Software Cartography

A Prototype for Thematic Software Maps

diploma thesis
for the philosophic-natural science faculty
university of Bern

presented by

Peter Loretan

April 2011

Leader of the work

Prof. Dr. Oscar Nierstrasz

Adrian Kuhn

Institute of Computer Science and Applied Mathematics

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Peter Loretan
pelo@imu.unibe.ch
http://scg.unibe.ch/research/softwarecartography

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Acknowledgments

First of all, I would like to thank Adrian Kuhn for his endurance. Meeting him was always as inspirational as helpful. The software cartography project would never have gone such a long way without his basic work on Latent Semantic Indexing and continual support.

I would also like to thank Prof. Oscar Nierstraz for giving me the opportunity to work in his group and for his inspiring lectures on software analysis.

I thank also the members of the Software Composition Group for their valuable comments and suggestions, especially Tudor Girba for helping to include an early version of SOFTWARECARTOGRAPHER in Muse. Also David Erni made a crucial contribution during the course of his own master thesis by upgrading a SOFTWARECARTOGRAPHER prototype to a eclipse plug-in and testing it on real programmer. The excellent results of his field studies helped me to focus on conceptual issues.

And last but not least, the people around me in my private life for not giving up to remind me of the remaining work during the last few months of my studies - my family, my house mates and my friends.

Peter Loretan
December 2010

Abstract

Software visualizations can provide a concise overview of a complex software system. Unfortunately, since software has no physical shape, there is no "natural" mapping of software to a two-dimensional space. As a consequence most visualizations tend to use a layout in which position and distance have no meaning, and consequently layout typical diverges from one visualization to another. We propose a consistent layout for software maps in which the position of a software artifact reflects its *vocabulary*, and distance corresponds to similarity of vocabulary. We use Latent Semantic Indexing (LSI) to map software artifacts to a vector space, and then use Multidimensional Scaling (MDS) to map this vector space down to two dimensions. The resulting consistent layout allows us to develop a variety of thematic software maps that express very different aspects of software while making it easy to compare them. The approach is especially suitable for comparing views of evolving software, since the vocabulary of software artifacts tends to be stable over time.

Contents

Chapter 1

Introduction

1.1 Software Cartography

Software visualization offers an attractive means to abstract from the complexity of large software systems [9, 16, 25, 29].

A single graphic can convey a great deal of information about various aspects of a complex software system, such as its structure, the degree of coupling and cohesion, growth patterns, defect rates, and so on.

Unfortunately, the great wealth of different visualizations that have been developed to abstract away from the complexity of software has led to yet another source of complexity: it is hard to compare different visualizations of the same software system and correlate the information they present.

We can contrast this situation with that of conventional thematic maps found in an atlas. Different phenomena, ranging from population density to industry sectors, birth rate, or even flow of trade, are all displayed and expressed using *the same consistent layout*. It is easy to correlate different kinds of information concerning the same geographical entities because they are generally presented using the same kind of layout. This is possible because (i) there is a "natural" mapping of position and distance information to a two-dimensional layout (the earth being, luckily, more-or-less flat, at least on a local scale), and (ii) by convention, North is normally considered to be "up".[1]

Software artifacts, on the other hand, have no natural layout since they have no physical location. Distance and orientation also have no obvious meaning for software.It is presumably for this reason that there are so many different and incomparable ways of visualizing software. A cursory survey of recent SOFTVIS and VISSOFT publications shows that the majority of the presented visualizations feature arbitrary layout, the most common being based on alphabetical ordering and *hash-key ordering*.(Hash-key ordering is what we get in most programming languages when iterating over the elements of a Set or Dictionary collection.)

[1] On traditional Muslim world maps, for example, South used to be on the top. Hence, if Europe would have fallen to the Ottomans at the Battle of Vienna in 1683, all our maps might be drawn upside down [13].

Consistent layout for software would make it easier to compare visualizations of different kinds of information, but what should be the basis for laying out and positioning representations of software artifacts within a "software map"? What we need is a semantically meaningful notion of position and distance for software artifacts which can then be mapped to consistent layout for 2-D software maps.

We propose to use *vocabulary* as the most natural analogue of physical position for software artifacts, and to map these positions to a two-dimensional space as a way to achieve consistent layout for software maps. Distance between software artifacts then corresponds to distance in their vocabulary. Drawing from previous work [17, 10] we apply Latent Semantic Indexing to the vocabulary of a system to obtain n-dimensional locations, and we use Multidimensional Scaling to obtain a consistent layout. Finally we employ digital elevation, hill-shading and contour lines to generate a landscape representing the frequency of topics.

Why should vocabulary be more natural than other properties of source code? First of all, vocabulary can effectively *abstract* away from the technical details of source code by identifying the key domain concepts reflected by the code [17]. Software artifacts that have similar vocabulary are therefore close in terms of the domain concepts that they deal with. Furthermore, it is known that: over time software tends to grow rather than to change [33], and the vocabulary tends to be more stable than the structure of software [2]. Although re-factorings may cause functionality to be renamed or moved, the overall vocabulary tends not to change, except as a side-effect of growth. This suggests that vocabulary will be relatively *stable* in the face of change, except where significant growth occurs. As a consequence, vocabulary not only offers an intuitive notion of position that can be used to provide a consistent layout for different kinds of thematic maps, but it also provides a robust and consistent layout for mapping an evolving system. System growth can be clearly positioned with respect to old and more stable parts of the same system.

We call our approach *Software Cartography*, and call a series of visualizations *Software Maps*, when they all use the same consistent layout created by our approach.

The contributions of this thesis are as follows:

- Identification and Motivation of the need for consistent layouts in software visualization
- A set of techniques to create a consistent layout of a software system: lexical information, LSI, and MDS
- Presentation of SOFTWARECARTOGRAPHER, a proof-of-concept implementation of software cartography and discuss the algorithms used
- Examples of thematic software maps that exploit consistent layout to display different information for the same system
- Considerations on how consistent layout can be used to illustrate the evolution of a system over time

1.2 Structure of this thesis

The next chapters of this thesis are structured as follows: Chapter 2 discusses related work by giving a short overview on similar projects. Chapter 3 presents the technique used in SOFTWARECARTOGRAPHER for mapping software to consistent layouts. Chapter 4 contains several case studies that illustrate consistent layouts for various thematic software maps. Chapter 5 open questions related to certain technical tasks and gives a short overview of David Erni's work on IDE integrated Code Maps. Finally, Chapter 6 concludes with some remarks about future work.

Chapter 2

State of the Art

2.1 Topic Maps

Using Muli-Dimensional Scaling (MDS) to create a map of information is by no means a novel idea. Topic maps, as they are called, have a long standing tradition in information visualization. [34]. In the following two subsections we will deliver a short insight into two mature projects, which use topic maps in order to gain information based on large sets of documents.

2.1.1 ThemeScape and SPIRE

ThemeScape is the best-known example of a text visualization tool that organizes topics found in documents into topic maps where physical distance correlates to topical distance and surface height corresponds to topical frequency [36].ThemeScape is part of a larger tool set that uses a variety of algorithms to cluster terms in documents.

In short, they represented each document in a set as a binary vector with the length equal to the number of used words from the dictionary. Their test set included from a few hundred up to 6 K documents and resulted in vectors with between 200 and 200 K units. The development team first used Multidimensional Scaling to project their high-dimensional document vectors to a 2D Layer. The first visualization of such an projection was computed in 1994 on a Sparc 3 workstation and took over 12h to scale dimension and position for a few hundred documents - a relatively critical cost in time for a project that aimed to analyze over 30'000 documents per day. Since the MDS algorithm calculates the distance between each pair of vectors, the computation time increases exponentially with the document sets size. To work around this computation bottleneck they invented their own scaling algorithm, called "Anchored Least Stress", suitable for sets larger then 5 K documents. This approach needs an initial cluster set. These where extracted from the documents using well-known techniques such as K-Means and complete linkage hierarchical clustering. In contrast to the MDS Algorithm the ALS Algorithm regards only the distance between this clusters and the document, starting with only a few elements of each vector and iteratively adding following elements. The landscape is then constructed by successively applying gaussian bodies to

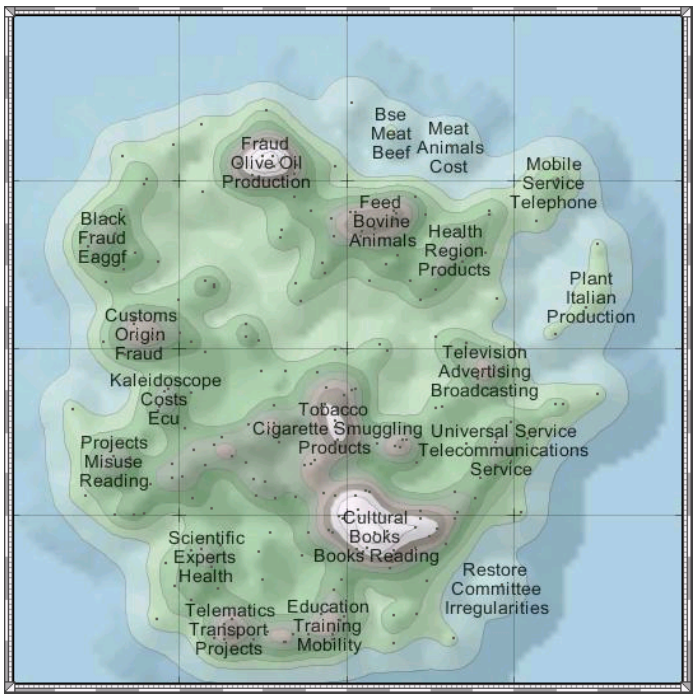


Figure 2.1: *ThemeScape: A tool shows concentrations of documents in a spatial format, based on the mathematics of the relationships. This is done using a number of real-time data feeds*

the vector parts and layering the contributions of the contributing topical terms, similar to our approach [36].

2.1.2 Atlas der Politischen Landschaften

Another project that inspired the work on code maps without direct application to software is the book "Atlas der Politischen Landschaften: Ein weltanschauliches Portraet der Schweiz"[12]. The authors of this book recomputed the location of each Swiss city and village only depending on their election results of the past fifty years. The technique to convert this rater huge amount of data is very similar to the one used in ThemeMaps and later adopted by SOFTWARECARTOGRAPHER. The election results are used to represent each village as a multidimensional vector. By multidimensional scaling this vector becomes a significant political location. The evaluation of each village represents its population, gaps between villages indicate political disagreement ore political setting with no representative in the electing population. By doing so, the point of compass of their maps gains a political context. E.g. Villages with similar election results are clustered together and their coordinates on the overall map of the Swiss political landscape clearly reflect their political perspective and weight. The map also not only clusters political perspectives, it gives the authors also a tool to determinate the meaning of orientation of the whole map. This enables inferences on single parts of the map, like: This city is located west in the landscape, it seems that people there elect more liberal. Or because there is a huge mountain in the east, the eastern

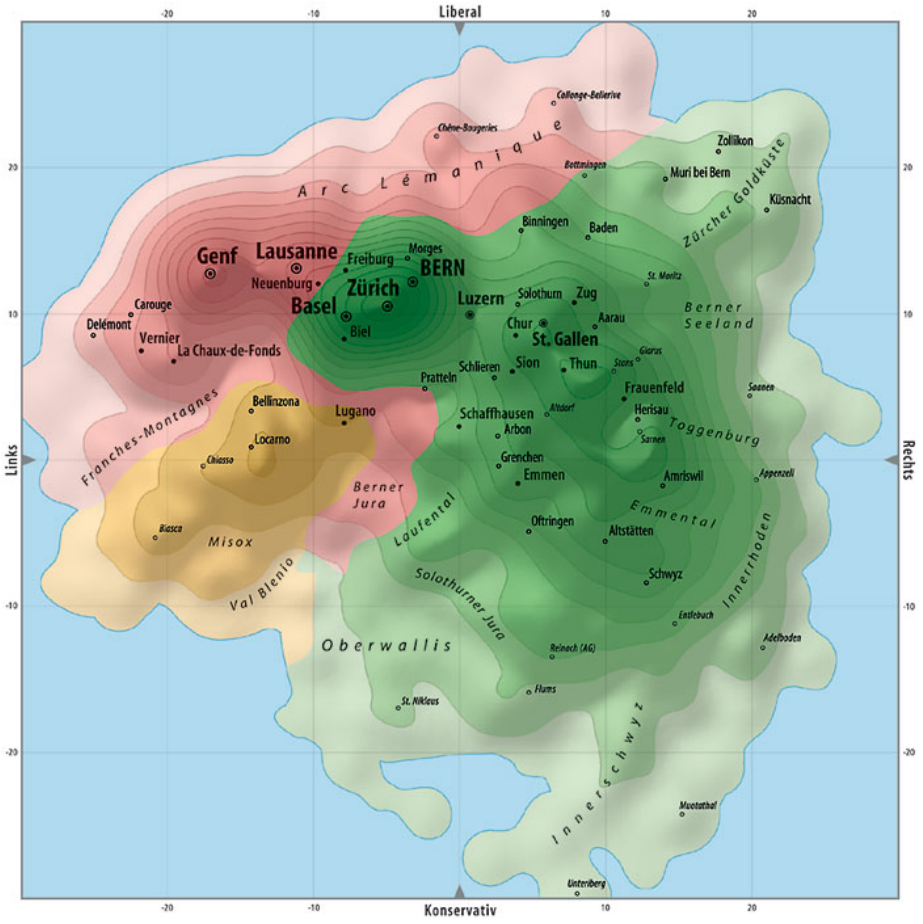


Figure 2.2: *Political landscape: A layout of swiss vilages depending on their election results over the last 50 years.*

area seems to be more flat, so the Swiss people tends to elect more conservative. These political maps where greatly honored by political science for their objective but also intuitive view on Swiss politics, adapting this idea to software would perhaps generate similar new refreshing views on code.

2.2 Software Maps

Topic maps in general and ThemeScape-style maps are rarely used in the software visualization community. We are unaware of their application in software visualization to produce consistent layouts for thematic maps, or to visualize the evolution of a software system. Therefore the next step is now to have a look on today's software visualization tools and compare their approach with ours.

Since Software consists always of multiple elements it is the first target for a software visualization to have a meaningful layout of these components. To identify these

elements in an object oriented language might be a easy task. To arrange them with meaning has no such clear projection from code to an image. There are several attempts to generate significant layouts. Most software visualization layouts are based on one or multiple of the three following approaches: 1) UML diagrams, 2) force-based graph drawing, and 3) treemap layouts.

2.2.1 UML diagrams.

UML diagrams generally employ arbitrary layout. Gudenberg *et al.* have proposed an evolutionary approach to layout UML diagrams in which a fitness function is used to optimize various metrics (such as number of edge crossings) [32]. Although the resulting layout does not reflect a distance metric, in principle the technique could be adapted to do so. Achieving a consistent layout is not a goal in this work.

Andriyevksa *et al.* have conducted user studies to assess the effect that different UML layout schemes have on software comprehension [1]. They report that the layout scheme that groups architecturally related classes together yields best results. They conclude that it is more important that a layout scheme convey a meaningful grouping of entities, rather than being esthetically appealing.

Byelas and Telea highlight related elements in a UML diagram using a custom "area of interest" algorithm that connects all related elements with a blob of the same color, taking special care to minimize the number of crossings [7]. The impact of an arbitrary layout on their approach is not discussed.

2.2.2 Graph drawing.

Graph drawing refers to a number of techniques to layout two- and three-dimensional graphs for the purpose of information visualization [34, 15]. Noack *et al.* offer a good starting point for applying graph drawing to software visualization [24].

Unlike MDS, graph drawing does not attempt to map an n-dimensional space to two dimensions, but rather optimizes a fitness function related to the spatial property of the output, *i.e.* of the visualization. Force-based layout for example, tries to minimize the number of edge crossings and to place all nodes as equally apart from each other as possible.

Jucknath-John *et al.* present a technique to achieve stable graph layouts over the evolution of the displayed software system [14], thus achieving consistent layout, while sidestepping the issue of reflecting meaningful position or distance metrics.

2.2.3 Treemap layout

Treemaps represent tree-structured information using nested rectangles [34]. Though treemaps can achieve a consistent layout, position and distance are not meaningful. First of all, they are often applied with arbitrary order of elements within packages, *i.e.* alphabetical order. Second, the layout algorithm does not guarantee any spatial constraints between the leaf packages contained in packages that touch at a higher level. Treemaps may contain very narrow and distorted rectangles. Balzer *et al.* proposed

a modification of the classical treemap layout using Voronoi tessellation [3]. Their approach creates esthetically more appealing treemaps, reducing the number of narrow tessels.

2.3 Cartography metaphors for software

In the software visualization literature however, topic maps are rarely used. Except for the use of graph splatting in RE Toolkit by Telea et al. [15], we are unaware of their prior application in software visualization. A number of software visualization tools have adopted metaphors from cartography. Typically these tools are part of reverse-engineering approach based on extracted models that abstract away from source code. Thus, these tools cannot be used to read source code or develop software.

A number of tools have adopted metaphors from cartography in recent years to visualize software. Usually these approaches are integrated in a tool within an interactive, explorative interface and often feature three-dimensional visualizations.

MetricView is an exploratory environment featuring UML diagram visualizations [31]. The third dimension is used to extend UML with polymetric views [20]. The diagrams use arbitrary layout, so do not reflect meaningful distance or position.

White Coats is an explorative environment also based on the notion of polymetric views [23]. The visualizations are three-dimensional with position and visual-distance of entities given by selected metrics. However they do not incorporate the notion of a consistent layout.

CGA Call Graph Analyser is an explorative environment that visualizes a combination of function call graph and nested modules structure [5]. The tool employs a $2\frac{1}{2}$-dimensional approach. To our best knowledge, their visualizations use an arbitrary layout.

CodeCity is an explorative environment building on the city metaphor [35]. CodeCity employs the nesting level of packages for their city's elevation model and uses a modified tree layout to position the entities, *i.e.* packages and classes. Within a package, elements are ordered by size of the element's visual representation. Hence, when changing the metrics mapped on width and height, the overall layout of the city changes, and thus, the consistent layout breaks.

VERSO is an explorative environment that is also based on the city metaphor [19]. Similar to CodeCity, VERSO employs a treemap layout to position their elements. Within a package elements are either ordered by their color or by first appearance in the system's history. As the leaf elements have all the same base size, changing this setting does not change the overall layout. Hence, they provide consistent layout, however within the spatial limitations of the classical treemap layout.

Data Mountain is a 3D document management system that allows the user to place documents at arbitrary positions on an inclined plane [26]. They use 2D interaction techniques and common pointing devices for all the interactions. Data Mountain is designed to specifically address the human spatial memory to assist with document management. They provide a user study that shows that spatial memory plays an important role in retrieving and localizing documents on the document storage plane. This

tool is only loosely related to SOFTWARECARTOGRAPHER but it is interesting to see that the spatial metaphor works in other areas.

Code Canvas is a research prototype that focuses on the spatial representation of code, oriented on developer's drawings on whiteboards [27]. It represents code on a two-dimensional infinite canvas. When zoomed out one can see an UML overview of the project, when zoomed in all the UML entities become source-code editors. The tool uses the same canvas to visualize the directional relationships and the architectural boundaries where it also allows editing of source code. To our knowledge, it relies on the developer to manually layout the source entities.

Chapter 3

Software Cartography

In this section we present the techniques used to achieve consistent layout for software maps. The number crunching is done by Latent Semantic Indexing and Multidimensional Scaling, whereas the rendering algorithms are mostly from geographic visualization [28]. The SOFTWARECARTOGRAPHER tool[1] provides a proof-of-concept implementation of our technique.

3.1 Lexical similarity

In order to define a consistent layout for software visualization, we need the position of software artifacts and the distance between them to reflect a natural notion of position and distance in reality.

Instead of looking for distance metrics in the graph structure of programs, we propose to focus on the vocabulary of source code artifacts as the space within which to define their position and distance. Lexical similarity denotes how close software artifacts are in terms of their source code's vocabulary. The vocabulary of the source code corresponds to the implemented technical or domain concepts. Artifacts with similar vocabulary are thus conceptually and topically close [17].

Latent Semantic Indexing (LSI) is an information retrieval technique originally developed for use in search engines, with applications to software analysis [22]. Since source code consists essentially of text, we can apply LSI to source code to retrieve the lexical distance between software artifacts.

The examples in this thesis use the HAPAX tool[2] to compute the lexical similarity between software artifacts. Given a software system $\mathcal{S}$ which is a set of software entities $s_1 \ldots s_n$ using terms $t_1 \ldots t_m$, then HAPAX uses LSI to generate an m-dimensional vector space $\mathbb{V}$ representing the lexical data of the software system.

In this vector space $\mathbb{V}$, each software entity is represented by a vector of its term frequencies. Thus, in information retrieval, $\mathbb{V}$ is often referred to as "term-document

[1] http://scg.unibe.ch/research/softwarecartography/
[2] http://scg.unibe.ch/staff/adriankuhn/hapax/

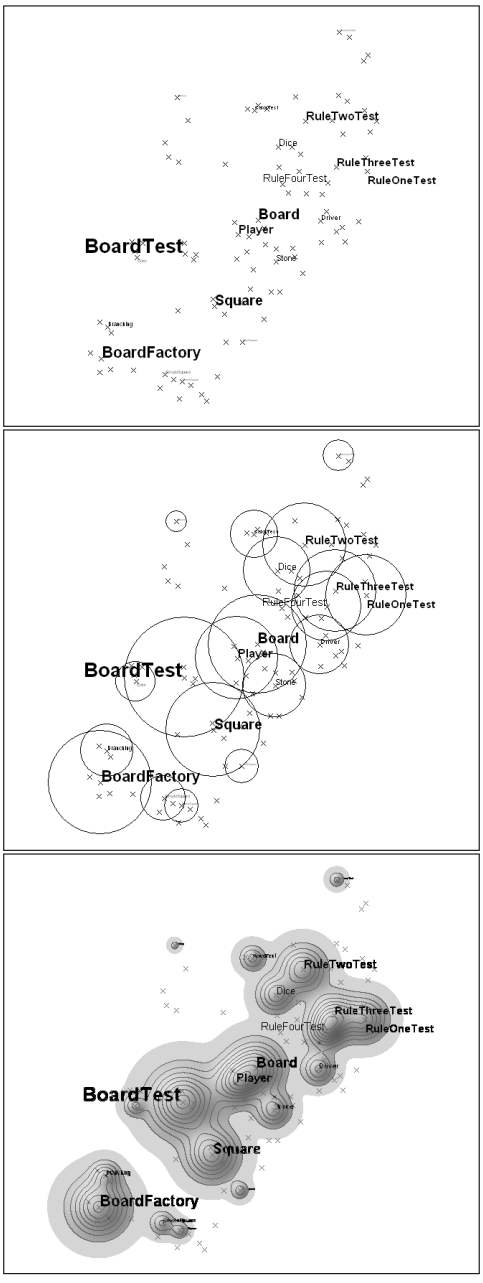


Figure 3.1: *Construction steps of a software map. From top to bottom: 1) dots in the visualization space, positioned with MDS, 2) circles around each entity's location, based on class size in KLOC, 3) digital elevation model with hill-shading and contour lines. Total computation time for the images above about 120 seconds on a 2.16 GHz Intel Core Duo MacBook Pro.*

matrix".

The terms $t_1 \dots t_m$ are the identifiers found in the source code: class names, methods names, parameter names, local variables names, names of invoked methods, et cetera. Thus, two documents (*i.e.* source files or classes) are not only similar if they are structurally related, but also if they use the same identifiers only. This has proven useful to detect high-level clones[21] and cross-cutting concerns[17].

3.2 Multidimensional scaling

The elements shown on the software map are the software entities $s_1 \dots s_n$ labeled with their file or class name. The visualization pane is two-dimensional, whereas Latent Semantic Indexing locates all software entities in an m-dimensional space. Therefor we must map positions in $\mathbb{V}$ down to two dimensions. There are three main techniques to do this, each of which is suitable for very different purposes:

1. *Principal Component Analysis* (PCA) is perhaps the most widely used of all three techniques. PCA yields the best low-level approximation with regard to variance and classification. It tries to preserve as much of the space's variance in the remaining dimensions. Hence, PCA is the best choice for classification problems.

2. *Singular-Value Decomposition* (SVD) treats the m-dimensional space as matrix $A_{n \times m}$, which is the mathematical equivalent of an m-dimensional vector space with n vectors. SVD yields the best low-rank approximation A' under a least-squares criterion. It tries to preserve as much of the space's eigenvalue. Following from this SVD is the best choice for lossy compression and signal reduction problems.

3. *Multidimensional Scaling* (MDS) tries to minimize a stress function while iteratively placing elements into a low-level space. MDS yields the best approximation of a vector space's orientation, *i.e.* preserves the relation between elements as best as possible. For this reason MDS is the best choice for data exploration problems.

For the purpose of software cartography, preserving the relative lexical similarity of software entities is most important. Thus, MDS is the best choice for mapping the LSI vector space to our target visualization space.

MDS attempts to arrange objects in a low-dimensional space, so that the distance between them in the target space reflects their similarity. The input for the algorithm is an $n \times n$ similarity square matrix, where n is equal to the number of objects to display. Each cell (x, y) of the matrix contains the similarity between object x and object y. The dimension of the solution space can range from 2 to $n-1$. As the number of target dimensions decreases, clearly the quality of the approximation deteriorates.

In our case we feed MDS with the lexical similarity of software artifact and map them on a 2-dimensional visualization space. When computing the similarity of lexical data, it is important to use a cosine or Pearson distance metric, as the standard Euclidian distance has no meaningful interpretation when applied to documents and term-frequencies!

3.2.1 Iterative scaling

MDS is an iterative algorithm. Given the similarity between objects as an input, it works as follows:

1. Assign all objects an arbitrary location in the solution space.
2. Determinate the goodness of fit, *i.e.* compare the distance between the objects in the solution space with their similarities given in the input.
3. If the stress value, *i.e.* the goodness of fit, is within a given threshold, the algorithm terminates.
4. Search for a monotonic transformation of the data. That is, far apart but similar objects are moved towards each other and close but not similar objects are moved away from each other. Proceed with step 3.

During the second step it is important to have a good measure for the goodness of the approximation. For this reason the stress value was introduced, which shows the natural goodness of a configuration as a single number. A STRESS value of 0 stands for an optimal solution where the distances between the objects in the configuration perfectly fits their dissimilarity. A higher stress value indicates an increased approximation level between distances and dissimilarities.

3.2.2 Quality indicators

The raw stress function for a given set of n objects ($x_i \in X_{n,d}$) in the d-dimensional target space compares the mutual distances $\hat{d_{ij}} = d(\hat{x_i}, \hat{x_j})$ to the original distances $d_{ij} = d_(x_i, x_j)$ from the similarity square matrix.

$$s = \sum_{i \neq j}^{n} (d_{ij} - \hat{d_{ij}})^2 = min$$

with distances

$$d_{ij}(x_i, x_j) = \sum_{k=1}^{d} (x_{ik} - x_{jk})^2$$

Additionally to the STRESS value Multidimensional Scaling uses a second variable, R, to express the quality of the projection of the height-dimensional vectors to the final, normally two dimensional vector space. R is the quadratic correlation of the distance to the disparities. R represents the level of linear adaptation of the disparities to the distances. In practice values around 0.9 are considered as good.

3.2.3 HiT-MDS

In the SOFTWARECARTOGRAPHER tool, we apply High-Throughput MDS (HiT-MDS), which is an optimized implementation of MDS particularly suited for dealing with large data sets [30]. The algorithm was originally designed for clustering multi-parallel gene

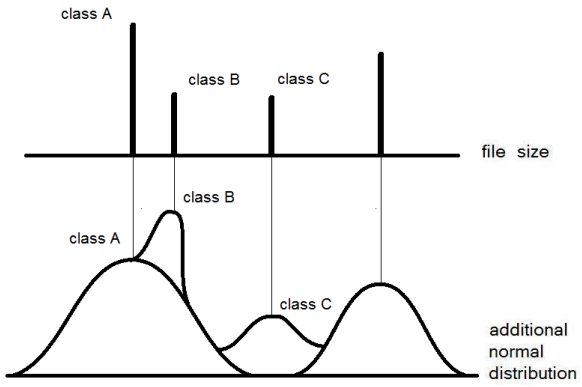


Figure 3.2: *Digital elevation model: each element is represented by a normal distribution according to its KLOC size, the distribution of all elements is summed up.*

expression probes. These data sets contain thousands of gene probes and the corresponding similarity matrix dimension reflects this huge data amount. The price paid for a fast computation is less accurate approximation and a simplified distance metric.

As a consequence of these optimizations, the generated output may vary when run several times on the same input, *i.e.* HiT-MDS uses non-deterministic heuristics. In practice, this appears to be good enough for our experiments with SOFTWARECARTOGRAPHER and software analysis.

3.3 Hill-shading and Contour Lines

In Figure 3.1 we see an overview of the steps taken to render a software map. To make the map more esthetical, we add a touch of three-dimensionality.

The hill-shading algorithm is well-known in geographic visualization. It adds hill shades to a map [28]. The algorithm works on a distinct height model (digital elevation model) rather than on trigonometric data vector date: each pixel has an assigned z-value, its height.

The digital elevation model of SOFTWARECARTOGRAPHER is is a simple matrix with discrete height information for all pixels of the visualization plane. As illustrated on Figure 3.2, each element (*i.e.* source file of class) is represented by a hill whose height corresponds to the element's KLOC size. The shape of the hill is determined using a normal distribution function. To avoid that closely located elements hide each other, the elevation of all individual elements is summed up.

The hill-shading algorithm renders a three-dimensional looking surface by determining an illumination value for each cell in that matrix. It does this by assuming a hypothetical light source and calculating the illumination value for each cell in relation to its neighboring cells.

Eventually, we add contour lines. Drawing contour lines on maps is a very common technique in cartography. Contour lines make elevation more evident than hill-shading alone. Since almost all real world maps make use of contour lines, maps with contour lines are very familiar to the user.

3.4 Evolution of a Software Systems

In order to visualize a series of a software systems over several released versions we had to find a way to resolve the orientation and heuristics problems. It was not possible to compare complete independent computed software maps because their orientation changed between each repeated computation of the analyzed versions. To fix this problem, we computed a map containing all files of all versions of a system. To display then a single version in such an total map we removed all files with different versions and only showed the desired set of files. Doing so the orientation over the series became stable.

The great drawback of this technique is, that it is not possible to enhance such a series with additional versions. Although such a enlarged series would have a nearly identical layout the loss of orientation would affect the user's understanding of the map and his sense for the file locations on the map. To expand an existing series we have to recompute the whole corpus containing all old versions and also the new part.

If we would manage to overcome the orientation problem with a form of anchor point or something similar we would not have to fear this drawback any more and could enhance existing Software Maps with very small effort.

3.5 Labeling

A map without labels is of little use. On a software map, all entities are labeled with their name (class or file name).

Labeling is a non-trivial problem, so we must make sure that no two labels overlap. Also labels should not overlap important landmarks. Most labeling approaches are semi-automatic and need manual adjustment. An optimal labeling algorithm does not exist [28]. For locations that are near to each other it is difficult to place the labels so that they do not overlap and hide each other. For software maps it is even harder due to often long class names and clusters of closely related classes.

The examples given in this paper show only the most important class names. SOFTWARECARTOGRAPHER uses a fully-automatic, greedy brute-force approach. Labels are placed either to the top left, top right, bottom left, or bottom right of their element. Smaller labels are omitted if covered by a larger label. Eventually, among all layouts, the one where most labels are shown is chosen.

The algorithm still leaves ugly overlapping labels in the map or hides important information randomly. The last cut still has to be done by manually placing and adjusting the labels. To show all the desired information on the map appears to be not possible.

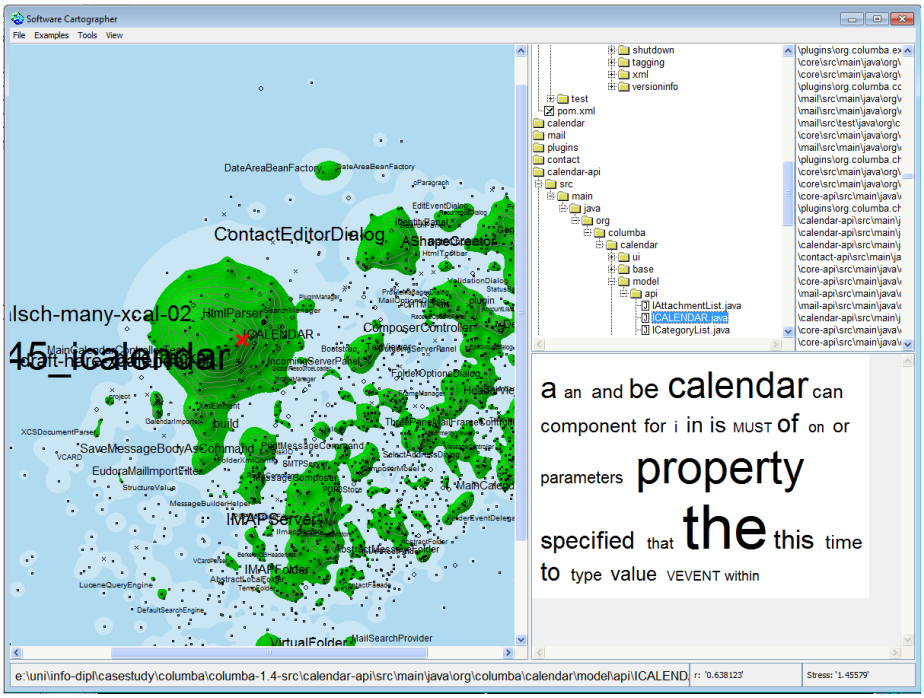


Figure 3.3: SOFTWARECARTOGRAPHER *with 100 labels displayed*

To circumscribe this problem, we introduced a system browser in the SOFTWARECARTOGRAPHER. The browser shows all file names in a list. The user interacts with the map, by clicking near a mountain's peak on the map to mark the corresponding file in the browser view. It is also possible to select a file name in the browser to mark the corresponding peak in the map view. This solution provides more space for further visualizations on the map and allows the user to find information of each file name in the map and vice versa a quick way to find files in the map.

Chapter 4

Case Studies

This section presents examples of software maps. In the first example the SOFTWARECARTOGRAPHER is fed with some well known Multidimensional Scaling test cases. The test data for these experiments are simple text documents and represent some real world scenarios. The second example visualizes the evolution of two software systems to illustrate the consistent layout of software maps. The third section shows an overview of six open-source systems to illustrate their distinct spatial layouts. The last section presents two examples for thematic cartography.

4.1 classic MDS Examples

4.1.1 Color Example

The idea of these examples is to feed the SOFTWARECARTOGRAPHER rather theoretical data with known solutions.

One of the best known examples to test Multidimensional Scaling layouts is the color cycle. Each color consists of at least three different parts and lies therefore in a three or multidimensional space. The goal of my first example is to do such an projection with SOFTWARECARTOGRAPHER and compare the results with known solutions.

Although SOFTWARECARTOGRAPHER is a fast reader, it is also color-blind. So we wrote text files to represent the colors. I.E. the color green is a text file containing the word 'blue' 64 times and the word 'yellow' 64 times. This way all colors of the RGB color system have a suitable text file representation. SOFTWARECARTOGRAPHER will now use Latent Semantic Indexing to compute the similarity matrix between this text representations as distance between the corresponding colors. Figure 4.1 shows the landscape generated from this matrix.

The output of this first experiment is very satisfactory. The SOFTWARECARTOGRAPHER solution shows the well known Color Cicle with white in the middle and the three base colors with largest distance between them.

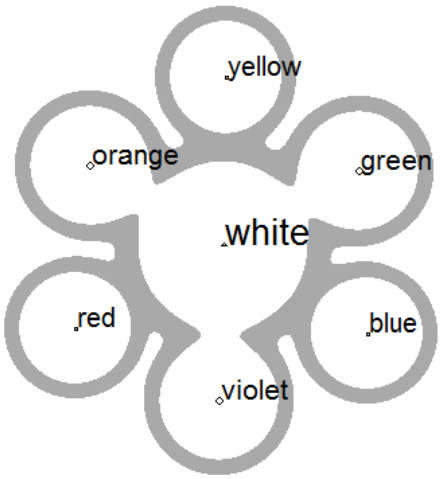
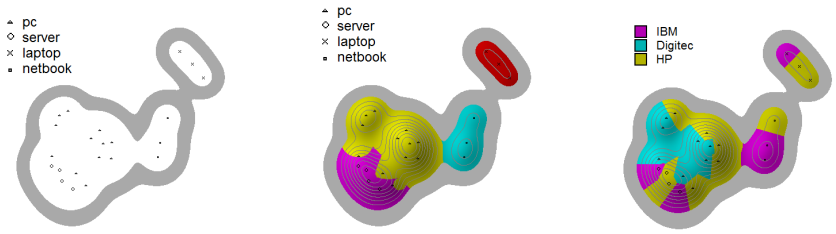


Figure 4.1: *Color Cycle Example: The 2 Dimensional interpretation of* SOFTWARECARTOGRAPHER *for the threedimensional rgb color model*

4.1.2 Computer Example

In the next example we will stay with real world use cases, but increase the number of dimensions and the degree of abstraction between objects and their representing text files. The classical example would be to cluster different car models, feeding text files with representations of the cars motors, maximal speed, economics and so on. Since I really don't like cars' and also would maybe miss the difference between a pickup and a sport car, I choose a similar scenario with computers instead of cars. To reflect the computers attributes in text style, I went to some price/performance evaluation sites and extracted their ratings on the 'computers' parts. *e.g.* quad core processors above 3 GHz reached a evaluation of 10. To represent the attributes of each computer system I simply wrote text files with the amount of words according to their evaluation. As example, the text file for a system with a 3 GHz quad core, 8 GB DDR2 RAM, etc. contains the word 'processor' 10 times as the 3 GHz quad core is evaluated with best score. The keyword 'ram' on the other hand will occur only 5 times, since DDR2 RAM reaches deeper ranks compared to DDR3.

Again, the SOFTWARECARTOGRAPHER found a meaningful visualization of the different computer systems (see Figure 4.2). The produced cluster lie along the virtual performance axis which is also naturally reflected by the systems' prices. The generated 'landscape' also shows several different islands, one for server type systems, one for laptops and netbooks and a third for personal computers. The height of the islands' top indicates the systems' overall performance. Huge mountains mean in this case higher evaluated attributes. The Server Hill contains the highest peak, while netbooks'



Layout and Cluster System types brands

Figure 4.2: *Computer Cluster Example: while the first picture shows significant cluster of computersystems, figure 2 helps to identify different system types in the clusters. Figures 3 shows the distribution of some brands*

evolution level lies close to sea level. Painting meaningful additional information on the top level of this landscape further improves the understanding of the entire 'computer-space'. By coloring the different types of systems, it shows that some pc setups tend more to the performance of servers than to the other representatives of their type.

4.2 Version Overview

The following two examples will represent benefits of a consistent layout over several versions. These examples will demonstrate the intuitive understanding of unknown software by watching a series of maps, each representing a single program's revision. Such a series of views on a growing map will allow us to inspect the projects evolution from a simple to a more complex system.

4.2.1 Ludo

Figure 4.3 shows the complete history of the Ludo system, consisting of four iterations. Ludo is used in a first year programming course to teach iterative development. The 4th iteration is the largest with 30 classes and a total size of 3-4 KLOC. I selected Ludo because in each iteration, a crucial part of the final system is added.

- The first map (figure 4.3, leftmost) shows the initial prototype. This iteration implements the board as a linked list of squares. Most classes are located in the south-western quadrant. The remaining space is occupied by ocean, since nothing else has been implemented so far.
- In the second iteration (figure 4.3, second from the left) the board class is extended with a factory class. In order to support players and stones, a few new classes and tests for future game rules are added. On the new map the test classes are positioned in the north-eastern quadrant, opposite to the other classes. This indicates that the newly added test classes implement a novel feature (*i.e.* testing of the game's "business rules") and are thus not related to the factory's domain of board initialization.

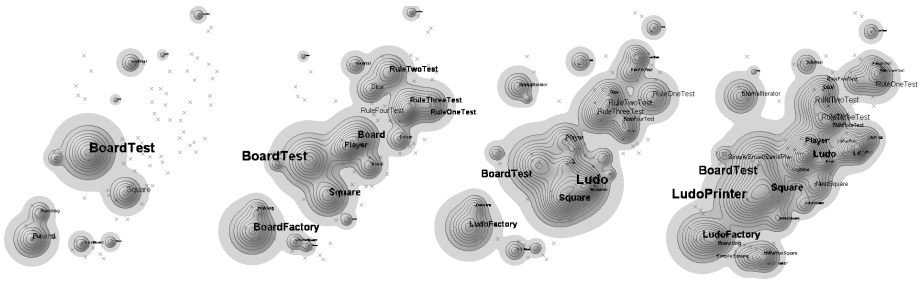


Figure 4.3: *From left to right: each map shows an consecutive iteration of the same software system. As all four views use the same layout, a user can build up a mental model of the system's spatial structure. For example,* `Board/LudoFactory` *is on all four views located in the south-western quadrant. See also figure 4.6 and 4.7 for more views of this system.*

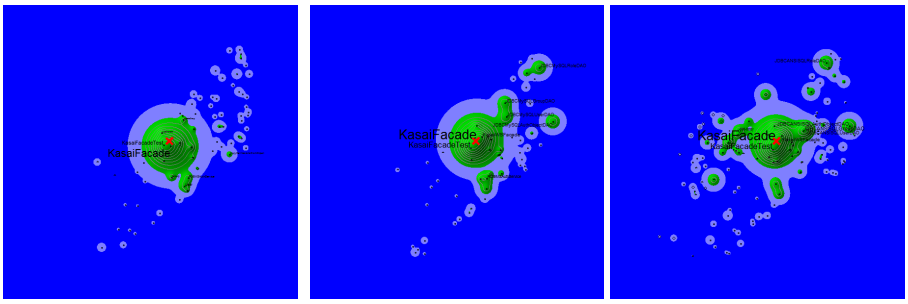
- During the third iteration (figure 4.3, second to the right) the actual game rules are implemented. Most rules are implemented in the `Square` and `Ludo` class, thus their mountain rises. In the south-west, we can notice that, although the `BoardFactory` class has been renamed to `LudoFactory`, its position on the map has not changed considerably.

- The fourth map (figure 4.3, rightmost) shows the last iteration. A user interface and a printer class have been added. Since both of them depend on most previous parts of the application they are located in the middle of the map. As the UI uses the vocabulary of all different parts of the system, the islands start to grow together.

The layout of the most important elements remains stable over all four iterations. For example, the `Board/LudoFactory` class is on all four views located in the south-western quadrant. This is due to LSI's robustness in the face of synonym and polysemy; as a consequence most renaming does not significantly change the vocabulary of a software artifact [17].

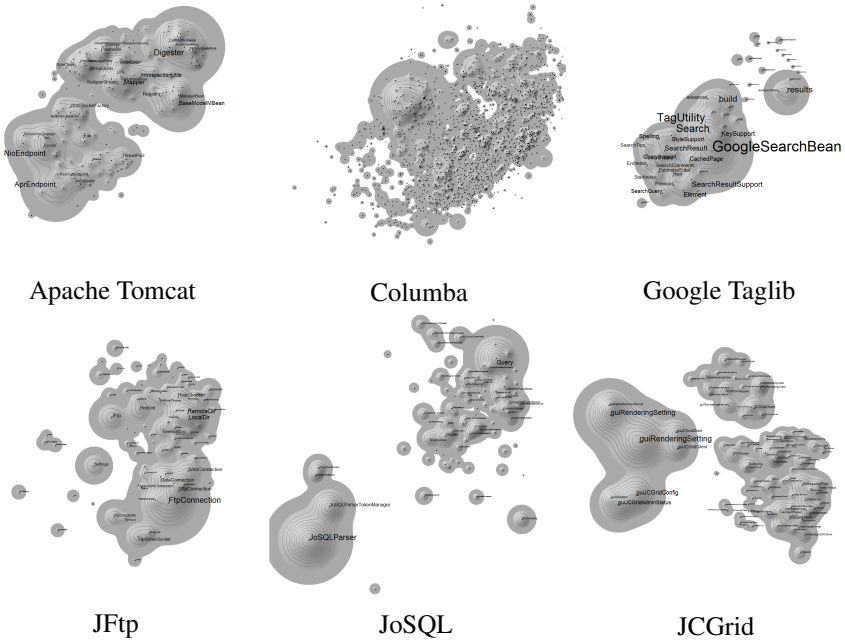
4.2.2 Kasai

Kasai is a 100% Java based authentication and authorization framework. It allows the developer to integrate into his application a granular, complete and manageable permission scheme. The goal of the framework is to provide a simple-to-use-yet-powerful security environment for multi-user applications. Unlike other frameworks, Kasai provides high security abstractions. It's targeted at the specific security requirements that arise in real-life applications such as Intranets, ERPs, CRMs, document managers, accounting systems, etc. In version 0.0.0 Kasi contains 69 Java and JSP files, version 2.0.0 contains over 120 Java and JSP files.



version 1 version 1.1 version 2

Figure 4.4: *Kasai Map: The first version shows a huge mountain, obviously this is the most important class to know - during the evolution towards version 2, the central mountain kept his relatively large volume while the newer and smaller files are placed arround the main massif*



Apache Tomcat Columba Google Taglib

JFtp JoSQL JCGrid

Figure 4.5: *Overview of the software maps of six open source systems. Each map reveals a distinct spatial structure. When consequently applied to every visualization, the consistent layout may soon turn into the system's iconic fingerprint. An engineer might* e.g. *point to the top left map and say: "Look, this huge* `Digester` *peninsula in the north, that must be Tomcat. I know it from last year's code review.".*

4.3 Open-source examples

We applied the software cartography approach to all systems listed in the field study by Cabral and Marques [8]. They list 32 systems, including four of each type of application (Standalone, Server, Server Applications, Libraries) and selected programming languages (Java, .NET).

Figure 4.5 shows the software map for six of these systems: Apache Tomcat, Columba, Google Taglib, JFtp, JCGrid and JoSQL. Each system reveals a distinct spatial structure. Some fall apart into many islands, like JFtp, whereas others cluster into one (or possibly two) large contents, like Columba and Apache Tomcat. The 36 case-studies raised interesting questions for future work regarding the correlation between a system's layout and code quality. For example, do large continents indicate bad modularization? Or, do archipelagos indicate low coupling?

Each system's size in TLC and total project size in Bytes is listed in Table 4.1. For a closer look at the figures please find the listed examples in the appendix.

System	# Top-level java classes	total files	total filesize (Byte)
Apache Tomcat	162	182	1'576'187
Columba	1'549	1'744	6'423'229
Google Taglib	20	59	160'268
JFtp	78	81	356'885
JCGrid	94	94	371'953
JoSQL	83	85	683'699

Table 4.1: *Statistics of the six systems in Figure 4.5.*

4.4 Thematic cartography examples

Software maps can be used as canvas for more specialized visualizations of the same system. In the following, I provide two thematic visualization of the Ludo system that might benefit from consistent layout. (The maps in this subsection are mockups, not yet fully supported by SOFTWARECARTOGRAPHER.)

- Boccuzzo and Gall present a set of metaphors for the visual shape of entities [4]. They use simple and well-known graphical elements from daily life, such as houses and tables. However they use conventional albeit arbitrary layouts, where the distribution of glyphs often does not bear a meaningful interpretation. The first map in Figure 4.6 employs their technique on top of a software map, using test tubes to indicate the distribution of test cases.
- Greevy *et al.* present a three-dimensional variation of System Complexity View to visualize a System's dynamic runtime state [11]. They connect classes with edges representing method invocation, and stack boxes on top of each other to represent a class's instances. Since System Complexity Views do not capture any notion of position, the lengths of their invocation edges do not express any real sense of distance.

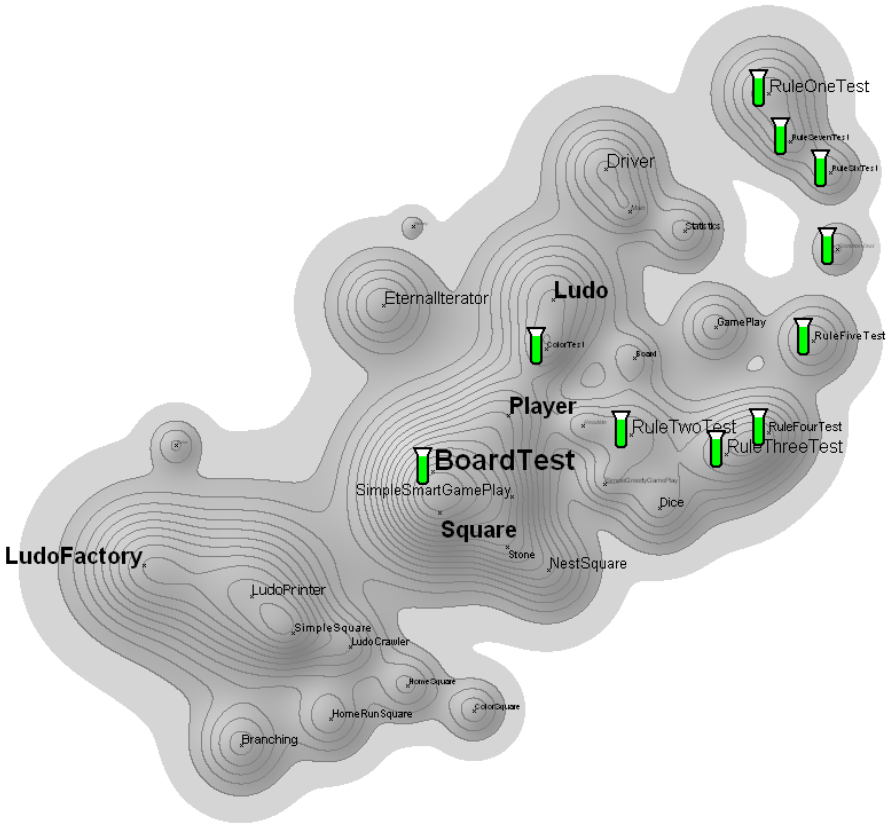


Figure 4.6: *Glyphs are drawn on top of the map, to display additional information. Each test tube glyph indicates the location of unit test case.*

Figure 4.7 employs their approach on top of a software map, drawing invocation edges in a two-dimensional plane. Here the distances have an interpretation in terms of lexical distance, so the lengths of invocation edges are meaningful. A short edge indicates that closely related artifacts are invoking each other, whereas long edges indicate a 'long-distance call' to a lexically unrelated class.

In figure 4.8 we see an industrial J2EE application in which artifacts are colored according to which kinds of files they are. Java source files are crosses and are all in the north west region. JSP files are squares and mostly placed in the south east corner of the island. XML files are triangles and property files are circles. Both of these are mostly in the central region.

Since Java files have to do with the underlying implementation, and JSP files are closer to domain concepts, it is not too surprising that these files are mostly in separate parts of the island. Strangely, however, we do find a number of JSP files in the north west, which could mean that they are more closely linked to the implementation. One of these (`abstract.jsp`), is interesting because it contains a large portion of Java code and only a small amount of HTML/JSP. This explains why its vocabulary places it squarely in the Java part of the island. The abstract.jsp is interesting because it contains a large

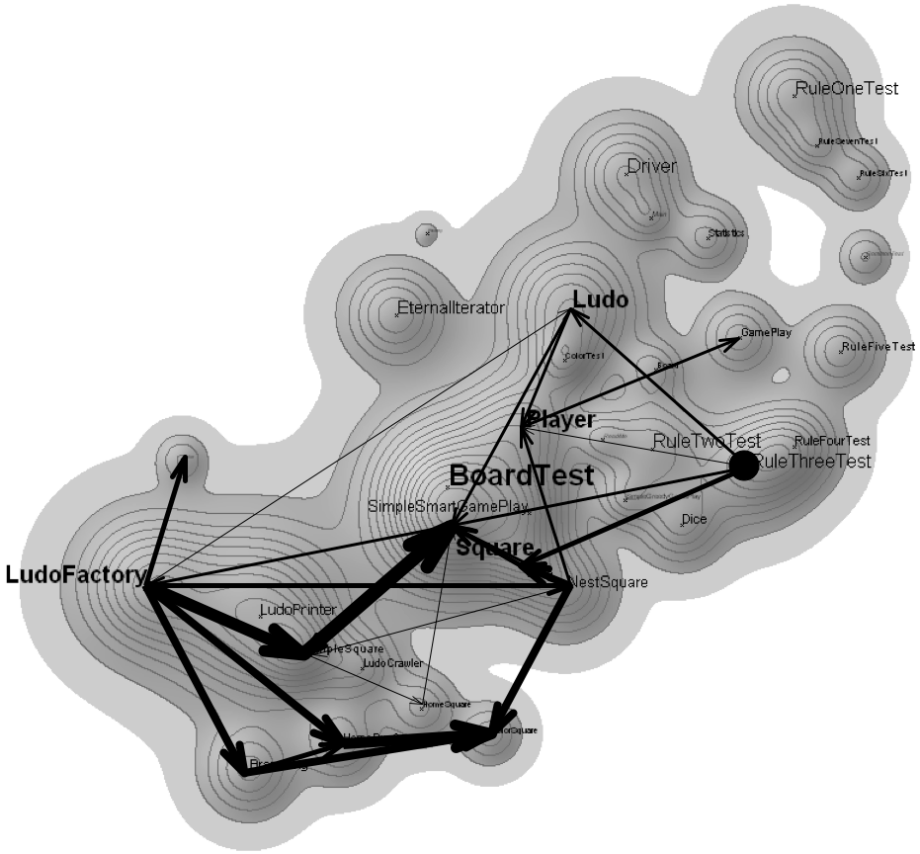


Figure 4.7: *Invocation edges are drawn on top of the map, showing the trace of executing the* `RuleThreeTest` *test case.*

portion of Java code and only a few part of HTML/JSP. So, this is probably why it is in between the rest of the Java code. In particular it uses methods from JspUtils.

Also unusual is a single, isolated XML file on a mountain peak in the south west. Again, the name (`component-definitions.xml`) suggest that it acts as a bridge between the two regions of the island. In fact, it references many of the JSP files, but is also quite large, and it contains many XML tags and attributes which occur nowhere else, which explains why it stands alone on top of a large hill. If you take a look at the component-definitions.xml you will see, that it is a configuration file that references quite a few JSP files, but it is standalone probably because it is huge, and the XML tags and attributes that are used a lot cannot be found in other places.

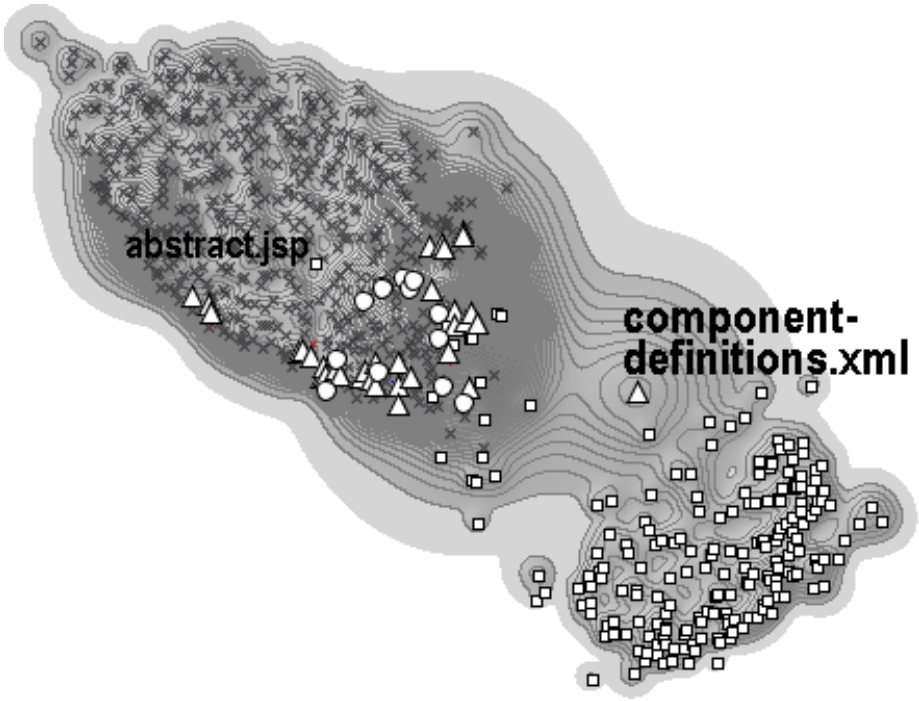


Figure 4.8: *The KMUAdmin JSP application. Java files are displayed as crosses, JSP files as squares, XML files as triangles, and property files as circles.*

Chapter 5

Discussion

5.1 Obstacles in the way of Software Cartography

During the work on the SOFTWARECARTOGRAPHER several problems and drawbacks of software cartography appeared. This chapter will address these points and give some ideas for solutions.

5.1.1 Lexical pollution

One of the difficulties in lexical analysis is to exclude text parts with no dependence to the task of the text. *e.g.* each java file in the google tag library contains identical licence text. For short classes this initial and with all other pieces identical text pollutes the significance of it's representing multidimensional lexical vector dramatically. To overcome this problem, the SOFTWARECARTOGRAPHER should have a possibility to detect and erase such contamination. While SOFTWARECARTOGRAPHER makes no different between different languages and regards them only as words and texts further versions should take more care of language dependent elements.

5.1.2 Heuristic HiT-MDS

Another drawback in using the Multidimensional Scaling algorithm is it's terrible scalability. Because the algorithm has to compute the distance between each part in the system while the starting dimension increases with each additional file, the computation time for the distance matrix increases exponentially. The High Throughput Multidimensional Scaling Algorithm overcomes this problem by a heuristic approach in the computation of the distance matrix. This trick removes the exponential computation time by accepting some imprecision. The results are minor changes between each repetition.

5.1.3 Orientation in MDS

The initial idea of drawing maps for software claimed to generate an intuitive orientation for developers working on large projects. The Multidimensional Scaling algorithm lets orientation vary. Computing the same map twice will deliver two nearly identical landscapes. Their orientation on the other hand will differ. A solution for this problem would be to give the cartographer some anchor points, files with known meaning and fixed position. *e.g.* a text file that contains all GUI typical expressions and rotating the landscape till this file lies north of the map's center [6]. An other approach could be to give the landscapes some additional metric. In Figure 4.2 this might be the performance price axis. By defining that the cheapest system has to lie in the south and the most expensive in the north the landscape's orientation would be fixed. Hermann and Leuthold used this variant to give their political landscapes the appropriate orientation [12].

5.2 Future Work

As future work, we can identify the following promising directions:

- Software maps at present are largely static. Larger software projects tend to produce flat desert similar landscapes bearing not enough information to distinguish between files. Clearly this phenomena can be overcome by a greater level of detail. We envision therefore a more interactive environment in which the user can 'zoom and pan' through the landscape to see features in closer detail, or navigate to other views of the software.
- Selectively displaying features would make the environment more attractive for navigation. SOFTWARECARTOGRAPHER yet includes a native file browser to select and mark single files or folders. It would be nice to support users additionally with notes on the map, where they can add comments and way marks as they perform their tasks.
- Orientation and layout are presently consistent for a single project only. We would like to investigate the usefulness of conventions for establishing consistent layout and orientation (*i.e.* 'testing' is North-East) that will work across multiple projects, possibly within a reasonably well-defined domain.
- The heuristic in the used Multidimensional Scaling algorithm effects the layout of the maps locally. Additionally to the loss of the global map orientation, the heuristic corrupts orientation in minor artifacts of the software maps, *i.e.* three or more files might have a complete independent vocabulary form the remaining project files. The representing islands of these files will be probably arranged in a cycle with no definite direction. Stepping back to an classical Multidimensional Scaling algorithm and adding something like an anchor file containing the whole projects lexicon might lead to a more consistent layout.
- In current work the SOFTWARECARTOGRAPHER does not distinguish between code and for development senseless text like Licenses or similar. In order to solve this lexical pollution problem a closer inspection of the written files would be useful. To filter such pollution without a clue about the text sense can be a

hard task for machines. In future work the user should be able to exclude such pollution manually without being forced to delete it in the analyzed source code.

5.3 IDE integrated CodeMap

The work on Software Maps has been extended by David Erni. He improved the SOFTWARECARTOGRAPHER prototype with Isomap techniques for consistent orientation and integrated these maps in an IDE [18]. The idea of his work is to support developers with a map reflecting the whole system at each time. The map should help developers to gain a stable mental model of their work. Doing so the miniature map function in the IDE should also allow the user to navigate through the whole system or display further metrics or information on the landscape top. Eventually he released a Eclipse plug-in called CodeMap and performed a user study to validate his assumption about the usage of the tool. This study revealed that programmers tended to misinterpret the layout as a measure of structural dependencies. Based on this observation Erni suggests an implementation of *anchored multidimensional scaling* such that developers can initialize the map to their more personal mental model.

Chapter 6

Conclusion

We have presented an approach to visualizing software based on a cartography metaphor, in which Latent Semantic Indexing is used to position the *vocabulary* of software entities in an m-dimension space, and Multidimensional Scaling is then used to map these positions to a two-dimensional display. Digital elevation, hill-shading and contour lines are applied to produce a software map. Finally, software maps can be generated to depict evolution over time of a software system, or they may be decorated to present various kinds of additional thematic information, such as package structure or call relationships.

In spite of the esthetic appeal of hill shading and contour lines, the main contribution of this thesis is not that the visualizations look like a cartographic map, but rather that (i) cartographic position and distance reflect topical position and distance for software entities, and (ii) consistent layout allows different software maps to be easily compared. In this way, software maps reflect world maps in an atlas that exploit the same consistent layout to depict various kinds of thematic information about geographical sites.

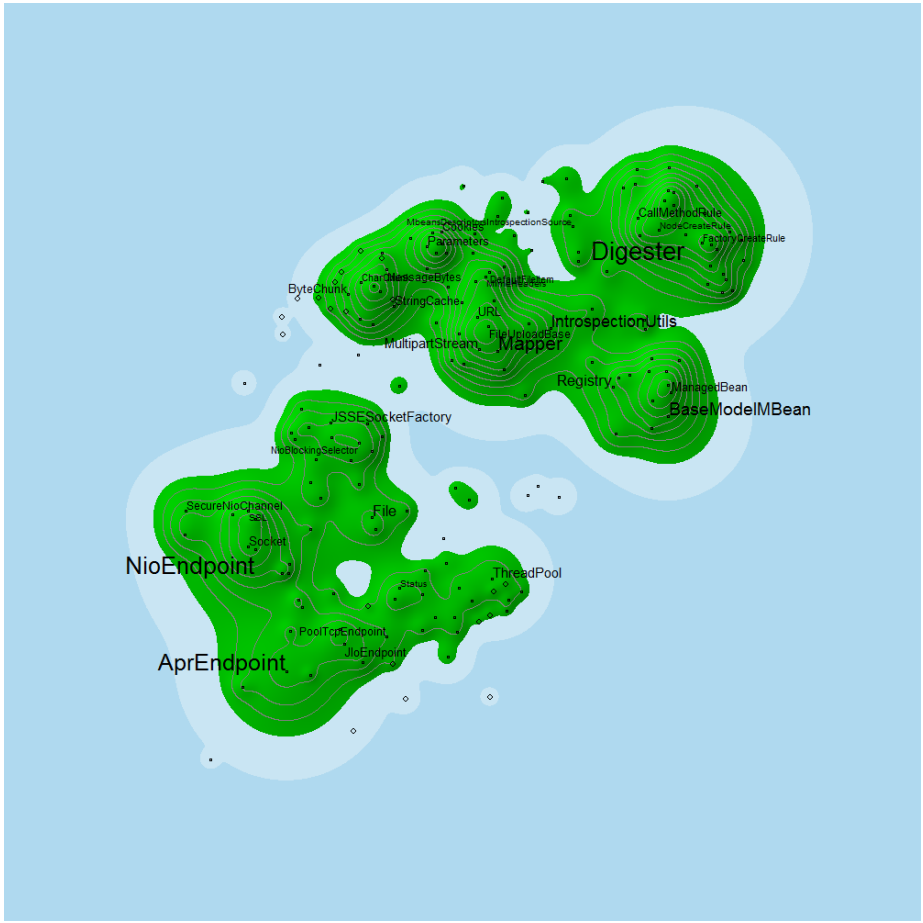
We have presented several examples to illustrate the usefulness of software maps to depict the evolution of software systems. The test series over several versions of such systems confirmed our initial contributions to consistent layout. The produced maps turned out to deliver enough characteristics of a project to help developers to build a mental model and to navigate with rather intuitive knowledge through a project as they would with ordinary alphabetical or hash key layout. Software Maps therefore proved to be helpful as background for further thematic visualizations.

The examples have been produced using SOFTWARECARTOGRAPHER, a proof-of-concept tool that implements our technique. Although a proper labeling of such maps with full file names led to unsolved problems, we found a way to navigate through such software maps by adding a file browser function to SOFTWARECARTOGRAPHER. We also added a word cloud representation for files to the maps in order to support the user with a mental image of the inspected location on the map and support a canvas for future metrics like UML similar class representation.

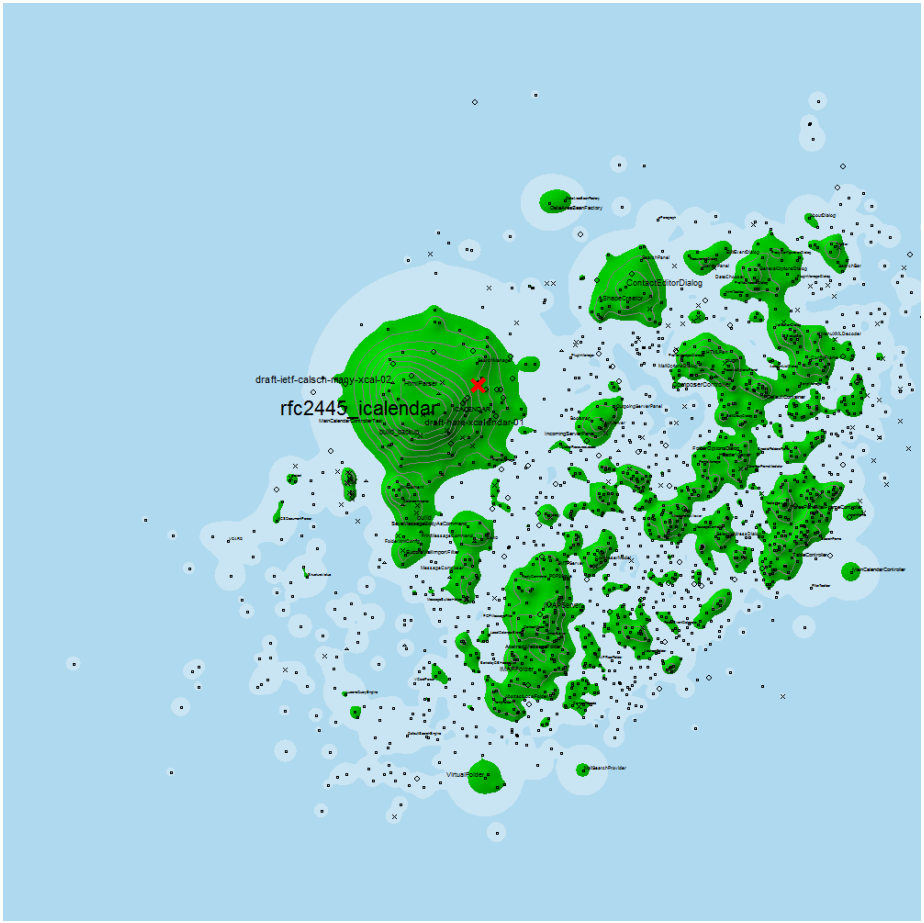
Appendix A

Open-source Examples

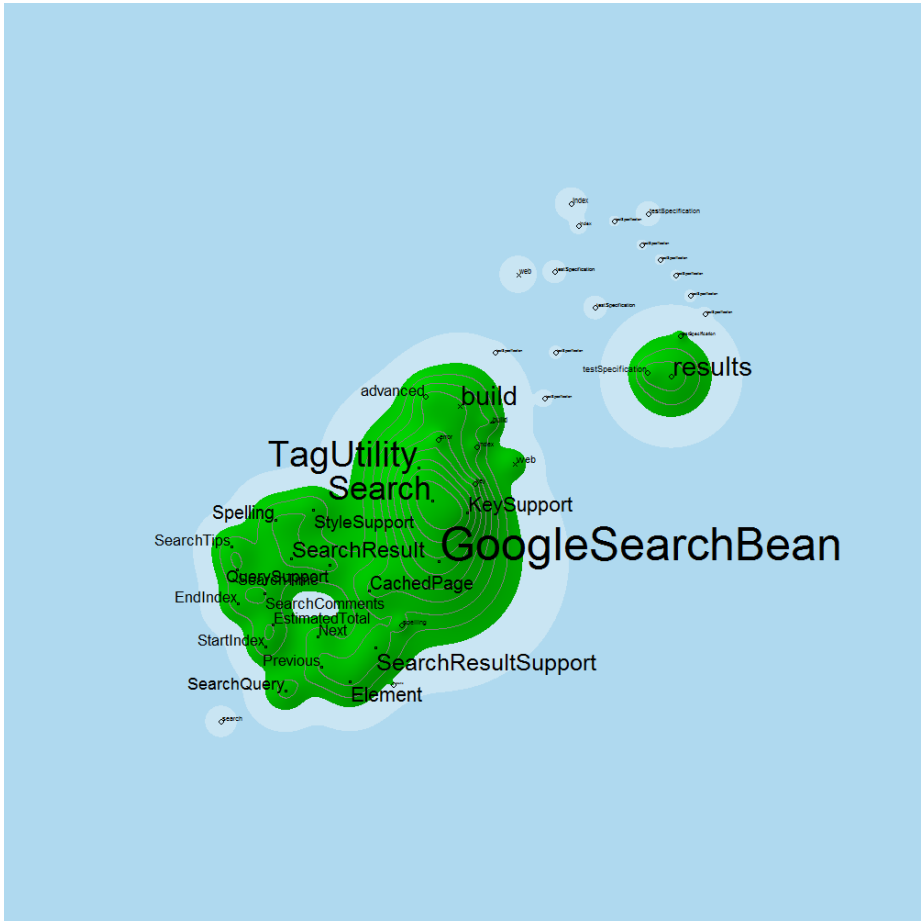
A.1 Tomcat


Digester
CallMethodRule
Parameters
ByteChunk
StringCache
URL
IntrospectionUtils
Mapper
MultipartStream
Registry
ManagedBean
BaseModelMBean
JSSESocketFactory
File
NioEndpoint
ThreadPool
Status
PoolTcpEndpoint
JIoEndpoint
AprEndpoint

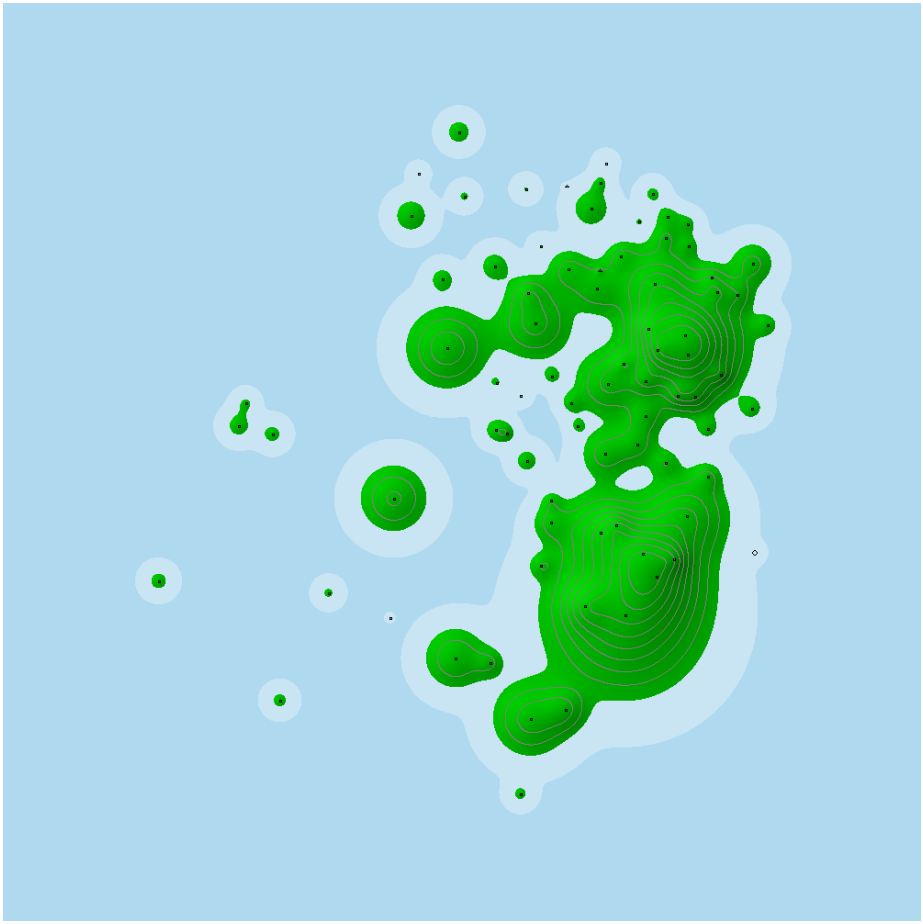
A.2 Columba



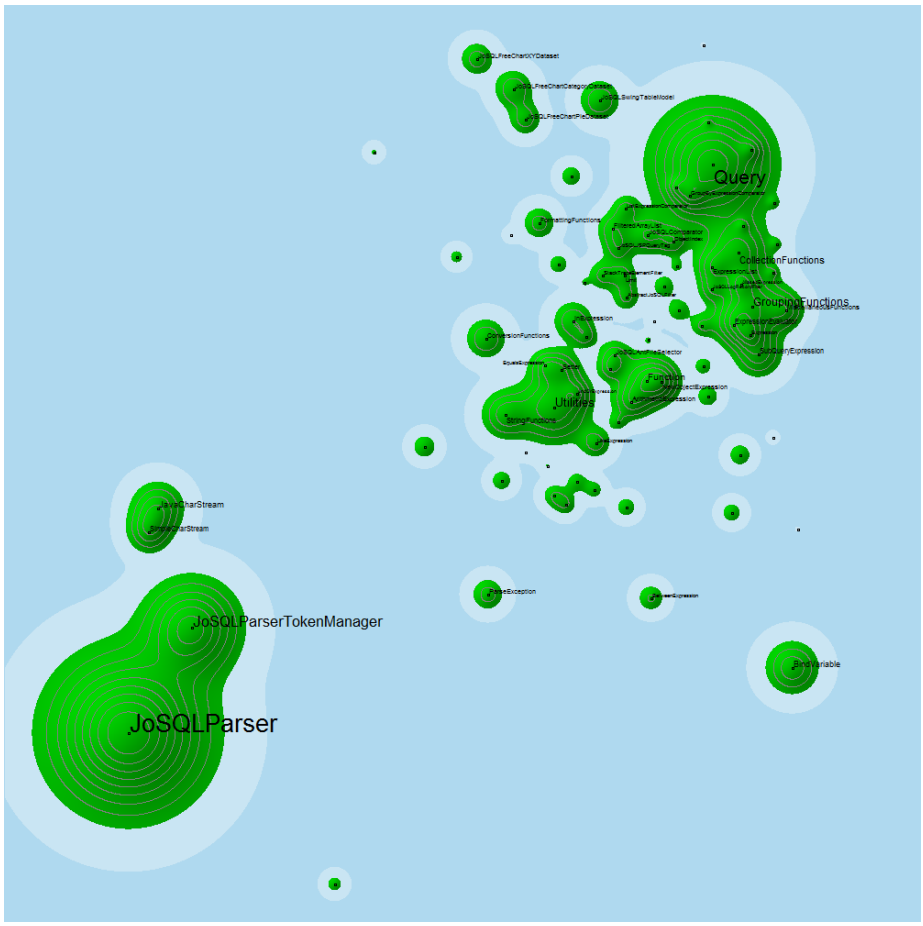
A.3 Google Taglib



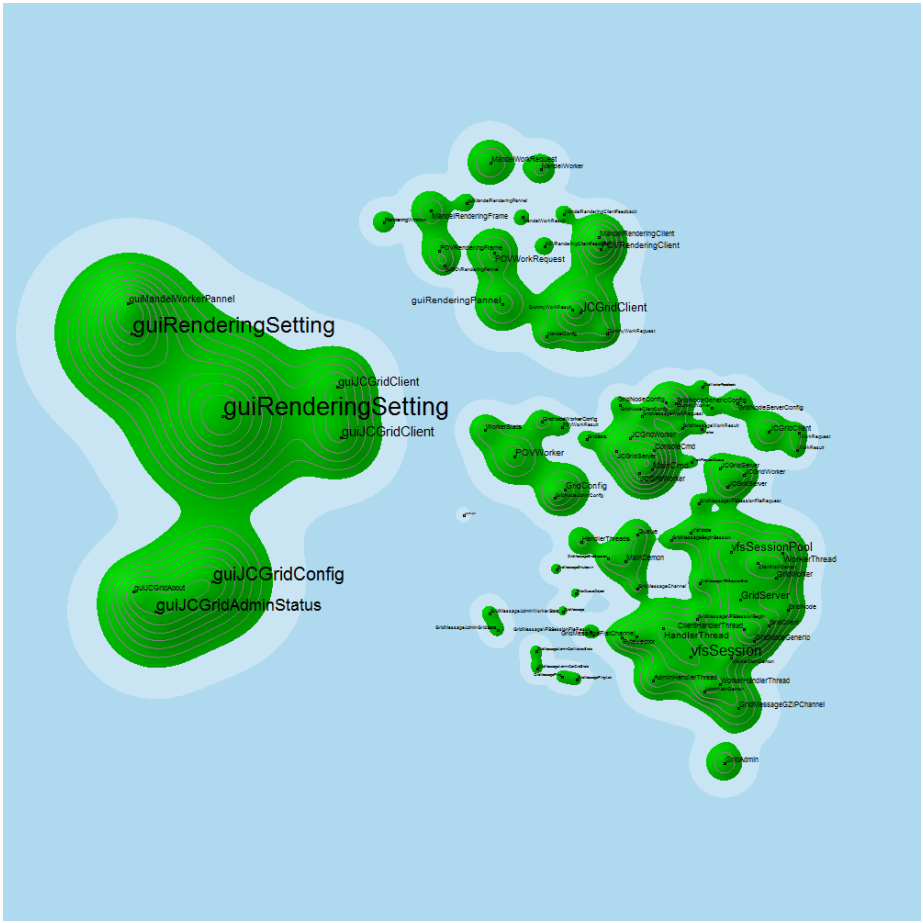
A.4 JFtp



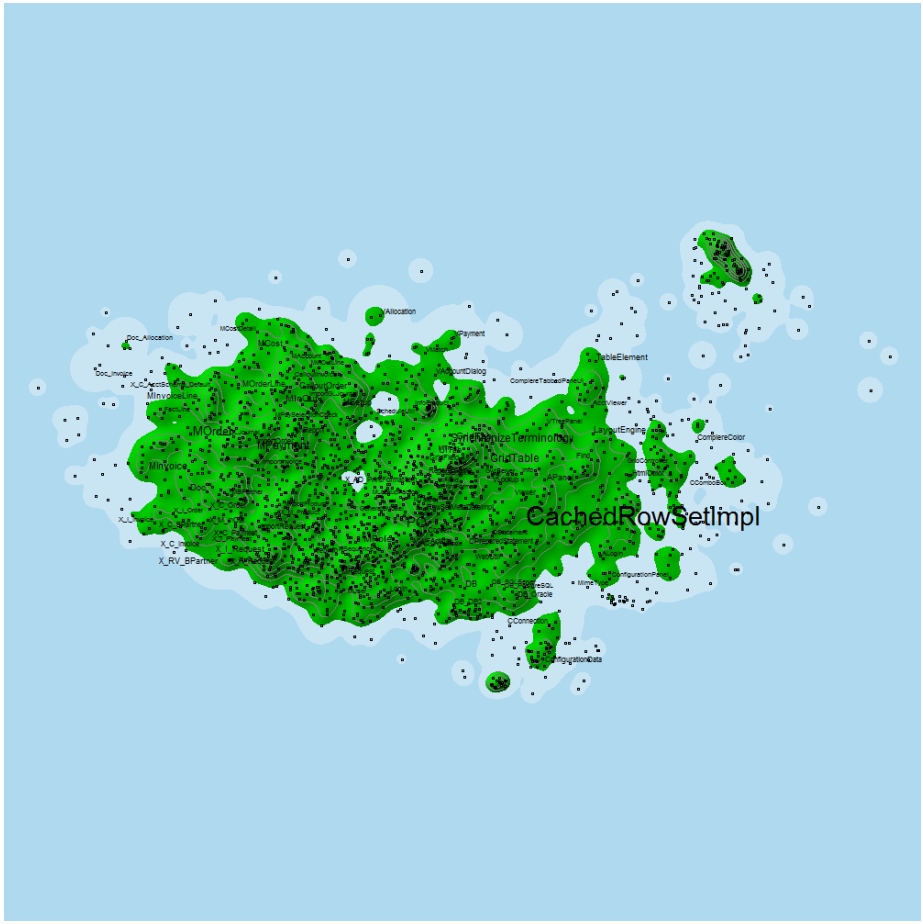
A.5 JoSQL


Query
CollectionFunctions
GroupingFunctions
Utilities
JoSQLParserTokenManager
JoSQLParser

A.6 JCGrid


guiRenderingSetting
guiJCGridClient
guiRenderingSetting
guiJCGridClient
guiJCGridConfig
guiJCGridAdminStatus
guiRenderingPannel
JCGridClient
POVWorkRequest
POVWorker
GridConfig
vfsSessionPool
GridServer
HandlerThread
vfsSession

A.7 Compire



Appendix B

User Guide

B.1 Installation

The above presented SOFTWARECARTOGRAPHER is distributed under MIT License. The software is available for download at the Smalltalk SCG store, but remains under development.

There is also a packaged version downloadable at the SOFTWARECARTOGRAPHER project page [1]. Please find further installation steps in the read-me file.

Both versions works with VisualWorks 7.5. Later VisualWorks releases are not supported. The HiT MDS Algorithm is coded in C and is accessed via make targets. For Windows users Cygwin www.cygwin.com or a similar command line is required. To enable Cygwin for SOFTWARECARTOGRAPHER the Windows path variable should be enhanced with the Cygwin binary location. A system with 512MB RAM should suffice for processing data sets up to 5,000 files.

B.2 Click-Through Example

In the following chapter we will give some Click-Through examples. The presented scenarios will give the user a short overview of the SOFTWARECARTOGRAPHER's features. The examples subsection includes the Color Cycle and Computer Cluster examples from page 27 and explains some of the views. The second subsection shows a way to import documents in order to generate a single Software Map or a series of software maps containing several versions of a software system and contains a list of the selectable parameter of SOFTWARECARTOGRAPHER.

Use either the in Cincom VisualWorks integrated button located on the left side of the 'System Browser' button or the executable from the packaged version to start the SOFTWARECARTOGRAPHER.

[1] http://scg.unibe.ch/download/codemap/software-cartographer-loretan2010.zip

B.2.1 Opening The Thesis Examples

1. Open the MDS computer cluster example discussed in the 27 by selecting 'Examples' SOFTWARECARTOGRAPHER menu and 'MDS Computer Cluster' in the responding drop down menu.

2. Klick on the picture near a mountain pike. The selected location is now marked with a red cross. The file name of the marked location is shown at the bottom of the SOFTWARECARTOGRAPHER window. The File is also selected in the file tree and file browser view on the left side of the SOFTWARECARTOGRAPHER window.

3. Open the 'Version JUnit' examples discussed on page 27 by selecting 'Examples' SOFTWARECARTOGRAPHER menu and 'Version JUnit' in the responding drop down menu.

4. To have more space for the word cloud and the two file views select 'View' in the SOFTWARECARTOGRAPHER menu and select 'Tabbed View'.

5. To browse between the different versions use the 'next' or 'prev' buttons in upper corners of the map. To select a specific version use the dropdown selection box between the buttons.

B.2.2 Opening A Java File System

1. To open a Java project click 'File' in the window menu and select 'Open Java Project'.

2. In the file browser dialog select a folder containing java, xml, jsp, php or txt file types.

B.2.3 Setting Up A Version Example

1. To compute series of software maps containing multiple versions of a software project the original folder structure has to be as following: a top folder with the projects name containing a separate folder for each version.

2. First open the top folder by selecting 'File' in the window menu and select 'Open Java Project'. The SOFTWARECARTOGRAPHER then computes a Software Map for the whole system containing all versions in one Map.

3. After this overview is computed, select 'Set top folder as version' in the 'Tools' menu. Doing so SOFTWARECARTOGRAPHER will switch to version mode and will additionally display a version control interface between the window menu and the landscape image.

4. To select a particular version use the 'next' and 'prev' button to page through the different versions or use the drop down menu between these two buttons to select a version.

B.2.4 Changing Parameter

To set different parameters for the map, select 'Tools' and 'Properties' in the windows menu.

1. 'Metrics and Symbols' properties let you choose, whether the map, the location symbols or neither should be colored. If 'Use color Model' is selected, two different dependency methods are available. In 'Next Neighbor' option, each pixel on the map is colored according to the next location. In 'dominant location' model each point on the map is colored by the location which has the most influences to the points height - that means very small files might disappear because they are covered by larger classes.

2. The 'Map' section let you choose how many labels will be showed in the map and in which size they are displayed. 'Colored' decides if the map base is colored or in black and white. There are also options to disable contour lines, water and hill shade display. There is also an option that controls height, by lowering this value the island surrounding medium water height display will shrink.

3. The 'File Types' option let you decide which file types are integrated in the visualization.

4. 'Model' parameter controls the map resolution and contour line step width. Please use this option with attention. The underlying model takes this value as base side length and will affect computation time dramatically.

Bibliography

[1] Olena Andriyevska, Natalia Dragan, Bonita Simoes, and Jonathan I. Maletic. Evaluating UML class diagram layout based on architectural importance. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:9, 2005.

[2] Giuliano Antoniol, Yann-Gael Gueheneuc, Ettore Merlo, and Paolo Tonella. Mining the lexicon used by programmers during sofware evolution. In *ICSM 2007: IEEE International Conference on Software Maintenance*, pages 14–23, October 2007.

[3] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, New York, NY, USA, 2005. ACM.

[4] Sandro Boccuzzo and Harald Gall. CocoViz: Towards cognitive software visualizations. *VISSOFT 2007. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:72–79, 2007.

[5] Johannes Bohnet and Jurgen Dollner. CGA call graph analyzer — locating and understanding functionality within the Gnu compiler collection's million lines of code. *VISSOFT 2007. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:161–162, 2007.

[6] Andreas Buja, Deborah F. Swayne, Michael L. Littman, Nathaniel Dean, Heike Hofmann, and Lisha Chen. Data visualization with multidimensional scaling. *Journal of Computational and Graphical Statistics*, 17(2):444–472, June 2008.

[7] Heorhiy Byelas and Alexandru C. Telea. Visualization of areas of interest in software architecture diagrams. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 105–114, New York, NY, USA, 2006. ACM.

[8] Bruno Cabral and Paulo Marques. Exception handling: A field study in Java and .NET. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 151–175. Springer Verlag, 2007.

[9] Stephan Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007.

[10] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[11] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing feature interaction in 3-D. In *Proceedings of VISSOFT 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*, pages 114–119, September 2005.

[12] Michael Hermann and Heiri Leuthold. *Atlas der politischen Landschaften*. vdf Hochschlverlag AG, ETH Zürich, 2003.

[13] Kenneth Hite, Craig Neumeier, and Michael S. Schiffer. *GURPS Alternate Earths*, volume 2. Steve Jackson Games, Austin, Texas, 1999.

[14] Susanne Jucknath-John and Dennis Graf. Icon graphs: visualizing the evolution of large class models. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 167–168, New York, NY, USA, 2006. ACM.

[15] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs*. Springer-Verlag, Berlin Heidelberg, 2001.

[16] Holger M. Kienle and Hausi A. Muller. Requirements of software visualization tools: A literature survey. *VISSOFT 2007. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 2–9, 2007.

[17] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[18] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Towards improving the mental model of software developers through cartographic visualization. *CoRR*, abs/1001.2386, 2010.

[19] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[20] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[21] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, November 2001.

[22] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[23] Cédric Mesnage and Michele Lanza. White Coats: Web-visualization of evolving software in 3D. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:40–45, 2005.

[24] Andreas Noack and Claus Lewerentz. A space of layout styles for hierarchical graph models of software systems. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 155–164, New York, NY, USA, 2005. ACM.

[25] Steven P. Reiss. The paradox of software visualization. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 19, 2005.

[26] George Robertson, Mary Czerwinski, Kevin Larson, Daniel C. Robbins, David Thiel, and Maarten van Dantzich. Data mountain: using spatial memory for document management. In *Symposium on User interface software and technology (UIST '98)*, pages 153–162, 1998.

[27] Kael Rowan. Code canvas, March 2009. http://blogs.msdn.com/kaelr/archive/2009/03/26/code-canvas.aspx, archived at http://www.webcitation.org/5mceC6NVX.

[28] Terry A. Slocum, Robert B. McMaster, Fritz C. Kessler, and Hugh H. Howard. *Thematic Carthography and Geographic Visualization*. Pearson Prentice Hall, Upper Saddle River, New Jersey, 2005.

[29] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005.

[30] Marc Strickert, Stefan Teichmann, Nese Sreenivasulu, and Udo Seiffert. High-throughput multi-dimensional scaling (HiT-MDS) for cDNA-Array expression data. In Wlodzislaw Duch, Janusz Kacprzyk, Erkki Oja, and Slawomir Zadrozny, editors, *ICANN*, volume 3696 of *Lecture Notes in Computer Science*, pages 625–633. Springer, 2005.

[31] Maurice Termeer, Christian F.J. Lange, Alexandru Telea, and Michel R.V. Chaudron. Visual exploration of combined architectural and metric information. *VISSOFT 2005. 3rd IEEE International Workshop on Volume*, 0:11, 2005.

[32] Jürgen Wolff v. Gudenberg, A. Niederle, M. Ebner, and Holger Eichelberger. Evolutionary layout of uml class diagrams. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 163–164, New York, NY, USA, 2006. ACM.

[33] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. The inevitable stability of software change. In *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, pages 4–13, Los Alamitos CA, 2007. IEEE Computer Society.

[34] Colin Ware. *Information Visualisation*. Elsevier, Sansome Street, San Fransico, 2004.

[35] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99, 2007.

[36] James A. Wise. The ecological approach to text visualization. *J. Am. Soc. Inf. Sci.*, 50(13):1224–1233, 1999.