuse metrics research (Mohagheghi and Conradi, 2008; Frakes and Terry, 1996; Selby, 2005) probably has some good points. Also, Johnson (1997) may have something.** FIXME!

3.4 Software Reuse Methods

Software reuse is not a single, uniform procedure or technique. Several different reuse techniques exist to cater different needs. Consequently, different reuse methods excel at different areas. In order to describe the advantages and disadvantages of the methods, we describe the using the reuse dimensions presented in the previous section.

Krueger (1992) and Johnson (1997), among others, present and analyze software reuse methods. From these, we have selected the most relevant for web environment, presenting those below.

3.4.1 High-Level Languages

High-level languages denote programming languages which are designed to be on a high abstraction level and thus contain features which are not necessary for a programming language but benefit or speed up the development. Traditional examples of these kind of features are automatic memory allocation (Krueger, 1992) and language constructs such as exceptions (Mitchell, 2003). More modern high-level language features are value type checking systems and abstracted support for parallel operations using futures (Totoo et al., 2012). It should be noted that the high-levelness of a language is a *relative* property, i.e. it is not possible to determine the requirements for a high-level language per se, only high-levelness of languages compared to other languages. For example, Krueger (1992) considers all programming languages above the abstraction level of the assembly language high-level languages, while Carro et al. (2006) consider e.g. lack of automatic memory management or type system a sign of lower-level language.

High-level languages *abstract* frequently used procedures into seemingly uncomplicated operations, thus reducing the work and cognitive capacity needed for developing the application. (Krueger, 1992, chap. 3)

As the number of elementary high-level language constructs is usually relatively low it is possible for programmers to master the use of those constructs with sufficiently little effort, rendering *selection* unproblematic. (Krueger, 1992, chap. 3)

Specialization of high-level language features is usually achieved by parameterizing the constructs, either implicitly or explicitly. For example, when instantiating a class in Java, the only parameterization needed for memory management is the actual object instance. However, e.g. exception handling always requires at least the logic needed for handling the exception. (Krueger, 1992, chap. 3)

Integration of high-level language features is automatically done when

compiling the software code. However, due to the nature of high-level languages, it is usually not possible to mix-and-match different programming languages easily in the same program. (Krueger, 1992, chap. 3)

The advantages of high-level languages are mainly related to the decreased need for developing frequently needed procedures manually, such as allocating or deallocating memory, case-by-case. These operations in high-level languages can be mapped into more complex procedures in some lower-level languages, effectively making them reusable software components. In practice, using high-level languages can yield a productivity gain up to 500 %. (Krueger, 1992, chap. 3)

The main disadvantage of using high-level languages is the potential decrease in performance. As with any software reuse, high-level programming languages abstract the supported procedures by making them more generic. This often leads to additional complexity and unnecessary operations on the compiled program. However, the decrease can often be minimized by using additional compile-time optimizations. (Carro et al., 2006)

On the web, the technologies used on the client-side are inherently fixed to descendants of HTML, CSS and JavaScript (World Wide Web Consortium, 2014, 2011; ECMA, 2011). Therefore, web application languages are relatively high-level by definition. However, it is still possible to raise the abstraction level by using e.g. CoffeeScript (Ashkenas, 2009) instead of JavaScript or LESS (Sellier, 2009) instead of CSS.

3.4.2 Design and Code Scavenging

Design and Code scavenging refers to the technique of scavenging pieces of software *ad hoc* from existing software systems and using the pieces as parts for a new software system (Krueger, 1992, chap. 4). The aim of this technique is to reduce the amount of work needed to build the system. For example, when building an UI component for choosing a date, the developer may scavenge the code for a calendar from an older software system.

Scavenging can be done without modifications to the code in the target code base (code scavenging) or by modifying the details of the scavenged code (design scavenging) (Krueger, 1992, chap. 4). The *abstraction* gained by

scavenging is therefore mostly informal and in some cases even its existence is questionable (Sametinger, 1997, chap. 3). Usually, there is no "hidden part" of the abstraction but the developer must maintain the functionality of all the code himself (Sametinger, 1997, chap. 3).

Usually there is no formal mechanism or support for *selecting* pieces of software to be scavenged. Therefore, the developer must rely on his memory, experience and word-of-mouth in order to find suitable pieces of software. (Sametinger, 1997, chap. 3)

Specialization is done by manually editing the scavenged source code. While it is often the fastest method of acquiring results, this requires the developer to deeply understand the scavenged implementation. It can also lead to fragmentation and maintainability issues in the future. (Krueger, 1992, chap. 4)

Integration of the scavenged code is done by copying and pasting the code to the target source code file. This may lead to namespace collisions between original and scavenged code which may result in the need for refactoring the code. (Krueger, 1992, chap. 4)

The main advantages of design and code scavenging are the ability to quickly include existing functionality to new software systems (Krueger, 1992, chap. 4). As it is usually not needed to prepare the code to be scavenged before scavenging it, the extent of possible pieces of software is often significantly larger than when using any other reuse method.

However, finding suitable pieces of software for scavenging is hard. Moreover, scavenging pieces of software often does not decrease the *cognitive distance* between the target and implementation of the system. It may also create issues with maintainability of the software. (Krueger, 1992, chap. 4)

3.4.3 Source Code Components

Using source code components is a type of reuse that chooses and uses software components from a component repository (Sametinger, 1997, chap. 3). Software component can be any piece of code, but in practice, components usually consist of one or more functions, modules or classes (Sametinger, 1997, chap. 3). An example of a source code component is a trigonometry

module which contains functions for sine, cosine and tangent calculations. When a developer needs to calculate sines in her program, she searches a component repository for trigonometry components and utilizes the component found in her own program (Krueger, 1992, chap. 5).

Ideally, source code components *abstract* the implementation details of the component inside. This means that the required cognitive distance between the concept and the implementation of the software system is lower when using source code components instead of e.g. code scavenging.

In order to be *selectable*, source code components should be accompanied by abstract names (function names) and descriptions of the functionality provided (Krueger, 1992, chap. 5). The names should describe *what* the components does instead of *how* it does it (Krueger, 1992, chap. 5). These names can then be used for reasoning about the purpose of the component and finding the component in source code component repositories – in order to use the component, the developer must be able to find it and to know what it does (Krueger, 1992, chap. 5).

Source code components can be *specialized* by modifying the source code Krueger (1992, chap. 5). However, as this technique yields unwanted consequences explained in the previous section, many components support specialization by parameterization. For example, the programmer could provide the sine function the angle in question. Additionally, when integrating the trigonometry module to her software system, the programmer could specify if the functions should use degrees or radians. In some components, specialization can also be achieved via subclassing (Krueger, 1992, chap. 5).

All modern programming languages support *integration* of reusable source code components written in the same language. Usually, the procedure is a very simple addition of source code files, which requires little to no effort on the programmer side. However, all source code components can't be used in the same program due to conflicts e.g. in naming and value types (Krueger, 1992, chap. 5).

The main advantages of using source code components are the abstraction provided and organized nature of the component repositories. Ideally, the repositories provide a search functionality so that even developers with no previous experience on the component domain can find the components

needed. Moreover, the abstraction level and the hiding of implementation details decreases the cognitive distance between the concept and the implementation of the system and reduce the source code needed to be written.

The main disadvantages of the source code components lie in the fact that the functionality must be deliberately designed to support reuse. The abstraction of the components is a major challenge (Krueger, 1992, chap. 5) in designing source code components. Additionally, the component repositories need administration and maintenance.

3.4.4 Software Schemas

!FIXME **Add similarly structured content as in the previous subsubsection. Or maybe drop?** FIXME!

3.4.5 Application Generators

Application generators are usually domain-specific generators which take very high level instructions (specifications) as input and then output significantly lower level software code (implementation) (Cleaveland, 1988; Krueger, 1992, chap. 7). On fundamental level, application generators differ from high-level language compilers mainly by being designed to work on a narrow domain and thus being able to support considerably higher-level instructions (Krueger, 1992, chap. 7). Unlike source code components, the reused components generated by application generators are usually not encapsulated or separated (Sametinger, 1997, chap. 3).

Application generators *abstract* the concept or specification of the software system, hiding the actual implementation completely from the user of the generator (Cleaveland, 1988). However, in some cases it may be necessary to modify the output of the generator which essentially removes the abstraction.

In principle, *selecting* application generations is moderately easy since the abstraction level of application generators is usually very high, rendering reasoning about the purpose of the generator fairly easy (Krueger, 1992, chap. 7). However, since application generators are usually suited for a very

narrow domain, it is usually difficult to find a suitable generator (Krueger, 1992, chap. 7).

Typically, software systems generated with application generators consist of variant and invariant parts (Krueger, 1992, chap. 7). Invariant part is the part of the program which the developer using the generator can't modify. The developer *specializes* the program by modifying the variant part. There are several methods of modifying the variant part. One of the simplest may be straightforward parameterization: the developer chooses the parameters of the system from a predefined set of alternatives. This method makes using the generation extraordinarily easy. However, it also limits the resulting application considerably.

On the other end of the spectrum, the application generator may require the variant parts to be inputted using a domain-specific or generic-purpose programming language. This makes the application generator incredibly versatile, but requires both more domain-specific and programming knowledge.

Typically, application generators generate complete applications which do not require further *integration* (Krueger, 1992, chap. 7). However, occasionally, the resulting applications are not independent per se, but require integration to other systems. This may be an issue since often it is not possible to select the integration interfaces freely, but to use the ones provided by the generator.

One of the main advantages of the application generators is the abstraction they provide. In some cases, the application generators may even require no programming language knowledge as long as the user has relevant domain-specific knowledge (Horowitz et al., 1985). Moreover, application generators excel when there is a need for building multiple similar applications (Krueger, 1992, chap. 7).

However, application generators require an unambiguous mapping between the specifications and implementation details (Krueger, 1992, chap. 7). Moreover, building application generators requires a reliable, generic implementation and user interfaces for developers (Cleaveland, 1988). Therefore, building application generators requires comprehensive domain-specific knowledge in addition to extensive software development expertise.

3.4.6 Very High Level Languages

!Fixme Or domain-specific languages? Add similarly structured content as in the previous subsubsection Fixme!

3.4.7 Transformational Systems

!Fixme Add similarly structured content as in the previous subsubsection. Or maybe drop? Fixme!

3.4.8 Software Architectures

!Fixme Add similarly structured content as in the previous subsubsection Fixme!

3.4.9 Software Frameworks

!Fixme Add similarly structured content as in the previous subsubsection. Are frameworks the same as architectures? If, drop. Fixme!

Chapter 4

Research Gap

Currently, research on geographic or map visualization is abundant !Fixme **Use a bunch of geoviz references to prove this? Or some other way?** Fixme!. Moreover, according to the visualization definition by Kosara (2007), it may be possible to abstract parts of the visualization implementation in order to achieve visualization process which requires less effort and technical knowledge. However, both research about geovisualization-related software reuse and actual reusable geovisualization components are scant. This implies that there is room for improvement in both research and implementation related to geovisualization software. !Fixme **IMHO this is way too short for a chapter. How to fix?** Fixme!

- Current research does not completely cover the field of this thesis
- There is research about visualizing geographic content (generally or on the web)
- According to literature, it is possible to abstract parts of the implementation of web geovisualization.
- However, there is little (or no) research on how to make geovisualization more efficiently

Chapter 5

Environment

5.1 Web Applications

Describe HTML5, JS and other related technology, because these are really the things that restrict the implementation.

Visualizations need to be implemented using web technology, because it is crucial for distributability and discoverability that the system works in web browsers... !Fixme **continue...** Fixme!

5.2 Geographic Visualization on the Web

Probably should describe the relevant web technology, some needed HTML5/JS features etc. !Fixme **to subsection of related webtech?** Fixme!

5.3 Current State of Building Map Visualizations on the Web

Describe the current libraries, frameworks and procedures used to build map visualizations on the web. Should these be evaluated separately? Use FindBooze, Peruskartta and Ottoapp as examples. Also, reason why it is unnecessarily laborious to write map visualizations every time from scratch.

When you use `pdflatex` to render your thesis, you can include PDF images directly, as shown by Figure 5.1 below.

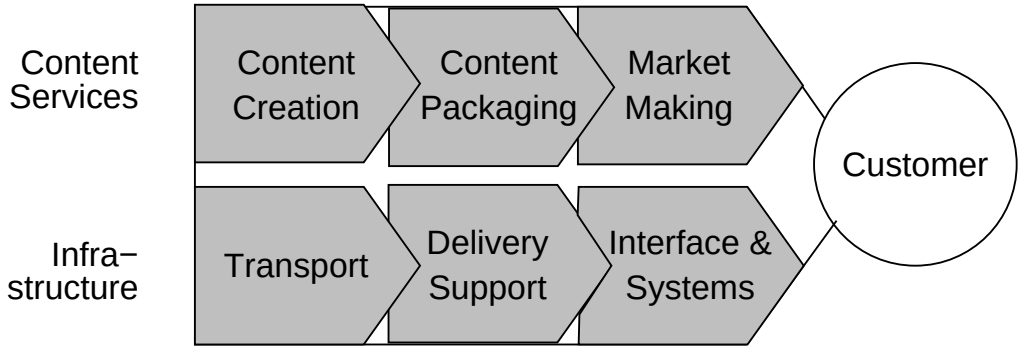
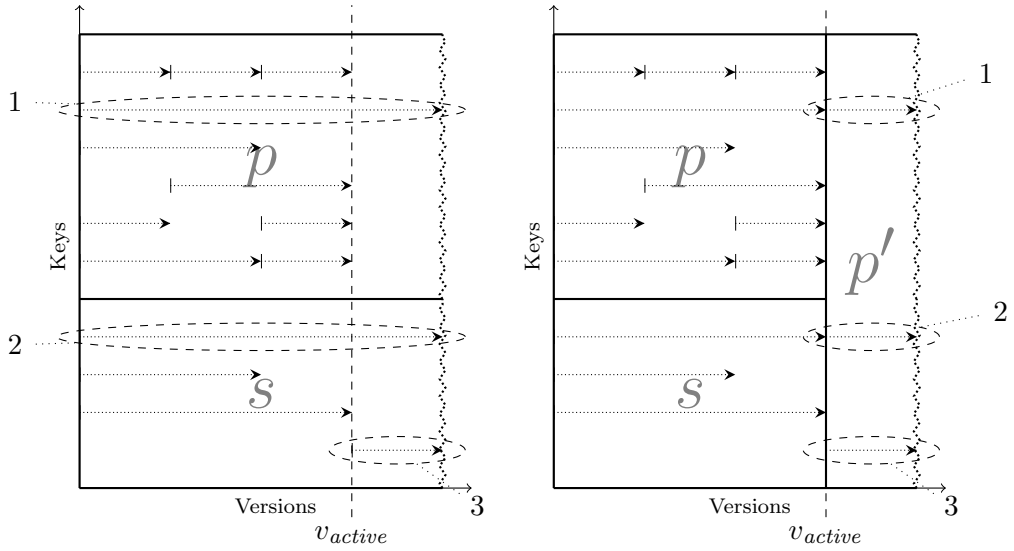


Figure 5.1: The INDICA two-layered value chain model.

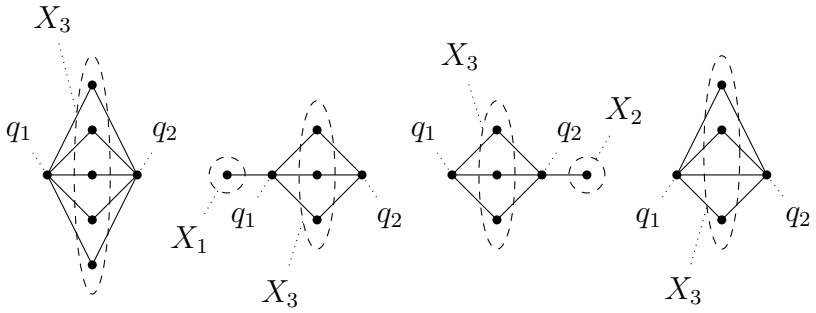
You can also include JPEG or PNG files, as shown by Figure 5.2.

Figure 5.2: Eeyore, or Ihaa, a very sad donkey.

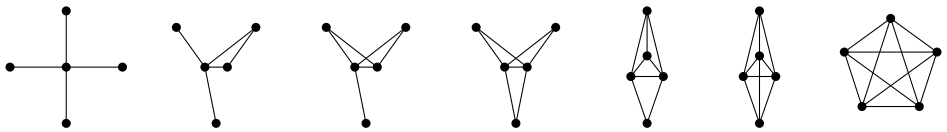


(a) Before (b) After

Figure 5.3: Example of a multiversion database page merge. This figure has been taken from the PhD thesis of Haapasalo (Haapasalo, 2010).



(a) Examples of obstruction graphs for the Ferry Problem



(b) Examples of star graphs

Figure 5.4: Examples of graphs draw with TikZ. These figures have been taken from a course report for the graph theory course (Göös et al., 2010).

Chapter 6

Methodology

6.1 Evaluating Software Reuse Effectiveness

Several methods for analyzing software reuse exist. Frakes and Terry (1996) present six general types of software reuse analysis models: cost-benefit analysis models, maturity assessment models, amount-of-reuse models, failure modes analysis, reusability assessment models and reuse library metrics.

Cost-benefit analysis models consider both development costs for the reusable component and the reuse productivity and quality benefits. Maturity assessment models categorize the matureness of a systematic software reuse program. Amount of reuse metrics assess the proportion of reused software in the software system created. Software reuse failure modes model introduces a set of reuse failures which are then used to assess a systematic reuse program. Reusability assessment models aim to analyze various software attributes to assess the reusability of a piece of code. Reuse library metrics concentrate on the reusability of software component libraries instead of single components.

Not all the models discussed above are applicable to this type of research. For instance, as maturity assessment models assess a reuse program as a whole instead of single reusable piece of software, it can not be used for this type of work. Moreover, it should be noted that these models are high-level analysis tools which do not take a stance on how to measure the detailed data required by the measurements.

In their review of multiple case studies, Mohagheghi and Conradi (2007) list several methods as alternatives for analyzing software reuse. Of them, notable ones include *controlled experiments*, *case studies*, *surveys* and *experience reports*. However, as the scope of this work does not allow for large sample sizes required by controlled experiment method, it is not a feasible alternative for this study. Moreover, using experience reports requires a considerable experience on creating and using reusable software.

According to Kitchenham and Pickard (1998), one method for conducting a case study is using a *sister project* comparison. In sister project comparison, a minimum of two different, but sufficiently similar software projects observed. In at least one of the projects, a new method is employed, while in at least one project, the old method is in use. All other practices and aspects should be left unchanged. Mohagheghi and Conradi (2007) argue that this kind of comparison is applicable for analyzing software reuse effectiveness for a specific piece of software. The sister project case study is appropriate when no systematic reuse program exists or when there are other barriers impeding the use of models presented by Frakes and Terry (1996). Moreover, it is possible to create the sister project synthetically by building the same kind of application twice, once with and once without the reusable component.

As presented above, in the higher abstraction level, analyzing projects is fairly straightforward. However, the actual low-level measurements for reusing software are much more complex. Mohagheghi and Conradi (2007) argue that measuring software reuse effectiveness precisely is difficult due to a number of factors: 1) Metrics are difficult to validate since there is no universally accepted definition of "quality" in software products, and 2) the productivity of development is difficult to measure and therefore highly subjective, vague or even erroneous metrics are utilized.

Both Frakes and Terry (1996) and Mohagheghi and Conradi (2007) agree that the size of the code needed to be written correlates inversely to the development productivity. Moreover, research by Banker et al. (1993); Gill and Kemerer (1991) show that software code complexity correlates inversely to the software maintenance productivity. Due to the nature of software maintenance and the fact that it is often hard to distinguish small-scale software development and maintenance (Chapin et al., 2001), it is likely that this prin-

ciple applies to developing new software to some degree as well. Therefore, it seems evident that in order to enable productive use of reusable software, the new code to be written (outside the reusable components) should be made simple and concise.

The conciseness of the code can be measured by several different metrics. According to Fenton and Pfleeger (1998), the number of lines in program source code is only one perspective. It can be complemented by measuring the functionality and complexity of the program.

According to Fenton and Pfleeger (1998), the most commonly used metric for program size is its length, i.e., the number of lines of source code. The metric can be refined by only considering effective lines, ignoring lines consisting of comments and whitespace. However, Fenton and Pfleeger (1998) encourages the use of both effective and physical line counts to determine the size of a program.

The functionality of the program can be determined with several different metrics. Fenton and Pfleeger (1998) presents the function point approach, which uses the number and complexity of external inputs and outputs, object point approach which uses the number of different screens and reports involved in the application, and so-called "bang metrics" which use the total number of primitives in the data-flow diagram of the program. It should be noted that all of these methods are highly subjective and only provide speculative metrics.

Also the complexity of the program can be determined with several different techniques. According to Fenton and Pfleeger (1998), the complexity of the program (solution) should ideally be not higher than the problem complexity. According to them, in the ideal case it is possible to determine the complexity of the program by determining the complexity of the program. However, this is not usually the case in real-life applications. McCabe (1976) presents a computational approach for determining the complexity of an application implementation. His approach is used by counting the number of cycles in the program flow graph. The advantage of this approach when compared to the method presented by Fenton and Pfleeger (1998) is that it can be used to compute complexity differences of multiple implementations of the problem.

!FIXME **Halstead measurements?** FIXME!

6.2 Research Method Chosen for the Analysis

For this work, we considered both the methods presented by Frakes and Terry (1996) and Mohagheghi and Conradi (2007). As stated in the previous section, several of the methods are only applicable for assessing large-scale reuse programs or cases with large sample size and therefore were not considered for this work. Of the remaining methods, a cost-benefit analysis was selected. For studying costs and benefits of the reuse, a case study with sister project comparison approach was deemed most applicable.

For comparing the projects, software size and complexity metrics by Fenton and Pfleeger (1998) were used, with the exception that measuring software complexity is done with the method by McCabe (1976) as it allows comparing different implementations of similar applications. !FIXME **Add specifics about tools, restrictions etc. See chapter 7.1 of achy** FIXME!

For gathering data, we implemented three !FIXME **confirm the number** FIXME! map visualizations with and without the framework. The first visualization is a simple use case which concentrates on the display of a custom map. The second visualization highlights POI data on a map, and the third one is the most complex, consisting of POI data, relations between POIs and a custom backend serving the data with real-time updates from a mobile client. The types of visualizations were selected to obtain data about a wide variety of different map visualizations. !FIXME **Probably the visualizations ought to a bit more complex than this** FIXME!

!FIXME **How are the measurements combined? (1st viz without framework + 2nd viz without framework + 3rd wiz without framework) ¡=¿ (1st viz with framework + 2nd viz with framework + 3rd wiz with framework + framework)? Or should the framework be excluded?** FIXME!

Code	**Name**	**Methods**	**Area**
T-110.6130	Systems Engineering for Data Communications Software	Computer simulations, mathematical modeling, experimental research, data analysis, and network service business research methods, (agile method)	T-110
Mat-2.3170 Simulation (here is an example of multicolumn for tables)		Details of how to build simulations	T-110
S-38.3184	Network Traffic Measurements and Analysis	How to measure and analyse network traffic	T-110

Table 6.1: Research methodology courses

Chapter 7

Implementation

You have now explained how you are going to tackle your problem. Go do that now! Come back when the problem is solved!

Now, how did you solve the problem? Explain how you implemented your solution, be it a software component, a custom-made FPGA, a fried jelly bean, or whatever. Describe the problems you encountered with your implementation work.

- First, analyze several visualizations made separately
  - Plain map =¿ Peruskartta
  - Dot map =¿ FindBooze
  - Isarithmic map =¿ Travel time visualization (to do)
  - Heatmap =¿ Helsinki Hot?
  - Choropleth map =¿ Use some existing?
  - Proportional symbol map =¿ to do
  - Flow map =¿ any way to find anything relevant?
- Consider multivariate possibilities
- Try to find common elements and patterns =¿ use these as a basis for framework implementation
- Need to build visualizations efficiently =¿ some kind of whole page scaffold needed

- Need to be able to embed visualizations =¿ container-specific solution needed

Chapter 8

Evaluation

You have done your work, but that's[1] not enough.

You also need to evaluate how well your implementation works. The nature of the evaluation depends on your problem, your method, and your implementation that are all described in the thesis before this chapter. If you have created a program for exact-text matching, then you measure how long it takes for your implementation to search for different patterns, and compare it against the implementation that was used before. If you have designed a process for managing software projects, you perhaps interview people working with a waterfall-style management process, have them adapt your management process, and interview them again after they have worked with your process for some time. See what's changed.

The important thing is that you can evaluate your success somehow. Remember that you do not have to succeed in making something spectacular; a total implementation failure may still give grounds for a very good master's thesis—if you can analyze what went wrong and what should have been done.

[1] By the way, do *not* use shorthands like this in your text! It is not professional! Always write out all the words: "that is".

Chapter 9

Discussion

At this point, you will have some insightful thoughts on your implementation and you may have ideas on what could be done in the future. This chapter is a good place to discuss your thesis as a whole and to show your professor that you have really understood some non-trivial aspects of the methods you used...

Chapter 10

Conclusions

Time to wrap it up! Write down the most important findings from your work. Like the introduction, this chapter is not very long. Two to four pages might be a good limit.

!Fixme **Remove "libproxy.aalto.fi" from all the URLs in references!** Fixme!

Bibliography

Gennady L. Andrienko and Natalia V. Andrienko. Interactive maps for visual data exploration. *International Journal of Geographical Information Science*, 13(4):355–374, 1999. ISSN 1365-8816. doi: 10.1080/136588199241247. URL http://dx.doi.org/10.1080/136588199241247.

Jeremy Ashkenas. CoffeeScript, December 2009. URL http://coffeescript.org. Accessed 7.9.2014.

Tarek Azzam and Stephanie Evergreen. *J-B PE Single Issue (Program) Evaluation, Volume 139 : Data Visualization, Part 1 : New Directions for Evaluation*. John Wiley & Sons, Somerset, NJ, USA, 2013. ISBN 9781118793374. URL http://site.ebrary.com.libproxy.aalto.fi/lib/aalto/docDetail.action?docID=10768989.

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, November 1993. ISSN 0001-0782. doi: 10.1145/163359.163375. URL http://doi.acm.org/10.1145/163359.163375.

Barbara Bartz Petchenik. From place to space: The psychological achievement of thematic mapping. *The American Cartographer*, 6(1):5–12, 1979. ISSN 0094-1689. doi: 10.1559/152304079784022763. URL http://www.tandfonline.com/doi/abs/10.1559/152304079784022763.

Tim Berners-Lee. Information management: A proposal, March 1989. URL http://www.w3.org/History/1989/proposal.html.

Tim Berners-Lee, Robert Cailliau, Jean-FranÃ§ois Groff, and Bernd Pollermann. World-wide web: The information universe. *Internet Research*,

2(1):52–58, December 1992. ISSN 1066-2243. doi: 10.1108/eb047254. URL http://www.emeraldinsight.com.libproxy.aalto.fi/journals.htm?articleid=1670698&show=abstract.

B. Boehm. Managing software productivity and reuse. *Computer*, 32(9): 111–113, September 1999. ISSN 0018-9162. doi: 10.1109/2.789755.

Manuel Carro, JosÃ© F. Morales, Henk L. Muller, G. Puebla, and M. Hermenegildo. High-level languages for small devices: A case study. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 271–281, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. doi: 10.1145/1176760.1176794. URL http://doi.acm.org/10.1145/1176760.1176794.

Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, January 2001. ISSN 1040-550X. URL http://dl.acm.org/citation.cfm?id=371697.371701.

J.C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988. ISSN 0740-7459. doi: 10.1109/52.17799.

ECMA. ECMAScriptÂ® language specification, June 2011. URL http://www.ecma-international.org/ecma-262/5.1/. Accessed 7.9.2014.

Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998. ISBN 0534954251.

W.B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Software*, 11(5):14–19, September 1994. ISSN 0740-7459. doi: 10.1109/52.311045.

William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996. ISSN 0360-0300. doi: 10.1145/234528.234531. URL http://doi.acm.org/10.1145/234528.234531.

G.K. Gill and C.F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17 (12):1284–1288, December 1991. ISSN 0098-5589. doi: 10.1109/32.106988.

Google. Maps API, June 2005a. URL `http://maps.google.com`. Accessed 23.7.2014.

Google. Maps, February 2005b. URL `http://maps.google.com`. Accessed 25.7.2014.

Mika Göös, Tuukka Haapasalo, and Leo Moisio. Finding hard instances of the ferry problem. Course report for the course T-79.5203/S-72.2420 Graph theory, Aalto SCI, 2010.

Tuukka Haapasalo. *Accessing Multiversion Data in Database Transactions*. PhD thesis, Department of Computer Science and Engineering, Aalto University School of Science and Technology, Espoo, Finland, 2010. `http://lib.tkk.fi/Diss/2010/isbn9789526033600/`.

E. Horowitz, A Kemper, and B. Narasimhan. A survey of application generators. *IEEE Software*, 2(1):40–54, January 1985. ISSN 0740-7459. doi: 10.1109/MS.1985.230048.

Ohad Inbar, Noam Tractinsky, and Joachim Meyer. Minimalism in information visualization: Attitudes towards maximizing the data-ink ratio. In *Proceedings of the 14th European Conference on Cognitive Ergonomics: Invent! Explore!*, ECCE '07, pages 185–188, New York, NY, USA, 2007. ACM. ISBN 978-1-84799-849-1. doi: 10.1145/1362550.1362587. URL `http://doi.acm.org/10.1145/1362550.1362587`.

Ralph E. Johnson. Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42, 1997. URL `http://dl.acm.org/citation.cfm?id=262799`.

Barbara Ann Kitchenham and Lesley M. Pickard. Evaluating software eng. methods and tools part 10: Designing and running a quantitative case study. *SIGSOFT Softw. Eng. Notes*, 23(3):20–22, May 1998. ISSN 0163-5948. doi: 10.1145/279437.279445. URL `http://doi.acm.org/10.1145/279437.279445`.

R. Kosara. Visualization criticism - the missing link between information visualization and art. In *Information Visualization, 2007. IV '07. 11th*

International Conference, pages 631–636, July 2007. doi: 10.1109/IV.2007.130.

Menno-Jan Kraak. The cartographic visualization process: From presentation to exploration. *The Cartographic Journal*, 35(1):11–15, June 1998. ISSN 0008-7041. doi: 10.1179/caj.1998.35.1.11. URL http://www.maneyonline.com.libproxy.aalto.fi/doi/abs/10.1179/caj.1998.35.1.11.

Menno-Jan Kraak and Alan MacEachren. Visualization for exploration of spatial data. *International Journal of Geographical Information Science*, 13(4):285–287, 1999. ISSN 1365-8816. doi: 10.1080/136588199241201. URL http://dx.doi.org/10.1080/136588199241201.

Menno-Jan Kraak and Ferjan Ormeling. *Cartography, Third Edition: Visualization of Spatial Data*. Guilford Press, June 2011. ISBN 9781609181949.

Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992. ISSN 0360-0300. doi: 10.1145/130844.130856. URL http://doi.acm.org/10.1145/130844.130856.

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

Doug Mcilroy. Mass-produced software components. pages 138–155, January 1969. URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003. ISBN 9780521780988.

Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, October 2007. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-007-9040-x. URL http://link.springer.com.libproxy.aalto.fi/article/10.1007/s10664-007-9040-x.

Parastoo Mohagheghi and Reidar Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17(3):13:1–13:31, June 2008. ISSN 1049-331X. doi: 10.1145/1363102.1363104. URL http://doi.acm.org/10.1145/1363102.1363104.

Johannes Sametinger. *Software engineering with reusable components*. Springer, 1997. URL http://www.google.com/books?hl=en&lr=&id=AxPQvGRs2wUC&oi=fnd&pg=PA1&dq=code+scavenging+reuse&ots=K85JD3_HCN&sig=66nJKlXrtJc3bHAtYMe4SCc2tTc.

Hansgeorg Schlichtmann. Visualization in thematic cartography: towards a framework. *The Selected Problems of Theoretical Cartography*, pages 49–61, 2002. URL http://rcswww.urz.tu-dresden.de/~wolodt/tc-com/pdf/sch_2000.pdf. Accessed 2014-07-21.

R.W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, June 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.69.

Alexis Sellier. LESS, 2009. URL http://lesscss.org/. Accessed 7.9.2014.

Terry A. Slocum and Robert B. McMaster. *Thematic Cartography and Geovisualization: Pearson New International Edition*. Pearson Education, 3rd edition, 2014. ISBN 9781292055442. URL https://www.dawsonera.com/abstract/9781292055442.

Prabhat Totoo, Pantazis Deligiannis, and Hans-Wolfgang Loidl. Haskell vs. f# vs. scala: A high-level language features and parallelism support comparison. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 49–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7. doi: 10.1145/2364474.2364483. URL http://doi.acm.org/10.1145/2364474.2364483.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1986. ISBN 0-9613921-0-X.

World Wide Web Consortium. Cascading style sheets level 2 revision 1 (CSS 2.1) specification, June 2011. URL `http://www.w3.org/TR/CSS2/`. Accessed 7.9.2014.

World Wide Web Consortium. HTML5, September 2014. URL `http://www.w3.org/html/wg/drafts/html/CR/`. Accessed 7.9.2014.

Appendix A

First appendix

This is the first appendix. You could put some test images or verbose data in an appendix, if there is too much data to fit in the actual text nicely.

For now, the Aalto logo variants are shown in Figure A.1.

(a) In English

(b) Suomeksi

(c) På svenska

Figure A.1: Aalto logo variants