Codemap

Improving the Mental Model of Software Developers through Cartographic Visualization

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

David Samuel Erni

Januar 2010

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Adrian Kuhn

Institut für Informatik und angewandte Mathematik

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:

David Samuel Erni
Pappelweg 29
CH-3013 Bern
http://www.deif.ch/
http://scg.unibe.ch/codemap

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

Abstract

Software is intangible and the knowledge about a software system and its architecture is often implicit. Thus the developers' mental model of their software system is an important factor in software engineering.

We want to provide developers, and everyone else involved in software development, with a *shared*, *spatial* and *stable* mental model of their software project. We aim to reinforce this by embedding a cartographic visualization in the IDE (Integrated Development Environment). The visualization is always visible in the IDE, similar to the overview map found in many computer games. For each development task, related information is displayed on the map.

In this thesis we present *Codemap*, an *Eclipse* plug-in that demonstrates the use of software cartography in the context of an IDE. We perform an informal user study to validate our assumptions about the usage of *Codemap*.

Acknowledgements

I would like to express my gratitude to everyone that supported me during the time I was working on this thesis. Only due to your support did I manage to successfully complete his thesis!

First of all I want to thank Adrian Kuhn – this work would not have been possible without him. He supported me by providing new ideas, input and motivation and by taking a lot of time for discussing and working with me.

Prof. Oscar Nierstrasz for giving me the opportunity to write this thesis at the Software Composition Group (SCG) and for his inspirational lectures and his support that contributed a lot to my ongoing interest in computer science.

The whole SCG staff that contributed to this work with their smart or critical comments or with interesting discussions and advice.

Everyone that participated in the user study for agreeing to be a guinea pig for my work.x

All the students that accompanied me, not only during the time of this work but during my whole studies, for the great time we had; be it over lunch, while learning or working in the SCG student pool.

My friends for motivating and supporting me and especially my parents for their unconditional support and for always believing in me.

Thank you!

Dave, January 2010

Contents

Abstract iii

Acknowledgements v

Contents vii

1 Introduction 1
- 1.1 Approach in a Nutshell . . . 2
- 1.2 Contributions . . . 4
- 1.3 Structure of the Thesis . . . 4

2 Related Work 5
- 2.1 Previous Work . . . 5
- 2.2 Other Approaches . . . 6
  - 2.2.1 Desiderata for Spatial Representation of Software . . . 7
  - 2.2.2 Other Layout Approaches . . . 8
  - 2.2.3 More Cartography and Spatiality Metaphors . . . 10

3 Software Cartography 13
- 3.1 From Cartography to Software Cartography . . . 13
- 3.2 On the Choice of Vocabulary . . . 14
- 3.3 The Cartography Pipeline . . . 16
  - 3.3.1 Lexical Similarity between Source Files . . . 16
  - 3.3.2 Creating a Two-Dimensional Layout . . . 17
  - 3.3.3 Hill-shading and Contour Lines . . . 18
  - 3.3.4 Labeling . . . 19
  - 3.3.5 Landscape Coloring and Overlays . . . 20
- 3.4 On Different Layout Algorithms . . . 20
  - 3.4.1 An Introduction to Multidimensional Scaling . . . 21
  - 3.4.2 High-Throughput Multidimensional Scaling . . . 22
  - 3.4.3 Isomap . . . 22
  - 3.4.4 Metric Multidimensional Scaling . . . 24
- 3.5 *Codemap's* Two-Dimensional Reduction . . . 25

4 *Codemap* **29**
4.1 On the Choice of *Eclipse* and *Java* . . . 29
4.2 Supported Programming Tasks . . . 30
4.2.1 Code Navigation . . . 30
4.2.2 Test Coverage . . . 32
4.2.3 Searching the Code . . . 33
4.2.4 Searching References/Declarations . . . 34
4.2.5 Browsing Call Hierarchies . . . 35
4.2.6 Collaboration . . . 37
4.3 Future Features . . . 38
4.3.1 Vocabulary View . . . 38
4.3.2 Vocabulary Search . . . 38
4.3.3 Further *Eclipse* Integration . . . 38
4.3.4 Labeling Schemes . . . 40
4.3.5 Elevation Schemes . . . 41
4.3.6 Finer Granularity . . . 41
4.3.7 Map Wizard . . . 41

5 Implementation **43**
5.1 Architecture . . . 43
5.1.1 Concurrent Calculation Pipeline . . . 43
5.1.2 Visualization Layers . . . 46
5.1.3 Eclipse Integration . . . 48
5.2 On Performance Improvements . . . 48
5.2.1 Fast Elevation Model . . . 48
5.2.2 Nearest Neighbor Lookup . . . 50
5.2.3 SWT Image Performance . . . 50
5.2.4 Translation of *FORTRAN* to JVM Bytecode . . . 51

6 Case Study **53**
6.1 User Study . . . 53

7 Conclusion **57**
7.1 Future Work . . . 58
7.2 Lessons Learned . . . 59

A *Codemap* Quickstart **61**
A.1 Obtaining Codemap . . . 61
A.2 Using *Codemap* . . . 61
A.2.1 Mapview . . . 62
A.2.2 Toolbar . . . 63
A.2.3 Search Bar . . . 64
A.2.4 Coloring Metrics . . . 64
A.2.5 Overlay Metrics . . . 64

B User Study Questions **69**

B.1 What is the Domain of *Outsight*? 69
B.2 Which Technologies are used in *Outsight*? 70
B.3 Which is the Architecture of Outsight? 71
B.4 Which Classes collaborate in a Feature? 72
B.5 Asses the Code Quality 73
B.6 Fix a Bug 74

List of Figures **75**

Listings **77**

Bibliography **79**

Chapter 1

Introduction

Software is intangible and the knowledge about a software system and its architecture is often implicit. Thus the developers' mental model of their software system is important because the most accurate documentation of a project often exists in the developers' heads only. The goal of this thesis is to provide developers with a *shared*, *spatial* and *stable* mental model of their software projects. We propose to embed a tool in the environment of their daily work *i.e.* in their Integrated Development Environment (IDE), to visually support their mental model. We propose this tool to be a cartographic visualization of their software project.

Our goal is that developers arrive at a better mental model based on the spatial representation provided by our tool rather than to visualize the developers' current mental model of their software systems. This decision is supported by the observation that the IDE often influences the programmers' mental model of software by the means of its source code representation. Compare for example the mental model held by an *Eclipse* developer with that of an `emacs` or `vim` user, or with the even more diverging mental model of development in exploratory runtime systems such as Smalltalk and Self [49]. We suggest our visualization to be always visible in the IDE, similar to the overview map found in many computer games, to support the emergence of the developers' mental model.

The mental model can then be used to aid the programmer in various development tasks, for example in code navigation. DeLine observes that developers are consistently lost in source code. He noticed that using textual landmarks to ease the navigation only places a large burden on the developers' cognitive memory [18]. Based on these observations he suggests the usage of new visualization techniques that address the spatial memory of developers to help them navigate their code. DeLine proposes four desiderata that should be satisfied by spatial software navigation [16]. In our most recent work [31] we generalized and extended this list as follows:

1. The visualization should show the entire program and be continuous.

2. The visualization should contain landmarks that allow the developers to find parts of the system perceptually, rather than relying on naming or other cognitive feats.
3. The visualization should remain visually stable as the system evolves (both locally and across distributed version control commits).
4. The visualization should be capable of showing global information overlays.
5. Distance in the visualization should have an intuitive and technically meaningful interpretation.

In previous work, Loretan performed a feasibility study by implementing a prototype of such a visualization called *Software Cartographer* [33]. It was shown that it is possible to generate cartographic maps for a software system that satisfy above desiderata. Previous work also provides a proposal for an algorithm that generates software maps and furthermore shows that this algorithm has proven valuable in generating maps for several software systems. That prototype however was not embedded into an IDE.

In this work we demonstrate the use of software cartography in the context of an IDE by implementing a tool called *Codemap* which aims at helping the programmer establish a consistent mental model of his software project.

1.1 Approach in a Nutshell

In this thesis we apply software cartography in the context of an IDE (Integrated Development Environment). We present a prototype called *Codemap* that aims to help programmers and their software development teams to establish a stable mental model of their software projects.

To achieve this goal, we must first propose a way to generate a stable visualization for software. Previous work suggests such a visualization to be based on vocabulary [31]. This is based on the observation that vocabulary abstracts away from technical details of source code [30] and that the vocabulary of an evolving software system is more stable than its structure [2].

The construction sequence of the visualization is similar to previous work [33, 31]:

First we parse the vocabulary of source files into a term-frequency matrix. Then we transform this term-frequency matrix using Latent Semantic Indexing (LSI) [15] to reduce its complexity. Afterwards a composition of two algorithms is used to calculate the two-dimensional embedding of the parsed source files. First, Isomap [59] is applied. Then Multidimensional Scaling (MDS) [8] is used to embed all software artifacts on the visualization pane. The application of Isomap is an improvement over previous work in order to assist MDS with the global layout. The result from this first step is a two-dimensional position on the visualization pane for each software entity.

In the next step, an elevation model is created. Each software entity contributes a Gaussian shaped basis function to the elevation model according to its size. The contributions of all software entities are summed up and normalized. The result of this step is a pixel by pixel elevation model.

In the final step, we use cartographic visualization techniques to render a landscape which is colored depending on the height of each pixel. Furthermore we add hill-shading and contour lines to render a visually appealing landscape. Then, metrics and markers are rendered in transparent layers on top of the landscape. Users can toggle the visibility (on/off) of each separate layer and thus customize the map according to their needs.

As described above, software cartography uses a spatial visualization of software systems to provide software development teams with a stable and shared mental model. Our cartographic visualization is most useful when it supports as many development tasks as possible. Therefore we integrated software cartography in the *Eclipse* IDE so that a map of the software system may always be present and may thus support as many development tasks as possible.

At the time of writing, *Codemap* supports the following programming tasks:

- Navigation within a software system, be it for development or analysis. *Codemap* is integrated with the package explorer and editor of *Eclipse* and linked to the user's current selection. Open files and the currently active file in the editor are indicated on the map. Double clicking on the map opens the closest file in the editor, right clicking on the map displays the default action menu for the current selection. When using heat map mode, recently visited classes are highlighted.
- Comparing software metrics to each other, *e.g.* to compare bug density with code coverage. *Codemap* is hooked into several *Eclipse* plug-ins in order to display their results on the map alongside the regular views. The map displays search results, compiler errors, and (given the *Eclemma* plug-in is installed) test coverage information.
- Social awareness of collaboration in the development team. *Codemap* can connect two *Eclipse* instances to show open files of other developers (given the *Eclipse Communication Framework* plug-in is installed). Colored icons are used to show the currently open files of the other developer, and the open files are updated continuously.
- Exploring a system during reverse engineering. *Codemap* is integrated with *Eclipse*'s structural navigation functions such as search for callers, implementers, and references. Pins are shown for search results, arrows represent call hierarchies. We apply the *Flow Map* algorithm [44] to avoid visual clutter by merging parallel arrow edges.

1.2 Contributions

This thesis builds on previous work, in which we first proposed software cartography for consistent layout of software visualizations [33, 31]. A first prototype of *Codemap* (back then still called *Software Cartographer*) was implemented by Peter Loretan and published at the WCRE 2008 conference[1] [33]. This led to an extended journal version that examined the stability of the software cartography algorithm [31].

The main contributions of this thesis are:

- *Composite layout algorithm.* Previous work presented a layout algorithm based on Multidimensional Scaling only [31]. In this thesis we propose the use of an improved algorithm that calculates the layout in two sequential steps. The first step calculates a global layout and serves as input for the second step which performs local optimizations.
- *IDE embedding.* Previous work presented a stand-alone analysis tool that was not linked to the developer's activities. We embed *Codemap* in an IDE, supporting the programmer in various development tasks. We provide overlays for same visualization that support different development tasks and thus help the programmer to compare the visualizations of different kinds by addressing his spatial memory of the system.
- *Performance.* Previous work was not optimized for performance due to different requirements. In this work, we improved the algorithm greatly in performance to guarantee a smooth user experience.
- *User study.* So far, there has been no validation of *Codemap* involving an user study. In this work, we perform an informal user study to validate our assumptions about the usage of the tool. We propose improvements to the tool that arose from these studies.

1.3 Structure of the Thesis

Chapter 2 discusses other research in this area. Chapter 3 explains how software cartography renders the two-dimensional cartographic visualization. Chapter 4 motivates why we integrate software cartography into an IDE and enumerates the of supported programming tasks. Chapter 5 discusses more details on the implementation of the plugin and gives an explanation of the performance improvements. Chapter 6 discusses the results of our user study and Chapter 7 concludes.

Appendix A provides a quick start on installing and using *Codemap* and Appendix B shows the questions designed for the user study.

[1] Working Conference on Reverse Engineering

Chapter 2

Related Work

There has been a lot of research in the field of software visualization. In this chapter we enumerate a selection of the publications that have a significant relation to our work. We compare them to our work by summarizing similarities and differences.

Section 2.1 describes *Software Cartographer*, the predecessor of *Codemap* and Section 2.2 compares our work to the state of the art in the field of software visualization.

2.1 Previous Work

This section gives a brief description of *Codemap*'s predecessor, *Software Cartographer*, and explains the development steps taken since.

Kuhn and Loretan present a prototype of Software Cartography called *Software Cartographer* as depicted in Figure 2.1. *Software Cartographer* is a standalone prototype to support the analysis of software projects. *Software Cartographer* is not embedded into an IDE but an external tool rendering maps for a project after the corresponding folder is selected from the file system.

Codemap is the successor of *Software Cartographer* and switches the focus from standalone analysis to IDE integration. *Codemap* aims at helping the developers arrive at a better mental model of their software project while working with the code. Thus *Codemap* is embedded into the *Eclipse* IDE and linked with several development tasks. Note that *Codemap* had to be ported from *Smalltalk* to *Java* to allow embedding in *Eclipse*.

Furthermore with *Codemap* we improved the layout algorithm and the performance, and we enabled linking with daily development tasks. *Codemap* is discussed in detail in Chapter 4.

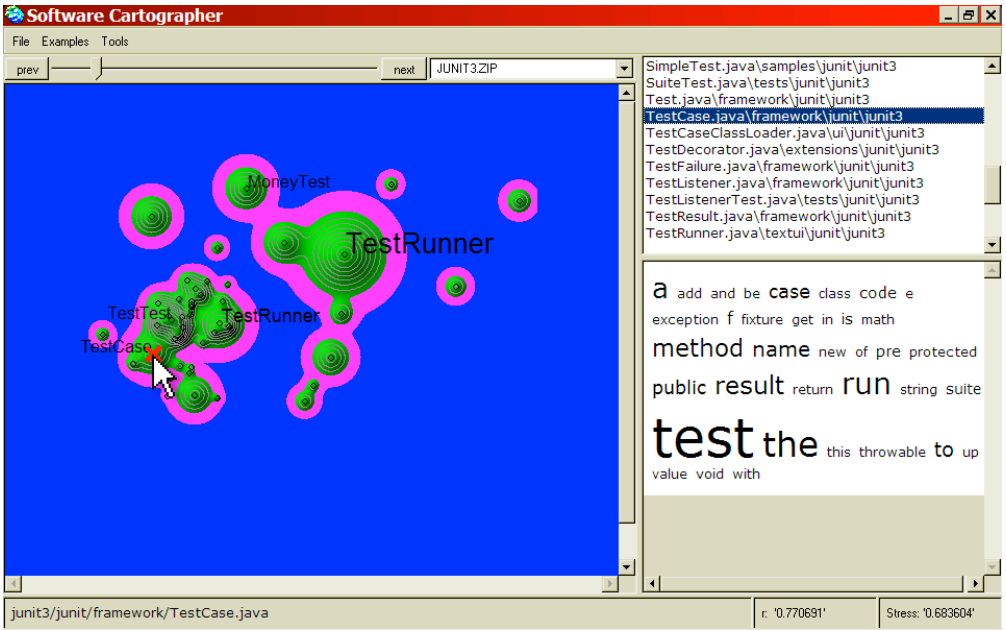

Figure 2.1: *Software Cartographer* was developed as a standalone prototype to support the analysis of software projects. It is not embedded into an IDE but renders maps for a project after selection from the file system.

2.2 Other Approaches

In this section we present existing work related to software cartography. In Subsection 2.2.1 we first summarize the work related closest to software cartography, DeLine's work on software navigation. Then we compare our approach to other work performed in the software visualization field. Subsection 2.2.2 enumerates other layout approaches in software visualization and Subsection 2.2.3 lists other tools that have adopted cartography metaphors.

Using dimensionality reduction to visualize information based on the metaphor of cartographic maps is by no means a new idea. *Topic maps*, as they are called, have a longstanding tradition in information visualization [66]. Our work was originally inspired by Michael Hermann's and Heiri Leuthold's work on the political landscapes of Switzerland [24].

Stable layouts have a long history in information visualization. As a starting point see *e.g.* the recent work by Frishman and Tal on online dynamic graph drawing [22]. They present an online graph drawing approach, which is similar to the pipeline presented in this work. Please refer to Subsection 2.2.2 for a detailed comparison of graph drawing and our approach.

2.2.1 Desiderata for Spatial Representation of Software

This section summarizes DeLine's work on software navigation. Most importantly an extended list of five desiderata for spatial representations is given.

DeLine's work on software navigation [16, 17] closely relates to software cartography. His work is based on the observation that developers are consistently lost in code [18] and that using textual landmarks only places a large burden on cognitive memory. He concludes the need for new visualization techniques that allow developers to use their spatial memory while navigating source code.

Software Terrain Map is, like *Codemap*, based on the metaphor of cartographic maps [16]. It provides a continuous landscape that provides visual landmarks to keep the users oriented. It mimics the continuous nature of cartographic maps by partitioning the screen into tiles and assigning them to software components. It uses function call analysis and file read-writes to measure the affinity of these components. The affinity is then used to layout similar components closer to each other. However, it lacks stability and locality and does not display labels for a better orientation [65].

DeLine proposes four desiderata [16] that should be satisfied by spatial software navigation: 1) the display should show the entire program and be continuous, 2) the display should contain visual landmarks that developers can find parts of the program perceptually rather than relying on names, 3) the display should remain visually stable during navigation [and evolution], and 4) the display should be capable of showing global program information overlays other than navigation.

An ad-hoc algorithm that satisfies the first and fourth properties is presented in the same work. As distance metric between software entities (here, methods) an arbitrary chosen score is used.

Our work satisfies all above desiderata, and completes them with a fifth desideratum that visual distance should have a meaningful interpretation. The scope of software cartography is broader than just navigation. It is also intended for reverse engineering and code comprehension in general. We can thus generalize the five desiderata for spatial representation of software as follows:

1. The visualization should show the entire program and be continuous.
2. The visualization should contain visualization landmarks that allow the developers to find parts of the system perceptually, rather than relying on name or other cognitive causes.
3. The visualization should remain visually stable as the system evolves.
4. The visualization should should be capable of showing global information overlays.
5. On the visualization, distance should have a meaningful interpretation.

2.2.2 Other Layout Approaches

This section summarizes other layout approaches used in software visualization and elaborates the differences to *Codemap*.

Most software visualization layouts are based on one or multiple of the following approaches: UML diagrams, force-based graph drawing, tree-map layouts and polymetric views.

ThemeScape is the best-known example of a text visualization tool that uses the metaphor of cartographic maps. Topics extracted from documents are organized into a visualization where visual distance correlates to topical distance and surface height corresponds to topical frequency [69]. The visualization is part of a larger toolset that uses a variety of algorithms to cluster terms in documents. For laying out small document sets MDS is used; for larger document sets a proprietary algorithm, called "Anchored Least Stress", is used. The digital elevation model is constructed by successively layering the contributions of the contributing topical terms, similar to our approach.

UML diagrams generally employ no particular layout and do not continuously use the visualization pane. The UML standard itself does not cover the layout of diagrams. Typically a UML tool will apply an unstable graph drawing layout (*e.g.* based on visual optimization such a reducing the number of edge crossings) when asked to automatically layout a diagram. However, this does not imply that the layout of UML diagrams is meaningless. UML diagrams are carefully created by architects, at least those made during the design process, so their layouts do have a lot of meaning. If you change such a diagram and re-show it to its owner, the owner will almost suddenly complain, since he invested time in drawing the diagram a certain way! Alas, this layout process requires manual effort.

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) [62]. Although the resulting layout does not reflect a distance metric, in principle the technique could be adapted to do so. 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 aesthetically 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 [12]. The impact of layout on their approach is not discussed.

Graph drawing refers to a number of techniques to layout two- and three-dimensional graphs for the purpose of information visualization [66, 27]. Noack et al. offer a good starting point for applying graph drawing to software visualization [43]. Jucknath-John et al. present a technique to achieve stable graph layouts over the evolution of the displayed software system [26], thus achieving consistent layout, while sidestepping

the issue of reflecting meaningful position or distance metrics.

Unlike our approach, graph drawing is concerned with the placement of vertices and edges such that visual properties of the output are optimized. For example, algorithms minimize the number of edge crossings or try to avoid that nodes overlap each other. Even though the standard force-based layouts can consider edge weights (which can be seen as a distance metric), edges with the same weight may have different lengths on the visualization pane depending on the connectedness of the graph at that position. Furthermore, the resulting placement is not continuous. The void between vertices is not continuous a spectrum of metric locations, as it is the case with our layout.

Graph splatting is a variation of graph drawing, which produces visualizations that are very similar to thematic maps [63]. Graph splatting represents the layout of graph drawing algorithms as a continuous scalar field. Graph splatting combines the layout of graph drawing with the rendering of thematic maps. Each vertex contributes to the field with a Gaussian shaped basis function. The elevation of the field thus represents the density of the graph layout at that position. Telea et al. apply Graph splatting in their RE toolkit to visualize software systems [58]. However, they are not concerned with stable layouts. Each run of their tool may yield a different layout.

Except for the use of graph splatting in RE Toolkit, we are unaware of the application of topic maps in software visualization. And even in the case of the RE toolkit, the maps are not used to produce consistent layouts for thematic maps, or to visualize the evolution of a software system.

Treemaps represent tree-structured information using nested rectangles [66]. Though treemaps make continuous use of the visualization pane, the interpretation of position and distance is implementation dependent. Classical treemap implementations are known to produce very narrow and thus distorted rectangles. Balzer et al. proposed a modification of the classical treemap layout using Voronoi tessellation [5]. Their approach creates aesthetically more appealing treemaps, reducing the number of narrow tessels. There are some treemap variations (*e.g.* the strip layout or the squarified layout) that can, and do, order the nodes depending on a metric. However, nodes are typically ordered on a local level only, not taking into account the global co-location of bordering leaf nodes contained in nodes that touch at a higher level. Many treemaps found in software visualization literature are even applied with arbitrary order of nodes, such as alphanumeric order of class names.

Polymetric views visualize software systems by mapping different software metrics on the visual properties of box-and-arrow diagrams [36, 37]. Many polymetric views are ordered by the value of a given software metric, so that relevant items appear first (whatever first means, given the layout). Such an order is more meaningful than alphabetic (or worse, hash-key ordering), but on the other hand only as stable as the used metric. The System Complexity view is by far the most popular polymetric view, and is often used as a base layout where our requirements for stability and consistence apply (see *e.g.* [23]). The layout of System Complexity uses graph drawing on inheritance relations, and orders the top-level classes as well as each layer of

subclasses by class names. Such a layout does not meet our desideratum for a stable and consistent layout.

2.2.3 More Cartography and Spatiality Metaphors

This section lists other visualization tools based on cartography or spatial metaphors.

A number of tools have adopted metaphors from cartography in recent years to visualize software. Usually these approaches are integrated in a tool with in an interactive, explorative interface and often feature three-dimensional visualizations. None of these approaches satisfies DeLine's desiderata.

MetricView is an exploratory environment featuring UML diagram visualizations [60]. The third dimension is used to extend UML with polymetric views [36]. The diagrams use arbitrary layout, so do not reflect meaningful distance or position.

White Coats is an exploratory environment also based on the notion of polymetric views [40]. 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 exploratory environment that visualizes a combination of a function call graph and nested module structure [7]. The tool employs a $2\frac{1}{2}$-dimensional approach. To our best knowledge, their visualizations use an arbitrary layout.

CodeCity is an exploratory environment building on the city metaphor [67]. 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 exploratory environment that is also based on the city metaphor [34]. 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 all have 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 [47]. 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 *Codemap* but it is interesting to see that the spatial metaphor works in other areas.

Code Thumbnails provides two different thumbnail views to navigate source-code [17]. The Code Thumbnail Scrollbar supplements the document's scrollbar with a thumbnail image of the entire document and helps navigating within a file. The Code Thumbnail Desktop supports navigation between files by showing a thumbnail image of every source file on a desktop surface. Code Thumbnails address the spatial memory of the developer by using the shape of the code itself as visual landmarks. Hence their layout is strongly coupled to the coding style of the individual developer and easily broken by reformatting the code or sorting class members. So unlike *Codemap* their layout strongly depends on the (visual) structure of the code.

Code Canvas is a research prototype that focuses on the spatial representation of code, oriented on developer's drawings on whiteboards [48]. It represents code on a two-dimensional infinite canvas. When zoomed out one 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 best knowledge, it relies on the developer to manually layout the source entities.

Microsoft Visual Studio 2010 uses a dependency graph visualization to draw an overview of the application [54]. The dependency line thickness is given by the depth of the dependency. Furthermore it displays an externals box to capture all the external dependencies, thus it can also display references to code outside the current project. Contrary to *Codemap* they use an arbitrary layout for their visualization, similar to the layout of polymetric views.

Chapter 3

Software Cartography

In this chapter, we present the main concept of software cartography. Section 3.1 discusses properties of traditional cartographic visualization. Section 3.2 presents why we base the layout on vocabulary. Section 3.3 discusses how we create a visualization inspired by cartography. Section 3.4 enumerates different approaches to create a layout based on vocabulary whereas Section 3.5 discusses the approach taken by *Codemap*.

3.1 From Cartography to Software Cartography

In this section we analyze properties of traditional cartographic visualizations. We segregate two important properties that should also hold for software cartography.

Software visualization is an attractive method to abstract away from the complexity of large software systems. A single visualization can represent a large amount of information (*e.g.* the structure, collaboration, coupling, code ownership, code growth ...) about complex software projects [19, 28, 45, 56]. Unfortunately it is hard to compare the great number of different visualizations of the same software project since almost every visualization adopts its own layout. That way the visualizations originally developed to abstract away from the complexity of software systems have led to another source of complexity, their incomparability.

The maps found in a conventional atlas stand in contrast to the situation in software visualization. Different information (*e.g.* population density, flow of trade, migration, industry sectors, birth rate, ...) is displayed on top of the same consistent layout, the natural shape of the underlying landscape. Thus for the viewer it is easy to correlate different visualizations concerning the same geographical location since they correspond to the viewer's underlying mental model of the landscape. This increases the viewer's understanding of the connections between the visualized information.

When comparing multiple data-sets representing different information for one geographical location the viewer can focus on the same location in the visualization and has just to process the visual change of the data representation.

This is possible because of the following reasons:

1. *Two-dimensional representation:* Geographic positions and distances can be easily mapped to a two-dimensional layout. On a local scale the earth can be considered as almost flat. On a global scale cartographic projections (*e.g.* the Mercator projection) can be used for a natural mapping.
2. *Cardinal directions:* There is a convention for the directions on a map; north is considered to be on the top.

On the other hand, software entities have no natural layout due to the lack of physical shape and location. Thus distance and orientation have no defined natural meaning for software entities. This is likely to be the cause for the big amount of different and incomparable visualization layouts. An informal shallow survey of recent publications at SOFTVIS and VISSOFT shows that most of their visualizations use an arbitrary layout. Most common the visualizations base their layout on the lexical order of elements, *e.g.* alphabetically or hash-key order[1].

Based on these perceptions from geographic maps we suggest that a consistent layout for software eases the comparison of visualizations that depict different information. As a next step we need to find a basis for the position of software entities on a so called "cartographic" software map. The requirements for that basis are that it must contain a semantically meaningful notion of position and distance that can be mapped to a two-dimensional visualization pane[2].

3.2 On the Choice of Vocabulary

In this section, we propose the usage of *vocabulary* as the analogue to physical position for software entities. We suggest to achieve a consistent, two-dimensional layout by mapping an n-dimensional vocabulary-based position down to two dimension. The distance between software entities is thus given by their lexical distance.

Why should we adopt vocabulary as distance metric, and not some structural property?

First of all, vocabulary can capture the key domain concepts of source code by abstracting away from its technical details [30]. Software entities with similar vocabulary are close by concept and topic. Another important point is, that vocabulary tends to be stable over time. It is known that the vocabulary of a software system is more stable than its structure [2] and that the vocabulary grows rather than changes over time [64].

[1] Hash-key order is what we get when iterating over a hash-based data structure *e.g.* Sets or Dictionaries

[2] Note that we do not require the representation to be two-dimensional initially. An n-dimensional representation that can be mapped down to two dimensions later is sufficient as well.

Consider, for example, programming languages where name overloading is applied. Even though overloaded methods differ in their implementation strategy, they will typically implement the same concept using the same vocabulary. Furthermore, lexical similarity has proven useful to detect high-level clones [38] and cross-cutting concerns [4] in software.

But what if programmers use the same name in a different context?

It is possible that semantically different scopes contain identical identifiers with different meanings. Consider, for example, two functions having mostly identifiers such as i, j, prev, next, end, stop, flag, ...; the one performs matrix computations, while the other is a hash-table implementation. Without the application of Latent Semantic Indexing (Subsection 3.3.1) the two would be classified as being very similar, but this is clearly not the case from a developer's perspective. Latent Semantic Indexing, however, can identify words that have different meaning depending on their context. LSI has the ability to resolve certain synonymy and polysemy [15].

What about operations during refactoring, that cause functionality to be renamed or moved?

Although refactoring causes code to be renamed or to be moved, the overall vocabulary of a software system tends not to change, except as a side-effect of a considerable growth of the project. Thus vocabulary remains relatively *stable* during changes. Because of this vocabulary can be used to provide a stable, consistent notion of position for software entities. Consider the example of a rename refactoring. Two effects may occur. In the first case, all occurrences of a symbol are replaced with a new symbol. This will not affect the map, since both lexical similarity and Latent Semantic Indexing are based on statistical analysis only. Replacing all occurrences of one term with a new term is, from the point of these technologies, a null operation. In the second case, some occurrences of a symbol are replaced with another symbol which is already used. This will indeed affect the layout of the map. Given that the new name was well chosen by the programmer, the new layout constitutes a better representation of the system. On the other hand, if the new name is a bad choice, the new layout is flawed. However, what constitutes bad naming is not merely a matter of taste. Approaches that combine vocabulary with structural information can indeed assess the quality of naming. Please refer to Høst's recent work on debugging method names for further reading [25].

As a consequence, vocabulary offers an understandable notion of position that can be used to provide a consistent layout for a system's software entities, even when given system is the subject of changes. The cartographic visualization presented in this work can also be used to show maps based on other notions of position, such as structural similarity. However, positions that are not based on vocabulary are likely to be less stable concerning changes in the software system. Hence other notions of position contradict our goal to help programmers establish a stable mental model of their software system.

In Section 3.1 we concluded that geographic positions can be easily mapped to a (two-dimensional) position and we required this to be the case for software entities in our

cartographic software visualization as well. In this section we elaborated that software entities, in fact contain an implicit notion of position and distance metric that arises from their vocabulary. We suggest to analyze the vocabulary of software entities and to determine their position by the term frequency in an n-dimensional space, where n represents the number of terms in the whole software project. Furthermore we suggest to apply Latent Semantic Indexing to handle synonymy and polysemy, this results in an m dimensional term-document matrix ($m << n$). As a final step, in analogy to geographic projections, we suggest to map these m dimensional positions down to two dimensions, while trying to preserve their relative distances as well as possible.

The second observation made about cartographic maps is that they contain a convention for the cardinal directions *i.e.* that across different maps the same set of locations are in the same place. Our cartographic visualization uses a distance based on lexical similarity, hence under the assumption that software entities dealing with the same domain contain similar vocabulary, they are located close to each other. The cardinal directions of cartographic maps thus is represented in software maps as the positioning of different software domains relative to each other.

3.3 The Cartography Pipeline

In this section we present the techniques that are used to achieve a consistent layout for software maps. The general approach of software cartography, as illustrated in Figure 3.1, is as follows:

1. We parse the vocabulary of source files into term-frequency histograms. All text found in raw source code is taken into account, including not only identifiers but also comments and literals.
2. We transform the term-frequency histograms using Latent Semantic Indexing (LSI) [15], an information retrieval technique that resolves synonymy and polysemy.
3. We use the composition of two algorithms, Isomap and Multidimensional Scaling, to map the the term-frequency histograms onto the 2D visualization pane. This preserves the lexical co-relation of source files as well as possible.
4. We use cartographic visualization techniques to render an aesthetically appealing landscape.

Each of the following subsections covers one step in above pipeline.

3.3.1 Lexical Similarity between Source Files

In this section, we explain how we extract term-frequency histograms from source files. We explain further, how the rank of the extracted term-frequency matrices is reduced,

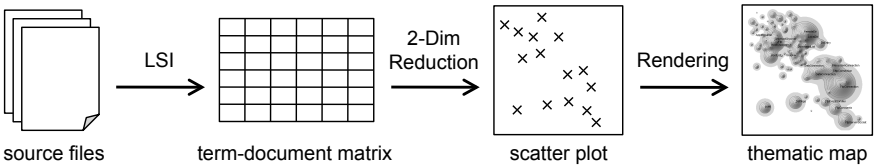

Figure 3.1: Software Cartography in a nutshell, from left to right: the raw text of source files is parsed and indexed using Latent Semantic Indexing. Then the high-dimensional term-document-matrix is mapped down to two dimensions using a composition of Isomap and Multidimensional Scaling, and finally cartographic visualization techniques are used to render the software map.

and how we resolve synonymy and polysemy.

As described in Section 3.2, the distance between software entities on the map is based on their lexical similarity. Lexical similarity is an Information Retrieval (IR) technique based on the vocabulary of text files. Formally, lexical similarity is defined as the cosine between the term frequency vectors of two text documents. That is, the more terms (*i.e.* identifiers names and operators, but also words in comments) two source files share, the closer they are on the map.

First, the raw source files are split into terms. Then a matrix is created, which lists for each document the occurrences of terms. Typically, the vocabulary of source code consists of 500–20'000 terms. In fact, studies have shown that the relation between term count and software size follows a power law [71]. For this work, we consider all text found in raw source files as terms. This includes class names, methods names, parameter names, local variables names, names of invoked methods, but also words found in comments and literal values. Identifiers are further preprocessed by splitting up the camel-case name convention which is predominantly used in Java source code. Note that since our approach is based on raw text, in theory any programming language that uses textual source files can be processed.

In a next step, Latent Semantic Indexing [15] is applied to reduce the rank of the term-document matrix to about 50 dimensions. LSI is able to resolve issues of synonymy and polysemy without the use of predefined dictionaries. This is advantageous for the vocabulary of source code which often deviates from common English usage. For more details on Latent Semantic Indexing and lexical similarity, please refer to Kuhn's previous work on software clustering [30].

3.3.2 Creating a Two-Dimensional Layout

In this section, we explain how the high-dimensional term-document matrix is mapped down to two dimensions.

In order to visualize the lexical similarity between software entities, we must find a

mapping that places source files (or classes, or packages, depending in our definition of a document) on the visualization pane. The placement should reflect the lexical similarity between source files.

We use the composition of two algorithms, Isomap and Multidimensional Scaling, to map the the term-frequency histograms onto the 2D visualization pane. This preserves the lexical co-relation of source files as well as possible.

Isomap [59], an algorithm similar to Multidimensional Scaling and Principal Component Analysis, maps the previously created multidimensional term-document matrix down to a two-dimensional layout. Given high-dimensional data, Isomap finds meaningful low-dimensional structures that are hidden in such a way that they might be invisible to Principal Component Analysis or Multidimensional Scaling. We call the result of Isomap "global layout" since it is used to assist Multidimensional Scaling with an initial configuration. That global layout is then refined further to a local layout using metric Multidimensional Scaling as described in Subsection 3.4.4. Please refer to Subsection 3.4.3 for a detailed explanation of Isomap and to Section 3.5 for a complete explanation of the layout algorithm used by *Codemap*.

3.3.3 Hill-shading and Contour Lines

This section explains the usage of cartographic visualization techniques to render a visually appealing map.

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

The hill-shading algorithm is well-known in geographic visualization. It adds hill shades to a map [55]. The algorithm works on a distinct height model (digital elevation model), where each pixel has an assigned z-value, its height, rather than on trigonometric data vectors.

The digital elevation model of *Codemap* is is a simple matrix with discrete height information for all pixels of the visualization plane. As illustrated on Figure 3.2, each element (ie source file of class) is represented by the 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 as illustrated in Figure 3.3.

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

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

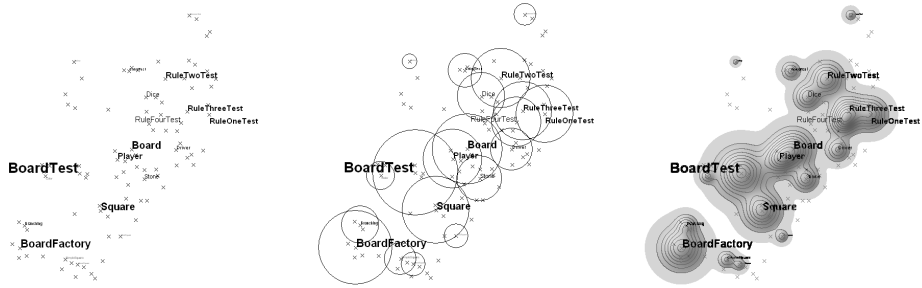

Figure 3.2: Construction steps: left) MDS placement of files on the visualization pane, middle) circles around each files location, based on class size in KLOC, right) digital elevation model with hill-shading and contour lines.

alone. Since almost all real world maps make use of contour lines, maps with contour lines are very familiar to the user.

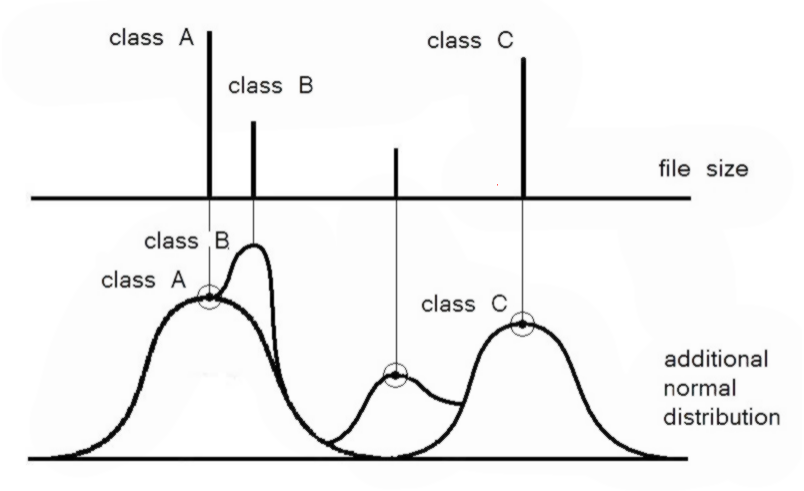

Figure 3.3: Digital elevation model: since classes might appear on the map very close to each other, we have to prevent larger classes from hiding smaller classes. Thus, the elevation model is built by summing up the volumes of all classes.

3.3.4 Labeling

This section introduces the labeling algorithm we use to annotate the landscape with textual information.

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: 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 [55]. 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 thesis show only the most important class names. *Codemap* 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.

3.3.5 Landscape Coloring and Overlays

This section presents two different ways in which software metrics can be displayed on top of the map.

Codemap provides two different positions to place the overlays. First we allow the landscape to be colored according to different metrics. This way of coloring enables the visualization of the data directly on the landscape. This is analogous to visualizations in geography where data like seismic activity, population size *etc.* can be shown as a cartogram. Figure A.5 gives an example of landscapes that are colored according to various metrics.

Secondly *Codemap* allows information overlays to be displayed on top of the generated landscape. Two examples of such visualizations are given in Figure 3.4, an overlay visualizing method calls and an overlay that shows search results. Since all overlays are displayed on top of the same layout, these overlays can be compared to each other. This might allow the user of *Codemap* to get a better understanding of the software since the co-location of data presented in two or more such an overlays might yield additional information.

3.4 On Different Layout Algorithms

This section enumerates different algorithms that can be used to map high-dimensional data onto a lower dimensional space. This is done as a preparation in order to better understand our final decision on the layout algorithm in Section 3.5.

During the development of *Codemap* we analyzed different algorithms to map our high-dimensional data down on a two-dimensional pane. Our final decision was to use a combination of two algorithms. The first step uses Isomap to compute a globally optimal layout. The second step uses Multidimensional Scaling, with the results of the first step as input, to compute a both globally and locally optimal layout.

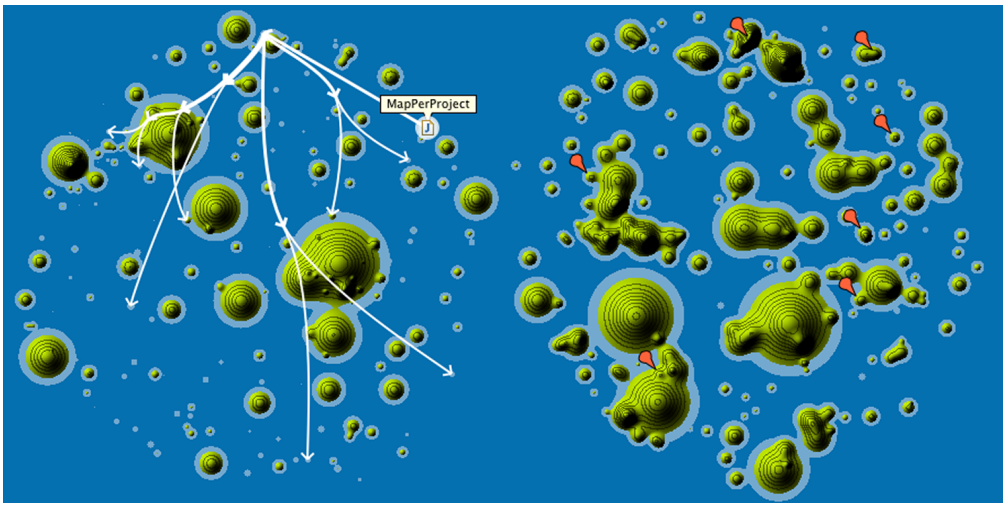

Figure 3.4: Two example overlays of *Codemap*. On the left is visualized which software components are called by a given method. The image on the right shows the results of a file search performed. Each result is marked with a pin.

The remainder of this section provides an overview of layout algorithms, and the next section (Section 3.5) motivates our choice of a combined algorithm. Subsection 3.4.1 first gives an introduction to Multidimensional Scaling. Subsection 3.4.2 discusses HiT-MDS, the MDS version we used in our previous work. Subsection 3.4.3 explains the Isomap algorithm and Subsection 3.4.4 discusses metric Multidimensional Scaling.

3.4.1 An Introduction to Multidimensional Scaling

This section gives an introduction to Multidimensional Scaling by showing its origin and enumerating different subtypes.

The term Multidimensional Scaling (MDS) is used to refer to a set of different statistical techniques that are all used to visualize proximities in low-dimensional space. The origins of Multidimensional Scaling lie in psychology where it helps to comprehend people's judgement on the members of an object-set. The first MDS method was proposed by Togerson [61]. His work is also the origin of the term Multidimensional Scaling. Nowadays Multidimensional Scaling is used across a variety of different fields [50] such as marketing, sociology, physics, political science and biology.

The term Multidimensional Scaling is generic and includes different subtypes. A distinction is made between [70]: Classical Multidimensional Scaling, Metric Multidimensional Scaling, and Nonmetric Multidimensional Scaling. Of the three variants, only classical and metric Multidimensional Scaling are of interest for us. However, classical Multidimensional Scaling is equivalent to Principal Component Analysis and

shall thus not be discussed separately as it is subsumed by the last step of the Isomap algorithm.

All Multidimensional Scaling variants have in common that they project elements from a high-dimensional space to a lower-dimensional space. The algorithms expect as input a matrix of pairwise dissimilarities between the elements, and return coordinates in the projection space for all elements.

3.4.2 High-Throughput Multidimensional Scaling

This section briefly explains HiT-MDS, the Multidimensional Scaling algorithm used by previous work, and explains why it has proven to be a suboptimal choice.

In previous work, we used High-Throughput MDS (HiT-MDS[3]) as layout algorithm [33, 31]. HiT-MDS is optimized for speed. It uses a heuristic to speed-up the computation time [57]. High-Throughput MDS was originally designed for clustering multi-parallel gene 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 fast computation is less accurate approximation and a simplified distance metric.

The heuristic HiT-MDS assumes that the input dissimilarities are *not* close to a constant value. Unfortunately, we found that our input data has exactly that property. What we observed was a "garbage in, structure out" effect. More recent versions of *Codemap* do not use HiT-MDS anymore [32]. For more details please refer to Section 3.5.

3.4.3 Isomap

This section introduces Isomap by Tenenbaum et al., which is an approach to reduce the dimensionality similar to Multidimensional Scaling and Principal Component Analysis [59].

Isomap finds meaningful low-dimensional structures that are hidden in high-dimensional data like global climate patterns or human gene information and might not be visible to Principal Component Analysis and Multidimensional Scaling. Figure 3.5 illustrates such a low-dimensional structure, the so called "Swiss roll"[4]. It represents a two-dimensional plane that is embedded in a three-dimensional space in spiral form. When measuring distances for that structure it is possible that the distance for two points is completely different, depending on whether one measures the distance in the three dimensional space or on the two-dimensional plane. In the figure, a dashed line indicates the euclidean distance, the solid line represents the distance measured by Isomap.

[3] http://dig.ipk-gatersleben.de/hitmds/hitmds.html

[4] Oddly this term is used in reference to a jelly roll that originates from Germany where it is known as *Biskuitrolle*.

Note that the distance measured by Isomap follows the two-dimensional, spiral plane of the data set.

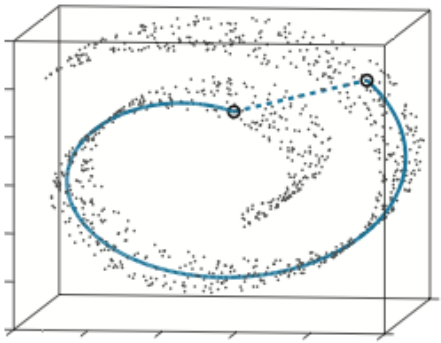

Figure 3.5: The "Swiss roll" data set illustrates how Isomap finds hidden low-dimensional structure in high-dimensional data. The dashed line indicates the euclidean distance whereas the solid line represents the distance measured by Isomap. Note that the distance measured by Isomap follows the two-dimensional, rolled plane of the data set. (Illustration taken from Tenenbaum et al. [59].)

Multidimensional Scaling and Principal Component Analysis, the classical approaches for dimensionality reductions, fail to detect structures like the "Swiss roll". Principal Component Analysis transforms your high-dimensional data into space of lower dimension and tries to best preserve the variance of the data in the high-dimensional space. Multidimensional Scaling transforms the high-dimensional data into a lower space and tries to preserve the distances between the points as measured in the higher dimensional space.

Isomap however first analyzes the data in the high dimensional space and tries to detect hidden structures. Isomap preserves the internal geometry of the lower dimensional structure during the dimensionality reduction. Isomap guarantees to recover this true dimensionality in the high-dimensional space even if it is highly folded. Since the documents in a TDM are on a unit sphere around the origin of the space, and thus embedded on a lower dimensional structure, Isomap is a very promising approach for *Codemap*'s dimensionality reduction.

To accomplish the reduction, Isomap performs the following three steps:

- Construct a neighborhood graph. Connect point m and n in the data set if n is among the the K nearest neighbors of m[5].
- Compute the shortest path distances for all pairs of points in the neighborhood graph.

[5] An alternative method would be to connect the point m to all points n that are closer than a given ϵ.

- Perform classical Multidimensional Scaling i.e. find the best configuration in the 2-dimensional space given the distances on the neighborhood graph.

The only free parameter needed by Isomap is *K*, the number of nearest neighbors to take into account for constructing the neighborhood graph.

3.4.4 Metric Multidimensional Scaling

This section introduces metric Multidimensional Scaling, an iterative approach to map high-dimensional data to a space of lower dimensionality.

The task of MDS is to map a set of points from a high-dimensional to a low-dimensional space by preserving their relative distances as good as possible. In our case, the high-dimensional space is the term-document matrix produced by Latent Semantic Indexing and the low-dimensional space is the two-dimensional visualization pane.

We refer to the set of points as configuration, we say that we are trying to find a good configuration for our classes on the two-dimensional pane. To measure the distances, MDS relies on dissimilarity values. Dissimilarity indicates how different two elements are. A low number means that two elements are similar, a hight value denotes their dissimilarity. A value of zero means that two elements are the same whereas the dissimilarity can go up infinitely.

MDS tries to find a two-dimensional configuration where the proportions between the dissimilarities are the same as in the space of higher dimension. Usually there is no such solution, therefore Multidimensional Scaling tries to find a solution that matches the dissimilarities as close as possible. To asses the quality of the two-dimensional result, a function called *Stress* is used. Stress calculates the badness-of-fit for the two-dimensional configuration. Clearly we want our result to have a small error, thus to find a good embedding of the higher dimensional points on our two-dimensional pane, we must minimize stress. In our case the least square method is used as an error estimate.

In practice this minimization is performed with an iterative approach. The configuration is refined with each pass, the algorithm stops as soon as a certain threshold is reached. The most popular method is stress majorization [14], which is also used by our implementation.

A function f which bounds another function g from above and touches g at one (or possibly more) points is called a majorizing function. Stress majorization as proposed by Leeuw [14] makes use of such majorizing functions to estimate the cost function while re-positioning the elements of a configuration.

Iterative majorization works as follows: suppose we have a point p and an initial guess for its coordinates. Now a majorizing function must touch the stress function at p's current location and must be located above it (or at most touch it again) at the other locations. Once a valid majorizing function for p is found, p s new estimate location is

set to the minimum of the majorizing function because the stress at that point is lower. In the next step, with a new majorizing function, the stress can be decreased again by moving the estimate location of p to the minimum of the new majorizing function. This process can be repeated iteratively until a satisfying result is found *i.e.* the stress is low enough. Since the algorithm is monotone it provides better results with each iteration.

The metric Multidimensional Scaling implementation used in *Codemap* is a manual port from C++ to *Java* of the implementation of GGobi/GGvis [10].

3.5 *Codemap's* Two-Dimensional Reduction

In this section we present the layout algorithm for *Codemap* that consists of two steps. In the first step, we apply Isomap to calculate a global layout that assists Multidimensional Scaling during the calculation of a local layout in second step.

The documents in a Term Document Matrix are on a unit sphere around the origin. That structure is invisible to either Principal Component Analysis or Multidimensional Scaling [59, 39], thus we use Isomap to calculate the global layout. The problem with the TDM structure is, that the more classes there are in a system, the higher the probability that in the term-document matrix produced by Latent Semantic Indexing two of them do not have any terms in common. Documents that do not have any terms in common have maximal pairwise distance. Using Multidimensional Scaling, this results in an arbitrary global layout since Multidimensional Scaling cannot meaningfully interpret input where many documents have maximal distance to each other. We call this unpleasant behavior of MDS the *simplex-problem* since parts of the output data form a simplex.

As described in Subsection 3.4.3 Isomap first analyzes the data in the high dimensional space and tries to detect hidden structures. Isomap preserves the internal geometry of the lower dimensional structure during the dimensionality reduction and guarantees to recover this true dimensionality in the high-dimensional space. Since the documents in a TDM are on a unit sphere around the origin of the space, and thus embedded on a lower dimensional structure, Isomap is a very promising approach for *Codemap*'s dimensionality reduction.

We use the term *global layout* to refer to a configuration that optimizes the global relations between the locations. In a good global layout the locations containing similar vocabulary are positioned closer to each other than locations with dissimilar vocabulary. Isomap calculates a good global layout but it tends to form clusters of locations. Furthermore Isomap tends to produce outliers that are responsible for the clustering since they force the rest of the visualization to be displayed in a smaller space. The global configuration calculated by Isomap is mathematically meaningful but we need to perform one more step to achieve a better usage of the visualization pane.

For this second step we use Multidimensional Scaling which is not suitable to calculate a meaningful global layout but has proven to be useful to calculate a visually appealing *local layout*. We use Isomap's output as initial configuration for Multidimensional Scaling since once a valid global solution is found it is possible to apply Multidimensional Scaling without running into the simplex problem. To the Multidimensional Scaling algorithm any global solution is valid input. When provided with the Isomap layout as an initial configuration, Multidimensional Scaling will only change the positions on a local scale.

Figure 3.6 compares two maps of the same project. The layout on the left uses Isomap only. The Layout on the right uses Isomap and performs an additional Multidimensional Scaling to make better usage of the visualization pane.

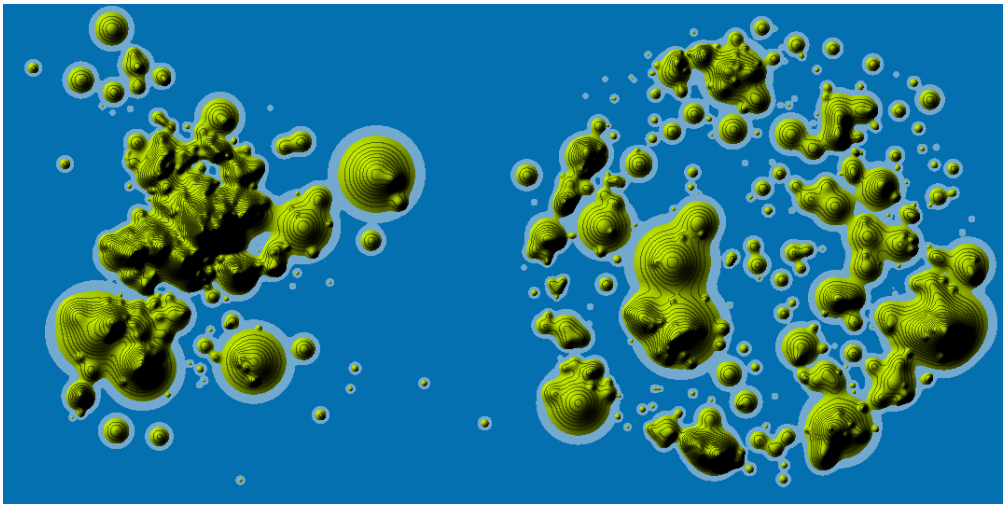

Figure 3.6: A comparison of two maps of the open-source project Vuze (formerly Azureus)[6]. On the left is a configuration using only Isomap, on the right is a configuration using Isomap to calculate the global layout and Multidimensional Scaling to calculate the local layout.

Figure 3.7 gives an overview of the rendering pipeline used by *Codemap*. Given the Term Document Matrix (TDM), *Codemap* calculates a global layout using Isomap. Then it applies Multidimensional Scaling using the global layout as initial configuration and the TDM for the proximity values. This calculates the local layout resulting in the final configuration.

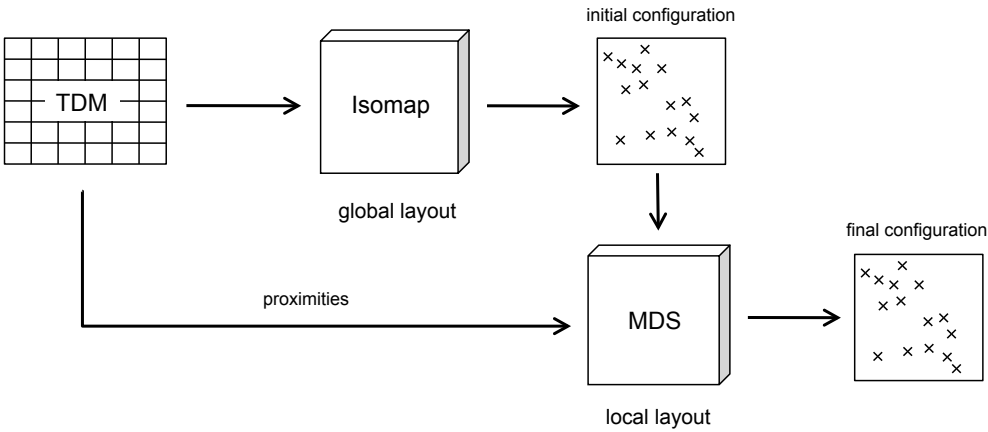

Figure 3.7: Given the Term Document Matrix (TDM), *Codemap* calculates a global layout using Isomap. Then it applies Multidimensional Scaling using the global layout as initial configuration and the TDM for the proximity values to calculate the local layout resulting in the final configuration.

Chapter 4

Codemap

In this chapter we describe the motivation to integrate *Codemap* into a development environment. Furthermore we elaborate in what ways *Codemap* aims at supporting the programmer by addressing his mental model of software during development.

As mentioned before, *Codemap* can be considered the successor of *Software Cartographer* as it improves the layout algorithm and switches the focus from analysis to IDE integration. We aim at supporting the developer in various daily development tasks by providing a stable cartographic visualization of his software project. Therefore we integrate software cartography into the *Eclipse* IDE so that a map of the project may always be visible while working with the code.

Section 4.1 discusses why we chose *Eclipse* as the target platform for our plugin. Section 4.2 discusses the programming tasks supported by *Codemap* and Section 4.3 discusses a set of features that have not yet been implemented but seem promising.

4.1 On the Choice of *Eclipse* and *Java*

In this section we discuss our decision to implement *Codemap* as an *Eclipse* plug-in.

Eclipse is an extensible platform that allows tool integration and is used by millions of developers worldwide [46]. Eclipse is used in a wide range of research projects all around the globe[1]. This large user base is especially valuable when we want to find early adopters for *Codemap* who provide us with early feedback. Furthermore when performing an evaluation of *Codemap* it is easier to find industry developers already familiar with *Eclipse*.

[1] http://wiki.eclipse.org/Eclipse_Research_Community

The *Eclipse* Java Development Tools (JDT) project provides the *Java* specific *Eclipse* plug-ins. It adds useful functionality like writing and navigating *Java* source code, refactoring, code formatting, debugging and many more. Since all these tools are already available they provide a good base for an IDE-extension like *Codemap*.

4.2 Supported Programming Tasks

The goal of *Codemap* is to provide developers with a shared, spatial and stable mental model of software projects. To achieve this goal a cartographic visualization is embedded in the IDE. This visualization is most useful when it supports as many development tasks as possible. In this section we present a concrete selection of development tasks that are supported by *Codemap*:

Codemap improves navigation within a software system since it is integrated with the package explorer and editor of *Eclipse*. The selection of *Codemap* is linked to *Eclipse*'s selection and vice versa.

Codemap is hooked into several *Eclipse* plugins such as *Eclemma* to display their results on the map. This enables comparison of different software metrics to each other, *e.g.* comparing error density to code coverage.

Codemap supports exploration of systems during reverse engineering as it is integrated with *Eclipse*'s structural navigation functions such as search for callers, implementers, and references. We display pins for search results and arrows to represent call hierarchies.

Codemap supports social awareness of collaboration in the development team by connecting two *Eclipse* instances and showing open files of other developers.

The following subsections give a feature centric overview of how *Codemap* supports these development tasks. Each of the following subsections describes the feature, the programming task and elaborates why the task is important and how it is supported by the presented feature. Furthermore it presents how support for the given task is added to *Codemap* and finally states how *Eclipse* supports the task without *Codemap*. Please also refer to Section 4.3 for a description of the features that have not yet been implemented.

4.2.1 Code Navigation

To help the developer navigate within a software system, *Codemap* is linked to *Eclipse*'s navigation functionality. It is integrated into *Eclipse*'s package explorer and the source code editor. Furthermore the selection of *Codemap* is linked to *Eclipse*'s selection and vice versa. This section discusses in detail how *Codemap* improves the navigation within *Eclipse*.

Developers navigate code, which means they browse, scan, search and seek code within their current domain of interest, usually using tools that support them in their intents. Navigating comes from the latin word *navigare* which means "sailing", in other words to bring a ship from a source position (where it is now) to its target position. Usually the target position is well known, the sailor just needs to find the best path. In our modern context, navigation means to find an object or location within a software system. Thus in this section, we are particularly interested in the way developers navigate through the source code to find an already known object.

Eclipse currently provides several navigation aides:

- Package Explorer/Project Explorer provides a tree-based navigation structure, similar to many file managers.
- Open Type/Open Resource allows one to open any resource within *Eclipse*'s scope by typing the file-name/object name.
- Editor Tabs, for each file loaded into the editor displays a tab (given that there is enough space left). This allows quick access to the files the developer is currently working on.
- Outline offers a structural overview of the file which is currently active in the editor.
- Call Hierarchy displays a hierarchical view of the calls from/to the element selected java member (see Subsection 4.2.5).
- Type Hierarchy displays subtypes and/or supertypes of a given type.
- Navigation History allows the user to navigate the previously opened files.

Unfortunately the number of files per project becomes more and more unmanageable as the projects grow in complexity over time. As a matter of fact, DeLine's work is based on the observation that developers often are lost in code while navigating [18]. *Codemap* tries to address this problem by providing a visualization that eases code-navigation. According to DeLine a visualization should contain visual landmarks to ease the developer's navigation within a system [16]. *Codemap* fulfills this requirement as the algorithms used to calculate the layout and the elevation model generate an unique landscape whose hills can serve as landmarks. Furthermore, *Codemap* is linked to the *Eclipse* editor. It continuously updates a label indicating the file currently active in the editor and also marks the open files on the map.

As a further navigation aid, it is possible to highlight recently visited classes. The highlighting is achieved by changing the coloring-metrics of the landscape to display a heat map where hot means that the file has been visited recently. *Codemap* builds a trace of the files recently visited and changes the color of the landscape. While visualizing, this trace is slowly faded from yellow (hot) to a darker color (cold) towards the end. In Figure 4.1 you can see a heat-map used to trace the locations visited last. Furthermore note that the currently active file is indicated with a text-label containing the name of the class with the open files being marked with an icon representing their type.

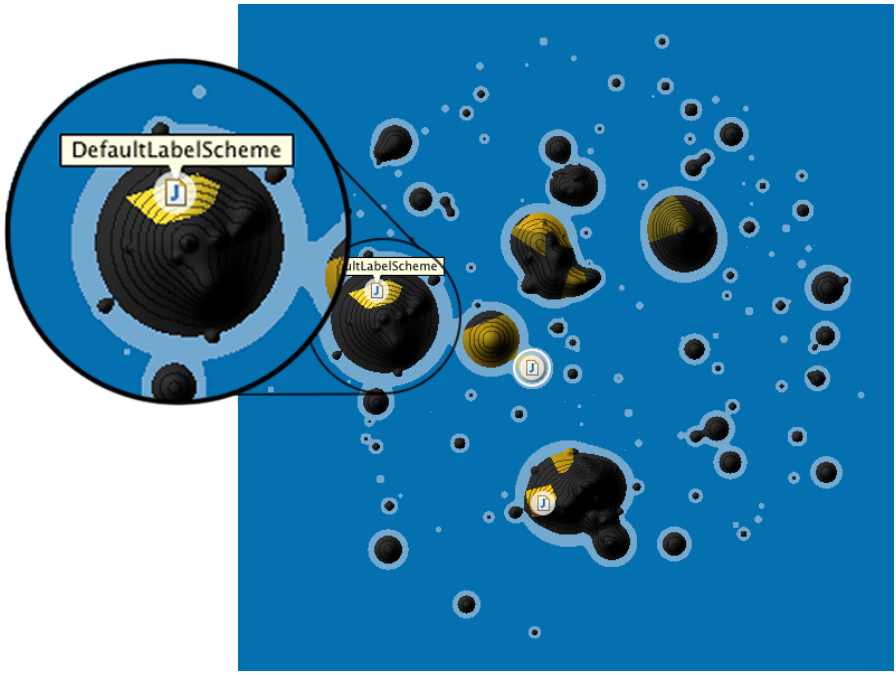

Figure 4.1: *Codemap* uses various techniques to ease code-navigation. In this figure you can see a heat-map used to trace the locations visited last. The currently active file is indicated with a text-label containing the name of the class and the open files are marked with an icon representing their type.

4.2.2 Test Coverage

To support software testing, which is crucial to software development, *Codemap* is linked to the coverage metrics provided by the *Eclipse* plug-in *EclEmma*. This section elaborates how *Codemap* aims to support testing.

Testing is crucial to software development and thus widely accepted as best practice. Test coverage is one of the measures used in software development, reporting to which degree the tests cover the code of a program. Miller and Maloney introduced test-coverage in 1963 when they explained that for the development team to know if a section of code executes correctly, this section must be executed by at least one test [41]. Regardless of the mixed results regarding the relationship between high test coverage and better software reliability [9], code coverage has been incorporated as a predictor for software quality [13].

Code coverage metrics for *Eclipse* are provided by the plug-in *EclEmma*[2], which itself is

[2] http://www.eclemma.org/

based on *EMMA*. *EMMA*[3] is a free code coverage toolkit to measure Java code coverage. EclEmma displays code-coverage results as sortable list as illustrated in Figure 4.2. Visual feedback on the coverage quality is available as a percentage and as a slide bar.

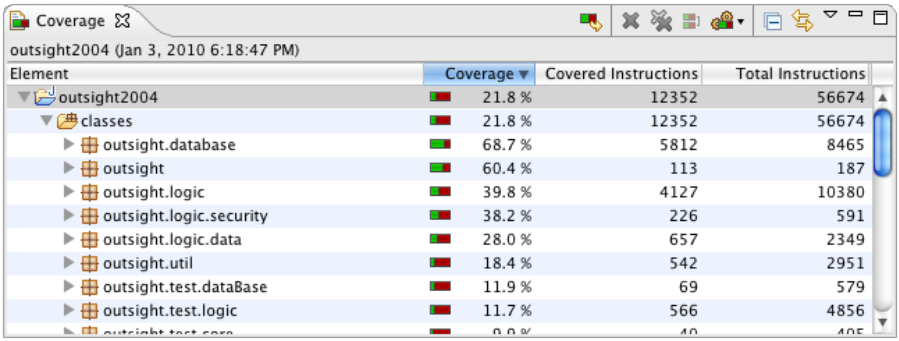

Figure 4.2: EclEmma displays coverage results as a sortable list. Visual feedback on the coverage quality is available as a percentage and as a slide bar.

Since testing is crucial and test coverage is a widely adopted metric we integrate support to display test coverage into *Codemap*. We display the coverage metrics directly on the island surface as illustrated in Figure 4.3 to leave place for other overlays and text-labels. We added coverage metrics based on the fact that visual information is processed much faster than textual information. Thus the programmer is provided with a better global impression of the test coverage of his software system.

4.2.3 Searching the Code

To present an overview of their distribution in a software system, *Codemap* visualizes search results by displaying pins on the map. This section explains in detail how the *Eclipse* search tools are enriched by *Codemap*.

Searching for text within code a common task performed quite often while programming [3]. Be it searching for code snippets online or searching code locally. Local searches might be performed while exploring an existing software system, during refactoring, and so on.

Eclipse provides different search plugins, ranging from generic text search to language specific search mechanisms. The search results are presented as a tree or a list.

Codemap contributes to *Eclipse*'s search tools by visualizing the search results on the map, as illustrated in Figure 4.4. This gives a brief overview of all the search results at once. This gives a better overall impression of the distribution of the results than the list provided by *Eclipse*. It can, for example, show in which part (in reference to

[3] http://emma.sourceforge.net/

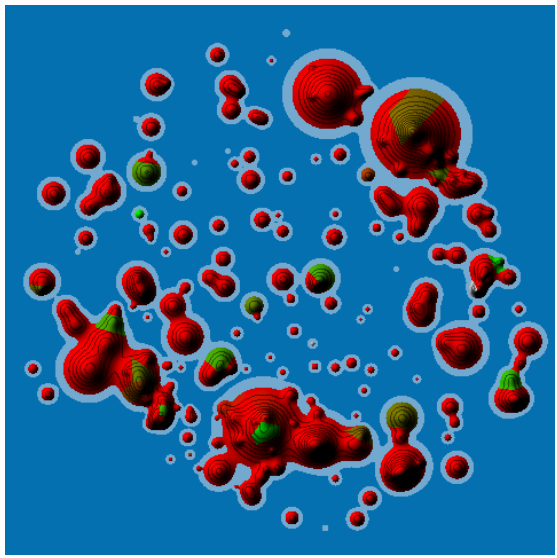

Figure 4.3: *Codemap*'s coverage metrics are displayed directly on the hills to leave place for other overlays and text-labels.

vocabulary) of the software the term searched for occurs the most. This visualizes how the search results are distributed over a software project.

4.2.4 Searching References/Declarations

Searching for references and declarations is a customized search provided by *Eclipse*. This section states how these searches are visualized by *Codemap*.

When programmers explore a system during reverse engineering they can rely on utilities that navigate the system's structure. We link *Codemap* to *Eclipse*'s specialized search utilities that can find callers and implementers of given functionality.

While browsing code, the programmer is interested where something has been defined or from where it is referenced. *Eclipse* provides utilities to search for references to an identifier or declarations of that identifier in different scopes, among others in project scope. This feature can be seen as a customized, language specific search, so the description of Subsection 4.2.3 applies here as well. The results of a domain-specific references/search are displayed the same way as the normal search results, as a list. Thus, *Codemap* visualizes these results as shown in Figure 4.4, the same way it visualizes the normal search results.

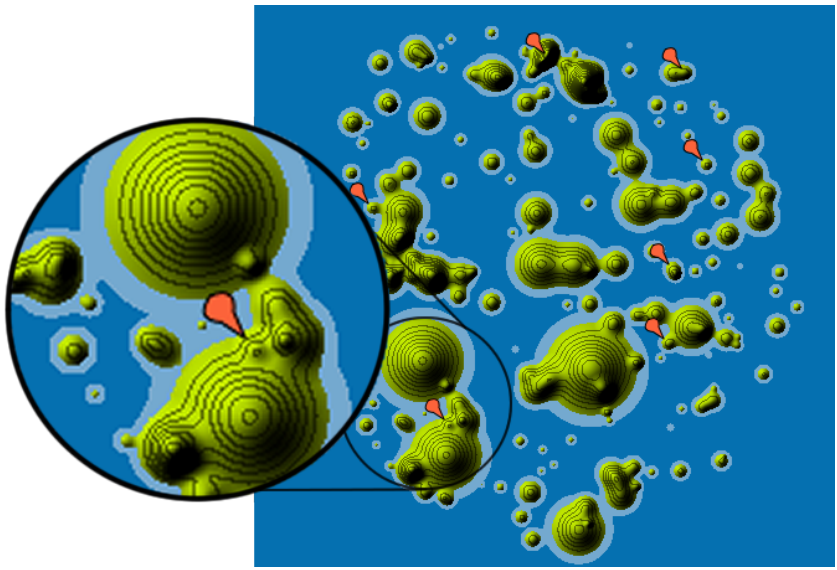

Figure 4.4: *Codemap* extends *Eclipse*'s search engine by displaying search results on the map using a google™style pin.

4.2.5 Browsing Call Hierarchies

Browsing for call hierarchies is a useful operation, especially for reverse engineering. This section explains how call hierarchies are visualized on top of the map.

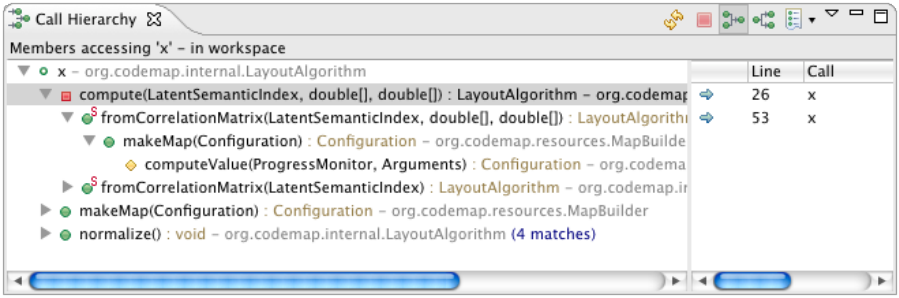

Figure 4.5: *Eclipse* displays the call hierarchy as tree that can be expanded and collapsed to show nested call hierarchies.

The call hierarchy enables the programmer to show calls to or from a method, constructor or field. This helps one understand the flow of the code and the complexity of method chains. This allows the programmer to check several code-levels and to explore many possible execution paths. *Eclipse* supports call-hierarchy exploration by

providing an entry "Open Call Hierarchy" in the right-click menu. Once clicked the view as illustrated in Figure 4.5 appears. The tree displayed in the call hierarchy view can be expanded and collapsed to show nested calls.

Codemap enriches these call hierarchies by showing an arrow based overlay (see Figure 4.6). For each node that is expanded in the call hierarchy view an arrow based graph is added representing the calls of that node. This additional visualization helps the programmer to understand which domain of the code calls which other domain. The traces are not visualized as straight arrows, but using automatically generated flow maps based on hierarchical clustering [44] to avoid visual clutter. These arrows can be interpreted as roads or shipping routes connecting the islands. For a detailed description of the algorithm please refer to Section 5.1.2.

Note that 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.

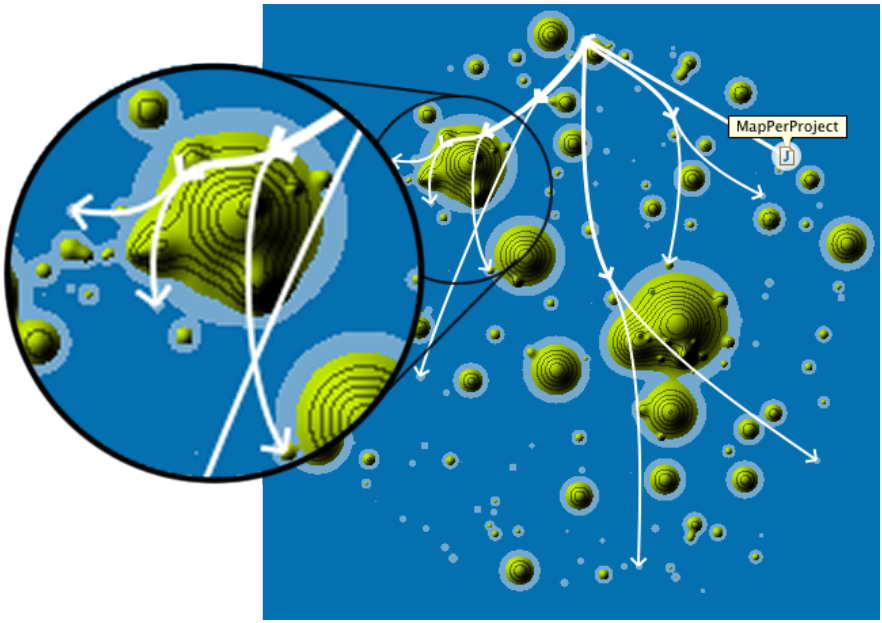

Figure 4.6: *Codemap* enriches call hierarchies shown by *Eclipse* by displaying an arrow-based overlay. For each expanded location in the *Eclipse* call hierarchy view, arrows representing the calls are shown.

4.2.6 Collaboration

In this section, we analyze how *Codemap* enriches collaboration by supporting collaborative awareness in development teams.

Collaboration is one of the most important activities in software engineering since software engineering projects are inherently cooperative. To produce a larger software system, it requires many engineers to coordinate their efforts [68]. Awareness of individual and group activities is critical to successful collaboration [20]. We propose to ease the collaboration by adding a feature to *Codemap* that supports the awareness of collaboration in the development team.

The Eclipse Communication Framework (ECF)[4] supports development of distributed *Eclipse* applications. ECF provides the library code needed to create distributed plugins easily as well as some example plugins. These examples include shared editing and instant messaging.

Building on ECF, we integrate collaboration support into *Codemap*. We chose to display which files are currently edited by a peer, once the collaboration feature is enabled. Collaboration is supported on top of popular protocols compatible to the Extensible Messaging and Presence Protocol (XMPP) or similar. As depicted in Figure 4.7 once sharing is enabled, a meeple is displayed for each file opened by remote collaborators.

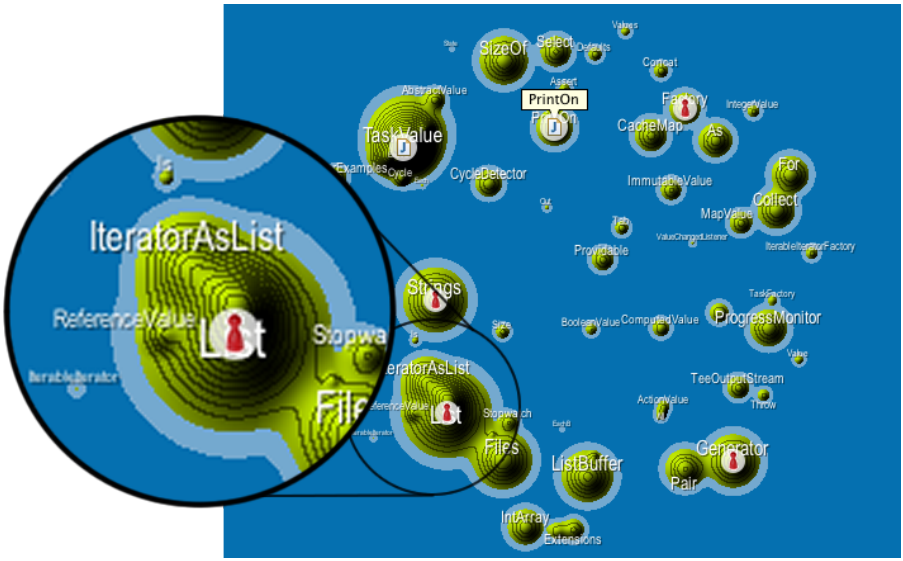

Figure 4.7: *Codemap* builds on the Eclipse Communication Framework, adding a collaborative overview. For each file that is opened by the remote collaborators, a meeple is displayed at that location.

[4] http://www.eclipse.org/ecf/

4.3 Future Features

In this section we discuss future extensions that can be built for *Codemap* based on or in addition to the current work. Some of the features listed below have not yet been implemented due to limited time while others arose from questions or feedback during presentations given or as a result of the interviews performed during the user study as described in Section 6.1. In the following subsections we already propose some solutions to issues or questions raised during the user study.

4.3.1 Vocabulary View

Since *Codemap* bases the proximity values of locations on the map on vocabulary it is interesting to make the vocabulary used visible to the user. To achieve this we can add an additional view to the *Eclipse* plugin, showing the vocabulary used by *Codemap*. The vocabulary can be visualized using different algorithms, for example as a tag-cloud or an ordinary sortable list. Furthermore the vocabulary view can be linked to the currently selected items on the map or even listen to the selection provided by *Eclipse*. This would allow the programmer to get a deeper understanding of the way the different domains appear on the map. A possibility to extend the vocabulary context even further is to provide search or filter mechanisms based on the vocabulary used for the layout, see Subsection 4.3.2.

4.3.2 Vocabulary Search

As described in Subsection 4.3.1 the fact that *Codemap*'s layout is based on vocabulary can be made more visible by providing utilities that base their operations on that vocabulary. One interesting feature would be to extend *Codemap*'s current search-bar and to provide additional search tools that search only within the vocabulary the visualization is based on instead of searching the full source-code. This search functionality should be coupled to the Vocabulary View as described in Subsection 4.3.1. This could help the developer to further understand the spatial distribution of the system's domains on the map.

4.3.3 Further *Eclipse* Integration

Codemap aims at providing the developer with a stable mental model of his software project by displaying a cartographic visualization in his IDE. We suppose that this visualization's usefulness grows with the development tasks it supports and present a brief overview of further features offered by *Eclipse* that can be extended by *Codemap*.

Type Hierarchy

Eclipse provides a tool that shows the hierarchy tree of a given type as depicted in Figure 4.8. This tree displays subtypes and/or supertypes of that type. *Codemap* can be extended by a feature that displays an overlay representing this hierarchy. The functionality can be similar to the call-hierarchy plugin already implemented (see Subsection 4.2.5) and visualization can look similar to Figure 4.6.

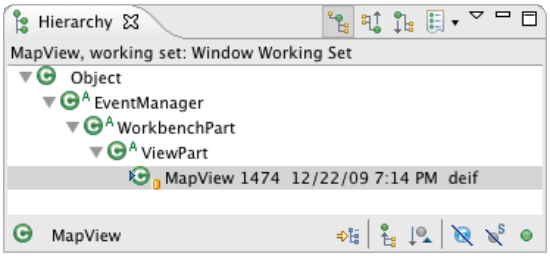

Figure 4.8: *Eclipse* provides a tool that shows the hierarchy tree of a given type, displaying subtypes and/or supertypes.

Debugger Integration

Since all the navigation problems occur during debugging as well, *Codemap* is available in the debug perspective, too. However, so far there are no debug-specific features implemented. Two possible extension are to show the call stack on the map to improve the developers' orientation during debugging or to display the breakpoints on the map since they represent the developers current points of interest and thus act as landmarks.

Bookmarks

Eclipse provides bookmarks that let the developer mark important locations in his code and get back to them quickly. The bookmarks view as depicted in Figure 4.9 then displays all these bookmarks as a list. Since the bookmarks represent the current locations of focus of the developer, we suggest *Codemap* to display them as well.

Profiling

TPTP[5] is the official profiling plugin available for *Eclipse*. TPTP allows profiling of *Java* applications, including JUnit tests and web applications. The aspects that can be traced

[5] http://www.eclipse.org/tptp/

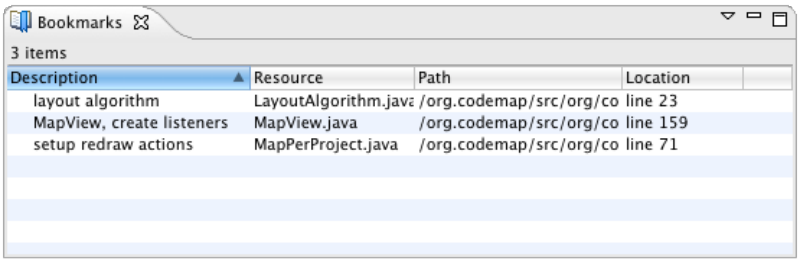

Figure 4.9: *Eclipse* allows the developer to set bookmarks in his code to be able to quickly get back to these locations later.

include execution tracing (where the application spent time) and memory tracing (how many objects are created).

Codemap can be extended with support for profiling by displaying profiling information on top of the map, for example as a heat-map or by replacing the height-information with the profiling information.

Revision Control Systems

As soon as a team of multiple people is working on the same project, the use of a revision control system is inevitable. *Codemap* can display information regarding revision control systems like local modifications, files that need updating and so on.

Code ownership is defined as an approximation of the percentage of lines owned by an author in a given revision [51]. Seeberger suggests that, since software systems change over time, the knowledgeable developer for a file changes as well [52]. Seeberger presents Chronia, a tool that implements an ownership map visualization to understand when and how different developers interacted in which way and in which part of the system [52]. *Codemap* can display code ownership information as well which is especially interesting in combination with other revision control based overlays.

4.3.4 Labeling Schemes

Labeling of locations as described in Subsection 3.3.4 improves greatly the usability of a map. Currently, *Codemap* labels all entities with their name *i.e.* the class or file name. Optionally, labels can be disabled completely. These are not the only two meaningful label schemes for *Codemap*. Another possibility for the labels is be to generate them automatically, based on the vocabulary used in the underlying entities. Kuhn proposes an automatic way to label and compare software components that uses the log-likelihood ratios of word frequencies [29]. This can help the developers to quickly understand the domain of the different software entities on the map.

4.3.5 Elevation Schemes

When generating the elevation scheme as described in Subsection 3.3.3, we take information from all the entities on the map into account. For each element, a hill corresponding to the element's lines of code is generated. The generated hills are summed up and thus a landscape emerges.

Of course, the height of a location can represent different information. To improve navigation, it can take the place of a heat-map, where recently visited locations are higher and locations not visited at all are merely above ocean level. Or it can be coupled to the current selection, elevating only these entities that are under selection.

Another interesting concept is to add negative elevations, where certain entities are represented by holes in the ground, reducing the height of the landscape instead of increasing it. This requires a meaningful definition of a hole *i.e.* we need to say which software entities cause holes and more importantly how a hole can be interpreted.

4.3.6 Finer Granularity

The smallest software entities that are directly represented in *Codemap* are classes, or more precisely source files. Even for contributions that are linked to methods, the visualization layer that represents these contributions only displays its elements in relation to the hills on the map. Since these hills represent classes, there is so far no meaningful representation for visualizations of finer granularity that take methods or attributes into account.

We envision an extension for *Codemap* that finds a meaningful way to visualize methods and attributes in the landscape. An idea that further develops the landscape metaphor is to display buildings on the map as well. Their position can be determined by the vocabulary of the method body in relation to the full vocabulary of the method's class or by the attribute name. Properties like the number of parameters or the length of the method can influence the shape of these buildings. Furthermore method calls can then be interpreted as roads connecting different buildings.

4.3.7 Map Wizard

In the current implementation, maps are directly linked to *Eclipse* projects. For the reason of simplicity, *Codemap* is linked to the selection provided by *Eclipse*. As soon as a new project is selected, the map representing that project is displayed in the visualization, or a new map is automatically generated if the project has not been selected before. Thus the user interface does not provide a possibility to generate a customized map, even though this is in no way restricted by the underlying software cartography algorithm.

We propose the implementation of a tool that enables the creation of customized maps, a map wizard. Customization can include the files that are displayed on the map and it can be possible to have one bigger map visualizing multiple projects at once. One can think of one big project that has been split into a library project and an application project using that library. Here, the developer might want to have a common visualization for both of these projects. Of course this is not restricted to two projects but can include an arbitrary number of projects.

Chapter 5

Implementation

This chapter dives into the implementation of *Codemap*. In Section 5.1 we have a look at the model of *Codemap* and how we integrate it into *Eclipse* from a technical perspective. Section 5.2 discusses the performance optimizations that reduced *Codemap's* calculation time drastically.

5.1 Architecture

This chapter describes the internal structure of *Codemap* and its three main architectural parts.

A simplified UML overview of *Codemap*'s internal structure can be seen in Figure 5.1. *Codemap* can be split in three main parts, as colored in Figure 5.1: A concurrent calculation model (red), the layered visualization (blue) and the view and interface to *Eclipse* (yellow).

Each of the following subsections covers one of the core *Codemap* components. Subsection 5.1.1 explains the small framework we wrote to model the dependencies in the calculation pipeline. Subsection 5.1.2 explains the layered visualization architecture and finally Subsection 5.1.3 describes *Codemap*'s two main point cuts to *Eclipse*.

5.1.1 Concurrent Calculation Pipeline

This section describes the underlying calculation model of *Codemap*. The calculation pipeline part is the heart of *Codemap*. Its input are *Java* source files from which it calculates a two-dimensional configuration and a cartographic visualization.

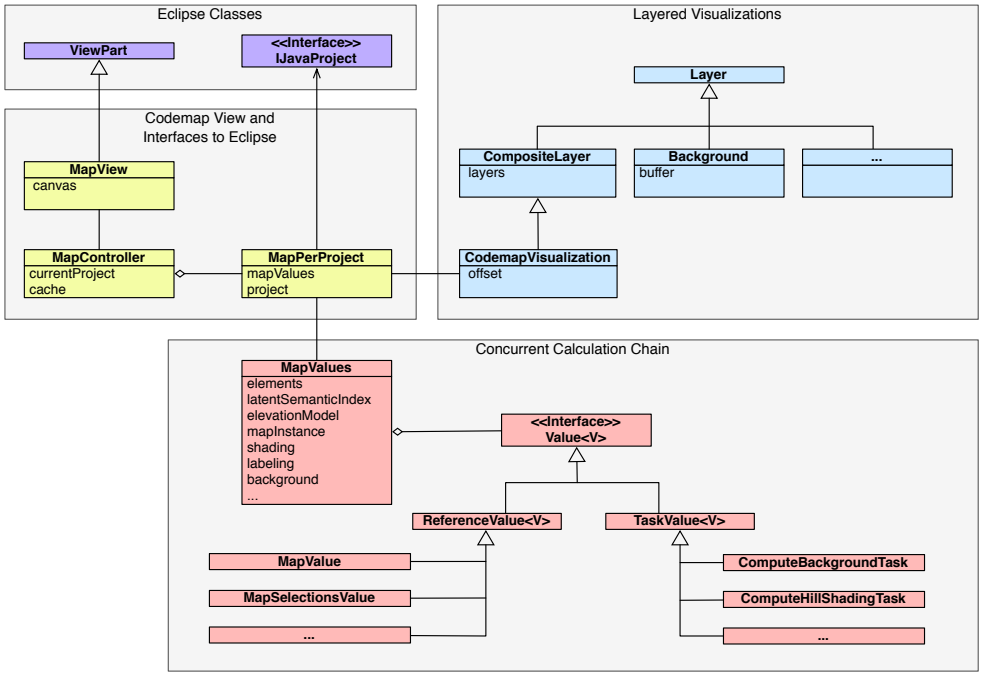

Figure 5.1: A simplified overview of *Codemap* as an UML class diagram. TOOL can be split in three main parts, as colored in the Legend, a concurrent calculation model (red), the layered visualization (blue) and the view and interfaces to *Eclipse* (yellow). Classes that originate from *Eclipse* are colored violet.

To generate a map of a project's code, many small steps need to be performed. A brief overview of the calculation pipeline is given in Figure 3.1. The overview is kept simple on purpose, since the underlying dependencies are far more complicated. Events from the user interface trigger recalculations at different steps in the pipeline. For example, when the size of the map changes, it must be rendered again, but the elements (classes) displayed on the map remain the same. Hence the vocabulary of the map remains untouched and the layout algorithm must not be re-run. However, if new files are added to a project, the whole map needs to be recalculated. To take this set of dependencies into account, a small framework was created. This section introduces the framework used to model these dependencies, but does not enumerate all the dependencies.

The underlying idea of the framework is to model dependencies as a dependency graph. Each node contains a value which might change. Upon changes, each node can fire events that trigger recalculations in the following nodes in the graph. Listing 5.1 shows the class *AbstractValue* which handles these dependencies and is responsible for triggering the notifications upon changes.

Listing 5.1: The AbstractValue class can trigger notifications upon changes.

```
public abstract class AbstractValue<V> implements Value<V> {

   private Collection<ValueChangedListener> listeners =
         new ConcurrentLinkedQueue<ValueChangedListener>();

   protected final void changed() {
      EventObject event = new EventObject(this);
      for (ValueChangedListener each: listeners) {
         each.valueChanged(event);
      }
   }

   public final void addDependent(ValueChangedListener listener) {
      listeners.add(listener);
   }

   public final void removeDependent(ValueChangedListener listener) {
      listeners.remove(listener);
   }
   ...
}
```

As illustrated in Figure 5.1 the values stored in each node of the dependency graph can either be simple references (*e.g.* an *Integer* representing the size the map) or they can be the result of a calculation (*e.g.* Multidimensional Scaling). Note that nodes holding primitive types have been omitted in the UML diagram since they are conceptually the same as nodes holding references.

Listing 5.2 shows some lines of the *TaskValue* class. A *TaskValue* instance adds itself as a dependent to each *Value* passed as constructor argument. Thus it receives change events from all these values. Since a *TaskValue* can depend on more than one value, it is required to delay the recalculation until all these values are available. Once all dependencies are met the calculation of the value can start. But that calculation can take some time, thus a *TaskValue* has to delay the update sent to its dependents until the calculation is over. To meet these requirements, each *TaskValue* has an internal state which can have one of the following values:

- Missing: indicates that the value is missing since the dependencies have changed. A recalculation will be triggered upon the next request of the value.
- Waiting: indicates that the we are waiting for all dependencies to be met.
- Working: indicates that the new value is being calculated.
- Done: indicates that the new value has been calculated, and the dependencies have not changed since.

Listing 5.2: A *TaskValue* instance adds itself as a dependent to each *Value* passed as constructor argument. The *computeValue* method is called as soon as we enter the working state.

```
public abstract class TaskValue<V> extends AbstractValue<V>
    implements ValueChangedListener {

   public TaskValue(String name, Value<?>... parts) {
      this.name = name;
      this.parts = parts;
      this.requiresAllArguments = true;
      for (Value<?> each: parts) each.addDependent(this);
      ...
   }

   protected abstract V computeValue(ProgressMonitor monitor,
       Arguments arguments);
   ...
}
```

The *Value* API provides us with the tools needed to build the dependency graph for *Codemap*'s calculation pipeline. Listing 5.3 shows a snippet of how the pipeline is put together.

Listing 5.3: Defining the dependencies between the different calculation steps

```
   public MapValues(MapValueBuilder make) {
      IntegerValue mapSize = new IntegerValue(INITIAL_SIZE);
      Value<MapScheme<Boolean>> hills = new
          ReferenceValue<MapScheme<Boolean>>();

      Value<LatentSemanticIndex> index = new
          ComputeEclipseIndexTask(elements);
      Value<Configuration> configuration = new
          ComputeConfigurationTask(index);
      Value<MapInstance> mapInstance =
          ComputeMapInstanceTask(mapSize, index, configuration);
      Value<DigitalElevationModel> elevationModel = new
          ComputeElevationModelTask(mapInstance, hills);
      ...
   }
```

5.1.2 Visualization Layers

Codemap's visualization architecture is split into different layers. This section gives a brief description of this architectural paradigm we used.

Since *Codemap* needs to support drawing of an arbitrary number of overlays, these overlays have been organized into layers. This allows them to be easily enabled/disabled since every layer is responsible for exactly one visualization. Furthermore moving the layer closer to the background or moving it towards the foreground is easy. Depth changes can be important when the content of one layer shadows other content.

Codemap's layers are implemented using the composite pattern and can thus be nested arbitrarily. User events like mouse moves, mouse clicks, mouse drags and keystrokes are propagated downwards the layer stack. Layers closer to the top receive these events earlier and thus can handle or even cancel them before they reach layers further down the stack. This is useful *e.g.* when multiple layers define behavior to handle mouse-clicks, but only the topmost of these layers should be allowed to execute its behavior.

Flow-Map Algorithm

One interesting layer built into *Codemap* is the call-hierarchy analysis layer. It displays a flow-chart based overlay as used by cartographers to show the movement of objects from one location to another [44]. This section gives a brief overview of the algorithm that calculates these visualizations.

As mentioned in Subsection 4.2.5 the trace overlay does not visualize the connections as straight arrows but groups arrows going in the same direction as depicted in Figure 4.6 to avoid visual clutter. This is achieved by making use of a flow map algorithm that groups the arrows by hierarchical clustering [44].

To calculate its layout the algorithm performs the following steps before the graph is rendered:

- Layout adjustment sorts all nodes according to their coordinates.
- Primary clustering uses the sorted nodes to find out about the spatial distribution of the nodes.
- Rooted clustering uses the previously calculated primary clustering to generate a flow-map with a given node r at the root. Furthermore it computes a weight value for each cluster/node.
- Spatial layout calculates the position of the branching nodes. The position is based on the weight of the child-clusters/nodes.
- Edge routing avoids intersections of edges by routing them around the bounding boxes of the same clusters.

5.1.3 Eclipse Integration

This section describes how *Codemap* extends *Eclipse i.e.* how we gather source-file information using the interfaces provided by the *Eclipse* Java Development Toolkit (JDT) and how we register *Codemap* to *Eclipse*'s event interface.

Codemap's part containing the interfaces to the *Eclipse* framework has two main interfaces as depicted in Figure 5.1.

The first interface accesses the project related information using the *IJavaProject* provided by the *Eclipse*-JDP API. We use the *IJavaProject* to access the underlying source file resources from which the vocabulary can be extracted. These resources are then injected into *Codemap*'s calculation pipeline, see Subsection 5.1.1.

The second interface is located where we integrate the view into *Eclipse*. To achieve this we make use of an extension point[1] provided by *Eclipse*. The two classes *MapView* and *MapController* are the two most important classes in this aspect. The former handles displaying the visualization for the currently selected project, the latter is responsible for registering event handlers and processing events that are generated from *Codemap* and *Eclipse*.

5.2 On Performance Improvements

This section describes the performance improvements we implemented in *Codemap* to enable a smooth user experience.

Software Cartorapher [33], the predecessor of *Codemap*, did not focus on performance as it was not embedded in an IDE and not used for live visualizations. Since *Codemap* is supposed to help the developer while he is working, it needs to calculate and update the visualization as quickly as possible. To achieve this, performance had to be improved in several places. The following subsections cover the most important performance optimizations.

Subsection 5.2.1 explains the performance improvements made during the calculation of the elevation model. Subsection 5.2.2 elaborates how the lookup for the nearest neighbor on the map was made faster. Subsection 5.2.3 explains how to create images in a fast way using SWT and finally Subsection 5.2.4 describes why we use automatically ported *FORTRAN* code for the eigenvalue decompositions.

5.2.1 Fast Elevation Model

This section describes the improvements implemented in the calculation of the elevation model.

[1] An extension point allows the contribution of functionality to a specific plugin.

As illustrated in Figure 3.3 *Codemap* builds the elevation model by calculating a normal distribution for each class and then sums up the volumes of all classes. Since our visualization is pixel-based, the elevation model must be recalculated once the size of a map changes. Approached naively this operation is quite expensive as it includes a pixel-wise rasterization of normal distributions. This means that *Codemap* would have to calculate the elevation for each pixel within a circle around the center of the hill *i.e.* the position of the class on the two-dimensional pane. The circle can be seen as an approximation of when the elevation is below a given threshold such that the influence it has on the elevation is not relevant any more.

Codemap optimizes the elevation model algorithm performance by not calculating the whole 360° of the elevation for one class but only a piece of one eighth *i.e.* 45°. This is possible since the elevation for each hill is a normal distribution which is completely symmetrical. We take one eighth since this especially eases the calculation of the full elevation for one hill. The calculated piece can simply be mirrored along the two axes and the two diagonals the two-dimensional coordinate system.

The calculation can be further optimized as follows:

The distance from each pixel in the triangle to the center of the approximated hill is given as follows:

$\delta(m,n) = \sqrt{m^2+n^2}$

We then use the normal distribution formula to calculate the value for each pixel:

$\phi(x) = \sigma e^{-\frac{1}{2}x^2}$ where σ is used for scaling and x represents the distance to the center of the distribution. Inserting the distance calculation we can rewrite the equation as follows:

$\phi(m,n) = \sigma e^{-\frac{(\sqrt{m^2+n^2})^2}{2}}$

We can then omit the square root and the square:

$\phi(m,n) = \sigma e^{-\frac{m^2+n^2}{2}}$

The distance function itself can be simplified as well. Given the triangle we can calculate the kth square number by summing up all the odd numbers:

$$\sum_{i=1}^{k} 2i-1$$

The sums are helpful, since we can generate the distance function iteratively. Our final formula is as follows:

$\phi(m,n) = \sigma e^{-\frac{(\sum_{i=1}^{m} 2i-1+\sum_{i=1}^{n} 2i-1)}{2}}$

5.2.2 Nearest Neighbor Lookup

This section describes the usage of kd-trees to gain a performance benefit when searching for the nearest hill (nearest neighbor) given pixel coordinates on the map.

When the user clicks on the map, we need to find the nearest neighbor of the point clicked. The naive approach, scanning the entire list of locations for the nearest neighbor, has the time complexity $O(N)$ where N represents the numbers of locations on the map. We need better performance for the nearest neighbor lookup, since it is used quite often *i.e.* during coloring, when processing mouse-clicks on the map and for the tooltip displaying the current location below the cursor.

A kd-tree as introduced by Friedmann and Bentley [6, 21] is a data structure that optimizes the search for nearest neighbors in a set of k dimensional vectors. A kd-tree is a binary tree that is built by splitting the space of all entries in the set into smaller subspaces. The root node represents the whole space. Each node in the tree represents a subspace that results by splitting the (sub-)space of the parent node in the tree along a hyperplane. The computation time required to build the tree is $O(N \log N)$, and the expected search time for the nearest neighbor is $O(\log N)$ [21].

This work uses an implementation of the kd-tree algorithm proposed by Moore [42] that supports insertion and deletion and chooses the splitting dimension depending on the depth in the tree. For our two-dimensional case, this means that splitting is performed by alternately choosing the x and y axis as the splitting dimension.

5.2.3 SWT Image Performance

This section describes the performance improvements we implemented to avoid some drawbacks that occur when using an SWT[2] graphics context for pixel-wise drawing.

While rendering *Codemap*'s visualization we discovered that rendering using a Graphics Context[3] is quite slow. *Codemap* draws the visualization pixel by pixel, so repeated calls to *GC.drawPoint()* were made. Our visualization has a default size of 512x512 pixels, in which case 262'144 pixels need to be rendered and for each pixel redundant boundary checks are performed.

One faster variant is not to use the Graphics Context but draw to the underlying ImageData[4] using *ImageData.setPixel()*. This method accesses the bitmap data directly, but has to check the depth of the image for each call. Thus it is faster than to use the Graphics Context, but there is still room for improvement.

[2] The Standard Widget Toolkit (SWT) is a graphical widget toolkit for the *Java* platform, provided as an alternative to *Java*'s default UI toolkits.

[3] `Class documentation for org.eclipse.swt.graphics.GC`

[4] `Class documentation for org.eclipse.swt.graphics.ImageData`

The even faster way is to create the bitmap data directly as byte array and then pass it to the *ImageData* constructor. That way we avoid as much overhead as possible when drawing pixel by pixel.

5.2.4 Translation of *FORTRAN* to JVM Bytecode

This section explains our decision to use classes originated in automatic translation from *FORTRAN* JVM Bytecode to perform algebraic calculations.

Seymour and Dongarra present a *FORTRAN*-to-*Java* translator that translates *FORTRAN* code to JVM bytecode instructions [53]. *Codemap* makes use of the automatically ported *FORTRAN* packages ARPACK and LAPACK for eigenvalue calculations in the Isomap algorithm. LAPACK is a software library to solve linear equations, eigenvalue problems and singular value decomposition. ARPACK is a software library to solve large scale eigenvalue problems based on the Arnoldi Method.

There is also a plain *Java* library that provides linear algebra functionality, JAMA[5]. We use the translated *FORTRAN* classes since they scale better, as our input matrix (which is the result of Latent Semantic Indexing) can be of quite high dimensionality.

[5] http://math.nist.gov/javanumerics/jama/

Chapter 6

Case Study

In this chapter we discuss the informal user study we performed to evaluate the approach taken by *Codemap*.

Kuhn and Loretan performed a feasibility study by implementing a prototype of software cartography called *Software Cartographer* [33]. They show the possibility to generate cartographic maps for software systems and provide some example maps analyzing different software systems. They provide an algorithm that generates software maps and they show that the algorithm has proven valuable. This work demonstrates the use of software cartography in the context of an IDE.

In this section we discuss the informal user study of *Codemap* we performed to validate our assumptions about the usage of the tool.

6.1 User Study

To get further feedback about *Codemap* we decided to perform a user study with professional developers and students. During this study we obstained considerable feedback. Most importantly we noticed that our choice to base the visualization on vocabulary had to be rethought. This section gives a detailed overview of the lessons learned during that study. Furthermore, in Section 4.3 we discuss some features that can be implemented based on the feedback from this study.

The study performed was not a controlled experiment but more an informal study to see whether the tool but also the questions asked in the study can be improved. This section gives an overview of the lessons learned during that study. Please refer to Appendix B for the tasks given during the study. For the duration of the study, the participants were asked to think aloud. That way the researcher supervising the study could gather results not only about the final result of the tasks assigned to the

participants, but also about their solution process which is actually more relevant for our results.

As a scenario for the experiment we chose "first contact with a previously unknown closed source system". We chose *Outsight*, a commercial web-application which is written in *Java*, as system for the study. The participants had limited time of 90 minutes to solve 5 exploratory tasks and to fix one bug report.

The results from the study are mixed and challenge our assumptions on how the developers would use *Codemap*. It became apparent that our initial decision to use lexical similarity as a distance metric for the cartographic layout must be rethought. Even though the participants were aware of distance metric implementation of *Codemap* they tended to interpret the visual distance as a measure of structural dependencies.

From the feedback that arose during from the study, we learned that the developers intuitively expected that the map represents their mental model of the software that they built from the system's architecture. Even though east/west and north/south directions on the map had a clear semantic interpretation the developers did not navigate along these axes.

One participant reported: *"When I see a big hill representing a class, I automatically click on it because I think it is important given its size"*. In fact, all participants tended to interpret large classes to be the most important for the system since they are the ones best visible. Another participant was biased by outliers *i.e.* classes that are peripheral on the map. These outliers had no other classes in their close proximity and where thus better visible than other classes that were part of a hill of multiple classes. On a similar note, one participant requested that the tool should mark the important classes by adding landmarks to the visualization. This is a difficult task since the definition of importance varies from developer to developer and even depends on the task one is working on. Hence automatically detecting important classes is not feasible. The confusion arising from the hill size however can be diminished by providing different metrics for the height the hills.

Another idea that originates from the study is to enable the map to focus on one hill only. The visualization should then only elevate the focused class and its collaborators. Additionally the collaboration can be visualized by displaying references and calls to the class under focus. This leads to another interesting feature, namely visualizing all collaborators of a given class.

Furthermore it was criticized that the maps are hard-linked to projects only. Indeed it is desirable if *e.g.* one is working on a project which is split into multiple sub-projects to have one common map for all these sub-projects. Another request was that the map should somehow display the vocabulary of the underlying visualization. One participant suggested to enrich the visualization with tag-clouds *e.g.* one cloud of the vocabulary for each class.

Another observation was that inexperienced developers (*i.e.* students) are more likely to find the map useful than professional developers. That was not unexpected, since for

power users *any* new way of using the IDE is likely to slow them down, and conversely for beginners *any* way of using the IDE is novel. The only exception to this observation was *Codemap*'s search bar, a one-click interface to *Eclipse*'s native file-search, that was used by all participants but one.

In general, participants reported that *Codemap* was most useful when it displayed search results, callers, implementers, and references. A participant reported: *"I found it very helpful that you get a visual clue of quantity and distribution of your search results"*. In fact, we observed that that participants almost never used the map for direct navigation but often for search and reverse engineering tasks.

Another participant concluded *"This task could have been solved without* Codemap*, but it was awesome to have visual feedback"*. In general the participants that had experience with other visualization tools appreciated that *Codemap* displays the visualization very fast and does not take minutes for data extraction or the visualization preparation.

Chapter 7

Conclusion

This thesis presents software cartography, a cartographic software visualization tool. Our approach visualizes software entities using a shared, spatial and stable layout. The layout is spatial since it is based on a map-like visualization and therefore uses the cartography metaphor. It is stable because the position of software entities on the map is based on their vocabulary, which is known to be more stable than structure [2] and rather grows than changes over time [64]. Finally the visualization is shared since the very same layout can be used by all developers and staff.

Software maps satisfy our five desiderata [31], which are a generalized and extended version of DeLine's four desiderata [16]:

1. Software maps show the entire program and are continuous.
2. Software maps contain landmarks that allow the developers to find parts of the system perceptually, rather than relying on naming or other cognitive feats.
3. Software maps remain visually stable as the system evolves (both locally and across distributed version control commits) since the layout is based on vocabulary.
4. Software maps are capable of showing global information overlays.
5. Distance in the visualization is based on vocabulary and thus has a technically meaningful interpretation.

The layout of software maps is based on the lexical similarity of software entities. Our algorithm uses Latent Semantic Indexing (LSI) [15] to position software entities in a multi-dimensional space. We present an improved composition of two algorithms to calculate the two-dimensional embedding. First Isomap [59] is applied to compute a globally optimal layout that is then used as input for the next step. The next step applies Multidimensional Scaling (MDS) [8] to embed the software entities in a two-dimensional visualization pane. Additionally we apply cartographic visualization

techniques to render an aesthetically appealing landscape.

We present *Codemap*, a plugin for the *Eclipse* IDE that displays software maps for *Eclipse* projects. *Codemap* aims to support the developer with a better mental model based on a visualization of the software he is working on. *Codemap* provides support for various development tasks ranging from code navigation over reverse engineering to collaboration support and social awareness. The goal that the developer can use his map-based mental model during these tasks while working in the IDE. In this way, software maps reflect world maps in an atlas that exploit the same consistent layout to depict various kinds of thematic information for geographical sites.

Since *Codemap* aims to support the developer during his daily work in the IDE, therefore it needs to calculate and update the visualization as quickly as possible. To guarantee a smooth user experience, performance was improved at several critical points in the program.

We performed an informal user study with professional developers and students to get more feedback about *Codemap* and to validate our assumptions. During that study, it became apparent that our initial decision to use lexical similarity as the only distance metric for the cartographic layout must be rethought. The participants tended to interpret the visual distance as a measure of structural dependencies even though they were aware of distance metric implementation of *Codemap*. This means that our initial goal, to support developers with an intuitive mental model, has not been met yet. Section 7.1 discusses possible future approaches to be taken to meet our goal and help developers at a better mental model of their software projects.

7.1 Future Work

The *Eclipse* centric future work has already been discussed in Section 4.3 and won't be repeated here. As further future work, we can identify the following promising directions:

- The user study revealed that programmers tended to misinterpret the layout as a measure of structural dependencies. Based on this observation we suggest an implementation of *anchored multidimensional scaling* such that developers can initialize the map to their mental model. Once a map has been initialized to a developer's mental model, the map is more likely to start co-evolving with and influencing the developer's mental model. Anchored MDS allows the developer to define anchors which influence the layout of the map [11]. Any software artifact can be used as an anchor, even those not present on the map as for example external libraries. With this future layout algorithm, developers may *e.g.* arrange the database layer in the south and all UI layer in the north using the respective libraries as anchors.
- Software maps at present are largely static. We envision 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. Instead of generating all the labels and thematic widgets up-front, users can annotate the map, adding comments and waymarks 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.
- Once the layout issues are fixed we plan to perform an empirical user study to evaluate the application of software cartography for software comprehension and reverse engineering, but also for source code navigation in development environments.

7.2 Lessons Learned

Lanza presents implementation recommendations and design guidelines that have proven very helpful during the design and implementation of *Codemap* [35]. We would like to contribute to this knowledge base by adding the most important lesson we learned during the implementation of our tool.

When building a visualization tool, one must not forget for whom the tool is built. The hypothetical user for the research prototype has a tremendous influence on some of the fundamental decisions taken during development. In our case, we decided to base the positioning of software entities on the map on their lexical distance. But during the informal user study it arose that this decision was rather confusing for all the participants. We learned painfully, that our very basic assumption, to use vocabulary to determine the software entity's position on the map, confuses the users. Our approach was similar to the waterfall model, where a software project is first implemented and then tested. First we built the visualization tool and then we performed an evaluation with potential users. In our opinion this approach does not work well as it misses too much early feedback!

Thus, based on our experience, we suggest to use an approach based on the agile methodology. We suggest to perform user studies as soon as possible during the development process of a (visualization) tool to learn more about the users and from the users. We reason that with earlier user-studies, the researcher is able to adapt the tool as soon as possible to the needs and to the understanding of its users and thus is able to build a tool that performs its purpose even better.

Appendix A

Codemap Quickstart

This Appendix provides a short introduction on how *Codemap* can be installed and on how the implemented features can be used.

A.1 Obtaining Codemap

Codemap requires at least Eclipse 3.5 and Java 6. *Codemap* provides optional functionality that depends on the Eclipse Communication Framework[1] and Eclemma [2]. These plugins are not required, but recommended to enjoy the full functionality of *Codemap*.

The Update Site for *Codemap* can be found at http://scg.unibe.ch/download/codemap. Perform the following steps to install *Codemap* from the update site:

1. From your Eclipse menu select "Help" → "Install New Software". In the Install dialog Figure A.1 enter http://scg.unibe.ch/download/codemap at the Work with field.
2. Check the latest *Codemap* version.
3. Press "Next" and follow the steps in the installation wizard.

A.2 Using *Codemap*

After installing *Codemap* and after having restarted Eclipse a welcome screen appears that explains some of the features of *Codemap*. For completeness, these explanations

[1] http://www.eclipse.org/ecf/
[2] http://www.eclemma.org/

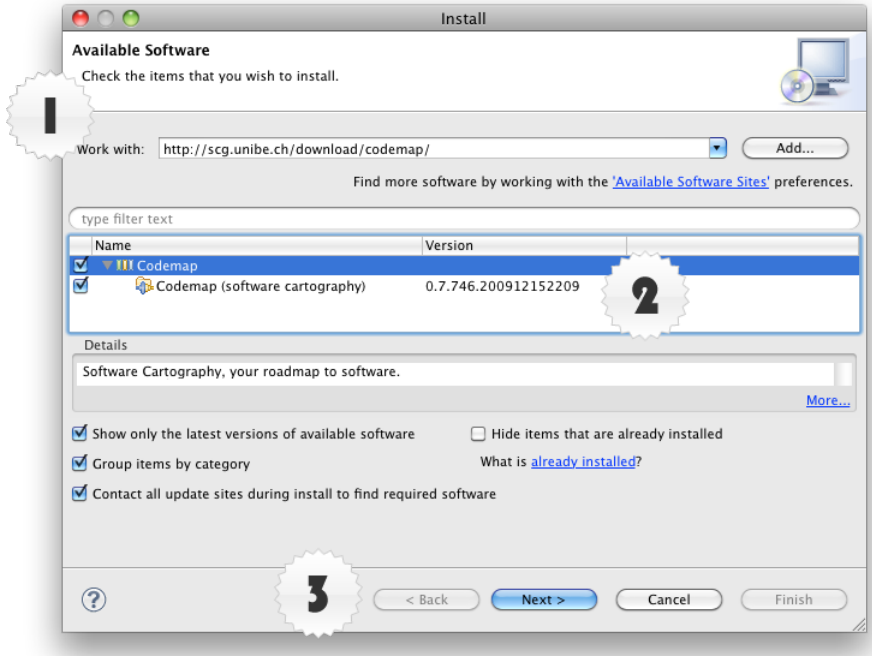

Figure A.1: Install *Codemap* from the update site at http://scg.unibe.ch/download/codemap . Select the latest *Codemap* version, press Next and follow the wizard steps.

have been added to the Quickstart as well.

The main feature of Codemap is the Codemap View Figure A.4 displaying the actual map of your code.

To activate Codemap View go to "Window" → "Show View" → "Other..." and select the view "Codemap View" in the category "Codemap" as depicted in Figure A.2.

In the following subsections, an introduction to each part of *Codemap* is given.

The Codemap View should appear in your Eclipse IDE like in Figure A.4.

A.2.1 Mapview

The Mapview (Badge 1 in Figure A.4) shows a map for the currently selected project. If you want to get the map for a different project, just select another one in the Package Explorer. Hills Represent Classes and their position on the map is determined by their vocabulary. The size of a hill reflects the number of lines in the corresponding class.

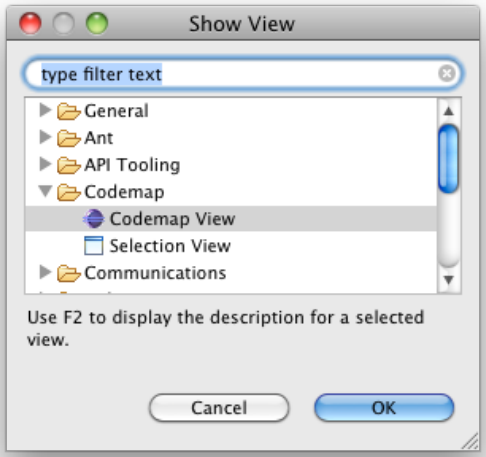

Figure A.2: To use Codmap select the view "Codemap View" in the category "Codemap".

If two hills are close enough, they melt together, and islands with complex shapes appear.

A.2.2 Toolbar

Figure A.3: Use the *Codemap* toolbar to change landscape-colors, displayed overlays and labeling.

Codemap's toolbar (Figure A.3) offers the following actions:

- Colors: Change the metrics that color the landscape. See Subsection A.2.4 for more details on the available coloring metrics.
- Layers: Show or hide different overlays. See Subsection A.2.5 for more details on the available overlay metrics.
- Labels: Change the way the locations on the map are labeled.
- Link with Current Selection: If this toggle is checked the *Codemap* view automatically reveals the Java element currently selected in other views or editors.

- Force Selection to Package Explorer: If this toggle is checked *Codemap* automatically reveals the current selection in the Package Explorer.

A.2.3 Search Bar

The Search Bar is embedded in the Codemap View and provides quick access to Eclipse's search utility. Upon searching it runs a text-search in the scope of the project associated with the current Codemap. The search results appear on the map under the condition that the Search Overlay is enabled. Please read Subsection A.2.5 for more information regarding the search overlay.

A.2.4 Coloring Metrics

The landscape itself can be colored according to different metrics. Figure A.5 shows the metrics that are currently implemented. The coloring metrics can be changed using the buttons provided in the toolbar, see Subsection A.2.2.

- Default: displays green islands in blue water.
- Color by package: Assigns a unique color to each package and colors the islands according to the package of each class.
- Heat-map coloring: Displays a heat-map in the range from yellow (hot) to black (cold). Hot means that the file was recently opened in the *Eclipse* editor.
- Coverage Coloring: Displays coverage information ranging from red (no coverage) to green (good coverage). Please refer to Section A.2.4 for more information.

Coverage Coloring

EclEmma is a Java code coverage tool for Eclipse. To enable the EclEmma overlay set the Checkbox on "Color by Coverage" in the Colors overlays Section. If you need instructions on how to use EclEmma, please refer to the EclEmma website: http://www.eclemma.org/. Once the coverage overlay is selected the current coverage is displayed. The coverage of a location on the map fades from red (no coverage) to green (covered) depending on the procentual coverage of the class represented by that location. The location is colored grey if no location-specific coverage information is available.

A.2.5 Overlay Metrics

On top of this landscape different metrics can be displayed. These metrics can be enabled/disabled on a per project level by using the buttons provided in the toolbar,

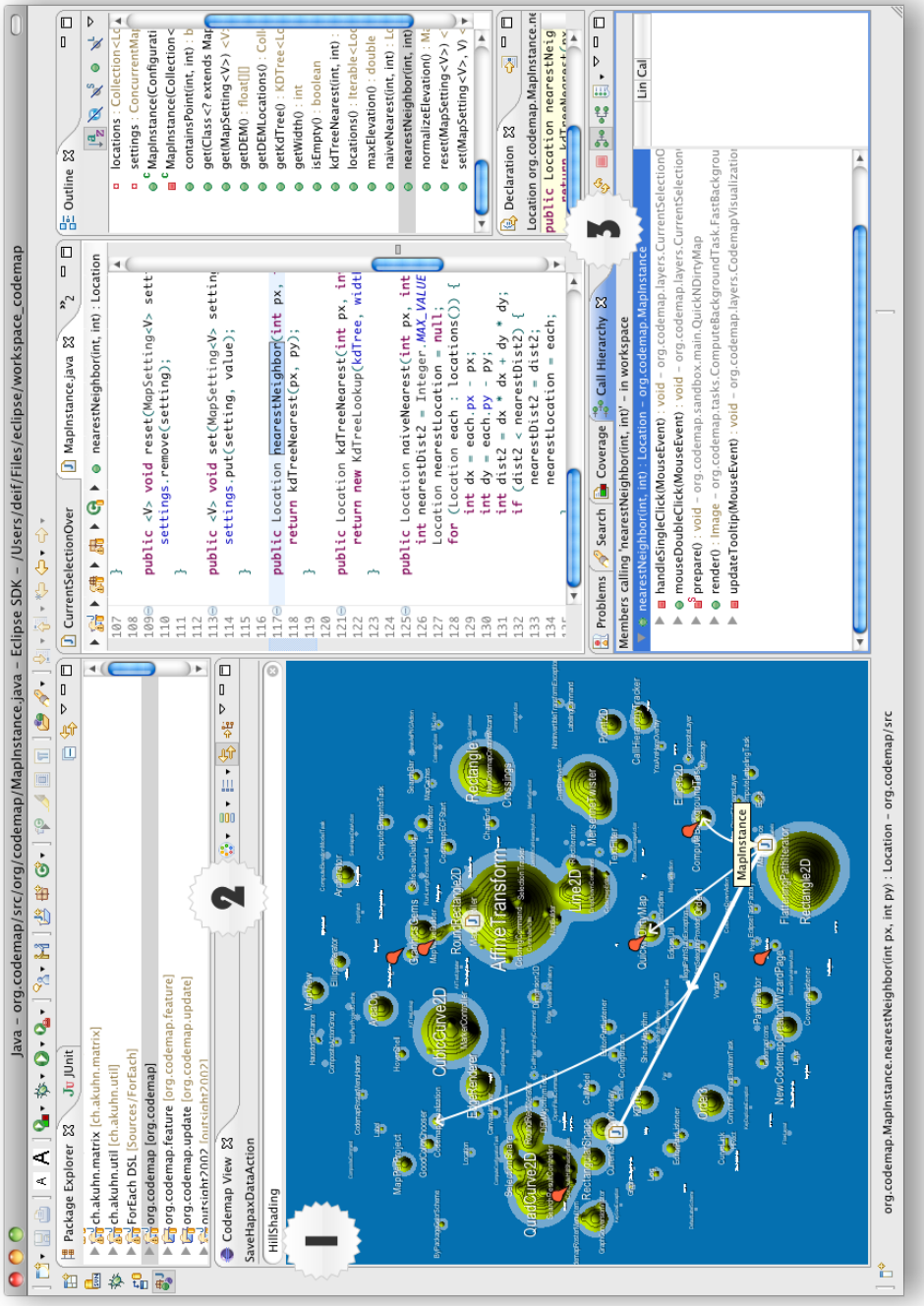

Figure A.4: A Codemap of the project org.codemap. Visible on the map are some search results, the currently opened and the currently active file. The arrows represent a call hierarchy.

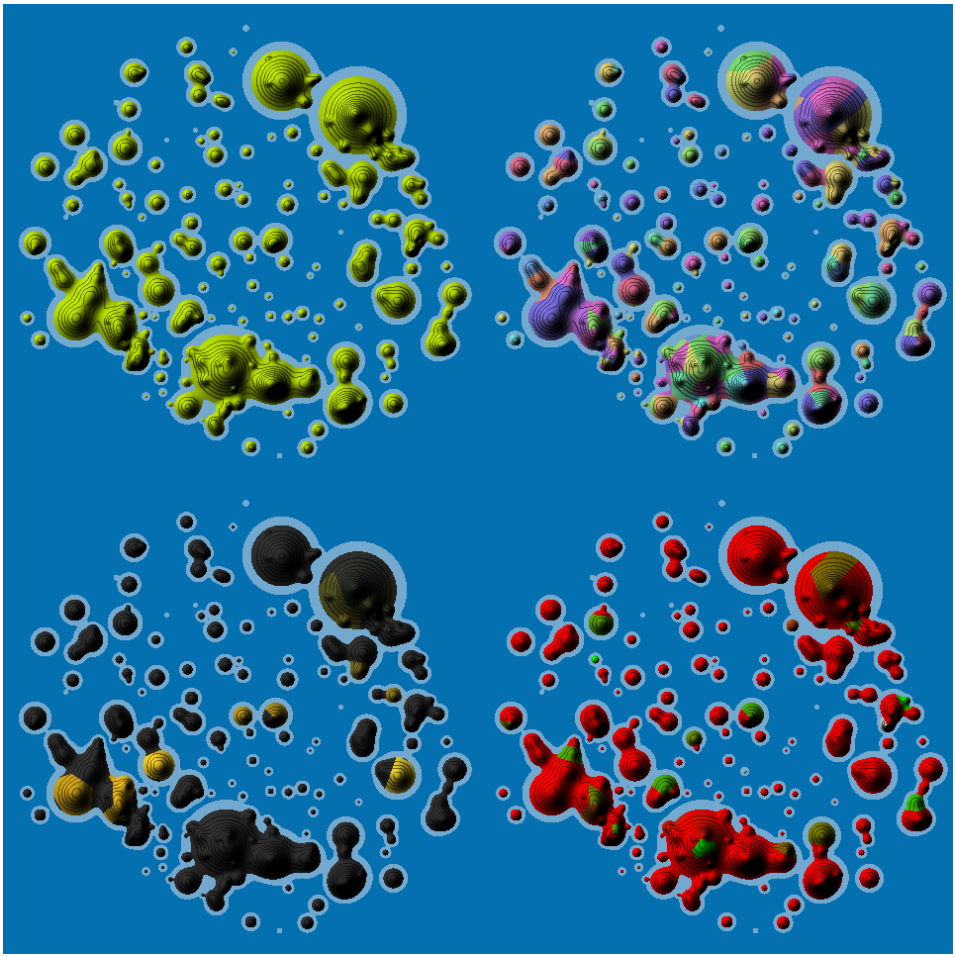

Figure A.5: The landscape generated by *Codemap* can be colored according to different metrics. In this figure one can see these metrics applied to the same project. Default coloring is displayed in the upper left. Coloring by package is displayed the upper right. On the bottom left one can see a heatmap coloring whereas the bottom right shows test-coverage.

see Subsection A.2.2. Some of the metrics are linked with other eclipse views.

- Call Hierarchy Overlay: Eclipse provides a great tool to show callees and callers of certain methods. To enable the Call Hierarchy Overlay set the Checkbox on "Link with Call Hierarchy" in the layers section.
- Markers: Eclipse provides markers to annotate problems and general information regarding your code in the source-files. To enable display of markers on the map check Makers in the layers section.
- Search Result Overlay: Eclipse provides search functionality to search your code. To enable display of search results on the map check "Search Results" in the layers section.
- Show active File: Display the file that is currently marked active in the Eclipse Editor.
- Show open Files: Display all files that are currently opened in the Eclipse Editor.
- Show Selection: Display or hide the elements that currently selected on the codemap.

Collaboration Overlay

The Eclipse Communication Framework (ECF) supports development of distributed Eclipse applications. ECF provides the library code needed to create distributed plugins easily as well as some example plugins. These examples include shared editing and instant messaging. Building on ECF, we integrate collaboration support for *Codemap*. We chose to display which files are currently edited by a peer, once the collaboration feature is enabled. Collaboration is supported on top of popular protocols compatible to the Extensible Messaging and Presence Protocol (XMPP) or similar. As depicted in Figure 4.7 once sharing is enabled, a meeple is displayed for each file opened by remote collaborators.

If you need instructions on how to use or install ECF, please refer to the official website: http://www.eclipse.org/ecf/

Once ECF is installed you need to connect to an existing instant messaging account using ECF. The ECF-based menu contribution added to *Codemap* shows up as soon as there are online buddies for the account you connected with.

To initiate sharing with your buddy, click on the *Eclipse* menubar icon (white triangle) in the *Codemap* view, select the entry "Share open files" as depicted in Figure A.6 to send a sharing request to your buddy. Once he accepts your request, his open files are displayed on your map and vice versa, as illustrated in Figure 4.7.

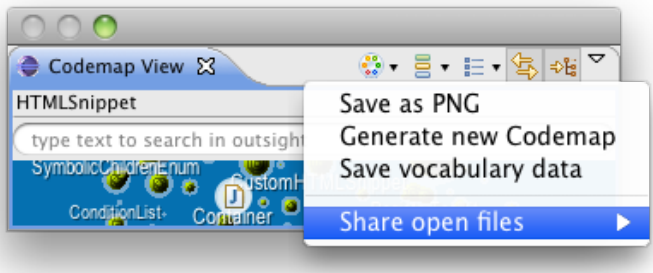

Figure A.6: *Codemap* allows to share the files that are currently open by an editor given that the Eclipse Communication Framework is installed and that the user is logged in to an IM-network.

Appendix B

User Study Questions

B.1 What is the Domain of *Outsight*?

Find out what the purpose of the given application is and identify the main collaborators. Take some time to explore the system and figure out its domain, and answer the following questions.

1. What is the domain of *Outsight*?
2. List the main collaborators of the application.
3. Draw a simple collaboration diagram of the application (example below).
4. Speculate about the main feature of program?

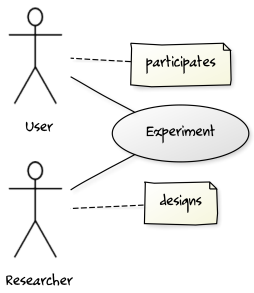

Figure B.1: Sample collaboration diagram.

B.2 Which Technologies are used in *Outsight*?

We want to get an overview of the technologies in use by *Outsight*. For this we need to identify the important technologies (like Ajax, XML, reStrucutedText, FTP,) that this application uses.

1. List the main technologies used by *Outsight*.

B.3 Which is the Architecture of Outsight?

In this task we are going to have a look at the architecture of *Outsight* given version 2004. Reverse engineer the architecture by answering the following questions.

1. Which are the main architectural components of *Outsight*?
2. How are these components related to one another? Which components should not directly call each other? You may draw an UML diagram of the components (example below).
3. Draw an UML diagram of the architecture: roughly at package level, only including key classes, if at all.
4. Which architecture paradigms are used (e.g. pipes and filters, layers, big ball of mud,)?

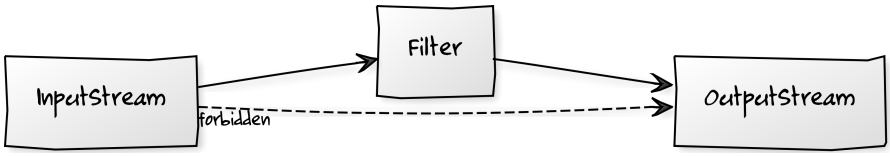

Figure B.2: Sample component diagram.

B.4 Which Classes collaborate in a Feature?

For each of the following feature descriptions list the classes that collaborate in the given feature.

1. Inactive users are reminded after some months, and eventually deleted if they don't log in after a certain number of reminders.
2. Depending on the kind of user, a user can see and edit more or less data. Which class(es) implement these permissions? Which class(es) define the permissions for each kind of user? Which class checks the permissions when accessing data?
3. Active search: the system compares the curriculum vitae of the users with stored searches of the companies and sends new matches to the company.

B.5 Asses the Code Quality

In this question we want to asses the code quality of *Outsight*. Answer the following questions regarding the quality:

1. Is the code-coverage ok? If you are not familiar with the eclemma code-coverage tool please refer to the tutorial.
2. Are there Godclasses (definition below)?
3. Are the classes organized in the right packages? should certain classes be moved to other packages? It's good enough to list one or two examples.

Godclass A godclass is an class that knows too much or does too much. A godclass contains most of a program's overall functionality as an "all-knowing" object, maintains most of the information about the entire program and provides most of the methods for manipulating its data.

B.6 Fix a Bug

Your manager just told you to deal with the following bug report:

> Issue Number 0015 (08-08-2004)
> On the overview website of the company registration: a list of cities or regions should be displayed instead of local-event names.

Try to figure out what he's telling you and gather the knowledge to fix the bug. As a reminder, you are not asked to actually perform the bug fix task, but just to acquire the necessary knowledge of the system to describe how you would perform the bug fix.

1. Describe how you would handle above bug report, e.g. how and where would you change the system, which classes are involved in your bug fix.

List of Figures

2.1 Software Cartographer, the predecessor of *Codemap* 6

3.1 Software Cartography in a nutshell 17
3.2 Software Cartography in a nutshell 19
3.3 Digital elevation model 19
3.4 Two Codemap overlays 21
3.5 The "Swiss roll" data set, a classical example for Isomap. 23
3.6 Comparison of Isomap without and with an additional application of Multidimensional Scaling to spread the layout 26
3.7 Calculation a Two-Dimensional Layout 27

4.1 A heat-map showing the last-visited locations 32
4.2 EclEmma displays coverage results as a sortable list 33
4.3 A sample coverage overlay 34
4.4 Visualizing search results 35
4.5 The *Eclipse* call hierarchy view 35
4.6 *Codemap*'s call hierarchy overlay 36
4.7 *Codemap*'s collaboration overlay 37
4.8 *Eclipse*'s type hierarchy view 39
4.9 *Eclipse*'s bookmarks view 40

5.1 A simplified overview of *Codemap* as an UML class diagram 44

A.1 The *Codemap* update site. 62
A.2 Show the *Codemap* view 63
A.3 The *Codemap* Toolbar 63
A.4 A Codemap of the project org.codemap. 65
A.5 Four different metrics of *Codemap* 66
A.6 *Codemap*'s collaboration support 68

B.1 Sample collaboration diagram. 69
B.2 Sample component diagram. 71

Listings

5.1 The AbstractValue class can trigger notifications upon changes. 45
5.2 A *TaskValue* instance adds itself as a dependent to each *Value* passed as constructor argument. The *computeValue* method is called as soon as we enter the working state. 46
5.3 Defining the dependencies between the different calculation steps 46

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] Sushil Bajracharya, Adrian Kuhn, and Yunwen Ye. Suite 2009: First international workshop on search-driven development - users, infrastructure, tools and evaluation. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 445–446, 2009.

[4] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM.

[5] 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.

[6] Jon L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, April 1980.

[7] 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.

[8] Ingwer Borg and Patriuck J. F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 2005.

[9] Lionel Briand and Dietmar Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. *Software Reliability Engineering, International Symposium on*, 0:148, 1999.

[10] 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.

[11] 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.

[12] 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.

[13] Mei-Hwa Chen, M.R. Lyu, and W.E. Wong. An empirical study of the correlation between code coverage and reliability estimation. *Software Metrics, IEEE International Symposium on*, 0:133, 1996.

[14] Jan de Leeuw. Applications of convex analysis to multidimensional scaling. In J.R. Barra, F. Brodeau, G. Romier, and B. Van Cutsem, editors, *Recent Developments in Statistics*, pages 133–146. North Holland Publishing Company, Amsterdam, 1977.

[15] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[16] Robert DeLine. Staying oriented with software terrain maps. In *DMS*, pages 309–314, 2005.

[17] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven M. Drucker, and George G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VL/HCC*, pages 11–18, 2006.

[18] Robert DeLine, Amir Khella, Mary Czerwinski, and George G. Robertson. Towards understanding programs through wear-based filtering. In *SOFTVIS*, pages 183–192, 2005.

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

[20] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, New York, NY, USA, 1992. ACM.

[21] Jerome H. Friedman, Jon L. Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[22] Yaniv Frishman and Ayellet Tal. Online dynamic graph drawing. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):727–740, 2008.

[23] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3D. In *Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization)*, September 2006.

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

[25] Einar W. Hoest and Bjarte M. OEstvold. Debugging method names. In *Proceedings of the 23nd European Conference on Object-Oriented Programming (ECOOP'09)*, LNCS, page To appear. Springer, 2009.

[26] 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.

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

[28] 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.

[29] Adrian Kuhn. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 175–178. IEEE, 2009.

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

[31] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution (JSME)*, 2010. To appear.

[32] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Towards improving the mental model of software developers through cartographic visualization, 2010. Under submission to NIER track of ICSE 2010.

[33] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, Los Alamitos CA, October 2008. IEEE Computer Society Press.

[34] 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.

[35] Michele Lanza. CodeCrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[36] 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.

[37] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[38] 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.

[39] J. W. Mcclurkin, L. M. Optican, B. J. Richmond, and T. J. Gawne. Concurrent processing and complexity of temporally encoded neuronal messages in visual perception. *Science*, 253(5020):675–677, August 1991.

[40] 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.

[41] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[42] Andrew Moore. An introductory tutorial on kd-trees. Technical Report Technical Report No. 209, Computer Laboratory, University of Cambridge, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1991.

[43] 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.

[44] Doantam Phan, Ling Xiao, R. Yeh, and P. Hanrahan. Flow map layout. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 219–224. IEEE Computer Society, 2005.

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

[46] BZ Research. 5th annual eclipse adoption study. November 2008.

[47] 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.

[48] 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.

[49] D. W. Sandberg. Smalltalk and exploratory programming. *SIGPLAN Not.*, 23(10):85–92, October 1988.

[50] Susan S. Schiffman, Lance M. Reynolds, and Forrest W. Young. *Introduction to Multidimensional Scaling: Theory, Methods, and Applications*. Academic Press, October 1981.

[51] Mauricio Seeberger. How developers drive software evolution. Master's thesis, University of Bern, January 2006.

[52] Mauricio Seeberger, Adrian Kuhn, Tudor Gîrba, and Stéphane Ducasse. Chronia: Visualizing how developers change software systems. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 345–346, March 2006. Tool demo.

[53] Keith Seymour and Jack Dongarra. Automatic translation of fortran to jvm bytecode. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 126–133, New York, NY, USA, 2001. ACM.

[54] Cameron Skinner. Code visualization, uml, and dsls. In *Professional Developers Conference (PDC09)*, November 2009. http://microsoftpdc.com/Sessions/FT08.

[55] 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.

[56] 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.

[57] 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.

[58] Alexandru Telea, Alessandro Maccari, and Claudio Riva. An open toolkit for prototyping reverse engineering visualizations. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 241–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[59] Joshua B. Tenenbaum, Vin Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.

[60] 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.

[61] Warren Torgerson. Scaling and psychometrika: Spatial and alternative representations of similarity data. *Psychometrika*, 51(1):57–63, March 1986.

[62] 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.

[63] Robert van Liere and Wim de Leeuw. Graphsplatting: Visualizing graphs as continuous fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):206–212, 2003.

[64] 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.

[65] Gina Venolia. Five attempts at spatializing code. In *New Paradigms in Using Computers (NPUC)*, July 2009. http://research.microsoft.com/projects/spatialcode/, http://research.microsoft.com/apps/pubs/default.aspx?id=81655.

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

[67] 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.

[68] Jim Whitehead. Collaboration in software engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 214–225, Washington, DC, USA, 2007. IEEE Computer Society.

[69] James A. Wise, James J. Thomas, Kelly Pennock, David Lantrip, Marc Pottier, Anne Schur, and Vern Crow. Visualizing the non-visual: spatial analysis and interaction with information from text documents. *infovis*, 00:51, 1995.

[70] Forrest W. Young. Muldidimensional scaling. In *Kotz-Johnson Encyclopedia of Statistical Sciences*, volume 5. John Wiley & Sons, Inc., 1985.

[71] Hongyu Zhang. Exploring regularity in source code: Software science and Zipf's law. *Reverse Engineering, Working Conference on*, 0:101–110, 2008.